UNIVERSITY OF NICOSIA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΛΕΥΚΩΣΙΑΣ

COMP-411 Compiler Design

Presented by
Dr Ioanna Dionysiou

Administrative

- [ALSU07] Chapter 4 - Syntax Analysis
  - Top-down parsing (LL(1) parsers) section 4.4

Chapter 4 Outline

- Top-down Parsing
  - Recursive-descent parsing
  - Non-recursive predictive parsing
  - Construction of predictive parser

Top-down Parsing

- It is an attempt to
  - Find a leftmost derivation for an input string
    - Construct a parse tree for the input
      - Start from the root and create the nodes of the parse tree in pre-order
- General form of top-down parsing
  - Recursive descent parsing
    - May involve backtracking to find the correct A-production to be applied
  - Predicting Parsing
    - Special case of recursive-decent parsing
    - Predictive parsers do not allow backtracking
      - Always choose the correct A-production by looking ahead the next input symbol

Predictive Parsers

- Given the input symbol (i.e. token name) *a* and the nonterminal A to be expanded, this type of parser unambiguously determines the proper alternative that derives a string beginning with a

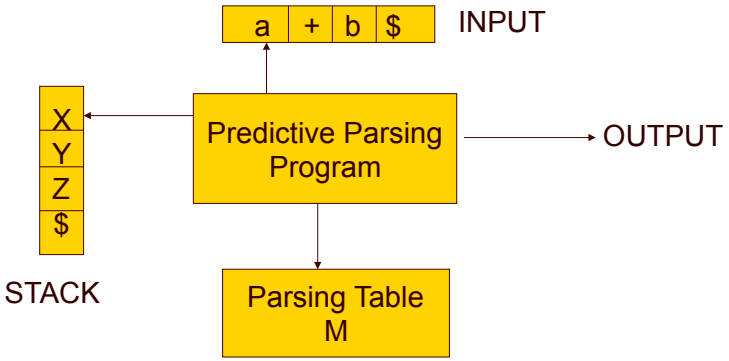
stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* ***end***

The keywords **if while begin** tell us which alternative is the one to succeed if we scan tokens *if while begin* respectively

Predictive Parsers

- It can be built using a stack
  - Explicitly
    - Nonrecursive manner with the aid of a parsing table
  - Implicitly (we will not cover this)
    - Recursive procedure calls

Nonrecursive Predictive Parsing



Nonrecursive Predictive Parsing

- The program considers
  - X, the symbol on top of the stack
  - a, the current input symbol.
  - These two symbols determine the action of the parser. There are 3 possibilities:
    - If X = a = $, the parser halts and announces successful completion of parsing
    - If X = a ≠$, the parser pops X off the stack and advances the input pointer to the next input symbol
    - If X is a nonterminal, the program consults entry M[X,a] of the parsing table M.
      - This entry will be either an X-production of the grammar or an error entry. If for example, M[X,a] = {X→UVW}, the parser replaces X on top of the stack by WVU (with U on the top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a] = error, the parser calls an error recovery routine.

Nonrecursive Predictive Parsing

Input : A string w and a parsing table M for grammar G

Output: If w is in L(G), a leftmost derivation of w; otherwise an error

Method: Initially the parser is in a configuration in which it has:
$S on the stack, with S the start symbol of G on top
w$ in the input buffer

The algorithm that utilizes the predictive parsing table M to produce a parse tree for an input is shown on the next slide

Nonrecursive Predictive Parsing

```
set ip to point to the first symbol of w
repeat
   let X be the top stack symbol and a the symbol pointed to by ip
   if X is a terminal or $ then
      if X = a then
         pop X from the stack and advance ip
      else
         error()
   else {X is a nonterminal}
      if M[X,a] = X →Y1Y2…Yk then
         pop X from the stack
         push Yk ,Yk-1 , …, Y1 onto the stack, with Y1 on top
         output the production X →Y1Y2…Yk
      else
         error()
until X = $ {stack is empty}
```


