COMS W4115 Fall 2020: Homework Assignment 2 Solutions

Programming Languages and Translators

Deadline: Friday, November 6, 2020 by 11:59 PM

Overview

This homework assignment will test your knowledge of context-free grammars (CFGs) and parsing, which you have learned in class. Please submit your assignment via Gradescope by the deadline. This is an *individual* assignment, and all answers must be *typed*. However, you are allowed to hand-draw any parse trees and scan and attach them in your final submission. Finally, you must adhere to the academic integrity policies of the course. In all context-free grammars, assume that S is the start state unless otherwise specified.

Total Points: 85
Number of Problems: 5

Problems

1. **(8 Points)** For each of the following context-free grammars, state the number of valid inputs supported by that grammar. As an example, the grammar $G : E \to a \mid b \mid c$ results in 3 valid inputs—a, b, and c. Note that the empty string may be a valid input wherever applicable (this means that if an empty string is a possible input, you should consider it as part of your total count). You also do not need to list the different possible inputs.

   (a) **(4 Points)**

   $$\begin{gathered} S \to A \\ A \to BBBwx \\ B \to CCCyz \\ C \to a \mid b \end{gathered}$$

   There are **512** possible inputs with this grammar. $S \to A \to BBBwx \to CCCyzCCCyzCCCyzwx$. Each of the nine Cs can take on one of either a or b, so there is a total of $2^9 = 512$ possible options.

   (b) **(4 Points)**

   $$\begin{gathered} S \to D \\ D \to EEE \\ E \to FFF \\ F \to c \mid d \mid \epsilon \end{gathered}$$

   There are **1023** possible inputs with this grammar. $S \to D \to EEE \to FFFFFFFFF$. Each of the nine Fs can take on one of c, d, or ϵ. However, the number of possible inputs is not 3^9 because ϵ is not an actual token; rather, it results in many repeated input strings. You will notice there are 2^9 (512) length-9 strings, 2^8 (256) length-8 strings, 2^7 (128) length-7 strings, 2^6 (64) length-6 strings, 2^5 (32) length-5 strings, 2^4 (16) length-4 strings, 2^3 (8) length-3 strings, 2^2 (4) length-2 strings, 2^1 (2) length-1 strings, and the empty string. Thus, $512+256+128+64+32+16+8+4+2+1 = 1023$ possible options.

2. **(8 Points)** Suppose there is a context-free grammar G such that the language of G, $L(G)$, supports palindrome inputs, where inputs consist of characters from the alphabet $\Sigma = \{a, c, e, r\}$.

   Examples of valid inputs for L:

   - *racecar*
   - a
   - The empty string

   Examples of invalid inputs for L:

   - *race* (not a palindrome)
   - *abcba* ($b \notin \Sigma$)
   - *ack* (composite of the first two cases)

   (a) **(6 Points)** Construct G.

   The grammar should be able to produce all palindrome inputs with the characters a, c, e, and r, as follows:

   $$S \rightarrow aSa \mid cSc \mid eSe \mid rSr \mid a \mid c \mid e \mid r \mid \epsilon$$

   (b) **(2 Points)** Is L a regular language? Explain why or why not.

   L is NOT a regular language because palindrome inputs require memory, *i.e.*, the ability to determine whether the string seen so far has reflectional symmetry with the rest of the string at the point of reflection. Clearly, this is not the case with regular languages, as automata have no concept of memory. Additionally, the palindromes may be of arbitrary length, so it is not possible to restrict the length of the palindrome and to apply that knowledge beforehand when generating the regular expression/finite automaton.

3. **(18 Points)** *Recursive descent* is a parsing technique that generates a parse tree using a top-down approach. The technique takes advantage of backtracking when there is a mismatch between a token and a production rule. However, recursive descent is not free of its limitations.

   For each of the following context-free grammars, identify whether it is possible to apply recursive descent parsing on all possible inputs.

   - If it is possible, clearly explain why it is possible (you do not need to provide a formal proof). Further, provide two non-trivial examples (*e.g.*, not the empty string) of valid inputs for the given CFG.
   - If it is not possible, clearly explain why it is not possible and provide an example of an input that would not be properly parsed by the given CFG using recursive descent. Finally, rewrite the production rules of the CFG (except for $S \to E$; leave it as is), using as few production rules as possible, such that recursive descent can be applied.

   (a) **(6 Points)**

$$\begin{aligned} S &\to E \\ E &\to XR \mid \epsilon \\ X &\to \epsilon \mid X \\ R &\to \text{int} * E \mid \epsilon \end{aligned}$$

   It is not possible to apply recursive descent parsing on all possible inputs with this grammar because there is left-recursion in the grammar occurring with rule $X \to X$. Although the rule $X \to \epsilon$ is also possible, there is no specific order of the rules in terms of parsing, *i.e.*, $X \to \epsilon$ is equally likely as $X \to X$ to be chosen. It does not matter whether or not $X \to X$ is ever chosen for an input; as long as there is some left-recursion anywhere in the grammar, recursive descent parsing fails.

   While $X \to \epsilon \mid X$ does not seem to be of the form $S \to S\alpha \mid \beta$, you should notice that $\alpha = \epsilon$ and $\beta = \epsilon$.

   One example of an input that would not be properly parsed by the grammar is int $*$, as the recursive descent parser might always choose the rule $X \to X$ in trying to resolve the first input token *int*.

