3 Parsers

Chapter Contents

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 $\mathtt{a} + \mathtt{b} \times \mathtt{c}$ and $\mathtt{huey} \div \mathtt{dewey} \times \mathtt{louie}$, 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 $\mathtt{a} + \mathtt{b} \times \mathtt{c}$ and $\mathtt{(a} + \mathtt{b)} \times \mathtt{c}$. Problem solved?

Why Not Regular Expressions?

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

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.

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

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 $(\mathrm{T}, \mathrm{NT}, S, \mathrm{P})$ where

- T is the set of terminal symbols, or words, in the language $\mathrm{L}(\mathrm{G})$.
- 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 $\mathrm{L}(\mathrm{G})$.
- P is the set of productions or rewrite rules in G. Each rule in P has the form $\mathrm{NT} \rightarrow (\mathrm{T} \cup \mathrm{NT})^+$, *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 $(\mathsf{a} + \mathsf{b}) \times \mathsf{c}$, we can use the following rewrite sequence.

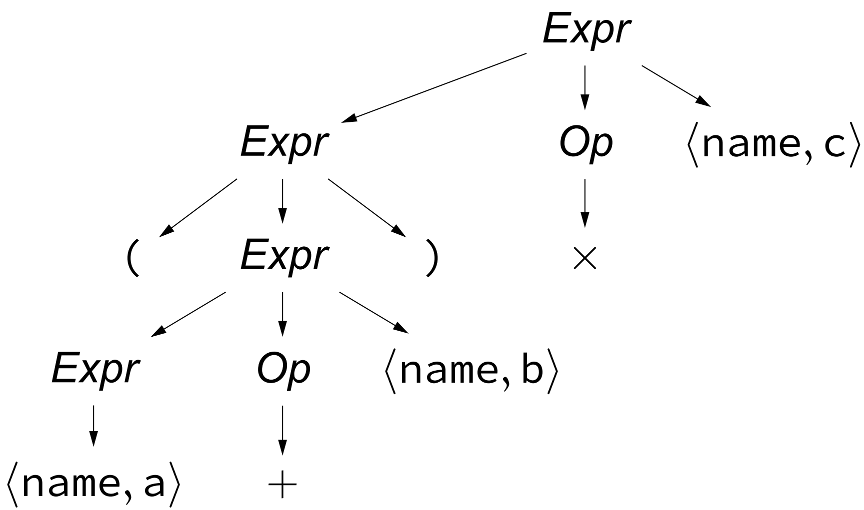
Rule	Sentential Form
	Expr
2	$\mathit{Expr}\ \mathit{Op}\ \mathsf{name}$
6	$\mathit{Expr} \times \mathsf{name}$
1	$(\ \mathit{Expr}\ ) \times \mathsf{name}$
2	$(\ \mathit{Expr}\ \mathit{Op}\ \mathsf{name}\ ) \times \mathsf{name}$
4	$(\ \mathit{Expr} + \mathsf{name}\ ) \times \mathsf{name}$
3	$(\ \mathsf{name} + \mathsf{name}\ ) \times \mathsf{name}$

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

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

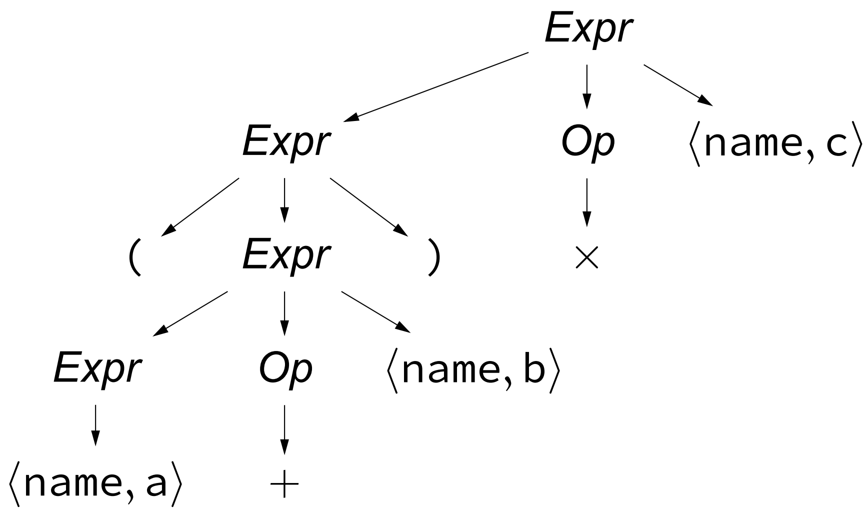
In the derivation on Slide 129, we rewrote the rightmost remaining nonterminal symbol at each step. One obvious alternative is to rewrite the leftmost nonterminal at each step.

