SEGUNDO PARCIAL SSL

CLASE	CAP-VOL	TEMAS
15	CAP2-VOL2	Autómatas Finitos con Pila
16	CAP7-VOL3	Máquina De Turing
17	COMPILACIÓN CAP3-VOL2	Introducción al proceso de compilación (parte 1)
18	CAP3-VOL2-2	Introducción al proceso de compilación (parte 2)
19	CAP3-VOL2-3	Introducción al proceso de compilación (parte 3)
20	CAP4-VOL2-1	Análisis léxico, sintáctico y semántico (parte 1)
21	CAP4-VOL2-2 COMPILACIÓN	Análisis léxico, sintáctico y semántico (parte 2)
22	BISON FLEX	BISON | FLEX

Autómatas Finitos con Pila

Los autómatas finitos con pila (AFPs), también conocidos como autómatas «*push-down*» son más poderosos que los Autómatas Finitos, porque además de reconocer a los Lenguajes Regulares, tienen la capacidad de reconocer a los Lenguajes Independientes de Contexto, como son, por ejemplo, las expresiones aritméticas y las sentencias de un Lenguaje de Programación.

Tipo de lenguaje formal	Gramática que lo genera	Autómata mínimo que lo reconoce
LIR (Lenguaje IRrestricto)	GIR (Gramática IRrestricta)	Máquina de Turing
LSC (Lenguaje Sensible al Contexto)	GSC (Gramática Sensible al Contexto)	Máquina de Turing
LIC (Lenguaje Independiente del Contexto)	GIC (Gramática Independiente del Contexto)	Autómata Finito con Pila
LR (Lenguaje Regular)	GR (Gramática Regular)	Autómata Finito

Un LIC es un Lenguaje Formal que es generado por una GIC. La GIC tiene la característica que toda producción es del tipo: `noterminal → (terminal + noterminal)*`

Ejemplo:

$L1 = \{a^n b^n / n \geq 1\}$ Es un LIC

L1 puede ser generado por una GIC con estas producciones: `S → aSb | ab`

El AFP es más poderoso que el AF. Esta potencialidad extra del AFP se debe a que, además de tener estados y transiciones entre los estados *tiene una memoria en forma de PILA* (stack) que permite almacenar, retirar y consultar cierta información que será útil para reconocer a los LICs. Un AF solo tiene control sobre los caracteres que forman la cadena a analizar, mientras que un AFP controla los caracteres a analizar y la pila.

Definición Formal: Un AFP está constituido por:

1. Un flujo de entrada, infinito en una dirección, que contiene la secuencia de caracteres que debe analizar, similar a un AF.
2. Un control finito formado por estados y transiciones etiquetadas, similar a un AF.
3. Una pila abstracta, que establece la gran diferencia con los AFs. Esta pila se representa como una secuencia de símbolos o caracteres tomados de cierto alfabeto, diferente al alfabeto sobre el que se construye un LIC reconocido por un AFP.

Un AFP es una 7-upla: M = (E, A, A', T, e0, p0, F), donde:

- E es un conjunto finito de estados;
- A es el alfabeto de entrada, cuyos caracteres se utilizan para formar la cadena a analizar;
- A' es el alfabeto de la pila;
- e0 es el estado inicial;
- p0 es el símbolo inicial de la pila, el que indica que la pila no tiene símbolos;
- F es el conjunto de estados finales;

- T es la función: T: E x (A ∪ {ε}) x A' → conjuntos finitos de E x A'* (para cada estado, símbolo de entrada o palabra vacía y símbolo del tope de la pila, determina la transición a otro estado y decide que se debe meter en la pila).

Analicemos qué significa la notación de la función T, con un ejemplo: T(4, a, Z) = {(4, RPZ), (5, ε)}.
Se lee así: si el AFP se encuentra en el estado 4, en el tope de la pila tiene el símbolo Z y lee el carácter a del flujo de entrada, entonces se queda en el estado 4 y agrega RP a la pila, o se mueve al estado 5 y quita el símbolo Z que está en el tope de la pila. (AFP no determinístico).

La forma de procesar la pila es la siguiente:

- El AFP realiza un pop del símbolo que está en el tope de la pila.
- El AFP realiza un push de los símbolos indicados, siendo el primero el que quedará en el tope de la pila.

En el ejemplo: T(4, a, Z) = {(4, RPZ), (5, ε)}
El AFP hace un *pop* del símbolo Z e, inmediatamente después, un *push* de Z, luego P y, finalmente R, que es el nuevo símbolo del tope de la pila. En el otro caso, ε significa que no agrega símbolos a la pila, solo hace el *pop* del símbolo que está en el tope.

Ejercicio
Describir qué significa la transición: T(4, ε ,Z) = { (5,RP) }.
Estando en el estado 4, es una transición epsilon, es decir, no hay ningún carácter y hay una Z en el top de la pila, transita, va al estado 5 (no pone a Z) y pone a PR.

Un AFP puede reconocer un LIC de dos maneras:

1. Por estado final, como en los AFs: se transita desde el estado inicial a uno final independientemente del contenido que quede en la pila.
2. Por pila vacía: Se transita desde el estado inicial hasta que tanto la entrada como la pila están vacías.

Los autómatas por pila vacía y por estado final son equivalentes, siempre es posible obtener un autómata con pila que reconozca por pila vacía a partir de uno que reconozca por estado final y viceversa.

AUTÓMATAS FINITOS CON PILA DETERMINÍSTICO

Los Autómatas Finitos con Pila no Determinísticos, para la misma terna de partida tienen varias transiciones.
Los Autómatas Finitos con Pila Determinísticos para cada estado, símbolo de entrada o ε, y símbolo en el tope de la pila hacen única transición a otro estado.

Si incluimos el flujo de entrada en la definición, entonces un AFPD está formado por una colección de estos ocho elementos:

- Un alfabeto de entrada A. Son los símbolos con los que se forman las palabras del LIC a reconocer . Hay un símbolo especial al que llamaremos fdc (fin de cadena), que solo puede aparecer al final de la cadena en estudio para indicar su terminación.
- Un flujo de entrada en una dirección, que contiene la cadena a analizar

Símbolo especial $

- Un alfabeto A' de símbolos de la pila. Hay un símbolo especial, al que llamaremos $, que indica "pila lógicamente vacía".
- Una pila (stack) infinita en una dirección. Inicialmente la pila está vacía, lo que se indica con el símbolo $ en el tope de la pila.
- Un conjunto finito y no vacío de estados E.
- Un estado inicial (único).
- Un conjunto de estados finales o de aceptación.
- Una función de transiciones entre los estados : T: E x (A $\{\varepsilon\}$) x A' $\rightarrow$ conjuntos finitos de E x A'*.

Ejemplo:

Sea $L1 = \{a^n b^n / n \geq 1\}$, construyamos un AFPD que lo reconozca:

- El alfabeto de entrada **A** está formado por **{a, b}**.
- El alfabeto de la pila **A'** tiene los símbolos **$**, para indicar que la pila está vacía, y **R**, que usaremos para indicar que el AFPD ha leído una a de la cadena que está analizando.
- Necesitamos obtener un AFPD que cuente la cantidad de **a**es con las que comienza la cadena que se está analizando, y luego pueda "descontar" de esa cantidad por cada **b** que se lea. Si al final la pila queda vacía, el AFPD reconocerá a la cadena como palabra del lenguaje L1.
- Representamos los movimientos del AFPD que resuelve este problema de esta manera: **e0,$ ⇒ a ⇒ e1,R$** comienza con el estado inicial y la pila está vacía; si lee una a se mueve al estado **e1** y agrega una **R** a la pila;
- **e1,R => b => e2,** ε se halla en el estado **e1** y en el tope de la pila hay un símbolo **R**; si lee una **b** se mueve al estado **e2** y quita la **R** del tope de la pila. Y así sucesivamente.

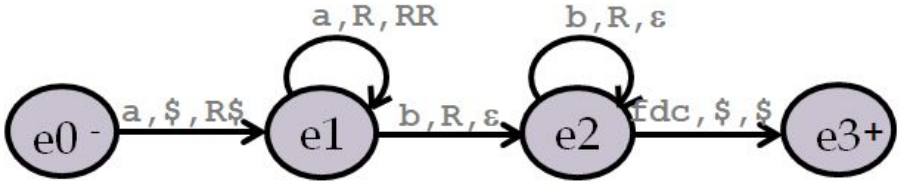


TABLA DE MOVIMIENTOS

La TM en un AFP o en un AFPD es similar a la Tabla de Transiciones en un Autómata Finito. La diferencia radica en que en la TM no podemos hablar solo de "estado", debemos hablar del par "estado y símbolo en el tope de la pila".

Ejemplo

Construyamos la TM de un AFPD que reconoce el lenguaje $L1 = \{a^n b^n / n \geq 1\}$

TM	a	b	fdc
e0,$	e1,$R		
e1,R	e1,RR	e2,R ε	

e2,R		e2,R ε	
e2,$			e3,$

El conjunto de estados de este AFPD es {e0, e1, e2, e3}, donde e0 es el estado inicial y e3 es el único estado final.

Ejercicios:

1. Definir formalmente el AFPD del ejemplo anterior:

TM	a	b	fdc
e0,$	e1,$R		
e1,R	e1,RR	e2,R ε	
e2,R		e2,R ε	
e2,$			e3,$

2. Describir los movimientos que realiza ese AFPD para reconocer la cadena aaabbb.
3. Describir el trabajo del AFPD para no reconocer la cadena aabbb.
4. Describir el trabajo del AFPD para no reconocer la cadena aaabb.
5. Describir el trabajo del AFPD para no reconocer la cadena aabba.
6. ¿Por qué no reconoce la palabra vacía?
   Para reconocerla, debe haber una transición para el fin de cadena para la palabra vacía. La definición de este lenguaje no la reconoce ya que está definido como $n \geq 1$.

Un AFPD puede reconocer un LIC de dos maneras:
Por estado final (como en los AFs): el lenguaje aceptado es el conjunto de palabras que permiten transitar desde el estado inicial a uno final, independientemente del contenido que queda en la pila.
Por pila vacía: el lenguaje aceptado es el conjunto de palabras que permiten transitar desde el estado inicial hasta una descripción instantánea en la que tanto la entrada como la pila están vacías.
Siempre que se pueda construir un AFPD que reconozca el estado final, se podrá construir un AFPD que reconozca por pila vacía.

Ejemplo:
La TM del AFPD que reconoce por pila vacía el lenguaje: $L1 = \{a^n b^n / n \geq 1\}$

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,R ε	
e2,R		e2,R ε	
e2,$			e2,$

El conjunto de estados de este AFPD es {e0, e1, e2}, donde e0 es el estado inicial y el conjunto de estados finales está vacío.

Ejercicios

7. Definir formalmente el AFPD del ejemplo anterior:

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,R ε	
e2,R		e2,R ε	
e2,$			e2,$

8. Describir los movimientos que realiza ese AFPD para reconocer la cadena aaabbb.
9. Describir el trabajo del AFPD para no reconocer la cadena aabbb.
10. Describir el trabajo del AFPD para no reconocer la cadena aaabb.
11. Describir el trabajo del AFPD para no reconocer la cadena aabba.
12. Definir formalmente un AFPD que reconozca el lenguaje $L2 = \{a^n b^{n+1} / n \geq 1\}$.
    la palabra vacía no existe en este lenguaje. siempre va a haber una “b” más que “a”

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,R	
e2,R		e2, ε	
e2,$			e3,$

13. Definir formalmente un AFPD que reconozca el lenguaje $L3 = \{a^{n+1} b^n / n \geq 1\}$.
    palabra mínima aab.

TM	a	b	fdc
e0,$	e1,$		
e1,$	e2,R$		
e2,R	e2,RR	e3, ε	
e3,R		e3, ε	e4,$

14. Definir formalmente un AFPD que reconozca el lenguaje $L4 = \{a^n b^n a^t / n \geq 1,\ t \geq 0\}$.

TM	a	b	fdc
e0,$	e1,R$		
e1,R	e1,RR	e2,R	
e2,R		e3, ε	
e3,R		e2,R	

e3,$	e3,$		e4,$

15. Sea el lenguaje L5 de todas las palabras sobre el alfabeto {a, b} en las que la cantidad de aes es igual a la cantidad de bes, como: ab, abba, bababaab, etc. Definir formalmente un AFPD que lo reconozca. Tener en cuenta que las palabras pueden empezar con a o con b.

TM	a	b	fdc
e0,$	e1,R$	e1,Z$	
e1,R	e1,RR	e1, ε	
e1,Z	e1, ε	e1,ZZ	
e1,$	e1,R$	e1,Z$	e2,$

AFPDs y las expresiones aritméticas

Las expresiones aritméticas que utilicen paréntesis para determinar un orden de evaluación, constituyen un LIC existente en la inmensa mayoría de los Lenguajes de Programación. Por lo tanto se podrá construir un AFPD que las reconozca, es decir, que determine si una secuencia de caracteres, que constituye una supuesta expresión aritmética, es sintácticamente correcta o no.

Ejemplo:

Sea el LIC de todas las expresiones aritméticas cuyo único operando es 4, los operadores son + (suma) y * (producto), y puede haber paréntesis. Algunos casos correctos son:

4

4 * (4 + 4)

4 + 4 + 4 * 4 + ((4))

Construiremos un AFPD que reconozca este LIC. Debemos tener muy en cuenta los paréntesis, siempre deben estar balanceados, para ello utilizaremos la pila. El alfabeto de la pila será {R, $}. Los estados serán {e0, e1, e2, e3} siendo e3 el único estado final La tabla de movimientos que define este AFPD será:

TM	4	+,*	(	)	fdc
e0,$	e1,$		e0,R$		
e0,R	e1,R		e0,RR		
e1,$	e1,$	e0,$			e3,$
e1,R	e1,R	e0,R		e2, ε	
e2,$		e0,$			e3,$
e2,R		e0,R		e2, ε	
e3,$					

Ejercicios

16. Definir formalmente el AFPD construido anteriormente:

TM	4	+,*	(	)	fdc
e0,$	e1,$		e0,R$		
e0,R	e1,R		e0,RR		
e1,$	e1,$	e0,$			e3,$
e1,R	e1,R	e0,R		e2, ε	
e2,$		e0,$			e3,$
e2,R		e0,R		e2, ε	
e3,$					

17. Escribir la secuencia de movimientos del AFPD para cada una de las siguientes expresiones e indicar si la acepta o no:

- **a.** ((4))
- **b.** 4 + 4 * (4 + (4))
- **c.** 4 + 4
- **d.** 4 + (((4)))

18. ¿Por qué un Autómata Finito no puede reconocer el lenguaje de las expresiones aritméticas definido anteriormente?

Máquina de Turing (MT)

La clase de autómatas que ahora se conoce como máquinas de Turing fue propuesta por Alan Turing en 1936. La máquina de Turing modela matemáticamente a una máquina que opera mecánicamente sobre una cinta. En esta cinta hay símbolos que la máquina puede leer y escribir, uno a la vez, usando un cabezal lector/escritor de cinta. Las máquinas de Turing (MT) se asemejan a los AF en que constan de un mecanismo de control y un flujo de entrada (la cinta); la diferencia es que las MT pueden mover sus cabezas de lectura hacia delante y hacia atrás y pueden leer o escribir en la cinta.
Una MT puede emplear su cinta como almacenamiento auxiliar y además de hacer operaciones de inserción y extracción, puede rastrear los datos de la cinta y modificar las celdas que desee sin alterar las demás. La MT trabaja con dos alfabetos: el alfabeto de la máquina que son los símbolos en el que están codificados los datos de entrada iniciales y el alfabeto de la cinta que son marcas especiales.

Una Máquina de Turing (MT) es un autómata determinístico con la capacidad de reconocer cualquier lenguaje formal.

Tipo de lenguaje formal	Gramática que lo genera	Autómata mínimo que lo reconoce
LIR (Lenguaje IRrestricto)	GIR (Gramática IRrestricta)	Máquina de Turing
LSC (Lenguaje Sensible al Contexto)	GSC (Gramática Sensible al Contexto)	Máquina de Turing
LIC (Lenguaje Independiente del Contexto)	GIC (Gramática Independiente del Contexto)	Autómata Finito con Pila
LR (Lenguaje Regular)	GR (Gramática Regular)	Autómata Finito

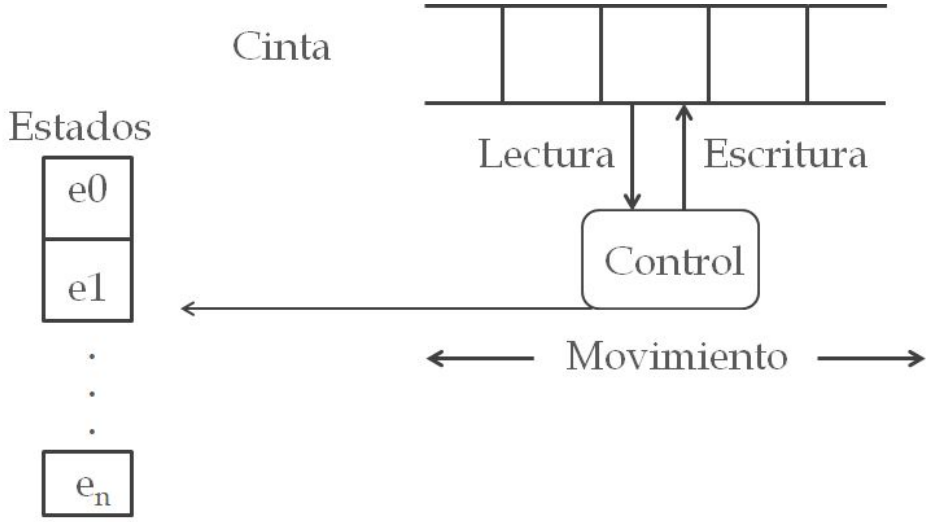
Una MT está formada por los siguientes elementos:

1. Un alfabeto A de símbolos o caracteres del lenguaje a reconocer;
2. Una cinta infinita dividida en una secuencia de celdas, cada una de las cuales contiene un carácter o un blanco. Aquí se coloca la cadena a analizar. El resto de las celdas contiene blancos;
3. Una cabeza de cinta que puede, en un solo paso leer el contenido de una celda de la cinta, reemplazarlo con otro carácter o dejar el mismo, y reposicionarse para apuntar a la celda contigua, ya sea la de la derecha (avanzando) o la de la izquierda (retrocediendo);
4. Un alfabeto A' de símbolos o caracteres que pueden ser escritos en la cinta por la cabeza de cinta. El espacio en blanco pertenece al conjunto de símbolos de la cinta, si la MT tiene que borrar una celda escribe en ella un espacio en blanco. Usaremos el símbolo para representar el espacio en blanco;
5. Un conjunto finito de estado. Uno de estos estados es el estado inicial y representa el estado en el cual la MT comienza los cálculos. Otro de los estados se conoce como estado de parada, una vez que la máquina llega a ese estado terminan todos los cálculos. El estado de parada de una MT difiere de los estados de aceptación de los AF en que éstos pueden continuar sus cálculos después de llegar a un estado de aceptación, mientras que una MT debe detenerse en el momento en que llegue a su

estado de parada. El estado inicial de una MT no puede ser a la vez el estado de parada, por lo tanto toda MT debe tener cuando menos dos estados;

6. Un programa: conjunto de reglas que nos dicen, en función del estado en que se encuentra la MT y del carácter leído por la cabeza de cinta, qué carácter escribir en la cinta en la misma posición, en qué dirección se debe mover la cabeza de cinta y a qué estado debe realizar la transición. Las acciones específicas que puede realizar una MT consisten en operaciones de escritura y de movimiento. La operación de escritura consiste en reemplazar un símbolo en la cinta con otro símbolo (puede ser el mismo símbolo que se leyó) y luego cambiar a un nuevo estado (el cual puede ser el mismo donde se encontraba antes). La operación de movimiento comprende mover la cabeza una celda a la derecha o a la izquierda y luego pasar a un nuevo estado (que puede ser igual al de partida).

