LL(k) grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose (between α & β) the right production to expand A in the parser tree at each step

How can it do it?

Guided by the input string!

LL(k) grammars

- An LL(k) grammar is a context-free grammar that can be parsed by predictive parser (no backtracking) which reads the input Left to right and construct a Leftmost derivation looking to k symbols in the input string

- A language that has a LL(k) grammar is said an LL(k) language

- LL(k) is a grammar that can predict the right production to apply with lookhead of most k symbols

$$LL(0) \subset LL(1) \subset LL(2) \subset ... \subset LL(*)$$

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

The parser will decide what to choose on the base of the input and of the following sets:

- The FIRST set: FIRST(α) with $\alpha \in (T \cup NT)^*$
- The FOLLOW set: FOLLOW(A) with $A \in NT$

The FIRST set

FIRST sets

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

We will learn how to compute it!

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This is almost correct See the next slide

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

Example

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	<u>(</u> Expr <u>)</u>
10		|	<u>number</u>
11		|	<u>id</u>

first(Expr')={ +,-, ε}

But what else I need to consider?

{ eof,) }

Predictive Parsing

What about ε-productions?

$\Rightarrow$ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too, where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

Later we will learn how to compute them!

Define FIRST$^+$($A \rightarrow \alpha$) as

- FIRST(α) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$

Predictive Parsing

Given a grammar that has the LL(1) property

- Can write a simple routine to recognize each lhs
- Code is both simple & fast

One kind of predictive parser is the recursive descent parser.

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$FIRST^+(A \rightarrow \beta_i) \cap FIRST^+(A \rightarrow \beta_j) = \varnothing$ if $i \neq j$

```
/* find an A */
if (current_word ∈ FIRST+(A→β1))
 recognise a β1 and return true
else if (current_word ∈ FIRST+(A→β2))
   recognise a β2 and return true
else if (current_word ∈ FIRST+(A→β3))
   recognise a β3 and return true
else
   report an error and return false
```

Of course, there is more detail to "recognize a β_i" a procedure for each nonterminal

Recursive Descent Parsing

Recall the expression grammar, after transformation

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	<u>(</u> Expr <u>)</u>
10		|	<u>number</u>
11		|	<u>id</u>

This produces a parser with six <u>mutually recursive</u> routines:

- Goal
- Expr
- EPrime
- Term
- TPrime
- Factor

Each recognizes one NT or T

The term <u>descent</u> refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

they return a boolean

```
Goal( )
   token ← next_token( );
   if (Expr( ) = true & token = EOF)
      then next compilation step;
      else
          report syntax error;
          return false;

Expr( )
  if (Term( ) = false)
     then return false;
     else return Eprime( );
```

0	Goal	→	Expr
1	Expr	→	Term Expr'

Recursive Descent Parsing II

```
Eprime( )
    if (token = '+' OR  token = '-' )
    then begin
        token ← next_token( );
        if Term( ) then return Eprime ();
        else report syntax error;
        end;
    else if  (token = ')' OR  token = EOF )
        then return true;
        else  return false;
```

2	Expr'	→	+ Term Expr'
3			| - Term Expr'
4			| ε

FIRST+(Expr'-> + Term Expr')={+}
FIRST+(Expr'-> - Term Expr')={-}
FIRST+(Expr'-> ε)={EOF ,) }

