26 DE MAYO DE 2017

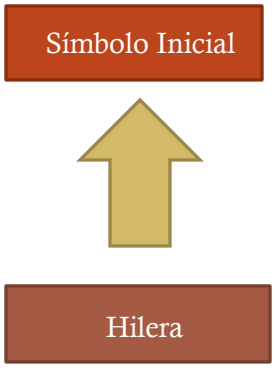
COMPILADORES E INTÉRPRETES

APUNTES

LUIS DANIEL CORDERO LEONHARDES

Análisis sintáctico
Parsing bottom up

Se empieza por la hilera y se debe llegar al símbolo inicial.



Donald Ervin Knuth (1938-Actualidad), USA

Matemático y científico de la computación.

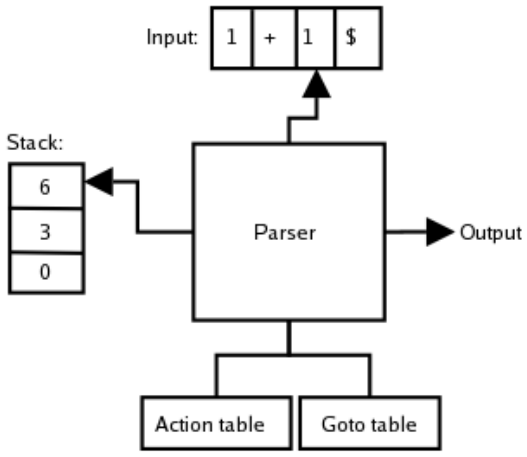
Autor del libro "The art of computer programming"

Tiene varias contribuciones en:

- Análisis de algoritmos.
- Compiladores, en el desarrollo del parsing LR (bottom up).
- Arquitectura de computadores.
- Creador de TEX (luego LATEX).
- Gano ~~una bacinilla~~ un premio Turing en 1974.

Parsing LR

- Creado por Knut en 1960s.
- L = El texto se lee de izquierda a derecha sin back tracking.
- R = si la hilera esta correcta se encuentra la derivación rightmost que la genera.
- Usualmente LR (k) con k símbolos de lookahead, ejemplo LR(0), LR(1), LR(2),etc
- El parser siempre tiene un símbolo actual, y se debe derivar el terminal que está más a la derecha, al ser bottom up comienza con 0.



Action Table Goto table

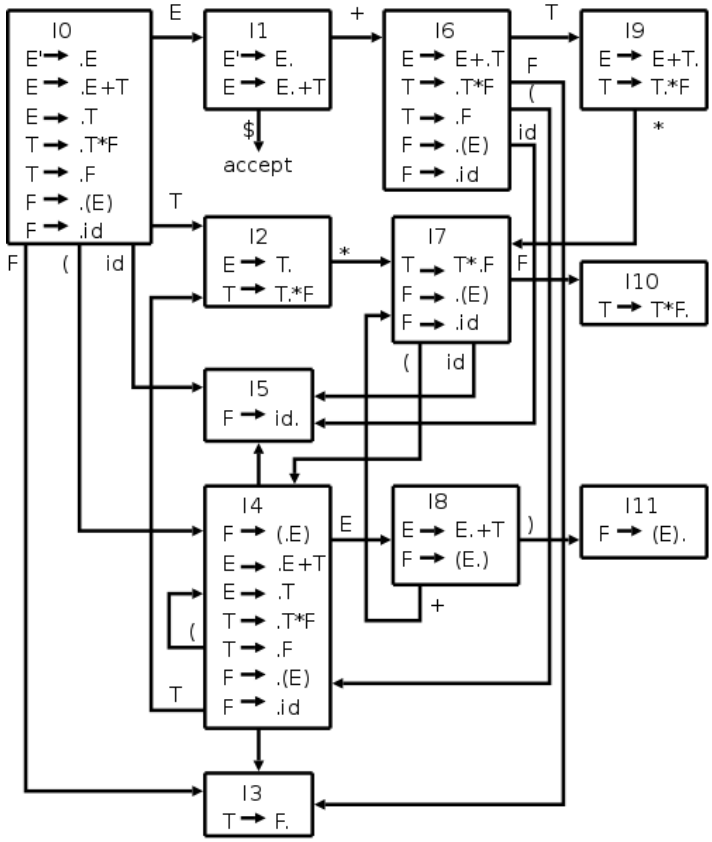
State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example LR Parsing

Grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

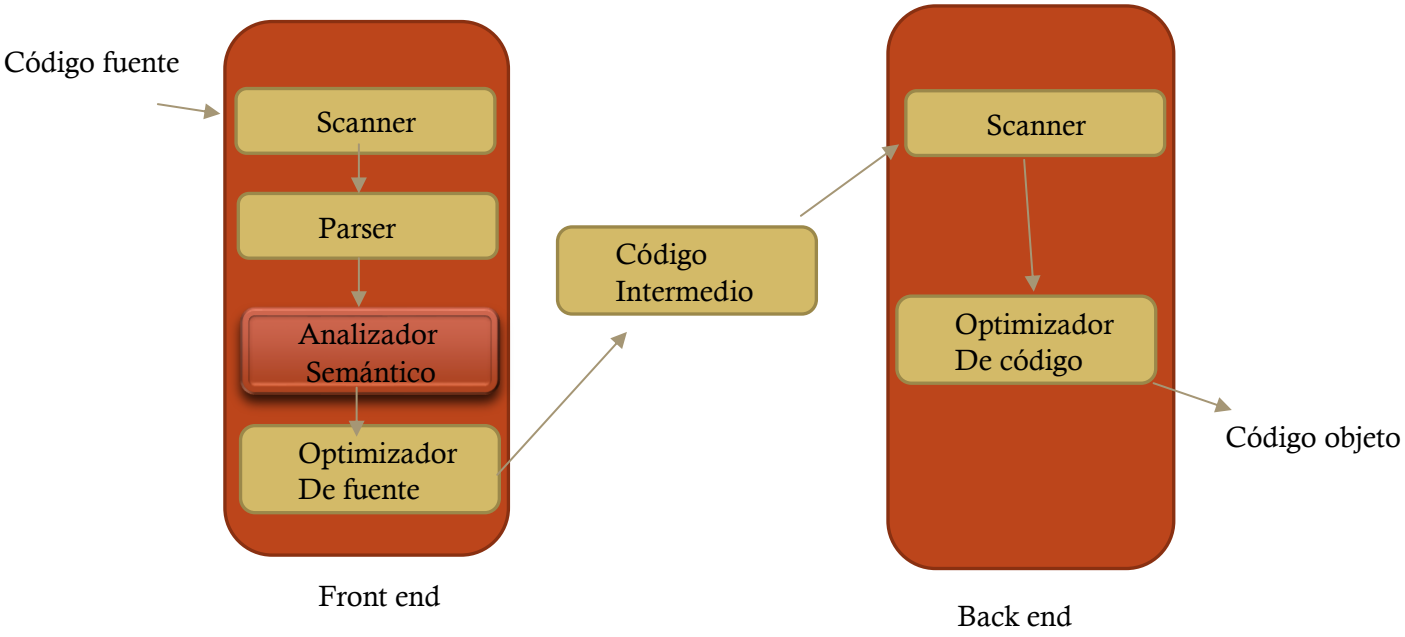


Propiedades del parser LR

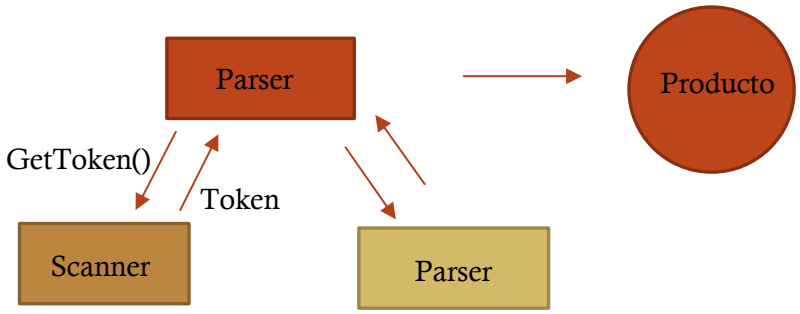
- Existen algoritmos que generan automáticamente las tablas de parsing LR
- Derivacion rigthmost encontrada bottom up de manera determinística
- Parser "shift reduce"
- El parsing LR tenía el problema de que generaba tablas muy grandes, que incluso en los tiempos actuales son consideradas grandes.
- Para solucionar esto se creó el SLR , (Simple LR), permitía mejor espacio pero sacrificaba ciertas gramáticas.
- Si el SLR no sirve existe el LALR (look ahead LR), que es intermedio a LR y SLR
- El SLR y LALR fueron creados por Frank DeRemer en 1969

Tarea de apuntador: el parser LR fue creado exactamente en 1965.

