CS 3100 – Models of Computation – Fall 2010
Notes for Lecture 15 around Assignment 6 – 10/19/2010

Topics:

• Reversing CFGs
• Obtaining NFA from purely right-linear CFGs
• Simplifying CFGs (also related to nullability discussed on Page 76)
• Chomsky normal form
• CFL Pumping Lemma
• Yacc
• CFG to PDA: see online material against L14 (“JFLAP files”)
• PDA to CFG

1 Reversing CFGs

Given a string $s$, let $s^R$ denote the reverse of $s$

Given strings $s$ and $t$, $(st)^R = t^Rs^R$

Applying this recursively,

$(stu)^R = (tu)^Rs^R = u^Rt^Rs^R$

This idea can be applied to CFGs:

$S \rightarrow A \ B \ C \ | \ 0 \ D \ | \ E \ 0 \ | \ F \ 0 \ G \ | \ 0 \ H \ 2$

can be turned into an $Sr$ grammar as follows:

$Sr \rightarrow Cr \ Br \ Ar \ | \ Dr \ 0 \ | \ 0 \ Er \ | \ Gr \ 0 \ Fr \ | \ 2 \ Hr \ 0$

2 NFA from Purely right-linear

Reversing

$S \rightarrow A \ 0 \ | \ B \ 1 \ | \ e$
$A \rightarrow C \ 1 \ | \ 0$
$B \rightarrow C \ 1 \ | \ 1$
$C \rightarrow 1 \ | \ C \ 0$

We obtain

$Sr \rightarrow 0 \ Ar \ | \ 1 \ Br \ | \ e$
$Ar \rightarrow 1 \ Cr \ | \ 0$
$Br \rightarrow 1 \ Cr \ | \ 1$
$Cr \rightarrow 1 \ | \ 0 \ Cr$
and an NFA:

\[
\begin{align*}
ISr - 0 & \rightarrow Ar \\
ISr - 1 & \rightarrow Br \\
ISr - e & \rightarrow F1 \\
Ar - 1 & \rightarrow Cr \\
Ar - 0 & \rightarrow F2 \\
Br - 1 & \rightarrow Cr \\
Br - 1 & \rightarrow F3 \\
Cr - 1 & \rightarrow F4 \\
Cr - 0 & \rightarrow Cr
\end{align*}
\]

3. Simplifying CFGs

We can simplify this CFG as follows:

\[
\begin{align*}
S & \rightarrow A \mid B \\
A & \rightarrow ( W A \mid ( X C \\
B & \rightarrow ( W B \mid ( X D \\
P & \rightarrow 0 Q \mid 2 \\
Q & \rightarrow P 0 \mid 3 \\
W & \rightarrow ( W W \mid ( X Y \\
X & \rightarrow ( W X \mid ( X Z \\
W & \rightarrow ) \\
B & \rightarrow e
\end{align*}
\]

- Notice that \( C, D, Y, Z \) are not generating symbols (they can never generate any terminal string). Hence we can eliminate production RHS using them.
- \( W \) and \( B \) are generating (\( W \rightarrow \) ) and \( B \rightarrow e \).
- \( X \) is not generating. Look at \( X \rightarrow ( W X \). While \( ( \) is generating and \( W \) is generating, \( X \) on the RHS isn’t generating – we are doing a “bottom-up marking.” The same style of reasoning applies also to \( X \rightarrow ( X Z \).
- Even \( A \) is not generating!
- While \( P \) and \( Q \) are generating, they are not reachable.

3.1 Nullability

The algorithm for a generating non-terminal is similar to the following from Page 76 for nullable variables:

- Declare all variables (non-terminals) non-nullable.
Repeat Go thru productions; if any has RHS empty or all entries are nullable, then mark the LHS variable nullable
Until there is no increase in the set of nullable variables (non-terminals).
(The book uses “variables” ; we often use “non-terminals”)

4 Chomsky Normal Form

• Get rid of all \( \varepsilon \) productions.
• Get rid of unit productions.
• Make productions binary.
• Move all terminals to unit productions.

4.1 Derivation length

Derivation length for a string of length \( n \): \( 2n - 1 \). So we can search all derivations systematically using dynamic programming.

4.1.1 Cocke-Kasami-Younger (CKY) parsing algorithm

The CKY parsing algorithm uses dynamic programming in a rather elegant manner. Basically, given any string, such as 0 0 1, and a Chomsky normal form grammar such as

\[
S \rightarrow S T \mid 0
\]
\[
T \rightarrow S T \mid 1
\]

the following steps describe how we “parse the string” (check that the string is a member of the language of the grammar):

• Consider all possible substrings of the given string of length 1, and determine all non-terminals which can generate them.
• Now, consider all possible substrings of the given string of length 2, and determine all pairs of non-terminals in juxtaposition which can generate them.
• Repeat this for strings of lengths 3, 4, . . . , until the full length of the string has been examined.

Given string: 001

0 0 1
^ ^ ^ ^
\mid \mid \mid \mid
0 1 2 3 are the positions in the string. See who (which non-terminals) can generate these positions.

Attempt to span position 0 thru 3.
\{S\} can yield posn 0--1 and \{S,T\} can yield posn 1--3.

The concat of \{S\} and \{S,T\} is \{SS, ST\}.

Both S and T can yield ST. Neither can yield SS. Thus we mark the "1,3" "0,3" positions with \{S,T\}.

We can now say that S can generate the string from position 0 thru 3. Hence parsed!

5 The CFL Pumping Lemma

Basic idea: Very long string needs very tall parse tree; therefore some non-terminal along the path repeats. Can do “switharoo” of non-terminals to pump trees!

Given any CFG \( G = (N, \Sigma, P, S) \), there exists a number \( p \) such that given a string \( w \) in \( L(G) \) such that \( |w| \geq p \), we can split \( w \) into \( w = uvxyz \) such that \( |vy| > 0 \) (one of \( v \) or \( y \) is non-empty), \( |vxy| \leq p \), and for every \( i \geq 0 \), \( uv^i xy^i z \in L(G) \).

\[ S \rightarrow (S) \mid T \mid e \]
\[ T \rightarrow [T] \mid TT \mid e. \]

Here is an example derivation:

\[ S \Rightarrow (S) \Rightarrow ((T)) \Rightarrow (([T])) \Rightarrow (([[]])) \]

Occurrence-1 \hspace{2cm} Occurrence-2

Occurrence-1 involves Derivation-1: \( T \Rightarrow [T] \Rightarrow [] \)
Occurrence-2 involves Derivation-2: \( T \Rightarrow e \)

Here, the second \( T \) arises because we took \( T \) and expanded it into \([T]\) and then to \([\ ]\).

Now, the basic idea is that we can use Derivation-1 used in the first occurrence in place of Derivation-2, to obtain a longer string:

\[ S \Rightarrow (S) \Rightarrow (((T))) \Rightarrow ((([T]))) \Rightarrow ((([[T]]))) \Rightarrow ((([[[]]]))) \]

Occurrence-1 Use Derivation-1 here

In the same fashion, we can use Derivation-2 in place of Derivation-1 to obtain a shorter string, as well:

\[ S \Rightarrow (S) \Rightarrow ((T)) \Rightarrow (()) \]

Use Derivation-2 here