Moves made by the parser

GRAMMAR G

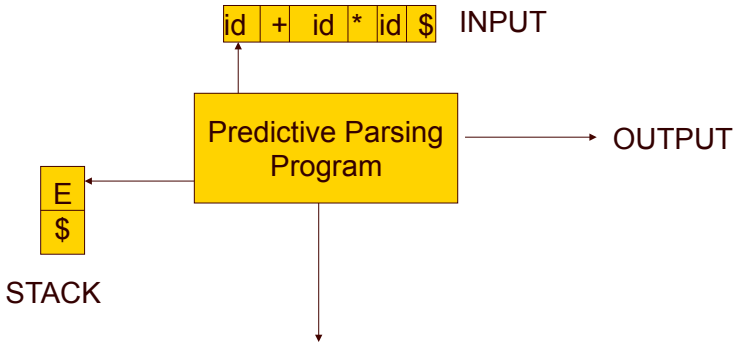
E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

INPUT string w

id + id * id

Derive the leftmost derivation for string w

Moves made by the parser



NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

Initial configuration

STACK	INPUT	OUTPUT
$E	**id + id * id** $	

Check the top of the stack (E) and the current input symbol (id)
Is there an entry in the parsing table?
M[E,id] = E->TE'
Replace E by TE', current input symbol is still id

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id + id * id** $	
$E'T	**id + id * id** $	E→TE'

Check the top of the stack (T) and the current input symbol (id)
Is there an entry in the parsing table?
M[T,id] = E->FT'
Replace T by FT', current input symbol is still id

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id + id * id** $	
$E'T	**id + id * id** $	E→TE'
$E'T'F	**id + id * id** $	T→FT'

What happens next???

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id + id * id** $	
$E'T	**id + id * id** $	E→TE'
$E'T'F	**id + id * id** $	T→FT'
$E'T' **id**	**id + id * id** $	F→**id**

Check the top of the stack (id) and the current input symbol (id)
Pop off the stack id
Since id=id, advance the current input pointer to +

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id** + **id** * **id** $	
$E'T	**id** + **id** * **id** $	E→TE'
$E'T'F	**id** + **id** * **id** $	T→FT'
$E'T'**id**	**id** + **id** * **id** $	F→**id**
$E'T'	+ **id** * **id** $	

Continue…

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id** + **id** * **id**$	
$E'T	**id** + **id** * **id**$	E→TE'
$E'T'F	**id** + **id** * **id**$	T→FT'
$E'T'**id**	**id** + **id** * **id**$	F→**id**
$E'T'	+ **id** * **id**$	
$E'	+ **id** * **id**$	T'→ ε

Continue…

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

STACK	INPUT	OUTPUT
$E	**id** + **id** * **id**$	
$E'T	**id** + **id** * **id**$	E→TE'
$E'T'F	**id** + **id** * **id**$	T→FT'
$E'T'**id**	**id** + **id** * **id**$	F→**id**
$E'T'	+ **id** * **id**$	
$E'	+ **id** * **id**$	T'→ ε
$E'T+	+ **id** * **id**$	E'→+TE'

Continue…

Moves made by parser

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

Repeat the process until encounter $ on the stack

STACK	INPUT	OUTPUT
$E	**id** + **id** * **id**$	
$E'T	**id** + **id** * **id**$	E→TE'
$E'T'F	**id** + **id** * **id**$	T→FT'
$E'T'**id**	**id** + **id** * **id**$	F→**id**
$E'T'	+ **id** * **id**$	
$E'	+ **id** * **id**$	T'→ ε
$E'T+	+ **id** * **id**$	E'→+TE'
$E'T	**id** * **id**$	

Final "Moves" table

STACK	INPUT	OUTPUT
$E	**id + id * id**$	
$E'T	**id + id * id**$	E→TE'
$E'T'F	**id + id * id**$	T→FT'
$E'T'**id**	**id + id * id**$	F→**id**
$E'T'	**+ id * id**$	
$E'	**+ id * id**$	T'→ ε
$E'T+	**+ id * id**$	E'→+TE'
$E'T	**id * id**$	
$E'T'F	**id * id**$	T→FT'
$E'T'**id**	**id * id**$	F→**id**
$E'T'	*** id**$	
$E'T'F*	*** id**$	T'→*FT'
$E'T'F	**id**$	
$E'T'**id**	**id**$	F→**id**
$E'T'	$	
$E'	$	T'→ ε
$	$	E'→ ε

