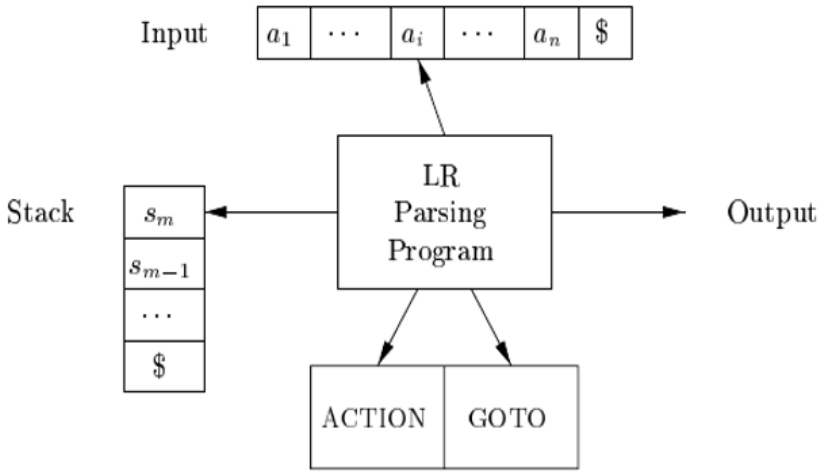
Análisis Sintáctico: *Parser LR: errores y generadores*
Compiladores 2020-1, Nota de clase 6

Lourdes Del Carmen González Huesca

30 de septiembre de 2019
Facultad de Ciencias UNAM

Revisemos en general un analizador LR y sus interacciones:



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. La tabla de parsing contiene las acciones a realizar por el analizador y las transciones. El autómata finito se usa para 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.

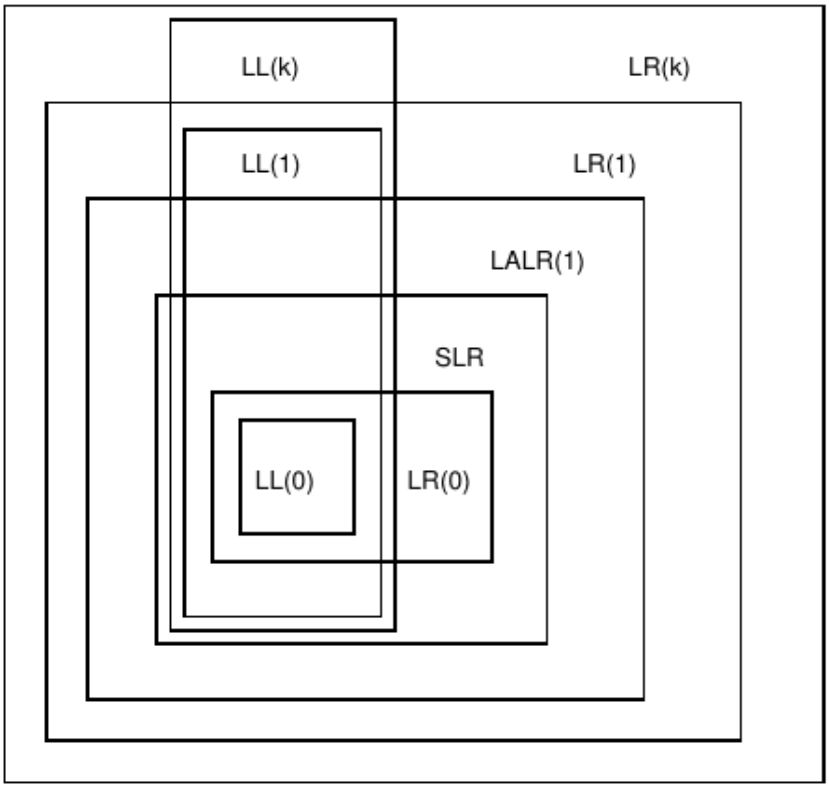
El autómata finito se representa por una tabla en donde se tienen entradas para estado y símbolos terminales que generan acciones y las entradas estado y símbolo no-terminal determinan las transiciones entre estados. Las acciones pueden ser shift o reduce.

Veamos 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.

La siguiente figura agrupa las clases de gramáticas:



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 que 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 revisamos sucede los siguiente:

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

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 que llevó a ese estado y también se tienen los símbolos en la pila:

- 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 a 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 caso de la tabla de parsing en donde haya un error y decidir tomando en cuenta el uso del lenguaje el error más común que pueda ser generado por el programador. Estos errores 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 elminiar 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 la tabla de parsing construida mediante el método LR y resolviendo los conflictos de manera manual ya que la gramática es ambigua, y la segunda es la 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	

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 se archivo 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;
}
```

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, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

[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, 2018. `https://www.cis.upenn.edu/~cis341/current/`.