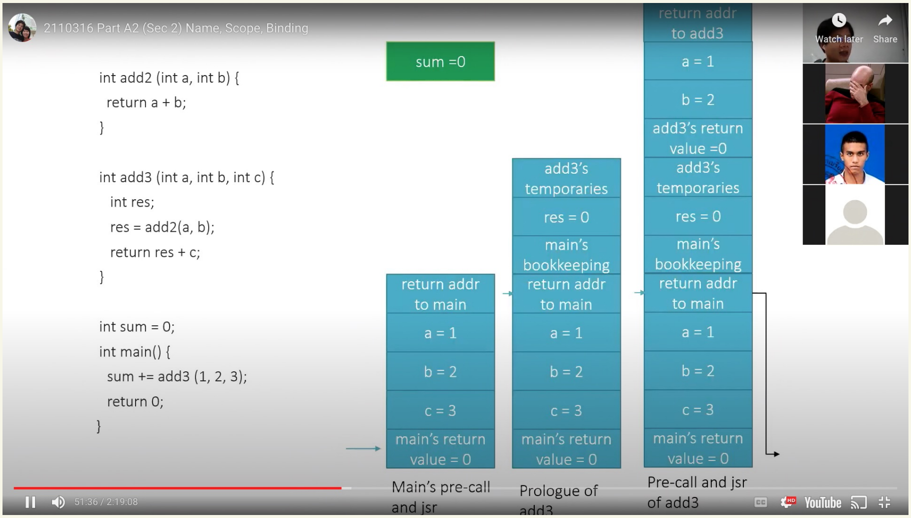


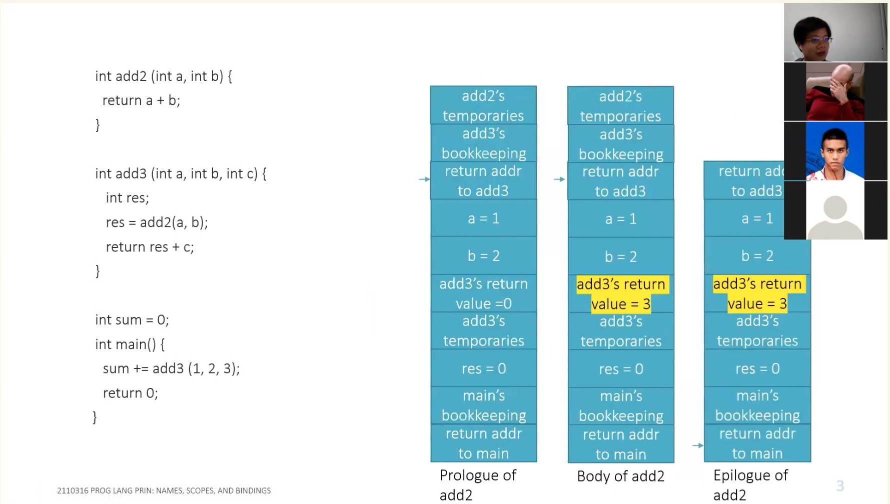
```
int add2 (int a, int b) {
  return a + b;
}

int add3 (int a, int b, int c) {
   int res;
   res = add2(a, b);
   return res + c;
}

int sum = 0;
int main() {
  sum += add3 (1, 2, 3);
  return 0;
}
```

sum =0

Main's pre-call and jsr: return addr to main | a = 1 | b = 2 | c = 3 | main's return value = 0

Prologue of add3: add3's temporaries | res = 0 | main's bookkeeping | return addr to main | a = 1 | b = 2 | c = 3 | main's return value = 0

Pre-call and jsr of add3: return addr to add3 | a = 1 | b = 2 | add3's return value =0 | add3's temporaries | res = 0 | main's bookkeeping | return addr to main | a = 1 | b = 2 | c = 3 | main's return value = 0

