Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Interpretes

Profesor:

Francisco Torres Rojas

Estudiante:
Ariana Michelle Bermúdez Venegas

Apuntes del 19 de mayo

Período:
I Semestre

Año:
2017

Apuntes

Análisis Sintáctico

Retomando lo que vimos en la clase anterior.

Construcción de Tabla de Parsing

Nuestra tabla **M** tiene:

- Una fila por cada No Terminal y
- Una columna por cada terminal y $

Entonces calculamos los FIRST() y los FOLLOW(), luego el PREDICT de cada **regla**

Y ahora si, lo importante para la tabla, para toda regla A → α: Para cada elemento a en PREDICT (A → α) agregue regla A → α en M[A][a].

Ejemplo 1 de Construcción de Tabla de Parsing

Regla	*PREDICT()*
1. S → (S) S	{ (}
2. S → ε	{) , $ }

	(	)	$
S	1	2	2

Ejemplo 2 de Construcción de Tabla de Parsing

Regla	*PREDICT()*
1. E → T E′	{ (, #}
2. E′ → OP T E′	{ +, - }
3. E′ → ε	{ $,) }
4. OP → +	{ + }
5. OP → -	{ - }
6. T → F T′	{ (, #}
7. T′ → M F T′	{ * }
8. T′ → ε	{ $,), +, -}
9. M → *	{ * }
10. F → (E)	{ (}
11. F → #	{ # }

	(	#	)	+	-	*	$
E	1	1					
E'			3	2	2		3
OP					5		
T	6	6					
T'			8	8	8	7	
M						9	
F	10	11					

Ejemplo 3 (secuencia de instrucciones)

- Recuerden que antes de hacer todo (FIRST(), FOLLOW(), etc), hay que quitarle la recursividad por la izquierda, y factorizar si es necesario.

	FIRST()	
No terminal	Pasada 1	Pasada 2
SQ	Ø	{<statement>}
S	{<statement>}	
SQ'	{ ;, ε }	

	FOLLOW()	
No terminal	Pasada 1	Pasada 2
SQ	{ $ }	
S	{ ;, $ }	
SQ'	{ $ }	

	FIRST()	*FOLLOW()*
SQ	{<statement>}	{ $}
S	{<statement>}	{ ;, $ }
SQ'	{ ;, ε }	{ $ }

Regla	*FIRST(α)*	*PREDICT()*
1. SQ → S SQ'	{<statement>}	{<statement>}
2. SQ' → ; SQ	{ ; }	{ ; }
3. SQ' → ε	{ ε }	{ $ }
4. S → <statement>	{<statement>}	{<statement>}

	<statement>	;	$
SQ	1		
S	4		
SQ'		2	3

Ejemplo 4 (secuencia de instrucciones)

Como siempre, calculamos el FIRST(), luego el FOLLOW()

	FIRST()	*FOLLOW()*
S	{ if, <other> }	{ $, else}
IFS	{ if }	{ $, else}
EL	{ else, ε }	{ $, else}
EX	{ 0, 1 }	{) }

Luego hacemos First con el α, osea lo que tenemos a la izquierda de la regla, con ayuda de esto calculamos el predict.

Regla	*FIRST(α)*	*PREDICT()*
1. S → IFS	{ if }	{ if }
2. S → <other>	{<other>}	{<other>}
3. IFS → if (EX) S EL	{ if }	{ if }
4. EL → else S	{ else }	{ else }
5. EL → ε	{ ε }	{ $, else }
6. EX → 0	{ 0 }	{ 0 }
7. EX → 1	{ 1 }	{ 1 }

Regla	*PREDICT()*
1. S → IFS	{ if }
2. S → <other>	{<other>}
3. IFS → if (EX) S EL	{ if }
4. EL → else S	{ else }
5. EL → ε	{ $, else }
6. EX → 0	{ 0 }
7. EX → 1	{ 1 }

Teniendo el predict, se puede hacer la table de parsing…

	if	<other>	else	0	1	$
S	1	2				
IFS	3					
EL			4 – 5			5
EX				6	7	

Encontramos un problema con la casilla que contiene (4 - 5), ya que hay un conflicto de reglas, cuando la computadora lee esto, no sabe para donde agarrar, y por ello es un caso de ambigüedad.

