LECTURE 12

LL(1) PARSER

SUBJECTS

LL(1) Grammar

Eliminating Left Recursion

Left Factoring

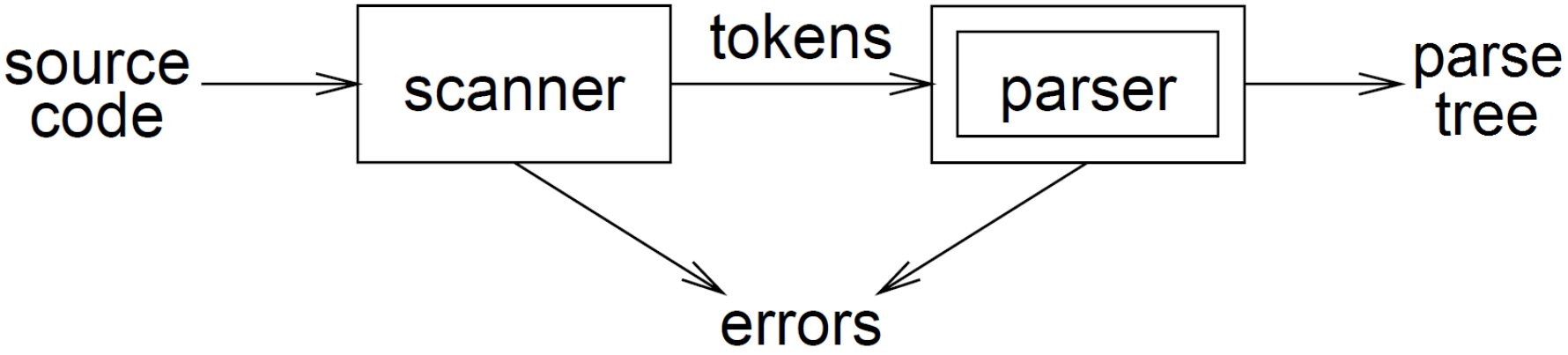
FIRST and FOLLOW sets

Parsing tables

LL(1) parsing

Many examples…

REVIEW: ROLE OF PARSER



PREDICTIVE PARSERS

uOttawa
L'Université canadienne
Canada's university

We saw that top-down parsers may need to backtrack when they select the wrong production

We want to avoid backtracking

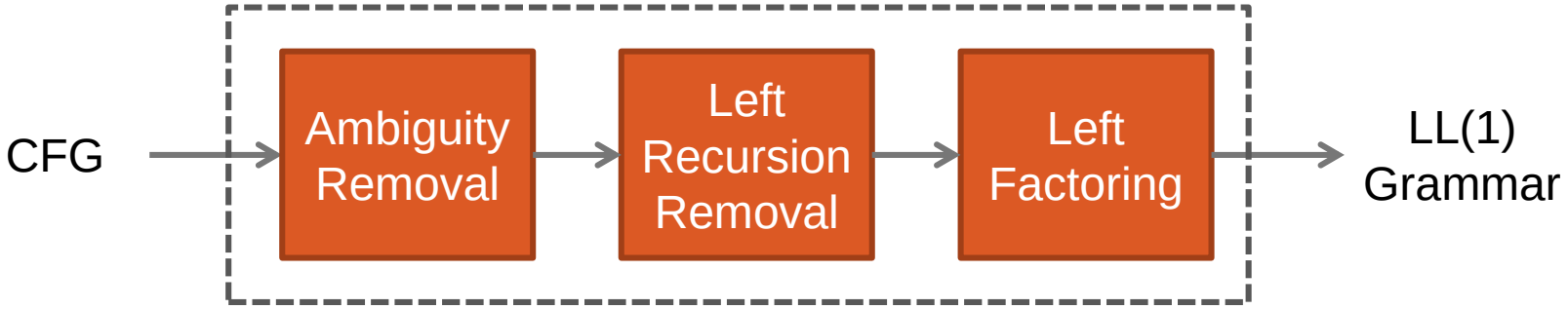
This is where predictive parsers come in useful

- LL(1): left to right scan, left-most derivation, 1-token look ahead
- LR(1): left to right scan, right most derivation, 1-token look ahead

LL(1) GRAMMAR

In order to use LL(1) parsers, the Context-Free Grammar has to be:

- Unambiguous (we have discussed ambiguity before)
- Without left recursion (we have discussed left recursion elimination before)
- Left factored (we will discuss left factoring today)



The above methods will convert many grammars to LL(1) form, but not all… There exit many exceptions.

