Context Free Grammar

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Grammar

Grammar is a certain kind of collection of rules for building strings. Like DFAs, NFAs, and regular expressions, grammars are mechanisms for defining languages rigorously.

Toy English Grammar

- An article A can be the word *a* or *the*.

$$A \to a$$
$$A \to the$$

- A noun N can be the word *dog*, *cat* or *rat*.

$$N \to dog \mid cat \mid rat$$

- A noun phrase P is an article followed by a noun.

$$P \to AN$$

- A verb V can be the word *love*, *hate* or *eat*.

$$V \to love \mid hate \mid eat$$

- A sentence S can be a noun phrase, followed by a verb, followed by another noun phrase

$$S \to PVP$$

Terminology

The following defines a grammar $\mathcal{G}$

$$
\begin{aligned}
S &\to PVP \\
P &\to AN \\
V &\to love \mid hate \mid eat \\
A &\to a \mid the \\
N &\to dog \mid cat \mid rat
\end{aligned}
$$

- Each rule is called a *production* or *production rule*.
- $X \to Y$ means X can be substitute by Y. $X \to Y_1 \mid Y_2$ is a compact form of

$$
\begin{aligned}
X &\to Y_1 \\
X &\to Y_2
\end{aligned}
$$

- S, P, V, A, N are *terminals*, *love*, *hate*, *eat* and etc are *non-terminals*.

Generator

The grammar is a language *generator*. Finite automaton is a language *recognizer*.

$$
\begin{aligned}
S &\rightarrow PVP \\
P &\rightarrow AN \\
V &\rightarrow love \mid hate \mid eat \\
A &\rightarrow a \mid the \\
N &\rightarrow dog \mid cat \mid rat
\end{aligned}
$$

S is the *start symbol*.

$$
\begin{aligned}
S &\Rightarrow PVP \\
&\Rightarrow ANVP \\
&\Rightarrow theNVP \\
&\Rightarrow thecatV \\
&\Rightarrow thecateatP \\
&\Rightarrow thecateatAN \\
&\Rightarrow thecateataN \\
&\Rightarrow thecateatarat
\end{aligned}
$$

Derivation

$$
\begin{aligned}
S &\Rightarrow P\,VP \\
&\Rightarrow A N VP \\
&\Rightarrow the N VP \\
&\Rightarrow thecat V \\
&\Rightarrow thecateat P \\
&\Rightarrow thecateat A N \\
&\Rightarrow thecateata N \\
&\Rightarrow thecateatarat
\end{aligned}
$$

- More one place to apply production rule
- Leftmost derivation is not required, e.g.,
  - $P love P \Rightarrow A N love P$
  - $P love P \Rightarrow P love A N$

Informal Definition

Grammar is a set of productions of the form $x \to y$.

- x and y may both contain lowercase and uppercase letters.
- x cannot be ϵ, but y can be ϵ.
- One uppercase letter is designated as the *start symbol*, usually S by convention.

Given a grammar $\mathcal{G}$, the language it defines is

$$\mathcal{L}(\mathcal{G}) = \{x \mid x \text{ is generated by } \mathcal{G}\}$$

When a sequence of permissible substitutions starting from S ends in a string of all lowercase, we say *the grammar generates that string*.

Example Grammar

Given a grammar $\mathcal{G}$

$$
\begin{aligned}
S &\to aS \mid X \\
X &\to bX \mid \epsilon
\end{aligned}
$$

The language defined by $\mathcal{G}$ is

Example Grammar

Given a grammar $\mathcal{G}$

$$S \to aS \mid X$$
$$X \to bX \mid \epsilon$$

The language defined by $\mathcal{G}$ is

$$\mathcal{L}(\mathcal{G}) = \mathcal{L}(a^* b^*)$$

Outline

Formal Definition

A grammar $\mathcal{G}$ is a 4-tuple

$$\mathcal{G} = (V, \Sigma, S, P)$$

- V is an alphabet, the *non-terminal* alphabet
- Σ is another alphabet, the *terminal* alphabet
- $V \bigcap \Sigma = \emptyset$
- $S \in V$ is the start symbol
- P is a *finite* set of productions, each of the form $\alpha \to \beta$,
  - α is $(V \bigcup \Sigma)^* V (V \bigcup \Sigma)^*$, i.e., a string of terminals and non-terminals containing at *least one* non-terminal
  - β is $(V \bigcup \Sigma)^*$, i.e., a string of terminals and non-terminals.

