Notas para la Materia de Compiladores

José Antonio Camarena Ibarrola

Marzo de 2008

Índice general

1. Introducción **5**
1.1. Objetivo 6
1.2. Justificación 6
1.3. Usuarios 6
1.4. Definiciones 7
1.5. El Análisis Léxico 8
1.6. El Análisis Sintáctico 9
1.7. El Análisis Semántico 9
1.8. Generador de Código Intermedio 9
1.9. El Optimizador de Código 10
1.10. La Tabla de Símbolos 10
1.11. Manejo de Errores 10

2. Análisis Léxico **11**
2.1. Construcción de Analizadores Léxicos 11
2.2. El Generador de Analizadores lexicos: *lex* 13

3. Análisis Sintáctico **17**
3.1. Análisis Sintáctico Descendente 17
3.1.1. Parser descendente recursivo 18
3.1.2. Parser predictivo descendente para gramáticas LL(1) . 19
3.2. Análisis Sintáctico Ascendente 32
3.2.1. Parsers LR 33
3.3. El generador de analizadores sintácticos: yacc 41

Capítulo 1

Introducción

Escribir un compilador es normalmente el primer gran Sistema de Software que un estudiante de Ingeniería en Computación desarrolla y requiere conocimientos de las siguientes áreas:

- Programación de Computadoras
- Arquitectura de Computadoras
- Teoría de Autómatas y Lenguajes Formales
- Ingeniería de Software
- Programación de Sistemas

El estudiante requiere entonces toda la ayuda posible. Estas Notas fueron escritas con el ánimo de facilitar la compresión de las técnicas que se utilizan para desarrollar las partes de un compilador que son el analizador lexicográfico, el analizador sintático, el generador de código intermedio y el optimizador de código. Hasta el momento, estas notas se centran en los dos primeros módulos pero incluyen una introducción a las valiosas herramientas conocidas como lex y yacc (o sus clones conocidos como flex y bison), este último no solo es útil en el análisis sintáctico sino en la generación de código.

1.1. Objetivo

Proveer al alumno con principios y técnicas útiles para la construcción de Compiladores. El alumno deberá ser capaz de implementar la traducción de un lenguaje de programación de alto nivel al lenguaje de máquina de un computador.

1.2. Justificación

El estudiante debe aprender como implementar compiladores utilizando la tecnología actual, estas notas deberán serle de gran ayuda.

1.3. Usuarios

Estudiantes de Ingeniería en COmputación de la Facultad de Ingeniería Eléctrica

1.4. Definiciones

Un compilador es un TRADUCTOR de un lenguaje de programación de alto nivel a un lenguaje ensamblador el cual será traducido a su vez a código objeto por algún ensamblador [1], [2], [3]. El conjunto de compiladores es un subconjunto del producto cartesiano del conjunto de Lenguajes de alto nivel, el conjunto de computadoras y el conjunto de sistemas operativos. Por Ejemplo el compilador de C++ para McCintosh en ambiente Linux se podría representar por:

(C++,MAC,Linux)

Otros ejemplos podrían ser:

(Modula2,PC,DOS), (Pascal,Risc,Unix), (Java,Sun,OS/2), Etc....

Para colmo, se generan versiones nuevas frecuentemente de compiladores ya existentes. Esto quiere decir que hay mucha gente el mundo trabajando en el desarrollo de los compiladores. El primer compilador de Fortran IV requirió de trabajo en equipo durante casi quince años. Sin embargo, ahora se cuenta con muchas herramientas CASE para el desarrollo de compiladores, así como bases matemáticas y toda una tecnología que por cierto se utiliza también para el desarrollo de otro tipo de software como:

- Editores de Ecuaciones (Ej TEX)
- CAD's de Electrónica (Ej OrCAD)
- Parser de Consultadores de BD (Ej SQL)
- Compilación de Textos (Ej Latex)
- Editores de código fuente
- Impresores de Código fuente
- Traductores de lenguajes formales

Un compilador tiene básicamente dos etápas:

1. La etapa de análisis del código fuente
2. La de Síntesis de código Ensamblador

La tecnología desarrollada para la etapa de análisis de código fuente se utiliza también en software como: Editores e impresores de Código fuente, Verificadores de software y también en Intérpretes. La etapa del análisis consta de tres partes o fases:

1. El analizador léxico
2. El analizador sintáctico
3. El analizador semántico

La etapa de síntesis también consta de tres partes:

1. El generador de código intermedio
2. El optimizador de código
3. El generador de código

Adicionalmente se debe de contar con dos partes más, las cuales no forman una etapa en sí sino que se utilizan a lo largo de las diversas fases del compilador. Estas son:

1. El administrador de tabla de símbolos
2. El manejador de errores

1.5. El Análisis Léxico

Se conoce también como análisis lineal debido a que se trata de un barrido secuencial del código fuente. La función de un analizador léxico es la de identificar a los elementos sintácticos básicos del lenguaje los cuales son indivisibles y a los que llamamos tokens. El anallizador debe de ignorar caracteres blancos (espacios, tabuladores y retornos de carro) y reconocer a la cadena mas larga de caracteres que forme un token válido. Por ejemplo, debe decidir que la cadena “for3” es un token del tipo identificador y no dos identificadores del tipo palabra reservada (for) seguida de otro token del tipo constante entera (3). El analizador léxico entrega el tipo de token de que se trata mediante una constante predeterminada y la cadena leída a la cual por cierto llamamos lexema. Una manera preferida de entregar el lexema es insertándolo en una tabla conocida como “Tabla de símbolos” y entregando la posición que el lexema ocupa en esa tabla. El analizador léxico es llamado por el analizador sintáctico cada vez que este requiere de otro token.

1.6. El Análisis Sintáctico

Se conoce también como análisis jerárquico debido a que convierte una secuencia de tokens en un árbol de sintaxis el cual es como sabemos una estructura jerárquica. Algunos compiladores construyen realmente el árbol, esto puede ser útil para la fase de optimización, otros en cambio, solamente llevan el registro de en que parte del árbol se encuentran.

La división entre el análisis léxico y el sintáctico es un tanto arbitraria, un buen criterio para dividirlos consiste en decidir si determinada construcción requiere recursividad o no. Por ejemplo, para reconocer identificadores no se necesita la recursión, basta con autómatas finitos, sin embargo, con estos no sería posible analizar expresiones matemáticas o sentencias estructuradas.

1.7. El Análisis Semántico

Verifica si tiene sentido el programa fuente, es en esta fase en donde se hace la verificación de tipos, algunos analizadores semánticos convierten el dato de menor precisión al de mayor precisión con el que se está realizando alguna operación, otros marcan conflicto de tipos. Pero todos deben marcar error si se utiliza un flotante como exponente de un arreglo por poner un ejemplo.

1.8. Generador de Código Intermedio

Esta fase del compilador no es en realidad una parte separada del compilador, la mayoría de los compiladores generan código como parte del proceso de análisis sintáctico, esto es debido a que requieren del árbol de sintaxis y si este no va a ser construido físicamente, entonces deberá acompañar al analizador sintáctico al barrer el árbol implícito. En lugar de generar código ensamblador directamente, los compliadores generan un código intermedio que es más parecido al código ensamblador, las operaciones por ejemplo nunca se hacen con más de dos operandos. Al no generarse código ensamblador el cual es dependiente de la computadora específica, sino código intermedio, se puede reutilizar la parte del compilador que genera código intermedio en otro compilador para una computadora con diferente procesador cambiando solamente el generador de código ensamblador al cual llamamos back-end, la desventaja obviamente es la lentitud que esto conlleva.

1.9. El Optimizador de Código

Debe al menos eliminar código que no hace nada así como variables no referenciadas. Un buen optimizador de código debe modificar la manera como se pretende implementar un algoritmo para que este sea ejecutado más eficientemente, los optimizadores de código pueden configurarse para optar por ahorro de memoria o por ahorro de tiempo de procesador.

1.10. La Tabla de Símbolos

tiene información acerca de los identificadores que aparecen en el programa, como el tipo de identificador (nombre de función, nombre de una variable, etc.), la memoria asignada, el ámbito o alcance del mismo, etc.