Análisis semántico



- Significado del programa.
- Revisa si tiene sentido lo que se pide que se haga.
- Determina su comportamiento a tiempo de corrida.
- Semántica estática.
- Declaraciones, revisiones de tipos.
- Construye la tabla de símbolos.
- Árbol de atributos (decorado).



La traducción es dirigida por la sintaxis.

Actores del análisis sintáctico

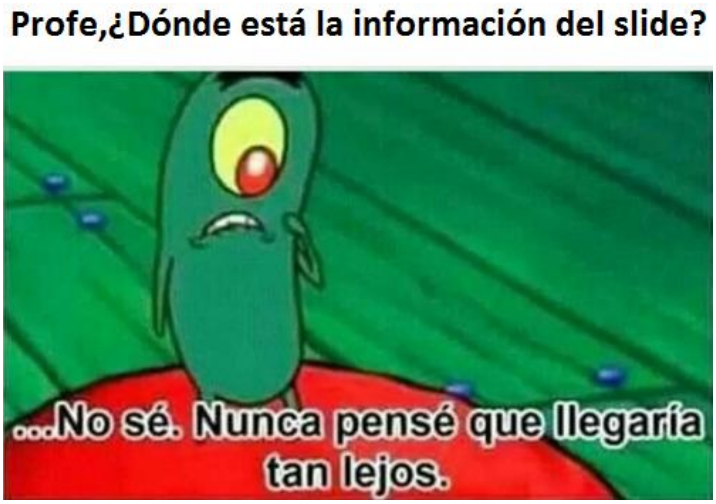
- Pila semántica.
- Registros semánticos.
- Acciones semánticas.

Geografía gramatical

¿A dónde van las acciones?

Declaraciones de variables

Aquí no hay slide.



Si se tiene el código.

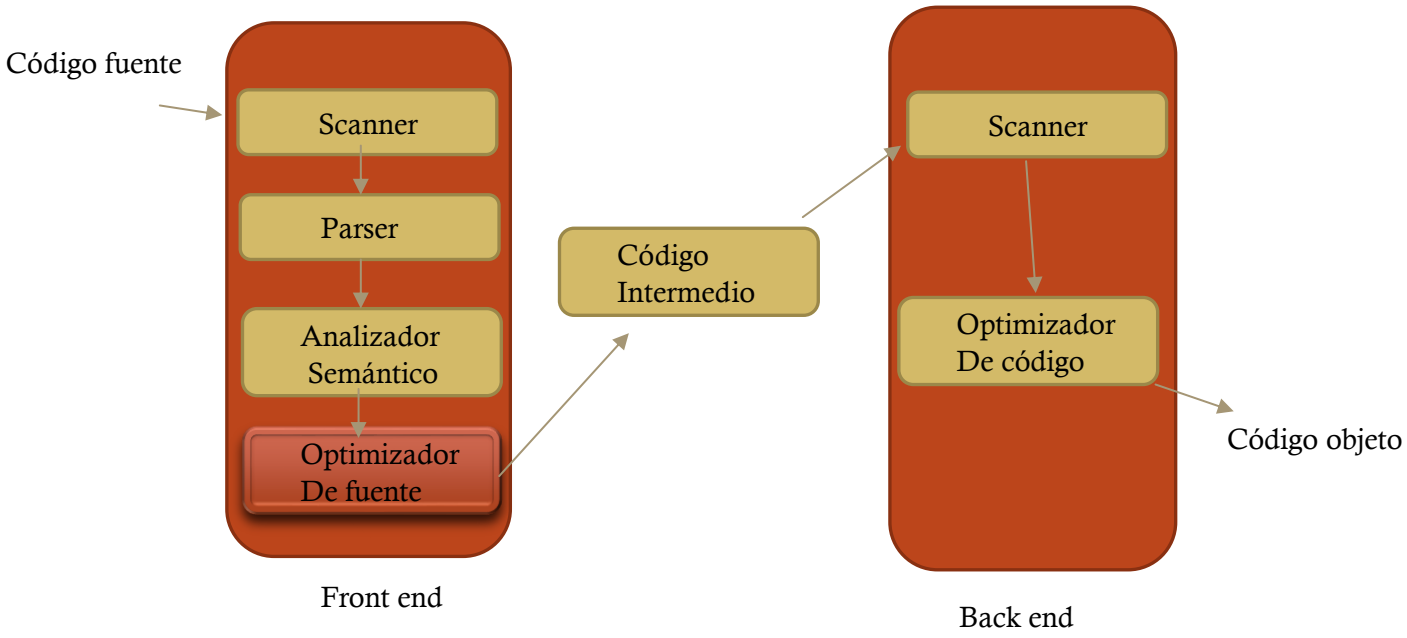
Int k;