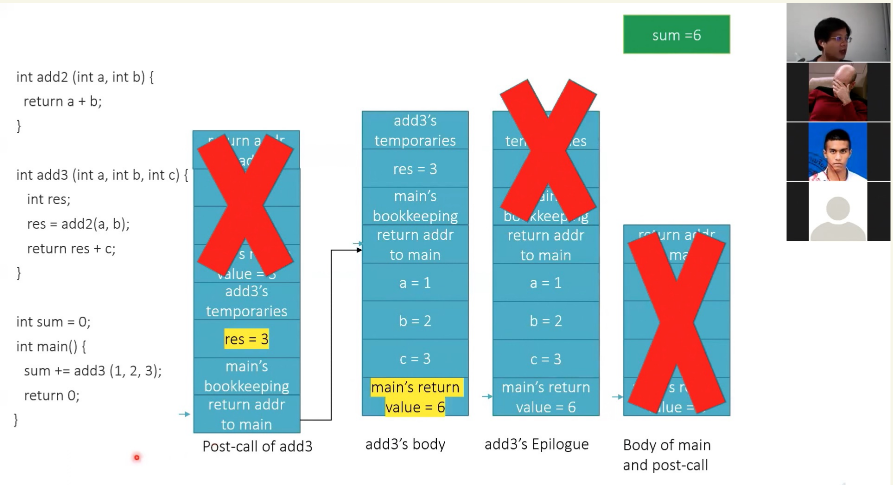


```
int add2 (int a, int b) {
  return a + b;
}

int add3 (int a, int b, int c) {
   int res;
   res = add2(a, b);
   return res + c;
}

int sum = 0;
int main() {
  sum += add3 (1, 2, 3);
  return 0;
}
```

Prologue of add2: add2's temporaries | add3's bookkeeping | return addr to add3 | a = 1 | b = 2 | add3's return value =0 | add3's temporaries | res = 0 | main's bookkeeping | return addr to main

Body of add2: add2's temporaries | add3's bookkeeping | return addr to add3 | a = 1 | b = 2 | add3's return value = 3 | add3's temporaries | res = 0 | main's bookkeeping | return addr to main

Epilogue of add2: return add to add3 | a = 1 | b = 2 | add3's return value = 3 | add3's temporaries | res = 0 | main's bookkeeping | return addr to main



```
int add2 (int a, int b) {
  return a + b;
}

int add3 (int a, int b, int c) {
   int res;
   res = add2(a, b);
   return res + c;
}

int sum = 0;
int main() {
  sum += add3 (1, 2, 3);
  return 0;
}
```

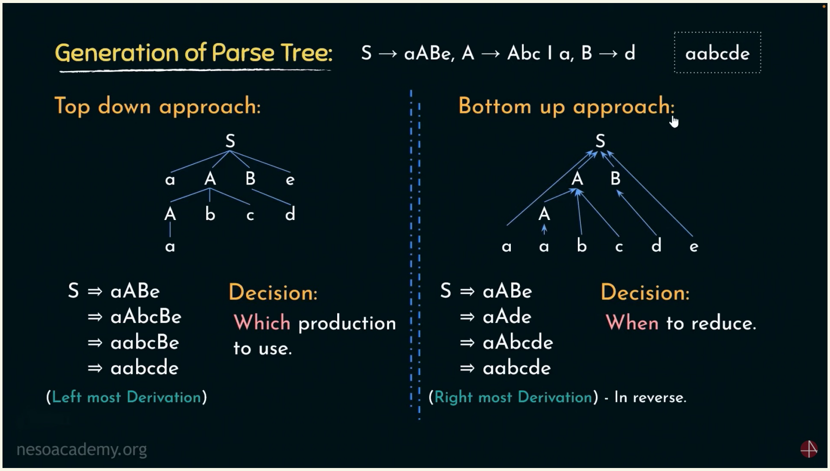
sum =6

Post-call of add3: add3's temporaries | res = 3 | main's bookkeeping | return addr to main

add3's body: add3's temporaries | res = 3 | main's bookkeeping | return addr to main | a = 1 | b = 2 | c = 3 | main's return value = 6

add3's Epilogue: return addr to main | a = 1 | b = 2 | c = 3 | main's return value = 6