El analizador léxico introduce registros en la tabla de símbolos pero deja en blanco campos cuya contenido no se puede determinar durante el análisis lineal, estos serán llenados y/o utilizados por las fases restantes

1.11. Manejo de Errores

Un compilador que se detiene al encontrar el primer error no es de mucha utilidad. Así pues, es necesario que el compilador sea capaz de recuperarse de los errores encontrados y de proporcionar mayor información que conduzca a la fácil reparación de los mismos, de hecho el compilador ideal debería corregir los errores el mismo.

La fase de análisis léxico detecta errores donde los caracteres no forman ningún token válido o en donde aparezcan caracteres inválidos.

La fase de análisis sintáctico detecta errores donde se violan las reglas de sintaxis.

La fase de análisis semántico detecta errores donde las construcciones son sintácticamente correctas, pero no tienen ningún sentido . Por ejemplo, la construcción

id1 + id2

no tiene coherencia si id1 es el nombre de una función y id2 es el nombre de un arreglo.

Capítulo 2

Análisis Léxico

El análisis léxico simplifica la labor del análisis sintáctico permitiendo que este no tenga que lidiar con caracteres sino con tokens. La obligación del analizador léxico, como se dijo antes es la de identificar la cadena mas larga de caracteres que constituya un token del buffer de entrada.

2.1. Construcción de Analizadores Léxicos

Básicamente hay dos formas en que en analizador léxico y el analizador sintáctico pueden interactuar:

La primera, consiste en un analizador léxico que haga una pasada sobre todos los caracteres que constituyen el código fuente y escriba en un archivo de salida la secuencia de tokens reconocidos, el analizador sintáctico utilizaría este archivo como su entrada y lo barrería completamente para formar el arbol de sintaxis en lo que sería realmente una segunda pasada del código fuente.

La segunda consiste en que el analizador sintáctico llame al analizador léxico cada vez que este requiera otro token, de manera que vaya formando el árbol mientras el analizador léxico barre el código fuente. De esta manera en una sola pasada se realizan ambas labores. Esta forma de trabajar es la que se prefiere por ser la más rápida. Los tokens son representados por una constante entera y los lexemas son introducidos en la tabla de símbolos, de manera que el analizador léxico regresa la constante entera y también la posición que ocupa en la tabla la cadena que constituye el lexema. Al trabajar así, el analizador léxico lee caracter a caracter el archivo fuente y normalmente para encontrar una cadena que constituye un lexema, ya

leyó un caracter de más,. Por ejemplo, la linea "a=3.14;" consta de cuatro tokens, el primero es un identificador de variable, el segundo es un operador de asignación, el tercero es una constante de punto flotante y el cuarto es el punto y coma. Al barrer la linea, para saber si se trata del operador de asignación o del operador de comparación ("==") despues de leer el caracter '=' debe de leer el siguiente caracter (El '3') de esta manera concluye que solo puede ser el operador de asignación, el problema es que ya leyó el '3', este caracter va a necesitarse la próxima vez que el analizador léxico llame al analizador sintáctico, por lo tanto antes de retornar debe *desleer* dicho caracter, es decir regresarlo al archivo de donde lo leyo para posteriormente volver a leerlo.

Para regresar caracteres a un flujo de entrada, el lenguaje C provee la función ungetc(c,flujo), sin embargo, leer un archivo caracter por caracter y encima regresando algunos caracteres es bastante lento, para acelerar el proceso de barrido, es muy conveniente leer un buén numero de caracteres de un solo golpe y guardarlos en una cadena que funcione como buffer interno con dos apuntadores auxiliares, uno apuntando al final del último lexema leido y otro al último caracter leido. De esta forma desleer un caracter es tan rápido como decrementar un apuntador. Si al final del buffer interno se coloca un caracter centinela se facilita el encontrar el final del buffer de entrada.

La elección del conjunto de tokens es muy importante en el desempeño del compilador. Si el conjunto es demasiado grande, la compilación se vuelve complicada, es decir, el analizador sería muy grande, si es demasiado pequeño, el analizador léxico es el complicado. Por ejemplo, si '$>$' fuera el lexema de un token y '$<$' fuera el de otro, entonces requeriríamos de dos reglas de producción en la gramática:

$$expresion \rightarrow expresion > expresion \tag{2.1}$$

y

$$expresion \rightarrow expresion < expresion \tag{2.2}$$

en lugar de la regla:

$$expresion \rightarrow expresion\ op_rel\ expresion \tag{2.3}$$

donde el token *op_rel* tiene el conjunto de lexemas posibles $>$, $<$, $==$, $!=$, $>=$, $<=$, $=>$ y $=<$

Un autómata finito es capaz de reconocer expresiones regulares, los tokens de un lenguaje pueden ser descritos como una expresión regular. Apoyándose en esto podemos implementar un analizador léxico como un autómata finito para el cual la cinta de entrada es el archivo con el código fuente

Básicamente se tratará de un ciclo que barre todo el archivo de entrada una sentencia switch y una variable que almacene el estado que guarda el autómata.

2.2. El Generador de Analizadores lexicos: *lex*

lex es una utileria de UNIX que genera el código fuente en C del autómata capaz de identificar los tokes especificados por el usuario en un archivo de entrada que tiene el siguiente formato:

```
%{
Declaraciones en C
%}
Declaraciones de lex (macros)
%%
fuente de lex (Expresiones regulares) y opcionalmente acciones en C
%%
Funciones en C
```

El archivo con la extensión .lex puede comenzar con definiciones de tipos, declaraciones de variables y constantes, así como prototipos de funciones y macros en C, todo esto encerrado entre %{ y %}. Enseguida, y también opcionalmente se declaran macros de lex las cuales serán usadas en la parte del fuente de lex. El fuente de lex va encerrado entre %% y %% y consta de expresiones regulares que definen tokens opcionalmente seguidas de acciones escritas en código C que se ejecutan siempre y cuando el autómata identifique un token mediante la expresión regular correspondiente. Finalmente, al final del archivo se pueden escribir funciones en C, estas funciones pueden ser llamadas desde las acciones asociadas a las expresiones regulares o desde otras funciones. También se pueden reemplazar funciones estandar como main(), yywrap() o yyerror() entre otras.

Las expresiones regulares de lex utilizan los operadores que se muestran en la Tabla 2.1.

Tabla 2.1: Operadores utilizados por *lex*

Operador	Descripción
[ ]	Corchetes para especificar conjuntos de caracteres
∧	Para designar el complemento de un conjunto
*	Cerradura de Kleen
+	Cerradura positiva
|	Operador **or**
()	Para agrupar expresiones
-	Para indicar un intervalo
{}	para usar una macro de las definidas en la sección de declaraciones de lex
?	para indicar una parte opcional
“”	comillas para especificar una secuencia de caracteres en orden estricto
\	para escapar caracteres de la interpretacion de lex
.	para especificar cualquier caracter
/	Se reconoce la expresión regular a la izquierda de / solo si se encuentra seguida de la expresion regular a la derecha de /

En el siguiente ejemplo, se declara la macro D para representar al conjunto de dígitos de 0 al 9 y la macro E para especificar un exponente en la notación ingenieril (Recuerde que en esta notación 1E3 es equivalente a 0.001), el cual consta opcionalmente de un signo + o un signo - seguido por una secuencia de digitos del cero al nueve o al menos uno solo de ellos. La primera expremasión regular dice que una secuencia de dígitos es un entero y que un flotante puede tener una secuencia de dígitos antes del punto decimal o despues o en ambos lugares y esta opcionalmente seguida de un exponente en la notación ingenieril.

```
D [0-9]
E [EDed][-+]?{D}+
%%
{D}+   puts("Entero");
{D}+"."{D}*{E}?  |
{D}*"."{D}+{E}?
puts("Flotante");
```

```
%%
main() { yylex(); }

yywrap() { return 1; }
```