***Term*, & *Tprime* follow the same basic lines**

Recursive Descent Parsing III

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
      token ← next_token( );
      return true;
  else if (token = Lparen)
      token ← next_token( );
      if (Expr( ) = true & token = Rparen) then
          token ← next_token( );
          return true;
  // fall out of if statement
  report syntax error;
      return false;
```

9	Factor	→	(Expr)
10		|	<u>number</u>
11		|	<u>id</u>

FIRST+(Factor-> (Expr))={(}

FIRST+(Factor-> number)= number}

FIRST+(Factor-> id)={id}

looking for Number, Identifier, or "(", found token instead, or failed to find Expr or ")" after "("

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - Context-free grammars ✓
- Top-down parsers
  - Algorithm & its problem with left recursion ✓
  - Ambiguity ✓
  - Left-recursion removal ✓
- Predictive top-down parsing
  - The LL(1) condition ✓
  - Simple recursive descent parsers ✓
  - Transforming a grammar to be LL(1)
  - First and Follow sets
  - Table-driven LL(1) parsers

What If My Grammar Is Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) grammar?

- In general, the answer is no, however, sometime it is yes

Assume a grammar G with productions $A \rightarrow \alpha\, \beta_1$ and $A \rightarrow \alpha\, \beta_2$

- If α derives anything other than ε, then

$$\text{FIRST}^+(A \rightarrow \alpha\, \beta_1) \cap \text{FIRST}^+(A \rightarrow \alpha\, \beta_2) \neq \varnothing$$

- And the grammar is not LL(1)
- If we pull the common prefix, α, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha\, A',\; A' \rightarrow \beta_1 \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^+(A' \rightarrow \beta_1) \cap \text{FIRST}^+(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

What If My Grammar Is Not LL(1) ?

Left Factoring

For each nonterminal A

find the longest prefix α common to 2 or more alternatives for A

if $\alpha \neq \varepsilon$ then

replace all of the productions

$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \alpha\,\beta_3 \mid \ldots \mid \alpha\,\beta_n \mid \gamma$

with

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

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

Repeat until no nonterminal has alternative rhs' with a common prefix

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists

Left Factoring Example

Consider a simple right-recursive expression grammar

0	Goal	$\rightarrow$	Expr
1	Expr	$\rightarrow$	Term + Expr
2		|	Term - Expr
3		|	Term
4	Term	$\rightarrow$	Factor * Term
5		|	Factor / Term
6		|	Factor
7	Factor	$\rightarrow$	<u>number</u>
8		|	<u>id</u>

To choose between 1, 2, & 3, an LL(1) parser must look past the <u>number</u> or <u>id</u> to *see* the operator.

FIRST⁺(1) = FIRST⁺(2) = FIRST⁺(3)

and

FIRST⁺(4) = FIRST⁺(5) = FIRST⁺(6)

Let's left factor this grammar.

Left Factoring Example

After Left Factoring, we have

0	Goal	$\rightarrow$	Expr
1	Expr	$\rightarrow$	Term Expr'
2	Expr'	$\rightarrow$	+ Expr
3		|	- Expr
4		|	ε
5	Term	$\rightarrow$	Factor Term'
6	Term'	$\rightarrow$	* Term
7		|	/ Term
8		|	ε
9	Factor	$\rightarrow$	<u>number</u>
10		|	<u>id</u>

Clearly,

FIRST⁺(2), FIRST⁺(3), & FIRST⁺(4)

are disjoint, as are

FIRST⁺(6), FIRST⁺(7), & FIRST⁺(8)

The grammar now has the LL(1) property

FIRST and FOLLOW Sets

