Predictive Parsers

Can we avoid backtracking? Yes, if for a given input symbol and given non-terminal, we can choose the alternative **appropriately.**

This is possible if the first terminal of every alternative in a production is unique:

`A → a B D | b B B`
`B → c | b c e`
`D → d`

parsing an input "abced" has no backtracking.

Left factoring to enable predication:

$A \to \alpha\beta \mid \alpha\gamma$

change to

$A \to \alpha A'$
$A' \to \beta \mid \gamma$

For predicative parsers, must eliminate left recursion

LL(k) Parsing

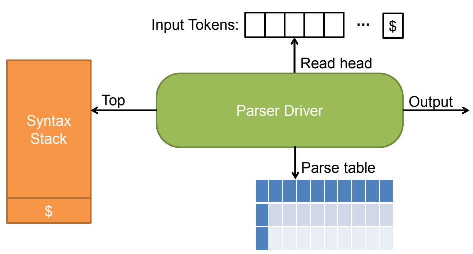
LL(k)

- L — left to right scan
- L — leftmost derivation
- k — k symbols of lookahead

in practice, k = 1

It is table-driven and efficient.

LL(k) Parser Structure



Syntax stack — hold right hand side (RHS) of grammar rules
Parse table — M[A,b] — an entry containing rule "A → …" or error
Parser driver — next action based on *(current token, stack top)*

Sample Parse Table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

Implementation with 2-D parse table:

- A row for each non-terminal
- A column for all possible terminals and $ (the end of input marker)
- Every table entry contains at most one production
  - Required for a grammar to be LL(1)
- No backtracking

Fixed action for each *(non-terminal, input symbol)* combination

LL(1) Parsing Algorithm

X — symbol at the top of the syntax stack
a — current input symbol

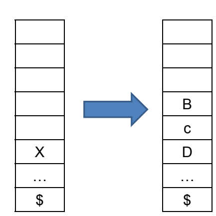
Parsing based on **(X, a)**:

If X = a = $, then
parser halts with "**success**"
If X = a ≠ $, then
pop X from stack **and** advance input head
If X ≠ a, then
Case (a): if X ∈ T, then
parser halts with "**failed**," input rejected
Case (b): if X ∈ N, M[X,a] = "X → RHS"
pop X **and** push RHS to stack in reverse order

Push RHS in Reverse Order

X — symbol at the top of the syntax stack
a — current input symbol

if M[X,a] = "X → B c D":



LL(1) Grammars

Remove left recursive and perform left factoring

Given the grammar:

```
E → T + E | T
T → int * T | int | ( E )
```

The grammar has no left recursion but requires left factoring.

After rewriting grammar, we have:

```
E → TX
X → + E | ε
T → int Y | ( E )
Y → * T | ε
```

LL(1) Parsing

Stack:

E ← Top
$

Input Tokens: | int | * | int | $ |

Read head (at first int)

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Stack:

E ← Top
$

Input Tokens: | int | * | int | $ |

Read head (at first int)

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Stack:

T ← Top
X
$

Input Tokens: | int | * | int | $ |

Read head (at first int)

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Stack:

T ← Top
X
$

Input Tokens: | int | * | int | $ |

Read head (at first int)

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Stack:

int ← Top
Y
X
$

Input Tokens: | int | * | int | $ |

Read head (at first int)

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Y
X
$

← Top

Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

Y
X
$

← Top

Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

*
T
X
$

← Top

Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

T
X
$

← Top

Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

T
X
$

← Top

Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing

int
Y
X
$

← Top

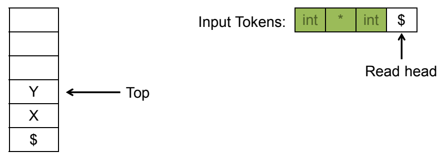
Input Tokens: | int | * | int | $ |

Read head

Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

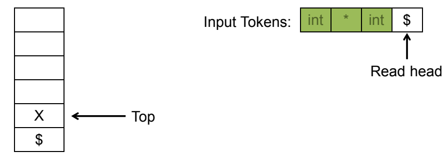
LL(1) Parsing



Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

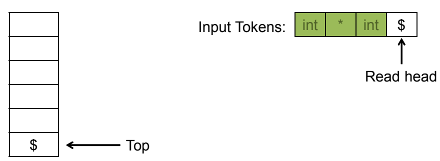
LL(1) Parsing



Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

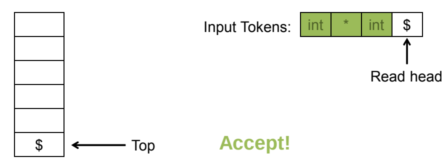
LL(1) Parsing



Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

LL(1) Parsing



Parse table

	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

Action List

Stack	Input	Action
E $	int * int $	E → TX
T X $	int * int $	T → int Y
int Y X $	int * int $	terminal
Y X $	* int $	Y → * T
* T X $	* int $	terminal
T X $	int $	T → int Y
int Y X $	int $	terminal
Y X $	$	Y → ε
X $	$	X → ε
$	$	Halt and accept

