Theory of Automata
Context Free Grammars

Week 8

Contents

- Simplification of CFGs
  - Killing Λ-Productions
  - Killing unit-productions
  - Removing Useless Variables
    - Symbols & Productions
  - Augmented Grammar
  - Removal of Left Recursion
  - Expression Grammar
  - Left Factoring

Killing Λ-Productions

Λ-Productions:

In a given CFG, we call a non-terminal N ***nullable***

- if there is a production N → Λ, or
- there is a derivation that starts at N and lead to a Λ.

  N ➔ ……… ➔ Λ

- Λ-Productions are undesirable.
- We can replace Λ-production with appropriate non-Λ productions.

Theorem 23

If L is CFL generated by a CFG having Λ-productions, then there is a different CFG that has no Λ-production and still generates either the whole language L (if L does not include Λ) or else generate the language of all the words in L other than Λ.

Replacement Rule.

1. Delete all Λ-Productions.
2. Add the following productions:

   For every production of the X → old string

   Add new production(s) of the form X → .., where right side will account for **every modification** of the old string that can be formed by **deleting all possible subsets** of null-able Non-Terminals, except that we do not allow X → Λ, to be formed if all the character in old string are null-able

Example

Consider the CFG

$S \rightarrow a \mid Xb \mid aYa$

$X \rightarrow Y \mid \Lambda$

$Y \rightarrow b \mid X$

X is nullable

Y is nullable

Old nullable Production	New Production
$X \rightarrow Y$	nothing
$X \rightarrow \Lambda$	nothing
$Y \rightarrow X$	nothing
$S \rightarrow Xb$	$S \rightarrow b$
$S \rightarrow aYa$	$S \rightarrow aa$

So the new CFG is

$S \rightarrow a \mid Xb \mid aa \mid aYa \mid b$

$X \rightarrow Y$

$Y \rightarrow b \mid X$

Example

Consider the CFG

$S \rightarrow Xa$

$X \rightarrow aX \mid bX \mid \Lambda$

X is nullable

Old nullable Production	New Production
$S \rightarrow Xa$	$S \rightarrow a$
$X \rightarrow aX$	$X \rightarrow a$
$X \rightarrow bX$	$X \rightarrow b$

So the new CFG is

$S \rightarrow a \mid Xa$

$X \rightarrow aX \mid bX \mid a \mid b$

Example

S → XY

X → Zb

Y → bW

Z → AB

W → Z

A → aA | bA | Λ

B → Ba | Bb | Λ

- Null-able Non-terminals are?
- A, B, Z and W

Example Contd.

S → XY
X → Zb
Y → bW
Z → AB
W → Z
A → aA | bA | Λ
B → Ba | Bb | Λ

Old nullable Production	New Production
X → Zb	X → b
Y → bW	Y → b
Z → AB	Z → A and Z → B
W → Z	Nothing new
A → aA	A → a
A → bA	A → b
B → Ba	B →a
B → Bb	B → b

So the new CFG is

S → XY
X → Zb | b
Y → bW | b
Z → AB | A | B
W → Z
A → aA | bA | a | b
B → Ba | Ba | a | b

Remove Nulls

$(a + b)^*bb(a + b)^*$

$S \rightarrow XY$
$X \rightarrow Zb$
$Y \rightarrow bW$
$Z \rightarrow AB$
$W \rightarrow Z$
$A \rightarrow aA \mid bA \mid \Lambda$
$B \rightarrow Ba \mid Bb \mid \Lambda$

Old	**Additional New Productions Derived from Old**
$X \rightarrow Zb$	$X \rightarrow b$
$Y \rightarrow bW$	$Y \rightarrow b$
$Z \rightarrow AB$	$Z \rightarrow A$ and $Z \rightarrow B$
$W \rightarrow Z$	Nothing
$A \rightarrow aA$	$A \rightarrow a$
$A \rightarrow bA$	$A \rightarrow b$
$B \rightarrow Ba$	$B \rightarrow a$
$B \rightarrow Bb$	$B \rightarrow b$

$$
\begin{aligned}
S &\rightarrow XY \\
X &\rightarrow Zb \mid b \\
Y &\rightarrow bW \mid b \\
Z &\rightarrow AB \mid A \mid B \\
W &\rightarrow Z \\
A &\rightarrow aA \mid bA \mid a \mid b \\
B &\rightarrow Ba \mid Bb \mid a \mid b
\end{aligned}
$$

Killing unit-productions

- **Definition:** A production of the form
  - non-terminal → one non-terminal

is called a **unit production**.