REVIEW: LEFT RECURSION

uOttawa
L'Université canadienne
Canada's university

A grammar is left recursive if:

"It has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α"

Top down parses cannot handle left-recursion in a grammar

ELIMINATING LEFT RECURSION

uOttawa
L'Université canadienne
Canada's university

Consider the grammar fragment:

$$\langle \text{foo} \rangle ::= \langle \text{foo} \rangle \alpha \mid \beta$$

Where α and β do not start with ⟨foo⟩

We can re-write this as:

$$\langle \text{foo} \rangle ::= \beta \langle \text{bar} \rangle$$
$$\langle \text{bar} \rangle ::= \alpha \langle \text{bar} \rangle \mid \varepsilon$$

Where ⟨bar⟩ is a new non-terminal

This Fragment contains no left recursion

LEFT FACTORING

For any two productions A→α | β, we would like a distinct way of choosing the correct production to expand

We define FIRST(α) as the set of terminals that appear first in some string derived from α

For a terminal **w**, we can say:

$$w \in \text{FIRST}(\alpha) \text{ iff } \alpha \overset{*}{\Rightarrow} wz$$

LEFT FACTORING

uOttawa
L'Université canadienne
Canada's university

Now going back to our two productions: A→α and A→β, we would like:

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

This would allow the parser to make a correct choice with a look ahead of only one symbol

LEFT FACTORING

Given this grammar:

1	⟨expr⟩	::=	⟨term⟩ + ⟨expr⟩
2		|	⟨term⟩ − ⟨expr⟩
3		|	⟨term⟩
4	⟨term⟩	::=	⟨factor⟩ ∗ ⟨term⟩
5		|	⟨factor⟩ / ⟨term⟩
6		|	⟨factor⟩
7	⟨factor⟩	::=	`num`
8		|	`id`

The parser cannot choose between productions 1, 2 and 3 given an input token of `num` or `id`

$$\text{FIRST}(1) \cap \text{FIRST}(2) \cap \text{FIRST}(3) \neq \phi$$

Left factoring is required to solve this problem!

LEFT FACTORING

uOttawa
L'Université canadienne
Canada's university

So how does it work?

For each non-terminal **A**, find the longest prefix **α** common to two or more of its alternatives

If $\alpha \neq \varepsilon$, **then replace all of the A productions**

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

With

$A \rightarrow \alpha\ A'$

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

Where A' is a new non-terminal

Repeat until no two alternatives for a single non-terminal have a common prefix

LEFT FACTORING

Therefore, in our grammar:

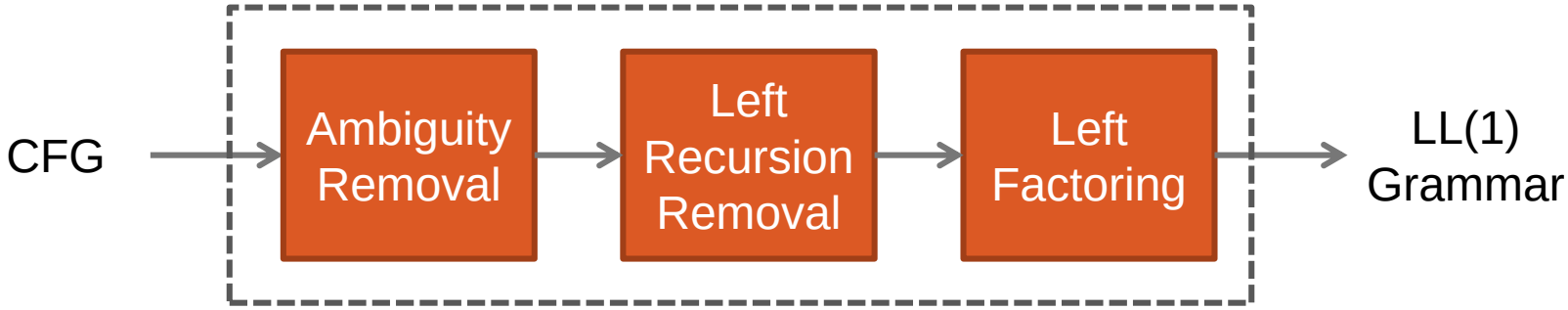
$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle + \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle - \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle * \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle / \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle
\end{array}
$$

