Top-Down Parsing

A **top-down parser** begins with the root of the parse tree and systematically extends the tree downward until its leaves match the classified words returned by the scanner.

General top-down parsing algorithm

1. select a nonterminal symbol on the lower fringe of the partially built parse tree
2. replace symbol with children corresponding to right-hand side of one of its productions
3. repeat this process until
   a) fringe only contains terminal symbols and input stream has been exhausted
      - $\rightarrow$ parsing succeeds
   b) clear mismatch occurs between fringe and input stream
      - $\rightarrow$ backtrack and try another production
      - $\rightarrow$ if there are no more possible productions, report an error

```
root ← node for the start symbol S
focus ← root
push(null)
word ← NextWord()
while true do
    if focus is a nonterminal then
        pick next rule to expand focus (A → β_1, β_2, ..., β_n)
        build nodes for β_1, β_2, ..., β_n as children of focus
        push(β_n, β_{n−1}, ..., β_2)
        focus ← β_1
    else if word matches focus then
        word ← NextWord()
        focus ← pop()
    else if word = eof and focus = null then
        accept the input and return root
    else
        backtrack
```

Top-Down Parsing

If the focus is a terminal symbol that does not match the input, the parser must backtrack.

The implementation of "*backtrack*" is straightforward

1. set `focus` to its parent in the partially-built parse tree and disconnects its children
2. if an untried rule remains with `focus` on its left-hand side
   - perform Lines 7 to 10 of algorithm on Slide 145
3. if no untried rule remains
   - move up another level and try again
   - if out of possibilities, report a syntax error and quit

Backtracking increases the asymptotic cost of parsing. In practice, it is an expensive way to discover syntax errors.

One key insight makes top-down parsing efficient: a large subset of the context-free grammars can be parsed **without** backtracking!

Transforming a Grammar for Top-Down Parsing

The efficiency of a top-down parser depends critically on its ability to pick the **correct** production each time that it expands a nonterminal

- if the parser chooses wisely, top-down parsing is efficient
- if the parser chooses poorly, the cost of parsing rises
- worst case behavior: the parser **does not** terminate!

Two **structural issues** with CFGs can lead to problems with top-down parsers

1. non-termination due to left recursion
2. backtracking due to unbounded lookahead

Next, we will look at transformations that the compiler writer can apply to the grammar to avoid these problems.

A Top-Down Parser with Oracular Choice

Assume that the parser has an **oracle** that picks the correct production at each point.

Example on next slide applies this parser to $a + b \times c$

- $\uparrow$ current position of the parser in the input
- $\rightarrow$ step in which the parser matches a terminal symbol and advances the input

At each step, the sentential form represents the lower fringe of the partially-built parse tree.

Implications of oracular choice

- number of steps **proportional** to derivation length plus input length
- **inconsistent** choices, *e.g.*, productions applied to *Expr* in first and second step

Rule	Sentential Form	Input
	Expr	$\uparrow$ name $+$ name $\times$ name
1	*Expr* $+$ *Term*	$\uparrow$ name $+$ name $\times$ name
3	*Term* $+$ *Term*	$\uparrow$ name $+$ name $\times$ name
6	*Factor* $+$ *Term*	$\uparrow$ name $+$ name $\times$ name
9	name $+$ *Term*	$\uparrow$ name $+$ name $\times$ name
$\rightarrow$	name $+$ *Term*	name $\uparrow$ $+$ name $\times$ name
$\rightarrow$	name $+$ *Term*	name $+$ $\uparrow$ name $\times$ name
4	name $+$ *Term* $\times$ *Factor*	name $+$ $\uparrow$ name $\times$ name
6	name $+$ *Factor* $\times$ *Factor*	name $+$ $\uparrow$ name $\times$ name
9	name $+$ name $\times$ *Factor*	name $+$ $\uparrow$ name $\times$ name
$\rightarrow$	name $+$ name $\times$ *Factor*	name $+$ name $\uparrow$ $\times$ name
$\rightarrow$	name $+$ name $\times$ *Factor*	name $+$ name $\times$ $\uparrow$ name
9	name $+$ name $\times$ name	name $+$ name $\times$ $\uparrow$ name
$\rightarrow$	name $+$ name $\times$ name	name $+$ name $\times$ name $\uparrow$

Eliminating Left Recursion

With the current version of our expression grammar, it is difficult to obtain a parser that makes **consistent**, **algorithmic** choices.

Example Assume our parser expands the **leftmost** nonterminal by applying productions in the **order** in which they appear in the grammar.

