Top-down Predictive Parsing
Article #16

Grammar

Original	Transformed
E -> E + T | T	E -> TE’
	E’ -> +TE’ | ε
T -> T * F | F	T -> FT’
	T’ -> *FT’ | ε
F -> v | (E)	F -> v | (E)

FIRST and FOLLOW sets

Nonterminal	FIRST	FOLLOW
E	{ v , (}	{) , $ }
E’	{ + , ε }	{) , $ }
T	{ v , (}	{ +,) , $ }
T’	{ * , ε }	{ + ,) , $ }
F	{ v , (}	{ * , + ,) , $ }

Parsing Table

Nonterminal	Input Symbol (token)					
	v	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	v			(E)		

Parsing Table

Nonterminal	Input Symbol (token)					
	v	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	v			(E)		

Parsing program (a + b) * c with Predictive Parser

Stack	Input	Table [A,a]
$E	(v+v)*v$	[E,(] => E->TE'
$E'T	(v+v)*v$	[T,(] => T->FT'
$E'T'F	(v+v)*v$	[F,(] => F->(E)
$E'T')E(	(v+v)*v$	Match and Remove (
$E'T')E	v+v)*v$	[E,v] => E->TE'
$E'T')E'T	v+v)*v$	[T,v] => T->FT'
$E'T')E'T'F	v+v)*v$	[F,v] => F->v
$E'T')E'T'v	v+v)*v$	Match and Remove v
$E'T')E'T'	+v)*v$	[T’,+] => T'-> ε
$E'T')E'	+v)*v$	[E’,+] => E'->+TE'
$E'T')E'T+	+v)*v$	Match and Remove +
$E'T')E'T	v)*v$	
$E'T')E'T'F	v)*v$	
$E'T')E'T'v	v)*v$	
$E'T')E'T'	)*v$	
$E'T')E'	)*v$	
$E'T')	)*v$	
$E'T'	*v$	
$E'T'F*	*v$	
$E'T'F	v$	
$E'T'v	v$	
$E'T'	$	
$E'	$	[E’,$] => E'-> ε
$	$	Match and Stop

Parsing Table with Error Recovery Synchronization Entries

Nonterminal	Input Symbol (token)					
	v	+	*	(	)	$
E	TE’			TE’	synch	synch
E’		+TE’			ε	ε
T	FT’	synch		FT’	synch	synch
T’		ε	*FT’		ε	ε
F	v	synch	synch	(E)	synch	synch

Parsing *if statement* grammar with predictive parser

<statement> ->
if <condition> then <statement>
| if <condition> then <statement> else <statement>
| <other statements>
<condition> -> boolean value

The following is an abstract presentation of the same grammar:

S -> i C t S | i C t S e S | O
C -> b
O -> a
This grammar is not an LL grammar because it contains a "left-factor"

Grammar

Original	Transformed
S -> i C t S | i C t S e S | O	S -> i C t S S' | O
	S' -> eS | ε
C -> b	C -> b
O -> a	O -> a

FIRST and FOLLOW sets

Nonterminal	FIRST	FOLLOW
S	{ i , a }	{ e , $ }
S'	{ e , ε }	{ e , $ }
C	{ b }	{ t }
O	{ a }	{ e , $ }

Parsing Table

Nonterminal	Input Symbol (token)					
	a	b	e	i	t	$
S	O			iCtSS'		
S'			eS ε			ε
C		b				
O	a					

The grammar is ambiguous because the [S', e] cell contains two entries.
A grammar whose parsing table has no multiple entries in a cell is said to be LL(1) grammar.

Another way to prove that the *if grammar* is ambiguous.

S -> i C t S | i C t S e S | O
C -> b
O -> a

Given the sentence:

if b then if b then a else a

or

i b t i b t a e a

The following two left-most derivations can be built to parse the sentence:

S => i **C** t S => i b t **S** => i b t i **C** t S e S => i b t i b t **S** e S
=> i b t i b t **O** e S => i b t i b t a e **S** => i b t i b t a e **O** => **i b t i b t a e a**
and

S => i **C** t S e S => i b t **S** e S => i b t i **C** t S e S => i b t i b t **S** e S
=> i b t i b t **O** e S => i b t i b t a e **S** => i b t i b t a e **O** => **i b t i b t a e a**

The grammar can be reworked to remove the ambiguity:

S -> M | U
M -> if C then M else M | O
U -> if C then S | if C then M else U

but the grammar is still not an LL(1) grammar.

LL(1) Grammar Definition

A grammar is an LL(1) grammar if and only if whenever the grammar has a production A -> α | β with two distinctive options α and β the following conditions hold:

1. **For no terminal a do both α and β have a in their FIRST sets.**
2. **At most one of α and β can derive the empty string.**
3. **If β => ε , then α does not have in its FIRST set any terminal which is in FOLLOW(A).**

Non- immediate left recursion

S -> Aa | b
A-> Ac | Sd | ε

Eliminating the recursion

Step 1.

S -> Aa | b
A-> Ac | Aad | bd | ε

Step 2.

S -> Aa | b
A -> bdA’ | A’
A’-> cA’ | adA’ | ε