El siguiente esquema representa la arquitectura de la Máquina de Turing:



Ejemplo 1

L1 = {ab}. El autómata más apropiado para reconocer palabras de este lenguaje es el Autómata Finito. Para este ejemplo usaremos la MT que tiene la capacidad de reconocer palabras de cualquier lenguaje formal.
DT de una MT que reconoce el lenguaje L1:

Los elementos que constituyen esta MT son:

1. un alfabeto A de símbolos del lenguaje a reconocer A= {a,b} ;
2. una cinta infinita con la cadena ab;
3. una cabeza de cinta;
4. un alfabeto A' de símbolos que pueden ser escritos en la cinta por la cabeza de cinta A' = {A,B} ;
5. un conjunto finito de estados;
6. un conjunto de reglas que representan las transiciones.

Nota: para este caso en particular se podría escribir en la cinta el mismo carácter que se lee, por ejemplo:

Ejemplo 2

Reglas del programa de la MT que reconoce $L2 = \{a^n b^n / n \geq 1\}$.
El autómata más apropiado para reconocer palabras de este lenguaje es el Autómata Finito con Pila. Como en el ejemplo anterior usaremos la MT que tiene la capacidad de reconocer palabras de cualquier lenguaje formal. Un alfabeto A = {a, b} y con un A'= {A,B}. Simbolizamos con R al movimiento de la cabeza de cinta hacia la derecha y con L al movimiento de la cabeza de la cinta hacia la izquierda. La cinta tendrá un blanco (□) después del último carácter de la cadena a analizar que actuará como centinela (infinitos blancos).

Programa en el que cada regla tiene la siguiente notación:
estado actual – carácter leído, carácter escrito, dirección – estado de llegada

```
e0 – a,A,R – e1 /* e0 estado inicial */
e0 – B,B,R – e3 /* encuentra una b marcada */
e1 – a,a,R – e1 /* lee todas las aes */
e1 – B,B,R – e1 /* encuentra la primera b marcada */
e1 – b,B,L – e2 /* marca una b */
e2 – a,a,L – e2 /* lee las aes y retrocede buscando una A */
e2 – B,B,L – e2 /* retrocede por las bes marcadas */
e2 – A,A,R – e0 /* encuentra una A y avanza */
e3 – B,B,R – e3 /* verifica que todas las bes están marcadas */
e3 - □,□,L – e4+ /* reconoce la cadena */
```

Ejemplo 3

Sea $L3 = \{a^n b^n c^n / n \geq 1\}$
No es un LR: no puede ser reconocido por un AFD.
No es un LIC: no puede ser reconocido por AFP.
Si puede ser reconocido por una MT. Construiremos una MT con un alfabeto A = {a, b, c} y con un A'= {X, Y, Z} Simbolizamos con R al movimiento de la cabeza de cinta hacia la derecha y con L al movimiento de la cabeza de la cinta hacia la izquierda. La cinta tendrá un blanco (□) después del último carácter de la cadena a analizar que actuará como centinela (infinitos blancos).

Se completa la MT con un programa en el que cada regla tiene la siguiente notación:
estado actual – carácter leído, carácter escrito, dirección – estado de llegada

```
e0 – a,X,R – e1 /* e0 es el estado inicial */
e0 – Y,Y,R – e6 /* ya marcó todas las aes */
e1 – a,a,R – e1 /*lee todas las aes que hay en la cinta*/
e1 – b,Y,R – e2 /* marca la primera b */
e1 – Y,Y,R – e4 /* lee una b marcada y cambia de estado*/
e2 – b,b,R – e2 /*lee todas las bes que hay en la cinta*/
e2 – c,Z,L – e3 /* marca una c */
```

```
e2 – Z,Z,R – e5 /* lee una c marcada y cambia de estado*/
e3 – Z,Z,L – e3 /* lee las ces marcadas y retrocede */
e3 – b,b,L – e3 /* lee las bes y retrocede */
e3 – Y,Y,L – e3 /* lee las bes marcadas y retrocede */
e3 – a,a,L – e3 /*lee las aes y retrocede (buscando X)*/
e3 – X,X,R – e0 /* encuentra una X y avanza */
e4 – Y,Y,R – e4 /* lee bes marcadas y avanza */
e4 – b,Y,R – e2 /* lee una B y la marca */
e5 – Z,Z,R – e5 /* lee ces marcadas y avanza */
e5 – c,Z,L – e3 /* lee una c y la marca */
e6 – Y,Y,R – e6 /* encuentra todas las bes marcadas */
e6 – Z,Z,R – e7 /* encuentra la 1° c marcada */
e7 – Z,Z,R – e7 /* verifica que todas las ces están marcadas*/
e7 – □,□,L – e8+ /* reconoce la cadena y transita al estado final */
```

La MT no solo reconoce cualquier lenguaje formal de la Jerarquía de Chomsky, sino que también puede realizar tareas computacionales.

Ejemplo 4

Dado un número binario, construimos una MT que obtiene su complemento. Sean A = A' = {0,1}. La MT puede tener el siguiente programa:

```
e0 – 0,1,R – e0 /* e0 es el estado inicial */
e0 – 1,0,R – e0
e0 – □,□,L – e1 /* transita al estado final */
```

Ejercicio 1

Se dispone de una cinta en la que hay un número m de 1s seguido de un número n >= m de aes. Construir una MT que cambie las primeras m aes por bes.

Ejercicio 2

Diseñar una Máquina de Turing que calcule la paridad de un número binario, que agregue un 0 si la cantidad de 1s es par o agregue un 1 si la cantidad de 1s es impar.

Ejercicio 3

Construir una MT que duplique una cadena de aes y bes en su cinta. Ejemplo: si la MT comienza con abbaa□, luego de procesar su programa debe terminar con abbaa□abbaa□.

Compilación

Compilación vs. intérprete

Un *compilador* analiza el programa fuente y lo *traduce* a otro equivalente escrito en otro lenguaje (lenguaje objeto). Un *intérprete* analiza el programa fuente y lo *ejecuta* directamente, sin generar ningún código equivalente.

Modelo de compilación en C

Preprocesador

El preprocesador acepta como entrada código fuente y se encarga de:

1. Eliminar los comentarios.
2. Interpretar y procesar las directivas de preprocesamiento, precedidas por el símbolo #. Por ejemplo las directivas #define, #include.

Una de las directivas más comúnmente empleadas en C es #define Define una constante (identificador) simbólico. El preprocesador sustituye las apariciones del identificador por el valor especificado. Por ejemplo, #define MAX_SIZE 100 establece el valor de la constante simbólica MAX_SIZE a 100. En el programa se utilizará la constante simbólica y el preprocesador sustituye cada aparación de MAX_SIZE por el literal 100 (de tipo int).

Compilador

El compilador (del inglés, compiler) analiza la sintaxis y la semántica del código fuente preprocesado y lo traduce, generando un archivo que contiene el código objeto. La extensión por defecto de estos archivos es .o (GNU). El programa proporcionado por GNU para realizar la compilación es gcc (las siglas gcc son el acrónimo de GNU C Compiler).

Linker

El enlazador (del inglés, linker) resuelve las referencias a objetos externos que se encuentran en un archivo fuente. Estas referencias son a objetos que se encuentran en

otros módulos compilados, ya sea en forma de archivos objeto o incorporados en alguna biblioteca.

Introducción al proceso de compilación

Nos ocuparemos del estudio del proceso que lleva a cabo un compilador y su relación con la sintaxis y la semántica de los Lenguajes de Programación. El programa a traducir siempre se encuentra en un flujo de entrada que está implementado a través de un archivo de texto, como ocurre en ANSI C con los archivos de extensión .c, o sea: El programa que se debe compilar es una secuencia de caracteres que termina con un centinela.

Conceptos básicos

Un compilador es un complejo programa que lee un programa escrito en un lenguaje fuente y lo traduce a un programa equivalente en un lenguaje objeto, normalmente más cercano al lenguaje de máquina. Al programa original se lo llama programa fuente y al programa obtenido se lo denomina programa objeto.

Proceso de compilación

El proceso de compilación está formado por dos partes:

1. El **ANÁLISIS** (**Front End**), que a partir del programa fuente, crea una representación intermedia del mismo.
2. La **SÍNTESIS** (**Back End**), que a partir de esta representación intermedia, construye el programa objeto.

Análisis del programa fuente

En la compilación, el ANÁLISIS está formado por tres fases:

- el Análisis Léxico;
- el Análisis Sintáctico;
- el Análisis Semántico.

Análisis léxico

El Análisis Léxico detecta diferentes elementos básicos que constituyen un programa fuente:

- identificadores;
- palabras reservadas;
- constantes;
- operadores;
- caracteres de puntuación.

Por lo tanto el Análisis Léxico sólo se ocupa de los Lenguajes Regulares que forman parte del Lenguaje de Programación.
Esta fase del análisis tiene como función detectar palabras (LEXEMAS) de estos Lenguajes Regulares (CATEGORÍAS LÉXICAS o TOKENS).
La entrada para el Análisis Léxico son caracteres y la salida son tokens. Los espacios que separan los lexemas son ignorados durante el Análisis Léxico.
Ejercicios:
Sea, en ANSI C, el siguiente fragmento de programa, 1) y la siguiente función, 2):

1)
```
. . .
int WHILE;
while (WHILE > (32))
      -2.46;
. . .
```
2)
```
void XX(void)
{
      int a;
      double a;
      if(a > a)
            return 12;
}
```

Diseñe una tabla con dos columnas: lexema y token. Complete la tabla con todos los lexemas hallados en el Análisis Léxico y las categorías a las que pertenecen.

Análisis sintáctico

El análisis Sintáctico trabaja con los tokens detectados durante el Análisis Léxico, es decir la entrada para el Análisis Sintáctico son esos tokens. Esta etapa de análisis sí conoce la sintaxis del Lenguaje de Programación y, en consecuencia, tendrá la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas. Sin embargo no podrá determinar si el programa, en su totalidad, es sintácticamente correcto.

Ejemplo

Durante el Análisis Sintáctico, la GIC que describe la sintaxis del lenguaje, por su propia independencia del contexto, no permite detectar si una variable fue declarada antes de ser utilizada, si la misma variable fue declarada varias veces y con diferentes tipos, y muchas otras situaciones erróneas.

Análisis léxico y sintáctico

Debemos diferenciar los términos "*Análisis Léxico*" vs. "*Analizador Léxico*" y "Análisis Sintáctico" vs. "Analizador Sintáctico".
Para evitar confusiones llamaremos: **Scanner** al "Analizador Léxico" y **Parser** al "Analizador Sintáctico".

Análisis sintáctico

Dada una definición sintáctica formal, como la que brinda una GIC, el Parser recibe tokens del Scanner y los agrupa en unidades, tal como está especificado por las producciones de la GIC utilizada. Mientras se realiza el Análisis Sintáctico, el Parser verifica la corrección de la sintaxis y, si encuentra un error sintáctico, el Parser emite el diagnóstico correspondiente. En la medida que las estructuras semánticas son reconocidas, el Parser llama a las correspondientes rutinas semánticas que realizarán el Análisis Semántico.

Análisis semántico

El Análisis Semántico realiza diferentes tareas, completando lo que hizo el Análisis Sintáctico. Una de estas importantes tareas es la “verificación de tipos”, para que cada operador trabaje sobre los operandos permitidos según la especificación del Lenguaje de Programación.

Ejercicio:

Informar cuáles son los errores detectados durante el Análisis Semántico: void XX(void) { int a; double a; if(a > a) return 12; }

Las rutinas semánticas llevan a cabo dos funciones:

1. Chequean la *semántica estática* de cada construcción, es decir, verifican que la construcción analizada sea legal y que tenga un significado. Verifican que las variables involucradas estén definidas, que los tipos sean correctos, etc.
2. Si la construcción es semánticamente correcta, las rutinas semánticas también hacen la traducción, es decir, generan el código para una Máquina Virtual que , a través de sus instrucciones, implementa correctamente la construcción analizada.

Ejemplo 1

Sea una producción: $E \rightarrow E + T$

Cuando esta producción es reconocida por el Parser, éste llama a las rutinas semánticas asociadas para chequear la semántica estática (compatibilidad de tipos) y luego, realizar las operaciones de traducción, generando las instrucciones para la Máquina Virtual que permitan realizar la suma.

Una definición completa de un LP debe incluir las especificaciones de su Sintaxis y de su Semántica. La Sintaxis se divide, normalmente, en componentes Independientes del Contexto y componentes Sensibles al Contexto. Las GICs sirven para especificar la Sintaxis Independientes del Contexto. Sin embargo, no todo el programa puede ser descripto por una GIC; por caso, la compatibilidad de tipos y las reglas de alcance de las variables son sensibles al contexto.

Ejemplo 2

En ANSI C, la sentencia $a = b + c$; es ilegal si cualquiera de las variables no estuviera declarada. Esta es una restricción Sensible al Contexto.

Resumiendo: debido a estas limitaciones de las GICs, las restricciones Sensibles al Contexto son manejadas como situaciones semánticas. En consecuencia, el componente semántico de un LP se divide, habitualmente, en dos clases:

- Semántica Estática (la que nos interesa en este momento);
- Semántica en Tiempo de Ejecución.

La Semántica Estática de un LP define las restricciones Sensibles al Contexto que deben cumplirse para que el programa sea considerado correcto. Algunas reglas típicas de Semántica Estática son:

1. Todos los identificadores deben estar declarados;
2. Los operadores y los operandos deben tener tipos compatibles;
3. Las rutinas (funciones) deben ser llamadas con el número apropiado de argumentos.

Ninguna de estas restricciones pueden ser expresadas mediante una GIC

Introducción al proceso de compilación (parte 2)

Un compilador simple

Se describirá un proceso formado por cuatro etapas:

1. La definición de un Lenguaje de Programación muy simple.
2. La escritura de un programa fuente en este lenguaje.
3. El diseño de un compilador para realizar la traducción.
4. La obtención de un programa objeto equivalente.

Lenguaje de programación MICRO

Charles Fischer, en su libro “Crafting a Compiler with C” (1991), presenta un LP que denomina Micro. Es un lenguaje muy simple que está diseñado para tener un LP sobre el que se pueda analizar la construcción de un compilador básico.
Fischer lo define informalmente de esta manera:

- El único tipo de dato es entero.
- Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.
- Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos.
- Las constantes son secuencias de dígitos (números enteros).
- Hay dos tipos de sentencias:

  Asignación

  `ID := Expresion;`

  `Expresion` es infija y se construye con identificadores, constantes y los operadores + y -; los paréntesis están permitidos

  Entrada/Salida

  ```
  leer(Lista de IDs);
  escribir(lista de Expresiones);
  ```
- Cada sentencia termina con un punto y coma (;). El cuerpo del programa está delimitado por inicio y fin.
- inicio, fin, leer y escribir son palabras reservadas y deben escribirse en minúscula.

Preguntas

1. ¿Qué significa “Expresión es infija”?
   Es cuando el operador está en el medio. Operando operador operando.
2. ¿Qué es ID?
   Un identificador. Cualquier id definido con el formato que debe tener.
3. ¿Qué son fin, Fin, finn?
   fin: palabra reservada; Fin y finn: serán identificadores, ya que cumplen con las reglas de los id.
4. ¿Qué significa “Todos los identificadores son declarados implícitamente”?
   No necesito declararlos previamente, los declaró al usarlos. Es un lenguaje que no necesita declaración de variables.

Agregamos una definición más precisa de Micro. Describimos las producciones de una Gramática Léxica que define los posibles lexemas que se pueden encontrar en un programa Micro y las producciones de una Gramática Sintáctica que permite precisar las construcciones válidas en Micro.

Gramática Léxica

```
<token> → uno de <identificador> <constante> <palabraReservada>
<operadorAditivo> <asignacion> <caracterPuntuacion>
<identificador> → <letra> {<letra o dígito>}
<constante> → <digito> {<digito>}
<letra o dígito> → uno de <letra> <digito>
<letra> → una de a-z A-Z
<dígito> → uno de 0-9
<palabraReservada> → una de inicio fin leer escribir
<operadorAditivo> → uno de + -
<asignacion> → :=
<caracterPuntuacion> → uno de ( ) , ;
```

El siguiente es un programa fuente en Micro:

```
inicio
      leer (a, b);
      cc := a + (b – 2);
      escribir(cc, a + 4);
fin
```

Ejercicio:

Diseñar una tabla con las columnas Lexema y token. Realizar el Análisis Léxico y completar la tabla.

Gramática Sintáctica

```
<programa> → inicio <listaSentencias> fin
<listaSentencias> → <sentencia> {<sentencia>}
<sentencia> → <identificador> := <expresion>;|
                leer(<listaIdentificadores>);|
                escribir(<listaExpresiones>);
<listaIdentificadores> → <identificador> {, <identificador>}
<listaExpresiones> → <expresion> {, <expresion>}
<expresion> → <primaria> {<operadorAditivo> <primaria>}
<primaria> → <identificador> | <constante> | ( <expresion> )
```

Ejercicio:

Según las gramáticas descriptas, escribir un programa en Micro.

Al construir el compilador, cada uno de los tokens tendrá su propio nombre:

EN EL PROGRAMA FUENTE	NOMBRE DEL TOKEN

inicio	INICIO
fin	FIN
leer	LEER
escribir	ESCRIBIR
:=	ASIGNACIÓN
(	PARENIZQUIERDO
)	PARENDERECHO
,	COMA
;	PUNTOYCOMA
+	SUMA
-	RESTA

La estructura de un compilador y el lenguaje MICRO

Análisis Léxico

El análisis léxico es realizado por un módulo llamado **Scanner**. Este analizador lee, uno a uno, los caracteres que forman un programa fuente. Produce y retorna la representación del correspondiente token, uno por vez, en la medida que es invocada por el **Parser**.