Rule	Sentential Form	Input
	Expr	$\uparrow$ name + name $\times$ name
1	*Expr* + *Term*	$\uparrow$ name + name $\times$ name
1	*Expr* + *Expr* + *Term*	$\uparrow$ name + name $\times$ name
1	...	$\uparrow$ name + name $\times$ name

With this grammar and consistent choice, the parser will continue to expand the fringe **indefinitely** because that expansion never generates a leading terminal symbol.

Eliminating Left Recursion

This problem arises because the grammar uses **left recursion** in some of its productions.

$$\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
\mathit{Expr} & \rightarrow & \mathit{Expr} - \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
\mathit{Term} & \rightarrow & \mathit{Term} \div \mathit{Factor}
\end{array}$$

With left recursion, a top-down parser can loop indefinitely without generating a leading terminal symbol that the parser can match.

Fortunately, we can reformulate a left-recursive grammar so that it uses **right recursion**, *i.e.*, any recursion involves the rightmost symbol in a rule.

Eliminating Left Recursion

$$\begin{array}{rcl} A & \rightarrow & A\alpha \\ & | & \beta \end{array} \qquad\qquad \begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \\ & | & \epsilon \end{array}$$

The translation from (direct) left recursion to right recursion is mechanical

- introduce a new nonterminal A' and transfer the recursion onto A'
- add a rule $A' \rightarrow \epsilon$, where ϵ represents the empty string

Note To expand the production $A' \rightarrow \epsilon$, the parser simply sets `focus` $\leftarrow$ `pop()`, which advances its attention to the next node, terminal or nonterminal, on the fringe.

Eliminating Left Recursion

In the classic expression grammar, direct left recursion appears in the productions for both *Expr* and *Term*.

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
 & & \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
 & | & \mathit{Term} \div \mathit{Factor} \\
 & | & \mathit{Factor}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Term}\ \mathit{Expr'} \\
\mathit{Expr'} & \rightarrow & +\ \mathit{Term}\ \mathit{Expr'} \\
 & | & -\ \mathit{Term}\ \mathit{Expr'} \\
 & | & \epsilon \\
\mathit{Term} & \rightarrow & \mathit{Factor}\ \mathit{Term'} \\
\mathit{Term'} & \rightarrow & \times\ \mathit{Factor}\ \mathit{Term'} \\
 & | & \div\ \mathit{Factor}\ \mathit{Term'} \\
 & | & \epsilon
\end{array}
$$

We obtain the **right-recursive variant** of the classic expression grammar (*cf.* next slide) by inserting these replacements back into the original grammar.

Eliminating Left Recursion

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	$+$ *Term Expr'*
3		$\mid$	$-$ *Term Expr'*
4		$\mid$	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	$\times$ *Factor Term'*
7		$\mid$	$\div$ *Factor Term'*
8		$\mid$	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		$\mid$	num
11		$\mid$	name

This right-recursive grammar specifies the **same** set of expressions as the original left-recursive grammar

- it eliminates the problem with nontermination
- it **does not** avoid the need for backtracking

Example The next slide shows the behavior of a top-down parser using this grammar on the input $\mathsf{a} + \mathsf{b} \times \mathsf{c}$

- it still assume oracular choice
- number of steps is still proportional to derivation length plus input length

Rule	Sentential Form	Input
	Expr	$\uparrow$ name + name $\times$ name
1	*Term Expr'*	$\uparrow$ name + name $\times$ name
5	*Factor Term' Expr'*	$\uparrow$ name + name $\times$ name
11	name *Term' Expr'*	$\uparrow$ name + name $\times$ name
$\rightarrow$	name *Term' Expr'*	name $\uparrow$ + name $\times$ name
8	name *Expr'*	name $\uparrow$ + name $\times$ name
2	name + *Term Expr'*	name $\uparrow$ + name $\times$ name
$\rightarrow$	name + *Term Expr'*	name + $\uparrow$ name $\times$ name
5	name + *Factor Term' Expr'*	name + $\uparrow$ name $\times$ name
11	name + name *Term' Expr'*	name + $\uparrow$ name $\times$ name
$\rightarrow$	name + name *Term' Expr'*	name + name $\uparrow$ $\times$ name
6	name + name $\times$ *Factor Term' Expr'*	name + name $\uparrow$ $\times$ name
$\rightarrow$	name + name $\times$ *Factor Term' Expr'*	name + name $\times$ $\uparrow$ name
11	name + name $\times$ name *Term' Expr'*	name + name $\times$ $\uparrow$ name
$\rightarrow$	name + name $\times$ name *Term' Expr'*	name + name $\times$ name $\uparrow$
8	name + name $\times$ name *Expr'*	name + name $\times$ name $\uparrow$
4	name + name $\times$ name	name + name $\times$ name $\uparrow$