En lex, el lexema leído es guardado en la variable yytext. Esta información es utilizada en el siguiente ejemplo, el cual no tiene macros pero si la parte de declaraciones de C donde se declara la variable k que es utilizada en las acciones. En este ejemplo, se suma 3 a los enteros múltiplos de 7.

```
%{
   int k;
%}
%%
[0-9]+ {
            k=atoi(yytext);
            if ((k%7)==0) printf("%d",k+3); else ECHO;
       }
%%
main() { yylex(); }

yywrap() { return 1; }
```

ECHO equivale a la acción probablemente mas utilizada de todas:

```
printf("%s",yytext)
```

En lex, la longitud del lexema leído se encuentra en la variable yyleng, este dato es utilizado en el siguiente ejemplo para hacer un estudio estadístico de longitudes de palabras de un archivo. la función yywrap() sirve para controlar lo que va a hacer un autómata cuando encuentre el final de un archivo, si debe finalizar o seguir esperando datos. Si yywrap regresa 1, el programa termina y si regresa 0, yylex() asume que yywrap() abrió otro archivo de manera que habrá mas datos para procesar. De cualquier manera yywrap() se ejecuta cada vez que se encuentra el final del archivo y en el siguiente ejemplo se utiliza para imprimir los resultados del estudio estadístico hecho sobre el archivo.

```
%{
   int longitudes[10];
%}
%%
[a-z]+ longitudes[yyleng]++;
%%
main() { yylex(); }

yywrap() {
   int i;
   printf("Longitud \t Numero de palabras\n");
   for (i=0;i<10;i++)
       if (longitudes[i]>0) printf("%d \t %d \n",i,longitudes[i]);
   return 1;
}
```

Las variables yyin y yyout indican la entrada y la salida estandar de lex respectívamente, estas pueden ser modificadas por las funciones en C.

Las funciones input() y unput() sirven para leer y desleer un caracter de la entrada respectívamente, estas pueden ser reescritas para hacer nuestras propias versiones de input y/o unput en cuyo caso es necesario escribir

```
#undef input
#undef unput
```

Esto es debido a que están implementadas como macros. Para obtener mayor información respecto a lex, ver la referencia [4].

Capítulo 3

Análisis Sintáctico

El analizador sintáctico tiene el propósito de construir el árbol de sintaxis del programa fuente. Los analizadores sintácticos pueden clasificarse en descendentes y ascendentes dependiendo de si construyen el árbol de abajo hacia arriba o de arriba hacia abajo. Los analizadores sintácticos descendentes facilitan la construcción de los árboles de sintaxis si es que estos han de construirse realmente, mientras que los ascendentes permiten manejar una mayor clase de gramáticas.

3.1. Análisis Sintáctico Descendente

Un analizador sintáctico descendente puede ir barriendo la secuencia de tokens al tiempo que construye el arbol de la sig. manera:

1. Se comienza a construir el arbol por la raíz etiquetada por el símbolo inicial S de la gramática.

2. Se elige una de las reglas de producción que tienen a S del lado izquierdo y se ponen como hijos de la raíz a los símbolos (terminales y variables) que se encuentran del lado derecho de la regla de producción seleccionada en el mismo orden en el que están en dicha regla . Los terminales constituyen hojas del arbol y los no-terminales nodos que serán la raíz de subárboles.

3. Se barren los nodos que se encuentran abajo de S verificando que las hojas concuerden con los tokens de la entrada y reemplazando las variables por un subárbol que corresponda a alguna regla de producción

donde aparezca la variable a la izquierda, los hijos del subárbol serán de nuevo etiquetados con los símbolos que aparezcan a la derecha de la regla de producción en cuestión.

4. Si alguna hoja no concuerda con el token leido se descarta la regla de producción seleccionada y se elije otra, de no haber se procede a descartar la regla de producción que dió origen al nivel inmediato superior, se elije otra regla y así sucesívamente hasta llegar al símbolo inicial. Si se han agotado las reglas para el símbolo inicial se marca error de sintaxis.

3.1.1. Parser descendente recursivo

Este método de análisis sintáctico se implementa asociando a cada variable de la gramática una función, que verifique por cada terminal que corresponda a una regla gramatical este corresponda con el token leido y por cada variable hace un llamado a la función correspondiente a esa variable. Por ejemplo, considere la gramática mostrada en la ecuación (3.1):

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow bA \\ A &\rightarrow cBd \\ B &\rightarrow d \end{aligned} \tag{3.1}$$

La función para la variable S sería:

```
S() {
   if ((lee_token()=='a')&&A()) return TRUE;
   deslee_token('a');
   return FALSE;
}
```

Una Función similar se implementa para A y otra para B.

Para construir un árbol de sintaxis de manera descendente es necesario elegir la secuencia de reglas de producción mediante las cuales se puede hacer

una derivación por la izquierda. El parser descendente descrito al inicio de la sección, así como el parser descendente rescursivo son muy lentos en la práctica, debido a que ambos realizan "back-track", es decir, no eligen de manera determinista a las reglas de producción y esta manera de trabajar consume demasiado tiempo. En ciertas gramáticas, es posible seleccionar las reglas de producción de manera determinista, estas gramáticas se conocen como gramáticas LL. Los parsers que pueden elegir de manera determinista (sin back-track) la secuencia de reglas de producción para realizar la derivación por la izquierda requerida se llaman parsers predictivos.

3.1.2. Parser predictivo descendente para gramáticas LL(1)

Un lenguaje LL es aquel que puede ser analizado barriendo la entrada de izquierda a derecha (Por eso la primera L) y al hacerlo, realiza una derivación por la izquierda (De ahí la segunda L). Un lenguaje LR es aquel que puede ser analizado barriendo la entrada de izquierda a derecha pero al hacerlo realiza una derivación por la derecha (De ahí la R). LL(k) es una gramática que puede ser analizada utilizando en cada paso k tokens de look ahead de la entrada para decidir cual regla de producción utilizar al realizar un paso en la derivación de la cadena de entrada.

Las gramáticas-S son un tipo particular de gramáticas LL(1) en la cual las partes derechas de las reglas de producción siempre comienzan con terminales y estos son diferentes para todas las reglas correspondientes a un no-terminal cualquiera. Formalmente, decimos que en una gramática-S todas las reglas de producción para cada uno de los no-terminales A son de la forma:

$$A \rightarrow a_1\alpha_1|a_2\alpha_2|...|a_m\alpha_m \qquad a_i \in V_T\ \forall_i, \quad a_i \neq a_j\ \forall_{i\neq j} \tag{3.2}$$

Para analizar este tipo de gramáticas, se puede utilizar el algoritmo formal que se presenta en la forma de una función M:

$$M : \{V \cup \{\#\} \times \{V_T \cup \{\#\}\} \rightarrow \{(\beta, i), pop, acepta, error\} \tag{3.3}$$

Es decir, la función M es una función que aplica el conjunto de pares ordenados que indican el símbolo en el tope de la pila y el caracter leido de la entrada en el conjunto de acciones a tomar, las cuales pueden ser: Elegir la i-ésima regla de producción que tiene β del lado derecho de la misma,

hacer pop a la pila, aceptar la entrada o bién. marcar error. La función M se describe como a continuación:

$$M(A,a) = \begin{cases} pop & si \quad A = a \quad \forall a \in V_T \\ acepta & si \quad A = \# \quad y \quad a = \# \\ (a\alpha, i) \quad si \quad A \rightarrow a\alpha & es\ la\ i - esima\ produccion \\ error & en\ cualquier\ otro\ caso \end{cases} \tag{3.4}$$

Por ejemplo, considere la gramática mostrada en la Ecuación (3.5):

$$\begin{array}{rl} 1. & S \rightarrow aS \\ 2. & S \rightarrow bA \\ 3. & A \rightarrow c \\ 4. & A \rightarrow dA \end{array} \tag{3.5}$$

A la cual agregamos la siguiente regla de producción:

$$0. \quad S' \rightarrow S\#$$

Para analizar la entrada abddc#, a la cual agregamos # al final antes de comenzar, procedemos de acuerdo al siguiente trazado de instantaneas:

(abddc#,S#,ε)$\vdash$(abddc#,aS#,1)$\vdash$(bddc#,S#,1)$\vdash$

(bddc#,bA#,12)$\vdash$(ddc#,A#,12)$\vdash$(ddc#,dA#,124)$\vdash$

(dc#,A#,124)$\vdash$(dc#,dA#,1244)$\vdash$(c#,A#,1244)$\vdash$

(c#,c#,12443)$\vdash$(#,#,12443)$\vdash$(ε,ε,12443)

Cada instánenea es una tercia ordenada que indica la parte de la entrada que aún no ha sido leida como primer elemento de la tercia. El segundo

elemento es el contenido de la pila, el tercer elemento es la secuencia de reglas de producción utilizadas para construir el arbol de sintaxis. Inicialmente la pila almacena S#, así que leyendo 'a' de la cadena de entrada y estando 'S' en el tope de la pila se debe elegir la regla 1 (S→aS) de manera que se saca "S" del stack y se mete "aS". Ahora tenemos en la entrada lo mismo que en el tope de la pila, entonces se avanza un caracter en la entrada y se saca una caracter de la pila (pop). Lo que leemos ahora de la entrada es una b, mientras que en el tope de la pila hay una S, por eso debemos elegir la regla 2 (S→bA) y entonces sacar la S de la pila y a cambio meter bA. Este proceso sigue hasta que de la entrada leeemos # y este mismo símbolo lo tenemos en el tope de la pila. Por tanto, aceptamos la entrada.

Para facilitar la implementación de este parser, se contruye una Tabla de Parsing que también agiliza el proceso. En cada paso de análisis, el parser lee un token de la entrada y obtiene el símbolo del tope de la pila, estos datos determinan la columna y el reglón respectívamente de una localidad de la tabla donde se especifica la acción. En la Tabla 3.1 se muestra la Tabla de parsing correspondiente a la gramática de la ecuación (3.5).

Tabla 3.1: Tabla de Parsing correspondiente a la gramática de la ecuación (3.5)

	a	b	c	d	#
S	(aS,1)	(bA,2)			
A			(c,3)	(dA,4)	
a	pop				
b		pop			
c			pop		
d				pop	
#					aceptar

La localidad S,a tiene (aS,1) para indicar que se debe utilizar la regla de producción número uno y meter en el stack la S y luego la a. La diagonal de la submatriz de terminales contra terminales siempre tiene pop puesto que si el símbolo leido y el que esta en el tope del stack son el mismo, entonces se debe de avanzar en la entrada y quitarlo del tope del stack. Las entradas en blanco de la tabla indican que se debe marcar error de sintaxis.

Las gramáticas LL(1) pueden tener reglas de producción que comienzen con un no-terminal, en ese caso se puede utilizar la función (3.6):

$$M(A,a) = \begin{cases} pop & si \quad A = a \quad \forall a \in V_T \\ acepta & si \quad A = \# \quad y \quad a = \# \\ (\beta, i) & si \quad a \in PRIMERO(\beta) \quad y \\ & A \rightarrow \beta \quad es\ la\ i-ésima\ producción \\ error & en\ cualquier\ otro\ caso \end{cases} \tag{3.6}$$

La función $PRIMERO(w)$ determina el conjunto de terminales que son el primer caracter de cadenas que pueden ser derivadas a partir de w. Es evidente que si dicha cadena conienza con un terminal, entonces ese conjunto es de un solo elemento, precísamente el terminal con el que comienza w.

Si $X \rightarrow Y_1 Y_2 ... Y_k$ es una regla de producción, se agrega a PRIMERO(X) todo lo que esté en PRIMERO(Y_i) siempre y cuando ε esté en PRIMERO(Y_j) $\forall 1 \leq j < i$, es decir, que $Y_j \overset{*}{\Rightarrow} \varepsilon \quad \forall 1 \leq j < i$. Si además $Y_i \overset{*}{\Rightarrow} \varepsilon$ ($\varepsilon \in PRIMERO(Y_i)$, se agrega a PRIMERO(X) todo lo que esté en PRIMERO(Y_{i+1}) y así sucesívamente.

Se puede entonces redefinir las gramáticas LL(1) como aquellas que tienen la forma:

$A \rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$ Tal que $PRIMERO(\alpha_i) \cap PRIMERO(\alpha_j) = \emptyset \quad \forall i \neq j$

La restricción impuesta se debe a que el parser LL(1) es determinístico y no debe tener varias alternativas sino decidir con un solo token de lookahead.

Ejemplo: Reconozcamos con un parser LL(1) la cadena:

(a,a):=(a,a)#
Mediante la gramática cuyas reglas de producción son:
0. S'→ S#

1. S→LB

2. B→;S;L

3. B→:=L

4. L→(EJ

5. J→,EJ

6. J→)

7. E→a

8. E→L

La Tabla 3.2 muestra la tabla de parsing para esta gramática:

Tabla 3.2: Tabla de Parsing para la gramática del Ejemplo

	a	:=	(	)	,	;	#
S			(LB,1)				
B		(:=,3)				(;S;L,2)	
L			((EJ,4)				
J				(),6)	(,EJ,5)		
E	(a,7)		(L,8)				
a	pop						
:=		pop					
(			pop				
)				pop			
,					pop		
;						pop	
#							Aceptar

Y el trazado del parsing de la cadena “(a,a):=(a,a)#” es el siguiente:

Entrada	Pila	Salida
(a,a):=(a,a)#	S#	
(a,a):=(a,a)#	LB#	1
(a,a):=(a,a)#	(EJB#	1,4
a,a):=(a,a)#	EJB#	1,4
a,a):=(a,a)#	aJB#	1,4,7
,a):=(a,a)#	JB#	1,4,7
,a):=(a,a)#	,EJB#	1,4,7,5
a):=(a,a)#	EJB#	1,4,7,5
a):=(a,a)#	aJB#	1,4,7,5,7
):=(a,a)#	JB#	1,4,7,5,7
):=(a,a)#	)B#	1,4,7,5,7,6
:=(a,a)#	B#	1,4,7,5,7,6
:=(a,a)#	:=L#	1,4,7,5,7,6,3
(a,a)#	L#	1,4,7,5,7,6,3
(a,a)#	(EJ#	1,4,7,5,7,6,3,4
a,a)#	EJ#	1,4,7,5,7,6,3,4
a,a)#	aJ#	1,4,7,5,7,6,3,4,7
,a)#	J#	1,4,7,5,7,6,3,4,7
,a)#	,EJ#	1,4,7,5,7,6,3,4,7,5
a)#	EJ#	1,4,7,5,7,6,3,4,7,5
a)#	aJ#	1,4,7,5,7,6,3,4,7,5,7
)#	J#	1,4,7,5,7,6,3,4,7,5,7
)#	)#	1,4,7,5,7,6,3,4,7,5,7,6
#	#	1,4,7,5,7,6,3,4,7,5,7,6
		1,4,7,5,7,6,3,4,7,5,7,6

Comenzamos leyendo '(' de la entrada y con 'S' en el tope de la pila, sin embargo no contamos con una producción S→(.... pero sí con la producción 1 (S→LB) y en vista de que PRIMERO(LB)= (, entonces seleccionamos esta como la primera producción a utilizar. Para el resto del ejemplo se procede como en el ejemplo de gramaticas-S.

Una gramática LL(1) también puede tener producciones ε, en cuyo caso la función M se convierte en:

$$M(A,a) = \begin{cases} pop & si \quad A = a \quad \forall a \in V_T \\ acepta & si \quad A = \# \quad y \quad a = \# \\ (\alpha, i) & si \quad a \in PRIMERO(\alpha) \quad y \\ & A \rightarrow \alpha \quad es\ la\ i-esima\ produccion \\ (\alpha, i) & si \quad a \in SIGUIENTE(A) \quad y \\ & A \rightarrow \alpha \quad es\ la\ i-esima\ produccion \\ & y \quad A \quad es\ nulificable \\ error & en\ cualquier\ otro\ caso \end{cases} \tag{3.7}$$

Donde: SIGUIENTE(A) es una función que calcula el conjunto de terminales que en alguna frase derivada de S estarían a la derecha de A, es decir, "en seguida" de A. S es el símbolo inicial de la gramática.

