Context-Free Grammars

Context-free languages are specified with a context-free grammar (CFG).

Formally, a CFG G is a 4-tuple (T, N, S, P), where:

T is the set of *terminal* symbols in the grammar.

N is the set of *nonterminals*, a set of syntactic variables that denote sets of (sub)strings occurring in the language.
These are used to impose a structure on the grammar.

S is a distinguished nonterminal ($S \in N$) denoting the entire set of strings in $L(G)$.
This is sometimes called the *start symbol*.

P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language.
Each production must have a single non-terminal on its left hand side.

The set $V = T \cup N$ is called the *vocabulary* of G

Context-Free Grammars

An example CFG:

$$
\begin{array}{lcl}
S & \rightarrow & SS \\
S & \rightarrow & (S) \\
S & \rightarrow & \epsilon
\end{array}
$$

A *derivation* for a string is a series of productions that lead from the start symbol to the string.

Example derivation:

$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (S)(()) \Rightarrow ()(())$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \rightarrow \gamma$

Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps

If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G

The language of G, $L(G) = \{w \in T | S \Rightarrow^+ w\}$

CFG Example

Another example CFG:

$$
\begin{array}{lcl}
S & \rightarrow & pSq \\
S & \rightarrow & \epsilon
\end{array}
$$

Example derivation:

$$
\begin{array}{lcl}
S & \Rightarrow & pSq \\
 & \Rightarrow & ppSqq \\
 & \Rightarrow & ppqq
\end{array}
$$

$L(G) = \{p^k q^k : k \geq 0\}$

What does this say about the relationship between CFG and RE?

CFG Example

Another example CFG:

$$
\begin{array}{lcl}
E & \rightarrow & E\ O\ E \\
E & \rightarrow & id \\
E & \rightarrow & num \\
E & \rightarrow & (E) \\
O & \rightarrow & + \\
O & \rightarrow & - \\
O & \rightarrow & * \\
O & \rightarrow & /
\end{array}
$$

This describes simple expressions over identifiers and numbers.

Derivations

We can view the productions of a CFG as rewriting rules.

Using our example CFG:

$$\begin{aligned} E &\Rightarrow E\ O\ E \\ &\Rightarrow E\ O\ E\ O\ E \\ &\Rightarrow id\ O\ E\ O\ E \\ &\Rightarrow id + E\ O\ E \\ &\Rightarrow id + num\ O\ E \\ &\Rightarrow id + num * E \\ &\Rightarrow id + num * id \end{aligned}$$

We have derived the sentence $id + num * id$.

We denote this $E \Rightarrow^* id + num * id$.

Such a sequence of rewrites is a *derivation* or a *parse*.

The process of discovering a derivation is called *parsing*.

Derivations

At each step, we chose a non-terminal to replace.

This choice can lead to different derivations.

Two are of particular interest:

leftmost derivation

the leftmost non-terminal is replaced at each step

rightmost derivation

the rightmost non-terminal is replaced at each step

The previous example was a leftmost derivation.

Rightmost Derivation

For the string $id + num * id$:

$$
\begin{aligned}
E &\Rightarrow E\ O\ E \\
&\Rightarrow E\ O\ id \\
&\Rightarrow E * id \\
&\Rightarrow E\ O\ E * id \\
&\Rightarrow E\ O\ num * id \\
&\Rightarrow E + num * id \\
&\Rightarrow id + num * id
\end{aligned}
$$

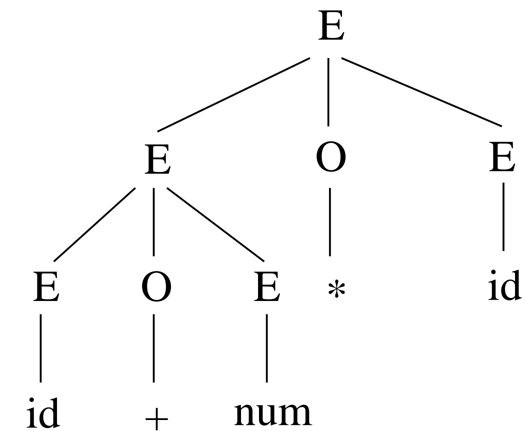
Again, $E \Rightarrow^* id + num * id$.

