# PRACTISE PROBLEMS AND SOLUTIONS

## LL(1) PARSER

### 1. Removing Left Recursion:

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha \mid \beta$$

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$

Of course, there may be more than one left-recursive part on the right-hand side. The general rule is to replace:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \in$$

***Example for Direct Left Recursion:***

Consider the following grammar.

E --> E + T | T

is left-recursive with "E" playing the role of "A", "+ T" playing the role of α, and "T" playing the role of β. Introducing the new nonterminal E', the production can be replaced by:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \in$

***Example for Indirect Left Recursion:***

$A \rightarrow B\ x\ y \mid x$

$B \rightarrow C\ D$

$C \rightarrow A \mid c$

$D \rightarrow d$

is indirectly recursive because

A ==> B x y ==> C D x y ==> A D x y.

That is, A ==> ... ==> Aγ where γ is D x y.

## 2. Left Factoring:

Many grammars have the same prefix symbols at the beginning of alternative right sentential forms for a nonterminal:

$$A \rightarrow \alpha\, \beta \mid \alpha\, \gamma$$

We replace this production with the following:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta \mid \gamma$$

***Example:***

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

The equivalent left factored grammar is

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \in$

$E \rightarrow b$

## 3. Calculating FIRST Function:

**Rule-1:**

For a production rule $X \rightarrow \in$,

$First(X) = \{ \in \}$

**Rule-2:**

For any terminal symbol 'a',

$First(a) = \{ a \}$

**Rule-3:**

For a production rule $X \rightarrow Y_1Y_2Y_3$,

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$
- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

***Example:***

Consider the grammar given below:

$E \rightarrow TX$

$X \rightarrow +E$

$X \rightarrow \varepsilon$

$T \rightarrow intY$

$T \rightarrow (E)$

$Y \rightarrow *T$

$Y \rightarrow \varepsilon$

Solution:

By Rule-1, if $X \rightarrow \in$ then add $\in$ to First(X).

| **Symbol** | **First** |
|---|---|
| ( | |
| ) | |
| + | |
| * | |
| int | |
| Y | $\in$ |
| X | $\in$ |
| T | |
| E | |

By Rule-2, The First of a terminal is the same terminal.

| **Symbol** | **First** |
|---|---|
| ( | ( |
| ) | ) |
| + | + |
| * | * |
| int | int |
| Y | $\in$ |
| X | $\in$ |
| T | |
| E | |

$E \rightarrow TX$

By Rule-3, if $X \rightarrow Y_1Y_2Y_3$, where $Y_1$, $Y_2$, $Y_3$, are non-terminals then First(X) = First($Y_1$)

So,

First(T)= {int, (}

T → intY        because First(int)=int

T → (E)        because First (( ) = (

Now we can go back and do First(E)

First(E)= {int, (}

We don't consider the first of X because T is not nullable.

First(X)= {ε, +}

X → +E

First (+) =+X → ε      handled at step 1

First(Y)= {ε, *}

Y → *T

First (*) =*

Y → ε handled at step 2

Put it all together

| **Symbol** | **First** |
|---|---|
| ( | ( |
| ) | ) |
| + | + |
| * | * |
| int | int |
| Y | ∈,* |
| X | ∈,+ |
| T | int, ( |
| E | int, ( |

**4. Calculating Follow Function:**

**Rule-1:**

For the start symbol S, place $ in Follow(S).

**Rule-2:**

For any production rule A → αB,

Follow(B) = Follow(A)

**Rule-3:**

For any production rule $A \rightarrow \alpha B \beta$,

- If $\in \notin First(\beta)$, then $Follow(B) = First(\beta)$
- If $\in \in First(\beta)$, then $Follow(B) = \{ First(\beta) - \in \} \cup Follow(A)$

The Follow functions for the above grammar is as follows:

***Step-1:***

a) By Rule-1: Follow(E)= {$}    E is our start symbol

| Symbol | First | Follow |
|---|---|---|
| ( | ( | N/A |
| ) | ) | |
| + | + | |
| * | * | |
| int | int | |
| Y | ∈,* | |
| X | ∈,+ | |
| T | int, ( | |
| E | int, ( | $ |

