3 Parsers

Chapter Contents

1. Expressing Syntax
2. Top-Down Parsing
   - Top-Down Recursive-Descent Parsers
   - Table-Driven LL(1) Parsers
3. Bottom-Up Parsing
   - The LR(1) Parsing Algorithm

Parsers

The parser determines if the input program, given as the stream of classified words produced by the scanner, is a **valid** sentence in the programming language.

In order to build a parser, we need…

- a **formal mechanism** (grammar) for specifying the syntax of the source language
- a **systematic method** of determining membership in this formally specified language

Parsing

Given a stream s of words and a grammar G, the process of finding a derivation in G that produces s is called *parsing*.

Expressing Syntax

As a problem, parsing is **very similar** to scanning. Therefore, we could be tempted to reuse the techniques introduced in the previous chapter.

For example, we could try to specify the syntax of the programming language using REs. However, REs lack the **expressive power** to describe the full syntax of most programming languages.

In this chapter, we will therefore introduce **context-free grammars**, which are used to express the syntax of most programming languages.

Why Not Regular Expressions?

Consider the problem of recognizing algebraic expressions over variables and the operators $+$, $-$, $\times$, and $\div$. To do so, we could define the following RE.

$$[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\,((+\mid-\mid\times\mid\div)\,[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\,)^*$$

This RE matches `a` $+$ `b` $\times$ `c` and `huey` $\div$ `dewey` $\times$ `louie`, but it does not suggest operator precedence.

To enforce other evaluation orders, normal algebraic notation includes parentheses. We could update our RE as follows.

$$(\,(\mid\epsilon\,)\,[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\,((+\mid-\mid\times\mid\div)\,[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\,(\,)\mid\epsilon\,)\,)^*$$

This RE matches both `a` $+$ `b` $\times$ `c` and `(a` $+$ `b)` $\times$ `c`. Problem solved?

Why Not Regular Expressions?

Unfortunately, the RE also matches many syntactically **incorrect** expressions, such as $\mathtt{a} + \mathtt{(b} \times \mathtt{c}$ and $\mathtt{a} + \mathtt{b)} \times \mathtt{c)}$.

We cannot write an RE that will match all expressions with balanced parentheses, because the language $(^m)^n$, where $m = n$ is **not regular**[2].

This is fundamental limitation of REs stems from the fact that the corresponding recognizers **cannot count** because they have only a finite set of states.

Note Paired constructs, such as `begin` and `end` or `then` and `else`, play an important role in most programming languages

[2] This can be shown with a simple proof based on the Pumping Lemma.

Context-Free Grammars

We need a more powerful notation that still leads to efficient recognizers. The traditional solution is to use a **Context-Free Grammar (CFG)**.

- a context-free grammar G is a set of rules or **productions** that describe how to derive sentences
- the collection of all sentences that can be derived from G is called **language defined by** G, denoted $L(G)$

The set of all languages defined by context-free grammars is called the set of **context-free languages**

Context-Free Grammars

Example Consider the following grammar, which we call BL.

$$
\begin{array}{r|lcl}
1 & \mathit{BatmanLyrics} & \rightarrow & \texttt{Nah}\ \mathit{BatmanLyrics} \\
2 & & | & \texttt{Batman!}
\end{array}
$$

The first production says “*BatmanLyrics* can derive the word `Nah`, followed by one or more *BatmanLyrics*”. The second rule reads “*BatmanLyrics* can also derive the word `Batman!`”.

- **nonterminal symbol**: syntactic variable representing a set of strings that can be derived from the grammar, *e.g.*, *BatmanLyrics*
- **terminal symbol**: word in the language defined by the grammar, *e.g.*, `Nah` and `Batman!`

Context-Free Grammars

To understand the relationship between a grammar G and the language it defines $\mathrm{L}(\mathrm{G})$, we need to specify how the productions in G are applied to derive sentences in $\mathrm{L}(\mathrm{G})$.

First, we must identify the **goal symbol** or **start symbol** of G

- represents the set of all strings in $\mathrm{L}(\mathrm{G})$
- cannot be one of the words in the language
- must be one of the nonterminal symbols introduced to add structure and abstraction

Example Since BL only has one nonterminal symbol, *BatmanLyrics* must be goal symbol.

Context-Free Grammars

Formally, a **Context-Free Grammar** G is a quadruple (T, NT, S, P) where

T is the set of terminal symbols, or words, in the language $L(G)$.