Example Grammar

$$S \rightarrow aS \mid X$$
$$X \rightarrow bX \mid \epsilon$$

This grammar $\mathcal{G} = (V, \Sigma, S, P)$ is defined as:

- $V = \{S, X\}$
- $\Sigma = \{a, b\}$
- $P = \{S \rightarrow aS, S \rightarrow X, X \rightarrow bX, X \rightarrow \epsilon\}$

Language Generated by Grammar

- $\Rightarrow$ w derives z, denoted as $w \Rightarrow z$, if and only if $\exists u, x, y, v \in \Sigma \bigcup V$ such that $w = uxv$, $z = uyv$ and $(x \rightarrow y) \in P$
- $\Rightarrow^*$ $w \Rightarrow^* z$ if and only if there is a derivation of 0 or more steps that starts with w and ends with z. Specifically, α is called a *sentential form* iff $S \Rightarrow^* \alpha$.

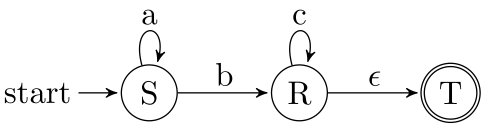
The language generated by a grammar $\mathcal{G}$ is

$$\mathcal{L}(\mathcal{G}) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$$

NFA to Grammar

Theorem

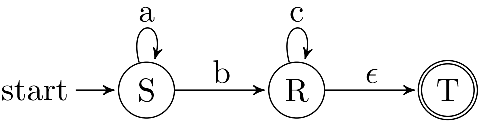
There is a grammar for every regular language.



The above NFA $\mathcal{M}$ describes

NFA to Grammar

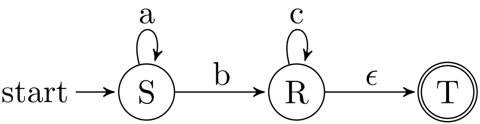
Theorem
There is a grammar for every regular language.



The above NFA $\mathcal{M}$ describes `a*bc*`

- For each state, our grammar will have a non-terminal symbol (S, R and T).
- The start state S will be the grammar's start symbol.
- The grammar will have one production for each transition of the NFA, and one for each accepting state

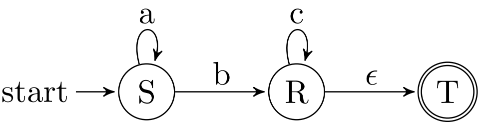
NFA to Grammar I



For each possible transition $Y \in \delta(X, z)$ in the NFA, we have a production in our grammar $X \to zY$.

Transition of $\mathcal{M}$	Production in $\mathcal{G}$
$\delta(S, a) = \{S\}$	$S \to aS$
$\delta(S, b) = \{R\}$	$S \to bR$
$\delta(R, c) = \{R\}$	$R \to cR$
$\delta(R, \epsilon) = \{T\}$	$R \to T$

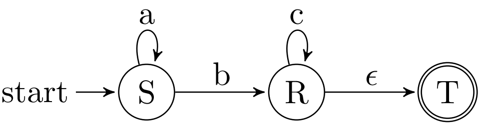
NFA to Grammar II



For each accepting state in the NFA, we have an ϵ-production in our grammar.