Existen dos formas principales de implementar un Scanner:

1. A través de la utilización de un programa auxiliar tipo lex, en el que los datos son tokens representados mediante Expresiones Regulares.
2. Mediante la construcción de una rutina basada en el diseño de un apropiado AFD, este método es el que utilizaremos en Micro.

Análisis Sintáctico

El análisis sintáctico es realizado por un módulo llamado **Parser**. Este analizador procesa los tokens que le entrega el **Scanner** hasta que reconoce una construcción sintáctica que requiere un procesamiento semántico. Entonces, invoca directamente a la rutina semántica que corresponde.

Existen dos formas fundamentales de Análisis Sintáctico:

1. El Análisis Sintáctico Descendente (top-down), que permite ser construido por un programador.
2. El Análisis Sintáctico Ascendente (bottom-up), que requiere el auxilio de un programa especializado tipo yacc.

Para Micro, construiremos un Parser basado en el Análisis Sintáctico Descendente.

Análisis Semántico

El análisis semántico, es la tercera fase de análisis que existe en un compilador. No existe un módulo independiente llamado Analizador Semántico. El análisis semántico se va realizando, en la medida que el **Parser** lo requiere, a través de las rutinas **semánticas**.
Las rutinas semánticas realizan el análisis semántico y también producen una salida en un lenguaje de bajo nivel para una Máquina Virtual, que forma parte de la etapa de Síntesis del compilador.
El análisis semántico:

- está inmerso dentro del Análisis Sintáctico;
- es el comienzo de la etapa de Síntesis del compilador.

Tabla de símbolos

La tabla de símbolos (TS) es una estructura de datos compleja que es utilizada para que contenga:

- las palabras reservadas del LP;
- los literales-constante;
- las constantes numéricas que existen en el programa (en forma de secuencia de caracteres).

Cada elemento de la TS está formada por una cadena y sus atributos.
En el caso de Micro, la TS contendrá las palabras reservadas y los identificadores. Cada cadena tendrá, como único atributo, un código que indique si la cadena representa una "palabra reservada" o "un identificador". En general, la TS es muy utilizada durante la compilación y, específicamente, en la etapa de Análisis por las rutinas semánticas.

Un analizador léxico para MICRO

Los lexemas que constituyen un programa fuente en este LP Micro son:

- identificadores;
- constantes;
- las cuatro palabras reservadas;
- paréntesis izquierdo y derecho;
- el punto y coma (para terminar cada sentencia);
- la coma (para formar una lista);
- el símbolo de asignación (:=);
- el operador suma;
- (+) el operador resta(-).

Cada lexema está formado por la mayor secuencia posible de caracteres para el token correspondiente, y esto requiere, a veces, que se lea el primer carácter que no pertenece a ese lexema, es decir: el carácter que actúa como centinela.

Ejemplo:

En la expresión abc+4, el identificador abc es detectado completamente por el Scanner cuando lee el carácter +, que es el primer carácter no válido (un centinela) para el token identificador.
Este carácter extra que se lee debe ser retornado porque será el primer carácter (o único) del siguiente lexema.

Si en cualquier momento de la compilación, el primer carácter del flujo de entrada que lee el Scanner para detectar el próximo lexema no es un carácter válido para ningún lexema en el LP Micro, entonces se detectó un error léxico. En este caso, se despliega un mensaje de error y se repara el error para continuar con el proceso. La reparación más sencilla consiste en ignorar este carácter espurio y reiniciar el Análisis Léxico a partir del carácter siguiente.

Ejemplo:

_Abc := 3;

El supuesto identificador comienza con un carácter espúreo para Micro (guión bajo). Esto produce un error léxico. ¿Por qué?

Si el Scanner de Micro debe procesar el texto abc*ab ¿Qué produce? ¿Por qué?

Otra tarea del Scanner es ignorar los espacios, ya sean blancos, tabulados o marcas de fin de línea, que actúen como separadores de lexemas dentro del texto del programa fuente.

El Scanner para Micro estará basado en la construcción de un AFD que reconoce los LRs que forman los tokens de Micro.

Ejercicios:

1. ¿Qué sucede si el programa fuente comienza con muchos espacios?
2. ¿Qué sucede si, en ANSI C, un literal cadena tiene espacios como “H O \n L\tA ” ?
3. ¿Qué sucede si el programa ANSI C tiene un carácter que es un espacio como ‘ ‘ o ‘\n’ ?

En cuanto al reconocimiento de las palabras reservadas, el primer problema es que son identificadores. Existen varias soluciones, veamos dos de ellas:

1. Que cada palabra sea reconocida, por el AFD diseñado, mediante un estado final propio para cada una de ellas, independiente del estado final para el reconocimiento de los identificadores generales.
2. Que las palabras reservadas sean colocadas, previamente, en las primeras entradas de la Tabla de Símbolos y con el atributo “reservada”; es decir, la TS ya contiene las palabras reservadas antes de comenzar el Análisis Léxico. Ésta es la que usaremos.

En esta solución, el AFD estará diseñado con un único estado final para reconocer a todos los identificadores, incluyendo las palabras reservadas. Cuando un identificador es reconocido por el Scanner, lo busca en la TS. Si lo encuentra y tiene el atributo “reservada”, sabe que es una palabra reservada y cuál es. En caso contrario, y si el identificador detectado todavía no se encuentra almacenado en la TS, lo agrega con el atributo “id”.

El Parser debe saber cuándo el Scanner le pasó el último token detectado en el flujo de entrada. Para ello se crea un token que llamaremos fdt (fin del texto). En consecuencia, cuando el Scanner llegue al final del flujo que contiene al programa fuente, retornará el token fdt.

Un analizador léxico (scanner) para MICRO

El Scanner será implementado como una función sin parámetros que retorna valores de tipo TOKEN. Definimos los siguientes tipos de tokens en ANSI C:

```
typedef enum {
```

```
        INICIO, FIN, LEER,ESCRIBIR, ID, CONSTANTE,PARENIZQ, PARENDERECHO,
        PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA,FDT
} TOKEN;
```

Suponemos que el prototipo de la función que implementa al Scanner es:

```
TOKEN Scanner(void);
```

El compilador Micro es muy simple. No obstante, la función Scanner no puede retornar solo un valor por enumeración y pretender que el compilador puede seguir desarrollando el trabajo necesario con su Parser y con las rutinas de análisis semántico.

Si la función **Scanner** retorna ID, los otros analizadores necesitan saber a qué identificador se refiere. Lo mismo ocurre cuando **Scanner** retorna CONSTANTE: los otros analizadores necesitan conocer su valor.
Por ello, supondremos que existen variables globales que contendrán esta información y que podrán ser accedidas por los otros analizadores.

AFD para implementar el scanner

Dado que el Scanner debe reconocer palabras de diferentes Lenguajes Regulares, es lógico suponer que el diseño de un AFD adecuado sea una buena solución. Construiremos un AFD con un único estado final para todos los identificadores, incluyendo las palabras reservadas, de esta manera el AFD será más compacto. Además, este AFD deberá realizar ciertas acciones en varios estados, como veremos a continuación.
Consideraciones para el AFD:

1. Todas las letras son equivalentes en cuanto a su funcionalidad, por lo que llamaremos L a la columna correspondiente.
2. Para los dígitos llamaremos D a la columna correspondiente.
3. Los caracteres + - ( ) ; := , tienen cada uno una función particular, por lo que habrá una columna para cada uno.
4. El centinela de “fin de texto” (fdt) tiene una función especial y muy importante ya que le indica al Parser que se terminaron los tokens. Deberá tener su propia columna en la TT del AFD.
5. El AFD ignorará los espacios. Habrá una columna (sp) para los tres caracteres especiales (, \t, \n).
6. Habrá una columna otro para cualquier otro carácter no mencionado anteriormente

La TT de un AFD que resuelve este problema es la siguiente:

TT	L	D	+	-	(	)	,	;	:	=	fdt	sp	otro
0 -	1	3	5	6	7	8	9	10	11	14	13	0	14
1	1	1	2	2	2	2	2	2	2	2	2	2	2
2+	14	14	14	14	14	14	14	14	14	14	14	14	14
3	4	3	4	4	4	4	4	4	4	4	4	4	4
4+	14	14	14	14	14	14	14	14	14	14	14	14	14
5+	14	14	14	14	14	14	14	14	14	14	14	14	14
6+	14	14	14	14	14	14	14	14	14	14	14	14	14
7+	14	14	14	14	14	14	14	14	14	14	14	14	14

8+	14	14	14	14	14	14	14	14	14	14	14	14	14
9+	14	14	14	14	14	14	14	14	14	14	14	14	14
10+	14	14	14	14	14	14	14	14	14	14	14	14	14
11	14	14	14	14	14	14	14	14	14	12	14	14	14
12+	14	14	14	14	14	14	14	14	14	14	14	14	14
13+	14	14	14	14	14	14	14	14	14	14	14	14	14
14	14	14	14	14	14	14	14	14	14	14	14	14	14

Ciertos estados provocarán que el Scanner lleve a cabo determinadas acciones:

1. En los estado 1 y 3 el Scanner debe almacenar el carácter leído en el buffer, un vector de caracteres externo a la función en el que se guardarán los caracteres que forman los presuntos identificadores (incluidas las palabras reservadas) y los dígitos que forman las presuntas constantes.
2. En el estado 2, el Scanner debe realizar varias acciones:
    a. ungetc;
    b. Determinar si es una palabra reservada o no;
    c. Si es un identificador, debe verificar que su longitud no supere los 32 caracteres;
    d. Si es un identificador correcto, almacenarlo en la TS.
3. En el estado 4, el Scanner debe analizar dos acciones:
    a. ungetc;
    b. Almacenar la secuencia de dígitos en la TS.
4. En el estado 13, el Scanner debe retornar el código que indique la detección del fdt para que el Parser conozca que se han terminado los lexemas del programa fuente.
5. En el estado 14, el AFD ha detectado un **Error Léxico**, y entonces debe llevar a cabo las acciones que correspondan.
6. En el estado 7, debe retornar el código que corresponde al carácter (

Conclusiones

El Parser invoca al Scanner cada vez que necesita un token. El Scanner analiza el flujo de entrada a partir de la posición que corresponde, detecta el próximo lexema en el Programa Fuente y retorna el token correspondiente, o tal vez encuentra un error léxico y le informa al Parser de esta situación anómala. En la construcción del Scanner es muy importante el AFD diseñado para reconocer los lexemas.
Además existen almacenamientos y funciones auxiliares que completan la implementación del Scanner en ANSI C, estos son:

- La función **ungetc** de ANSI C.
- El vector de caracteres externo que llamamos **buffer**, utilizado para almacenar los caracteres de los identificadores y los dígitos de las constantes a medida que son reconocidos por el AFD.
- La función **`void AgregarCaracter(int);`** que añade un carácter al buffer.
- La función **`TOKEN EsReservada(void);`** que dado un identificador reconocido por el AFD y almacenado en el buffer, retorna el token que le corresponde (ID o el código de alguna de las cuatro palabras reservadas).

- La función **`void LimpiarBuffer(void);`** que vacía el buffer para un próximo uso.
- La función **feof** de ANSI C para detectar el centinela del texto, el fdt, y saber que todos los caracteres del programa fuente fueron procesados.
- La función **fgetc** de ANSI C, para leer cada carácter del flujo de entrada.
- Las funciones ANSI C:
  - **`isspace`**
  - **`isalpha`**
  - **`isalnum`**
  - **`isdigit`**

Introducción al proceso de compilación (parte 3)

Un Parser para MICRO (analizador sintáctico)

El Parser trabaja con tokens y es guiado por la GIC que describe la sintaxis del LP. Estos tokens le son provistos uno a uno por el Scanner, cada vez que el Parser lo invoque.
Recordemos la Gramática Sintáctica que hemos definido anteriormente:

```
<programa>    → inicio <listaSentencias> fin
<listaSentencias>   → <sentencia> {<sentencia>}
<sentencia>   → <identificador> := <expresion>;|
                    leer(<listaIdentificadores>);|
                    escribir(<listaExpresiones>);
<listaIdentificadores> → <identificador> {, <identificador>}
<listaExpresiones>     → <expresion> {, <expresion>}
<expresion>   → <primaria> {<operadorAditivo> <primaria>}
<primaria>    → <identificador> | <constante> | ( <expresion> )
```

Extenderemos y modificaremos esta GIC para que refleje cómo será utilizada por el Parser:

1. Agregaremos una producción global que defina la totalidad del programa que es analizado, recordemos que el programa fuente es un texto que termina con un centinela.
2. Utilizaremos los nombres que le dimos a los tokens en la declaración que presentamos anteriormente:

```
typedef enum {
      INICIO,    FIN,    LEER,ESCRIBIR,    ID,    CONSTANTE,PARENIZQ,
      PARENDERECHO, PUNTOYCOMA, COMA, ASIGNACION, SUMA, RESTA,FDT
} TOKEN;
```

La GIC actualizada que utilizará el Parser será:

```
<objetivo> → <programa> FDT
<programa> → INICIO <listaSentencias> FIN
<listaSentencias> → <sentencia> {<sentencia>}
<sentencia> → ID ASIGNACION <expresion> PUNTOYCOMA |
LEER PARENIZQ  <listaIdentificadores> PARENDERECHO PUNTOYCOMA|
ESCRIBIR PARENIZQ <listaExpresiones> PARENDERECHO PUNTOYCOMA
<listaIdentificadores> → ID {COMA ID}
<listaExpresiones> → <expresion> {COMA <expresion>}
<expresion> → <primaria> {<operadorAditivo> <primaria>}
<primaria> → ID | CONSTANTE |
   PARENIZQ <expresion> PARENDERECHO
<operadorAditivo> → uno de SUMA RESTA
```

En general, no es común que se realice esta transformación. En esta oportunidad la hacemos para afianzar este concepto: *los terminales de la GIC utilizada en el Análisis Sintáctico son los tokens que retorna el Scanner y no los caracteres o las secuencias de caracteres, según corresponda, que se encuentran en el Programa Fuente.*

Árbol de Análisis Sintáctico (AAS) y Análisis Sintáctico Descendente Recursivo (ASDR)

Utilizaremos una Técnica de Análisis Sintáctico muy conocida, llamada ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO (ASDR) Utiliza rutinas que pueden ser recursivas, cuya ejecución va «construyendo» un Árbol de Análisis Sintáctico (AAS) para la secuencia de entrada (formada por tokens) que debe reconocer. Estos árboles no necesariamente son estructuras físicas sino que son implícitas.

Árbol de Análisis Sintáctico (AAS)

Un Árbol de Análisis Sintáctico (AAS) parte del axioma de una GIC y representa la derivación de una construcción (declaración, sentencia, expresión, bloque y hasta el programa completo).

Un AAS tiene las siguientes propiedades:

1. La raíz del AAS está etiquetada con el axioma de la GIC.
2. Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
3. Cada nodo interior está etiquetado con un noterminal.
4. Si A es un noterminal y X1, X2, …, Xn son sus hijos de izquierda a derecha, entonces existe la producción A-> X1,…, Xn donde cada Xi puede ser un noterminal o un terminal.

Ejercicio

Sea el programa Micro:

```
inicio
        a := 23;
        escribir(a, b+2);
fin
```

Construya el árbol de análisis sintáctico para este programa.

Análisis Sintáctico Descendente Recursivo (ASDR)

El ASDR es una de las técnicas más simples en la construcción de un compilador, pero no se puede utilizar con cualquier GIC. La idea básica del ASDR es que cada noterminal de la Gramática Sintáctica tiene asociado una rutina de Análisis Sintáctico que puede reconocer cualquier secuencia de tokens generada por ese noterminal A esta rutina la llamaremos PAS: Procedimiento de Análisis Sintáctico. En ANSI C, los PAS se construyen como funciones `void`.
Cada PAS implementa un noterminal de la Gramática Sintáctica. La estructura de cada PAS sigue fielmente el desarrollo del lado derecho de la producción que implementa, es decir: dentro de un PAS, tanto los noterminales como los terminales del lado derecho de la producción deben ser procesados en el orden en que aparecen.

Esto se realiza de la siguiente forma:

1. Si se debe procesar un noterminal , invocamos al PAS correspondiente, que, por convención, lo llamaremos con el mismo nombre, A. Esta función puede ser recursiva.
2. Para procesar un terminal t, invocamos a la función llamada Match con argumento t.

Match(t) invoca al Scanner para obtener el próximo token del flujo de tokens de entrada. Si el token obtenido por el Scanner es t, es decir, coincide con el argumento con el que se invoca Match, todo es correcto porque hubo concordancia y el token es guardado en una variable global llamada *tokenActual*. En cambio si el token obtenido por el Scanner no coincide con el argumento t, entonces se ha producido un Error Sintáctico; se debe emitir un mensaje y tener en cuenta esta situación porque el proceso de compilación ya no puede ser correcto. Además se debe realizar algún tipo de «reparación» del error para poder continuar con el Análisis Sintáctico.

Desarrollaremos algunos PAS para ver cómo trabajan.
El Parser comienza su actividad cuando se invoca al PAS que corresponde al noterminal, cuya producción hemos agregado en la GIC actualizada:

```
<objetivo> -> <programa> FDT
  void Objetivo (void) {
     Programa( );
     Match (FDT);
  }
void Objetivo (void) {
        /* <objetivo> -> <programa> FDT */
   Programa( );
   Match (FDT);
}
```

Esto significa: para determinar que el Análisis Sintáctico de un programa en Micro ha finalizado, debemos hallar la correspondencia con la secuencia de tokens generada por y seguida de FDT, el token centinela que agregamos para indicar la terminación del programa fuente.

Siguiendo el orden de las producciones de la GIC, para el noterminal tenemos un PAS que se puede construir fácilmente a partir de su producción:

```
<programa> -> INICIO <listaSentencias> FIN
void Programa(void)
{
  Match(INICIO);
  ListaSentencias();
  Match(FIN);
}
```

Para el noterminal `<listaSentencias>` el PAS es un poco
más complicado porque hay una secuencia opcional de sentencias.
Es decir `<listaSentencias>` es una o más sentencias como se ve en la definición de este noterminal

```
<listaSentencias> -> <sentencia> {<sentencia>}
```

Además existen tres tipos de sentencias:

```
<sentencia> -> ID ASIGNACION <expresion> PUNTOYCOMA |
LEER PARENIZQ <listaIdentificadores> PARENDERECHO PUNTOYCOMA|
ESCRIBIR PARENIZQ <listaExpresiones> PARENDERECHO PUNTOYCOMA
```

Cada uno de estos tres tipos de sentencias comienzan con un token diferente (ID, LEER, ESCRIBIR). Por lo tanto, conociendo el primer token ya sabemos cuál es el tipo de sentencia que corresponde.