NT is the set of nonterminal symbols that appear in the productions of G.

S is a nonterminal designated as the **goal symbol** or **start symbol** of the grammar. S represents the set of sentences in $L(G)$.

P is the set of productions or rewrite rules in G.
Each rule in P has the form $NT \rightarrow (T \cup NT)^+$, *i.e.*, it replaces a **single nonterminal** with a string of one or more grammar symbols.

Deriving Sentences

A **derivation** is a sequence of rewriting steps that begins with the grammar's goal symbol and ends with a sentence in the language.

1. start with a prototype string that contains just the goal symbol
2. pick a nonterminal symbol α in the prototype string
3. choose a grammar rule $\alpha \rightarrow \beta$
4. rewrite α with β
5. repeat until there are no more nonterminal symbols left

A string of symbols that occurs as one step in a valid derivation is called **sentential form**

- any sentential form can be derived from the start symbol in zero or more steps
- similarly, from any sentential form we can derive a valid sentence in zero or more steps

Deriving Sentences

Example We demonstrate derivation using the grammar BL from before.

1. we start with the goal symbol *BatmanLyrics*
2. we can rewrite *BatmanLyrics* with either Rule 1 or 2
   - Rule 2: the string becomes `Batman!` and has no further rewritings, *i.e.*, `Batman!` is a valid sentence in $\mathrm{L}(\mathrm{BL})$
   - Rule 1: the string becomes `Nah` *BatmanLyrics*, which has one nonterminal left; rewriting it with Rule 2 leads to `Nah Batman!`, another sentence in $\mathrm{L}(\mathrm{G})$

Deriving Sentences

In the following, we will often represent such derivations in tabular form.

Rule	Sentential Form
	BatmanLyrics
2	`Batman!`

Rule	Sentential Form
	BatmanLyrics
1	`Nah` *BatmanLyrics*
2	`Nah Batman!`

As a notational convenience, we will use $\rightarrow^+$ to mean “derives in one or more steps”.

- *BatmanLyrics* $\rightarrow^+$ `Batman!`
- *BatmanLyrics* $\rightarrow^+$ `Nah Batman!`

More Complex Examples

The *BatmanLyrics* grammar is too simple to exhibit the **power** and **complexity** of CFGs. Instead, we revisit the example that showed the shortcomings of REs.

$$
\begin{array}{r|rcl}
1 & \mathit{Expr} & \rightarrow & (\ \mathit{Expr}\ ) \\
2 & & | & \mathit{Expr}\ \mathit{Op}\ \texttt{name} \\
3 & & | & \texttt{name} \\
4 & \mathit{Op} & \rightarrow & + \\
5 & & | & - \\
6 & & | & \times \\
7 & & | & \div
\end{array}
$$

Note The goal symbol of this grammer is *Expr*.

More Complex Examples

To generate the sentence $(\mathtt{a} + \mathtt{b}) \times \mathtt{c}$, we can use the following rewrite sequence.

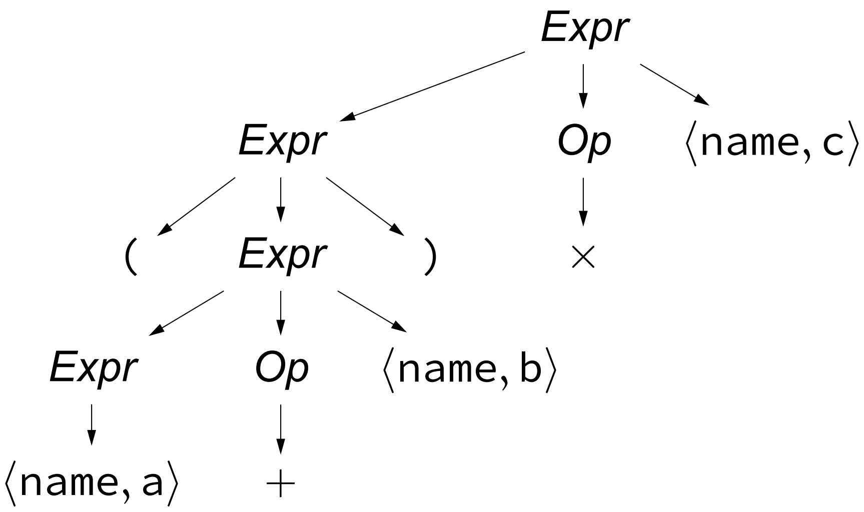
Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	−
6		|	×
7		|	÷

Recall Grammars deal with syntactic categories (name), rather than lexemes (a, b, c).