- The following theorem allows us to get rid of unit productions:

Theorem 24:

If there is a CFG for the language L that has no Λ-productions, then there is also a CFG for L with no Λ-productions and **no unit productions**.

Proof of Theorem 24

- This is another proof by constructive algorithm.
- **Algorithm:** For every pair of non-terminals A and B, if the CFG has a unit production A → B, or if there is a chain

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow B$$

where X_1, X_2, ... are non-terminals, create new productions as follows:

- If the non-unit productions from B are

$$B \rightarrow s_1 \mid s_2 \mid \ldots$$

where s_1, s_2, ... are strings, we create the productions

$$A \rightarrow s_1 \mid s_2 \mid \ldots$$

Example

- Consider the CFG

  $S \rightarrow A \mid bb$

  $A \rightarrow B \mid b$

  $B \rightarrow S \mid a$
- The non-unit productions are

  $S \rightarrow bb$ $\quad A \rightarrow b$ $\quad B \rightarrow a$
- And unit productions are

  $S \rightarrow A$

  $A \rightarrow B$

  $B \rightarrow S$

Example contd.

- Let's list all unit productions and their sequences and create new productions:

$S \rightarrow A$	gives	$S \rightarrow b$
$S \rightarrow A \rightarrow B$	gives	$S \rightarrow a$
$A \rightarrow B$	gives	$A \rightarrow a$
$A \rightarrow B \rightarrow S$	gives	$A \rightarrow bb$
$B \rightarrow S$	gives	$B \rightarrow bb$
$B \rightarrow S \rightarrow A$	gives	$B \rightarrow b$

- Eliminating all unit productions, the new CFG is

  $S \rightarrow bb \mid b \mid a$

  $A \rightarrow b \mid a \mid bb$

  $B \rightarrow a \mid bb \mid b$
- This CFG generates a finite language since there are no non-terminals in any strings produced from S.

Useless Symbols

- A symbol that is not useful is useless
- Let a CFG G. A symbol X ε (V U ∑) is useful if there is a derivation

$$S \underset{G}{\overset{*}{\Rightarrow}} UxV \underset{G}{\overset{*}{\Rightarrow}} w$$

Where U and V ε (V U ∑) and w ε ∑*.

- A terminal is useful if it occurs in a string of the language of G.
- A variable is useful if it occurs in a derivation that begins from S and generates a terminal string

For a variable to be useful two conditions must be satisfied.

1. The variable must occur in a sentential form of the grammar
2. There must be a derivation of a terminal string from the variable.

- A variable that occurs in a sentential form is said to be reachable from S.
- A two part procedure is presented to eliminate useless symbols.

Useless Productions

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

$$S \rightarrow A$$

$$A \rightarrow aA$$ Useless Production

Some derivations never terminate...

Another grammar:

$$S \to A$$

$$A \to aA$$

$$A \to \lambda$$

$$B \to bA$$ Useless Production

Not reachable from S

In general:

if $S \Rightarrow \ldots \Rightarrow xAy \Rightarrow \ldots \Rightarrow w$

and w contains only terminals

$w \in L(G)$

then variable A is useful

otherwise, variable A is useless

A production $A \rightarrow x$ is useless
if any of its variables is useless

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Productions

Variables $S \rightarrow A$ useless

useless $A \rightarrow aA$ useless

useless $B \rightarrow C$ useless

useless $C \rightarrow D$ useless

Removing Useless Productions

Example Grammar:

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

First: find all variables that can produce strings with only terminals

$$S \to aS \mid A \mid C$$

$$A \to a$$

$$B \to aa$$

$$C \to aCb$$

Round 1: $\{A, B\}$

$$S \to A$$

Round 2: $\{A, B, S\}$

Keep only the variables
that produce terminal symbols: $\{A, B, S\}$

(the rest variables are useless)

$$S \to aS \mid A \mid \cancel{C}$$

$$A \to a$$

$$B \to aa$$

$$\cancel{C \to aCb}$$

$$S \to aS \mid A$$

$$A \to a$$

$$B \to aa$$

Remove useless productions

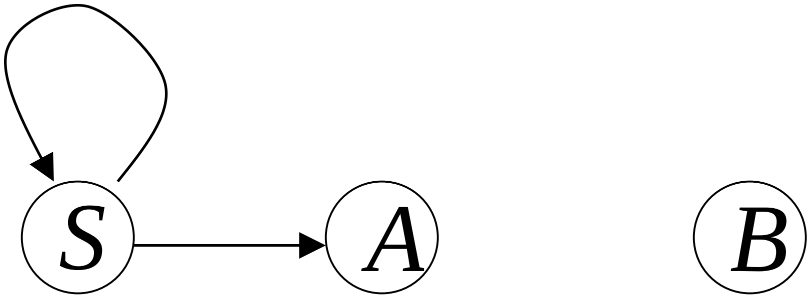
Second: Find all variables reachable from S