When we perform the left factoring (on expr and term), we get:

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
& | & - \langle \text{expr} \rangle \\
& | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
& | & / \langle \text{term} \rangle \\
& | & \varepsilon
\end{array}
$$

LL(1) PARSER

So we now know how to take a Context-Free Grammar, and transform it into an LL(1) grammar (at least we can try…)



LL(1) PARSER

We need to implement an LL(1) parser that can analyse the syntax of an input string of tokens without backtracking

- Of course given that the grammar is compatible with such parser

In order to do that, we need to find two sets for each non-terminal:

- FIRST (we have briefly discussed this set earlier)
- FOLLOW

FIRST SET CALCULATION

Rules to calculate the FIRST set:

1. FIRST(terminal) is {terminal}

2. If A→aα, and a is a terminal:

   $\{a\} \in FIRST(A)$

3. If A→Bα, and rule B→ε does **NOT** exist in the grammar:

   $FIRST(B) \in FIRST(A)$

4. If A→Bα, and rule B→ε **DOES** exist in the grammar:

   $\{(FIRST(B) - \varepsilon) \cup FIRST(\alpha)\} \in FIRST(A)$

FIRST SET CALCULATION

uOttawa
L'Université canadienne
Canada's university

Let's apply these rules to an example.

Given the grammar:

```
<S> ::= <X><Y>
<X> ::= h
<Y> ::= p
```

FIRST(X) = {h} (applying 2nd rule)

FIRST(Y) = {p} (applying 2nd rule)

FIRST(S) = FIRST(X)

= {h} (applying 3rd rule)

1) FIRST(terminal) is {terminal}

2) If A→aα, and a is a terminal:
{a}∈FIRST(A)

3) If A→Bα, and rule B→ε does **NOT** exist:
FIRST(B)∈FIRST(A)

4) If A→Bα, and rule B→ε **DOES** exist:
{(FIRST(B) − ε) ∪ FIRST(α)}∈FIRST(A)

FIRST SET CALCULATION

uOttawa
L'Université canadienne
Canada's university

Another example…

Given the grammar:

<S> ::= <X><Y>
<X> ::= h | ε
<Y> ::= p

FIRST(X) = {h, ε} (2nd rule)

FIRST(Y) = {p} (2nd rule)

FIRST(S) = [FIRST(X)- ε] ∪ FIRST(Y)
= {h, p} (4th rule)

1) FIRST(terminal) is {terminal}

2) If A→aα, and a is a terminal:
{a}∈FIRST(A)

3) If A→Bα, and rule B→ε does **NOT** exist:
FIRST(B)∈FIRST(A)

4) If A→Bα, and rule B→ε **DOES** exist:
{(FIRST(B)- ε)∪FIRST(α)}∈FIRST(A)

FOLLOW SET

Informal definition:

The follow set of non-terminal A (i.e. Follow(A)) contains all the terminals that appear AFTER A in any string generated by the grammar G.

FOLLOW SET CALCULATION

uOttawa
L'Université canadienne
Canada's university

Rules to calculate the FOLLOW set:

1. {$} ∈ FOLLOW(S) → (where S is the starting symbol)

2. If A→ αB:

   FOLLOW(A) ∈ FOLLOW(B)

3. If A→αBγ, and γ→ε does **NOT** exist in the grammar:

   FIRST(γ) ∈ FOLLOW(B)

4. If A→αBγ, and γ→ε **DOES** exist in the grammar:

   {(FIRST(γ)− ε) ∪ FOLLOW(A)} ∈ FOLLOW(B)

FOLLOW SET CALCULATION

uOttawa
L'Université canadienne
Canada's university

Let's apply these rules to an example.