Parse Tree

Derivations can also be represented as a graph, *i.e.*, as a **parse tree** or a **syntax tree**.



Different Dervivation Orders

Note This simple CFG for expressions **cannot** generate a sentence with unbalanced or improperly nested parentheses.

- only Rule 1 can generate an opening parenthesis
- but Rule 1 also generates the matching closing parenthesis

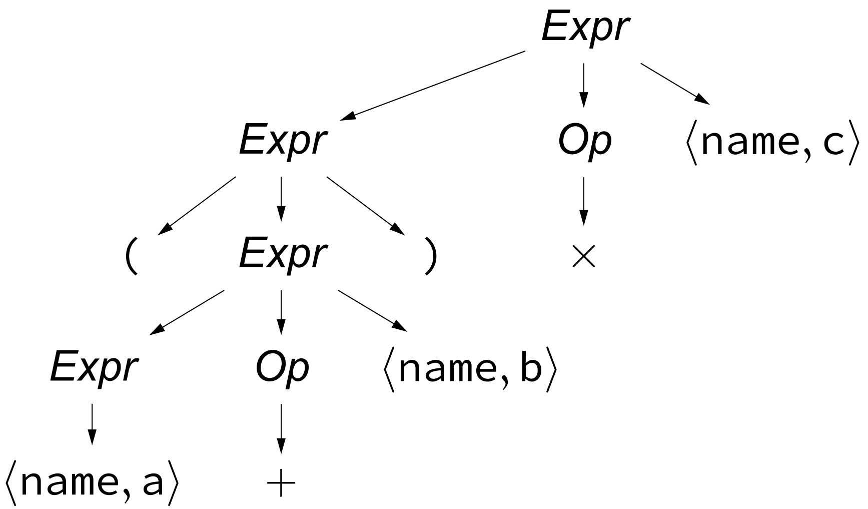
In the derivation on Slide 129, we rewrote the rightmost remaining nonterminal symbol at each step. One obvious alternative is to rewrite the leftmost nonterminal at each step.

Both choices are valid. They are called **rightmost derivation** and **leftmost derivation**, respectively.

Different Derivation Orders

The leftmost derivation of $(\mathtt{a} + \mathtt{b}) \times \mathtt{c}$ is as follows.

Rule	Sentential Form
	Expr
2	*Expr Op* name
1	(*Expr*) *Op* name
2	(*Expr Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name



Note The parse tree is **identical** to the one before, since it does not represent the order in which the productions were applied.

Ambiguous Grammars

It is important that each sentence in the language defined by a CFG has a **unique** rightmost (or leftmost) derivation.

Ambiguous Grammar

A grammar G is ambiguous if some sentence in $\mathrm{L}(\mathrm{G})$ has more than one rightmost (or leftmost) derivation.

An ambiguous grammar can produce multiple derivations and multiple parse trees. Multiple parse trees imply **multiple possible meanings** for a single program!

Ambiguous Grammars

A classic example of an ambiguous construct is the so-called *“dangling else”* problem.