Para calcular SIGUIENTE(A) para todos los no terminales A, se aplican las siguientes reglas hasta que no se pueda agregar nada mas a ningún conjunto siguiente:

1. Póngase # en SIGUIENTE(S), donde S es el símbolo inicial y # es el delimitador derecho de la entrada.

2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que esté en PRIMERO(β) se excepto ε se pone en siguiente(B)

3. Si hay una producción $A \rightarrow \alpha B$ o una producción $A \rightarrow \alpha B \beta$, donde PRIMERO($\beta$) contenga ε, entonces todo lo que esté en SIGUIENTE(A) se pone en SIGUIENTE(B).

Ej. La cadena ccd# es reconocida por el parser LL(1) utilizando la siguiente gramática:

$$\begin{array}{ll} 0. & S \rightarrow A\# \\ 1. & A \rightarrow Bb \\ 2. & A \rightarrow Cd \\ 3. & B \rightarrow aB \\ 4. & B \rightarrow \varepsilon \\ 5. & C \rightarrow cC \\ 6. & C \rightarrow \varepsilon \end{array} \tag{3.8}$$

Al principio el parser procede como en el ejemplo anterior, selecciona la regla de producción 2, luego la 5 y luego la 5 otra vez, el trazado de intantáneas hasta ahí es:

(ccd#,A#,ε)⊢(ccd#,Cd#,2)⊢(ccd#,cCd#,25)⊢

(cd#,Cd#,25)⊢(cd#,cCd#,255)⊢(d#,Cd#,255)⊢

Entonces, leyendo 'd' de la entrada y con C en el tope de la pila se debe seleccionar la regla 5 o la 6 pero 'd' no está ni en PRIMERO(cC) ni en PRIMERO(ε), sin embargo como C es nulificable y 'd' está en SIGUIENTE(C)=d se debe elegir la regla 6, el resto del parsing es evidentemente:

(d#,d#,2556)⊢(#,#,2556)⊢(ε,ε,2556)

La Tabla 3.3 muestra la Tabla de parsing para esta gramática.

En general, podemos definir a las gramáticas LL(1) como aquellas en las que para todas las reglas de producción con un mismo no terminal del lado izquierdo.

$$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n \tag{3.9}$$

Se cumple que:

$$PRIMERO(\alpha_i) \cap PRIMERO(\alpha_j) = \emptyset \quad \forall i \neq j \tag{3.10}$$

Tabla 3.3: Tabla de Parsing para la gramática del ejemplo

	a	b	c	d	#
A	(Bb,1)	(Bb,1)	(Cd,2)	(Cd,2)	
B	(aB,3)	(ε ,4)			
C			(cC,5)	(ε,6)	
a	pop				
b		pop			
c			pop		
d				pop	
#					Aceptar

Además, si $\alpha_i \stackrel{*}{\Rightarrow} \varepsilon$, es decir, si A es nulificable entonces se cumple también que:

$$SIGUIENTE(A) \cap PRIMERO(\alpha_j) = \emptyset \quad \forall i \neq j \tag{3.11}$$

Manejo de errores

El parser LL(1) detecta un error de sintaxis cuando llega a una localidad vacia de la tabla de parsing. Indicar "Error en la linea ..." y abortar puede ser un comportamiento adecuado para un parser teórico abstracto, sin embargo, para un compilador real tal reacción es inaceptable. Un compilador real debe reportar el tipo de error e intentar continuar con el análisis sintáctico como si no hubiera encontrado ningún error, tal vez mas adelante encuentre otros errores que tambien denerá reportar de manera que cuando el programador modifique el código fuente, corrija el mayor numero de errores que le sea posible antes de recompilar.

Los parsers LL(1) poseen la característica del prefijo válido, lo cual significa que si el parser no detecta ningun error en la primera porción $a_1a_2...a_{k-1}$ de la entrada $a_1a_2...a_n$, debe existir una secuencia de símbolos $a_ka_{k+1}...a_m$ tal que $a_1a_2...a_m$ es una cadena válida del lenguaje. La propiedad del prefijo válido implica que el parser detecta el error lo antes posible en el proceso de barrido de izquierda a derecha de la entrada. Esta propiedad también elimina la necesidad de borrar y/o insertar símbolos en el stack en el proceso de recuperación del error. Cuando el parser detecta un error en el símbolo a_k,

el parser puede modificar los símbolos $a_k a_{k+1} ... a_m$ y continuar con el proceso de análisis sintáctico.

Ejemplo: Para la siguiente gramática:

E'→E#
E→TA
A→+TA
A→ ε
T→a
T→(E)

La tabla de parsing se muestra en la Tabla 3.4.

Tabla 3.4: Tabla de Parsing para la gramática del ejemplo

	a	(	)	+	#
E	(TA,1)	(TA,1)			
A			(ε ,3)	(+TA,2)	(ε ,3)
T	(a,4)	((E),5)			
a	pop				
(		pop			
)			pop		
+				pop	
#					Aceptar

Sea la cadena de entrada:

$a)\#$

Un trazado del análisis efectuado por el parser se muestra en la Tabla 3.5.

Tabla 3.5: Trazado para la entrada a)#

Paso	Entrada	Stack	Salida
1	a)#	E#	
2	a)#	TA#	1
3	a)#	aA#	14
4	)#	A#	14
5	)#	#	143

Se detectó un error de sintaxis en el momento en que se estaba leyendo ')' de la entrada y había # en el tope de la pila puesto que la tabla de parsing está en blanco en la localidad que corresponde a estos dos datos, de manera que por la propiedad del prefijo válido, se concluye que el primer caracter erroneo es el ')', sin embargo, del paso 4 al 5 se eliminó el no-terminal 'A', el cual tenía la información de que se esperaba una cadena del tipo "+T+T...+T", información útil para poder continuar el proceso de análisis sintáctico luego de recuperarse del error. Este problema de símbolos no terminales eliminados prematuramente del stack ocurre solo cuando los símbolos son nulificables, para prevenirlo eficientemente podemos implementar el stack como un arreglo con dos índices, al sacar un símbolo del stack solamente movemos el índice superior, pero si en el proceso de recuperación de un error de sintaxis detectamos un símbolo nulificable que acaba de ser eliminado del stack, simplemente regresamos el índice superior una posición y se recupera el símbolo anolado. De esta manera retenemos la mayor cantidad de información contextual posible.

El reporte del error y método de recuperación varia dependiendo del error mismo. Un método muy usado consiste en numerar las localidades vacias de la tabla de parsing y diseñar una función para cada una de estas, la cual será llamada cuando el autómata utilice una de estas entradas. En el ejemplo anterior son 28 entradas y se muestran en la Tabla 3.6.

Tabla 3.6: Tabla de Parsing con 28 índices de errores

	a	(	)	+	#
E	(TA,1)	(TA,1)	1	2	3
A	4	5	(ε ,3)	(+TA,2)	(ε ,3)
T	(a,4)	((E),5)	6	7	8
a	pop	9	10	11	12
(	13	pop	14	15	16
)	17	18	pop	19	20
+	21	22	23	pop	24
#	25	26	27	28	Aceptar

En el error numerado como 1, por ejemplo, observamos que el no-terminal E será generado por cualquiera de las siguientes dos reglas de la gramática:

E'→E#
T→(E)

Y en vista de que el error se presentó al leer ')', el diseñador del error puede decidir enviar el siguiente mensaje:

“FALTA EXPRESION ENTRE PARENTESIS O SOBRA ')' EN LA LINEA ...”

