CSE396 Spr26 Lecture Tue. 3/24: Chomsky NF and the CFL Pumping Lemma

Prelim II most likely on Tuesday April 21.

First, some more examples and review. The following is a viable grammar for a fragment of Java:

```
E   -->  E2 ASSGTOP E | E2          //assignment is right-associative
E2  -->  E2 BINOP E3 | E3           //BINOPs are left-associative
E3  -->  +E3 | -E3 | ++P | --P | E4
E4  -->  P++ | P-- | P
P   -->  (E) | LITERAL | VARIABLE   (etc.)
ASSGTOP --> = | += | -=   (etc.)
BINOP   --> == | != | + | - | * | /   (etc.)
LITERAL --> /any number or quoted string etc./
VARIABLE--> /any legal identifier/.
```

The grammar does not specify *whitespace* rules. Rather, those are handled by a prior **lexing** stage of compilation. The C/C++/Java/C# family of languages generally fiollows the "greedy" lexing rule of making tokens as long as possible. The typifying example is that a string `+++` gets broken as `++ +` rather than `+ ++`, even if the latter would be found legal at later stages. With that said, the grammar presumes that whitespace has already been taken care of. So we can do the derivation

```
E ==> E2 ==> E2 BINOP E3 ==> E3 BINOP E3
   ==> E4 BINOP E3 ==> P++ BINOP E3 ==>^2 x++ BINOP E3 ==> x++ + E3
   ==> x++ + ++P ==>^2 x++ + ++y
```

Yes, this is actually kosher in C/C++/Java. The operation ++y returns a **reference** to the incremented variable y, but that is readable for a value just like y itself is. That value gets added to the post-incremented value of x. Now the code in CSE396parses.c actually wrote "x+++ ++y" which still gets lexed correctly, but "x+++++y" would get mis-lexed as if it were (x++)++ + y. That is illegal in the real world because (x++) is a pure value, so it can't be post-incremented again. With the parentheses, "(x++)++ + y" is actually legal in the grammar---try it!

A knottier question is whether a ++ token can ever be sandwiched between two + or - operations. As "x++ + ++y" shows, it can come before one or after one, but how about both at once? It's a little bit dicey, because the grammar has the rules `E3 --> +E3 | -E3` which allow putting as many unary + and - signs before a variable as you please. OK, the real-world reason you can't do

x + ++ +y

is that "+y" is again a pure value and cannot be pre-incremented. But does the grammar itself prevent this at the parsing stage? The answer is *yes*, and we can see it by structural induction. Assign to the grammar nonterminal `P` the meaning

- "Every string x I derive begins and ends with a parenthesis, alphanumeric char, or quotes."

This is immediately preserved by the productions for `P`. Then we can assign to all of `E,E2,E3,E4,P` the meaning,

- "Any ++ or -- token in a string I derive is immediately preceded or followed by a parenthesis, alphanumeric char, or quotes."

The "preceded or followed by" clauses rules out having a binary operator both before and after.

The other example to review is that *augmenting* an SI proof of soundness can often show that the grammar is *not* comprehensive for the stated property. For the property of deriving the language A of all positive even-length strings over $\Sigma = \{a, b\}$, consider the grammar $G =$:

$$S \rightarrow aSa \mid bY \mid Yb,$$
$$Y \rightarrow YS \mid a \mid b.$$

Is G **sound**, i.e., is $L(G) \subseteq A$? Answer is yes. To prove it via SI, define the following:

- P_S = "Every x that I derive has $|x| \geq 2$ and $|x|$ even."
- P_Y = "Every y that I derive has $|y|$ odd."

A run-through of each rule makes it clear that these properties hold. But is G **comprehensive**, meaning $L(G) \supseteq A$? A spot-check shows that G cannot derive the string "aa", nor "aaaa", and so on. We can highlight the issue by augmenting the property of S:

- P_S = "Every x that I derive has $|x| \geq 2$ and $|x|$ even and includes at least one b."