Output corresponds to leftmost derivation

Construction of Predictive Parsing Table

- We will use two functions
  - FIRST(α), where α is any string of grammar symbols
    - Set of terminals that begin the strings derived from α
    - If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α)
  - FOLLOW(A), where A is a nonterminal
    - Set of terminals that can appear immediately to the right of nonterminal A in some sentential form
      - $S \overset{*}{\Rightarrow} \alpha A d \beta$ $S \overset{*}{\Rightarrow} \alpha A B d \beta$ with B→ε
      - $S \overset{*}{\Rightarrow} \alpha A$ then FOLLOW(S) belongs to FOLLOW(A)

Compute FIRST(X) for all grammar symbols X

- Apply the rules until no more terminals or ε can be added to any FIRST set
  - Rule 1
    - If X is terminal, then FIRST(X) = {X}
  - Rule 2
    - If X→ε is a production, then add ε to FIRST(X) FIRST(X)∪{ε}
  - Rule 3
    - If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1),… FIRST(Y_{i-1}); that is ($Y_1 Y \overset{*}{\Rightarrow} .Y_{i-1}$ ε). If ε is in FIRST(Y_j) for all j=1,2,…,k, then add ε to FIRST(X)

In other words, everything in FIRST(Y_1) except ε is surely in FIRST(X). If Y_1 does not derive ε then we add nothing more to FIRST(X). But, if it does then we add FIRST(Y_2) and so on…If all Y can derive ε, then add ε to FIRST(X)

Compute FOLLOW(X) for all nonterminals

- Apply the rules until nothing can be added to any FOLLOW set
  - Rule 1
    - Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker
  - Rule 2
    - If A→αBβ is a production, then everything in FIRST(β) except for ε is placed in FOLLOW(B)
  - Rule 3
    - If there is a production A→αB, or a production A→αBβ where FIRST(β) contains ε (β derives ε) then everything in FOLLOW(A) is included in FOLLOW(B)

Compute FIRST for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

Tip
FIRST(terminal) look the right side of the productions to find all terminals

FIRST(terminal)={terminal}

FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = {id}

Compute FIRST for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

Tip
FIRST(nonterminal) look the left side of the production and check what is after the arrow

FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = {id}

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}
FIRST(E') = FIRST(+) ∪ {ε} = {+, ε}
FIRST(T') = FIRST(*) ∪ {ε} = {*, ε}

Compute FOLLOW for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = { id }

FOLLOW(E)

Rule 1
FOLLOW(E) = {$}

Rule 2 (where does E appear on the right side but not as the last symbol?)

F → (E)
FOLLOW(E) = FOLLOW(E) U FIRST()) = {$,)}

Rule 3 (where does E appear on the right side as the last symbol?or what follows derives ε?)
Not applicable

RESULT: FOLLOW(E) = {$,)}

Compute FOLLOW for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = { id }

FOLLOW(E')

Rule 1
Not applicable

Rule 2
Not applicable

Rule 3
E → TE`
FOLLOW(E') = FOLLOW(E) = {$,)}
E` → +TE`
No need to check this one!

FOLLOW(E') = {$,)}

Compute FOLLOW for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = { id }

FOLLOW(T)

Rule 1

Not applicable

Rule 2

E → TE`

FOLLOW(T) = FIRST(E') = {+} (do not add ε)

E` → +TE`

No need to check this one!

Rule 3

E → TE`

FOLLOW(T) = FOLLOW(T) U FOLLOW(E) = {+, $,)}

E` → +TE`

FOLLOW(T) = FOLLOW(T) U FOLLOW(E') = {+, $,)}

FOLLOW(T) = {+,$,)}

Compute FOLLOW for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = { id }

FOLLOW(T')

Rule 1

Not applicable

Rule 2

Not applicable

Rule 3

T → FT`

FOLLOW(T') = FOLLOW(T)= {+, $,)}

T` → *FT`

No need to check this one!