Given the grammar:

```
<S> ::= <X><Y>
<X> ::= h
<Y> ::= p
```

FOLLOW(S) = {$} (1st rule)

FOLLOW(X) = FIRST(Y)

= {p} (3rd rule)

FOLLOW(Y) = FOLLOW (S)

= {$} (2nd rule)

1) $\{\$\} \in FOLLOW(S)$

2) If $A \rightarrow \alpha B$:

$FOLLOW(A) \in FOLLOW(B)$

3) If $A \rightarrow \alpha B \gamma$, and $\gamma \rightarrow \varepsilon$ does **NOT** exist:

$FIRST(\gamma) \in FOLLOW(B)$

4) If $A \rightarrow \alpha B \gamma$, and $\gamma \rightarrow \varepsilon$ **DOES** exist:

$\{(FIRST(\gamma) - \varepsilon) \cup FOLLOW(A)\} \in FOLLOW(B)$

FOLLOW SET CALCULATION

uOttawa
L'Université canadienne
Canada's university

Another example…

Given the grammar:

<S> ::= <X><Y>
<X> ::= h
<Y> ::= p | ε

FOLLOW(S) = {$} (1st rule)

FOLLOW(X) = [FIRST(Y)- ε]∪FOLLOW(S)

= {p, $} (4th rule)

FOLLOW(Y) = FOLLOW (S)

= {$} (2nd rule)

1) $\{\$\} \in FOLLOW(S)$

2) If A→ αB:
$FOLLOW(A) \in FOLLOW(B)$

3) If A→αBγ, and γ→ε does **NOT** exist:
$FIRST(\gamma) \in FOLLOW(B)$

4) If A→αBγ, and γ→ε **DOES** exist:
$\{(FIRST(\gamma)-\varepsilon) \cup FOLLOW(A)\} \in FOLLOW(B)$

FIRST AND FOLLOW

Let's calculate **FIRST** and **FOLLOW** for each non-terminal in our famous grammar:

Do the easy ones first:

FIRST(factor)={num, id}

FIRST(term')={*, /, ε}

FIRST(expr')={+, -, ε}

And then the more challenging ones:

FIRST(term)=FIRST(factor)={num, id}

FIRST(expr)=FIRST(term)={num, id}

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

FIRST "Rules"

1) FIRST(terminal) is {terminal}

2) If A→aα, and a is a terminal:
{a}∈FIRST(A)

3) If A→Bα, and rule B→ε does **NOT** exist:
FIRST(B)∈FIRST(A)

4) If A→Bα, and rule B→ε **DOES** exist:
{(FIRST(B)- ε)∪FIRST(α)}∈FIRST(A)

FIRST AND FOLLOW

Let's calculate **FIRST** and **FOLLOW** for each non-terminal in our famous grammar:

Start with the easy ones:

FOLLOW(expr)={$}

FOLLOW(expr′)=FOLLOW(expr)={$}

Grammar

⟨expr⟩	::=	⟨term⟩⟨expr′⟩
⟨expr′⟩	::=	+⟨expr⟩
	|	−⟨expr⟩
	|	ε
⟨term⟩	::=	⟨factor⟩⟨term′⟩
⟨term′⟩	::=	∗⟨term⟩
	|	/⟨term⟩
	|	ε
⟨factor⟩	::=	num
	|	id

FOLLOW "Rules"

1) {$}∈FOLLOW(S)

2) If A→ αB:

FOLLOW(A) ∈ FOLLOW(B)

3) If A→αBγ, and γ→ε does **NOT** exist:

FIRST(γ) ∈ FOLLOW(B)

4) If A→αBγ, and γ→ε **DOES** exist:

{(FIRST(γ)−ε)∪FOLLOW(A)}∈FOLLOW(B)

FIRST AND FOLLOW

Let's calculate **FIRST** and **FOLLOW** for each non-terminal in our famous grammar:

Let's do the harder ones:

FOLLOW(term)=[FIRST(expr')-ε]∪FOLLOW(expr)

= {+, -, $}

FOLLOW(factor)=[FIRST(term')-ε]∪FOLLOW(term)

= {*, /, +, -, $}

FOLLOW(term')= FOLLOW(term)