OK, is G comprehensive for $A \setminus (aa)^+$? I.e., does every even-length string with at least one b belong to $L(G)$? That makes a good self-study exercise...

Chomsky Normal Form

A language L is a **context-free language** (CFL) if there is a context-free grammar G such that $L = L(G)$. We will show that every CFL has a grammar in a particular form:

Definition: A CFG $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** (**ChNF**) if every rule has the form

- $A \rightarrow c$, with $c \in \Sigma$, or
- $A \rightarrow BC$, where $A, B, C \in V$ (note: we can have $B = C$ or $B, C = A$ etc.)

Most sources also require that the start symbol S cannot be on the right-hand side of a rule. That doesn't really matter, since one can make an "alias" S', give it the same rules as S, and then S' in place of S on right-hand sides of *all* rules. A grammar in ChNF cannot derive ϵ, but that is the only limitation:

Theorem 1: For every CFG G_0 we can build a CFG G in ChNF such that $L(G) = L(G_0) \setminus \{\epsilon\}$.

Proof: The main steps are to eliminate the following four kinds of rules:

1. **Null rules** of the form $A \to \epsilon$.
2. **Unit rules** of the form $A \to B$.
3. **Mixed rules** $A \to X$ where X has both terminals and variables.
4. **Long rules** $A \to X$ where $|X| \geq 3$.

Optionally, we can also 5. eliminate **deadwood** variables that cannot derive any terminal strings at all, removing all rules that have these variables once we identify them, and 6. delete **unreachable** variables that cannot occur in a derivation from S and their rules too.

The last two steps 3 and 4 are easy. For each terminal symbol $c \in \Sigma$ we can introducing the "aliasing variable" A_c whose only rule is $A_c \to c$, and use A_c in place of c everywhere else. If we have a higher-length rule $A \to B_1 B_2 \cdots B_\ell$ where $\ell \geq 3$, then we can introduce "step variables" $R_2, \ldots R_{\ell-1}$ and replace the long rule by the binary rules $A \to B_1 R_2$, $R_2 \to B_2 R_3$, up through $R_{\ell-1} \to B_{\ell-1} B_\ell$. The new variables are not used anywhere else---if we have more long rules, then we make more "stepping variables" for each one. Provided we got rid of null and unit rules, the resulting grammar is in ChNF. (Yes, ChNF splurges on making new variables.) You can actually do those two steps first, but in practical terms they make an ugly mess. Exercises will emphasize starting with step 1 on the original grammar. If you do steps 5 and 6, they should be at the end and in that order.

The first two steps illustrate a nifty pattern in algorithm design that one may call "iterative growth with a terminating bound." Best just to jump right in:

Conversion to Chomsky NF Step 1

Define $NULLABLE$ to be the set of variables (in the original grammar G_0) that can derive ϵ in one or more steps. The "or more" makes this more complicated than the set $NULL$ of all variables that have null rules. Now run the following code.

1. $NULLABLE = NULL$;
2. `keepGoing = true;`
3. `while (keepGoing) {`
4. `    keepGoing = false;`

5. `for` (each rule $A \to X$ where $A \notin NULLABLE$) {
6. `if` ($X \in NULLABLE^*$) `then` {
7. $NULLABLE$ += $\{A\}$;
8. `keepGoing = true;`
9. }
10. }
11. }
12. return $NULLABLE$;

The use of a Boolean "flag" like `keepGoing` imay be "bad style" but it's a useful crutch. This loop

- **terminates** because if `keepGoing` is made true again, then a new variable gets added to the set NULLABLE, but this set cannot grow to be larger than the orioginal set V_0 of variables in G_0. So eventually the while-loop exits.
- is **sound** because whenever a new variable A is added, it came from a rule $A \to X$ where every character in X is already in $NULLABLE$, so the whole right-hand side can derive a string of ϵ's, so A can derive ϵ.
- is **comprehensive** because whenever B can derive ϵ in some number m of steps, that derivation must begin with a rule $B \to Y$ where all elements of Y can derive ϵ in at most $m-1$ steps. By "strong induction" on m (of a kind we have just seen in comprehensiveness proofs for languages of grammars), all elements of Y get placed into the set $NULLABLE$. Since the last such placement sets keepGoing = true, the for-loop activates another time and works on the rule $B \to Y$, whereupon B is added to $NULLABLE$. Thus the final output includes all variables that can derive ϵ.