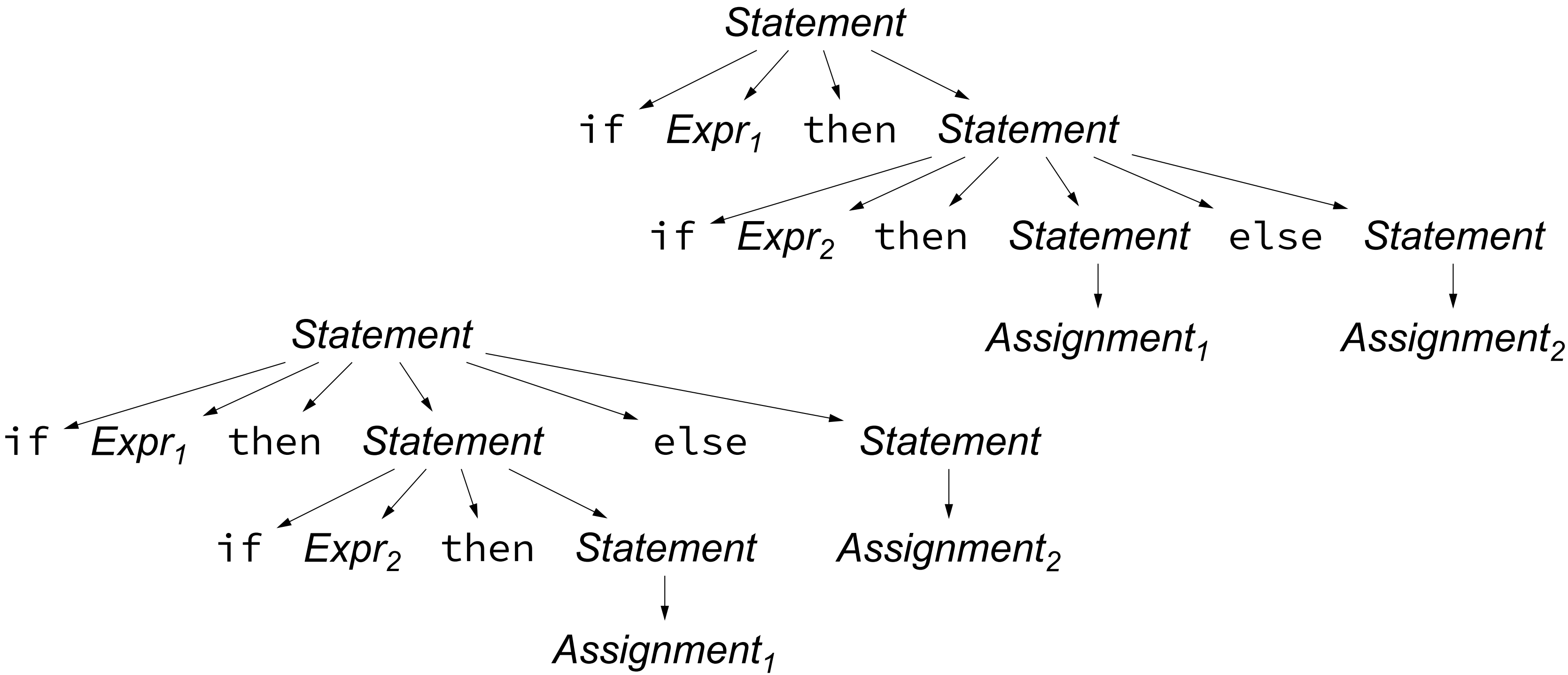
$$\begin{array}{r|lcl}
1 & \textit{Statement} & \rightarrow & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement}\ \texttt{else}\ \textit{Statement} \\
2 & & | & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement} \\
3 & & | & \textit{Assignment} \\
4 & & | & \ldots\textit{other statements}\ldots
\end{array}$$

This grammar fragment shows that the `else` is optional.

Problem The following line of code has **two distinct** rightmost derivations.

$$\texttt{if}\ \textit{Expr}_1\ \texttt{then}\ \texttt{if}\ \textit{Expr}_2\ \texttt{then}\ \textit{Assignment}_1\ \texttt{else}\ \textit{Assignment}_2$$

Ambiguous Grammars



Ambiguous Grammars

To remove this ambiguity, the grammar must be modified to encode a rule that determines which `if` controls an `else`.

$$
\begin{array}{r|rcl}
1 & \mathit{Statement} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Statement} \\
2 & & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{WithElse}\ \texttt{else}\ \mathit{Statement} \\
3 & & \mid & \mathit{Assignment} \\
4 & \mathit{WithElse} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{WithElse}\ \texttt{else}\ \mathit{WithElse} \\
5 & & \mid & \mathit{Assignment}
\end{array}
$$

The solution **restricts** the set of statements that can occur in the `then` part of an `if-then-else` construct.

- accepts the same set of sentences as the original grammar
- ensures that each `else` has an unambiguous match to a specific `if`
- encodes a simple rule: bind each `else` to the innermost unclosed `if`

Ambiguous Grammars

The modified grammar has **only one** rightmost derivation for the example.