***Step-2:***

$E \rightarrow TX$

a.) Follow(T) contains (atleast) the First(X)= {∈, +} – {∈} = {+}

$T \rightarrow (E)$

b.) Follow(E) contains (atleast) the First ( ) ) = {)}

So now Follow(E) is {), $} (From step 1a)

The table so far:

| Symbol | First | Follow |
|---|---|---|
| ( | ( | N/A |
| ) | ) | |
| + | + | |
| * | * | |
| int | int | |
| Y | ∈,* | |
| X | ∈,+ | |
| T | int, ( | + |
| E | int, ( | ), $ |

***Step-3:***

E → TX

a.) Follow(X) contains (atleast) Follow(E)= {), $} (from step 2b)

$\varepsilon \in$First(X) so:

b.) Follow(T) contains (atleast) Follow(E)= {), $, +} (from step 2a and 2b)

X → +E

c.) Follow(E) contains (atleast) Follow(X)= {), $} (from step 3a)

T → intY

d.) Follow(Y) contains (atleast) Follow(T)= {), $, +} (from step 3b)

Y → *T

e.) Follow(T) contains (atleast) Follow(Y)= {), $} (from step 3d)

We do this whole process again until no more additions happen:

E → TX

f.) Follow(X) contains (atleast) Follow(E)= {), $} (no change)

$\varepsilon \in$First(X) so:

g.) Follow(T) contains (atleast) Follow(E)= {), $, +} (no change)

X → +E

h.) Follow(E) contains (atleast) Follow(X)= {), $} (no change)

T → intY

i.) Follow(Y) contains (atleast) Follow(T)= {), $, +} (no change)

Y → *T

j.) Follow(T) contains (atleast) Follow(Y)= {), $} (no change)

<table>
<tr><th>Symbol</th><th>First</th><th>Follow</th></tr>
<tr><td>(</td><td>(</td><td rowspan="5">N/A</td></tr>
<tr><td>)</td><td>)</td></tr>
<tr><td>+</td><td>+</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>int</td><td>int</td></tr>
<tr><td>Y</td><td>∈,*</td><td>), $, +</td></tr>
<tr><td>X</td><td>∈,+</td><td>), $</td></tr>
<tr><td>T</td><td>int, (</td><td>), $, +</td></tr>
<tr><td>E</td><td>int, (</td><td>), $</td></tr>
</table>

**PRACTISE PROBLEM**

## 1. Construct LL(1) Parsing Table for the following grammar

```
E -> E + T | T
T -> T * F | F
F -> (E) | int
```

**Solution:**

**Steps to construct LL(1) Parsing Table:**

1. Remove Left Recursion and Left Factoring and Write the equivalent grammar.
2. Find First and Follow for each non-terminal in the grammar written in step-1.
3. Construct LL(1) parsing table using the First and Follow Functions.
4. Do input checking for the string.

***Step-1: Removal of Left Recursion and Left Factoring:***

The grammar doesnot contain left factoring. But it has left recursion on E→E+T and T→T*F.

Consider E→E+T | T:

After removing left recursion, the equivalent grammar is

```
E  -> T E'
E' -> + T E' | ∈.
```

Similarly, for T→T*F | F,

```
T  -> F T'
T' -> * F T' | ∈
```

We get the following grammar:

```
E  -> T E'
E' -> + T E' | ∈
T  -> F T'
T' -> * F T' | ∈
F  -> (E) | int
```

Note, the F production didn't get changed at all. That's because F didn't appear on the leftmost position of any of the productions on the right-hand side of the arrow.