= {+, -, $}

Grammar

⟨expr⟩	::=	⟨term⟩⟨expr′⟩
⟨expr′⟩	::=	+⟨expr⟩
	|	−⟨expr⟩
	|	ε
⟨term⟩	::=	⟨factor⟩⟨term′⟩
⟨term′⟩	::=	∗⟨term⟩
	|	/⟨term⟩
	|	ε
⟨factor⟩	::=	num
	|	id

FOLLOW "Rules"

1) {$}∈FOLLOW(S)

2) If A→ αB:

FOLLOW(A) ∈ FOLLOW(B)

3) If A→αBγ, and γ→ε does **NOT** exist:

FIRST(γ) ∈ FOLLOW(B)

4) If A→αBγ, and γ→ε **DOES** exist:

{(FIRST(γ)-ε)∪FOLLOW(A)}∈FOLLOW(B)

FIRST AND FOLLOW

Summary:

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Using these sets, we build a parse table

- This parse table is necessary to perform LL(1) parsing

SHORT BREAK

Imagine, you are not in class, but on the beach, *and I do not exist…*

Doesn't that feel good!!

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & +\langle \text{expr} \rangle \\
 & | & -\langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & *\langle \text{term} \rangle \\
 & | & /\langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

We will add two entries associated with num and id

	num	id	+	-	*	/	$
expr							
expr'							
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

We will add two entries associated with num and id

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'							
term							
term'							
factor							

PARSE TABLE

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

Fill the expr' in the same way

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'							
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

Fill the expr' in the same way

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

What about the epsilon? Use the FOLLOW set to add the epsilon rule…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

What about the epsilon? Use the FOLLOW set to add the epsilon rule…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

⟨expr⟩ ::= ⟨term⟩⟨expr′⟩
⟨expr′⟩ ::= +⟨expr⟩ | −⟨expr⟩ | ε
⟨term⟩ ::= ⟨factor⟩⟨term′⟩
⟨term′⟩ ::= ∗⟨term⟩ | /⟨term⟩ | ε
⟨factor⟩ ::= num | id

No, epsilon, just use the FIRST set

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term							
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

⟨expr⟩ ::= ⟨term⟩⟨expr′⟩
⟨expr′⟩ ::= +⟨expr⟩ | −⟨expr⟩ | ε
⟨term⟩ ::= ⟨factor⟩⟨term′⟩
⟨term′⟩ ::= ∗⟨term⟩ | /⟨term⟩ | ε
⟨factor⟩ ::= num | id

No, epsilon, just use the FIRST set

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

This one has an epsilon, use the FIRST and FOLLOW sets

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'							
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

This one has an epsilon, use the FIRST and FOLLOW sets

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'			<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

Fill the row for factor…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'			<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor							

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

Fill the row for factor…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'			<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id					

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & | & - \langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & | & / \langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

Add dashes to the remaining cells to indicate: no entry

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>					
expr'			<expr'>→ +<expr>	<expr'>→ -<expr>			<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>					
term'			<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id					

PARSE TABLE

uOttawa
L'Université canadienne
Canada's university

FIRST Sets

```
FIRST(expr)= {num, id}
FIRST(expr')= {+, -, ε}
FIRST(term)= {num, id}
FIRST(term')= {*, /, ε}
FIRST(factor)= {num, id}
```

FOLLOW Sets

```
FOLLOW(expr)={$}
FOLLOW(expr')={$}
FOLLOW(term)= {+, -, $}
FOLLOW(term')= {+, -, $}
FOLLOW(factor)= {*, /, +, -, $}
```

Grammar

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & +\langle \text{expr} \rangle \\
 & | & -\langle \text{expr} \rangle \\
 & | & \varepsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & *\langle \text{term} \rangle \\
 & | & /\langle \text{term} \rangle \\
 & | & \varepsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

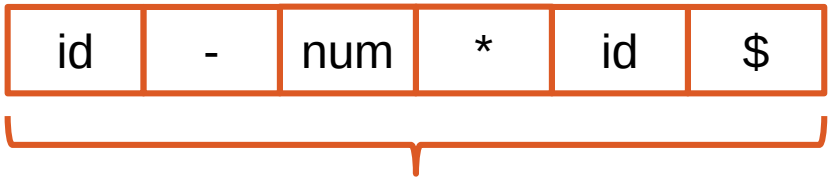
PARSING USING LL(1)

