Compiladores
Facultad de Ciencias UNAM
Análisis Sintáctico: *Parser LR: errores y generadores* [*]

Lourdes Del Carmen González Huesca [**]

9 de abril de 2024

Resumen

En este punto del curso se analizan autómatas que nos ayudan a reconocer si un programa puede ser generado o no a partir de una gramática dada. Dichos autómatas utilizan estrategias de reconocimiento con base en las características de la gramática. Durante este proceso de reconocimiento existe la posibilidad de que nuestro programa fuente o cadena contenga errores. Esta nota está enfocada en mostrar algunas tácticas para el manejo de los errores que se puedan encontrar en la cadena durante su análisis.

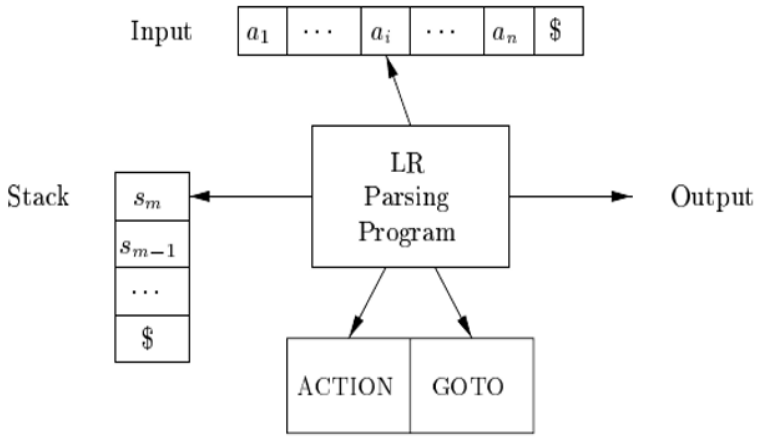
Por otro lado existen generadores de analizadores sintácticos, que son programas que dada una gramática y otros datos son capaces de generar un autómata que decida si una cadena pertenece al lenguaje generado por la gramática o no. Durante esta nota se da un panorama del generador `Yacc` (*Yet another compiler compiler*).

Términos clave:

Es: Parser LR, Parser Shift-Reduce, Manejo de errores en LR, Modo pánico, Recuperación a nivel de frase, `Yacc`.

En: LR Parser, Shift-Reduce Parsing, Error recovery in LR Parsing, Panic-mode error recovery, Phrase-level recovery, `Yacc`.

Revisemos en general un analizador LR y sus interacciones:



[*] Los ejemplos e imágenes están tomados del libro "Compilers, Principles, Techniques and Tools" [1]

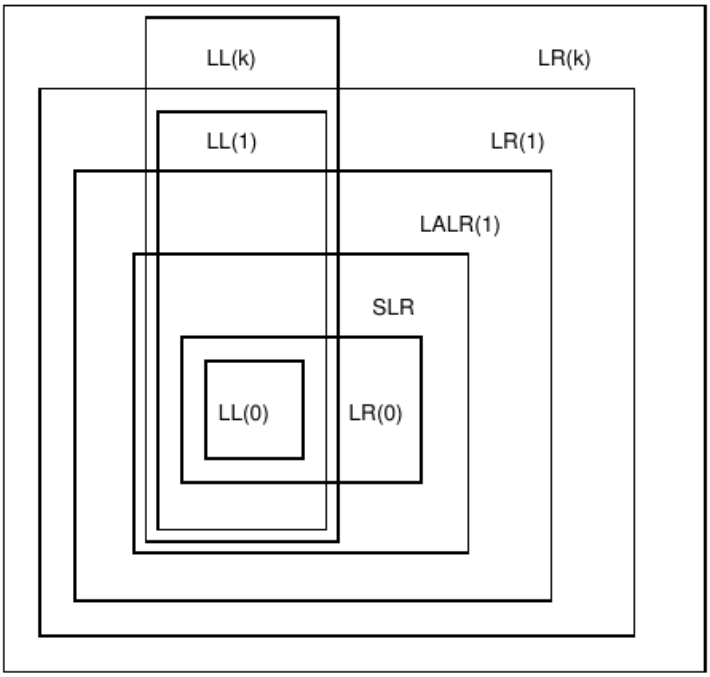
[**] Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

Hay al menos dos pilas en el analizador, una que lleva los estados para reconstruir una expresión y otra que almacena los símbolos procesados (operación shift) o los símbolos de las reducciones. Estos últimos se representan como el contenido en la pila *output*.
La tabla de parsing contiene las acciones a realizar por el analizador y las transiciones. Esta tabla fue construida con un autómata finito cuya finalidad es llevar control de los estados del parser y así identificar las producciones que se pueden reducir o los símbolos que se pueden agregar a la pila.