***Step-2: Find FIRST and FOLLOW positions***

What is the **FIRST**(E)? What are the terminals that can appear at the beginning of the stream when we're looking for an E? Well, E -> T E', so whatever occurs at the beginning of E will be the same as what happens at the beginning of T.

```
FIRST(E) => FIRST(T)
```

**FIRST**(E') is easy, we have the terminal +, and ∈.

```
FIRST(E') = { +, ∈ }
```

And we'll continue with the others:

```
FIRST(T) => FIRST(F)
FIRST(T') = { *, ∈ }
FIRST(F) = { (, int }
```

**FIRST**(F) is just the set of terminals that are at the beginnings of its productions.

So, to sum up:

```
FIRST(E) = { (, int }
FIRST(E') = { +, ∈ }
FIRST(T) = { (, int }
FIRST(T') = { *, ∈ }
FIRST(F) = { (, int }
```

What is FOLLOW(E)? As E is the starting non-terminal, add $ to FOLLOW(E). Look on all of the right-hand sides (after the arrow) of all of the productions in the grammar. What terminals appear on the right of the E?  It's a ). So,

```
FOLLOW(E) = { $, ) }
```

How about **FOLLOW**(E')? There's nothing after either of the E's in the grammar. If we had derived E' from E in `E -> TE'`, then whatever follows the E is the same as whatever follows the E'. For the other production, we get that whatever follows E' is the same as whatever follows E'. That's a tautology.

```
FOLLOW(E') => FOLLOW(E)
```

Now, let's do the rest:

```
FOLLOW(T) = ?
```

T is always followed by E'. So, whatever terminals begin E' must be the terminals that follow T. So this means that **FOLLOW**(T) => **FIRST**(E'). It seems a bit weird, but since when we're done, we'll have no more non-terminals in our stream, so only the terminals are important to look at. Since T will disappear into some terminals, those are the same that will tell us that we're starting to do E'.

```
FOLLOW(T) => FIRST(E')
FOLLOW(T') => FOLLOW(T)
FOLLOW(F) => FIRST(T')
```

If we solve this system of equations, we get:

```
FOLLOW(S) = { $ }
FOLLOW(E) = { $, ) }
FOLLOW(E') = { $, ) }
FOLLOW(T) = { +, ∈ }
```

Hold on! What's that ∈ doing there? We can't have that. Where did it come from? Ah, it came from including **FIRST**(E'). Well, if E' -> ∈, then whatever terminals follow E' (**FOLLOW**(E')) will be the terminals we're looking for. (These are not the droids you're looking for.) These are $ and ). Let's add those to our list:

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

There's that ∈ again. If T' -> ∈, then we want to add **FOLLOW**(T') in there.

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

Now, Let's fill in the parsing table. Each of the cells should be filled in with the production that the non-terminal on the left column takes when we see the terminal on the top in our input stream.

| | + | * | ( | ) | int | $ |
|---|---|---|---|---|---|---|
| E | | | | | | |
| E' | | | | | | |
| T | | | | | | |
| T' | | | | | | |
| F | | | | | | |

Our FIRST calculations have told us exactly what we need! They tell us what terminals we're allowed to see on the input stream when we're looking for one of those non-terminals.

We'll reprint the FIRST equations to remind ourselves what the values were:

```
FIRST(E)  = { (, int }
FIRST(E') = { +, ∈ }
FIRST(T)  = { (, int }
FIRST(T') = { *, ∈ }
FIRST(F)  = { (, int }
```

So, let's fill in the first row:

| | + | * | ( | ) | int | $ |
|---|---|---|---|---|---|---|
| E | | | E -> T E' | | E -> T E' | |
| E' | | | | | | |
| T | | | | | | |
| T' | | | | | | |
| F | | | | | | |

Now, the second row:

FIRST(E') contains + and ∈. Remember what we did in the FOLLOW function when we found the ∈? We took the FOLLOW of that production, FOLLOW(E'), which turned out to be ) and $. So, let's take that ∈ production whenever we see ) or $.

| | + | * | ( | ) | int | $ |
|---|---|---|---|---|---|---|
| E | | | E -> T E' | | E -> T E' | |
| E’ | E′ -> +T E' | | | E′ -> ∈ | | E′ -> ∈ |
| T | | | | | | |
| T’ | | | | | | |
| F | | | | | | |

Similarly, let’s fill in the remaining rows.

| | + | * | ( | ) | int | $ |
|---|---|---|---|---|---|---|
| E | | | E -> T E' | | E -> T E' | |
| E’ | E′ -> +T E' | | | E′ -> ∈ | | E′ -> ∈ |
| T | | | T -> F T' | | T -> F T' | |
| T’ | T′ -> ∈ | T' -> * F T' | | T′ -> ∈ | | T′ -> ∈ |
| F | | | F -> (E) | | F -> int | |

