Universidad Nacional Autónoma de México

Facultad de Ciencias

Tarea 03

Compiladores

Bonilla Reyes Dafne
Castañón Maldonado Carlos Emilio
García Ponce José Camilo
Velasco García Jorge Daniel

Profesora: Lourdes del Carmen González Huesca

Ayudante: Fausto David Hernández Jasso
Ayudante: Juan Alfonso Garduño Solís
Ayudante: Juan Pablo Yamamoto Zazueta

Marzo 2024

1. Considera la siguiente gramática:

$$E \to -E \mid (E) \mid VE'$$
$$E' \to -E \mid \epsilon$$
$$V \to idV'$$
$$V' \to (E) \mid \epsilon$$

a) **(1.5 pts)** Construye la tabla de parsing para un parser tipo LL(1) mostrando el cálculo de los conjuntos `First` y `Follow`.

✪ `First y Follow`:

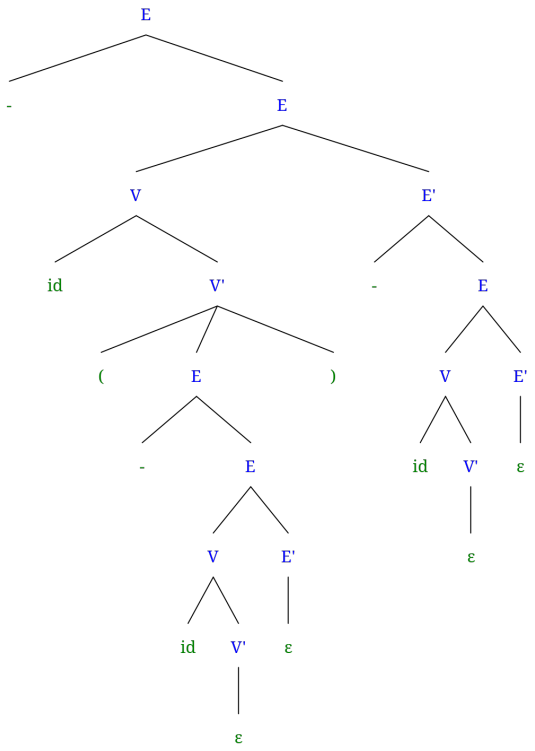
	FIRST	FOLLOW
E	{-, (} U First(V) = {-, (, id}	{), $}
E'	{-, ε}	Follow(E) = {), $}
V	{id}	First(E') U Follow(E) = {-,), $}
V'	{(, ε}	Follow(V) = {-,), $}

✪ `Tabla de parsing:`

	-	(	)	id	$
E	E -> - E	E -> (E)		E -> V E'	
E'	E' -> - E		E' -> ε		E' -> ε
V				V -> id V'	
V'	V' -> ε	V' -> (E)	V' -> ε		V' -> ε

b) **(2pts)** Muestra el árbol de sintaxis que se obtiene al ejecutar el algoritmo para procesar la cadena `-id(-id)-id`. Incluye una tabla para ver el progreso del algoritmo.

✪ `Árbol de sintaxis:`



✪ Tabla de algoritmo:

RECONOCIDO	PILA	ENTRADA	SALIDA
	$ E	- id (- id) - id $	
	$ E -	- id (- id) - id $	E -> - E
-	$ E	id (- id) - id $	match -
-	$ E' V	id (- id) - id $	E -> V E'
-	$ E' V' id	id (- id) - id $	V -> id V'
- id	$ E' V'	(- id) - id $	match id
- id	$ E') E (	(- id) - id $	V' -> (E)
- id (	$ E') E	- id) - id $	match (
- id (	$ E') E -	- id) - id $	E -> - E
- id (-	$ E') E	id) - id $	match -
- id (-	$ E') E' V	id) - id $	E -> V E'
- id (-	$ E') E' V' id	id) - id $	V -> id V'
- id (- id	$ E') E' V'	) - id $	match id
- id (- id	$ E') E'	) - id $	V' -> ε
- id (- id	$ E')	) - id $	E' -> ε
- id (- id)	$ E'	- id $	match)
- id (- id)	$ E -	- id $	E' -> - E
- id (- id) -	$ E	id $	match -
- id (- id) -	$ E' V	id $	E -> V E'
- id (- id) -	$ E' V' id	id $	V -> id V'
- id (- id) - id	$ E' V'	$	match id
- id (- id) - id	$ E'	$	V' -> ε
- id (- id) - id	$	$	E' -> ε
			terminar