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of symbols that appear as the first one in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentential form

FOLLOW(S) = {EOF}, where S is the starting symbol

To build FOLLOW sets, we need FIRST sets …

Computing FIRST Sets

For a grammar symbol X, FIRST(X) is defined as follows.

- For every terminal X, FIRST(X) = {X}.
- For every nonterminal X, if $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production, then
  - $\text{FIRST}(Y_1) \subseteq \text{FIRST}(X)$.
  - Furthermore, if $Y_1, Y_2, \ldots, Y_k$ are nullable ($Y_i \overset{*}{\rightarrow} \varepsilon$) then

$$\text{FIRST}(Y_{k+1}) \subseteq \text{FIRST}(X).$$

FIRST

- We are concerned with FIRST(X) only for the nonterminals of the grammar
- FIRST(X) for terminals is trivial
- According to the definition, to determine FIRST(A), we must inspect all productions that have A on the left

FIRST Example

Let the grammar be

$$E \rightarrow T\,E'$$
$$E' \rightarrow +\,T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *\,F\,T' \mid \varepsilon$$
$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

Find FIRST(E)

- E occurs on the left in only one production

  $$E \rightarrow T\,E'$$

- Therefore, FIRST(T) $\subseteq$ FIRST(E)
- Furthermore, T is not nullable

  Therefore, FIRST(E) = FIRST(T)
- We have yet to determine FIRST(T)

FIRST Example

Let the grammar be

$$E \rightarrow T\ E'$$

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

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ F\ T' \mid \varepsilon$$

$$F \rightarrow (E) \mid \textbf{id} \mid \textbf{num}$$

Find FIRST(T)

- T occurs on the left in only one production

  $$T \rightarrow F\ T'$$

- Therefore, FIRST(F) $\subseteq$ FIRST(T)
- Furthermore, F is not nullable
- Therefore, FIRST(T) = FIRST(F)
- We have yet to determine FIRST(F)

FIRST Example

Let the grammar be

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid$ **id** $\mid$ **num**

- Find FIRST(F).

  FIRST(F) = {(, **id**, **num**}

- Therefore,
  - FIRST(E) = {(, **id**, **num**}
  - FIRST(T) = {(, **id**, **num**}

- Find FIRST(E')
  - FIRST(E') = {+}
- Find FIRST(T')
  - FIRST(T') = {*}

Computing FOLLOW Sets

- For a grammar symbol X, FOLLOW(X) is defined as follows
  - If S is the start symbol, then EOF $\in$ FOLLOW(S)
  - If $A \rightarrow \alpha B \beta$ is a production, then FIRST(β) $\subseteq$ FOLLOW(B)
  - If $A \rightarrow \alpha B$ is a production, or $A \rightarrow \alpha B \beta$ is a production and β is nullable, then FOLLOW(A) $\subseteq$ FOLLOW(B)

FOLLOW

- We are concerned about FOLLOW(X) only for the nonterminals of the grammar.
- According to the definition, to determine FOLLOW(A), we must inspect all productions that have A on the right.

FOLLOW Example

Let the grammar be

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid$ **id** $\mid$ **num**

Find FOLLOW(E).

- E is the start symbol, therefore EOF $\in$ FOLLOW(E).
- E occurs on the right in only one production.

  $F \rightarrow (E)$.
- Therefore FOLLOW(E) = {EOF,) }

FOLLOW Example

Let the grammar be

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \varepsilon$

$F \rightarrow (E) \mid$ **id** $\mid$ **num**

Find FOLLOW(E').

- E' occurs on the right in two productions.

  $E \rightarrow T\,E'$

  $E' \rightarrow +\,T\,E'$

- Therefore,
  FOLLOW(E') = FOLLOW(E) = {EOF,) }

