CSE 311 Lecture 21: Context-Free Grammars

Emina Torlak and Sami Davies

Topics

Regular expressions

A brief review of Lecture 20.

Context-free grammars

Syntax, semantics, and examples.

Regular expressions

A brief review of Lecture 20.

Sets of strings as languages

A *language* is a sets of strings with specific syntax, e.g.:

Syntactically correct Java/C/C++ programs.

The set Σ^* of all strings over the alphabet Σ.

Palindromes over Σ.

Binary strings with no 1's before 0's.

Regular expressions let us specify *regular languages*, e.g.:

All binary strings.

The strings $\{0000, 0010, 1000, 1010\}$.

All strings that contain the string "CSE311".

Regular expressions over Σ: syntax

Basis step:

$\emptyset$, ε are regular expressions.

a is a regular expression for any $a \in \Sigma$.

Recursive step:

If A and B are regular expressions, then so are

AB, $A \cup B$, and A^*.

Examples: regular expressions over $\Sigma = \{0, 1\}$

Basis: $\emptyset, \varepsilon, 0, 1$.

Recursive: $01011, 0^*1^*, (0 \cup 1)0(0 \cup 1)0$, etc.

Regular expressions over Σ: semantics

A regular expression over Σ represents a set of strings over Σ.

$\emptyset$ represents the set with no strings.

ε represents the set $\{\varepsilon\}$.

a represents the set $\{a\}$.

AB represents the concatenation of the sets represented by A and B: $\{a \bullet b \mid a \in A, b \in B\}$.

$A \cup B$ represents the union of the sets represented by A and B: $A \cup B$.

A^* represents the concatenation of the set represented by A with itself zero or more times: $A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup AAAA \cup \dots$

This just defines a recursive function definition for computing the meaning of a regular expression:

$$
\begin{aligned}
\text{language}(\emptyset) &= \{\,\} \\
\text{language}(\varepsilon) &= \{\varepsilon\} \\
\text{language}(a) &= \{a\} \text{ for all } a \in \Sigma \\
\text{language}(AB) &= \{a \bullet b \mid a \in \text{language}(A), b \in \text{language}(B)\} \\
\text{language}(A \cup B) &= \text{language}(A) \cup \text{language}(B) \\
\text{language}(A^*) &= \{\varepsilon\} \cup \text{language}(A) \cup \text{language}(AA) \cup \dots
\end{aligned}
$$

Understanding regex semantics

What is the meaning of $(0 \cup 1)^*$?

$$
\begin{aligned}
\text{language}(\emptyset) &= \{\,\} \\
\text{language}(\varepsilon) &= \{\varepsilon\} \\
\text{language}(a) &= \{a\} \text{ for all } a \in \{0, 1\} \\
\text{language}(AB) &= \{a \bullet b \mid a \in \text{language}(A), b \in \text{language}(B)\} \\
\text{language}(A \cup B) &= \text{language}(A) \cup \text{language}(B) \\
\text{language}(A^*) &= \{\varepsilon\} \cup \text{language}(A) \cup \text{language}(AA) \cup \ldots
\end{aligned}
$$

Understanding regex semantics

What is the meaning of $(0 \cup 1)^*$?

$$
\begin{aligned}
\text{language}((0 \cup 1)^*) &= \{\varepsilon\} \cup \text{language}(0 \cup 1) \cup \text{language}((0 \cup 1)(0 \cup 1)) \cup \dots \\
&= \{\varepsilon\} \cup \text{language}(0) \cup \text{language}(1) \cup \text{language}((0 \cup 1)(0 \cup 1)) \cup \dots \\
&= \{\varepsilon\} \cup \{0\} \cup \{1\} \cup \text{language}((0 \cup 1)(0 \cup 1)) \cup \dots \\
&= \{\varepsilon, 0, 1\} \cup \text{language}((0 \cup 1)(0 \cup 1)) \cup \dots \\
&= \{\varepsilon, 0, 1\} \cup \{a \bullet b \mid a \in \text{language}(0 \cup 1), b \in \text{language}(0 \cup 1)\} \cup \dots \\
&= \{\varepsilon, 0, 1\} \cup \{a \bullet b \mid a \in \text{language}(0) \cup \text{language}(1), b \in \text{language}(0) \cup \text{language}(1)\} \cup \\
&= \{\varepsilon, 0, 1\} \cup \{a \bullet b \mid a \in \{0\} \cup \{1\}, b \in \{0\} \cup \{1\}\} \cup \dots \\
&= \{\varepsilon, 0, 1\} \cup \{a \bullet b \mid a \in \{0, 1\}, b \in \{0, 1\}\} \cup \dots \\
&= \{\varepsilon, 0, 1\} \cup \{00, 01, 10, 11\} \cup \dots
\end{aligned}
$$