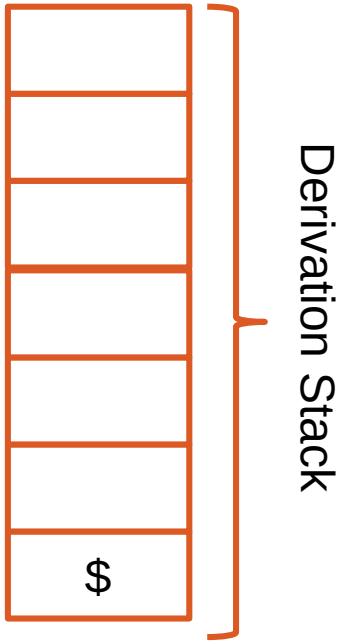
In order to implement an LL(1) parser, we need to use the following data structures:

- Parse table (can be implemented with a 2D array or something fancier)
- Stack (that will contain the derivations)
- List (that will contain the token input stream)

PARSING USING LL(1)

Example:





Parse Table

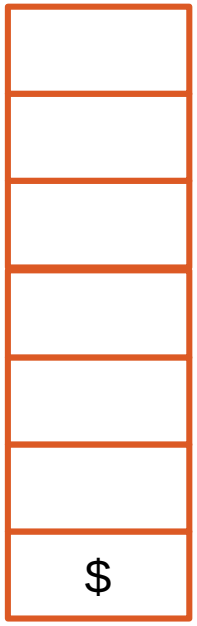
	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

uOttawa
L'Université canadienne
Canada's university

Example:

id	-	num	*	id	$



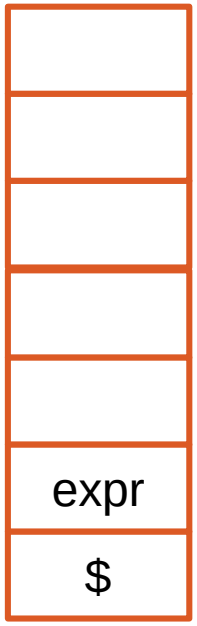
Start by pushing the starting symbol (goal) into the stack

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$



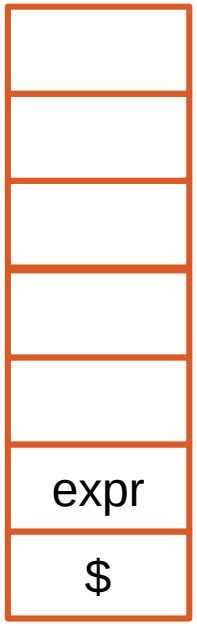
Start by pushing the starting symbol (goal) into the stack

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$



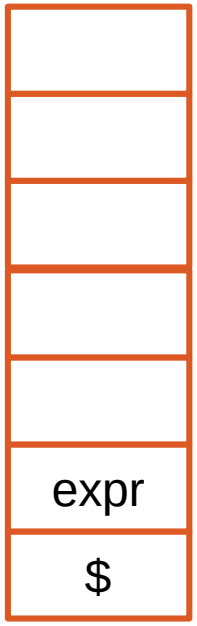
On the head of the input stream, we have **id**
On the top of the stack, we have **expr**
Using the parsing table, we retrieve the rule: **<expr>→<term><expr'>**

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$



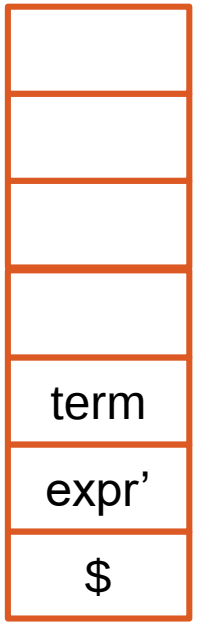
→**POP** expr and **PUSH** term and expr’