Body of main and post-call



Generation of Parse Tree:

S → aABe, A → Abc | a, B → d

aabcde

Top down approach:

$S \Rightarrow aABe$
$\Rightarrow aAbcBe$
$\Rightarrow aabcBe$
$\Rightarrow aabcde$

(Left most Derivation)

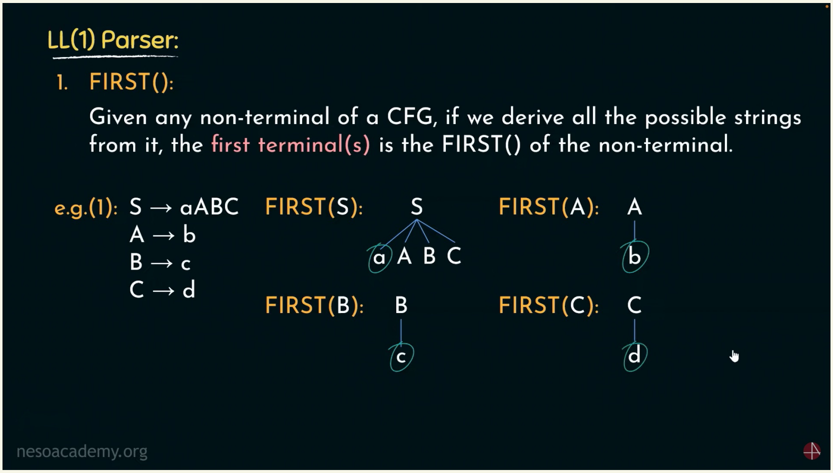
Decision: Which production to use.

Bottom up approach:

$S \Rightarrow aABe$
$\Rightarrow aAde$
$\Rightarrow aAbcde$
$\Rightarrow aabcde$

(Right most Derivation) - In reverse.

Decision: When to reduce.



LL(1) Parser:

1. FIRST():

Given any non-terminal of a CFG, if we derive all the possible strings from it, the first terminal(s) is the FIRST() of the non-terminal.

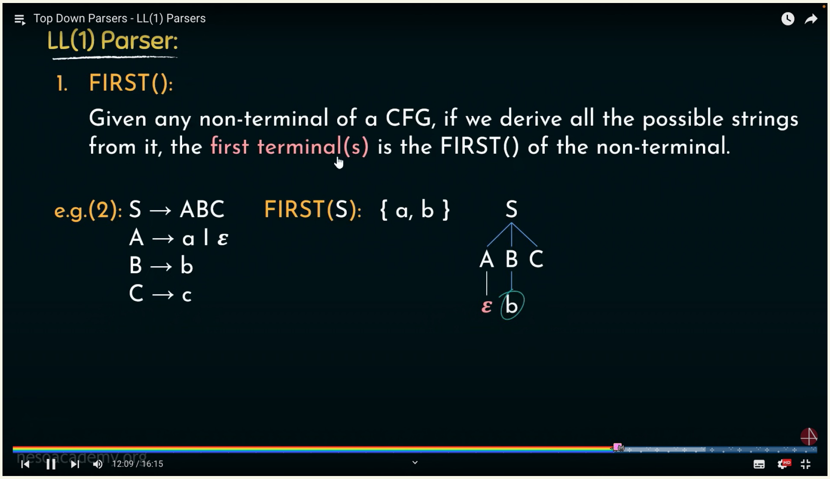
e.g.(1): S → aABC
A → b
B → c
C → d

FIRST(S):

FIRST(A):

FIRST(B):

FIRST(C):