Utilizaremos la función `ProximoToken` que retorna el próximo token a ser correspondido. Como se ve en las producciones de , el próximo token debería ser `ID, LEER o ESCRIBIR`. Entonces si el próximo token retornado por `ProximoToken` es uno de los tres mencionados, el Parser tratará de reconocer una sentencia de la secuencia opcional definida en el lado derecho de `<listaSentencias>`. Caso contrario, la lista completa de sentencias fue procesada.

Desarrollo de PAS para el noterminal <listaSentencias>

```
void ListaSentencias (void) {
/* <listaSentencias> -> <sentencia> {<sentencia>} */
  Sentencia( ); //la primera de la lista de sentencias
  while(1) { //ciclo indefinido
       switch(ProximoToken( )) {
  case ID:
  case LEER:
  case ESCRIBIR: //token correcto
    Sentencia( ); //procesa Sentencia
  default:
    return;
       } //fin switch
  } //fin while
}
```

Al definir el PAS correspondiente al noterminal sentencia nos encontramos con el problema que hay varias producciones que tienen al noterminal sentencia en su lado izquierdo.

```
<sentencia> -> ID ASIGNACION <expresion> PUNTOYCOMA |
LEER PARENIZQ <listaIdentificadores> PARENDERECHO PUNTOYCOMA|
ESCRIBIR PARENIZQ <listaExpresiones> PARENDERECHO PUNTOYCOMA
```

En consecuencia, debemos decidir cuál de las producciones deberá ser procesada. En el caso de Micro, esta situación se simplifica porque cada producción para el noternimal **`sentencia`** comienza con un terminal diferente y, por lo tanto, con solo conocer cuál es el próximo terminal ya sabemos que producción aplicar. El PAS para el noterminal sentencia puede ser construido de esta manera:

```
void Sentencia(void) {
 TOKEN tok = ProximoToken();
 switch(tok){
  case ID: //<sentencia> -> ID := <expresion> PUNTOYCOMA
   Match(ID); Match(ASIGNACION);
   Expresion(); Match(PUNTOYCOMA); break;
  case LEER:
  //<sentencia>->LEER(<listaIdentificadores>)PUNTOYCOMA
   Match(LEER); Match(PAREIZQ);
   ListaIdentificadores(); Match(PARENDERECHO);
   Match(PUNTOYCOMA); break;
  case ESCRIBIR:
 //<sentencia>->ESCRIBIR(<listaExpresiones>)PUNTOYCOMA
   Match(ESCRIBIR); Match(PARENIZQ);
   ListaExpresiones(); Match(PARENDERECHO);
```

```
   Match(PUNTOYCOMA); break;
  default:
   ErrorSintactico(tok); break;
 }
}
```

Otro PAS, con una dificultad diferente a los anteriores, aunque siempre reflejando la estructura de la correspondiente producción de la GIC:

```
void Expresion (void) {
//<expresion> -> <primaria> {<operadorAditivo> <primaria>}
 TOKEN t;
 Primaria();
 for(t=ProximoToken();t==SUMA||t==RESTA;t=ProximoToken())
 {
   OperadorAditivo();
   Primaria();
 }
}
```

```
Otro PAS en el que aparece un llamado a ErrorSintactico():
void OperadorAditivo (void) {
//<operadorAditivo> -> uno de SUMA RESTA
 TOKEN t = ProximoToken();
 if(t == SUMA || t == RESTA)
 {
   Match(t);
 }
 else
   ErrorSintactico();
}
```

Hemos comenzado con la construcción del Parser. Hasta el momento se trata de un conjunto de procedimientos que denominamos PAS que implementan las definiciones dadas por las producciones de la Gramática Sintáctica de Micro. Por supuesto, se ha visto que también hay un grupos de funciones auxiliares que contribuyen en la construcción del Parser. Estas funciones son:

- Match
- ProximoToken
- ErrorSintactico

Ejercicio

Sea el programa Micro:

```
inicio
      escribir(4);
fin
```

Escribir la secuencia de llamadas a los PAS que realizar el Análisis Sintáctico de este programa.

Etapa de Traducción, Rutinas Semánticas y Aplicación de PAS

La etapa de traducción de un programa en Micro, que pertenece a lo que se denomina Síntesis dentro de la compilación, consiste en la generación de código para una máquina virtual (MV).
Supongamos que esta MV tiene instrucciones con el siguiente formato: OP A,B,C:

- OP es el código de operación ;
- A y B designan a los operandos, que pueden ser nombres de variables o constantes enteras;
- C especifica la dirección donde se almacena el resultado de la operación.

Para algunos códigos de operación, A o B o C pueden no ser usadas.

Ejemplo

Una instrucción de esta máquina virtual:
`Declara A, Entera`
Declara la variable A como entera, no utiliza el 3er. operando.

Gran parte de la traducción es realizada por **rutinas semánticas** llamadas por el Parser. A la Gramática Sintáctica conocida se le pueden agregar **símbolos de acción** para especificar cuándo se debe realizar un procesamiento semántico.
Los símbolos de acción, indicados como #nombre, se pueden ubicar en cualquier lugar del lado derecho de una producción. A cada símbolo de acción le corresponde una rutina semántica implementada como una función.

Ejemplo

A un símbolo de acción #sumar se corresponderá la rutina semántica Sumar.

Cuando se crea un PAS, las invocaciones a las rutinas semánticas son insertadas en las posiciones designadas por los símbolos de acción. Los símbolos de acción no forman parte de la sintaxis especificada por una GIC, sino que le agregan “comentarios” a esta GIC para indicar cuándo se necesitan ejecutar las acciones semánticas correspondientes.
Ejemplo:
La producción para `<operadorAditivo>` con el agregado del símbolo de acción que le corresponde, sería:
`<operadorAditivo> → SUMA #procesar_op | RESTA #procesar_op`

Información Semántica

Al diseñar las rutinas semánticas, es muy importante la especificación de los datos sobre los cuales operan y la información que producen. El enfoque utilizado en este caso es asociar un registro semántico a cada símbolo gramatical, tanto noterminal (`<expresión>`) como terminal (`ID`). Como caso especial, habrá ciertos símbolos con registros semánticos nulos , sin datos (`PUNTOYCOMA`).

Consideremos la siguiente producción, ampliada con el agregado del correspondiente símbolo de acción:
`<expresion> -> <primaria> + <primaria> #sumar`

Por cada aparición del noterminal `<primaria>` en el lado derecho de esta producción, se generará un registro semántico. Cada registro semántico contendrá datos sobre cada operando, como, por ejemplo, dónde está almacenado y cuál es su valor.
Cuando la función Sumar es invocada como resultado de la aparición del símbolo de acción `#sumar`, se le deben pasar estos registros semánticos como argumentos. Como resultado, Sumar producirá un nuevo registro semántico correspondiente al noterminal `<expresion>` con la información necesaria.

Ejemplo

En el caso de Micro, tenemos solo estos dos registros semánticos:

1. REG_OPERACION, que solo contendrá el valor del token SUMA o RESTA
2. REG_EXPRESION, que contendrá el tipo de expresión y el valor que le corresponde, este valor puede ser una cadena (para el caso de un identificador) o un número entero (para el caso de una constante). La cadena será almacenada en un vector de caracteres previamente definido y tendrá una longitud máxima de 32 caracteres, límite de los identificadores.

Gramática Sintáctica de Micro con Símbolos de Acción

Agregamos una nueva producción:
`<identifcador> -> ID #procesar_id`
Esta producción es muy útil porque ID aparece en varios contextos diferentes de la GIC, y necesitamos llamar a la función `Procesar_ID` inmediatamente después que el Parser haya encontrado el correspondiente ID, para acceder a los caracteres que se encuentran en el buffer y así construir el registro semántico apropiado.

Además se utilizarán algunas funciones auxiliares en la construcción del compilador:

1. Generar: función que recibe cuatro argumentos que son cadenas, que corresponden al código de operación y a los tres operandos de cada instrucción de la MV; esta función producirá la correspondiente instrucción en el flujo de salida.
2. Extraer: una función tal que dado un registro semántico, retorna la cadena que contiene. Esta cadena puede ser un identificador, un código de operación, representar una constante antes de ser convertida a número entero, etc.

Dado que la Tabla de Símbolos de Micro es muy simple porque el LP es muy sencillo, las rutinas que se necesitan para operar con ella son solo dos:

1. Buscar: una función tal que dada una cadena que representa a un identificador, determina si ya se encuentra en la Tabla de Símbolos.
2. Colocar: almacena una cadena en la Tabla de Símbolos.

La rutina auxiliar utilizada por varias rutinas semánticas es Chequear, su implementación en ANSI C:

```
void chequear(char *s)
{
  if(!Buscar(s)) //la cadena está en la TS? NO:
  {
        Colocar(s);
             //almacenarla, es el nombre de un variable
```

```
    Generar(“Declara”,s,”Entera”,””);
            //genera la instrucción
  }
}
```

Funciones auxiliares en ANSI C que son necesarias para definir las rutinas semánticas que corresponden a los símbolos de acción de Micro:

```
void Comenzar(void); //inicializaciones semánticas

void Terminar(void){
 /*genera las instrucciones para terminar la ejecución del programa*/
 Generar(“Detiene”,””,””,””);
}

void Asignar(REG_EXPRESION izquierda, REG_EXPRESION derecha){
 //genera la instrucción para la asignación
 Generar(“Almacena”,Extraer(derecha),izquierda.nombre,””);
}
```

La Gramática Sintáctica para Micro, ampliada con los símbolos de acción correspondientes será:

```
<objetivo> -> <programa> FDT #terminar
<programa> -> #comenzar inicio <listaSentencias> fin
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identificador> := <expresion>; #asignar;|
  leer(<listaIdentificadores>);|
  escribir(<listaExpresiones>);
<listaIdentificadores> -> <identificador> #leer_id
     {, <identificador> #leer_id}
<listaExpresiones> -> <expresion> #escribir_exp
      {, <expresion> #escribir_exp}
<expresion> -> <primaria>
  {<operadorAditivo> <primaria> #gen_infijo}
<primaria> -> <identificador> | CONSTANTE #procesar_cte|
    ( <expresion> )
<operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op
<identificador> -> ID #procesar_id
```

Algunas rutinas semánticas

Veamos algunas de las rutinas semántica utilizadas en el compilador de Micro:

```
void Leer(REG_EXPRESION in) {
  //genera la expresión para leer
  Generar(“Read”,in.nombre,”Entera”,””);
}
REG_EXPRESION ProcesarCte (void) {
      /*convierte la cadena que representa número a
       número entero y constituye un registro semántico*/
      REG_EXPRESION t;
      t.clase = CONSTANTE;
```

```
      sscanf(buffer,"%d",&t.valor);
      return t;
    }
```

Procedimiento de Análisis Sintáctico (PAS) Con Semántica Incorporada

Recordemos uno de los PAS que hemos descripto anteriormente:

```
//<expresion> -> <primaria> {<operadorAditivo> <primaria>}
 TOKEN t;
 Primaria();
 for(t=ProximoToken();t==SUMA||t==RESTA;t=ProximoToken())
 {
   OperadorAditivo();
   Primaria();
 }
}
```

Si ahora le agregamos el procedimiento semántico, este es el definitivo.

```
void Expresion (REG_EXPRESION *resultado) {
//<expresion>-><primaria>{<operadorAditivo><primaria>#gen_infijo}
 REG_EXPRESION operandoIzq, operandoDer;
 REG_OPERACION op;
 token t;
 Primaria(&operandoIzq);
 for(t=ProximoToken();t==SUMA||t==RESTA;t=ProximoToken())
 {
   OperadorAditivo(&op);
   Primaria(&operandoDer);
   operandoIzq = GenInfijo(operandoIzq,op,operandoDer);
 }
 *resultado = operandoIzq;
}
```

Ejemplo de Análisis Sintáctico Descendente y Traducción

Consideremos el proceso de compilación del siguiente programa en Micro:

```
inicio
      A := BB - 34 + A;
fin
```

Recordemos que a todo programa fuente, el compilador le agrega el token fdt (fin de texto) para detectar el fin de flujo de entrada a compilar.

En la siguiente tabla se describen 35 pasos con las acciones tomadas con el Parser en la medida que procesa el programa:

Paso	Acción del Parser	Programa sin Procesar	Instruc. Generada para MV
1	Llamar Objetivo()	inicio A:=BB-34+A; fin FDT	
2	Llamar Programa()	inicio A:=BB-34+A; fin FDT	

3	Acción Semántica Comenzar()	inicio A:=BB-34+A; fin FDT	
4	Match(INICIO)	inicio A:=BB-34+A; fin FDT	
5	Llamar ListaSentencias()	A:=BB-34+A; fin FDT	
6	Llamar Sentencia()	A:=BB-34+A; fin FDT	
7	Llamar Identificador()	A:=BB-34+A; fin FDT	
8	Match(ID)	A:=BB-34+A; fin FDT	
9	Acción Semántica ProcesarId()	:=BB-34+A; fin FDT	Declara A, Entera
10	Match(ASIGACION)	:=BB-34+A; fin FDT	
11	Llamar Expresion()	BB-34+A; fin FDT	
12	Llamar Primaria()	BB-34+A; fin FDT	
13	Llamar Identificador()	BB-34+A; fin FDT	
14	Match(ID)	BB-34+A; fin FDT	
15	Acción Semántica ProcesarId()	-34+A; fin FDT	Declara BB, Entera
16	Llamar OperadorAditiva()	-34+A; fin FDT	
17	Match(RESTA)	-34+A; fin FDT	
18	Acción Semántica ProcesarOp()	34+A; fin FDT	
19	Llamar Primaria()	34+A; fin FDT	
20	Match(CONSTANTE)	34+A; fin FDT	
21	Acción Semántica ProcesarCte()	+A; fin FDT	
22	Acción Semántica GenInfijo()	+A; fin FDT	Declara Temp&1,Entera Resta BB, 34, Temp&1
23	Llamar OperadorAditivo()	+A; fin FDT	
24	Match(SUMA)	+A; fin FDT	
25	Acción Semántica ProcesarOp()	A; fin FDT	
26	Llamar Primaria()	A; fin FDT	
27	Llamar identificador()	A; fin FDT	
28	Match(ID)	A; fin FDT	

29	Acción Semántica ProcesarId()	; fin FDT	
30	Acción Semántica GenInfijo()	; fin FDT	Declara Temp&2,Entera Suma Temp&1, A, Temp&2
31	Acción Semántica Asignar()	; fin FDT	Almacena Temp&2, A
32	Match(PUNTOYCOMA)	; fin FDT	
33	Match(FIN)	fin FDT	
34	Match(FDT)	FDT	
35	Acción Semántica Terminar()		Detiene

Concluye correctamente la compilación y para el programa Micro:

```
inicio
      A := BB - 34 + A;
fin
```

Se generaron las siguientes instrucciones para la MV:

```
Declara A, Entera
Declara BB, Entera
Declara Temp&1, Entera
Resta BB, 34, Temp&1
Declara Temp&2, Entera
Suma Temp&1, A, Temp&2
Almacena Temp&2, A
Detiene
```

Análisis léxico, sintáctico y semántico (parte 1)

Anteriormente vimos una introducción al diseño de un compilador para un LP muy simple. Ahora nos ocuparemos de los elementos que forman parte de la etapa de compilación conocida como análisis:

- Análisis Léxico;
- Análisis Sintáctico;
- Análisis Semántico.

También veremos como la TABLA DE SÍMBOLOS interactúa con todos ellos.

Introducción a la tabla de símbolos

La Tabla de Símbolos, también conocida como diccionario, es un conjunto de estructuras de datos que se utiliza, como mínimo, para contener todos los identificadores (nombres de variables o de funciones) del programa fuente que se está compilando, junto con los atributos que posee cada identificador. Interactúa con las tres fases del análisis.

Para los identificadores sus atributos son:

- Para variables: tipo y ámbito (parte del programa donde tiene validez).
- Para funciones: cantidad de parámetros y tipo de cada uno, métodos de transferencia, tipo del valor que retorna.

Estos atributos se codifican mediante números enteros o valores por enumeración.

Muchos diseñadores de compiladores también usan la tabla de Símbolos para almacenar en ella todas las palabras reservadas del Lenguaje de Programación, cada una de ellas tendrá el atributo que indica que es una palabra reservada.

Ejemplo:

```
int XX (double a){
      char s[12];
      double b;
      b = a + 1.2;
      if(b > 4.61)
            return 1;
      else
            return 0;
}
```

Esta función posee diferentes identificadores: palabras reservadas nombre de función nombres de variables.

El identificador `XX` debe poseer atributos que indiquen:

- que es una función,
- que retorna un valor de tipo `int`,
- que tiene un parámetro, y que es de tipo `double`.

Podríamos representarlo así: `(XX, funcion, int, 1, double)`

Identificadores representados en la TS:

```
(XX, funcion, int, 1, double)
(a, parametro, double)
(s, arreglo, 1, char, 12)
(b, variable, double)
```

Análisis léxico

El Análisis Léxico, realizado por el Scanner, es el proceso que consiste en recorrer el flujo de caracteres que forman el Programa Fuente, detectar los lexemas que componen este programa, y traducir la secuencia de esos lexemas en una secuencia de tokens cuya representación es más útil para el resto del compilador.
Flujo de Caracteres → Secuencia de Lexemas → Secuencia de Tokens

Ejercicios 1

1. Defina dos tokens que puedan estar formados por más de 50 lexemas cada uno.
2. ¿Verdadero o falso? Justificar.
    a. Un lexema es una secuencia de uno o más caracteres.
    Verdadero.
    b. El espacio entre dos lexemas es un lexema.
    Falso, los espacios se ignoran.
    c. Un token puede ser representado mediante un número entero.
    Verdadero.

Debemos tener en cuenta: el Análisis Léxico es un proceso que se realiza sobre palabras (los lexemas) de Lenguajes Regulares (los tokens). Cuando estos LRs son infinitos, sus lexemas son obtenidos recién cuando se detecta el centinela. Si los LRs son finitos, pueden requerir o no un centinela para obtener el token correspondiente.

Ejemplo

```
int abcd (void){
      int a, b = 4;
      a = b + 18;
      return a;
}
```

1. El nombre de la función, abcd, es un identificador. Pertenece a un LR infinito, requiere centinela para saber cuándo termina, en este caso es el carácter espacio.
2. El carácter {, pertenece a un LR (token) finito, no requiere centinela, en ANSI C, no hay lexemas que comience con { y que luego puedan tener más caracteres.
3. El operador + pertenece a un LR finito, el lenguaje de los operadores. Este LR contiene tres operadores que comienzan con + (+, ++, +=). Cuando el Scanner encuentra el carácter + no puede afirmar que encontró un lexema, necesita conocer la existencia de un centinela para confirmarlo.

Ejercicios 2

1. Los siguientes Lexemas en ANSI C requieren centinela para ser detectados?
    a. el identificador abc
    b. el carácter
    c. el operador >
    d. la constante 2.14
    e. la palabra reservada for
    f. el literal cadena “abc”
    g. el operador !

   h. el operador ++
2. Supongamos que existe un LP que tiene tres operadores /, //, //+. Explicar cómo hace el Scanner para detectar cada uno de estos lexemas.