Al llegar a la regla de la gramática:

Declaración -> tipo ID:

En la parte de la gramática “tipo * ID” donde se encuentra el “*” se revisa el tipo, este dato se obtiene de la pila semántica.

Optimizador de fuente



- Introduce optimizaciones
- Calculo de constantes, propagación de constantes
- Eliminación de código innecesario
- Se puede hacer sobre árbol anotado
- Es fácil convertir un árbol en estructura lineal
- Genera un código similar a un ensamblador de 3 direcciones

Constant propagation

Proceso de sustituir los valores de constantes conicidas en tiempo de compilación se combina con constant folding, por ejemplo

```
Int x = 14;
Int y = 7 - x / 2;
Return y * (28 / x + 2);
```

Se convierte en:

```
Int x = 14;
Int y = 7 - 14 / 2;
Return y * (28 / 14 + 2);
```

Y luego en

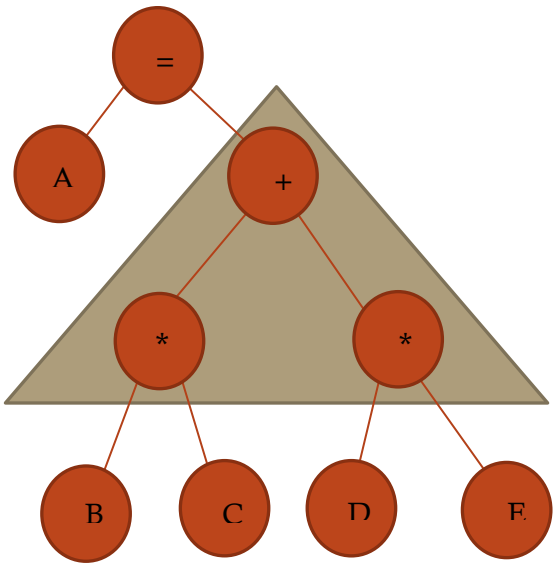
```
Int x = 14;
Int y = 0;
Return 0;
```

Código de 3 direcciones

- Conocido como TAC o 3AC
- Cada instrucción tiene máximo 3 argumentos
- Típicamente una asignación y un operador binario
- Muchas variables intermedias temporales
- Etiquetas (parecido a ensamblador)
- Comparar contra cero y bifurcar
- Se vale tener Go To