2. Considera la siguiente gramática

$$\begin{aligned} stmt &\rightarrow \textbf{if } bexpr \textbf{ then } stmt \textbf{ else } stmt | \textbf{while } bexpr \textbf{ do } stmt | \textbf{begin } stmts \textbf{ end} \\ stmts &\rightarrow stmt; stmts | \epsilon \\ bexpr &\rightarrow id_i \end{aligned}$$

donde las expresiones booleanas están representadas por terminales de la forma id_i.
Además, considera la siguiente gramática, alternativa a la anterior:

$$\begin{aligned} stmt &\rightarrow \textbf{if } bexpr \textbf{ then } stmtTail | \textbf{while } bexpr \textbf{ do } stmt | \textbf{begin } list \textbf{ end} | s \\ stmtTail &\rightarrow \textbf{else } stmt | \epsilon \\ list &\rightarrow stmt\ listTail \\ listTail &\rightarrow ; list | \epsilon \\ bexpr &\rightarrow id_i \end{aligned}$$

donde id_i y s se consideran terminales para las guardias booleanas y otros enunciados.

a) **(1.5 pts)** Explica las diferencias entre ambas gramáticas.

Son gramáticas que generan lenguajes similares a primera vista (pero no logramos encontrar cadenas que se puedan generar con ambas gramáticas), estos lenguajes son como estructuras de control en lenguajes de programación, pero hay ciertas cadenas que la gramática 1 genera, pero la gramática 2 no y viceversa.

Por ejemplo:

I) La gramática 1, puede generar la cadena `begin end`, con `stmt` $\rightarrow$ `begin stmts end` $\rightarrow$ `begin` ε `end` = `begin end`, pero con la gramática 2 no se puede generar debido a que el no terminal `list` no se puede derivar en una ε solita y tenemos `begin list end`.

II) La gramática 2, puede generar la cadena `if` id_1 `then`, con `stmt` $\rightarrow$ `if bexpr then stmtTail` $\rightarrow$ `if` id_1 `then stmtTail` $\rightarrow$ `if` id_1 `then` ε = `id` id_1 `then`, pero con la gramática 1 no se puede generar debido a que el no terminal `stmt` no es puede derivar en una ε solita y tenemos `if bexpr then stmt else stmt`, en general la gramática 1 no permite `if` cortos.

III) La gramática 2 tiene `s` (esto permite poder generar un solo terminal desde `stmt`), pero la gramática 1 no y también los `begin end` de la gramática 1 siempre tiene ; antes del end (o no tienen nada entre el `begin` y `end`), pero en la gramática 2 no sucede esto.

IV) La gramática 2 solo puede tener un `then` y luego un `else` (o ε) pero ningún `stmt` luego de `then`, como en la gramática 1.

b) **(2 pts)** Muestra la tabla de parsing predictivo para la segunda gramática.