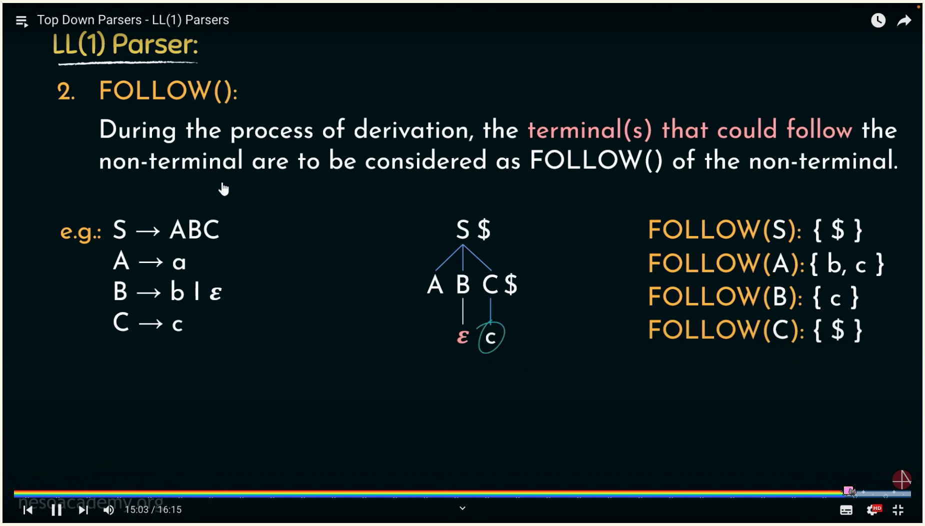
LL(1) Parser:

1. FIRST():

Given any non-terminal of a CFG, if we derive all the possible strings from it, the first terminal(s) is the FIRST() of the non-terminal.

e.g.(2): S → ABC
A → a | ε
B → b
C → c

FIRST(S): { a, b }



Top Down Parsers - LL(1) Parsers

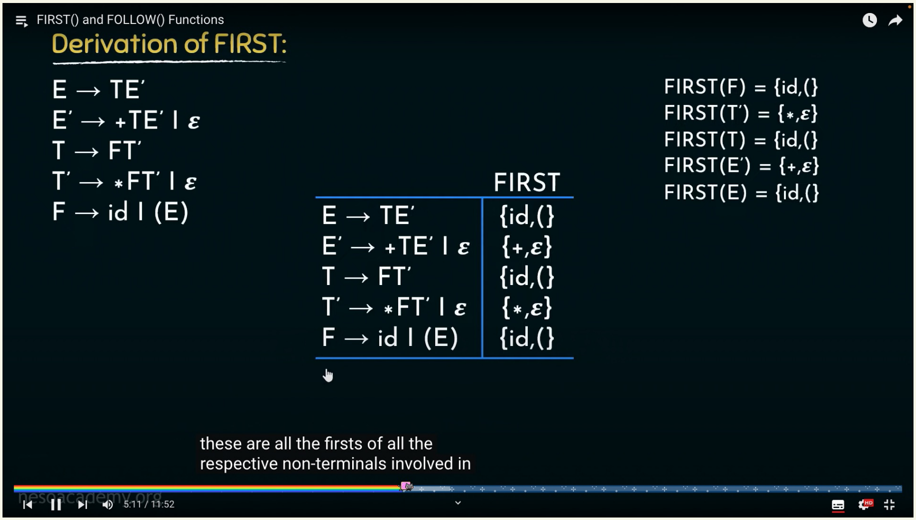
LL(1) Parser:

2. FOLLOW():

During the process of derivation, the terminal(s) that could follow the non-terminal are to be considered as FOLLOW() of the non-terminal.

e.g.: $S \rightarrow ABC$
$A \rightarrow a$
$B \rightarrow b \mid \varepsilon$
$C \rightarrow c$