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$



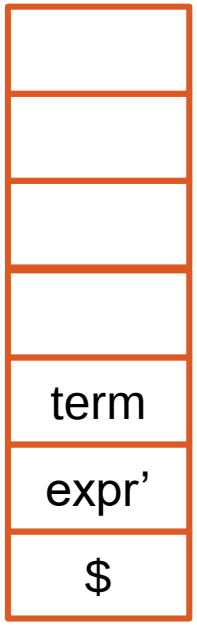
On the head of the input stream, we have **id**
On the top of the stack, we have **term**
Using the parsing table, we retrieve the rule: **<term>→<factor><term'>**

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$



→**POP** term and **PUSH** factor and term'

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	-	num	*	id	$

factor
term’
expr’
$

On the head of the input stream, we have **id**
On the top of the stack, we have **factor**
Using the parsing table, we retrieve the rule: **<factor>→id**

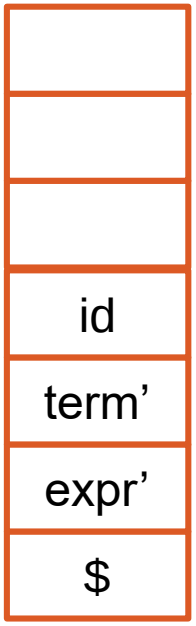
	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

uOttawa
L'Université canadienne
Canada's university

Example:

id	-	num	*	id	$



Whenever we have a terminal on top of the stack, we check if it matches the head of the list

If it does not →syntax does not follow grammar

If it does, REMOVE head of the list and POP the stack

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

uOttawa
L'Université canadienne
Canada's university

Example:

-	num	*	id	$

term'
expr'
$

Whenever we have a terminal on top of the stack, we check if it matches the head of the list

If it does not →syntax does not follow grammar

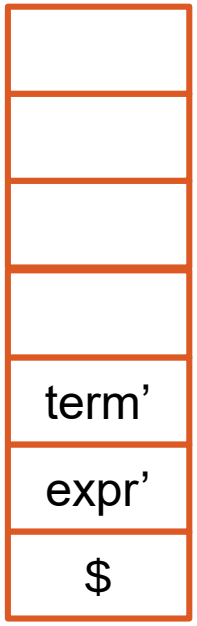
If it does, REMOVE head of the list and POP the stack

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

-	num	*	id	$



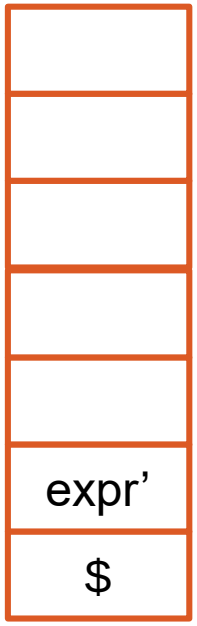
On the head of the input stream, we have **-**
On the top of the stack, we have **term’**
Using the parsing table, we retrieve the rule: **<term’>→ε**
Therefore, we should simply POP

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

-	num	*	id	$



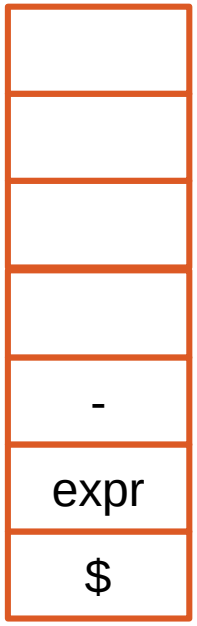
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

-	num	*	id	$



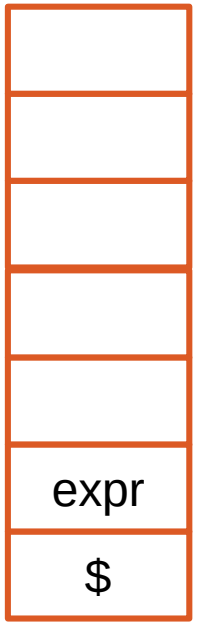
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

num	*	id	$



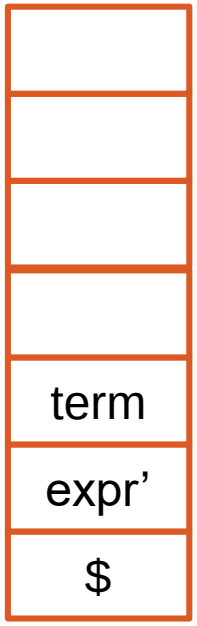
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