Ejemplo 1

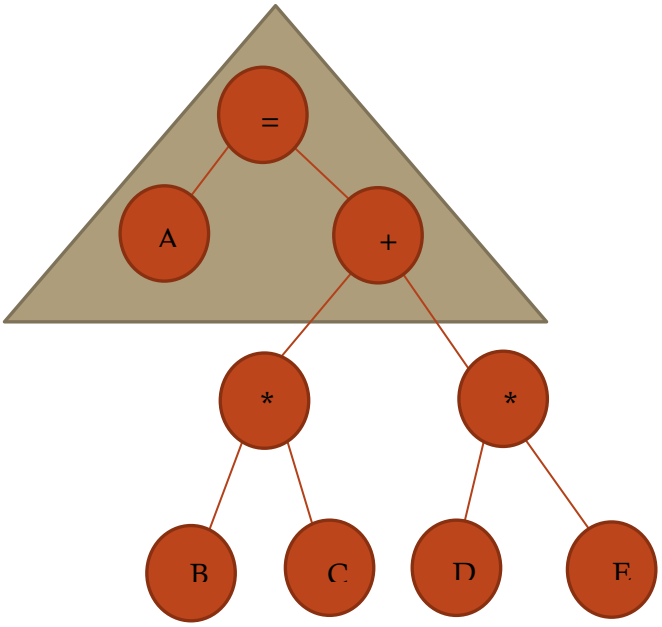
A = B + C +D * E



T1 = B*C

T2= D*E

T3 = T1+T2



T1 = B*C

T2= D*E

T3 = T1+T2

T4 = A =T3

Ejemplo 2

```
If (a  < (b+c))
        A= a-c;
C = b* c
```

```
T1 = B + C
T2 = A < T1
IFZ T2 L1
T3 = A – C
T4 = A = T3
L1: T5 = B* C
T6 = C = T5
```

Generador de código intermedio

- Back End del compilador
- Convierte el código intermedio en código Real
- ¿Ensamblador? ¿Lenguaje maquina?
- Representación de datos muy importante

De código intermedio a ensamblador

- Hay un número finito de instrucciones de código intermedio.
- Se puede tener patrones predefinidos en ensamblador para cada una
- La eficiencia no se considera todavía
- Es relativamente fácil de hacer
- Fácil de cambiar arquitectura de destino.

Ejemplo 1

A = B * C + D * E

T1 = B * C

T2 = D * E

T3 = T1 + T2

T4 A = T3

```
mov ax, c
mul B
mov T1, ax
------------------------
mov ax, E
mul D
mov T2, ax
----------------------------
mov ax, T2
add ax, T1
mov T3, ax
--------------------------
mov ax, T3
mov A, ax
mov T4, ax
```

Ejemplo 2

```
if (a < (b + c))
        a = a – c;
c = b * c;
```

```
T1 = B + C
T2 = A < T1
IFZ T2 L1
T3 = A – C
T4 = A = T3
L1: T5 = B* C
T6 = C = T5
```

```
mov ax, C
add ax, B
mov T1, ax
----------
mov ax, T1
cmp ax, A
setl ax
mov T2, ax
----------
mov ax, T2
cmp $0, ax
jeq L1
----------
mov ax, A
sub ax, C
mov T3, ax
----------
mov ax, T3
mov A, ax
mov T4, ax
----------
L1: mov ax, C
mul B
mov T5, ax
----------
mov ax, T5
mov C, ax
mov T6, ax
```

Optimizador de código

- Mejora el código generado en el paso previo
- Instrucciones equivalentes, pero más rápidas
- Modos de direccionamiento más apropiados
- Eliminar código redundante
- Eliminar código innecesario
- Optimizar para espacio o para tiempo

Ejemplo 1

A = B * C + D * E

```
mov ax, C
mul B
mov T1, ax
-------------------------
mov ax, E
mul D
mov T2, ax
-----------------------------
mov ax, T2
add ax, T1
mov T3, ax
-----------------------------
mov ax, T3
mov A, ax
mov T4, ax
```

```
mov ax, C
mul B
mov dx, ax
mov ax, D
mul E
add ax, dx
mov A, ax
```

Ejemplo 2

```
if (a < (b + c))
        a = a – c;
c = b * c;
```

```
mov ax, C
add ax, B
mov T1, ax
-----------------------
mov ax, T1
cmp ax, A
setl ax
mov T2, ax
-----------------------
mov ax, T2
cmp $0, ax
jeq L1
-----------------------
mov ax, A
sub ax, C
mov T3, ax
---------------------
mov ax, T3
mov A, ax
mov T4, ax
-----------------------
L1: mov ax, C
mul B
mov T5, ax
-------------------------
mov ax, T5
mov C, ax
mov T6, ax
```

```
mov ax, C
mov bx, B
add ax, bx
cmp ax, A
jle L1
mov ax, A
sub ax, C
mov A, ax
L1: mov ax,c
mul B
mov C, ax
```