En la tabla de parsing, las entradas son parejas estado–símbolo:

- estado – símbolos terminales que generan acciones (shift o reduce)
- estado – símbolo no-terminal determinan las transiciones entre estados

La siguiente figura agrupa las clases de gramáticas:



De esta clasificación se desprenden algunas propiedades de los analizadores LR:

- Toda gramática SLR es no-ambigua pero no toda gramática no-ambigua es de la clase SLR.
- Toda gramática que tiene un analizador SLR es una gramática de la clase LR(1) pero no viceversa.
- Toda gramática de la LALR es una gramática de la clase LR(1) pero no viceversa.

Manejo de errores en LR

Los errores en un analizador LR son las entradas de la tabla de parsing en donde no es posible realizar una acción para un item y un símbolo, es decir, está vacía esa entrada y no es posible continuar con el análisis de la cadena de entrada en el estado actual. Por lo tanto, los errores sólo serán producidos por las acciones y no por las transiciones en la parte de la tabla para la función GoTo.
Para los tipos de analizadores que hemos revisado sucede lo siguiente:

- Para LR(0), nunca habrá reducciones si hay errores.
- Para SLR y LALR se pueden hacer reducciones pero nunca habrá un shift de un símbolo erróneo.

La finalidad del compilador es que al menos la etapa del análisis sintáctico termine, por lo tanto para asegurar que el parsing se complete se puede ignorar un parte o alterar la cadena de entrada, así como los elementos almacenados en la pila. Veamos dos formas de manejo de errores.

Modo pánico Esta forma de manejo ignora la posible subcadena que contiene un error y continúa con el análisis, reportando el error. Cuando se llega a una entrada de la tabla con error, se conoce el símbolo no-terminal A que llevó a ese estado y también se tienen los símbolos en la pila. Entonces se procede como sigue:

- Revisar la pila de estados para encontrar un estado I en donde la transición GoTo ha usado el símbolo no-terminal A.
- Sacar de la pila cero o más símbolos hasta encontrar un símbolo b que sigue a A.
- Se guarda el estado resultante de GoTo(I, A) y se continúa con el análisis.

Recuperación de nivel de frase Esta opción debe examinar cada entrada vacía de la tabla de parsing, es decir, en donde haya un error y decidir el error más común que pueda ser generado por el o la programadora.

Estos errores se identifican tomando en cuenta el uso del lenguaje definido por la gramática y se manejan al definir procedimientos especiales que usualmente modifican la pila o la cadena de entrada. De esta forma, estos procedimientos complementan la tabla con otras acciones:

- agregar o eliminar símbolos de la pila o de la cadena de entrada o de ambos, por ejemplo vaciar la pila si ya se ha procesado toda la cadena de entrada;
- alterar o mover símbolos de la cadena de entrada;
- retirar un estado de la pila.

Estas acciones alternativas deben asegurar que el analizador no se convierta en un ciclo infinito, esto se puede prevenir al eliminar sólo un símbolo de la cadena de entrada o al evitar sacar un estado que pase por un símbolo no-terminal ya que esto eliminaría una construcción o subárbol que ha sido generado sin errores.

Ejemplo 1 (Expresiones aritméticas)**.** Considera la siguiente gramática

$$E \to E + E \mid E * E \mid (E) \mid id$$

La figura incluye a) la tabla de parsing construida mediante el método LR y resolviendo los conflictos de manera manual ya que la gramática es ambigua, y b) la tabla que incluye el manejo de errores:

State	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

State	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

1. Siguiendo el método SLR hay un conflicto en el estado I_7 dado que no se sabe si se debe hacer una reducción usando la regla $E \to E + E$ o un shift de un operador + ó $*$ dado que ambos pertenecen a FOLLOW(E).
   Este y otros errores similares se resuelven mediante la precedencia y asociatividad de los operadores, lo cual implica un retroceso en el diseño del parser al tener que modificar las reglas de la gramática. Debido a esto es que es deseable que la gramática sea lo menos ambigua posible.

2. Al analizar la tabla del punto anterior se pueden llenar algunas entradas vacías de la tabla al posponer la detección del error, por ejemplo en los estados I_7, I_8, I_9 y columnas para **id** y (se utilizan reducciones desables pero nunca se van a concretar.
   El resto de las entradas pueden tener una de las subrutinas para manejo de error obtenidas al analizar los errores más comúnes en estos casos.
   Es también que en este análisis se decida optar por ignorar símbolos de entrada y realizar una acción, usualmente reduce, para continuar con el proceso de parsing.

   e1: Esta subrutina es llamada en los estados I_0, I_2, I_4 e I_5 que son los estados que esperan leer el inicio de una operación, ya sea un *id* o un paréntesis izquierdo. Pero la cadena de entrada tiene otro símbolo terminal o el fin de cadena de entrada.
   El error se maneja al agregar el estado I_3 y devolver un mensaje *missing operand*.