Constructing the Parse Table

We need to know what non-terminals to place our productions in the table?

We know that we have restricted our grammars so that left recursion is eliminated and they have been left factored. That means that each production is uniquely recognizable by the first terminal that production would derive.

Thus, we can construct our table from 2 sets:

- For each symbol A, the set of terminals that can begin a string derived from A. This set is called the FIRST set of A
- For each non-terminal A, the set of terminals that can appear after a string derived from A is called the FOLLOW set of A

First(α)

First(α) = set of terminals that start string of terminals derived from α.

Apply following rules until no terminal or ε can be added

1. If $t \in T$, then **First**(t) = { t }.
   For example **First**(+) = { + }.

2. If $X \in N$ and $X \to \varepsilon$ exists (nullable), then add ε to **First**(X).
   For example, **First**(Y) = { *, ε}.

3. If $X \in N$ and $X \to Y_1Y_2Y_3 \ldots Y_m$, where $Y_1, Y_2, Y_3, \ldots Y_m$ are non-terminals, then:
   for each *i* from 1 to *m*
   if $Y_1 \ldots Y_{i-1}$ are all nullable (or if *i* = 1)
   First(X) = **First**(X) $\cup$ **First**(Y_i)

Follow(α)

Follow(α) = { t | S $\Rightarrow$ *α t β }

- Intuition: if $X \to A\ B$, then **First**(B) $\subseteq$ **Follow**(A)
- However, B may be ε, i.e., $\beta \overset{*}{\Rightarrow} \varepsilon$

Apply following rules until no terminal or ε can be added

1. $\$ \in$ **Follow**(S), where S is the start symbol.
   e.g., **Follow**(E) = {\$ ... }.

2. Look at the occurrence of a non-terminal on the right hand side of a production which is followed by something
   If $A \to \alpha\ B\ \beta$, then **First**(β) - {ε} $\subseteq$ **Follow**(B)

3. Look at N on the RHS that is not followed by anything,
   if ($A \to \alpha\ B$) or ($A \to \alpha\ B\ \beta$ and $\varepsilon \in$ **First**(β)),
   then **Follow**(A) $\subseteq$ **Follow**(B)

Algorithm to Compute FIRST, FOLLOW, and nullable

```
Initialize FIRST and FOLLOW to all empty sets, and nullable to all
false.

foreach terminal symbol Z
        FIRST[Z] ← {Z}
do
        foreach production X → Y_1Y_2 … Y_k
                if Y_1 … Y_k are all nullable (or if k = 0)
                        then nullable[X] ← true
        foreach i from 1 to k, each j from i + 1 to k
                if Y_1 … Y_{i-1} are all nullable (or if i = 1)
                        then FIRST[X] ← FIRST[X] ∪ FIRST[Y_i]
                if Y_{i+1} … Y_k are all nullable (or if i = k)
                        then FOLLOW[Y_i] ← FOLLOW[Y_i] ∪ FOLLOW[X]
                if Y_{i+1} … Y_{j-1} are all nullable (or if i + 1 = j )
                        then FOLLOW[Y_i] ← FOLLOW[Y_i] ∪ FIRST[Y_j ]
until FIRST, FOLLOW, and nullable did not change in this iteration.
```

Example

Grammar:

E → T X
X → + E | ε
T → int Y | (E)
Y → * T | ε

First Set:

E → **T** X
X → **+** E
X → **ε**
T → **int** Y
T → **(** E)
Y → ***** T
Y → **ε**

Follow Set:

$
E → **T X**
X → + **E**
T → int **Y**
T → (**E)**
Y → * **T**

Symbol	First	Follow
(	(	
)	)	
+	+	
*	*	
int	Int	
Y	*, ε	$,), +
X	+, ε	$,)
T	(, int	$,), +
E	(, int	$,)

Constructing LL(1) Parse Table

To construct the parse table, we check each $A \to \alpha$

- For each terminal $a \in$ **First**(α), add $A \to \alpha$ to M[A, α].
- If $\varepsilon \in$ **First**(α), then for each terminal $b \in$ **Follow**(A),
  - add $A \to \alpha$ to M[A, α].
- If $\varepsilon \in$ **First**(α) and $\$ \in$ **Follow**(A), then add $A \to \alpha$ to M[A, \$].