Accepting state of $\mathcal{M}$	Production in $\mathcal{G}$
T	$T \to \epsilon$

NFA to Grammar III



$$S \to aS \mid bR$$
$$R \to cR \mid T$$
$$T \to \epsilon$$

Given a string *abc*

$$\begin{array}{lllllll} \mathcal{M}: & (S, abc) & \mapsto (S, bc) & \mapsto (R, c) & \mapsto (R, \epsilon) & \mapsto (T, \epsilon) & \\ \mathcal{G}: & S & \Rightarrow aS & \Rightarrow abR & \Rightarrow abcR & \Rightarrow abcT & \Rightarrow abc \end{array}$$

Outline

Context Free

We only discuss grammar with production like

$$S \to Sa$$

Only a *single non-terminal* symbol appear at left-hand side of each production, which is called context-free grammar (CFG).
Grammars with production like $aS \to Sa$ is context-sensitive.

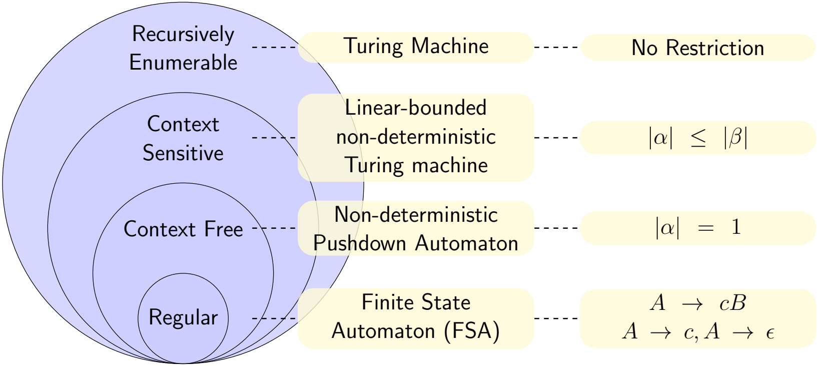
Formal Definition

A grammar $\mathcal{G}$ is a 4-tuple

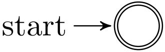
$$\mathcal{G} = (V, \Sigma, S, P)$$

- V is an alphabet, the *non-terminal* alphabet
- Σ is another alphabet, the *terminal* alphabet
- $V \bigcap \Sigma = \emptyset$
- $S \in V$ is the start symbol
- P is a *finite* set of productions, each of the form $\alpha \rightarrow \beta$,
  - α is V, i.e., a single non-terminal
  - β is $(V \bigcup \Sigma)^*$, i.e., a string of terminals and non-terminals.

Chomsky Hierarchy

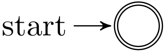


Revisit $\{a^n b^n\}$ with NFA

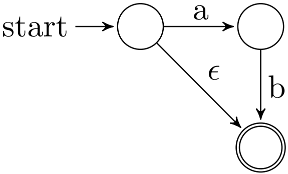
- $n \leq 0$ 

Revisit $\{a^n b^n\}$ with NFA

- $n \leq 0$

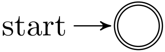


- $n \leq 1$

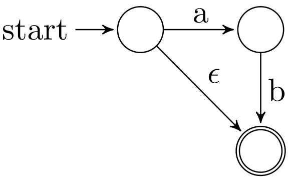


Revisit $\{a^n b^n\}$ with NFA

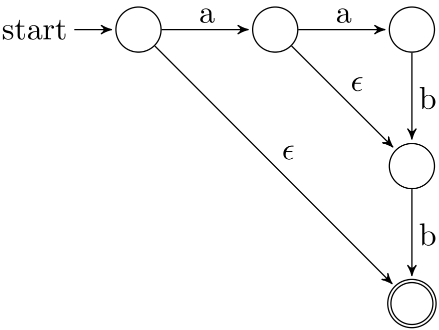
- $n \leq 0$



- $n \leq 1$



- $n \leq 2$



Revisit $\{a^n b^n\}$ with CFG

$$S \rightarrow aSb \mid \epsilon$$

- $S \Rightarrow ab$
- $S \Rightarrow aSb \Rightarrow aabb$
- $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$
- etc.

Regular Language to CFG

If the language is regular, conversion to CFG is easy.