FOLLOW(T') = {+,$,)}

Compute FOLLOW for Grammar G

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}
FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = { id }

FOLLOW(F)

Rule 1

Not applicable

Rule 2

T → FT`

FOLLOW(F) = FIRST(T') = {*}

T` → *FT`

No need to check this one!

Rule 3

T → FT`

FOLLOW(F) = FOLLOW(F) U FOLLOW(T) = {*, +, $,)}

T` → *FT`

FOLLOW(F) = FOLLOW(F) U FOLLOW(T') = {*, +, $,)}

FOLLOW(F) = {*,+,$,)}

Summary

GRAMMAR G

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

FIRST(+) = {+}
FIRST(*) = {*}
FIRST(( ) = { (}
FIRST( )) = {) }
FIRST(id) = {id}

FIRST(E) = {(, **id**}
FIRST(T) = {(, **id**}
FIRST(F) = {(, **id**}
FIRST(T') = {*, ε}
FIRST(E') = {+, ε}

FOLLOW(E) = {$,)}
FOLLOW(F) = {*,+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(T) = {+,$,)}
FOLLOW(E') = {$,)}

Construct a Predictive Parsing Table

- Idea
  - Suppose that A →α is a production with b in FIRST(α).
    - The parser will expand A by α when the current symbol is b.
    - The only complication is when α is ε or α derives ε.
      - In this case we expand A again by α if the current input symbol b is in FOLLOW(A) or if the $ on the input has been reached and $ is in FOLLOW(A)

Predictive Parsing Table

Input : Grammar G

Output: Parsing Table M

Method:

1) For each production A→α of the grammar, do steps 2 and 3
   2) For each terminal *b* in FIRST(α), add A→α to M[A, *b*]
   3) If ε is in FIRST(α), add A→α to M[A,c] for each terminal c in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add A→α to M[A,$]
4) Make each undefined entry of M be **error**

In-class Exercise

Derive the parsing table for grammar G using the predictive parsing table algorithm.

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

E → TE`
E` → +TE` | ε
T → FT`
T` → *FT`| ε
F → (E) | **id**

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'						
T						
T'						
F						

E → TE`

FIRST(TE') = FIRST(T) = {(, id}
add E → TE` to M[E,(]
add E → TE` to M[E,id]

Is ε in FIRST(T)?
No, so done for this production

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'				
T						
T'						
F						

E' → +TE`

FIRST(+TE') = FIRST(+) = {+}
add E' → +TE` to M[E',+]

Is ε in FIRST(+)?
No, so done for this production

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T						
T'						
F						

E' → ε

Is ε in FIRST(ε)?
Yes, so apply rule 3
FOLLOW(E') = {$,)}
add E' → ε to M[E', $]
add E' → ε to M[E',)]

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'						
F						

T→ TF'

FIRST(TF') = FIRST(T) = {(, id}
add T → FT` to M[T,(]
add T → FT` to M[T,id]

Is ε in FIRST(T)?
No, so done for this production

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'			T'→*FT'			
F						

T'→ *FT'

FIRST(*FT') =FIRST(*)= {*}
add T' → *FT' to M[T',*]

Is ε in FIRST(*)?
No, so done for this production

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F						

T' → ε

Is ε in FIRST(ε)?
Yes, so apply rule 3
FOLLOW(T') = {+,$,)}
add T' → ε to M[T', +]
add T' → ε to M[T', $]
add T' → ε to M[T',)]

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**					

F → id

FIRST(id) = {id}
add F → id to M[F,id]

Is ε in FIRST(id)?
No, so done for this production

Let's start the table

NON TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→ **id**			F→ (E)		

F → (E)

FIRST((E)) = FIRST(() = {(}
add F → (E) to M[F,(]

Is ε in FIRST(()?
No, so done for this production

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is set to be LL(1)
  - L
    - scanning inputs from left to right
  - L
    - producing leftmost derivation
  - 1
    - using one input symbol of lookahead at each step to make parsing action
- LL(1)
  - No ambiguity (use left-factoring)
  - No left recursion (eliminate left recursion)