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

Different Derivation Orders

The leftmost derivation of $(a + b) \times c$ 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 $L(G)$ 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.

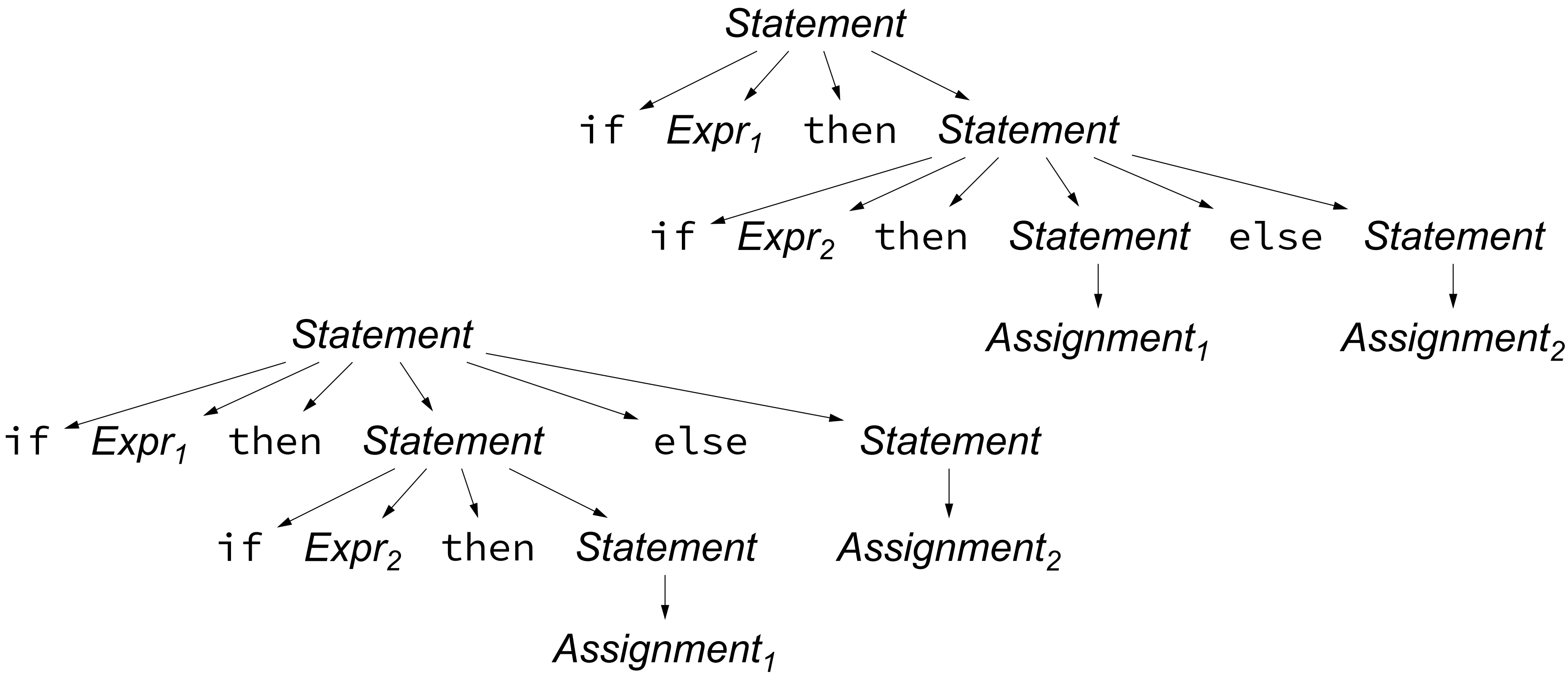
$$
\begin{array}{r|lcl}
1 & \textit{Statement} & \rightarrow & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement}\ \texttt{else}\ \textit{Statement} \\
2 & & | & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement} \\
3 & & | & \textit{Assignment} \\
4 & & | & \ldots\textit{other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

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`.