Eliminating Left Recursion

So far, we have only tackled **direct** left recursion. There can also be **indirect** left recusion, which is caused by chains of “transitive” productions.

$$\alpha \rightarrow \beta,\ \beta \rightarrow \gamma, \text{and } \gamma \rightarrow \alpha\delta \implies \alpha \rightarrow^{+} \alpha\delta$$

Indirect left recursion can be obscured by a long chain of productions. Therefore, we need a **more systematic approach** to convert indirect left recursion into right recursion.

We can eliminate all left recursion from a grammar using **two** simple techniques

- forward substitution to convert indirect left recursion into direct left recursion
- rewriting direct left recursion as right recursion

Eliminating Left Recursion

```
impose an arbitrary order on nonterminals A_1, A_2, ..., A_n
for i ← 1 to n do
    for j ← 1 to i − 1 do
        if there is a production A_i → A_j γ and A_j → δ_1|δ_2|...|δ_n then
            replace A_i → A_j γ with a set of productions A_i → δ_1γ|δ_2γ|...|δ_nγ
    rewrite the productions to eliminate any direct left recursion on A_i
```

Note This algorithm assumes that the original grammar has no cycles ($A \rightarrow^{+} A$) and no ϵ-productions.

Backtrack-Free Parsing

The need to backtrack is the **major source of inefficiency** in a leftmost, top-down parser.

Example With consistent choice, such as considering rules in order of appearance in the grammar on Slide 154, the parser would have backtracked on each `name`.

For this grammar, we can avoid backtracking with a simple modification

- when selecting a rule, the parser considers the focus symbol **and** the next symbol
- using one such **lookahead symbol**, the parser can disambiguate all of the choices

Backtrack-Free or Predictive Grammar

A context-free grammar for which the leftmost, top-down parser can **always** predict the correct rule with lookahead of at most one word.

Backtrack-Free Parsing

To avoid the need for backtracking in a parser, we need to understand what property makes a grammar backtrack-free.

> **Intuition**
>
> At each point in the parse, the choice of an expansion is obvious because **each** alternative for the leftmost nonterminal leads to a **distinct** terminal symbol.
>
> Comparing the next word against those choices reveals the correct expansion.

Formalizing this intuition will required some notation…

Backtrack-Free Parsing

FIRST

For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

The domain of FIRST is the set of grammar symbols, $\mathtt{T} \cup \mathtt{NT} \cup \{\epsilon, \mathtt{eof}\}$ and its range is $\mathtt{T} \cup \{\epsilon, \mathtt{eof}\}$.

$\alpha \in \mathtt{T} \cup \{\epsilon, \mathtt{eof}\}$	FIRST(α) has exactly one member α
$A \in \mathtt{NT}$	FIRST(A) contains all terminal symbols that can appear as the leading symbol in any sentential form derived from A

foreach $\alpha \in T \cup \{\epsilon, eof\}$ **do**

    $\text{FIRST}(\alpha) \leftarrow \alpha$

foreach $A \in NT$ **do**

    $\text{FIRST}(A) \leftarrow \emptyset$

while FIRST *sets are still changing* **do**

    foreach $p \in P$ *of the form* $A \rightarrow \beta_1\beta_2\ldots\beta_k$, *where* $\beta_i \in T \cup NT$ **do**

        $rhs \leftarrow \text{FIRST}(\beta_1) - \{\epsilon\}$

        $i \leftarrow 1$

        while $\epsilon \in \text{FIRST}(\beta_i)$ *and* $i \leq k - 1$ **do**

            $rhs \leftarrow rhs \cup (\text{FIRST}(\beta_{i+1}) - \{\epsilon\})$

            $i \leftarrow i + 1$

        if $i = k$ *and* $\epsilon \in \text{FIRST}(\beta_k)$ **then**

            $rhs \leftarrow rhs \cup \{\epsilon\}$

        $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup rhs$

Backtrack-Free Parsing

For the right recursive expression grammar shown on Slide 154, the initial step of the algorithm produces the following FIRST sets of the **terminal symbols**.

	num	name	$+$	$-$	$\times$	$\div$	(	)	eof	ϵ
FIRST	num	name	$+$	$-$	$\times$	$\div$	(	)	eof	ϵ

Once the fixed-point computation terminates, the FIRST sets of the **nonterminal symbols** are as follows.