Tambien debe ejecutar las siguentes acciones antes de continuar la compilación:

1. Si el error consiste en que sobra el caracter ')', ignorarlo de la entrada

2. Si el error es que falta la expresión entre paréntesis, borrar del stack los símbolos E y ')'.

3.2. Análisis Sintáctico Ascendente

Un análizador sintáctico ascendente contruye el árbol de sintaxis de la cadena de entrada de abajo hacia arriba, es decir, de las hojas (tokens) hacia la raíz (El símbolo inicial de la gramática). Este proceso es equivalente al de una derivación por la derecha en orden inverso, es decir al de seleccionar las reglas de producción a aplicar de manera que la primera que se seleccionara fuera la última que se usaría al hacer dicha derivación. Por ejemplo, mediante la gramática:

1. E→E+T
2. E→T
3. T→T*F
4. T→F
5. F→(E)
6. F→id

Se puede derivar la cadena id*(id+id) por la derecha de la siguiente manera:

$$E \Rightarrow T \Rightarrow T{*}F \Rightarrow T{*}(E) \Rightarrow T{*}(E{+}T) \Rightarrow T{*}(E{+}F) \Rightarrow T{*}(E{+}id) \Rightarrow$$

$$\Rightarrow T{*}(T{+}id) \Rightarrow T{*}(F{+}id) \Rightarrow T{*}(id{+}id) \Rightarrow F{*}(id{+}id) \Rightarrow id{*}(id{+}id)$$

Para realizar tal derivación se utilizaron las reglas 2,3,5,1,4,6,2,4,6,2,4,6 en ese orden. Sin embargo, en vista de que un parser ascendente parte de la última cadena (la entrada) hacia la primera (formada por solamente el simbolo inicial) debe seleccionar esas mismas reglas de producción pero en orden inverso, de manera que la primera regla que debe seleccionar es la 6.

Existen diversos parsers ascendentes, algunos utilizan información acerca de la jerarquía de los tokens para decidir cual regla de producción utilizar. Otros utilizan la técnica del desplazamiento y reducción (shit/reduce), en la cual avanzan en el barrido de la entrada (desplazamiento) o utilizan alguna regla de producción para avanzar en el proceso de derivacion por la derecha. Los parsers LR son de estos últimos, estos en particular barren la entrada de

izquierda a derecha, por lo cual debe detectar cuando la parte leida coincide con la parte derecha de alguna regla de producción, en el ejemplo anterior, al principio lee id y lo reconoce como la parte derecha de la regla 6 y por eso se aplica primero dicha regla. A estas cadenas que coinciden con la parte derecha de alguna regla de producción se les llama mangos, asideras o agarraderas (handle en inglés).

3.2.1. Parsers LR

Los parsers LR(1) pueden de manera determinista decidir cual regla utilizar con solo un token de look ahead. Son autómatas finitos con un stack para recordar de que manera han llegado a un determinado estado. Al ir barriendo la entrada de izquierda a derecha, van consultando en cada paso una tabla de ACCIONES muy parecida a las tablas de parsing LL(1) donde se indica que debe hacer el autómata: shift y cambiar de estado o reducir mediante determinada regla de producción, aceptar la entrada o marcar error. También utilizan una tabla GOTO que tiene la información de a que estado cambiarse cuando hace una reducción. En la Figura 3.1 se muestra un parser LR(1) conceptualmente:

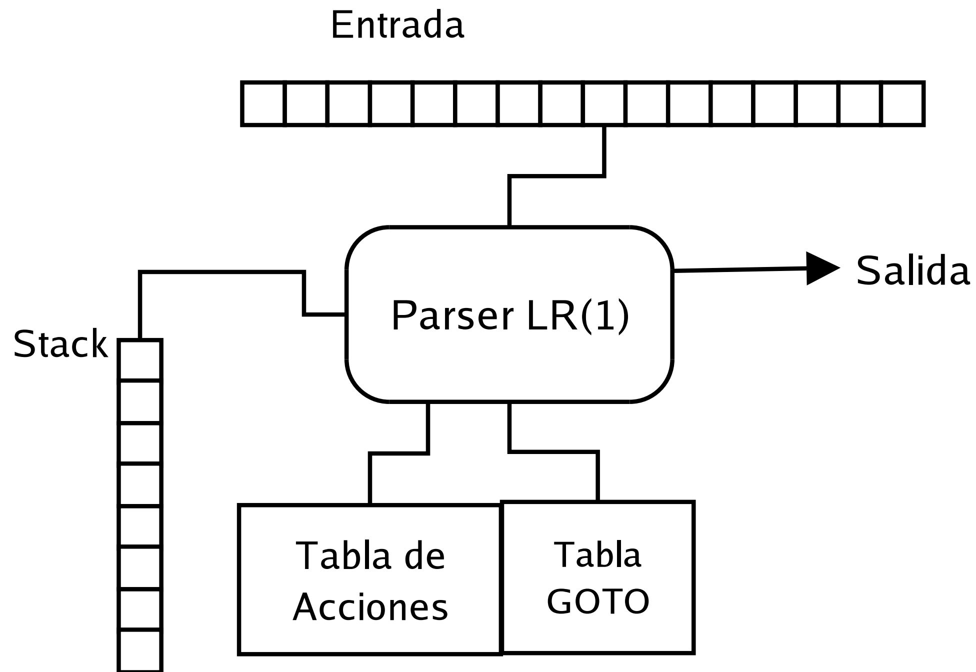

Figura 3.1: Parser LR(1)

Para entender el funcionamiento de este parser supondremos que contamos con las tablas Acción y Goto en el siguiente ejemplo:

Para la gramática:

0.-E'→E#
1.-E→E+T
2.-E→T
3.-T→TF
4.-T→F
5.-F→F*
6.-F→a
7.-F→b

Donde la regla 0 ha sido agregada como en los parsers LL. La Tabla Acción y GOTO se muestra en la Tabla 3.7.

Para entender como funciona el parser analicemos la entrada a+b*a#, a la cual se le ha agregado caracter # para marcar el final de la entrada.

Inicialmente el autómata se encuentra en el estado 0 y lee de la entrada

Tabla 3.7: Tabla Acción-GOTO

	ACCIÓN						GOTO		
	+	*	a	b	#	ε	E	T	F
0			s4	s5			1	2	3
1	s7				s6				
2	r2		s4	s5	r2				8
3	r4	s9	r4	r4	r4				
4	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7				
6						aceptar			
7			s4	s5				10	3
8	r3	s8	r3	r3	r3				
9	r5	r5	r5	r5	r5				
10	r1		s4	s5	r1				8

el caracter 'a'. De acuerdo a la tabla de acciones debe de hacer shift (avanzar en la entrada) y pasar al estado 4. Asi es que se meten al stack la 'a' y el 4. Ahora el autómata se encuentra en el estado 4 y está leyendo el caracter '+'. La tabla de acciones dicta que se debe hacer una reducción por la regla de producción 6 (F→a). Por tanto se sacan dos elementos del stack (El doble de la longitud de la cadena de la parte derecha de la regla de producción utilizada) descubriendo de nuevo al estado 0 que queda en el tope del stack. En vista de que esa regla de producción 6 tiene F del lado izquierdo entramos a la tabla GOTO con F y estado 0. Así pues, nos cambiamos al estado 3 y metemos la F y el 3 al stack. El resto del análisis se puede seguir del trazado completo que se muestra en la Tabla 3.2.1.

Como se aprecia, en el stack hay una cadena del tipo (SX)+ Donde S es un estado y X es un símbolo variable o terminal. El estado en que se encuentra el autómata es el que está en el tope de la pila. Los símbolos se metieron al stack solo para hacer evidente el hecho de que cuando en la parte superior del stack se encuentra una agarradera se debe hacer una reducción, por ejemplo, la penúltima cadena del stack tiene 9T6+1E0# sin estados se convierte en T+E# en la parte superior se encuentra T+E que en orden inverso es E+T, la parte derecha de la regla de produccion 1 (E→E+T), por

Tabla 3.8: Trazado de la entrada a+b*a#

Stack	Entrada	Acción
0#	a+b*a#	s4
4a0#	+b*a#	r6
3F0#	+b*a#	r4
2T0#	+b*a#	r2
1E0#	+b*a#	s7
7+1E0#	b*a#	s5
5b7+1E0#	*a#	r7
3F7+1E0#	*a#	s9
9*3F7+1E0#	a#	r5
3F7+1E0#	a#	r4
3F7+1E0#	a#	s4
4a10T—7+1E0#	#	r6
8F10T7+1E0#	#	r3
10T7+1E0#	#	r1
1E0#	#	s6
6#0#	ε	aceptar

eso la acción es r1 se debe sacar T+E con todo y estados y meter E al stack. Es claro el por qué se sacan del stack un número de elementos igual al doble de la longitud de la cadena derecha de la regla de producción utilizada ya que por cada símbolo hay un estado. Si se opta por no almacenar los símbolos sino solamante los estados no se deberá sacar el doble de la longitud sino la longitud solamente.