✪ `First y Follow`:

	FIRST	FOLLOW
stmt	{if, while, begin, s}	{$} U Follow(list) U First(listTail) U Follow(stmtTail) = {$, end, ;}
stmtTail	{else, ε}	Follow(stmt) = {$, end, ;}
list	First(stmt) = {if, while, begin, s}	{end}
listTail	{;, ε}	Follow(list) = {end}
bexpr	{id_i}	{then, do}

✪ Tabla de parsing:

	if	then	while	do	begin	end	s	else	;	id_i	$
stmt	stmt -> if bexpr then stmtTail		stmt -> while bexpr do stmt		stmt -> begin list end		stmt -> s				
stmtTail						stmtTail -> epsilon		stmtTail -> else stmt	stmtTail -> epsilon		stmtTail -> epsilon
list	list -> stmt listTail		list -> stmt listTail		list -> stmt listTail		list -> stmt listTail				
listTail						listTail -> epsilon			listTail -> ; list		
bexpr										bexpr -> id_i	

c) **(1.5 pts)** Usa la idea de sincronización de símbolos para resolver errores y completar la tabla del inciso anterior.

✪ Tabla de parsing que maneja errores:

	if	then	while	do	begin	end	s	else	;	id_i	$
stmt	stmt -> if bexpr then stmtTail	scan	stmt -> while bexpr do stmt	scan	stmt -> begin list end	synch	stmt -> s	scan	synch	scan	synch
stmtTail	scan	scan	scan	scan	scan	stmtTail -> epsilon	scan	stmtTail -> else stmt	stmtTail -> epsilon	scan	stmtTail -> epsilon
list	list -> stmt listTail	scan	list -> stmt listTail	scan	list -> stmt listTail	synch	list -> stmt listTail	scan	scan	scan	synch
listTail	scan	scan	scan	scan	scan	listTail -> epsilon	scan	scan	listTail -> ; list	scan	synch
bexpr	scan	synch	scan	synch	scan	scan	scan	scan	scan	bexpr -> id_i	synch

✪ Explicación:

- **Manejo de error `synch`:** si el token actual de la entrada es $ o está en el `Follow` del no terminal del tope de la pila, entonces hacemos pop a la pila (saltarnos la derivación este no terminal).
- **Manejo de error `scan`:** si el token actual de la entrada no es $ ni en el `Follow` o `First` del no terminal del tope de la pila, entonces hacemos `pops` consecutivos a la entrada hasta encontrar un token en el `First` o `Follow` del no terminal para seguir el procedimiento del parsing.
- **Manejo de error `inserción`:** si el tope de la pila es un terminal que no coincide con el siguiente token, entonces hacemos pop en la pila.

d) **(1.5 pts)** Muestra el procesamiento de la cadena *while* id_2 *do begin* s ; *if* id_1 *then* s ; *end* con la segunda gramática y con el manejo de errores.

RECONOCIDO	PILA	ENTRADA	SALIDA
	$ stmt	while id_2 do begin s ; if id_1 then s ; end $	
	$ stmt do bexpr while	while id_2 do begin s ; if id_1 then s ; end $	stmt -> while bexpr do stmt
while	$ stmt do bexpr	id_2 do begin s ; if id_1 then s ; end $	match while
while	$ stmt do id_i	id_2 do begin s ; if id_1 then s ; end $	bexpr -> id_i
while id_1	$ stmt do	do begin s ; if id_1 then s ; end $	match id_1
while id_1 do	$ stmt	begin s ; if id_1 then s ; end $	match do
while id_1 do	$ end list begin	begin s ; if id_1 then s ; end $	stmt -> begin list end
while id_1 do begin	$ end list	s ; if id_1 then s ; end $	match begin
while id_1 do begin	$ end listTail stmt	s ; if id_1 then s ; end $	list -> stmt listTail
while id_1 do begin	$ end listTail s	s ; if id_1 then s ; end $	stmt -> s
while id_1 do begin s	$ end listTail	; if id_1 then s ; end $	match s
while id_1 do begin s	$ end list ;	; if id_1 then s ; end $	listTail -> ; list
while id_1 do begin s ;	$ end list	if id_1 then s ; end $	match ;
while id_1 do begin s ;	$ end listTail stmt	if id_1 then s ; end $	list -> stmt listTail
while id_1 do begin s ;	$ end listTail stmtTail then bexpr if	if id_1 then s ; end $	stmt -> if bexpr then stmtTail
while id_1 do begin s ; if	$ end listTail stmtTail then bexpr	id_1 then s ; end $	match if
while id_1 do begin s ; if	$ end listTail stmtTail then id_i	id_1 then s ; end $	bexpr -> id_i
while id_1 do begin s ; if id_1	$ end listTail stmtTail then	then s ; end $	match id_1
while id_1 do begin s ; if id_1 then	$ end listTail stmtTail	s ; end $	match then
while id_1 do begin s ; if id_1 then	$ end listTail stmtTail	; end $	error, scan (pop a token "s" de la entrada) "token inesperdado: s"
while id_1 do begin s ; if id_1 then	$ end listTail	; end $	stmtTail -> epsilon
while id_1 do begin s ; if id_1 then	$ end list ;	; end $	listTail -> ; list
while id_1 do begin s ; if id_1 then ;	$ end list	end $	match ;
while id_1 do begin s ; if id_1 then ; list	$ end	end $	error, synch (pop a "list" de la pila) "se esperaba un list"
while id_1 do begin s ; if id_1 then ; list end	$	$	match end
			terminar