Rule	**Sentential Form**
	Statement
1	`if` *Expr* `then` *Statement*
2	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Statement*
3	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Assignment*
5	`if` *Expr* `then` `if` *Expr* `then` *Assignment* `else` *Assignment*

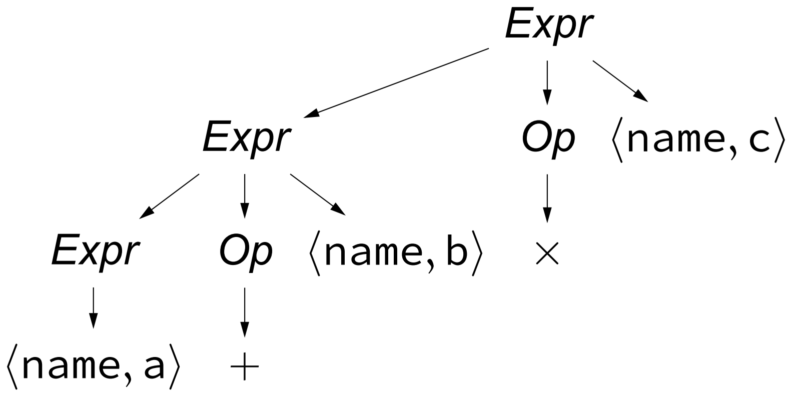
The `if-then-else` ambiguity points out the relationship between meaning and grammatical structure.

Encoding Meaning into Structure

Ambiguity is not the only situation where meaning and grammatical structure interact.

Consider the parse tree that would be built from a rightmost derivation of the simple expression a + b × c.

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
2	*Expr Op* name × name
4	*Expr* + name × name
3	name + name × name



Encoding Meaning into Structure

One natural way to evaluate the expression is with a simple **postorder** treewalk.

- addition is performed **before** multiplication, *i.e.*, $(a + b) \times c$
- this evaluation **contradicts** rules of algebraic precedence, *i.e.*, $a + (b \times c)$

The problem lies in the **structure** of the grammar: it treats all of the arithmetic operators in the same way, without any regard for precedence.

Recall the parse tree for $(a + b) \times c$, shown on Slide 129

- the parenthetic subexpression adds an **extra level** to the parse tree by being forced to go through an **extra production** (Rule 1)
- this extra level would force a postorder treewalk to evaluate the parenthetic subexpression **before** it evaluates the multiplication

→ We can use this effect to encode operator precedence levels into the grammar.

Encoding Meaning into Structure

In the simple expression grammar, we have **three** levels of precedence

1. **highest precedence** for ( )
2. **medium precedence** for × and ÷
3. **lowest precedence** for + and −

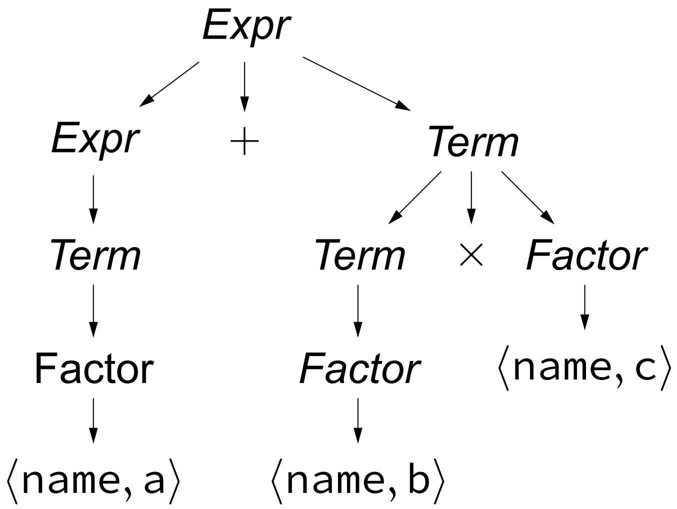
Approach

- group the operators at **distinct** levels
- use a nonterminal to **isolate** the corresponding part of the grammar

$$
\begin{array}{r|rcl}
0 & \mathit{Goal} & \rightarrow & \mathit{Expr} \\
1 & \mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
2 & & | & \mathit{Expr} - \mathit{Term} \\
3 & & | & \mathit{Term} \\
4 & \mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
5 & & | & \mathit{Term} \div \mathit{Factor} \\
6 & & | & \mathit{Factor} \\
7 & \mathit{Factor} & \rightarrow & (\ \mathit{Expr}\ ) \\
8 & & | & \texttt{num} \\
9 & & | & \texttt{name}
\end{array}
$$

Encoding Meaning into Structure

Rule	Sentential Form
	Expr
1	*Expr* + *Term*
4	*Expr* + *Term* × *Factor*
9	*Expr* + *Term* × name
6	*Expr* + *Factor* × name
9	*Expr* + name × name
3	*Term* + name × name
6	*Factor* + name × name
9	name + name × name



In this form, the grammar derives a parse tree for a + b × c that is **consistent** with standard algebraic precedence.