El problema a resolver ahora es el como contruir las tablas Acción y goto que utiliza este parser. Para ello nos auxiliaremos del concepto de elemento, un elemento es una regla de la gramática donde la parte derecha de la misma está dividida en dos partes, la primera parte es la que ya fué verificada con la entrada y la segunda es la que se espera que se verifique con la parte de entrada que está por leerse. Si la marca que separa estas dos partes es un punto, entonces los elementos tienen la forma:

$$A \rightarrow \alpha.\beta \tag{3.12}$$

El conjunto inicial de elementos de la gramática se forma tomando las reglas de producción que tengan el símbolo inicial de la gramática del lado izquierdo y colocándoles un punto al inicio de la parte derecha de estas, para el ejemplo anterior serían:

E→.E+T
E→.T

Luego se obtiene la cerradura de este conjunto, esto se logra agregando por cada elemento de la forma $A \rightarrow \alpha.B\beta$ los elementos $B \rightarrow .\gamma$ por cada regla de producción $B \rightarrow \gamma$ de la gramática. En este ejemplo debido al elemento E→.T debemos de agregar los elementos T→.TF y T→.F. En vista de que el elemento T→.F ya forma parte del conjunto se deben agregar también los elementos F→.F*, F→.a y F→.b el proceso debe continuar hasta que ya no sea posible agregar nuevos elementos a este conjunto. El conjunto de elementos inicial queda entonces como:

C_0={
E'→.E# ,
E→.E+T ,
E→.T ,
T→.TF ,
T→.F ,
F→.F* ,
F→.a ,
F→.b
}

Este conjunto denota al estado inicial del autómata, para determinar los estados a los que pasa el autómata a partir de este estado al leer E se buscan los elementos que tengan E ensequida del punto (E'→.E# y E→.E+T) y se cambian por los mismos elementos con el punto desplazado una posición a la derecha (E'→E.# y E→E.+T) y despues se obtiene la cerradura de este conjunto, en este caso el conjunto es igual a su cerradura puesto que después del punto solo hay terminales. Por lo tanto, el estado al que pasa el autómata al leer E estando en el estado cero queda denotado por el conjunto de elementos:

C_1= {
E'→E.# ,
E→E.+T
}

De igual manera, al leer T estando en el estado 0, se pasa al estado 2, el cual queda denotado por el conjunto:

C_2={
E→T. ,
T→T.F ,
F→.F* ,
F→.a ,
F→.b
}

Si al obtener el obtener el conjunto de elementos que denota al estado al que pasa el autómata al leer algún símbolo X estando en determinado estado m se llega a un conjunto de elementos que ya existe, digamos el n, entonces no se agrega un estado nuevo sino que simplemente el autómata pasará del estado m al estado n al leer el símbolo X.

En la siguiente Tabla se indican todos los conjuntos de elementos:

Edo ant	Edo	Símbolo leído	Conjunto de elementos	Comentario
0	1	E	$\{E' \rightarrow E.\#$, $E \rightarrow E. + T\ \}$	
0	2	T	$\{E \rightarrow T.$, $T \rightarrow T.F$, $F \rightarrow .F*$, $F \rightarrow .a$, $F \rightarrow .b\ \}$	Estado inadecuado
0	3	F	$\{T \rightarrow F.$, $F \rightarrow F.*\ \}$	Estado inadecuado
0	4	a	$\{F \rightarrow a.\ \}$	
0	5	b	$\{F \rightarrow b.\ \}$	
1	6	#	$\{E' \rightarrow E\#\ \}$	
1	7	+	$\{E \rightarrow E + .T$, $T \rightarrow .TF$, $T \rightarrow .F$, $T \rightarrow .F*$, $F \rightarrow .a$, $F \rightarrow .b\ \}$	
2	8	F	$\{T \rightarrow TF.$, $F \rightarrow F.*\ \}$	Estado inadecuado
3	9	*	$\{F \rightarrow F * .\ \}$	
7	10	T	$\{E \rightarrow E + T.$, $T \rightarrow T.F$, $F \rightarrow .F*$, $F \rightarrow .a$, $F \rightarrow .b\ \}$	Estado inadecuado

Los estados como el 9 cuyo conjunto de elementos ({ F→F*. }) es solo de solo un elemento y este tiene el punto hasta el final se llaman estados reductores puesto que si se llega a un estado como este no se hara un desplazamiento sino una reducción. En este caso la reducción se haría mediante la regla 5 (F→F*) por razones obvias. Los estados como el 10 se llaman estados inconvenientes puesto que uno de los elementos del conjunto asociado tiene el punto hasta el final indicando que se puede hacer una reducción mediante una determinada regla de producción mientras que otros elementos no tienen el punto hasta el final indicando que se puede hacer un desplazamiento, es decir, existe un conflicto desplazamiento/reducción. Cuando estos estados no aparecen se dice que se trata de un parser LR(0). Cuando estos conflictos se pueden resolver con solo analizar el próximo token a leer, entonces se trata de un parser SLR(1).

De la gramática se obtienen los conjuntos:

SIGUIENTE(E) = { +, # }

SIGUIENTE(T) = { a, b, #, + }

SIGUIENTE(F) = { *, a, b, #, + }

Para el estado inadecuado 2, la reducción por la regla 2 (E→T) se haría si el siguiente token es '+' o '#' puesto que SIGUIENTE(E)=+,#, en cambio un desplazamiento se debe hacer si el siguiente token es 'a' o 'b'. Esto se sabe de los elementos F→.a y F→.b del conjunto asociado a este estado. En vista de que los conjuntos {+,#} y {a,b} son conjuntos disjuntos esta ambiguedad es solucionada si se conoce un token adelante. De la misma menera se procede en los otros estados inadecuados puesto que esta gramática es SLR(1). El autómata finito se muestra en la Figura 3.2.

Este procedimiento para construir las tablas ACCIÓN y GOTO no es muy conveniente realizarlo manualmente cuando se trata de la gramática de un lenguaje de programación real. Sin embargo, en vista de que el procedimiento es bastante mecánico, es relatívamente facil automatizarlo de manera que se le especifique la gramática a un programa que construya las tablas en cuestión.