Cada vez que el Scanner requiere conocer un centinela para detectar un lexema, luego debe retornarlo al flujo de entrada porque ese carácter será el primero a tratar en la búsqueda del próximo lexema Una de las primeras decisiones que debe tomar el diseñador de un compilador es la elección del conjunto de tokens, es decir, el conjunto de lenguajes Regulares diferentes que será reconocido durante el Análisis Léxico. Esto es importante para facilitar el Análisis Sintáctico.
Como resultado de la actividad del Scanner, cada token se representa mediante un número entero o un valor de un tipo enumerado, y así será representado por el resto del compilador. Pero no es suficiente con obtener el token al cual pertenece cierto lexema. Las fases siguientes del compilador también requieren conocer el lexema.

Ejemplo

Sean los operadores de comparación >, >=, < y <=. Supongamos que el diseñador del compilador decide que estos cuatro lexemas pertenecen al token “Operadores de Comparación”. Entonces, además de conocerse el token, el lexema debe ser utilizado para distinguir cada operador de comparación entre los cuatro que componen este token y esto es fundamental para generar el programa objeto. Si bien la distinción entre estos cuatro operadores de comparación no afecta la actividad del Análisis Sintáctico, el Scanner deberá retornar el par ordenado (operadorComparacion, cuálEs) para que en el Análisis Semántico y en la etapa de Síntesis se conozca la información exacta, y así se puedan generar las instrucciones correspondientes.

Los lexemas son almacenados en la Tabla de Símbolos. En un LP simple que solo tenga variables globales y declaraciones, es común que el Scanner almacene un identificador ni bien lo detecta, si todavía no se encuentra en la TS. Además de identificadores, a menudo también se almacenan las palabras reservadas y los literales cadenas. De ser así, cada lexema se representará mediante un índice que indica la posición que ocupa en la TS. Entonces el Scanner retornará el par ordenado (palabraReservada, índice) o (identificador, índice) o (literalCadena, índice) según corresponda.

En el caso de las constantes numéricas la situación es más complejas y se presentan dos soluciones:

1. Hay Scanners que guardan la secuencia de caracteres que forman cada constante numérica, sea entera o real, en la TS. En este caso el Scanner retorna (tipoConstanteNumerica, índice).
2. Hay otro diseño de Scanner que convierte la secuencia de caracteres que forma la constante al valor numérico que corresponde, sin guardarlo en la TS. En este caso el Scanner retorna el par ordenado (tipoConstanteNumerica, valor).

Analicemos que sucede con Lenguajes de Programación con estructuras de bloque. El Scanner no puede almacenar ni buscar un identificador en la TS porque el identificador puede pertenecer a distintos bloques, con diferentes atributos y el Scanner no tiene la capacidad de distinguir los diferentes bloques.

Ejemplo

Sea el siguiente fragmento de un programa ANSI C:

```
. . .
{ /*bloque 1*/
int a;
. . .
       {/*bloque 2*/
              char a[20];
              . . .
       }
       . . .
}
{/*bloque 3*/
       struct (int x;} a;
       . . .
}
. . .
```

Hay 3 bloques. Además el bloque 2 está anidado dentro del bloque 1. Hay 3 variables con el mismo nombre, con atributos muy distintos.
Si el Scanner fuera el encargado de almacenar estos identificadores en la TS, encontraría que el 2° identificador “ya está almacenado”, porque el Scanner no conoce la sintaxis del LP y, por lo tanto, no sabe qué es un bloque.
La solución es que el Scanner retorne el identificador detectado y que una rutina semántica resuelva el papel que juega ese identificador , según el bloque en el que está declarado. Para ello normalmente el Scanner utiliza un espacio de cadenas, que es un gran vector de caracteres para almacenar los identificadores y retorna un puntero a este espacio. Posteriormente, en la TS se almacena dónde comienza el identificador en este espacio y su longitud. El bloque en el que está declarado el identificador debería ser otro atributo para el mismo.

Hay una situación interesante que produce un cambio de atributo. Supongamos la siguiente declaración en ANSI C: **`typedef int entero;`** cuando el Scanner analiza esta secuencia de caracteres, detecta 4 lexemas:
la palabra reservada **`typedef`**
la palabra reservada **`int`**
el identificador **`entero`**
el carácter de puntuación **`;`**
En el caso del identificador **`entero`**, el Scanner retornará un puntero al espacio de caracteres y el atributo ID. Pero la declaración **`typedef`** crea una nueva palabra clave **`entero`**. Después que el Parser procesa esta sentencia, él mismo, conjuntamente con la rutina semántica correspondiente, se encargará de colocar el atributo correcto en la TS: un código que represente “nombre de tipo”.

Ejercicios 3

sea la declaración en ANSI C: **`typedef char strings[200][32];`**

1. ¿Qué lexemas detecta el Scanner?
2. ¿Qué información tendrá el identificador strings en la TS después que el Parser procese esta declaración.

Las técnicas usadas en el Análisis Léxico son útiles en muchas otras aplicaciones de programación, no sólo en la construcción de compiladores. Un Scanner es un reconocedor de patrones, y este reconocimiento puede ser aplicado en programas como editores y manejadores de bases de datos bibliográficos.

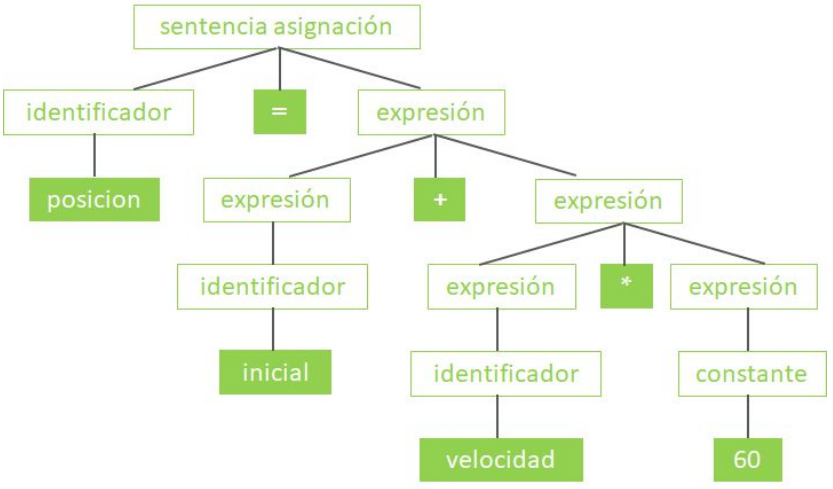
Recuperación de errores

Ciertos caracteres son ilegales en ANSI C porque no pertenecen a los alfabetos de ninguno de los LRs que forman este Lenguaje de Programación como @ o ` (apóstrofe invertido), excepto en dos circunstancias especiales, cuáles? El Scanner se puede recuperar de estos errores de varias maneras. La más simple es descartar el carácter inválido, imprimir un mensaje de error adecuado, pasar un código de error al Parser y continuar el Análisis Léxico a partir del siguiente carácter. El resultado final de todo el proceso ya no será una compilación correcta.

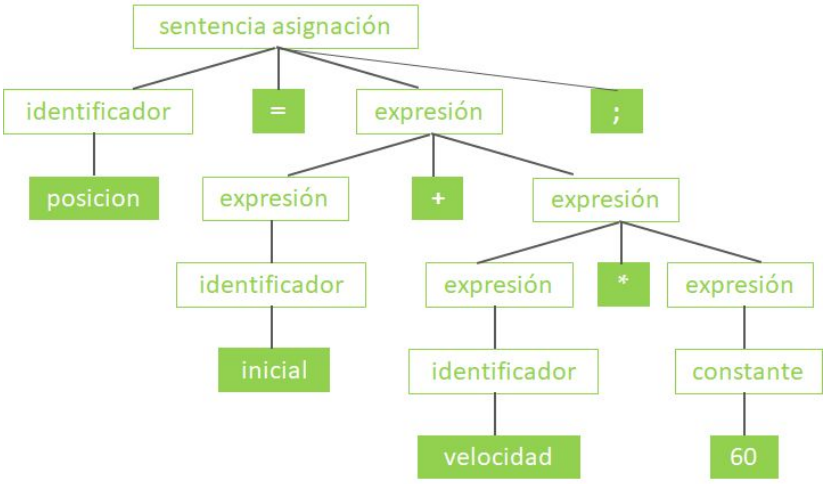
Análisis sintáctico

El Análisis Sintáctico es llevado a cabo por el Parser. Éste verifica si los tokens que recibe del Scanner forman secuencias o construcciones válidas, según la Gramática Sintáctica del LP correspondiente. El Parser está formado por un grupo de rutinas que convierte el flujo de tokens en un **árbol de análisis sintáctico**. Éste árbol representa de modo jerárquico a la secuencia analizada, donde los que forman la construcción son las hojas del árbol.

Ejercicios



1. Realizar una lista de noterminales y terminales de la GIC representados por los nodos del árbol.
2. ¿A qué corresponde en la GIC la raíz del árbol?
3. Si el LP es ANSI C ¿cuál es el error que se detecta en el árbol?



4. Corrigiendo el error ¿cuál es la construcción que representa el árbol?
   posicion = inicial+velocidad*60;
5. Construir una derivación a izquierda que obtenga la secuencia de terminales representada en el árbol.

Tipos de análisis sintácticos y de GICs

Al derivar una secuencia de tokens, si existe más de un noterminal en una cadena de derivación debemos elegir cuál es el próximo noterminal que se va a expandir, o sea cuál será reemplazado por su lado derecho de la producción. Se utilizan dos tipos de derivaciones que determinan cuál será el noterminal a tratar: la derivación a izquierda y la derivación a derecha.
DERIVACIÓN A IZQUIERDA: ocurre cuando se reemplaza el primer noterminal que se encuentra en una cadena de derivación leída de izquierda a derecha.
DERIVACIÓN A DERECHA: ocurre cuando se reemplaza el último noterminal que se encuentra en una cadena de derivación leída de izquierda a derecha.

Es importante destacar la relación que existe entre el tipo de Análisis Sintáctico y el tipo de derivación:

- Análisis Sintáctico Descendente (top-down), produce Derivación a Izquierda que comienza en el noterminal llamado axioma y finaliza con los terminales que forman la construcción analizada.
- Análisis Sintáctico Ascendente (bottom-up) donde se utiliza Derivación a Derecha de una manera especial.

Un Parser Ascendente utiliza Derivación a Derecha , pero en orden inverso. La última producción aplicada en la Derivación a Derecha, es la primera producción que es “descubierta” mientras que la primera producción utilizada, la que involucra al axioma, es la última producción en ser “descubierta”. “Reduce el árbol de análisis sintáctico” hasta llegar al axioma. La secuencia de producciones reconocidas en el Análisis Sintáctico Ascendente es exactamente la inversa a la secuencia de producciones que forma la Derivación a Derecha.

Ejemplo

Para mostrar los dos tipos de derivaciones usaremos una GIC muy simple, sea la GIC con producciones:

1 S → aST 3 T → cT
2 S → b 4 T → d

aabcdd es sintácticamente correcta?

1. Derivación a Izquierda – Análisis Sintáctico Descendente (comienza en el axioma)

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
S (axioma)	S -> aST (1)
aST	S -> aST (1)
aaSTT	S -> b (2)
aabTT	T -> cT (3)
aabcTT	T -> d (4)
aabcdT	T -> d (4)
aabcdd	correcta

2. Derivación a Derecha

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
S (axioma)	S -> aST (1)
aST	T -> d (4)
aSd	S -> aST (1)
aaSTd	T -> cT (3)
aaScTd	T -> d (4)
aaScdd	S -> b (2)
aabcdd	correcta

3. Análisis Sintáctico Ascendente (a partir de la tabla 2, en orden inverso)

Cadena de Derivación	Próxima Producción a Aplicar
aabcdd	S -> b (2)
aaScdd	T -> d (4)
aaScTd	T -> cT (3)
aaSTd	S -> aST (1)
aSd	T -> d (4)

aST	S -> aST (1)
S	correcta

Si bien las GICs son muy utilizadas para definir la sintaxis de un LP, hay reglas sintácticas que no pueden ser expresadas empleando solo GICs. Por ejemplo, la regla que indica que toda variable debe ser declarada antes de ser usada no puede ser expresada en una GIC. En la práctica los detalles sintácticos que no pueden ser representados en una GIC son considerados parte de la semántica estática y son chequeados por rutinas semánticas.

Errores en las GICs

Una GIC es un mecanismo utilizado para definir una sintaxis. Una GIC puede tener errores que deben evitarse antes de construir un compilador. Uno de los errores más importantes surge cuando una GIC permite que un programa tenga una construcción con dos o más árboles sintácticos diferentes, que es equivalente a decir dos o más derivaciones por izquierda diferentes.

Consideremos la GIC con dos producciones:
(1) E -> E + E
(2) E -> num
Si derivamos la expresión num + num + num obtenemos dos derivaciones por izquierda diferentes:

Derivación 1	Derivación 2
E (1)	E (1)
E + E (2)	E + E (1)
num + E (1)	E + E + E (2)
num + E + E (2)	num + E + E (2)
num + num + E (2)	num + num + E (2)
num + num + num	num + num + num

Una GIC que permite dos derivaciones a izquierda para obtener la misma cadena de terminales se denomina ambigua. Estas gramáticas deben ser evitadas.

Gramáticas LL y LR

Los conceptos de derivación por izquierda y derivación por derecha se aplican directamente a los dos tipos de Análisis Sintáctico más comunes (Descendente y Ascendente). La derivación es un proceso muy cercano al que utiliza el Parser para analizar una construcción dada.

Un Análisis Sintáctico LL consiste en analizar el flujo de tokens de izquierda a derecha (L left) por medio de una derivación por izquierda (L left). El ASDR (Análisis Sintáctico Descendente Recursivo) utilizado anteriormente es un ejemplo de LL. Una gramática LL es una GIC que puede ser utilizada en un Análisis Sintáctico LL. No todas las gramáticas son LL, pero, la mayoría de los LP pueden describir sus respectivas sintaxis por medio de las gramáticas LL.

Una gramática LL especial es la llamada LL(1). Esta gramática puede ser utilizada por un Parser LL con un solo símbolo de preanálisis. Esto es: si un noterminal tiene varias producciones, el Parser puede decidir cuál lado derecho debe aplicar con solo conocer el próximo token del flujo de entrada.

Ejemplo

La GIC con producciones: **R → aR | b** Permite construir un Parser LL(1) o Parser Predictivo porque con solo conocer el próximo token de la secuencia analizada, llamado símbolo de preanálisis, determina inequívocamente, si aplica una producción o la otra. Lo visto en el ejemplo se puede aplicar a: **R → aR | a ?**

Otro tipo de análisis sintáctico es el Análisis Sintáctico LR. El Análisis Sintáctico LR consiste en analizar el flujo de tokens de izquierda a derecha (L left) pero por medio de una derivación a derecha (R right), aunque utilizada a la inversa: la última producción expandida en la derivación será la primera en ser reducida. De todas las gramáticas LR, la que más nos interesa en este caso es la gramática LR(1), porque solo requiere conocer el próximo token (el símbolo de preanálisis) para hacer un Análisis Sintáctico correcto, como en el caso de la LL(1).

En general, un Parser LL(1) puede ser construido por una programador a partir de una GIC adecuada. En cambio, un Parser LR(1) requiere utilización de un programa especial tipo *yacc*.

Gramáticas LL(1) y aplicaciones

Ya hemos visto que la sintaxis de un Lenguaje de Programación se define mediante una GIC. Tengamos claro que la GIC no puede definir aquello que sea sensible al contexto. Supongamos que un LP tiene una construcción que consiste en una lista de expresiones aritméticas simples (con operadores de suma y producto), separadas con punto y coma (;).

La siguiente sería una GIC que genera este LIC:

```
GIC1:
1 listaExpresiones: expresión |
2 listaExpresiones ; expresión
3 expresión: término |
4 expresión + término
5 término: factor |
6 término * factor
7 factor: num |
8 ( expresión )
```

La GIC descripta es recursiva a izquierda (producciones 2, 4 y 6). Las producciones recursivas son muy importantes porque permiten generar y describir un LIC infinito con un número finito de producciones. Sin embargo, la GIC1 tiene una problema muy importante para aplicarla al Análisis Sintáctico Descendente de estas listas de expresiones. Muchos Parsers, como el ASDR, no pueden manejar las producciones recursivas a izquierda porque los procedimientos que implementan esas producciones recursivas entrarían en un ciclo infinito.

Ejemplo
Volvamos a la GIC1. Si el Parser tiene que tratar `factor`, definido en las producciones 7 y 8.

```
7 factor: num |
8 ( expresión )
```

No habrá inconvenientes. Sabe cómo elegir el lado derecho de la producción con solo conocer el primer símbolo o token:
• Si éste es **`num`**, el Parser aplicará la producción 7 y reemplazará el factor por **`num`**.
• Si es un **`(`** aplicará la producción 8.
No sucede lo mismo con las producciones 5 y 6:

```
5 término: factor |
6 término * factor
```

En el caso de la 6, su lado derecho comienza con `término` y, por lo tanto, el procedimiento entraría en un ciclo indefinido. El Parser de una GIC LL(1) debe ser capaz de elegir entre los distintos lados derechos de un determinado noterminal con solo conocer el p**róximo token**. Y esto no lo puede hacer con las producciones 5 y 6.

Comparemos las siguientes derivaciones que se definen utilizando las producciones 5, 6 y 7:

```
5 término: factor |
6 término * factor
7 factor: num |
. . . y . . .
término (5)             término (6)
factor (7)              término * factor (5)
num                     factor * factor (7)
                        num * factor
                        . . .
```

En la primera derivación utilizamos las producciones 5 y 7, mientras que en la segunda las producciones 5, 6 y 7. Pero en ambos casos, obtenemos cadenas de derivación que comienzan con el mismo terminal (num) y esto no es correcto en el método de Análisis Sintáctico que estamos viendo.

Esta situación ocurre cuando el Parser no puede decidir qué producción aplicar se llama CONFLICTO y una de las tareas más difíciles en el diseño de un compilador es la creación de una GIC que no tenga conflictos. Varias técnicas producen GICs equivalentes y sin conflictos. Si bien las GICs resultantes no son útiles para una documentación clara del Lenguaje de Programación generado, si lo son para el Análisis Sintáctico.

Repasemos

LL(1) es una GIC que trabaja con un Parser que lee la entrada de izquierda a derecha y que realiza una derivación por izquierda.
LL(1) es muy útil porque solo necesita conocer un símbolo de preanálisis (un token) para realizar la derivación por izquierda.
A continuación veremos como transformar una GIC, que genera y describe la sintaxis de un LP, para que esa GIC tenga la propiedad de ser LL(1).