This last part is a mouthful, but if you get it, then you grok induction at "programming ninja" level. The case $Y = \epsilon$ is the base case. Actually, we don't even need to be conscious of a base case at all. We can initialize $NULLABLE = \varnothing$ in step 1. By the regular expression identity $\varnothing^* = \epsilon$, the rules $A \to \epsilon$ automatically satisfy the "$X \in NULLABLE^*$" condition in step 6, so all the original null rules and their variables get added "inductively" in the first iteration of the for-loop.

You can get Step 5 by the same kind of loop. Use a set called $LIVE$ instead of $NULLABLE$. Initialize $LIVE = \Sigma$ in step 1 of the code. Then the revised steps 5 and 6 of the code tell you that if $A \to X$ is a rule and $X \in LIVE^*$, then A too derives a terminal string, so we can add A to the set $LIVE$. On termination, all variables not in $LIVE$ are deadwood.

One final thing to note is that if the grammar has k variables and r rules, then the total time of the while-loop is basically $O(kr)$, which is **"polynomial in the overall size of the grammar G_0."**

What we **do** with the set $NULLABLE$ is not "polynomial", however. Since we haven't done step 4 yet, there might be long rules $A \to X$ where X has a sizable number ℓ of occurrences of nullable variables (the same nullable variable might appear more than once in X):

Make 2^ℓ rules that correspond to deleting every possible subset of the occurrences of nullable variables. This includes preserving the rule $A \to X$. (The new rules are **sound** because we could always emulate them in the original grammar G_0 by deriving the deleted variables to ϵ in the first place.)

Then finally **delete all null rules** and call the resulting grammar G_1. If $B \to \epsilon$ is a null rule where B is not the start symbol, then we could only use it from another rule $A \to X$ where B occurrs in X. In that case, we could use the new rule where the B was already deleted from X. And so on. The only casualty is when we have the rule $S \to \epsilon$, either from the get-go in G_0 or because we have a rule $S \to B_1 B_2 \cdots B_\ell$ where every B_i is nullable, so deleting the whole set of them left $S \to \epsilon$. This ultimately is why we get $L(G_1) = L(G_0) \setminus \{\epsilon\}$. But every other string in the language of the original grammar gets preserved in the new one.

For a footnote, the text actualy avoids the exponential blowup by implicitly mixing in the idea of the long-rules step 4. Or you could explicitly do step 4 first. This avoids the 2^ℓ blowup, but can create a bigger "$O(kr)$" time on the loop by inflating both k and r.

Conversion to Chomsky NF Steps 2--6

- Make a directed graph whose nodes are all variables involved in unit rules $A \to B$.
- Take the **transitive closure** of this graph. (This too is done by "iterative growing.")
- For all cases of $A \Longrightarrow^* C$ in the transitive closure, and all rules $C \to Y$, add the rule $A \to Y$.
- Finally **delete** all unit rules.

The new rules are **sound** because you could have used the unit rules to do $A \Longrightarrow^* Y$ anyway in multiple steps. The whole thing is **comprehensive** because in any sequence like $A \Longrightarrow B \Longrightarrow C \Longrightarrow Y$ you can now do $A \Longrightarrow Y$ in one step, so no derivations are "lost." It is also important to note that no new ϵ-rules are introduced in this (nor any new long rules if you do step 4 ahead of time).

The reachability step is actually another instance of taking a transitive closure---or rather, doing a breadth-first search from S of the graph that given any rule $A \to X$ includes an edge (A, B) for every variable B occurring in X.