FOLLOW(S): { $ }
FOLLOW(A): { b, c }
FOLLOW(B): { c }
FOLLOW(C): { $ }



FIRST() and FOLLOW() Functions

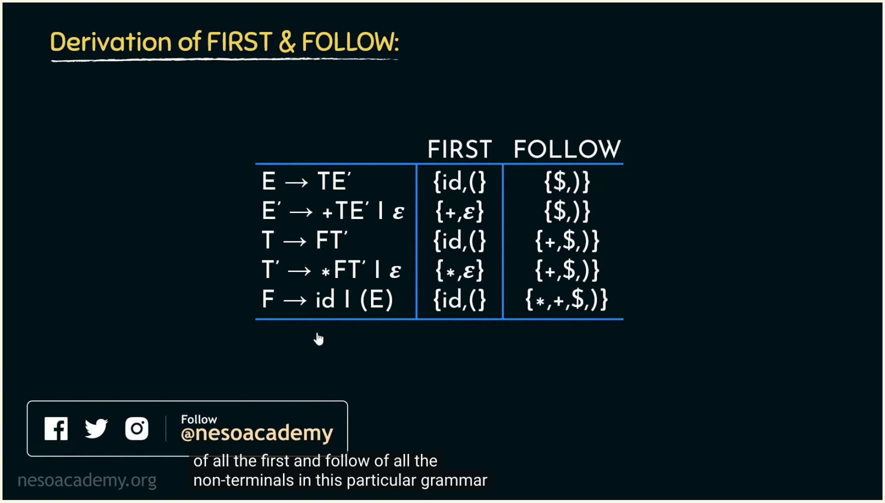
Derivation of FIRST:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

	FIRST
$E \rightarrow TE'$	{id,(}
$E' \rightarrow +TE' \mid \varepsilon$	$\{+,\varepsilon\}$
$T \rightarrow FT'$	{id,(}
$T' \rightarrow *FT' \mid \varepsilon$	$\{*,\varepsilon\}$
$F \rightarrow id \mid (E)$	{id,(}

FIRST(F) = {id,(}
FIRST(T′) = $\{*,\varepsilon\}$
FIRST(T) = {id,(}
FIRST(E′) = $\{+,\varepsilon\}$
FIRST(E) = {id,(}

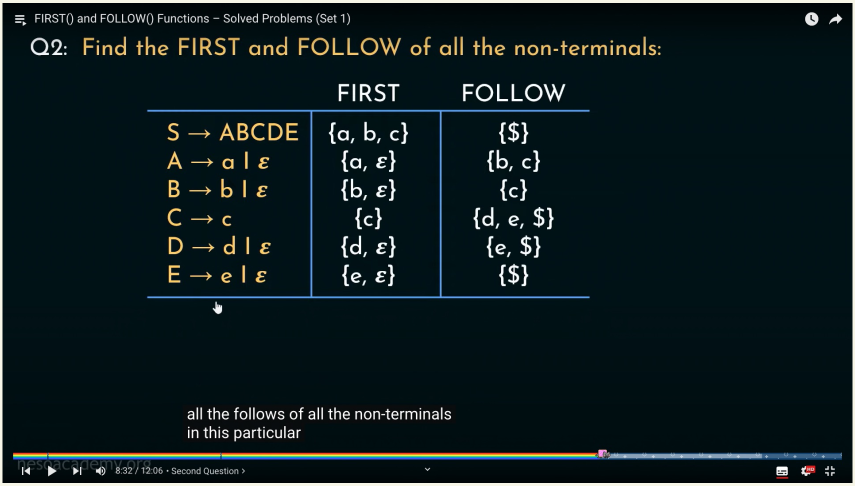
these are all the firsts of all the
respective non-terminals involved in



Derivation of FIRST & FOLLOW:

	FIRST	FOLLOW
$E \rightarrow TE'$	{id,(}	{$,)}
$E' \rightarrow +TE' \mid \varepsilon$	$\{+,\varepsilon\}$	{$,)}
$T \rightarrow FT'$	{id,(}	{+,$,)}
$T' \rightarrow *FT' \mid \varepsilon$	$\{*,\varepsilon\}$	{+,$,)}
$F \rightarrow id \mid (E)$	{id,(}	{*,+,$,)}