Constructing LL(1) Parse Table

For each terminal $a \in$ **First**(α), add $A \to \alpha$ to M[A, α].

Grammar:

E → T X
X → + E
X → ε
T → int Y
T → (E)
Y → * T
Y → ε

Symbol	First	Follow
(	(	
)	)	
+	+	
*	*	
int	int	
Y	*, ε	$,), +
X	+, ε	$,)
T	(, int	$,), +
E	(, int	$,)

	int	*	+	(	)	$
E	E → T X			E → T X		
X			X → + E			
T	T → int Y			T → (E)		
Y		Y → * T				

Constructing LL(1) Parse Table

If $\varepsilon \in$ **First**(α), then for each terminal b $\in$ **Follow**(A), add `A → α` to M[A, α].

Grammar:

```
E → T X
X → + E
X → ε
T → int Y
T → ( E )
Y → * T
Y → ε
```

Symbol	First	Follow
(	(	
)	)	
+	+	
*	*	
int	int	
Y	*, ε	$,), +
X	+, ε	$,)
T	(, int	$,), +
E	(, int	$,)

	int	*	+	(	)	$
E	E → T X			E → T X		
X			X → + E		X → ε	
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	

Constructing LL(1) Parse Table

If $\varepsilon \in$ **First**(α) and \$ $\in$ **Follow**(A), then add `A → α` to M[A, \$].

Grammar:

```
E → T X
X → + E
X → ε
T → int Y
T → ( E )
Y → * T
Y → ε
```

Symbol	First	Follow
(	(	
)	)	
+	+	
*	*	
int	int	
Y	*, ε	$,), +
X	+, ε	$,)
T	(, int	$,), +
E	(, int	$,)

	int	*	+	(	)	$
E	E → T X			E → T X		
X			X → + E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → * T	Y → ε		Y → ε	Y → ε

Is a Grammar LL(1)?

Observation

If a grammar is LL(1), then each of its LL(1) table entries contain at most one rule. Otherwise, it is not LL(1)

Two methods to determine if a grammar is LL(1) or not:

1. Construct LL(1) table, and check if there is a multi-rule entry

or

2. Check each rule as if the table were being constructed:

G is LL1(1) **iff** for a rule `A →` $\alpha \mid \beta$

a) **First**(α) ∩ **First**(β) = ∅
b) at most one of α and β can derive ε
c) If β derives ε, then **First**(α) ∩ **Follow**(β) = ∅

Ambiguous Grammars

Some grammars may need more than one token of lookahead (k). However, some grammars are not LL regardless of how the grammar is changed.

```
S → if C then S | if C then S else S | …
C → b
```

change to

```
S → if C then S X |  …
X → else S | ε
C → b
```

problem sentence: "if b then if b then a else a"

"else" $\in$ **First**(X)
First(X)-ε ⊆ **Follow**(S)
X → else ... | ε
"else" $\in$ **Follow**(X)

Removing Ambiguity

To remove ambiguity, it is possible to rewrite the grammar.

For the "if-then-else" example, how to rewrite?

May not even need to rewrite in this case, we can just use the `X → else S` production over the `X → ε`

However, by changing the grammar,

- It might make the other phases of the compiler more difficult
- It becomes harder to determine semantics and generate code
- It is less appealing to programmers

LL(1) Summary

LL(1) parsers operate in linear time and at most linear space relative to the length of input because:

Time — each input symbol is processed constant number of times

Space — stack is smaller than the input (in case we remove `X → ε`)

Summary

First and **Follow** sets are used to construct predictive parsing tables

Intuitively, **First** and **Follow** sets guide the choice of rules:

- For non-terminal **A** and input **t**, use a production rule $A \rightarrow \alpha$ where **t** $\in$ **First(**α**)**
- For non-terminal **A** and input **t**, if $A \rightarrow \alpha$ and t $\in$ **Follow**(A), use the production $A \rightarrow \alpha$ where $\varepsilon \in$ **First(**α**)**

What is LL(0)?

Why are LL(2) ... LL(k) are not widely used ?