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FIRST	(, num, name	$+, -, \epsilon$	(, num, name	$\times, \div, \epsilon$	(, num, name

Backtrack-Free Parsing

FIRST sets simplify the implementation of a top-down parser!

Example The parser tries to expand an *Expr'* using the rules of the right-recursive expression grammar.

It can use the lookahead symbol and the first sets to choose between Rules 2, 3, and 4.

$$\begin{array}{l|lcl} 2 & \textit{Expr'} & \rightarrow & + \textit{ Term Expr'} \\ 3 & & | & - \textit{ Term Expr'} \\ 4 & & | & \epsilon \end{array}$$

Symbol	Rule	Reason
$+$	2	$+ \in$ FIRST$(+$ *Term Expr'*$)$, $+ \notin$ FIRST$(-$ *Term Expr'*$)$, and $+ \notin$ FIRST(ϵ)
$-$	3	$- \notin$ FIRST$(+$ *Term Expr'*$)$, $- \in$ FIRST$(-$ *Term Expr'*$)$, and $- \notin$ FIRST(ϵ)

Rule 4, the ϵ-production, poses a slightly harder problem: FIRST(ϵ) is just $\{\epsilon\}$, which matches no word returned by the scanner.

Backtrack-Free Parsing

Dealing with ϵ-productions

- parser should apply the ϵ-production if the lookahead symbol is **not a member** of the FIRST set of any other alternative
- to differentiate between **legal input** and **syntax errors**, it needs to know which words can appear as the leading symbol after a valid application of an ϵ-production

FOLLOW

For a nonterminal A, FOLLOW(A) contains the set of words that can occur immediately after A in a sentence.

```
foreach A ∈ NT do
    FOLLOW(A) ← ∅
FOLLOW(S) ← {eof}
while FOLLOW sets are still changing do
    foreach p ∈ P of the form A → β1β2...βk, where βi ∈ T ∪ NT do
        lhs ← FOLLOW(A)
        for i ← k down to 1 do
            if βi ∈ NT then
                FOLLOW(βi) ← FOLLOW(βi) ∪ lhs
                if ϵ ∈ FIRST(βi) then
                    lhs ← lhs ∪ (FIRST(βi) − {ϵ})
                else
                    lhs ← lhs ∪ FIRST(βi)
            else
                lhs ← FIRST(βi)                    // Note that since βi ∉ NT, FIRST(βi) = {βi}
```

Backtrack-Free Parsing

Once the fixed-point computation terminates, the FOLLOW sets of the **nonterminal symbols** of the expression grammar are as follows.

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FOLLOW	$\texttt{eof},)$	$\texttt{eof},)$	$\texttt{eof}, +, -,)$	$\texttt{eof}, +, -,)$	$\texttt{eof}, +, -, \times, \div,)$

Example Recall the expansion of *Expr'* on Slide 163. The parser applies Rule 4 only if the lookahead symbol is in FOLLOW$($*Expr'*$)$, which contains `eof` and `)`. Any other symbol causes a syntax error.

Backtrack-Free Parsing

Using FIRST and FOLLOW, we can specify precisely the condition that makes a grammar backtrack free for a top-down parser.

Backtrack-Free Grammar

For a production $A \rightarrow \beta$, we define its augmented FIRST set, FIRST$^+$.

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

A grammar is **backtrack-free** if the following property holds for any nonterminal A with multiple right-hand sides, *i.e.*, $A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$.

$$\forall\ 1 \leq i, j \leq n, i \neq j : \text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset$$

Backtrack-Free Parsing

Example Is the right-recursive expression grammar backtrack-free?

	Rule	FIRST	FIRST$^+$
2	*Expr'* $\rightarrow$ + *Term Expr'*	$\{+\}$	$\{+\}$
3	*Expr'* $\rightarrow$ − *Term Expr'*	$\{-\}$	$\{-\}$
4	*Expr'* $\rightarrow$ ϵ	$\{\epsilon\}$	$\{\epsilon, \texttt{eof},)\}$
6	*Term'* $\rightarrow$ $\times$ *Factor Expr'*	$\{\times\}$	$\{\times\}$
7	*Term'* $\rightarrow$ $\div$ *Factor Expr'*	$\{\div\}$	$\{\div\}$
8	*Term'* $\rightarrow$ ϵ	$\{\epsilon\}$	$\{\epsilon, \texttt{eof}, +, -,)\}$
9	*Factor* $\rightarrow$ (*Expr*)	$\{(\}$	$\{(\}$
10	*Factor* $\rightarrow$ num	$\{\texttt{num}\}$	$\{\texttt{num}\}$
11	*Factor* $\rightarrow$ name	$\{\texttt{name}\}$	$\{\texttt{name}\}$