$$
\begin{aligned}
\text{language}(\emptyset) &= \{\} \\
\text{language}(\varepsilon) &= \{\varepsilon\} \\
\text{language}(a) &= \{a\} \text{ for all } a \in \{0, 1\} \\
\text{language}(AB) &= \{a \bullet b \mid a \in \text{language}(A), b \in \text{language}(B)\} \\
\text{language}(A \cup B) &= \text{language}(A) \cup \text{language}(B) \\
\text{language}(A^*) &= \{\varepsilon\} \cup \text{language}(A) \cup \text{language}(AA) \cup \dots
\end{aligned}
$$

Regular expressions in practice

Used to define the *tokens* in a programming language.

Legal variable names, keywords, etc.

Used in `grep`, a Unix program that searches for patterns in a set of files.

For example, `grep "311" *.md` searches for the string “311” in all Markdown files in the current directory.

Used in programs to process strings.

These slides are generated with the help of regular expressions :)

Context-free grammars

Syntax, semantics, and examples.

Regular expressions can specify only regular languages

But many languages aren't regular, including simple ones such as

- palindromes, and
- strings with an equal number of 0s and 1s.

Many programming language constructs are also irregular, such as

- expressions with matched parentheses, and
- properly formed arithmetic expressions.

Regular expressions can specify only regular languages

But many languages aren't regular, including simple ones such as

palindromes, and
strings with an equal number of 0s and 1s.

Many programming language constructs are also irregular, such as

expressions with matched parentheses, and
properly formed arithmetic expressions.

Context-free grammars are a more powerful formalism that lets us specify all of these example languages (i.e., sets of strings)!

Context-free grammars over Σ: syntax

A context-free grammar (CFG) is a finite set of *production rules* over:

- An alphabet Σ of *terminal symbols*.
- A finite set V of *nonterminal symbols*.
- A *start symbol* from V, usually denoted by $\mathbf{S}$ (i.e., $\mathbf{S} \in V$).

Context-free grammars over Σ: syntax

A context-free grammar (CFG) is a finite set of *production rules* over:

An alphabet Σ of *terminal symbols*.

A finite set V of *nonterminal symbols*.

A *start symbol* from V, usually denoted by $\mathbf{S}$ (i.e., $\mathbf{S} \in V$).

A production rule for a nonterminal $\mathbf{A} \in V$ takes the form

$\mathbf{A} \rightarrow w_1 \mid w_2 \mid \ldots \mid w_k$

where each $w_i \in (V \cup \Sigma)^*$ is a string of nonterminals and terminals.

Context-free grammars over Σ: syntax

A context-free grammar (CFG) is a finite set of *production rules* over:

An alphabet Σ of *terminal symbols*.

A finite set V of *nonterminal symbols*.

A *start symbol* from V, usually denoted by $\mathbf{S}$ (i.e., $\mathbf{S} \in V$).

A production rule for a nonterminal $\mathbf{A} \in V$ takes the form

$$\mathbf{A} \to w_1 \mid w_2 \mid \ldots \mid w_k$$

where each $w_i \in (V \cup \Sigma)^*$ is a string of nonterminals and terminals.

Only nonterminals can appear on the left-hand side of a production rule.

Context-free grammars over Σ: semantics

A CFG over Σ represents a set of strings over Σ.

Compute (or *generate*) a string from this set as follows:

1. Begin with the start symbol $\mathbf{S}$ as the current string.
2. If the current string contains a nonterminal $\mathbf{A}$, apply the rule $\mathbf{A} \to w_1 \mid \ldots \mid w_k$ to replace $\mathbf{A}$ in the current string with one of the w_i's.
3. Repeat step 2 until the current string contains only terminals.

Context-free grammars over Σ: semantics

A CFG over Σ represents a set of strings over Σ.

Compute (or *generate*) a string from this set as follows:

1. Begin with the start symbol $\mathbf{S}$ as the current string.
2. If the current string contains a nonterminal $\mathbf{A}$, apply the rule $\mathbf{A} \to w_1 \mid \ldots \mid w_k$ to replace $\mathbf{A}$ in the current string with one of the w_i's.
3. Repeat step 2 until the current string contains only terminals.

A CFG represents the set of all strings over Σ that can be generated in this way.