Parse Trees

The result of a derivation can be expressed using a *parse tree*, where:

- The *root* node is the start symbol of the grammar.
- Each *internal* node is a non-terminal with the symbols from one of its production rules (from left to right) as its children.
- The *leaves* of the tree are terminal symbols, which when read from left to right give the string which has been derived.

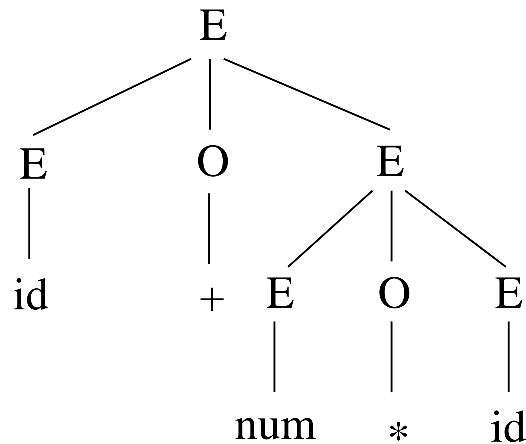
For example, the following parse tree can be derived for the string $id + num * id$:



Ambiguity

A grammar is *ambiguous* if its language contains some string that has two different parse trees (this means it has two distinct leftmost or rightmost derivations).

For example, the following parse tree can also be derived for the string $id + num * id$:

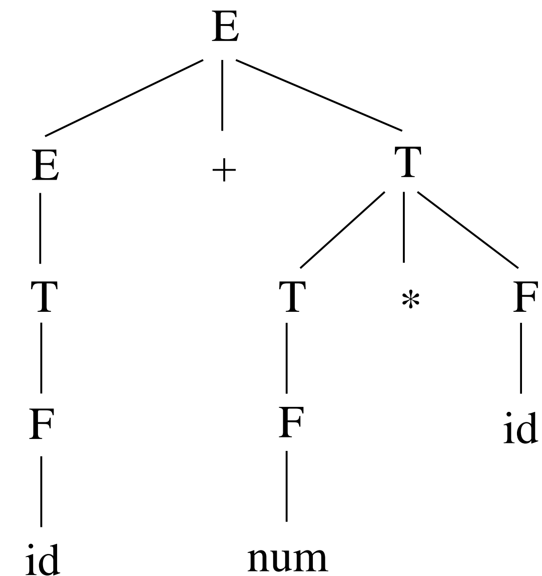


Ambiguity

The previous ambiguity can be avoided with the following grammar:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T/F \mid F \\ F & \rightarrow & id \mid num \mid (E) \end{array}$$

The only parse tree which can be derived for the string $id + num * id$ is as follows:



Extending Finite State Automata

Not every context-free language can be recognized by a FSA.

Can we extend the FSA concept to create FSA-like machines that recognize a language iff it is a context-free language?

Consider $L = \{wcw^r | w \in \Sigma^*\}$

An automaton that recognizes this must "remember" w until it sees c and then checks for w^r (w reversed).

This can be done with a stack memory.

Checking for matching parentheses can also be done with a stack memory.

Pushdown Automata

Consider a NFA with stack memory.

This is called a *pushdown automaton* (PDA).

For any CFG rule $A \rightarrow wBx$, a pushdown automaton can imitate it by reading w, putting x on the stack and going to state B.

The PDA can decide (non-deterministically) to pop values off the stack and compare against the next input characters.

Theorem: A language is context-free iff it can be recognized by a pushdown automaton.

PDA Definition

A pushdown automaton is a 6-tuple $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$, where:

- Q is a finite set of *states*.
- Σ is an alphabet (*input symbols*).
- Γ is an alphabet (*stack symbols*).
- $q_0 \in Q$ is the *initial state*.
- $F \subseteq Q$ is the set of *final states*.
- δ, the *transition function*, is from $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ to $Q \times (\Gamma \cup \{\epsilon\})$.