This finishes the proof of Theorem 1. ☒

For future reference, we note that using the text's rendition of the "nullables" step 1, the entire algorithm runs in time polynomial in k, r, and ℓ, where k is the number of variables, r is the number of rules, and ℓ is the maximum length of the right-hand side of a rule. Or we can just let n be the total size of the

grammar in characters, summing the lengths of all rules, and then we can say the running time is "polynomial in n." The text remarks on this in chapters 7--9 which we will just hunt-and-peck a few things from in the last two weeks. Well, in regard to Chapter 4 we will also take note that if a grammar G in ChNF derives a string x of length $n \geq 1$ at all, then it derives x in exactly $2n - 1$ steps, $n - 1$ of which use productions of the form $A \rightarrow BC$, and n to fill in terminal symbols one at a time.

The CFL Pumping Lemma

The main benefit of Chomsky normal form is conceptual, although ultimately cosmetic:

> For a grammar G in Chomsky NF, **all parse trees are binary trees**.

This will allow us to use intuition about binary trees gained in CSE250 and other courses. The text uses ℓ-ary trees where ℓ is the maximum length of the right-hand side of a rule, but IMHO this generalitty makes its formulas a little harder to visualize. The theorem statement itself is not affected:

Theorem: For any context-free language L there is a number N (called the "CFL pumping length") so that for any $x \in L$ with $|x| \geq N$, we can break x into five substrings as $x =: uvwyz$ such that:

- $|uvw| \leq N$,
- at least one of u and w is nonempty, and
- for all $i \geq 0$, the "pumped string" $x^{(i)} = yu^ivw^iz$ also belongs to L. In particular with $i = 0$, the string yvz belongs to L. (This is called "pumping down.")

Proof: We can take a grammar G **in ChNF** such that $L(G) = L \setminus \{\epsilon\}$. (The presence or absence of ϵ in L ultimately doesn't matter.) Then G has some number k of variables. Take $N = 2^k$. Now consider any $x \in L$. Then x has a parse tree T in the grammar G. The root of the tree is S.

- If every path from the root has k or fewer internal nodes, then T has at most 2^{k-1} leaves---which means it can't yield a terminal string as long as x.
- So there must be path of $k + 1$ or more internal nodes.
- Among the bottom $k + 1$ internal nodes of that path, some variable A must appear at least twice. (This is another case of the Pigeonhole Principle.)
- The yield of the lower A becomes the string v. The yield of the upper A also gives us u to the left of v and w to the right of v. Because the upper A is at most $k + 1$ above the leaves, its whole yield uvw had size at most $2^k = N$.
- If the lower A is reached from the upper A by left-children only (i.e., an all-leftward path), then we get $u = \epsilon$. If by an all-rightward path then $w = \epsilon$. If the path has both left and right turns, then both u and w are nonempty. In any event, they can't both be empty.
- We can make a legal parse tree by overwriting the upper A by the subtree of the lower A. That zaps both u and w, leaving a parse tree for the string yvz. Which therefore belongs to L; it is the $i = 0$ case of $x^{(i)} = yu^ivw^iz$.

- We can also graft a copy of the whole subtree of the upper A onto the lower A. The lower A and its derivation of v still occurs in the copy. But we get both u and w twice over, giving the string $x^{(2)} = yu^2vw^2z$. [Technote: It is not literally the string x^2 which would be $yuvwzyuvwz$ as a double-word. Using the parentheses in "(2)" is a math way of saying it has something to do with 2 but not literal repetition of the whole x.]
- Grafting like that multiple times can give us $x^{(i)} = yu^ivw^iz$ for all $i \geq 3$ as well.

[All of these bullets are best viewed in pictures of binary trees. Those were shown in https://cse.buffalo.edu/~regan/cse396/CSE396lect040219.pdf
The continuation will pick up on Thursday with https://cse.buffalo.edu/~regan/cse396/CSE396lect040419.pdf .]