Use a Dependency Graph

$$S \to aS \mid A$$

$$A \to a$$

$$B \to aa$$



not reachable

Keep only the variables reachable from S

(the rest variables are useless)

Final Grammar

$$S \to aS \mid A$$

$$A \to a$$

$$\cancel{B \to aa}$$

$$S \to aS \mid A$$

$$A \to a$$

Remove useless productions

Set of variables that Derive terminal symbols

- Input = CFG (V, ∑, P , S)

- TERM = { A | there is a rule A→w ε P with w ε ∑*
- repeat
  - PREV = TERM
  - For each variable in A ε V do
    - If there is a rule A → w and w ε (PREV U ∑)* then TERM = TERM U {A}
- Until PREV = TERM

Example

- Consider following CFG

G: $S \rightarrow AC \mid BS \mid B$

$A \rightarrow aA \mid aF$

$B \rightarrow CF \mid b$

$C \rightarrow cC \mid D$

$D \rightarrow aD \mid BD \mid C$

$E \rightarrow aA \mid BSA$

$F \rightarrow bB \mid b$

$S \rightarrow AC \mid BS \mid B$
$A \rightarrow aA \mid aF$
$B \rightarrow CF \mid b$
$C \rightarrow cC \mid D$
$D \rightarrow aD \mid BD \mid C$
$E \rightarrow aA \mid BSA$
$F \rightarrow bB \mid b$

- New Grammar from TERM will be

G_T:
$S \rightarrow BS \mid B$
$A \rightarrow aA \mid aF$
$B \rightarrow b$
$E \rightarrow aA \mid BSA$
$F \rightarrow bB \mid b$

Iteration	TERM	PREV
0	{B, F}	{}
1	{B, F, A, S}	{B, F}
2	{B, F, A, S, E}	{B, F, A, S}
3	{B, F, A, S, E}	{B, F, A, S, E}

Construction of set of reachable Variables

- Input = CFG (V, ∑, P , S)
- REACH = {S}

1. PREV = null
2. repeat
   i. NEW = REACH – PREV
   ii. PREV = REACH
   iii. For each variable A in NEW do
      i. For each rule A → w do add all variables in w to REACH
3. Until REACH = PREV

G_T:

$S \rightarrow BS \mid B$

$A \rightarrow aA \mid aF$

$B \rightarrow b$

$E \rightarrow aA \mid BSA$

$F \rightarrow bB \mid b$

G'_T:

$S \rightarrow BS \mid B$

$B \rightarrow b$

Iteration	REACH	PREV	NEW
0	{S}	{}	{}
1	{S, B}	{S}	{S}
2	{S, B}	{S, B}	{B}
3	{S, B}	{S, B}	{}

Removing All

- **Step 1:** Remove Nullable Variables
- **Step 2:** Remove Unit-Productions
- **Step 3:** Remove Useless Variables

Augmented Grammar

- Add a new start symbol $S_0 \rightarrow S$
- This change guarantees that the start symbol of the new grammar will not occur on the *rhs* of any rule.

Removal of Left Recursion

- Consider the grammar A → Aa | b

- Halting condition of top-down parsing depend upon the generation of terminal prefixes to discover dead ends.
- Repeated application of above rule fail to generate a prefix that can terminate the parse.

Removing left recursion

- To remove left recursion from A, the ***A*** rules are divided into two groups.

  Left recursive and others

$A \rightarrow Au_1 \mid Au_2 \mid Au_3 \mid \ldots \mid Au_j$

$A \rightarrow V_1 \mid V_2 \mid V_3 \mid \ldots \mid V_k$

Solution:

$A \rightarrow V_1 \mid V_2 \mid V_3 \mid \ldots \mid V_k \mid V_1Z \mid V_2Z \mid \ldots \mid V_kZ$

$Z \rightarrow u_1Z \mid u_2Z \mid u_3Z \mid \ldots \mid u_jZ \mid u_1 \mid u_2 \mid u_3 \mid \ldots \mid u_j$

Removal of Left Recursion

Or Equivalently

$A \rightarrow Au_1 \mid Au_2 \mid Au_3 \mid \ldots. \mid Au_j$

$A \rightarrow V_1 \mid V_2 \mid V_3 \mid \ldots \mid V_k$

Solution:

$$A \rightarrow V_1Z \mid V_2Z \mid \ldots \mid V_kZ$$

$$Z \rightarrow u_1Z \mid u_2Z \mid u_3Z \mid \ldots \mid u_jZ \mid \lambda$$

Example

- $A \rightarrow Aa \mid b$

Solution:

$A \rightarrow bZ \mid b$

$Z \rightarrow aZ \mid a$

OR

$A \rightarrow bZ$

$Z \rightarrow aZ \mid \lambda$

- $A \rightarrow Aa \mid Ab \mid b \mid c$

$A \rightarrow bZ \mid cZ \mid b \mid c$

$Z \rightarrow aZ \mid bZ \mid a \mid b$

Consider another example

- $A \rightarrow AB \mid BA \mid a$
- $B \rightarrow b \mid c$

$A \rightarrow BAZ \mid aZ \mid BA \mid a$

$Z \rightarrow BZ \mid B$

$B \rightarrow b \mid c$

- The above transformations remove left-recursion by creating a right-recursive grammar; but this changes the associativity of our rules. Left recursion makes left associativity; right recursion makes right associativity. Example : We start out with a grammar :

$$Expr \rightarrow Expr + Term \,|\, Term$$
$$Term \rightarrow Term * Factor \,|\, Factor$$
$$Factor \rightarrow (Expr) \,|\, Int$$

- After having applied standard transformations to remove left-recursion, we have the following grammar :

$$Expr \rightarrow Term\ Expr'$$
$$Expr' \rightarrow + Term\ Expr' \,|\, \epsilon$$
$$Term \rightarrow Factor\ Term'$$
$$Term' \rightarrow * Factor\ Term' \,|\, \epsilon$$
$$Factor \rightarrow (Expr) \,|\, Int$$

- Parsing the string 'a + a + a' with the first grammar in an LALR parser (which can recognize left-recursive grammars) would have resulted in the parse tree:

```
            Expr
          /   |   \
      Expr    +    Term
     /  |  \          \
  Expr  + Term        Factor
   |       |         |
  Term    Factor     Int
   |       |
 Factor   Int
   |
  Int
```

- This parse tree grows to the left, indicating that the '+' operator is left associative, representing (a + a) + a.
- But now that we've changed the grammar, our parse tree looks like this :

```
     Expr ---
    /        \
  Term      Expr' --
   |       /  |     \
 Factor   +  Term   Expr' ------
   |          |      |  \        \
  Int       Factor   +  Term    Expr'
              |          |        |
             Int       Factor     ϵ
                         |
                        Int
```

- We can see that the tree grows to the right, representing a + (a + a). We have changed the associativity of our operator '+', it is now right-associative. While this isn't a problem for the associativity of addition, it would have a significantly different value if this were subtraction.

- The problem is that normal arithmetic requires left associativity. Several solutions are: (a) rewrite the grammar to be left recursive, or (b) rewrite the grammar with more nonterminals to force the correct precedence/associativity, or (c) if using YACC or Bison, there are operator declarations, %left, %right and %nonassoc, which tell the parser generator which associativity to force.

Expression Grammar

Ambiguous	non-Ambiguous	No Left Rec
$S \rightarrow E$	$S \rightarrow E$	$S \rightarrow E$
$E \rightarrow E + E$	$E \rightarrow E + T$	$E \rightarrow T\ E'$
$E \rightarrow E - E$	$E \rightarrow E - T$	$E' \rightarrow +T\ E' \mid -T\ E' \mid \lambda$
$E \rightarrow E * E$	$E \rightarrow T$	$T \rightarrow FT'$
$E \rightarrow E / E$	$T \rightarrow T * F$	$T' \rightarrow * FT' \mid / F\ T' \mid \lambda$
$E \rightarrow (E)$	$T \rightarrow T / F$	$F \rightarrow (E)$
$E \rightarrow num \mid id$	$T \rightarrow F$	$F \rightarrow num \mid id$
	$F \rightarrow (E)$	
	$F \rightarrow num \mid id$	

Left Factoring

The grammar is unambiguous and non-left-recursion but we don't know which rule to apply on a symbol in:

$$
\begin{aligned}
S &\rightarrow VU_1 \\
&\rightarrow VU_2 \\
&\rightarrow VU_3 \\
&\dots\dots\dots \\
&\rightarrow VU_n
\end{aligned}
$$

However, it can be factored as follows.

$$
\begin{aligned}
S &\rightarrow VB \\
B &\rightarrow U_1 \\
&\rightarrow U_2 \\
&\rightarrow U_3 \\
&\dots\dots\dots \\
&\rightarrow U_n
\end{aligned}
$$

- Should we apply right factoring too?