Encoding Meaning into Structure

Note A postorder treewalk over this parse tree will first evaluate b × c and then add the result to a.

- this implements the standard rules of arithmetic precedence
- using nonterminals to enforce precedence **adds** interior nodes to the parse tree
- substituting the individual operators for occurrences of *Op* **removes** interior nodes

We can use this **trick** to ensure precedence elsewhere

- **array subscripts** should be applied before standard arithmetic operations
- **type casts** have higher precedence than arithmetic but lower precedence than parentheses or subscripting operations
- **assignment operator** should have lower precedence than arithmetic operations operations

Discovering a Derivation for an Input String

The process of constructing a derivation from a specific input sentence is called **parsing**.

If the language is unambiguous, we can think of the parse tree as the parser's output.

- **root** of parse tree is know as it is given by the goal symbol of grammar
- **leaves** of parse tree are known as they must match the output of the scanner

Two distinct and opposite approaches for constructing the tree suggest themselves

1. **Top-down parsers** begin with the root and grow the tree toward the leaves
2. **Bottom-up parsers** begin with the leaves and grow the tree toward the root

In either scenario, the parser makes a **series of choices** about which productions to apply. Most of the complexity in parsing lies in the mechanisms for making these choices.

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
        - → parsing succeeds

    b) clear mismatch occurs between fringe and input stream
        - → backtrack and try another production
        - → 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 ($A \rightarrow \beta_1, \beta_2, \ldots, \beta_n$)
        build nodes for $\beta_1, \beta_2, \ldots, \beta_n$ as children of focus
        push($\beta_n, \beta_{n-1}, \ldots, \beta_2$)
        focus ← $\beta_1$
    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	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
3	*Term* + *Term*	↑ name + name × name
6	*Factor* + *Term*	↑ name + name × name
9	name + *Term*	↑ name + name × name
→	name + *Term*	name ↑ + name × name
→	name + *Term*	name + ↑ name × name
4	name + *Term* × *Factor*	name + ↑ name × name
6	name + *Factor* × *Factor*	name + ↑ name × name
9	name + name × *Factor*	name + ↑ name × name
→	name + name × *Factor*	name + name ↑ × name
→	name + name × *Factor*	name + name × ↑ name
9	name + name × name	name + name × ↑ name
→	name + name × name	name + name × name ↑

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 × name
1	*Expr* + *Term*	$\uparrow$ name + name × name
1	*Expr* + *Expr* + *Term*	$\uparrow$ name + name × name
1	…	$\uparrow$ name + name × 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.

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

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*.

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

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 a + b × c

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

Rule	Sentential Form	Input
	Expr	↑ name + name × name
1	*Term Expr’*	↑ name + name × name
5	*Factor Term’ Expr’*	↑ name + name × name
11	name *Term’ Expr’*	↑ name + name × name
→	name *Term’ Expr’*	name ↑ + name × name
8	name *Expr’*	name ↑ + name × name
2	name + *Term Expr’*	name ↑ + name × name
→	name + *Term Expr’*	name + ↑ name × name
5	name + *Factor Term’ Expr’*	name + ↑ name × name
11	name + name *Term’ Expr’*	name + ↑ name × name
→	name + name *Term’ Expr’*	name + name ↑ × name
6	name + name × *Factor Term’ Expr’*	name + name ↑ × name
→	name + name × *Factor Term’ Expr’*	name + name × ↑ name
11	name + name × name *Term’ Expr’*	name + name × ↑ name
→	name + name × name *Term’ Expr’*	name + name × name ↑
8	name + name × name *Expr’*	name + name × name ↑
4	name + name × name	name + name × name ↑

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 $A_1, A_2, \ldots, A_n$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $i - 1$ do
        if there is a production $A_i \rightarrow A_j\gamma$ and $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_n$ then
            replace $A_i \rightarrow A_j\gamma$ with a set of productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_n\gamma$
    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, $\mathrm{T} \cup \mathrm{NT} \cup \{\epsilon, \mathtt{eof}\}$ and its range is $\mathrm{T} \cup \{\epsilon, \mathtt{eof}\}$.