of all the first and follow of all the
non-terminals in this particular grammar



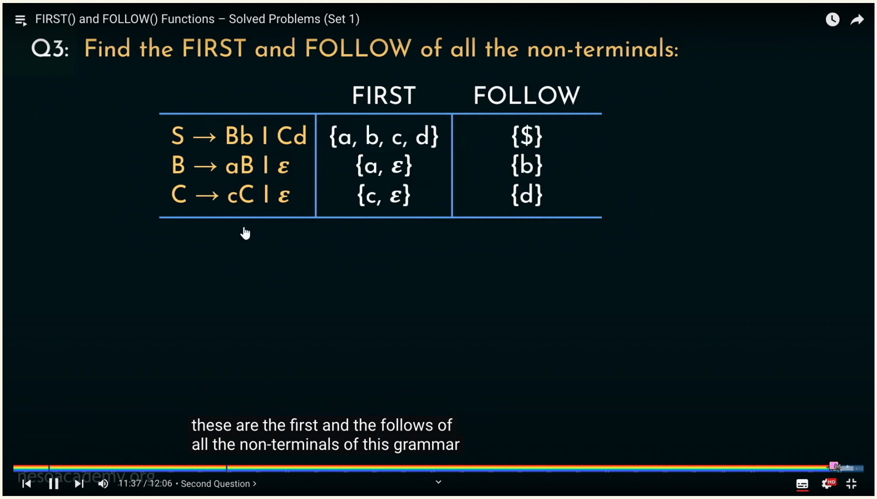
FIRST() and FOLLOW() Functions – Solved Problems (Set 1)

Q2: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → ABCDE	{a, b, c}	{$}
A → a | ε	{a, ε}	{b, c}
B → b | ε	{b, ε}	{c}
C → c	{c}	{d, e, $}
D → d | ε	{d, ε}	{e, $}
E → e | ε	{e, ε}	{$}

all the follows of all the non-terminals
in this particular

8:32 / 12:06 • Second Question ›



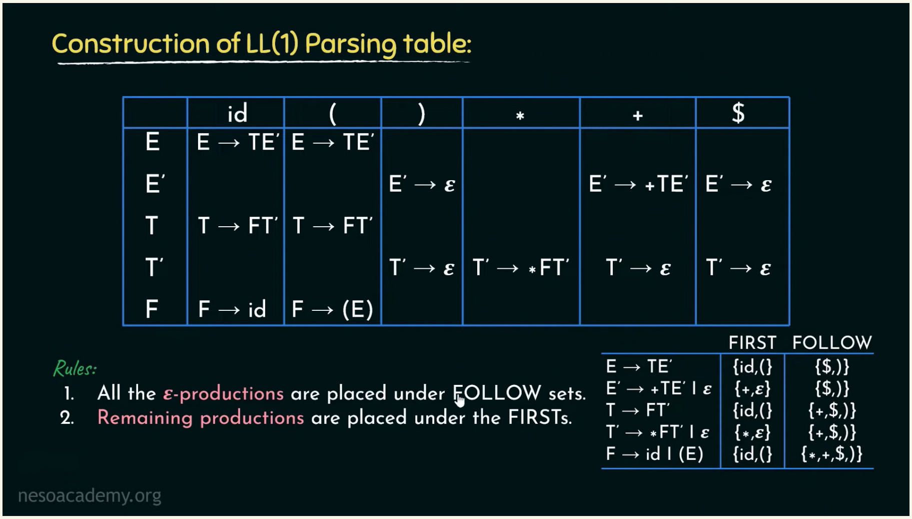
FIRST() and FOLLOW() Functions – Solved Problems (Set 1)

Q3: Find the FIRST and FOLLOW of all the non-terminals:

	FIRST	FOLLOW
S → Bb | Cd	{a, b, c, d}	{$}
B → aB | ε	{a, ε}	{b}
C → cC | ε	{c, ε}	{d}