- construct an NFA, and
- convert to right-linear grammar (regular grammar)

Regular Language to CFG Example 0

$$S \to aR$$
$$R \to bR \mid aT$$
$$T \to \epsilon$$

This grammar describes

Regular Language to CFG Example 0

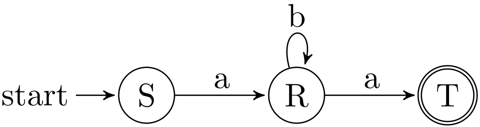
$$S \rightarrow aR$$
$$R \rightarrow bR \mid aT$$
$$T \rightarrow \epsilon$$

This grammar describes `ab*a`.
A corresponding NFA accepts this language.

Regular Language to CFG Example 0

$$S \to aR$$
$$R \to bR \mid aT$$
$$T \to \epsilon$$

This grammar describes `ab*a`.
A corresponding NFA accepts this language.

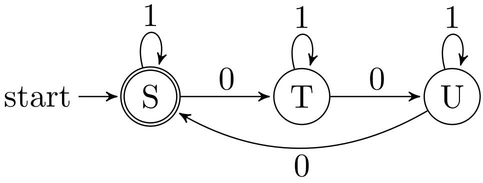


Regular Language to CFG Example 1

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \text{the number of 0's is divisible by 3}\}$$

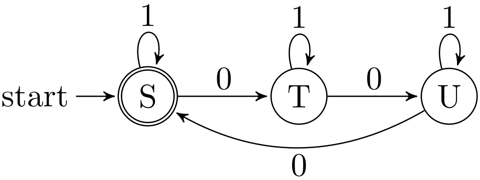
Regular Language to CFG Example 1

$$\mathcal{L} = \{x \in \{0,1\}^* \mid \text{the number of 0's is divisible by 3}\}$$



Regular Language to CFG Example 1

$\mathcal{L} = \{x \in \{0,1\}^* \mid \text{the number of 0's is divisible by 3}\}$



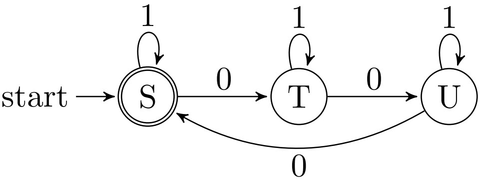
$S \rightarrow 1S \mid 0T \mid \epsilon$

$T \rightarrow 1T \mid 0U$

$U \rightarrow 1U \mid 0S$

Regular Language to CFG Example 1

$\mathcal{L} = \{x \in \{0,1\}^* \mid \text{the number of 0's is divisible by 3}\}$



$$S \to 1S \mid 0T \mid \epsilon$$
$$T \to 1T \mid 0U$$
$$U \to 1U \mid 0S$$

$$S \to T0T0T0S \mid T$$
$$T \to 1T \mid \epsilon$$

Balanced Pairs to CFG

CFLs often involves balanced pairs.

- $\{a^n b^n\}$ every a paired with b
- $\{xx^R \mid x \in \{a, b\}^*\}$ each x paired with its mirror image x^R
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$ same number of a's on each side

To get matching pairs, following recursive production is often used

$$R \to xRy$$

- $\{a^n b^n\}$
- $\{xx^R \mid x \in \{a, b\}^*\}$
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$
- $\{a^n b^{3n}\}$

- $S \to aSb \mid \epsilon$

Balanced Pairs to CFG

CFLs often involves balanced pairs.

- $\{a^n b^n\}$ every a paired with b
- $\{xx^R \mid x \in \{a, b\}^*\}$ each x paired with its mirror image x^R
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$ same number of a's on each side

To get matching pairs, following recursive production is often used

$$R \rightarrow xRy$$

- $\{a^n b^n\}$
- $\{xx^R \mid x \in \{a, b\}^*\}$
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$
- $\{a^n b^{3n}\}$

- $S \rightarrow aSb \mid \epsilon$
- $S \rightarrow aSa \mid bSb \mid \epsilon$

Balanced Pairs to CFG

CFLs often involves balanced pairs.