   e2: Se llama en los estados I_0, I_2, I_4 e I_5 donde se encuentra un paréntesis derecho.
   El error se maneja al remover el paréntesis derecho de la cadena de entrada y devolver el mensaje *unbalanced right parenthesis*.

   e3: Esta subrutina es llamada en los estados I_1 e I_6 que son los estados que esperan leer un operador y se encuentran con un *id* o un paréntesis derecho.
   El error se maneja al agregar el estado I_4 y devolver un mensaje *missing operator*.

   e4: Se llama en el estado I_6 cuando se encuentra el símbolo de fin de entrada o cadena. El error se maneja al agregar el estado I_9 y devolver un mensaje *missing right parenthesis*.

La siguiente tabla es el análisis de la cadena $id +)$

STACK	SYMBOLS	INPUT	ACTION
0		**id** +) $	
0 3	**id**	+) $	
0 1	E	+) $	
0 1 4	$E +$	) $	"unbalanced right parenthesis" e2 removes right parenthesis
0 1 4	$E +$	$	"missing operand" e1 pushes state 3 onto stack
0 1 4 3	$E +$ **id**	$	
0 1 4 7	$E +$	$	
0 1	$E +$	$	

Generadores de analizadores sintácticos

Los generadores son herramientas que facilitan la construcción de los analizadores sintácticos, los más usuales son los que generan analizadores tipo LALR. En esta parte veremos el analizador `Yacc` *Yet another compiler compiler* que data de los años 1970's. Actualmente existen muchas versiones de `Yacc`, por ejemplo `bison`; y es muy común que se utilice junto con `Lex` (para referencia, véase: `http://ftp.mozgan.me/Compiler_Manuals/LexAndYaccTutorial.pdf`). Este analizador está incluido en los sistemas UNIX bajo el comando `yacc`. A continuación se describe *grosso modo* su funcionamiento:

- se debe especificar el lenguaje en un archivo extensión `.y` con la estructura: declaraciones, reglas y subrutinas; en ese orden y separados por los símbolos %%

  ```
  %{
  Parte 1 : declaraciones para el compilador C
  %}
  Parte 2 : declaraciones para yacc
  %%
  Parte 3 : esquemas de traduccion
  (producciones + acciones semanticas)
  %%
  Parte 4 : funciones C complementarias
  ```

  Las declaraciones para `yacc` incluyen las definiciones de los símbolos terminales y no-terminales así como la definición de prioridades y tipo de asociatividad de los operadores en la gramática. Los esquemas de traducción son las producciones de la gramática y las acciones a realizar en cada caso.

- ejecutar el comando `yacc` con el archivo `.y` que devolverá un archivo con extensión `tab.c` que contiene una representación del analizador LALR escrito en C junto con algunas rutinas;

- al compilar el último archivo junto con una biblioteca (usualmente `ly`) se obtendrá un archivo con la transformación de la especificación.

Ejemplo 2 (Expresiones aritméticas)**.** Considera la gramática siguiente

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \mid num$$

La especificación está dada por el siguiente archivo:

```
% {
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
% }
% token NUMBER

% left '+' '-'
% left '*' '/'
% right UMINUS
%%
lines : lines expr '\n' { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr    { $$ = $1 + $3; }
     | expr '-' expr    { $$ = $1 - $3; }
     | expr '*' expr    { $$ = $1 * $3; }
     | expr '/' expr    { $$ = $1 / $3; }
     | '(' expr ')'     { $$ = $2; }
     | '-' expr %prec UMINUS { $$ = - $2; }
     | NUMBER
     ;
%%
int main(void){
if(yyparse() == 0)
printf("Analisis completo\n");
```

```
}

int yyerror(void)
{
fprintf(stderr, "error de sintaxis\n");
return 1;
}
```

La última parte puede incluir el resultado de un generador automático de lexer, por ejemplo `Flex`, mediante `#include "lex.yy.c"` o inclusive definir el propio lexer como se ejemplifica en el código:

```
yylex() {
    int c;
    while ( ( c = getchar() ) = = ' ' );
    if ( (c = = '.')  (isdigit(c)) ) {
       ungetc(c, stdin);
       scanf("% lf", &yylval);
       return NUMBER;
    }
    return c;
}
```

Ejemplo 3 (Calefacción)**.** Lenguaje para un controlador de calefacción (interacción con el usuario) [1]. El lenguaje requiere tokens para temperatura (números) y el estado (encendido/apagado).

```
[0-9]+                yylval=atoi(yytext); return NUMBER;
heat                  return TOKHEAT;
on|off                yylval=!strcmp(yytext,"on"); return STATE;
target                return TOKTARGET;
temperature           return TOKTEMPERATURE;
\n                    /* ignore end of line */;
[ \t]+                /* ignore whitespace */;
```