these are the first and the follows of
all the non-terminals of this grammar

11:37 / 12:06 • Second Question ›



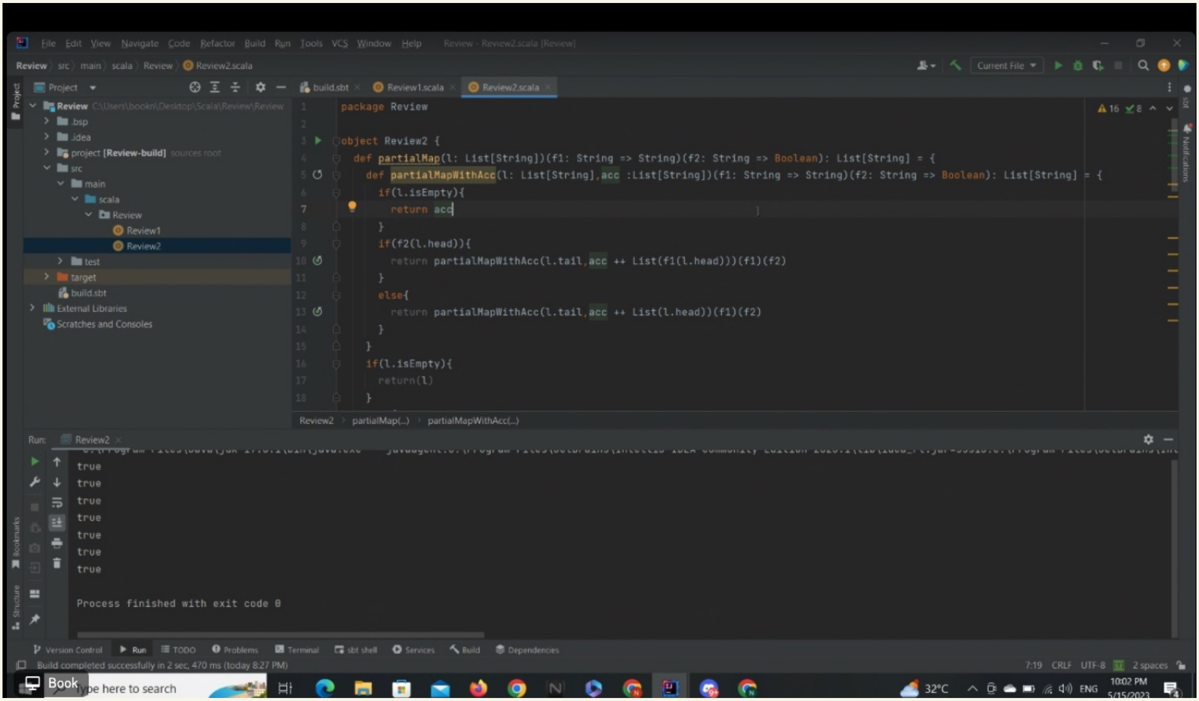
Construction of LL(1) Parsing table:

	id	(	)	*	+	$
E	E → TE′	E → TE′				
E′			E′ → ε		E′ → +TE′	E′ → ε
T	T → FT′	T → FT′				
T′			T′ → ε	T′ → *FT′	T′ → ε	T′ → ε
F	F → id	F → (E)				

Rules:

1. All the ε-productions are placed under FOLLOW sets.
2. Remaining productions are placed under the FIRSTs.

	FIRST	FOLLOW
E → TE′	{id,(}	{$,)}
E′ → +TE′ | ε	{+,ε}	{$,)}
T → FT′	{id,(}	{+,$,)}
T′ → *FT′ | ε	{*,ε}	{+,$,)}
F → id | (E)	{id,(}	{*,+,$,)}



```scala
package Review

object Review2 {
  def partialMap(l: List[String])(f1: String => String)(f2: String => Boolean): List[String] = {
    def partialMapWithAcc(l: List[String],acc :List[String])(f1: String => String)(f2: String => Boolean): List[String] = {
      if(l.isEmpty){
        return acc
      }
      if(f2(l.head)){
        return partialMapWithAcc(l.tail,acc ++ List(f1(l.head)))(f1)(f2)
      }
      else{
        return partialMapWithAcc(l.tail,acc ++ List(l.head))(f1)(f2)
      }
    }
    if(l.isEmpty){
      return(l)
    }
```

```
true
true
true
true
true
true
true

Process finished with exit code 0
```

Build completed successfully in 2 sec, 470 ms (today 8:27 PM)