If $((q, a, \alpha), (q', \beta)) \in \delta$, then when in state q with α at the top of the stack, M may read a from input, replace α with β on the top of the stack, and enter state q'.

M accepts $w \in \Sigma^*$ iff $(q, w, \epsilon) \vdash_M^* (q', \epsilon, \epsilon)$ for $q' \in F$.

Push: $((q, a, \epsilon), (q', \beta))$ pushes β onto the stack.

Pop: $((q, a, \alpha), (q', \epsilon))$ pops α from the stack.

PDA Example

$L = \{wcw^r : w \in \{a, b\}^*\}$
(palindromes of odd length)

Let $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$

- $Q = \{s, f\}$
- $\Sigma = \{a, b, c\}$
- $\Gamma = \{a, b\}$
- $q_0 = s$
- $F = \{f\}$
- $\delta = \begin{array}{l} ((s, a, \epsilon), (s, a)) \\ ((s, b, \epsilon), (s, b)) \\ ((s, c, \epsilon), (f, \epsilon)) \\ ((f, a, a), (f, \epsilon)) \\ ((f, b, b), (f, \epsilon)) \end{array}$

Example: $abbcbba$

PDA Example

$L = \{ww^r : w \in \{a,b\}^*\}$
(palindromes of even length)

Let $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$

- $Q = \{s, f\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, b\}$
- $q_0 = s$
- $F = \{f\}$
- $\delta = \begin{array}{l} ((s,a,\epsilon),(s,a)) \\ ((s,b,\epsilon),(s,b)) \\ ((s,\epsilon,\epsilon),(f,\epsilon)) \\ ((f,a,a),(f,\epsilon)) \\ ((f,b,b),(f,\epsilon)) \end{array}$

Just like previous example, except for Transition 3.

Example: $abbbba$

PDA Example

$L = \{w \in \{a,b\}^*$: w has an equal number of a's and b's$\}$

Let $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$

- $Q = \{s, q, f\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, b, c\}$
- $q_0 = s$
- $F = \{f\}$
- $\delta = \begin{array}{l} ((s,\epsilon,\epsilon),(q,c)) \\ ((q,a,\epsilon),(q,a)) \\ ((q,a,b),(q,\epsilon)) \\ ((q,b,\epsilon),(q,b)) \\ ((q,b,a),(q,\epsilon)) \\ ((q,\epsilon,c),(f,\epsilon)) \end{array}$

c is a special symbol to mark the bottom of the stack.

Keep either the excess a's or excess b's in the stack.

Deterministic PDA

A PDA is deterministic if we can always decide which transition to use next.

$\{wcw^r : w \in \{a,b\}^*\}$ is deterministic – once a 'c' is read, start matching w^r against the stack.

$\{ww^r : w \in \{a,b\}^*\}$ is not deterministic:

$$\delta = \begin{matrix} ((s,a,\epsilon),(s,a)) \\ ((s,b,\epsilon),(s,b)) \\ ((s,\epsilon,\epsilon),(f,\epsilon)) \\ ((f,a,a),(f,\epsilon)) \\ ((f,b,b),(f,\epsilon)) \end{matrix}$$

Rule 3 is confused with rules 1 or 2.

A language L is *deterministic context-free* if there is some deterministic PDA M that recognizes L.

Theorem: Not every non-deterministic PDA can be converted to an equivalent deterministic PDA.

This has serious implications for the efficient parsing of context-free languages.

Pumping Lemma

The pumping lemma for context-free languages can be stated as follows:

Let L be an infinite context-free language. Then there are strings v, w, x, y and z such that $wy \neq \epsilon$ and $vw^kxy^kz \in L$ for each $k \geq 0$

For example, the language $L = \{a^nb^nc^n\}$ cannot be rewritten as vw^kxy^kz, so it cannot be context-free.

Proof:

The pumped variables w and y cannot contain two distinct letters.

At least one of w and y is non-empty.

Therefore, not all letters can get pumped up equally.