Example context-free grammars

$\mathbf{S} \to 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

$\mathbf{S} \to 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

$\mathbf{S} \to (\mathbf{S}) \mid \mathbf{SS} \mid \varepsilon$

CFG for $\{0^n 1^n : n \geq 0\}$, strings an equal number of 0s and 1s.

Example context-free grammars

$\mathbf{S} \to 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes.

$\mathbf{S} \to 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

$\mathbf{S} \to (\mathbf{S}) \mid \mathbf{SS} \mid \varepsilon$

CFG for $\{0^n 1^n \ : n \geq 0\}$, strings an equal number of 0s and 1s.

Example context-free grammars

$\mathbf{S} \to 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes.

$\mathbf{S} \to 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

The set of strings denoted by the regular expression 0^*1^*.

$\mathbf{S} \to (\mathbf{S}) \mid \mathbf{SS} \mid \varepsilon$

CFG for $\{0^n1^n : n \geq 0\}$, strings an equal number of 0s and 1s.

Example context-free grammars

$\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes.

$\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

The set of strings denoted by the regular expression $0^* 1^*$.

$\mathbf{S} \rightarrow (\mathbf{S}) \mid \mathbf{SS} \mid \varepsilon$

The set of all strings of matched parentheses.

CFG for $\{0^n 1^n \ : n \geq 0\}$, strings an equal number of 0s and 1s.

Example context-free grammars

$\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes.

$\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

The set of strings denoted by the regular expression 0^*1^*.

$\mathbf{S} \rightarrow (\mathbf{S}) \mid \mathbf{SS} \mid \varepsilon$

The set of all strings of matched parentheses.

CFG for $\{0^n 1^n : n \geq 0\}$, strings an equal number of 0s and 1s.

$\mathbf{S} \rightarrow 0\mathbf{S}1 \mid \varepsilon$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$\mathbf{E}$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E}$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E}$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E}$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E}$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E}$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E} \Rightarrow (2 * x) + y$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \to \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E} \Rightarrow (2 * x) + y$$

Can this CFG generate $x + y * z$ in two entirely different ways?

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E} \Rightarrow (2 * x) + y$$

Can this CFG generate $x + y * z$ in two entirely different ways?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow x + \mathbf{E} \Rightarrow x + \mathbf{E} * \mathbf{E} \Rightarrow x + y * \mathbf{E} \Rightarrow x + y * z$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E} \Rightarrow (2 * x) + y$$

Can this CFG generate $x + y * z$ in two entirely different ways?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow x + \mathbf{E} \Rightarrow x + \mathbf{E} * \mathbf{E} \Rightarrow x + y * \mathbf{E} \Rightarrow x + y * z$$

$$\mathbf{E} \Rightarrow \mathbf{E} * \mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} * \mathbf{E} \Rightarrow x + \mathbf{E} * \mathbf{E} \Rightarrow x + y * \mathbf{E} \Rightarrow x + y * z$$

Another example CFG: simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Can this CFG generate $(2 * x) + y$?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow (\mathbf{E}) + \mathbf{E} \Rightarrow (\mathbf{E} * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * \mathbf{E}) + \mathbf{E} \Rightarrow (2 * x) + \mathbf{E} \Rightarrow (2 * x) + y$$

Can this CFG generate $x + y * z$ in two entirely different ways?

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} \Rightarrow x + \mathbf{E} \Rightarrow x + \mathbf{E} * \mathbf{E} \Rightarrow x + y * \mathbf{E} \Rightarrow x + y * z$$

$$\mathbf{E} \Rightarrow \mathbf{E} * \mathbf{E} \Rightarrow \mathbf{E} + \mathbf{E} * \mathbf{E} \Rightarrow x + \mathbf{E} * \mathbf{E} \Rightarrow x + y * \mathbf{E} \Rightarrow x + y * z$$

This is perfectly valid according to the CFG rule, but it violates operator precedence for arithmetic! How can we write our grammar to enforce operator precedence?

