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, $(st)^R = t^R s^R$

Applying this recursively,

$(stu)^R = (tu)^R s^R = u^R t^R s^R$

This idea can be applied to CFGs:

```
S -> A B C | 0 D | E 0 | F 0 G | 0 H 2
```

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

```
Sr -> Cr Br Ar | Dr 0 | 0 Er | Gr 0 Fr | 2 Hr 0
```

2 NFA from Purely right-linear

Reversing

```
S -> A 0 | B 1 | e
A -> C 1 | 0
B -> C 1 | 1
C -> 1 | C 0
```

We obtain

```
Sr -> 0 Ar | 1 Br | e
Ar -> 1 Cr | 0
Br -> 1 Cr | 1
Cr -> 1 | 0 Cr
```

and an NFA

```
ISr - 0 -> Ar
ISr - 1 -> Br
Isr - e -> F1
Ar - 1 -> Cr
Ar - 0 -> F2
Br - 1 -> Cr
Br - 1 -> F3
Cr - 1 -> F4
Cr - 0 -> Cr
```

3 Simplifying CFGs

We can simplify this CFG as follows:

```
S -> A | B

A -> ( W A  |  ( X C
B -> ( W B  |  ( X D

P -> 0 Q | 2

Q -> P 0 | 3

W -> ( W W  |  ( X Y
X -> ( W X  |  ( X Z

W -> )
B -> e
```

- 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 -> )` and `B -> e`).
- `X` is not generating. Look at `X -> ( 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 -> ( 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 ε 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
^ ^ ^ ^
| | | |
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.
```

```
0         0               0                 0
a 1       {S} 1           {S} 1             {S}    1
b c 2     b   {S} 2       {}  {S}   2       {}    {S}   2
d e f 3   d    e  {T} 3   d   {S,T} {T} 3   {S,T} {S,T} {T} 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 $i \geq 0$, $uv^ixy^iz \in L(G)$.

```
S -> ( S ) | T | e

T -> [ T ] | T T | e.
```

Here is an example derivation:

```
S => ( S ) => (( T )) => (( [ T ] )) => (( [  ] ))
                 ^            ^

               Occurrence-1   Occurrence-2

Occurrence-1 involves Derivation-1: T => [ T ] => [ ]
Occurrence-2 involves Derivation-2: T => 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 => (S) => ((T))  => (( [ T ] )) => (( [[ T ]] )) => (( [[ ]] ))
              ^             ^

              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 => ( S ) => ( ( T ) ) => ( ( ) )
                  ^

                  Use Derivation-2 here
```