FOLLOW Example

Let the grammar be

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \varepsilon.$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \varepsilon.$

$F \rightarrow (E) \mid$ **id** $\mid$ **num**

Find FOLLOW(T)

- T occurs on the right in two productions

  $E \rightarrow T\ E'$

  $E' \rightarrow +\ T\ E'$

- Therefore,
  FOLLOW(T) contains FIRST(E') = {+}

- However, E' is nullable, therefore it also contains
  FOLLOW(E) = {EOF,) } and
  FOLLOW(E') = {EOF,) }

- Therefore, FOLLOW(T) = {+, EOF,) }

FOLLOW Example

Let the grammar be

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'\ |\ \varepsilon.$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'\ |\ \varepsilon.$

$F \rightarrow (E)\ |$ **id** $|$ **num**

Find FOLLOW(T')

- T' occurs on the right in two productions.

  $T \rightarrow F\ T'$

  $T' \rightarrow *\ F\ T'$

- Therefore, FOLLOW(T') = FOLLOW(T) = {EOF,), +}.

FOLLOW Example

Find FOLLOW(F)

Let the grammar be

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'\ |\ \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'\ |\ \varepsilon$

$F \rightarrow (E)\ |$ **id** $|$ **num**

- F occurs on the right in two productions.

  $T \rightarrow F\ T'$

  $T' \rightarrow *\ F\ T'$

- Therefore, FOLLOW(F) contains FIRST(T') = {*}
- However, T' is nullable, therefore it also contains

  FOLLOW(T) = {+, EOF,)} and

  FOLLOW(T') = {EOF,), +}
- Therefore, FOLLOW(F) = {*, EOF,), +}.

Classic Expression Grammar

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	<u>number</u>
10		|	<u>id</u>
11		|	<u>(</u> Expr <u>)</u>

Symbol	FIRST	FOLLOW
<u>num</u>	<u>num</u>	Ø
<u>id</u>	<u>id</u>	Ø
+	+	Ø
-	-	Ø
*	*	Ø
/	/	Ø
<u>(</u>	<u>(</u>	Ø
<u>)</u>	<u>)</u>	Ø
<u>eof</u>	<u>eof</u>	Ø
ε	ε	Ø
Goal	<u>(</u>,<u>id</u>,<u>num</u>	EOF
Expr	<u>(</u>,<u>id</u>,<u>num</u>	<u>), EOF</u>
Expr'	+, -, ε	<u>)</u>, EOF
Term	<u>(</u>,<u>id</u>,<u>num</u>	+,-,),EOF
Term'	*, /, ε	+,-,),EOF
Factor	<u>(</u>,<u>id</u>,<u>num</u>	+,-,*,/,),EOF

Classic Expression Grammar

Symbol	FIRST	FOLLOW
Goal	(,id,num	EOF
Expr	(,id,num	), EOF
Expr'	+, -, ε	), EOF
Term	(,id,num	+,-,),EOF
Term'	*, /, ε	+,-,),EOF
Factor	(,id,num	+,-,*,/,),EOF

Define FIRST+(A→α) as

- FIRST(α) ∪ FOLLOW(A), if ε ∈ FIRST(α)
- FIRST(α), otherwise

Prod'n	FIRST+	
0	(,id,num	Goal ->Expr
1	(,id,num	Expr ->Term Expr'
2	+	Expr'-> +Term Expr'
3	-	Expr'-> -Term Expr'
4	),EOF	Expr'-> ε
5	(,id,num	Term-> Factor Term'
6	*	Term'->*Factor Term'
7	/	Term'->/ Factor Term'
8	+,-,), EOF	Term'-> ε
9	number	Factor-> number
10	id	Factor-> id
11	(	Factor-> (Expr)

Building Top-down Parsers for LL(1) Grammars

Given an LL(1) grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Cannot expand Factor into an operator ⇒ error

Building Top-down Parsers

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	<u>number</u>
10		|	<u>id</u>
11		|	<u>(</u> Expr <u>)</u>

Strategy

- Encode knowledge in a table
- Use a standard "skeleton" parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
  - <u>(</u> Expr <u>)</u> or <u>Identifier</u> or <u>Number</u>
- Table might look like:

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	(	)	EOF
<u>Factor</u>	—	—	—	—	10	9	11	—	—

Expand Factor by rule 9 with input "number"

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need an interpreter for the table (skeleton parser)

LL(1) Skeleton Parser