num	*	id	$



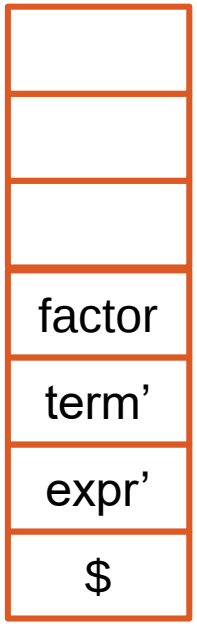
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

num	*	id	$



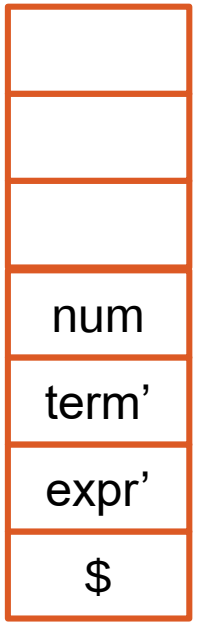
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

num	*	id	$



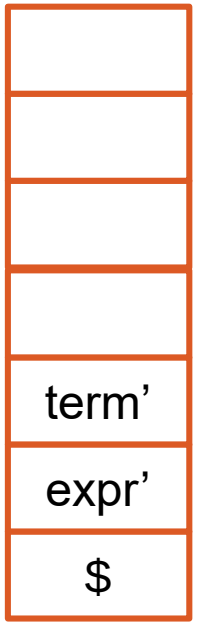
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

*	id	$



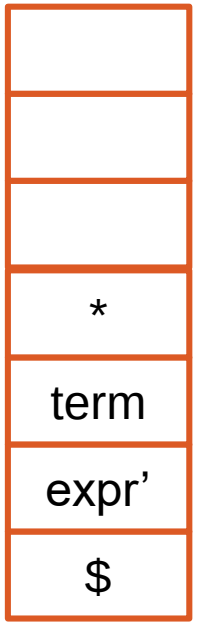
And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

*	id	$



And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	$

term
expr’
$

And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	$

factor
term’
expr’
$

And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

id	$

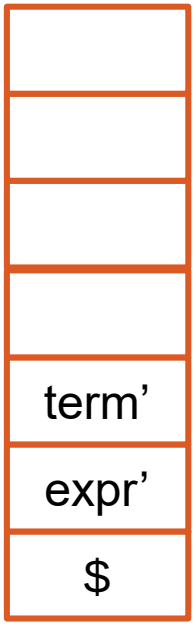
id
term’
expr’
$

And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:



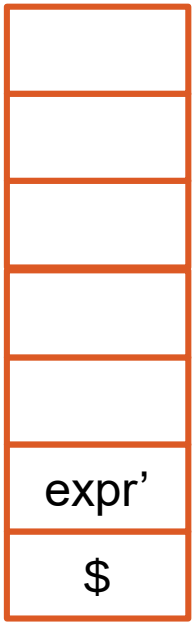
$

And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr’	-	-	<expr’>→ +<expr>	<expr’>→ -<expr>	-	-	<expr’>→ ε
term	<term>→ <factor><term’>	<term>→ <factor><term’>	-	-	-	-	-
term’	-	-	<term’>→ ε	<term’>→ ε	<term’>→ *<term>	<term’>→ /<term>	<term’>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:



$

And continue the same way…

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

PARSING USING LL(1)

Example:

$

$

We have verified that the input string is a sentence of the grammar!!

	num	id	+	-	*	/	$
expr	<expr>→ <term><expr'>	<expr>→ <term><expr'>	-	-	-	-	-
expr'	-	-	<expr'>→ +<expr>	<expr'>→ -<expr>	-	-	<expr'>→ ε
term	<term>→ <factor><term'>	<term>→ <factor><term'>	-	-	-	-	-
term'	-	-	<term'>→ ε	<term'>→ ε	<term'>→ *<term>	<term'>→ /<term>	<term'>→ ε
factor	factor→ num	factor → id	-	-	-	-	-

THANK YOU!

QUESTIONS?