Note

We only need to consider the rules that have multiple right-hand sides

Only Rules 4 and 8 have FIRST$^+$ sets that differ from their FIRST sets

Intersecting the FIRST$^+$ sets of rules with alternate right-hand sides proves that the grammar is backtrack-free

Left-Factoring to Eliminate Backtracking

Not all grammars are backtrack-free. Assume we extend the expression grammar as shown to include function calls and array-element references.

$$
\begin{array}{r|rcl}
11 & \mathit{Factor} & \rightarrow & \texttt{name} \\
12 & & | & \texttt{name [} \mathit{ArgList} \texttt{]} \\
13 & & | & \texttt{name (} \mathit{ArgList} \texttt{)} \\
15 & \mathit{ArgList} & \rightarrow & \mathit{Expr\ MoreArgs} \\
16 & \mathit{MoreArgs} & \rightarrow & \texttt{,}\ \mathit{Expr\ MoreArgs} \\
17 & & | & \epsilon
\end{array}
$$

Because productions 11, 12, and 13 all begin with `name`, they have **identical** FIRST$^+$ sets. When expanding *Factor* with a lookahead of `name`, the parser may need to backtrack.

Note With a lookahead of two the need to backtrack can be avoided here.

Left-Factoring to Eliminate Backtracking

Fortunately, we can transform the problematic productions to create **disjoint** FIRST$^+$ sets.

$$
\begin{array}{r|rcl}
11 & \mathit{Factor} & \rightarrow & \texttt{name}\ \mathit{Arguments} \\
12 & \mathit{Arguments} & \rightarrow & [\ \mathit{ArgList}\ ] \\
13 & & | & (\ \mathit{ArgList}\ ) \\
14 & & | & \epsilon
\end{array}
$$

The rewrite adds a new nonterminal *Arguments* and pushes the alternate suffixes for *Factor* into right-hand sides for *Arguments*.

The process of extracting and isolating common prefixes in a set of productions is called **left-factoring**. Left-factoring can often eliminate the need to backtrack.

Note In general it is undecidable whether or not a backtrack-free grammar exists for an arbitrary context-free language.

Left-Factoring to Eliminate Backtracking

We can left factor any set of rules that has alternate right-hand sides with a common prefix.

$$A \rightarrow \alpha\beta_1|\alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|\gamma_2|\ldots|\gamma_m$$

The transformation introduces a new nonterminal B to represent the alternate suffixes for α and rewrites the original productions according to the following pattern.

$$\begin{aligned} A &\rightarrow \alpha B|\gamma_1|\gamma_2|\ldots|\gamma_m \\ B &\rightarrow \beta_1|\beta_1|\ldots|\beta_n \end{aligned}$$

To left factor a complete grammar, we must inspect each nonterminal, discover common prefixes, and apply the transformation in a **systematic** way.

Top-Down Recursive-Descent Parsers

Backtrack-free grammars lend themselves to simple and efficient parsing with a paradigm called **recursive descent**.

Constructing a recursive-descent parser

- for each nonterminal, construct a procedure to recognize its right-hand sides
- these mutually recursive procedures call one another to recognize nonterminals
- recognize terminals by direct matching

Example Consider the three rules for *Expr'* in the right-recursive expression grammar.

$$
\begin{array}{r|lcl}
2 & \textit{Expr'} & \rightarrow & + \ \textit{Term Expr'} \\
3 & & | & - \ \textit{Term Expr'} \\
4 & & | & \epsilon
\end{array}
$$

```
procedure Expr'()
    // Expr' → + Term Expr' | − Term Expr'
    if word = + or word = − then
        word ← NextWord()
        if Term() then
            return Expr'()
        else
            return false
    // Expr' → ε
    else if word = ) or word = eof then
        return true
    // no match
    else
        report a syntax error
        return false
```

Table-Driven LL(1) Parsers

As the FIRST^{+} sets completely dictate the parsing decisions, we can automatically generate efficient top-down parsers for backtrack-free grammars.

LL(1) Parser

- scans input Left to right
- constructs a Leftmost derivation
- uses a lookahead of 1 symbol

Grammars that work in an LL(1) scheme are often called LL(1) grammars. By definition, LL(1) grammars are backtrack-free.