- $\{a^n b^n\}$ every a paired with b
- $\{xx^R \mid x \in \{a, b\}^*\}$ each x paired with its mirror image x^R
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$ same number of a's on each side

To get matching pairs, following recursive production is often used

$$R \to xRy$$

- $\{a^n b^n\}$
- $\{xx^R \mid x \in \{a, b\}^*\}$
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$
- $\{a^n b^{3n}\}$

- $S \to aSb \mid \epsilon$
- $S \to aSa \mid bSb \mid \epsilon$
- $S \to aSa \mid R,\ R \to bR \mid \epsilon$

Balanced Pairs to CFG

CFLs often involves balanced pairs.

- $\{a^n b^n\}$ every a paired with b
- $\{xx^R \mid x \in \{a,b\}^*\}$ each x paired with its mirror image x^R
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$ same number of a's on each side

To get matching pairs, following recursive production is often used

$$R \to xRy$$

- $\{a^n b^n\}$
- $\{xx^R \mid x \in \{a,b\}^*\}$
- $\{a^n b^i a^n \mid n \geq 0, i \geq 1\}$
- $\{a^n b^{3n}\}$

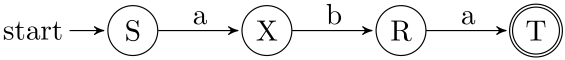
- $S \to aSb \mid \epsilon$
- $S \to aSa \mid bSb \mid \epsilon$
- $S \to aSa \mid R,\ R \to bR \mid \epsilon$
- $S \to aSbbb \mid \epsilon$

Non-Regular Grammar to NFA

$$S \to abR$$
$$R \to a$$

- $S \to abR$ is equivalent to $S \to aX, X \to bR$
- $R \to a$ is equivalent to $R \to aT, T \to \epsilon$

$$S \to aX$$
$$X \to bR$$
$$R \to aT$$
$$T \to \epsilon$$



Grammar Concatenation

$$L = \{a^n b^n c^m d^m\}$$

- It is easy to write grammars separately
  - $L_1 = \{a^n b^n\}$: $S_1 \rightarrow aS_1 b \mid \epsilon$
  - $L_2 = \{c^m d^m\}$: $S_2 \rightarrow cS_2 d \mid \epsilon$
- Since $L = L_1 L_2$, $S \rightarrow S_1 S_2$

Grammar Union

$$L = \{z \in \{a, b\}^* \mid z = xx^R \text{ or } |z| \text{ is odd}\}$$

- $L_1 = \{xx^R \mid x \in \{a, b\}^*\}$

$$S_1 \to aS_1a \mid bS_2b \mid \epsilon$$

- $L_2 = \{z \in \{a, b\}^* \mid |z| \text{ is odd}\}$

$$S_2 \to XXS_2 \mid X$$
$$X \to a \mid b$$

- Since $L = L_1 \bigcup L_2$, $S \to S_1 \mid S_2$

Outline

Introduction

Backus-Naur Form (BNF) was developed by John Backus and Peter Naur, independently from CFG.

- Every symbol is enclosed by <>
- $\rightarrow$ replace by `::=`.

```
<syntax>     ::= <rule> | <rule> <syntax>
<rule>       ::= <space> "<" <TEXT> ">" <space>
                 "::=" <space> <expression> <line-end>
<space>      ::= " " <space> | ""
<expression> ::= <list> | <list> "|" <expression>
<line-end>   ::= <space> <EOL> | <line-end> <line-end>
<list>       ::= <term> | <term> <space> <list>
<term>       ::= <literal> | "<" <rule-name> ">"
<literal>    ::= "\"" <TEXT> "\"" | "'" <TEXT> "'"
```

BNF Notation – Kleene Closure

The symbol $\{\}$ is used for "zero or more".

```
<unsigned> ::= <nonzero> { <digit> }
<digit>    ::= 0 | <nonzero>
<nonzero>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The $\{\alpha\}$ is equivalent to $A \rightarrow A\alpha \mid \alpha$.