```
word ← NextWord()            // Initial conditions, including
push $ onto Stack            // a stack to track the border of the parse tree
push the start symbol, S, onto Stack
TOS ← top of Stack

loop forever
  if TOS = $ and word = EOF then
     break & report success   // exit on success

   else if TOS is a terminal then
     if TOS matches word then
        pop Stack             // recognized TOS
        word ← NextWord()
     else report error looking for TOS  // error exit

   else                       // TOS is a non-terminal
     if TABLE[TOS,word] is A→ B1B2…Bk then
        pop Stack             // get rid of A
        push Bk, Bk-1, …, B1  // in that order
     else break & report error expanding TOS

  TOS ← top of Stack
```

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Filling the table

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Prod'n	FIRST+	
0	(,id,num	Goal ->Expr
1	(,id,num	Expr ->Term Expr'
2	+	Expr'-> +Term Expr'
3	-	Expr'-> -Term Expr'
4	),EOF	Expr'-> ε
5	(,id,num	Term-> Factor Term'
6	*	Term'->*Factor Term'
7	/	Term'->/ Factor Term'
8	+,-,), EOF	Term'-> ε
9	number	Factor-> number
10	id	Factor-> id
11	(	Factor-> (Expr)

Filling in TABLE[X,y], $X \in NT$, $y \in T$

1. write the rule $X \rightarrow \beta$, if $y \in \text{FIRST}^{+}(X \rightarrow \beta)$
2. write error if rule 1 does not define

If any entry has more than one rule, G is not LL(1)

We call this algorithm the LL(1) table construction algorithm

Actions of the LL(1) Parser for x + y x z

Rule	Stack	Input
—	eof *Goal*	↑ name + name x name
0	eof *Expr*	↑ name + name x name
1	eof *Expr′ Term*	↑ name + name x name
5	eof *Expr′ Term′ Factor*	↑ name + name x name
11	eof *Expr′ Term′* name	↑ name + name x name
→	eof *Expr′ Term′*	name ↑ + name x name
8	eof *Expr′*	name ↑ + name x name
2	eof *Expr′ Term* +	name ↑ + name x name
→	eof *Expr′ Term*	name + ↑ name x name
5	eof *Expr′ Term′ Factor*	name + ↑ name x name
11	eof *Expr′ Term′* name	name + ↑ name x name
→	eof *Expr′ Term′*	name + name ↑ x name
6	eof *Expr′ Term′ Factor* x	name + name ↑ x name
→	eof *Expr′ Term′ Factor*	name + name x ↑ name
11	eof *Expr′ Term′* name	name + name x ↑ name
→	eof *Expr′ Term′*	name + name x name ↑
8	eof *Expr′*	name + name x name ↑
4	eof	name + name x name ↑

Prod'n	FIRST+	
0	(,id,num	Goal ->Expr
1	(,id,num	Expr ->Term Expr'
2	+	Expr'-> +Term Expr'
3	-	Expr'-> -Term Expr'
4	),EOF	Expr'-> ε
5	(,id,num	Term-> Factor Term'
6	*	Term'->*Factor Term'
7	/	Term'->/ Factor Term'
8	+,-,), EOF	Term'-> ε
9	number	Factor-> number
10	id	Factor-> id
11	(	Factor-> (Expr)

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Actions of the LL(1) Parser for x + / y

Rule	Stack	Input
—	eof *Goal*	↑ name + ÷ name
0	eof *Expr*	↑ name + ÷ name
1	eof *Expr′ Term*	↑ name + ÷ name
5	eof *Expr′ Term′ Factor*	↑ name + ÷ name
11	eof *Expr′ Term′* name	↑ name + ÷ name
→	eof *Expr′ Term′*	name ↑ + ÷ name
8	eof *Expr′*	name ↑ + ÷ name
2	eof *Expr′ Term* +	name ↑ + ÷ name
→	eof *Expr′* $\boxed{Term}$	name + $\boxed{↑ ÷}$ name

Prod'n	FIRST+	
0	(,id,num	Goal ->Expr
1	(,id,num	Expr ->Term Expr'
2	+	Expr'-> +Term Expr'
3	-	Expr'-> -Term Expr'
4	),EOF	Expr'-> ε
5	(,id,num	Term-> Factor Term'
6	*	Term'->*Factor Term'
7	/	Term'->/ Factor Term'
8	+,-,), EOF	Term'-> ε
9	number	Factor-> number
10	id	Factor-> id
11	(	Factor-> (Expr)

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Exercises

Let G be the following grammar:
S::= prog B end
B::= L B l L
L::= x A
A::= a A l x A l ;

- Is G in LL(1)? If yes, write its parsing table. If not, explain why.

Let G be the grammar below:
S::= S U l x
U::= x U U l x z

- Is G in LL(1)? If yes, write its parsing table. If not, explain why

S ::= Au l bv
A = a l bAv

• G e` in LL(1)? If not modify the grammar (if it is possible) to make it LL(1) and then write its parsing table.