***Step-4: Input Checking***

Let’s take the string 3+5*7

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |

Now, look at the table. If we have an E on the stack, and our input is a 3 (an integer), we pick the transition E -> T E'. This means, we pop E off the stack and replace it with T E'.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E’T | 3+5*7$ | Replace T by T -> F T' |

Note, we haven't done anything to the input yet. We're still only dealing with non-terminals. Now we have a T on the top of the stack, and an integer on the input. Yes, we go to T -> F T'.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E’T | 3+5*7$ | Replace T by T -> F T' |
| $E’T’F | 3+5*7$ | Replace F by F -> int |

Let's continue another step. If we have an F on the input stack and see an integer, we do F -> int.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E’T | 3+5*7$ | Replace T by T -> F T' |
| $E’T’F | 3+5*7$ | Replace F by F -> int |

| | | |
|---|---|---|
| $E'T'int | 3+5*7$ | Pop int and 3 |

Now we have an int at the top of the stack, and an int beginning the input stream. We pop both of these off.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E'T | 3+5*7$ | Replace T by T -> F T' |
| $E'T'F | 3+5*7$ | Replace F by F -> int |
| $E'T'int | 3+5*7$ | Pop int and 3 |
| $E'T' | +5*7$ | Replace T' by T' -> ∈ |

Now we have a T' on the top of the stack, and a + on the input. T' -> ∈. This means we pop T' off our stack and do nothing to the input.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E'T | 3+5*7$ | Replace T by T -> F T' |
| $E'T'F | 3+5*7$ | Replace F by F -> int |
| $E'T'int | 3+5*7$ | Pop int and 3 |
| $E'T' | +5*7$ | Replace T' by T' -> ∈ |
| $E' | +5*7$ | Replace E' by E' -> + T E'. |

We now have an E' on the top of the stack, and a + on the input. Now we do the production E' -> + T E'.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E'T | 3+5*7$ | Replace T by T -> F T' |
| $E'T'F | 3+5*7$ | Replace F by F -> int |
| $E'T'int | 3+5*7$ | Pop int and 3 |
| $E'T' | +5*7$ | Replace T' by T' -> ∈ |
| $E'T+ | +5*7$ | Replace E' by E' -> + T E'. |
| $E'T+ | +5*7$ | Pop + from both |

And again, we have a terminal at the top of the stack, and a matching terminal on the input stream. So we pop them both, and continue.

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E'T | 3+5*7$ | Replace T by T -> F T' |
| $E'T'F | 3+5*7$ | Replace F by F -> int |
| $E'T'int | 3+5*7$ | Pop int and 3 |
| $E'T' | +5*7$ | Replace T' by T' -> ∈ |
| $E'T+ | +5*7$ | Replace E' by E' -> + T E'. |
| $E'T | 5*7$ | |

At this point, we're in a pretty similar position to step 2. Let's do a few steps and the result is

| Stack | Input | Action |
|---|---|---|
| $E | 3+5*7$ | Replace E by E -> T E' |
| $E'T | 3+5*7$ | Replace T by T -> F T' |
| $E'T'F | 3+5*7$ | Replace F by F -> int |

| | | |
|---|---|---|
| $E’T’int | 3+5*7$ | Pop int and 3 |
| $E’T’ | +5*7$ | Replace T’ by T' -> ∈ |
| $E’T+ | +5*7$ | Replace E’ by E' -> + T E'. |
| $E’T | 5*7$ | Pop + from both |
| $E’T | 5*7$ | Replace T by T->FT’ |
| $E’T’F | 5*7$ | Replace F by F -> int |
| $E’T’int | 5*7$ | Pop int and 5 |
| $E’T’ | *7$ | Replace T’ by T' -> *FT’ |
| $E’T’F* | *7$ | Pop * from both |
| $E’T’F | 7$ | Replace F by F -> int |
| $E’T’int | 7$ | Pop int and 7 |
| $E’T’ | $ | Replace T’ by T' -> ∈ |
| $E’ | $ | Replace E’ by E' -> ∈ |
| $ | $ | **Accepted** |