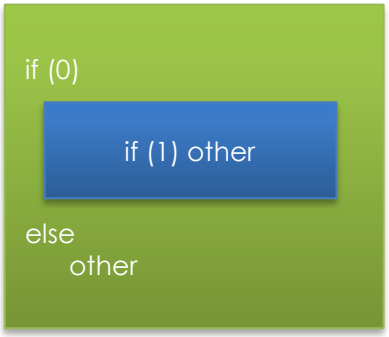
¿Qué hacemos?

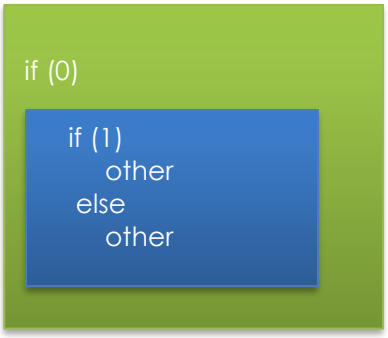
Ambigüedad del if-then-else

Si consideramos el siguiente código:

if (0) if (1) other else other

Pensariamos, con la gramática vista, hay dos posibilidades o formas de verlo.





¿Cuál esta correcta? Ambas, sin embargo, es más natural para nosotros asociar el else al if más cercano. Este es el problema del "dangling else".

Volviendo a la pregunta que nos hicimos, lo que haremos es forzar que else se asocie con el if abierto más cercano. Este comportamiento se puede forzar también en la Tabla de Parsing. Y el macheteo que hicimos es, escoger siempre la regla 4 sobre la regla 5. 😊

Antes

	if	<other>	else	0	1	$
S	1	2				
IFS	3					
EL			4 – 5			5
EX				6	7	

Después

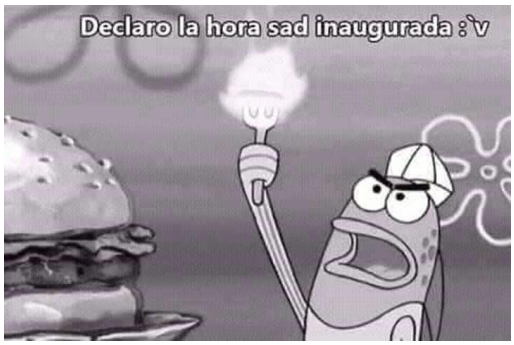
	if	<other>	else	0	1	$
S	1	2				
IFS	3					
EL			4			5
EX				6	7	

Ojo, adelantándonos, podemos decir que esto por definición no es LL(1). (Luego lo retomaré).

Resumen

- Se toma una CFG.
- Eliminar recursividad por la izquierda.
- Factorizar por la izquierda.

- Calcular FIRST() para todos los no terminales.
- Calcular FOLLOW)= para todos los no terminales.
- Calcular PREDICT() para todas las reglas.
- Construir tabla de parsing.



- No todas las gramáticas permiten este proceso. ☹

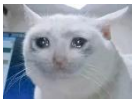
Parsing LL(1)

Definición:

- Una CFG es LL(1) si para cualquier par de reglas diferentes $(A \rightarrow \alpha)$ y $A \rightarrow \beta$ se cumple que $PREDICT(A \rightarrow \alpha) \cap PREDICT(A \rightarrow \beta) = \emptyset$

-Una gramática que es LL(1) permite que uno construya una tabla de parsing LL(1) para ella.

-Estas gramáticas (LL(1)) son un subconjunto propio de todas las gramáticas libre de contexto (CFG).