```
<unsigned> ::= <nonzero> <more>
<more>     ::= <more> <digit> | <digit>
<digit>    ::= 0 | <nonzero>
<nonzero>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

BNF Notation – Optional

The symbol [ ] is used for optional items.

```
<stmt> ::= if <cond> then <stmt> [else <stmt>]
```

The $[\alpha]$ is equivalent to $A \to \alpha \mid \epsilon$.

```
<stmt> ::= if <cond> then <stmt> <else>
<else> ::= else <stmt> | ""
```

BNF Notation – Miscellaneous

Usage	Notation
termination	;
alternation	|
optional	[ ]
repetition	{ }
grouping	( )
Terminal string	" " or ' '

See C99 YACC/Bison grammar
http://www.quut.com/c/ANSI-C-grammar-y-2011.html

BNF Example I

```
<exp> ::= <exp> - <exp>
        | <exp> * <exp>
        | <exp> = <exp>
        | <exp> < <exp>
        | (<exp>)
        | a | b | c
```

Some strings generated by this grammar

- `a < b`
- `(a - (b * c))`
- `a = c`
- `a * b = c`

BNF Example II

```
<binary> ::= <digit> | <binary> <digit>
<digit>  ::= 0 | 1
```

This grammar generates strings with 0 and 1.

Outline

Left/Right-most Derivation

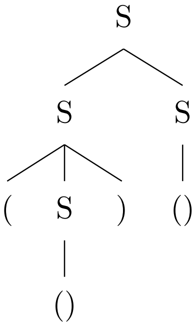
$$S \rightarrow SS \mid (S) \mid ()$$

This is balanced-parentheses grammar.
Given a string `(())()`, we have multiple ways to generate it.

- Leftmost derivation $S \Rightarrow SS \Rightarrow (S)S \Rightarrow (())S \Rightarrow (())()$
- Rightmost derivation $S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (())()$

Parse Tree

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



Parse trees are trees labeled by symbols of a particular grammar.

Leaves are labeled by a terminal or ϵ.

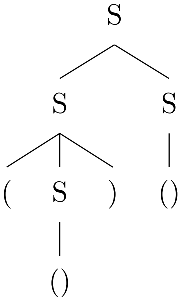
Interior nodes are labeled by a variable (i.e., non-terminals), the children of which are labeled by the right side of the production or the parent.

Root is labeled by the start symbol.

Yield of Parse Tree

The concatenation of the labels of the leaves in left-to-right order, i.e., pre-order traversal, is the *yield* of the parse tree.

The yield of



is `(())()`.

Ambiguous Grammar

A grammar is *ambiguous* if there is a string in the language that is the yield of more than one distinct parse trees.

Theorem
For every parse tree, there is a unique leftmost, and a unique rightmost derivation.

A grammar is *ambiguous* if

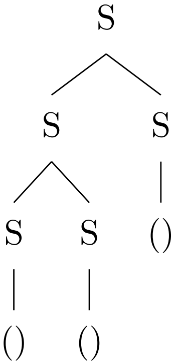
- the grammar generates a sentence with more than one leftmost derivation or
- the grammar generates a sentence with more than one rightmost derivation.

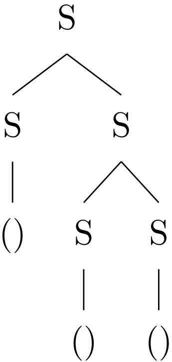
Ambiguous Grammar Example

Two ways to generate `()()()` from grammar $S \rightarrow SS \mid (S) \mid ()$.

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

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





Why Ambiguous?

Ambiguity is a *property* of grammars, NOT languages.

- $S \to SS \mid (S) \mid ()$ is ambiguous.
- $\begin{array}{l} S \to (RS \mid \epsilon \\ R \to) \mid (RR \end{array}$ R generates strings that have one more right ')'

Ambiguity Good or Bad?