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

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*

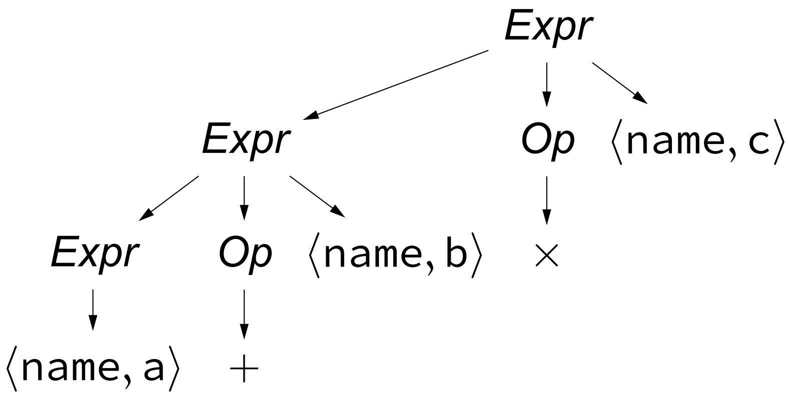
The `if-then-else` ambiguity points out the relationship between meaning and grammatical structure.

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 $\mathtt{a} + \mathtt{b} \times \mathtt{c}$.

Rule	**Sentential Form**
	Expr
2	*Expr Op* name
6	*Expr* $\times$ name
2	*Expr Op* name $\times$ name
4	*Expr* $+$ name $\times$ name
3	name $+$ name $\times$ 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.*, $(\mathtt{a} + \mathtt{b}) \times \mathtt{c}$
- this evaluation **contradicts** rules of algebraic precedence, *i.e.*, $\mathtt{a} + (\mathtt{b} \times \mathtt{c})$

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 $(\mathtt{a} + \mathtt{b}) \times \mathtt{c}$, 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

$\rightarrow$ 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 $\times$ and $\div$
3. **lowest precedence** for $+$ and $-$

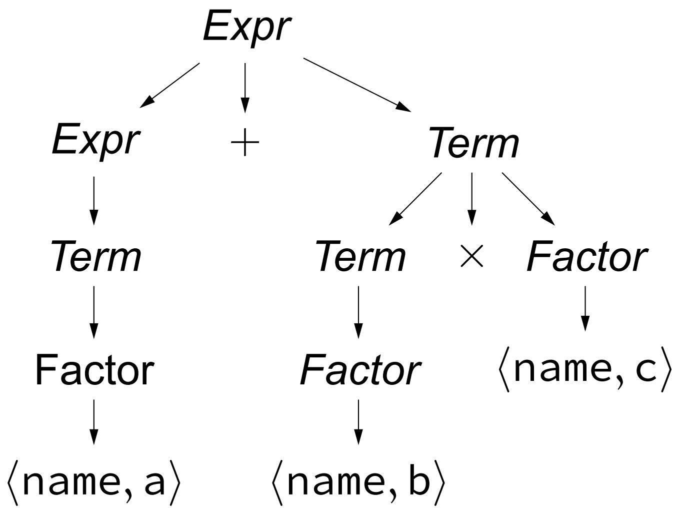
Approach

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

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

Encoding Meaning into Structure

Rule	Sentential Form
	Expr
1	$\mathit{Expr} + \mathit{Term}$
4	$\mathit{Expr} + \mathit{Term} \times \mathit{Factor}$
9	$\mathit{Expr} + \mathit{Term} \times \text{name}$
6	$\mathit{Expr} + \mathit{Factor} \times \text{name}$
9	$\mathit{Expr} + \text{name} \times \text{name}$
3	$\mathit{Term} + \text{name} \times \text{name}$
6	$\mathit{Factor} + \text{name} \times \text{name}$
9	$\text{name} + \text{name} \times \text{name}$



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

Encoding Meaning into Structure

Note A postorder treewalk over this parse tree will first evaluate $b \times c$ 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.