-**No todas son LL(1).** 

Pero para las otras por dicha ya hay herramientas como Bison :3

Propiedades de Parsers LL(1)

- Sea G un CFG, si G es LL(1), entonces:
  - Si una hilera no se rechaza, se garantiza que obtenemos la derivación más izquierda correcta que la genera.
  - G no es ambigua.
  - La complejidad temporal y espacial del parsing LL(1) es lineal con respecto al largo de la hilera.

Parsers LL(2)

- Un Parser LL(1) puede **predecir** la regla que debe usarse con sólo **un símbolo** de lookahead.
- Hay gramáticas que no son LL(1), aunque no sean recursivas por la izquierda y estén factorizadas.
- Puede ser que viendo 2 símbolos por adelantado se pueda predecir la regla que deba usarse.
- Parsing LL(2)
- Claro para nosotros implica un montón de pasos y mayor complejidad.

- *Por dicha, hasta el momento no nos hemos tenido que complicar con el procedimiento del LL(2).*

- Se definen las funciones:

    - $FIRST_2()$
    - $FOLLOW_2()$
    - $PREDICT_2()$

- La tabla de LL(2) tiene $(n^2 + n)$ columnas, donde **n** es la cantidad de terminales de la gramática.
- *Y para alegrarnos de que no tenemos que hacer esto, vamos a ver un ejemplo de cómo se vería la tabla de parsing.*

Reglas:
1. S → (S)S
2. S → ε

Tabla de parsing:

	((	()	($	)(	))	)$
S						

Parsers LL(k)

- Caso general: parsers LL(k)
- Un lookahead de k simbolos para decidir que regla usar.
- Hay:
    - $FIRST_k()$
    - $FOLLOW_k()$
    - $PREDICT_k()$
- Tablas con $(n^k + n^{k-1})$ columnas, donde **n** es la cantidad de terminales de la gramática.
- Casi todos los lenguajes de programación son LL(1), o si no, se pueden usar otro tipo de parsers.

Descenso Recursivo

¿Recuerdan Micro?

- Para el primer proyecto del curso, nos dejaron una progra de un COMPILADOR completo, para MICRO.

- La gramática de MICRO era LL(1)… pero como no sabíamos nada, no usamos ninguna tabla.
- Usamos sin saber, una técnica de compilación conocida como "Descenso Recursivo". La cual es equivalente a LL(1) (ósea si una gramática es LL(1), podemos usar el descenso recursivo, y viceversa, **ambas técnicas tienen los mismos requisitos**).
- Básicamente se toma la gramática y se pasa a código. *A lo chancho… jaja*
- Siempre hay un símbolo actual de la hilera que se está parseando.
- Cada no terminal tendrá un procedimiento asociado.
- Dentro del proceso de cada no terminar se escoge a cuál regla de la gramática aplicar, dependiendo del símbolo actual.
- El lado derecho de cada regla de la gramática se convierte en llamadas a los procedimientos de cada no terminal o un **match** del símbolo actual con el terminal esperado.

Ejemplo de Descenso Recursivo

Gramática:
E → i E'
E' → + i E' | ε

Código:

```
1
2   E (){
3       match('i');
4       Eprima();
5   }
6
7
8   Eprima(){
9       if (c == '+'){
10          match('+');
11          match('i');
12          Eprima();
13      }
14      else
15          return;
16  }
17
18  match(char t){
19      if(t == c)
20          c = gettoken();
21      else
22          printf("ERROR\n");
23  }
24
25  main(){
26      c = gettoken();
27      E(); // como es el simbolo inicial es lo que va a llamar
28  }
```

Ejemplo de Descenso Recursivo 2

Gramática:
SQ → S SQ'
SQ' → ; SQ | ε
S → <statement>

Código:

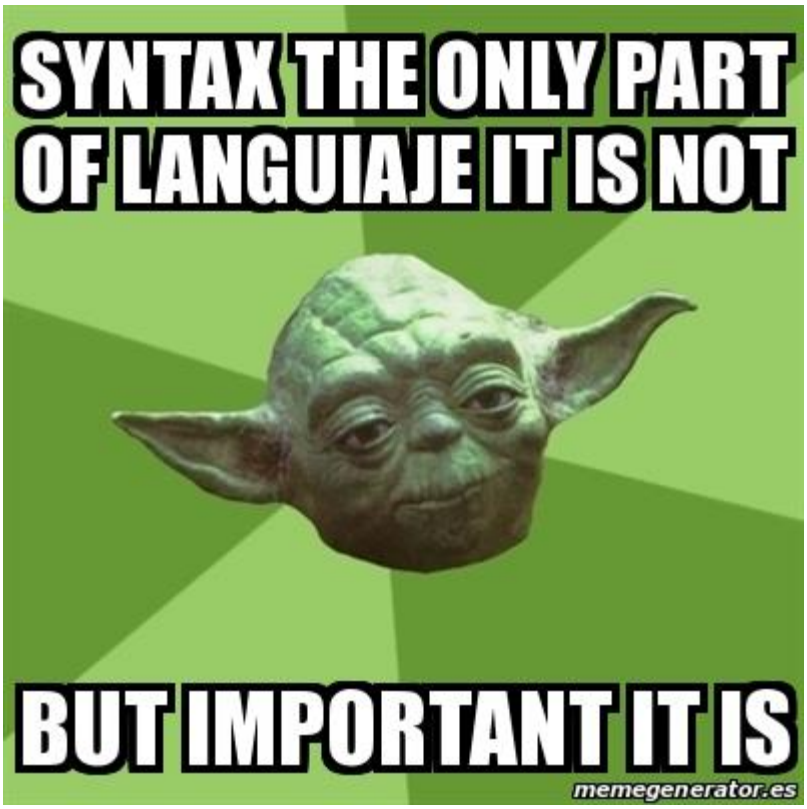
```
S(){
    match('<statement>');
}

SQ(){
    S ();
    SQprima();
}

SQprima(){
    if (c == ';'){
        match(';');
        SQ();
    }
    else
        return;
}

match(char t){
    if(t == c)
        c = gettoken();
    else
        printf("ERROR\n");
}

main(){
    c = gettoken();
    SQ(); // como es el simbolo inicial es lo que va a llamar
}
```

Gramática:
S → IFS | <other>
IFS → if (EX) S EL
EL → else S | ε
EX → 0 | 1

Código:

```
S(){
    if (c == 'if')
    {
        IFS();
    }
    else
        match('<other>');
}

IFS(){
    match('if');
    match('(');
    EX();
    match(')');
    S();
    EL();
}

EL(){
    if (c == 'else')
    {
        match('else');
        S();
    }
    else
        return;
}

match(char t){
    if(t == c)
        c = gettoken();
    else
        printf("ERROR\n");
}

main(){
    c = gettoken();
    S(); // como es el simbolo inicial es lo que va a llamar
}
```

Análisis Sintáctico



Mensajes de Error – Errores y LL(1)

- El algoritmo detecta hileras inválidas - entrada vacía en tabla LL(1)
- Estrategia simplista:
  - Que le diga: "Tiene un error :v", "Error de sintaxis aquí".
  - No revisa el resto de la hilera.
- Mejor enfoque:
  - **Mensajes de error:** dar el mejor diagnostico posible de la causa del error.
  - **Recuperación de error:** volver a sincronizar el autómata para reportar otros errores en hilera.
- Es casi un arte.
- Distintos compiladores dan mensajes diferentes para el mismo error.
- Hay muchas causas posibles.
- Es particular para cada lenguaje y situación.
- A veces se hacen de manera incremental:

- Dar solo el número de error. (ERROR 17)
- Ver ejemplos del error e ir mejorando la redacción.