- Bad. It leaves meaning of some programs ill-defined since we cannot decide its syntactical structure uniquely.
- Good. Ambiguous grammars are often used in LR parsing because of simplicity.

$$
\begin{aligned}
E \rightarrow & E + E \\
& | E * E \\
& | (E) \mid \epsilon
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
$$

Solution to Ambiguity

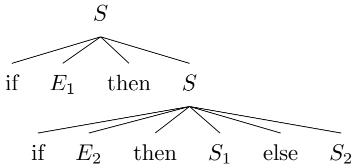
1. Disambiguate the grammar, i.e., rewrite the grammar.
2. In practice use disambiguating declarations.

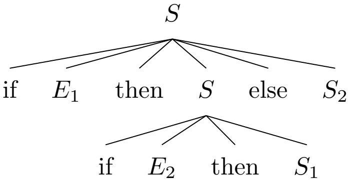
Disambiguate the Grammar 0

$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S \\
\mid\ & \text{if } E \text{ then } S \text{ else } S
\end{aligned}
$$

This is *the dangling else* problem.

`if` E_1 `then` `if` E_2 `then` S_1 else S_2





Disambiguate the Grammar 1

Disambiguating Rule: Match each `else` with the closest unmatched `then`. To incorporate the rule, we need to distinguish between matched M and unmatched U.

$$
\begin{aligned}
S &\to M \mid U \\
M &\to \text{ if } E \text{ then } M \text{ else } M \\
U &\to \text{ if } E \text{ then } S \\
&\mid \text{ if } E \text{ then } M \text{ else } U
\end{aligned}
$$

Disambiguating Declarations

In practice, instead of rewriting the grammar,

1. use the more natural (ambiguous) grammar
2. along with *disambiguating declarations*

Most tools (e.g., YACC) allow *precedence* and *associativity* declaration for terminals

Associativity

Productions that are left (right)-recursive force evaluation in left-to-right (right-to-left) order, i.e., left (right) associativity.

$$
\begin{aligned}
S &\rightarrow E \\
&\quad \mid S + E \\
&\quad \mid S * E \\
E &\rightarrow a \mid b \mid c \mid S
\end{aligned}
$$

Parse $a + b * c$ and $a * b + c$.

Precedence

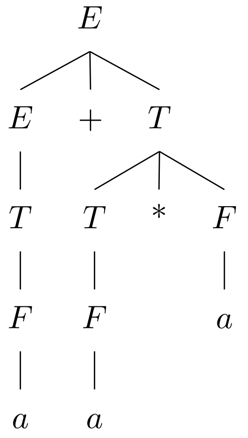
Precedence is introduced by adding new non-terminal symbols.

- Symbols with lowest precedence closest to the start symbol.
- Lower symbol in the parse tree has higher precedence.

Given following grammar

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

Consider the string $a + a * a$.



Inherent Ambiguity

It would be nice if for every ambiguous grammar, there were some way to fix the ambiguity. However, some CFL's are *inherently ambiguous*.

$$\{0^i 1^j 2^k \mid i = j \text{ or } j = k\}$$

One possible grammar

$$S \to AB \mid CD$$
$$A \to 0A1 \mid 01$$
$$B \to 2B \mid 2$$
$$C \to 0C \mid 0$$
$$D \to 1D2 \mid 12$$

Consider the string 012

- $S \Rightarrow AB \Rightarrow 01B \Rightarrow 012$
- $S \Rightarrow CD \Rightarrow 0D \Rightarrow 012$

Ambiguity Summary

- Ambiguity is good since it simplifies the grammar
- Ambiguity is bad since it is ambiguous

As for ambiguity solution

- We have no general techniques to handle ambiguity
- Impossible to remove ambiguity automatically

LL/LR Parser

LL L-eft-to-right scan, L-eftmost derivation

LR L-eft-to-right scan, R-ightmost derivation

- As for more in depth review, see difference between LL and LR
- As for the comparison, pros and cons of LL and LR

Outline

Limitation?

$$L = \{a^n b^n\}$$

$$S \to aSb \mid \epsilon$$

$$L = \{a^n b^n c^n\}$$

$$S \to \dots$$