Obtención de gramáticas LL(1)

Para ser útiles y eficientes, los Análisis Sintácticos Descendentes deben basarse en GICs que sean LL(1). El problema principal es que estas gramáticas no pueden ser recursivas a izquierda, aunque hemos visto que esta propiedad es muy útil para la descripción de la sintaxis de los Lenguajes de Programación. Además, la GIC LL(1) tampoco puede tener un noterminal con dos o más producciones cuyos lados derechos comiencen con el mismo terminal.

Veamos algunas operaciones que nos permiten transformar la GIC empleada en la documentación de la sintaxis de un LP en una GIC equivalente pero apta para ser utilizada por un Parser LL(1) o Parser Predictivo.

Factorización a Izquierda (cuando hay un prefijo común)

Sea

```
sentencia-if :    if(condición) sentencia else sentencia
                  if(condición) sentencia
```

Este par de producciones son muy útil para definir la sentencia if en ANSI C pero no es apto para una GIC LL(1) porque ambos lados derechos comienzan con el mismo token y por lo tanto el Parser no sabrá cuál de las dos producciones seleccionar.
La solución es modificar las producciones de este terminal de tal forma que el prefijo común quedé aislado en una sola producción, en general:

sea $A : \alpha\beta 1$
$\ldots$
$\alpha\beta n$

Donde α es una secuencia de uno o más símbolos (el prefijo) y cada βi es una secuencia de cero o más símbolos, diferentes para cada producción. Entonces factorizando a izquierda, obtenemos estas producciones equivalentes:

$A : \alpha\beta$
$B : \beta 1$
$\ldots$
βn

Si aplicamos este algoritmo a la <sentencia if>, entonces las producciones:

```
sentencia-if :    if(condición) sentencia else sentencia
                  if(condición) sentencia
```

Se transforman en las producciones equivalentes:

```
sentencia-if :    if(condicion) sentencia opción-else
opción-else :     else sentencia
                  ε
```

Ejercicio

Resuelva el problema de prefijos comunes que presenta la siguiente GIC, con producciones:

S → aTbRba | aTbc

T → a | Ta

R → c | cRb

Eliminación de la Recursividad a Izquierda

Como ya hemos visto, las gramáticas LL(1) no pueden ser recursivas a izquierda. Sin embargo la recursividad a izquierda es necesaria para lograr la asociatividad a izquierda de un operador. Existe una forma de eliminar la recursividad a izquierda en una GIC, obteniendo una GIC equivalente.
Sea el par de producciones: $X \rightarrow X\alpha \mid \beta$ Donde X es un noterminal, α y β son secuencia de terminales y noterminales. Este par de producciones generan la $ER = \beta\alpha^*$
Podemos eliminar la recursividad a izquierda para las producciones del noterminal $X(X \rightarrow X\alpha \mid \beta)$ de la siguiente forma:

$X \rightarrow \beta Z$ toda secuencia comienza con β

$Z \rightarrow \alpha Z \mid \varepsilon$ seguida de cero o más α

Hemos eliminado la recursividad a izquierda mediante una transformación a “recursividad a derecha”.

Ejercicio

Una expresión infija básica puede considerarse como una lista de operandos separados por operadores, excepto un caso especial. Su definición más formal sería:
1) un solo operando, o
2) un operando seguido por uno o más pares operador/operando.
Esta estructura se puede definir mediante la siguiente GIC recursiva a izquierda:

```
listaExp :  operando
            listaExp operador operando
```

Eliminar la recursividad a izquierda.

Símbolos de preanálisis: conjunto primero

Para construir un Parser que realice un Análisis Sintáctico Descendente LL(1) es necesario que para cada noterminal de la GIC, el Parser pueda determinar la producción a aplicar con solo conocer cual es el símbolo de preanálisis, o sea el próximo token del flujo de tokens a procesar. Para ello deberemos construir el conjunto PRIMERO para cada no terminal de la GIC.
Informalmente sería:
Si el noterminal *expresión* tiene varias producciones, entonces `PRIMERO(`*expresión*`)` sería el conjunto formado por todos los **terminales** (o tokens) que puedan comenzar una expresión. En el caso de ANSI C y otros LPs, este conjunto incluye números, el signo menos, el paréntesis que abre y otros tokens, pero nunca puede incluir al paréntesis que cierra.

Sea la producción `A → X1 . . . Xn` en la que cada `Xi` puede ser un terminal o un noterminal. Se necesita conocer el conjunto de todos los posibles símbolos de preanálisis que indican que esta producción será elegida. Este conjunto está formado por aquellos terminales que pueden ser producidos por `X1 . . . Xn`
Como un símbolo de preanálisis es un único terminal, se necesita el primer símbolo que puede ser producido por `X1 . . . Xn`. Llamaremos `Primero(X1 . . . Xn)` al conjunto de “primer símbolo” que pueden ser producidos por `X1 . . . Xn`

Obtención del conjunto Primero

Supongamos que existen dos o más producciones para un determinado noterminal. Para distinguir entre estas producciones se debe examinar el conjunto Primero del lado derecho de cada una de las producciones.
Este conjunto se define así:
Sea la producción `A → X1 . . . Xn;`
entonces `Primero( X1 . . .Xn)`
es el conjunto de terminales que pueden iniciar cualquier cadena de derivación que se obtenga a partir de `X1 . . . Xn`
Si `X1 . . . Xn` puede derivar en `ε`, entonces el conjunto `Primero` también contiene `ε`.

Para comprender mejor esta definición veremos la descripción del algoritmo para obtener el conjunto Primero:

1. Si el primer símbolo, `X1`, es un terminal, entonces `Primero(X1 . . . Xn) = X1`
2. Si `X1` es un noterminal, entonces se calculan los conjuntos `Primero` para cada lado derecho de las producciones que tenga `X1`
3. Si `X1` puede generar `ε`, entonces `X1` puede ser eliminada y, entonces `Primero(X1 . . . Xn)` depende de `X2`.
4. Si `X2` es un terminal, entonces es incluido en `Primero(X1 . . . Xn)`
5. Si `X2` es un noterminal, entonces se calculan los conjuntos `Primero` para cada lado derecho de las producciones que tenga `X2`
6. En forma similar, si `X1` como `X2` pueden producir `ε`, consideremos `X3`, luego `X4`, etc.

Supongamos que tratamos de obtener los símbolos de preanálisis que sugieren que se debe aplicar la producción `A → X1 . . . Xn`
¿Qué ocurre si cada `Xi` puede producir `ε`?
En este caso, el símbolo de preanálisis para `A` está determinado por aquellos terminales que siguen inmediatamente al noterminal `A` en las cadenas de derivación que contengan al noterminal `A`.
Este conjunto de terminales de denomina `Siguiente(A)` y es útil porque define el contexto derecho para el noterminal `A`.

Para obtener el conjunto `Siguiente(A)`, inspeccionamos la GIC buscando todas las ocurrencias de `A`. En cada producción se pueden dar estas situaciones:

- El noterminal `A` puede estar seguido por el terminal `x (. . . Ax . . . )`, en cuyo caso `X` pertenece a `Siguiente(A)`
- El noterminal `A` puede estar seguido por el noterminal `B (. . . AB . . .)`, en cuyo caso `Siguiente(A)` incluye `Primero(B)`
- El noterminal `A` puede ser el último símbolo del lado derecho en cierta producción de un noterminal `T (T → Y1 . . .YnA)`, en cuyo caso `Siguiente(A)` incluye `Siguiente(T)`.

Análisis léxico, sintáctico y semántico (parte 2)

Usando una pila para implementar un Parser predictivo

Un Análisis Sintáctico Predictivo puede ser implementado usando una GIC, una pila y un algoritmo en el que la pila es inicializada con el axioma de la GIC.

1. Si la pila está vacía y el flujo de tokens de entrada ha llegado al fdt, el Análisis Sintáctico se ha completado.
2. Si el símbolo que está en el tope de la pila es un noterminal, reemplácelo por su lado derecho pero invirtiendo el orden de los símbolos. Por ej., si la producción es A -> BCD, y A es el noterminal que está en el tope de la pila, aplique la operación *pop* para eliminar A y luego aplique la operación *push* a los símbolos que forman el lado derecho de la producción pero en el orden inverso, o sea D, C, B.
   Si se trata de una producción-ε, solo se elimina (*pop*) el símbolo que está en el tope de la pila sin reemplazarlo por nada. Volver a 1.
3. Si el símbolo que está en el tope de la pila es un terminal, debe coincidir con el símbolo de preanálisis (token):
   a. Si no lo es, ha ocurrido un **Error Sintáctico**;
   b. Si lo es, elimine (*pop*) ese terminal y avance al próximo token del flujo de entrada. Volver a 1.

Ejemplo

Supongamos la GIC en la el noterminal E tiene las siguientes producciones:

```
1 E → aEb |
2     cd
```

Si E está en el tope de la pila, el Parser debe determinar cuál de las dos producciones debe aplicarse en el proceso de derivación. Esta decisión es tomada en función del símbolo de preanálisis: si este es a lo reemplazará en la pila por **bEa**, si es c aplicará la producción 2 y, si es otro símbolo habrá un error sintáctico.

Ejemplo

Sea la GIC LL(1) que define una lista de expresiones aritméticas (con suma, producto y paréntesis), cada una de las cuales termina en ;. La lista completa termina con un **fdt** (fin de token) y puede ser vacía. Las producciones son:

```
1 LE: fdt              6 T: F T'
2 E ; LE               7 T': * F T'
3 E: T E'              8 e
4 E': + T E'           9 F: n
5 e                    10 ( E )
```

Veamos como un Parser con pila procesa la expresión 1+2

```
1 LE: fdt      4 E': + T E'      7 T': * F T'      9 F: n
2 E ; LE       5     ε           8     ε           10  ( E )
3 E: T E'      6 T: F T'
```

	PILA	ENTRADA	COMENTARIO
1	LE	1+2; fdt	Aplicar producción 2
2	-	1+2; fdt	Pop noterminal LE
3	LE;E	1+2; fdt	Aplicar producción 3
4	LE;	1+2; fdt	Pop noterminal E
5	LE;E'T	1+2; fdt	Aplicar producción 6
6	LE;E'	1+2; fdt	Pop noterminal T
7	LE;E'T'F	1+2; fdt	Aplicar producción 9
8	LE;E'T'	1+2; fdt	Pop noterminal F
9	LE;E'T'**n**	1+2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
10	LE;E'T'	+2; fdt	Aplicar producción 8
11	LE;E'	+2; fdt	Pop noterminal T'
12	LE;E'	+2; fdt	Aplicar producción 4
13	LE;	+2; fdt	Pop noterminal E'
14	LE;E'T**+**	+2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
15	LE;E'T	2; fdt	Aplicar producción 6
16	LE;E'	2; fdt	Pop noterminal T
17	LE;E'T'F	2; fdt	Aplicar producción 9
18	LE;E'T'	2; fdt	Pop noterminal F
19	LE;E'T'**n**	2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
20	LE;E'T'	; fdt	Aplicar producción 8
21	LE;E'	; fdt	Pop noterminal T'
22	LE;E'	; fdt	Aplicar producción 5
23	LE;	; fdt	Pop noterminal E'
24	LE;	; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar

25	LE	fdt	Aplicar producción 1
26	-	fdt	Pop noterminal LE
27	fdt	fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
28	-		Fin Análisis Sintáctico

Veamos qué sucede cuando existe un error sintáctico. Para el flujo de entrada **1+2*;**
El proceso de análisis sintáctico realizado en el ejemplo anterior resultará útil porque los pasos 1 al 19 son idénticos.

	PILA	ENTRADA	COMENTARIO
1	LE	1+2; fdt	Aplicar producción 2
...	...	...	...
19	LE;E'T'**n**	2; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
20	LE;E'T'	*; fdt	Aplicar producción 7
21	LE;E'	*; fdt	Pop noterminal T'
22	LE;E'T'F*	; fdt	El terminal en el tope de la pila concuerda con el símbolo de preanálisis; eliminarlo y avanzar
23	LE;E'T'F	; fdt	El no terminal F tiene 2 producciones, ninguna comienza con un terminal que coincida con el símbolo de preanálisis => **ERROR SINTÁCTICO**

Lenguaje de los paréntesis anidados, AFP y Parser dirigido por Tabla

Las expresiones con operadores infijos, como tiene la mayoría de los LP, tienen cierta complejidad debido a que la gramática debe representar la precedencia y asociatividad de cada operador y debe mostrar el uso correcto de los paréntesis.

Ejemplo

(4*(2+((3))))

es una expresión sintácticamente correcta. Estos paréntesis anidados constituyen un LIC que puede ser reconocido por un AFP.

Un ***Parser Dirigido por una Tabla*** (table-driven parser) está basado en un AFP. Este Parser es dirigido por una máquina de estados y utiliza una pila para mantener un registro del progreso de esta máquina de estados.

Ejemplo

Supongamos un LIC cuyas palabras son paréntesis anidados, como (()(())). Una GIC que genera este LIC puede tener las siguientes producciones:

```
listaParéntesis     :   plista
                  plista listaParéntesis
plista : ( listaParéntesis )
      ε
```

Construiremos el AFP que reconozca el LIC de los paréntesis anidados. Estas son las características del AFP, algunas de las cuales lo diferencian del AFP tradicional:

1. La pila está integrada al autómata.
2. En la pila se guardan números enteros que representan los nombres de los estados.
3. En el tope de la pila estará el número de estado actual.
4. La tabla de estados tiene tantas filas como estados haya y tantas columnas como caracteres o símbolos tenga el alfabeto. En este caso los símbolos son solo **(** , **)** y **fdt**
5. El contenido de la Tabla de Estados no mostrará “el próximo estado” sino la acción que debe llevar a cabo para cada par (estado en el tope de la pila, símbolo de entrada).
6. Hay cuatro tipos de acciones:
   a. aceptada: la secuencia de caracteres analizada es una palabra del lenguaje.
   b. error: hay un error sintáctico en la cadena analizada.
   c. push N: para insertar el estado N en la pila, N es el nuevo estado actual.
   d. pop: para quitar el estado en el tope de la pila, habrá un nuevo estado actual.
7. Con cada operación push o pop se avanza un carácter en la sentencia de entrada.

Esta es la tabla de Estados del AFP que reconoce el lenguaje de los paréntesis:

Estado	**(**	**)**	**fdt**
0	push 1	error	aceptado
1	push 1	pop	error

Ejemplo

Usemos este AFP para realizar el Análisis Sintáctico de (( )(( )))

Pila (tope a la derecha)	Entrada a analizar	Próxima acción
0	**(( )(( ))) fdt**	Push 1 y avanza
01	**( )(( ))) fdt**	Push 1 y avanza
011	**)(( ))) fdt**	Pop y avanza
01	**(( ))) fdt**	Push 1 y avanza
011	**( ))) fdt**	Push 1 y avanza
0111	**))) fdt**	Pop y avanza

011	`)) fdt`	Pop y avanza
01	`) fdt`	Pop y avanza
0	`fdt`	Acepta

Análisis Sintáctico Ascendente (Bottom-up)

Estudiaremos el Análisis Sintáctico Ascendente realizado a partir de una GIC LR(1).
L: el flujo de tokens es leído de izquierda a derecha.
R: utiliza derivación a derecha.
Toda gramática LL(1) es también LR(1), pero no la inversa. Para construir un Parser a partir de una gramática LR se requiere una herramienta tipo *yacc* que ayude a construir las tablas utilizadas por el Parser.
El ASA es realizado por un Parser Ascendente, que es un AFP: requiere un conjunto de estados y una pila. El Parser construye el árbol sintáctico en forma ascendente (desde las hojas hacia la raíz). Atendiendo solo la pila , el ASA trabaja así:

1. Si los elementos que están en la parte superior de la pila forman el lado derecho de una producción, se quitan (*pop*) y se reemplazan por el noterminal del lado izquierdo (*push*). Esto se conoce como la operación de REDUCCIÓN. Si se trata de la producción nula, no hay pop. REDUCCIÓN es el proceso de reemplazar el lado derecho de una producción en la pila con el correspondiente noterminal del lado izquierdo.
2. Caso contrario, *push* el símbolo de entrada actual (un token) en la pila y avance en el flujo de tokens de entrada (*input*). Esta operación se conoce como DESPLAZAMIENTO porque desplaza el siguiente token desde el flujo de entrada a la pila. DESPLAZAMIENTO es el proceso de mover un token desde el *input* a la pila, *push* al token actual y avanza el *input* hasta el próximo token.
3. Si la operación previa consistió en una REDUCCIÓN que resultó con el noterminal objetivo (el axioma) en el tope de la pila, el *input* es aceptado y el ASA se ha completado. De lo contrario volver al paso 1.

Ejemplo

Una GIC para poder realizar el ASA debe tener un noterminal “objetivo” con una sola producción. Esta gramática no podría ser utilizada para un Análisis Sintáctico Descendente porque es recursiva a izquierda. Realizaremos el proceso ASA con la expresión: 1*(2+3)

```
0 S -> E          2 E -> T          4 T -> F          6 F -> num
1 E -> E + T      3 T -> T * F      5 F -> ( E )
```

	PILA (tope a la derecha)	INPUT (flujo de tokens)	PRÓXIMO PASO
1	**vacía**	1*(2+3)	Desplazar
2	**num**	*(2+3)	Reducir con producción 6
3	**F**	*(2+3)	Reducir con producción 4

4	**T**	*(2+3)	No se puede reducir con producción 2 porque después no existe E* => Desplazar
5	**T***	(2+3)	Desplazar
6	**T*(**	2+3)	Desplazar
7	**T*(num**	+3)	Reducir con producción 6
8	**T*(F**	+3)	Reducir con producción 4
9	**T*(T**	+3)	Reducir con producción 2
10	**T*(E**	+3)	Desplazar
11	**T*(E+**	3)	Desplazar
12	**T*(E+num**	)	Reducir con producción 6
13	**T*(E+F**	)	Reducir con producción 4
14	**T*(E+T**	)	Reducir con producción 1
15	**T*(E**	)	Desplazar
16	**T*(E)**		Reducir con producción 5
17	**T*F**		Reducir con producción 3
18	**T**		Reducir con producción 2
19	**E**		Reducir con producción 0 porque no hay input
20	**S**		FIN CORRECTO

Ahora procesaremos un input incorrecto: **1*(2+3))**
El proceso y la tabla diseñada serán iguales hasta el paso 18

```
0 S -> E          2 E -> T          4 T -> F          6 F -> num
1 E -> E + T      3 T -> T * F      5 F -> ( E )
```

	PILA (tope a la derecha)	INPUT (flujo de tokens)	PRÓXIMO PASO
1	**vacía**	1*(2+3)	Desplazar
...	...	...	...
18	**T**		Reducir con producción 2
19	**E**	)	No se puede reducir E al objetivo porque todavía hay input => Desplazar

20	**E)**		La GIC no tiene producción con E) en su lado derecho y no hay más tokens => no se puede reducir ni desplazar => FIN INCORRECTO

Recursividad

- Ya no es necesario que las gramáticas están factorizadas.
- Es indiferente que sean recursivas a derecha o a izquierda.
- No debe ser una gramática ambigua.