e) **(1.5 pts)** Muestra una cadena, no trivial, que no pertenezca al lenguaje de la segunda gramática junto con su procesamiento.
¿Es suficiente la propuesta que diste en el inciso c) para manejar los errores?

RECONOCIDO	PILA	ENTRADA	SALIDA
	$ stmt	begin s ; while id_1 ; s do end $	
	$ end list begin	begin s ; while id_1 ; s do end $	stmt -> begin list end
begin	$ end list	s ; while id_1 ; s do end $	match begin
begin	$ end listTail stmt	s ; while id_1 ; s do end $	list -> stmt listTail
begin	$ end listTail s	s ; while id_1 ; s do end $	stmt -> s
begin s	$ end listTail	; while id_1 ; s do end $	match s
begin s	$ end list ;	; while id_1 ; s do end $	listTail -> ; list
begin s ;	$ end list	while id_1 ; s do end $	match ;
begin s ;	$ end listTail stmt	while id_1 ; s do end $	list -> stmt listTail
begin s ;	$ end listTail stmt do bexpr while	while id_1 ; s do end $	stmt -> while bexpr do stmt
begin s ; while	$ end listTail stmt do bexpr	id_1 ; s do end $	match while
begin s ; while	$ end listTail stmt do id_i	id_1 ; s do end $	bexpr -> id_i
begin s ; while id_1	$ end listTail stmt do	; s do end $	match id_1
begin s ; while id_1 do	$ end listTail stmt	; s do end $	error de insercion, diferentes terminales (pop a "do" de la pila) "se esperaba un do"
begin s ; while id_1 do stmt	$ end listTail	; s do end $	error, synch (pop a "stmt" de la pila) "se esperaba un stmt"
begin s ; while id_1 do stmt	$ end list ;	; s do end $	listTail -> ; list
begin s ; while id_1 do stmt ;	$ end list	s do end $	match ;
begin s ; while id_1 do stmt ;	$ end listTail stmt	s do end $	list -> stmt listTail
begin s ; while id_1 do stmt ;	$ end listTail s	s do end $	stmt -> s
begin s ; while id_1 do stmt ; s	$ end listTail	do end $	match s
begin s ; while id_1 do stmt ; s	$ end listTail	end $	error, scan (pop a token "do" de la entrada) "token inesperdado: do"
begin s ; while id_1 do stmt ; s	$ end	end $	listTail -> epsilon
begin s ; while id_1 do stmt ; s end	$	$	match end
			terminar

Creemos que con los 3 casos para manejar errores en la tabla del parsing funciona bien, ya que revisamos los posibles casos (encontrar en la entrada un no terminal o un terminal que no podemos procesar con el no terminal del tope de la pila y cuando un terminal en la entrada no coincide con el terminal de la entrada) que pueden afectar al parser.

Referencias

[1] Syntax analysis