Esta información estará almacenada en un archivo `tokenizer.l` para generar un lexer usando los tokens:
`NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE`
Es decir, se puede generar el lexer mediante el comando: `lex tokenizer.l`

Los tokens serán usados junto con la siguiente gramática para obtener un analizador sintáctico:

- la interacción con el usuario será con una serie/stream de comandos
- `heat_switch` para el encendido/apagado y `target_set` para la temperatura

$$
\begin{aligned}
\textit{commands} &\rightarrow \textit{commands command} \\
\textit{command} &\rightarrow \textit{heat\_switch} \mid \textit{target\_set} \\
\textit{heat\_switch} &\rightarrow \textit{TOKHEAT STATE} \\
\textit{target\_set} &\rightarrow \textit{TOKTARGET TOKTEMPERATURE NUMBER}
\end{aligned}
$$

Finalmente, la especificación está dada por el siguiente archivo `controller.y`:

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
```

[1] Ejemplo tomado de `https://tldp.org/HOWTO/Lex-YACC-HOWTO-4.html`

```
}

int yywrap()
{
        return 1;
}

main()
{
        yyparse();
}

%}

%token NUMBER TOKHEAT STATE TOKTARGET TOKTEMPERATURE

%%
commands: /* empty */
        | commands command
        ;

command: heat_switch
        | target_set
        ;

heat_switch:
        TOKHEAT STATE
        {
                if($2)
                        printf("\tHeat turned on\n");
                else
                        printf("\tHeat turned off\n");
        }
        ;

target_set:
        TOKTARGET TOKTEMPERATURE NUMBER
        {
                printf("\tTemperature setto %d\n",$3);
        }
        ;
%%
```

Finalmente, el parser se obtiene mediante

`yacc -d controller.y`

Y para ejecutarse:

`gcc -w lex.yy.c y.tab.c -o calefaccion -ly`

donde `lex.yy.c` es el lexer, `y.tab.c` es el parser, `calefaccion` es el nombre del código objeto y `ly` es la biblioteca necesaria para compilar el parser.

Ejercicios

1. Muestra la tabla de parsing **LALR** para la siguiente gramática:

   $S \rightarrow A \mid xb$

   $A \rightarrow aAb \mid B$

   $B \rightarrow x$

2. Considera la siguiente gramática ambigua:

   $S \rightarrow AS \mid b$

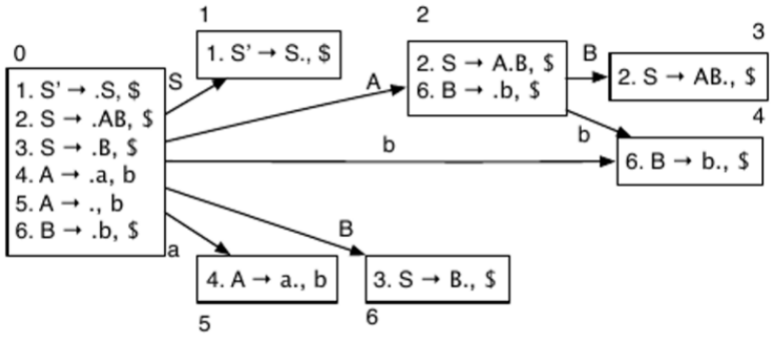
   $A \rightarrow AS \mid a$

   a) Construye la colección de conjuntos de items siguiendo el método de **LR(0)**. Al construir la tabla correspondiente se obtendrán algunos conflictos, ¿cuáles son?

   b) Suponer que se decide consevar a tabla de conflictos y se seleccionará no-determiísticamente alguna acción cuando haya un conflicto. Muestra las diferentes secuencias de acciones para la cadena de entrada *abab*

   c) Describe detalladamente una propuesta para manejar los conflictos en la tabla.

3. Considera el siguiente autómata **LR(1)**:



   a) Proporciona la tabla completa con las acciones y transiciones entre estados.

   b) Hay un estado que tiene un conflicto ¿cuál es el estado? Describa el tipo de conflicto además de explicarlo con sus propias palabras analizando las cadenas que pueden llevar al conflicto.

   c) Describa detalladamente una propuesta para manejar los conflictos en la tabla.

4. Realiza una investigación acerca de dos generadores de parsers (diferentes a `yacc`) describiendo el tipo de gramática que recibe de entrada o el formato del archivo que describe el lenguaje, y el tipo de parser que genera.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[4] François Pottier. Presentaciones del curso Compilation (inf564) École Polytechnique, Palaiseau, Francia. `http://gallium.inria.fr/~fpottier/X/INF564/`, 2016. Material en francés.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[7] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos. `https://www.cis.upenn.edu/~cis341/current/`, 2018.