Análisis semántico

Desde el punto de vista de la etapa de Análisis del compilador, el Análisis Semántico consiste, básicamente, en la realización de tareas de verificación y de conversión que no puede realizar el Análisis Sintáctico. Los Lenguajes de Programación a los que nos referimos tienen una sintaxis representada mediante gramáticas INDEPENDIENTE DEL CONTEXTO (significa que no tiene en cuenta lo precedente ni lo posterior, no tiene en cuenta el contexto en el que se define la sintaxis de cualquier construcción del LP).
El Análisis Semántico debe verificar que se cumplan todas las reglas sensibles al contexto como, por ejemplo, que una variable haya sido declarada antes de su utilización. Para ello, el Análisis Semántico utiliza mucho la Tabla de Símbolos. Hay reglas sintácticas que no se pueden expresar por medio de GICs. Por ejemplo: la regla que dice “toda variable debe ser declarada antes de utilizarla”, no puede ser representada por una GIC.

Ejemplo

Sea la producción de una GIC que define así la sintaxis de una asignación en algún LP:
`Asignación: identifacadorVariable = constanteEntera;`
En esta producción no puede saber si la variable ha sido declarada o no. En la práctica, los detalles sintácticos que no pueden ser representados por la GIC son resueltos mediante rutinas semánticas.

Otras dificultades semánticas que aparecen en el proceso de compilación y que deben ser resueltas durante el Análisis Semántico son:

- La GIC no puede informarnos si una variable usada en una expresión es del tipo correcto.
- La GIC no puede reconocer declaraciones múltiples en una mismo bloque (como int x; float x;).
- Si una expresión aritmética es sintácticamente correcta pero los operandos son de diferentes tipos numéricos, debe haber un proceso de conversión (si el LP lo permite) para que las instrucción del lenguaje objeto puedan operar.

Ejercicio

Sea la expresión 24 + 4.16

1. ¿Qué ocurre durante el Análisis Léxico?
   si lo analizo desde el punto de vista léxico, está bien escrito.

2. ¿Qué ocurre durante el Análisis Sintáctico?
   desde este punto de vista, corresponde a una expresión. Si busco la GIC de una expresión esto me va a dar ok. Sintácticamente es correcto.
3. ¿Qué ocurre durante el Análisis Semántico?
   durante el análisis semántica detecta que los operandos son de distinto tipo, entonces transforma el operando en una constante real para luego poder realizar la operación de seuma.

Una GIC define un lenguaje compuesto por un conjunto de secuencias de token. En general, toda secuencia de tokens derivable desde una GIC se considera sintácticamente válida para el compilador. Cuando la Semántica Estática es chequeada por las rutinas semánticas en un programa sintácticamente válido, se pueden descubrir errores semánticos.
Por ejemplo, en Pascal, la sentencia
`A := 'X' + true;`
no tiene errores sintácticos pero sí tiene un error semántico porque el operador + no está definido para sumar un carácter a un valor booleano.

ANSI C: Derivable VS Sintácticamente correcto

Varias clases atrás se introdujeron dos conceptos:

- construcción DERIVABLE de una GIC.
- construcción SINTÁCTICAMENTE CORRECTA.

Si bien este tema es aplicable a muchos LPs , nos ocuparemos específicamente del ANSI C. En muchos casos, una construcción cumple con ambos conceptos, pero en otras veces, no. Si es DERIVABLE, significa que la GIC lo genera; pero si es derivable y no es SINTÁCTICAMENTE CORRECTA, significa que la GIC lo genera pero en la fase de Análisis Semántico se detecta un error insalvable.

Ejemplo

Para ejemplificar esta última situación, partamos de un subconjunto muy simplificado de la GIC que genera expresiones en ANSI C:

```
1 expAsignacion : expUnaria = expCondicional
2 expCondicional: expUnaria
3 expUnaria   :   expPrimaria
4 expPrimaria  :  identificador
                  constante
```

Según esta GIC, se puede DERIVAR la expresión 2 = 4, aunque no sea correcta. Obviamente no le podemos asignar un valor a una constante. Esta inconsistencia sintáctica se determina durante el Análisis Semántico.

Ejercicios

1. Sea el siguiente bloque en ANSI C:

```
{
int a;
2 = a;
double b;
```

```
b = 4++;
}
```

   a. Describir el resultado del Análisis Léxico.
   b. Describir el resultado del Análisis Sintáctico.
   c. Describir el resultado del Análisis Semántico.

2. Sea el siguiente bloque en ANSI C:

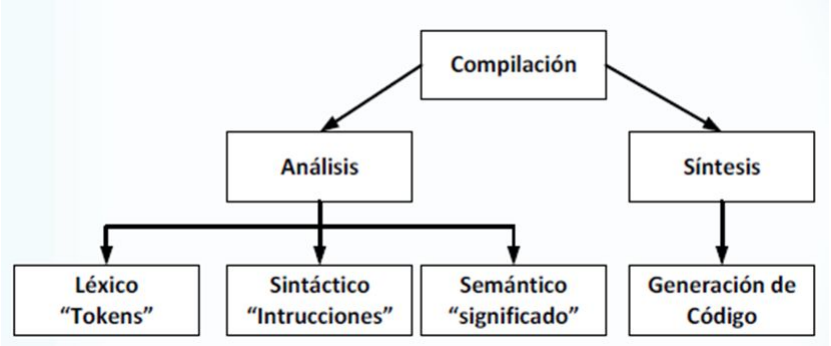
```
{
int a, b;
double c;
a = b + c;
}
```

   a. Describir el resultado del Análisis Léxico.
   b. Describir el resultado del Análisis Sintáctico.
   c. Describir el resultado del Análisis Semántico.

Proceso de compilación (RESÚMEN)

Un compilador es un programa que lee un programa en un lenguaje y lo traduce a un programa equivalente en otro lenguaje y además informa al usuario sobre la presencia de errores en el programa de entrada. Se divide en dos fases:

- La fase de ANÁLISIS (Front End), que analiza la entrada y genera estructuras intermedias. Está formado por el Análisis Léxico, Análisis Sintáctico y Análisis Semántico.
- La fase de SÍNTESIS (Back End), se ocupa de la Generación de código intermedio, la Generación de código de máquina.

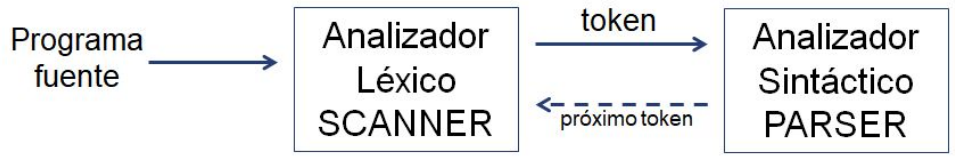


Análisis léxico

Consiste en detectar palabras de los Lenguajes Regulares que forman parte del Lenguaje de Programación, esta tarea es realizada por el analizador léxico o Scanner.

La función primordial es agrupar caracteres de la entrada en tokens Estos tokens son suministrados ("bajo demanda") al analizador sintáctico Los tokens se pasan como valores "simples", por ejemplo: Identificador, Operador, Constante, CaracterPuntuación,etc.

Normalmente, un Analizador Léxico no retorna una lista de tokens, sino que retorna un token cuando el Analizador Sintáctico se lo pide.



Tokens

```
Identificadores
Constantes
Palabras reservadas (if, while, etc.)
Operadores ( +, -, *, /)
Comparadores (>, <, >=, <=, ==)
etc.
```

Los tokens se diferencian de la cadena de caracteres que representan. La cadena de caracteres es el Lexema o valor léxico.
Puesto que un token puede representar más de un lexema, el Scanner debe enviar información adicional al Parser, en forma de atributo/s. Esa información será usada en las próximas etapas del compilador. Por cada token detectado, el Analizador Léxico entrega un par: **<token, atributo>**

Algunos tokens requieren algo más que su propia identificación. Por ejemplo:

- las constantes requieren su valor.
- los identificadores requieren el String.

Es decir que el scanner debe realizar, a veces, una doble función: identificar el token y evaluarlo.

Atributos de los tokens

Lexema, nro. de línea en que aparece el token, valor, etc.
En la práctica, la información adicional para cada token se almacena en una **Tabla de Símbolos**, y el atributo entregado es el puntero a la entrada correspondiente en la Tabla de Símbolos.

La Tabla de Símbolos

Es una estructura de datos que contiene un registro para cada identificador (y todo otro token que pueda representar más de un lexema) utilizado en el código fuente, con campos que contienen información relevante para cada símbolo (atributos). Cuando el Análisis Léxico detecta un token de tipo identificador, u otro token que pueda representar más de un lexema, lo ingresa en la Tabla de Símbolos. La misma será utilizada por las siguientes etapas del compilador.

Errores léxicos

En el proceso para detectar tokens, el scanner puede encontrar errores A estos errores se los denomina Errores Léxicos. Algunos errores léxicos típicos son:

- nombres ilegales de identificadores: un nombre contiene caracteres inválidos.
- números incorrectos: un número contiene caracteres inválidos o no está formado correctamente, por ejemplo 3,14 o 0.3.14 en vez de 3.14

Los errores léxicos se deben a descuidos del programador. En general, la recuperación de errores léxicos es sencilla y siempre se traduce en la generación de un error de sintaxis que será detectado más tarde por el analizador sintáctico cuando el analizador léxico devuelve un componente léxico que el analizador sintáctico no espera en esa posición. Los métodos de recuperación de errores léxicos se basan en saltarse caracteres en la entrada hasta que un patrón se ha podido reconocer; o bien usar otros métodos más sofisticados que incluyen la inserción, borrado, sustitución de un carácter en la entrada o intercambio de dos caracteres consecutivos.

Construcción de un scanner

1. Definir las diferentes categorías léxicas o tokens, por ejemplo, por medio de expresiones regulares.
2. Construir un programa que analice los caracteres del programa fuente a compilar e identifique los tokens.

El scanner puede detectar los tokens a partir de la implementación de Autómatas Finitos o con un programa de reconocimiento de patrones desarrollado en Lex o Flex utilizando las expresiones regulares extendidas (regex) correspondientes a la definición de los tokens.

Análisis sintáctico

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes de contexto (GIC). El análisis sintáctico impone una estructura jerárquica a la cadena de componentes léxicos, que se representará por medio de árboles sintácticos. Los árboles se construyen con los tokens que envía el Scanner.

El Parser es el encargado de hacer el análisis sintáctico. Existen dos formas de análisis sintáctico:

- análisis DESCENDENTE (Top-Down) construye el árbol desde la raíz (axioma) hasta llegar a las hojas;
- análisis ASCENDENTE(Bottom-Up) construye el árbol desde las hojas hacia la raíz (axioma).

Análisis sintáctico descendente

El Árbol de Análisis Sintáctico (AAS) que se construye tiene las siguientes propiedades:

- La raíz del AAS está etiquetada con el axioma de la GIC.
- Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
- Cada nodo interior está etiquetado con un noterminal.
- Este árbol es abstracto, no se crea en memoria sino que se expresa a través de la lista de reglas aplicadas.

La forma más natural de implementar este analizador descendente es asociar un procedimiento con cada noterminal (PAS: Procedimiento de Análisis Sintáctico) Cada

procedimiento comprueba si el token en estudio es correcto y si es incorrecto (error sintáctico) actúa en consecuencia (mensaje de error y/o estrategia de recuperación).

Construcción de Procedimientos de Análisis Sintáctico (PAS)

Los PAS reflejan las definiciones dadas por las producciones de la gramática sintáctica. Recordemos lo visto anteriormente sobre GIC's: *En general: sean **v** y **v''** no terminales y sea t un terminal. Entonces las producciones de una GIC corresponden a este formato general:*
v → **(v'' + t)*** donde **v** y **v''** *pueden representar el mismo noterminal.*
La estructura de cada PAS sigue el desarrollo del lado derecho de la producción. Las producciones pueden ser recursivas o no.

Producciones no recursivas

Los terminales y no terminales son procesados en el orden en que aparecen.

a) Del lado derecho hay una única sentencia. Ejemplo:

```
A → <E> <C> PUNTO
void A (void) {
      E( );
      C( );
      Match (PUNTO);
}
```

b) Del lado derecho hay más de una sentencia. En estos casos es necesario averiguar cuál es el próximo token que recibirá el parser para decidir cuál sentencia se deberá procesar. Para averiguar cuál será el próximo token se usa la función ProximoToken().

Producciones recursivas

Se utiliza un ciclo infinito del cual se sale cuando se averigua que el token que recibirá el parser no es el esperado.

Análisis sintáctico descendente predictivo

- Análisis Sintáctico Descendente: • También denominado "Top Down".
- Análisis Sintáctico en el que se parte del axioma hasta llegar al programa de usuario.
- Se dice que este proceso es LL. "Left" porque lee el programa de izquierda a derecha. "Left" porque aplica las reglas buscando la correspondencia con la parte izquierda.

Gramáticas LL(1) Para que un Análisis Sintáctico Descendente sea eficiente debe basarse en GICs que sean LL(1). Para ello es necesario:

- Eliminación de la ambigüedad (cuando genera más de un árbol de derivación).
- Eliminación de la recursividad a izquierda.
- Factorización a izquierda.

NO ES POSIBLE USAR ANÁLISIS SINTÁCTICO DESCENDENTE SI LA GRAMÁTICA ES RECURSIVA A IZQUIERDA

Se debe cambiar la gramática eliminando la recursividad a izquierda.

Sea el par de producciones: X → Xα | β Donde X es un noterminal, α y **b** son secuencia de terminales y noterminales. Se puede eliminar la recursividad de la siguiente forma:

X → β Z

Z → α Z | ε

Cuando un no terminal de una GIC tiene dos o más producciones con un prefijo común hay que evitarlo ya que no se puede determinar el símbolo de preanálisis. Hay que realizar una factorización a izquierda.

Ejemplo

```
sentencia-if   :  if(condición) sentencia else sentencia
                  if(condición) sentencia
```

Se transforman en las producciones equivalentes:

```
sentencia-if  :   if(condicion) sentencia opción-else
opción-else   :   else sentencia
                  ε
```

Conjunto primero

El cálculo de los conjuntos PRIMERO y SIGUIENTE, es necesario para la construcción de los analizadores sintácticos, tanto ascendentes como descendentes. El conjunto PRIMERO es el conjunto de terminales que pueden aparecer primeros en las cadenas que se derivan de aquella que estamos analizando.

Si hacemos el cálculo de PRIMERO de un símbolo de la gramática, por ejemplo X, se expresaría como PRIMERO(X) y se calcularía de la manera siguiente:

1. Si X es un símbolo que pertenece al conjunto de los terminales, entonces PRIMERO(X) = {X}.
2. Si X es ε, entonces PRIMERO(X) = {ε}.
3. Si X es un no terminal y X → Y1 Y2... Yn se incluirá los que haya en PRIMERO(Y1) y si de Y1 deriva ε, además de otros terminales, se incluirán estos terminales y en lugar de ε, lo que haya en PRIMERO(Y2), y así sucesivamente hasta llegar a Yn. Solo se incluiría ε, si estuviera en todas las producciones que se derivan de Y1 Y2... Yn.
4. Repetir hasta que no se puedan añadir más terminales o ε a ningún conjunto PRIMERO.

Ejemplo

X → Y1 Y2

Y1 → a B | ε

Y2 → b C | ε

PRIMERO(X) = PRIMERO(Y1)

y para calcular PRIMERO(Y1) tenemos que calcular PRIMERO(aB) y PRIMERO(ε) que son las dos producciones que se derivan de Y1.

(1) PRIMERO(aB) = {a} (2) PRIMERO(ε) = {ε}.

Por tanto obtenemos PRIMERO (Y1) = {a, ε}.

Al aplicar (3), y puesto que en PRIMERO(Y1) se encuentra eε, entonces hay que calcular también PRIMERO(Y2), y sacar del cálculo de PRIMERO(X) a ε.

Esto sería: {a, ε} - {ε} ∪ PRIMERO(Y2) = {a} ∪ {b, ε} = {a, b, ε}.

Como ε, se encuentra en todas las producciones también se incluiría (si Y2 fuera de la siguiente forma, Y2 → b C, y por tanto no incluyera ε, no se incluiría).

Por tanto PRIMERO(X) = {a, b, ε}.

Análisis sintáctico ascendente

Análisis Sintáctico Ascendente:

- También denominado “Bottom Up”.
- Análisis Sintáctico en el que se divide el programa de usuario hasta llegar al axioma.
- Se dice que este proceso es LR. “Left” porque lee el programa de usuario de izquierda a derecha. “Right” porque para aplicar las reglas busca asociar con el lado derecho de la definición de cada regla.
- Ya no es necesario que las gramáticas están factorizadas.
- Es indiferente que sean recursivas a derecha o a izquierda.
- No debe ser una gramática ambigua.

Cualquier mecanismo de análisis ascendente consiste en partir de una configuración inicial e ir aplicando operaciones, cada una de las cuales permite pasar de una configuración origen a otro destino. El proceso finalizará cuando la configuración del destino llegue a ser tal que se llegue al axioma y se hayan consumido todos los tokens.

Las operaciones disponibles son las siguientes

1. ACEPTAR: se acepta la cadena: EOF y axioma inicial.
2. RECHAZAR: la cadena de entrada no es válida.
3. REDUCIR: Consiste en reemplazar el lado derecho de una producción con el correspondiente noterminal del lado izquierdo.
4. DESPLAZAR: Se consume token siguiente que forma parte de la lectura de la entrada del usuario de izquierda a derecha.

Análisis ascendente de gramática LR(1). La técnica de análisis sintáctico LR(k). La abreviatura LR obedece a que la cadena de entrada es examinada de izquierda a derecha, mientras que la “R” indica que el proceso proporciona el árbol sintáctico mediante la secuencia de derivaciones a derecha en orden inverso. Por último, la “k” hace referencia al número de tokens de pre-búsqueda utilizados para tomar las decisiones sobre si reducir o desplazar.

Análisis semántico

Completa el trabajo del análisis sintáctico. La tarea es realizada por las rutinas semánticas, ellas se encargan de chequear la semántica estática de cada construcción y generar código para una máquina virtual. El Analizador Semántico finaliza la fase de Análisis del compilador

y comienza la fase de Síntesis, en la cual se comienza a generar el código objeto. Las rutinas semánticas son llamadas por el Parser.
El chequeo semántico se encarga, por ejemplo, de:

- comprobar si los tipos de datos que intervienen en las expresiones son compatibles;
- comprobar si los argumentos de una función o procedimiento coinciden en tipo y cantidad con los parámetros;
- comprobar si un identificador ha sido declarado antes de utilizarlo;
- verificar que no exista declaración repetida de identificadores.