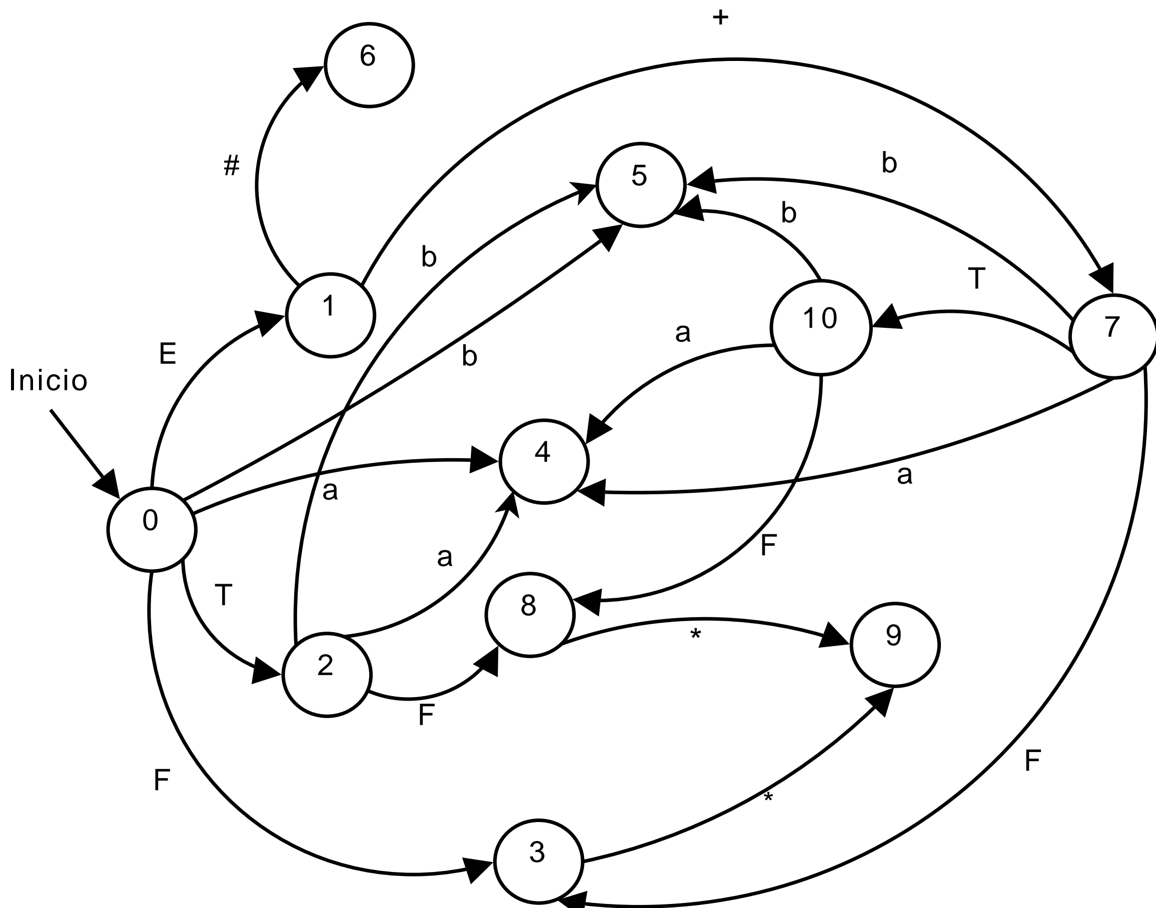

Figura 3.2: Automata Finito

3.3. El generador de analizadores sintácticos: yacc

El yacc es una utilería de UNIX que no solo construye las tablas definidas en la sección anterior sino al autómata mismo también, produce el código del autómata en lenguaje C.

El archivo fuente de yacc es una archivo con la extensión .y y que tiene el siguiente formato:

```
%{
declaraciones en C
%}
declaraciones de yacc
%%
reglas de la gramática
%%
funciones en C
```

La sección de declaraciones así como la de funciones de C son omitibles. Cada regla de la gramática tiene la forma A: BODY donde 'A' representa un no-terminal y BODY una secuencia de terminales y no-terminales. Los terminales pueden especificarse con literales encerrados entre comillas simples o por un nombre designado en la parte de declaraciones mediante la directiva %token.

El símbolo inicial de la gramática por default es el no-terminal de la primera regla de producción que aparezca después del primer %%. Sin embargo, puede especificarse explícitamente con la directiva %start.

El siguiente ejemplo de yacc reconoce expresiones artiméticas y las evalúa, se trata pues de una calculadora. En la parte de declaraciones de C, se define el tipo de datos de la pila de yacc, esto se hace definiendo la macro YYSTYPE, luego, en la parte de declaraciones de yacc se definen los tokens '+','-','*','/' y NUMBER. Este último es un tipo predefinido de yacc para los numeros enteros o reales. La primera regla de producción sirve para ignorar los retornos de carro, la segunda regla sirve para permitir un sinnúmero de expresiones aritméticas separadas por \n. La acción asociada es precísamente imprimir el resultado de la evaluación de una expresión, el cual es el valor de la variable expre que en esta regla lo representa $2 puesto que ocupa la segunda posición del lado derecho de la regla lista→ lista expre '\n'.

Las demás reglas son las de la gramática de un lenguaje de expresiones aritméticas en infijo y con paréntesis para la agrupación de expresiones. La evaluación se vá llevando a cabo a medida que se asciende por el arbol de sintáxis de la expresión aritmética en cuestión. A la variable del lado izquierdo de la regla de producción se le asigna un valor al que se puede acceder mediante $$ y las variables del lado derecho tienen un valor al cual se accede mediante $1, $2, $3, etc. dependiendo de su posición. Por ejemplo, en la regla de producción expre → expre '/' expre, los expre del lado derecho ocupan las posiciones 1 y 3, por eso, la acción asociada es la de dividir el valor de la

expresión a la izquierda del operador '/' (es decir, \$ 1) entre el valor de la expresión a la derecha del mismo (o sea \$ 3), el resultado es lo que regresa la expresión y debe almacenarse en \$\$. Algo similar se hace con las demás reglas de la grmática. El listado se muestra a continuación:

```
%{
#define YYSTYPE double  /* Tipo de la pila de datos de yacc */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%left UNARYMINUS
%%
lista:
     | lista '\n'
     | lista expre '\n' { printf("\t%lf\n",$2); }
     ;
expre: NUMBER           { $$ = $1; }
     | '-' expre %prec UNARYMINUS {$$ = -$2; }
     | expre '+' expre  { $$ = $1 + $3; }
     | expre '-' expre  { $$ = $1 - $3; }
     | expre '*' expre  { $$ = $1 * $3; }
     | expre '/' expre  { $$ = $1 / $3; }
     | '(' expre ')'    { $$ = $2; }
     ;
%%
#include <stdio.h> #include <ctype.h>

char *nom_prog; int num_lineas=1;

main(int argc,char *argv[]) {
    nom_prog=argv[0];
    yyparse();
}

yylex() { int c;
   while ((c=getchar())==' '||c=='\t');
```

```
   if (c==EOF) return 0;
   if (c=='.'||isdigit(c)) {
       ungetc(c,stdin);
       scanf("%lf",&yylval);
       printf("Token numero: %lf\n",yylval);
       return NUMBER;
   }
   if (c=='\n') num_lineas++;
   printf("Token caracter: %c\n",c);
   return c;
}

yyerror(char *s) {
   warning(s,(char *)0);
}

warning(char *s,char *t) {
   fprintf(stderr,"%s: %s",nom_prog,s);
   if (t) fprintf(stderr," %s",t);
   fprintf(stderr," cerca de la linea %d\n",num_lineas);
}
```

En la parte de funciones de C se debe incluir al menos la función main(), la cual debe llamar a la función yyparse() que genera yacc. La función yyparse(). La función yylex() regresa el siguiente token identificado en la entrada y en caso de este token sea NUMBER, su valor en la variable yylval. La función yylex() puede ser creada con la utilería lex de UNIX, sin embargo, en este caso se trata de una función muy simple por lo cual se implementó manualmente. La función ignora espacios en blanco y tabuladores, los retornos de carro solo sirven para incrementar el contador de líneas num_lineas. Cuando encuentra un caracter punto o un dígito decimal, determina que se va a leer un token del tipo NUMBER, pero debe primero desleer el caracter leido para poder leer el valor numérico completo. Cualquier otro caracter simplemente lo lee y lo pasa al analizador sintáctico para us uso, de manera que en caso de haber un error será este último quien lo detecte. De hecho, si hubiese un error, yyparse() llamará a la función yyerror() para que lo trate, en este caso la función simplemente llama a la función warning() para que indique la naturaleza del error puesto que no se recuperará del error. Existe la manera

en yacc de recuperarse del error, consulte [4], [5].

Bibliografía

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compiladores. Principios, Técnicas y Herramientas.* Addison Wesley Iberoamericana, 1990.

[2] J.-P. Tremblay and P. G. Sorenson, *The Theory and Practice of Compiler Writing.* Mac Graw Hill, 1985.

[3] A. Holub, *Compiler Design in C.* Prentice Hall, 1990.

[4] J. R. Mason and D. Brown, *lex and yacc.* O'Reilly Associates Inc., 1990.

[5] B. W. Kernighan and R. Pike, *El Entorno de Programación UNIX.* Prentice Hall, 1987.