The most common implementation technique for an LL(1) parser generator uses a **table-driven skeleton parser**.

```
word ← NextWord()
stack.push(eof)
stack.push(S)
focus ← stack.peek()
loop
    if focus = eof and word = eof then report success and exit the loop
    else if focus ∈ T or focus = eof then
        if focus matches word then
            stack.pop()
            word ← NextWord()
        else report an error looking for symbol at top of stack
    else
        if table[focus, word] is A → B1B2...Bk then
            stack.pop()
            for i ← k down to 1 do
                if Bi ≠ ϵ then stack.push(Bi)
        else report an error expanding focus
    focus ← stack.peek()
```

Table-Driven LL(1) Parsers

Given a nonterminal A and a lookahead symbol w, `table`$[A, w]$ specifies the correct expansion.

The algorithm to build `table` (shown on the right) is straightforward.

If the grammar meets the backtrack-free condition, the algorithm will produce the correct `table` in $\mathcal{O}(|\mathrm{P}| \times |\mathrm{T}|)$ time.

If the grammar is not backtrack-free, the algorithm will try to assign more than one production to some elements of `table`.

build FIRST, FOLLOW, *and* FIRST^+ *sets*
foreach $A \in \mathrm{NT}$ **do**
    foreach $w \in \mathrm{T}$ **do**
        `table`$[A, w] \leftarrow$ `error`
    foreach $p \in \mathrm{P}$ *with form* $A \rightarrow \beta$ **do**
        foreach $w \in \mathrm{FIRST}^+(A \rightarrow \beta)$ **do**
            `table`$[A, w] \leftarrow p$
        if `eof` $\in \mathrm{FIRST}^+(A \rightarrow \beta)$ **then**
            `table`$[A,$ `eof`$] \leftarrow p$

Table-Driven LL(1) Parsers

Example The LL(1) parse table for the right-recursive expression grammar is shown below. Productions are denoted by their numbers, — denotes an error.

	`eof`	$+$	$-$	$\times$	$\div$	(	)	`name`	`num`
Goal	—	—	—	—	—	0	—	0	0
Expr	—	—	—	—	—	1	—	1	1
Expr'	4	2	3	—	—	—	4	—	—
Term	—	—	—	—	—	5	—	5	5
Term'	8	8	8	6	7	—	8	—	—
Factor	—	—	—	—	—	9	—	11	10

Table-Driven LL(1) Parsers

Example

The table on the right shows the actions of the LL(1) expression parser for the input string $a + b \times c$.

The central column shows the contents of the stack, which holds the partially completed lower fringe of the parse tree.

The parse concludes successfully when it pops *Expr'* from the stack, leaving `eof` exposed on the stack and `eof` as the next symbol.

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

Table-Driven LL(1) Parsers

Example

Consider the actions of the LL(1) parser on the **illegal** input string $\mathtt{x} + \div \mathtt{y}$.

It detects the **syntax error** when it attempts to expand a nonterminal *Term* with lookahead symbol $\div$.

Looking up `table[`*Term*`,`$\div$`]` returns "—", which indicates this syntax error.

Rule	Stack	Input
—	`eof` *Goal*	$\uparrow$ name $+ \div$ name
0	`eof` *Expr*	$\uparrow$ name $+ \div$ name
1	`eof` *Expr' Term*	$\uparrow$ name $+ \div$ name
5	`eof` *Expr' Term' Factor*	$\uparrow$ name $+ \div$ name
11	`eof` *Expr' Term'* name	$\uparrow$ name $+ \div$ name
$\rightarrow$	`eof` *Expr' Term'*	name $\uparrow + \div$ name
8	`eof` *Expr'*	name $\uparrow + \div$ name
2	`eof` *Expr' Term* $+$	name $\uparrow + \div$ name
$\rightarrow$	`eof` *Expr' Term*	name $+ \uparrow \div$ name

Direct-Coded LL(1) Parsers

In analogy to direct-coded scanners (*cf.* Slide 109), an LL(1) parser generator could also emit a **direct-coded parser**.

Building a direct-coded parser

- build FIRST, FOLLOW, and FIRST^{+} sets
- iterate through the grammar in the same way as the table-construction algorithm
- for each nonterminal, generate a procedure that recognizes all its right-hand sides

Direct-coded parsers have the same speed and locality advantages as recursive-descent parsers and direct-coded scanners, but retain the advantages of a grammar-generated system, such as a concise, high-level specification and reduced implementation effort.