Los errores que se pueden producir si no se comprueba lo que acabamos de mencionar pertenecen a la semántica estática, son los errores que se pueden determinar en tiempo de compilación. Existen otros errores que sólo se manifiestan durante la ejecución, por ejemplo una división por cero o un subíndice fuera de rango.

Tabla de símbolos

La tabla de símbolos es una estructura de datos que contiene toda la información relativa a cada identificador que aparece en el programa fuente. Los identificadores pueden ser nombres de variables, tipos de datos, funciones, etc. Cada lenguaje de programación tiene características propias que se reflejan en la tabla de símbolos. Cada elemento de la estructura de datos que compone la tabla de símbolos está formado al menos por el identificador y sus atributos. Los atributos son la información relativa de cada identificador.
Algunos de los atributos habituales son: Especificación del identificador: variable, tipo de datos, función, etc. Tipo: en el caso de variables será el identificador de tipo (real, entero, carácter, cadena de caracteres, o un tipo definido previamente). En el caso de las funciones puede ser el tipo devuelto por la función. Dimensión o tamaño. Puede utilizarse el tamaño total que ocupa una variable en bytes, o las dimensiones. Los atributos pueden variar de unos lenguajes a otros, debido a las características propias de cada lenguaje y a la metodología de desarrollo del compilador.
La tabla de símbolos se crea por cooperación del análisis léxico y el análisis sintáctico. El análisis léxico proporciona la lista de los identificadores, y el análisis sintáctico permite rellenar los atributos correspondientes a cada identificador. El analizador semántico también puede rellenar algún atributo. El analizador semántico y el generador de código obtienen de la tabla de símbolos la información necesaria para realizar su tarea.

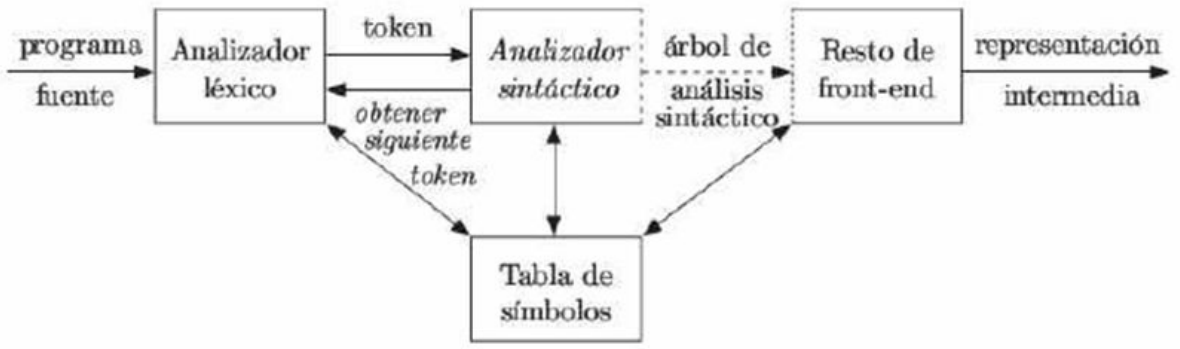


Figura 4.1: Posición del analizador sintáctico en el modelo del compilador

BISON

YACC

Las siglas del nombre significan
Y*et*
A*nother*
C*ompilers*
C*ompiler,*
Es decir, "Otro generador de compiladores más".

Es un utilitario para generar parsers. Recibe un archivo en el que se define una gramática, similar a BNF y las acciones semánticas a realizar (en código C) Presupone un escáner al que puede llamar invocando la función yylex. Puede trabajar con lex pero NO es un requisito. Yacc está discontinuado, la herramienta que se usa hoy día se llama Bison. Bison es compatible con yacc.

Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto en un programa en C que analiza esa gramática. Todas las gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes.
Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

```
%{
Declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional
```

Declaraciones en C

Se definen variables y funciones que después se usarán en las acciones de las reglas gramaticales. También se incluyen los encabezados .h que sean necesarios.

Declaraciones de Bison

Permiten definir elementos de la gramática como los símbolos terminales y no terminales, la precedencia de operadores y el tipo de valor semántico de cada símbolo.

Reglas gramaticales

Se define la gramática con anotaciones a analizar, con la notación propia de Bison, o sea, las producciones de la gramática.

Código C adicional

Puede contener cualquier código C que desee utilizar. Suele ir la definición del analizador léxico yylex, más subrutinas invocadas por las acciones en las reglas gramaticales.

Declaraciones en C

En esta sección deberá incluir todas las declaraciones que se utilizarán en el código y deberán enmarcarse por los signos
%{
%}
Las declaraciones pueden ser:

- Includes;
- Defines;
- Variables Globales.

Ejemplo

```
%{
#include <stdio.h>
#define …
int matrizEstados [34] [22] ;
int q;
typedef pila_ass* pilaAss;
…
%}
```

Declaraciones en Bison

En esta sección deberá incluir todas las declaraciones de:

- tokens (%token);
- asociatividades (%left, %right,…);
- start symbol (%start).

No es obligatorio declarar el start symbol o axioma explícitamente dentro de la sección de declaraciones. Si no se declara se tomará el no terminal que se encuentre en la primera regla escrita en la sección de reglas

Declaración de tokens

Los símbolos terminales de la gramática se denominan tokens y deben declararse en esta sección. Por convención se suelen escribir los tokens en mayúsculas y los símbolos no terminales en minúsculas.

Hay tres maneras de escribir símbolos terminales en la gramática. Aquí se describen las dos más usuales:

- Un token declarado se escribe con un identificador, de la misma manera que un identificador en C. Por convención, debería estar todo en mayúsculas. Cada uno de estos nombres debe definirse con una declaración de %token.
- Un token de carácter se escribe en la gramática utilizando la misma sintaxis usada en C para las constantes de un carácter; por ejemplo, ‘+’ es un tipo de token de carácter. Un tipo de token de carácter no necesita ser declarado a menos que necesite especificar el tipo de datos de su valor semántico, asociatividad, o

precedencia. Por convención, un token de carácter se utiliza únicamente para representar un token consistente en ese carácter en particular.

Ejemplo

```
%token ID
%token CTE
%left MAS MENOS
%right IGUAL
%token PARENTIZQ, PARENTDER, MAS , MENOS
%token ASIGNACION ":="
```

Reglas gramaticales

Esta sección deberá incluir todas las definiciones de las reglas gramaticales necesarias para llevar adelante el análisis sintáctico. Deberá enmarcarse por los signos:
%%
%%

Una regla gramatical de Bison tiene la siguiente forma general:

```
resultado : componentes...
          ;
```

resultado es el símbolo no terminal que describe esta regla.
componentes son los diversos símbolos terminales y no terminales que están reunidos por esta regla.

Los nombres de los elementos no terminales pueden:

- tener una longitud arbitraria;
- estar conformados por letras, puntos, guión bajo y dígitos (no pueden empezar con éstos).
- Mayúsculas y minúsculas son indistintas.

Los elementos terminales deberán corresponderse con los objetos definidos en la sección declaración de tokens Por convención se suelen escribir los tokens en mayúsculas y los símbolos no terminales en minúsculas.
Se pueden escribir por separado varias reglas para el mismo resultado o pueden unirse con el carácter de barra vertical '|' así:

```
resultado: componentes-regla1...
         | componentes-regla2...
         ...
         ;
```

Ejemplo

```
programa : INICIO lsentencias FIN
           ;
lsentencias : sentencia
            | lsentencias sentencia
           ;
```

Bison puede manejar tanto recursión a izquierda como a derecha, sin embargo es recomendable tratar de evitar la recursión a derecha dado que es lenta. En caso que una

producción sea ε se deja vacía, o se pone un comentario (`/*epsilón*/`) o bien se usa la declaración `%empty`.

Ejemplo
```
noterm1 : /*epsilón*/
          | noterm1 UNTOKEN ;
noterm2 : %empty | ALGO ;
```

No es necesario declarar todos los tokens, en particular aquellos que se componen de un único carácter puedo usarlos entre comillas simples: exp : exp '+' exp ;

Ejemplo
LIC analizado por el parser es generado por la GIC:
S → aTc
T → aTc|b
%%
S :'a' T 'c' ;
T :'a' T 'c' |'b' ;
%%

Precedencia y asociatividad

Bison permite escribir una BNF ambigua, pero resuelve el problema indicando precedencia y asociatividad por separado. Puedo indicar la asociatividad y precedencia de un operador cambiando `%token` por `%left, %right` o `%nonassoc` Si se quiere declarar precedencia pero sin asociatividad, se utiliza `%precedence`.
Para un subconjunto de lenguaje C podría tener las siguientes declaraciones:
%right '='
%left '<' '>'
%left '+' '-'

Acciones

Una acción acompaña a una regla sintáctica y contiene código C a ser ejecutado cada vez que se reconoce una instancia de esa regla. La tarea de la mayoría de las acciones es computar el valor semántico para la agrupación construida por la regla a partir de los valores semánticos asociados a los tokens o agrupaciones más pequeñas. Una acción consiste en sentencias de C rodeadas por llaves, muy parecido a las sentencias compuestas en C. Se pueden situar en cualquier posición dentro de la regla; la acción se ejecuta en esa posición.
El código C en una acción puede hacer referencia a los valores semánticos de los componentes reconocidos por la regla con la construcción **$n**, que hace referencia al valor de la componente n-ésima. El valor semántico para la agrupación que se está construyendo es **$$**.
Esta regla construye una **exp** de dos agrupaciones **exp** más pequeñas conectadas por un token de signo más. En la acción, **$1** y **$3** hacen referencia a los valores semánticos de las dos agrupaciones **exp** componentes, que son el primer y tercer símbolo en el lado derecho de la regla. La suma se almacena en **$$** de manera que se convierte en el valor semántico

de la expresión de adición reconocida por la regla. Si hubiese un valor semántico útil asociado con el token **+**, debería hacerse referencia con **$2**.
Si no especifica una acción para una regla, Bison suministra una por defecto: **$$ = $1**. De este modo, el valor del primer símbolo en la regla se convierte en el valor de la regla entera. Por supuesto, la regla por defecto solo es válida si concuerdan los dos tipos de datos. No hay una regla por defecto con significado para la regla vacía; toda regla vacía debe tener una acción explícita a menos que el valor de la regla no importe.

Invocación

La rutina para llamar al parser generado por bison tiene la declaración:
`int yyparse (void)`
y devuelve:
0 si tuvo éxito,
1 si hay errores sintácticos,
2 si no alcanzó la memoria ram para ejecutar.

La función del analizador léxico, **yylex**, reconoce tokens desde el flujo de entrada y se los devuelve al analizador. Bison no crea esta función automáticamente. Se debe escribir de manera que `yyparse` pueda llamarla. En programas simples, `yylex` se define a menudo al final del archivo de la gramática de Bison. En programas un poco más complejos, lo habitual es crear un programa en Flex que genere automáticamente esta función y enlazar Flex y Bison.
No hace falta agregar una regla objetivo explícitamente ya que bison la provee. Si nuestro axioma es **programa**, bison agrega la regla: `$accept: programa $end` donde **`$accept`** y **`$end`** son no terminales predefinidos por bison (que no puede ser usados en la gramática). **`$end`** se activa cuando flex envía EOF (lo hace automáticamente, no es necesario una regla del tipo `<<EOF>> {return EOF;}`

Compilación y ejecución

Al ejecutar el comando
bison nombre_fuente.y
o
yacc nombre_fuente.y
se crea un fichero en C llamado
nombre_fuente.tab.c
Existe la opción **–y** que fuerza a que el archivo de salida se llame
y.tab.c

Otra opción útil es la opción **–d**, que genera el archivo con las definiciones de tokens que necesita Flex (si se ha usado la opción **–y**, el archivo se llama **y.tab.h**).
bison –y –d arhivo.y
Este archivo **y.tab.h** normalmente se incluye en la sección de declaraciones del fuente de Flex.

El fichero **y.tab.c** se puede compilar con la instrucción **gcc y.tab.c**
Los pasos para usar conjuntamente Flex y Bison serán normalmente:

1. **yacc -yd fuente.y**

2. **flex fuente.l**
3. **gcc y.tab.c lex.yy.c –o salida**

Estos pasos generan un ejecutable llamado salida que nos permite comprobar qué palabras pertenecen al lenguaje generado por la gramática descrita en el fichero Bison.
Para poder probarlo con entradas más largas, lo más sencillo es crear archivos de texto y usar redirecciones para que el ejecutable lea estos ficheros como entrada a analizar. Por ejemplo si creamos el analizador llamado `salida` y un fichero de texto con datos para su análisis, llamado `entrada.txt`, podemos ejecutar
salida < entrada.txt

FLEX

Introducción

Flex es un una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones. Es compatible con Lex, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL.
Los ficheros de entrada de Flex (normalmente con extensión .l), siguen el siguiente esquema:

```
%%
patrón1 {acción1}
patrón2 {acción2}
...
```

Donde:
patrón: expresión regular;
acción: código C con las acciones a ejecutar cuando se encuentre concordancia del patrón con el texto de entrada.

Flex recorre la entrada hasta que encuentra una concordancia y ejecuta el código asociado. El texto que no concuerda con ningún patrón lo copia tal cual a la salida. Por ejemplo:

```
%%
a*b {printf("X");};
re ;
```

El ejecutable correspondiente transforma el texto: **abre la puertaab** en **X la puertX** pues ha escrito X cada vez que ha encontrado ab o aab y nada cuando ha encontrado re.

Flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. Flex genera como salida un fichero fuente en C, 'lex.yy.c', que define una función 'yylex()'.

El fichero de entrada de Flex está compuesto de tres secciones Estas secciones están separadas por una línea donde aparece únicamente un **%%**

```
definiciones
%%
```

```
reglas
%%
código de usuario
```

Ejemplo

```
%{
/*Detecta e imprime los números enteros*/
#include <stdio.h>
%}
%%
[0-9]+ {printf("%s\n", yytext);}
.|\n ;
%%
int main(void){
	yylex();
	return 0;
}
```

Secciones

1. Definiciones:
   está delimitada por %{ y %}, introduce codificación en ANSI C que será copiada en el programa final. También puede contener definición de nombres sencillas para simplificar la especificación del scanner: `nombre defincion`
   Ejemplo:
   ```
   DIGITO [0-9]
   ID [a-z][a-z0-9]*
   ```
2. Reglas:
   Está delimitada por %% al comienzo y al final. Cada regla está formada por dos partes: `patron accion` patron descripto mediante una metaER; accion descripta mediante un bloque en ANSI C. Ambas partes están separadas por uno o más espacios.
3. Subrutinas del usuario:
   Puede contener cualquier codificación en ANSI C.

Patrones

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares. Ejemplos:

- x empareja el carácter x
- . cualquier carácter excepto una línea nueva
- [xyz] un conjunto de caracteres, empareja una x, una y o z
- [abj-oZ] un conjunto de caracteres con un rango; empareja una a, una b, cualquier letra desde la j hasta la o, o una Z
- [^A-Z] cualquier carácter menos los que aparecen en el conjunto. En este caso, cualquier carácter EXCEPTO una letra mayúscula.
- [^A-Z\n] cualquier carácter EXCEPTO una letra mayúscula o una línea nueva
- r* cero o más r's, donde r es cualquier expresión regular
- r+ una o mas r's
- r? cero o una r (es decir, "una r opcional")

- r{2,5} entre dos y cinco concatenaciones de r
- r{4} exactamente 4 r's
- "[xyz]\"foo" la cadena literal: [xyz]"foo
- rs la expresion regular r seguida por la expresion regular s (concatenación)
- r|s una r o una s

Emparejamiento de la entrada

Cuando el escáner generado está funcionando, analiza su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Si encuentra más de un emparejamiento, toma el que empareje el texto más largo. Si encuentra dos o más emparejamientos de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de Flex.
Ejemplo

```
%%
aa {printf(“1”);}
aab {printf(“2”);}
uv {printf(“3”);}
xu {printf(“4”);}
```

Con la entrada **Laabdgf xuv**, daría como salida **L2dgf 4v**

1. Se copiará L
2. Se reconocerá aab (porque es más largo que aa) y realizará el printf(“2”)
3. Se copiará dgf
4. Se reconocerá xu (porque aunque tienen la misma longitud que uv, xu está antes que uv en el fuente) y realizará printf(“4”);
5. Se copiará v

Acciones

Cada patrón en una regla tiene una acción asociada, que puede ser cualquier código en C. El patrón finaliza en el primer carácter de espacio en blanco que no sea una secuencia de escape; lo que queda de la línea es su acción. Si la acción está vacía, entonces cuando el patrón se empareje el token de entrada simplemente se descarta.

Si la acción contiene un ‘{‘, entonces la acción abarca hasta que se encuentre el correspondiente ‘}’, y la acción podría entonces cruzar varias líneas. Flex es capaz de reconocer las cadenas y comentarios de C.

El analizador generado

La salida de Flex es el fichero **lex.yy.c**, que contiene la función de análisis **yylex()**, varias tablas usadas por esta para emparejar tokens, y unas cuantas rutinas auxiliares y macros. Siempre que se llame a **yylex()**, este analiza tokens desde el fichero de entrada global **yyin** (que por defecto es igual a stdin). La función continúa hasta que alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia `return`.

Variables

Algunos de los diferentes valores disponibles al usuario en las acciones de reglas:
`char *yytext`

Apunta al texto del token actual (última palabra reconocida en algún patrón). Por ejemplo: `printf("%s",yytext)` lo escribiría en pantalla

`int yyleng`

Contiene la longitud del token actual

Compilación y ejecución

Al ejecutar el comando **flex nombre_fichero_fuente** se creará un fichero en C llamado **lex.yy.c**

Si se compila este fichero con la instrucción `gcc -o ejecutable lex.yy.c` se obtendrá como resultado un fichero ejecutable llamado `ejecutable`.

Una vez se ha creado el fichero ejecutable se puede ejecutar directamente. Flex esperará que se introduzca texto por la entrada estándar (teclado) y lo analizará. Para terminar con el análisis normalmente hay que pulsar `<CTRL> z` (Windows).

Para poder probarlo con entradas más largas, lo más sencillo es crear archivos de texto y usar redirecciones para que el ejecutable lea estos ficheros como entrada a analizar. Por ejemplo si hemos creado un analizador llamado `prueba1` y un fichero de texto con datos para su análisis, llamado `entrada.txt`, podemos ejecutar `prueba1 < entrada.txt`

Nota final

Flex requiere un formato bastante estricto de su fichero de entrada. En particular los caracteres no visibles (espacios en blanco, tabuladores, saltos de línea) fuera de sitio causan errores difíciles de encontrar. Sobre todo es muy importante no dejar líneas en blanco de más ni empezar reglas con espacios en blanco o tabuladores.