$\alpha \in \mathrm{T} \cup \{\epsilon, \mathtt{eof}\}$ FIRST(α) has exactly one member α

$A \in \mathrm{NT}$ 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**
    $\mathsf{FIRST}(\alpha) \leftarrow \alpha$
**foreach** $A \in NT$ **do**
    $\mathsf{FIRST}(A) \leftarrow \emptyset$
**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**
        $rhs \leftarrow \mathsf{FIRST}(\beta_1) - \{\epsilon\}$
        $i \leftarrow 1$
        **while** $\epsilon \in \mathsf{FIRST}(\beta_i)$ *and* $i \leq k-1$ **do**
            $rhs \leftarrow rhs \cup (\mathsf{FIRST}(\beta_{i+1}) - \{\epsilon\})$
            $i \leftarrow i+1$
        **if** $i = k$ *and* $\epsilon \in \mathsf{FIRST}(\beta_k)$ **then**
            $rhs \leftarrow rhs \cup \{\epsilon\}$
        $\mathsf{FIRST}(A) \leftarrow \mathsf{FIRST}(A) \cup rhs$
```

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.

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

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

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 $A \in NT$ do
    FOLLOW($A$) $\leftarrow \emptyset$
FOLLOW($S$) $\leftarrow$ {eof}
while FOLLOW *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
        lhs $\leftarrow$ FOLLOW($A$)
        for $i \leftarrow k$ *down to* 1 do
            if $\beta_i \in NT$ then
                FOLLOW($\beta_i$) $\leftarrow$ FOLLOW($\beta_i$) $\cup$ lhs
                if $\epsilon \in$ FIRST($\beta_i$) then
                    lhs $\leftarrow$ lhs $\cup$ (FIRST($\beta_i$) $-$ {$\epsilon$})
                else
                    lhs $\leftarrow$ lhs $\cup$ FIRST($\beta_i$)
            else
                lhs $\leftarrow$ FIRST($\beta_i$)                // Note that since $\beta_i \notin NT$, FIRST($\beta_i$) $=$ {$\beta_i$}
```

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	`eof`,)	`eof`,)	`eof`, $+$, $-$,)	`eof`, $+$, $-$,)	`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'* → + *Term Expr'*	$\{+\}$	$\{+\}$
3	*Expr'* → − *Term Expr'*	$\{-\}$	$\{-\}$
4	*Expr'* → ϵ	$\{\epsilon\}$	$\{\epsilon, \texttt{eof},)\}$
6	*Term'* → × *Factor Expr'*	$\{\times\}$	$\{\times\}$
7	*Term'* → ÷ *Factor Expr'*	$\{\div\}$	$\{\div\}$
8	*Term'* → ϵ	$\{\epsilon\}$	$\{\epsilon, \texttt{eof}, +, -,)\}$
9	*Factor* → (*Expr*)	$\{(\}$	$\{(\}$
10	*Factor* → num	$\{\texttt{num}\}$	$\{\texttt{num}\}$
11	*Factor* → 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.

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

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.

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

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.

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

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

$$\begin{array}{lcl} A & \longrightarrow & \alpha B|\gamma_1|\gamma_2|\ldots|\gamma_m \\ B & \longrightarrow & \beta_1|\beta_1|\ldots|\beta_n \end{array}$$

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' $\rightarrow$ + Term Expr' | $-$ Term Expr'
    if word $=$ $+$ or word $=$ $-$ then
        word $\leftarrow$ NextWord()
        if Term() then
            return Expr'()
        else
            return false
    // Expr' $\rightarrow$ $\epsilon$
    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 $A \rightarrow B_1 B_2 \ldots B_k$ then
            stack.pop()
            for $i \leftarrow k$ down to 1 do
                if $B_i \neq \epsilon$ then stack.push($B_i$)
        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 $\mathcal{O}(|P| \times |T|)$ 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 $A \in NT$ do
    foreach $w \in T$ do
        table[$A$, $w$] $\leftarrow$ error
    foreach $p \in P$ with form $A \rightarrow \beta$ do
        foreach $w \in$ FIRST$^+(A \rightarrow \beta)$ do
            table[$A$, $w$] $\leftarrow p$
        if eof $\in$ 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	+	−	×	÷	(	)	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 × 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 `x` + ÷ `y`.

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

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

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' Term*	`name` + ↑ ÷ `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.