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Building precedence in simple arithmetic expressions

$$\mathbf{E} \to \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \to \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \to (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \to x \mid y \mid z$$
$$\mathbf{N} \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

- $\mathbf{E}$ generates expressions; it's the start symbol.
- $\mathbf{T}$ generates terms.
- $\mathbf{F}$ generates factors.
- $\mathbf{I}$ generates identifiers.
- $\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$\mathbf{E}$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$
\begin{aligned}
\mathbf{E} &\to \mathbf{T} \mid \mathbf{E} + \mathbf{T} \\
\mathbf{T} &\to \mathbf{F} \mid \mathbf{F} * \mathbf{T} \\
\mathbf{F} &\to (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N} \\
\mathbf{I} &\to x \mid y \mid z \\
\mathbf{N} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \to \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \to \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \to (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \to x \mid y \mid z$$
$$\mathbf{N} \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

- $\mathbf{E}$ generates expressions; it's the start symbol.
- $\mathbf{T}$ generates terms.
- $\mathbf{F}$ generates factors.
- $\mathbf{I}$ generates identifiers.
- $\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \to \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \to \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \to (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \to x \mid y \mid z$$
$$\mathbf{N} \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

- $\mathbf{E}$ generates expressions; it's the start symbol.
- $\mathbf{T}$ generates terms.
- $\mathbf{F}$ generates factors.
- $\mathbf{I}$ generates identifiers.
- $\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{F} * \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

- $\mathbf{E}$ generates expressions; it's the start symbol.
- $\mathbf{T}$ generates terms.
- $\mathbf{F}$ generates factors.
- $\mathbf{I}$ generates identifiers.
- $\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{F} * \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{I} * \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{F} * \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{I} * \mathbf{T}$$
$$\Rightarrow \mathbf{x} + \mathbf{y} * \mathbf{T}$$

Building precedence in simple arithmetic expressions

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We use multiple production rules to encode precedence.

$\mathbf{E}$ generates expressions; it's the start symbol.
$\mathbf{T}$ generates terms.
$\mathbf{F}$ generates factors.
$\mathbf{I}$ generates identifiers.
$\mathbf{N}$ generates numbers.

Example: generating $x + y * z$

$$\mathbf{E} \Rightarrow \mathbf{E} + \mathbf{T} \Rightarrow \mathbf{T} + \mathbf{T} \Rightarrow \mathbf{F} + \mathbf{T} \Rightarrow \mathbf{I} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{F} * \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{I} * \mathbf{T}$$
$$\Rightarrow \mathbf{x} + \mathbf{y} * \mathbf{T} \Rightarrow \mathbf{x} + \mathbf{y} * \mathbf{F} \Rightarrow \mathbf{x} + \mathbf{y} * \mathbf{I} \Rightarrow \mathbf{x} + \mathbf{y} * \mathbf{z}$$

Visualizing CFG derivations with parse trees

Suppose that a grammar G generates a string x.

The sequence of steps (rule applications) that generates x is called a *derivation*.

We represent derivations as *parse trees*.

The root of the tree is the start symbol.

The internal nodes are the nonterminal symbols in the derivation.

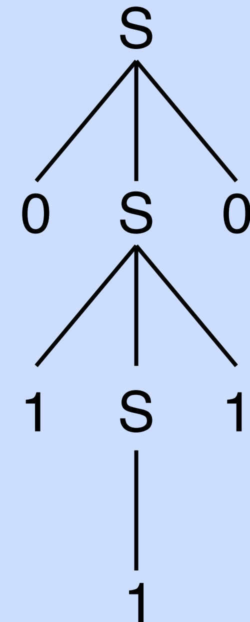
The leaves are the terminal symbols in the derivation.

Palindrome grammar

$\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

Derivation of 01110

$\mathbf{S} \Rightarrow 0\mathbf{S}0 \Rightarrow 01\mathbf{S}10 \Rightarrow 01110$



In practice, CFGs are often given in Backus-Naur Form

Backus-Naur Form (BNF) is a notation for CFGs developed for specifying the syntax of programming languages.

- Production rules use ::= instead of →.
- Nonterminals are denoted by names enclosed in angle brackets, e.g., `<identifier>`, `<digit>`, `<expression>`, etc.

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

```
<expression> ::= <term> | <expression> + <term>
<term>       ::= <factor> | <factor> * <term>
<factor>     ::= (<expression>) | <identifier> | <number>
<identifier> ::= x | y | z
<number>     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Summary

A regular expression defines a set of strings over an alphabet Σ.

- $\emptyset$, ε, and $a \in \Sigma$ are regular expressions.
- If A and B are regular expressions, then so are (AB), $(A \cup B)$, A^*.
- Many practical applications, from `grep` to everyday programming.

Context-free grammars (CFGs) are a more expressive formalism for specifying strings over an alphabet Σ.

- A CFG consists of a set of *terminal symbols*, a set of *nonterminal symbols* including the distinguished *start symbol*, and a set of *production rules* that specify how to rewrite nonterminals in a string.
- Used for specifying programming language syntax and for parsing.