   To simplify this grammar, we can eliminate left-recursion as done in the slides. However, you may also recognize that $X \to \epsilon \mid X$ is trivially a no-op. X's rules can be removed entirely, and all uses of X can simply be replaced with ϵ. Thus, we can create the following grammar:

$$\begin{aligned} S &\to E \\ E &\to \text{int} * E \mid \epsilon \end{aligned}$$

   (b) **(6 Points)**

$$\begin{aligned} S &\to E \\ E &\to AB \mid \epsilon \\ A &\to (A) \mid \text{int} \mid +E \\ B &\to \text{int}\ B \mid * B \mid * \mid \text{int} \end{aligned}$$

   It is possible to apply recursive descent on all possible inputs, as there is no left-recursion (direct or indirect) in this grammar. While this grammar could be left-factored, our recursive descent parser is a general parser that is able to handle grammars that are not left-factored (as was discussed numerous times on Piazza and even added as a homework update).

We accepted any two *non-trivial* valid inputs for this grammar. Two examples are (int) $*$ int and $+$ int $*$ int $*$ int.

(c) **(6 Points)**

$$\begin{gathered} S \to E \\ E \to WXY \\ W \to a \mid Yb \\ X \to (W) \mid Z * Z \\ Y \to \epsilon \mid E + X \\ Z \to \text{int } Z \mid \text{int} \mid + \mid \epsilon \end{gathered}$$

It is not possible to apply recursive descent parsing on all possible inputs with this grammar because there is *indirect* left-recursion in the grammar occurring with rule $E \to^+ E + XbXY$.

One example of an input that would not be properly parsed by the grammar is $a(a)+(a)b$ int$*$int. Obviously, any other valid inputs from this grammar would also be accepted.

To simplify this grammar, we can eliminate left-recursion as done in the slides. Due to the indirect left-recursion, the elimination of left-recursion is not as straightforward, but it is easier to start with $E \to aXY \mid bXY \mid E + XbXY$ as the new derivations for E (by mere substitution for W). Thus, we can create the following grammar by eliminating left-recursion:

$$\begin{gathered} S \to E \\ E \to aXYE' \mid bXYE' \\ E' \to +XbXYE' \mid \epsilon \\ W \to a \mid Yb \\ X \to (W) \mid Z * Z \\ Y \to \epsilon \mid E + X \\ Z \to \text{int } Z \mid \text{int} \mid + \mid \epsilon \end{gathered}$$

You'll notice that there is no more left-recursion in this new grammar (check each of the left-most non-terminals to convince yourself), so recursive descent can be applied normally. We can't eliminate any left-most non-terminals as is because E and E' both still depend on X and Y, and X depends on W. On the bright side, handling inputs like $a(a)+(a)b$ int$*$int should now be a breeze!

Since we did specify to use as few productions as possible, you can substitute W in the first rule for X with the derivations from W to get (a) and (Yb) (which gets rid of W and reduces the total number of rules needed by 1):

$$\begin{gathered} S \to E \\ E \to aXYE' \mid bXYE' \\ E' \to +XbXYE' \mid \epsilon \\ X \to (a) \mid (Yb) \mid Z * Z \\ Y \to \epsilon \mid E + X \\ Z \to \text{int } Z \mid \text{int} \mid + \mid \epsilon \end{gathered}$$

Since we have a general recursive descent parser, we don't have to worry about removing the left-factoring (on the "(") with this substitution. However, this extra substitution isn't really necessary to get full credit. We were mostly looking for proper handling of the left-recursion.

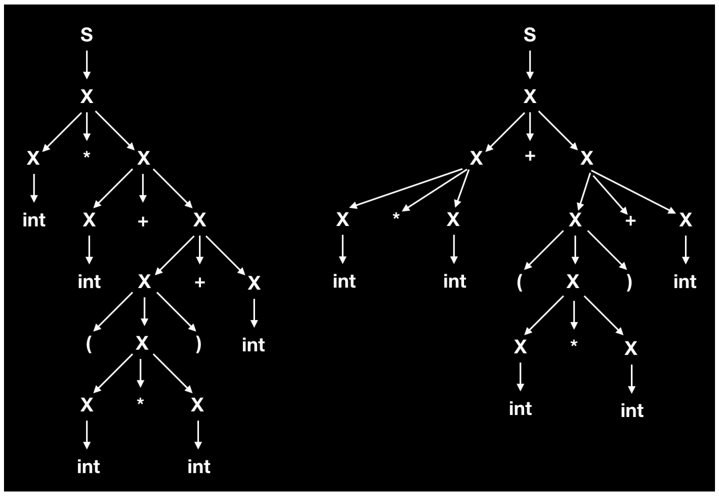
4. **(26 Points)** Consider a context-free grammar G given by the following set of production rules:

$$S \to X$$
$$X \to X * X \mid X + X \mid (X) \mid \text{int}$$

Further, suppose we have an input stream of tokens given by $S = 8 * 2 + (10 * 4) + 9$.

(a) **(10 Points)** Using the CFG G, construct two parse trees for S.

We accepted any two out of the C possible unique parse trees. Here is a sample answer:



(b) **(4 Points)** Let C be the number of unique parse trees that can be generated for S, given G. Find C.

The correct value of C is **5**. You should be able to reason why $C = 5$ based on the derivations you can arrive at. As you can see from part (a), you can arrive at different parse trees by choosing the first derivation differently (the first parse tree uses the rule $X * X$, while the second uses the rule $X + X$).

(c) **(12 Points)** A CFG is considered *ambiguous* if there is more than one left-most derivation tree or more than one right-most derivation tree.

Is G an ambiguous grammar? If it is not ambiguous, please explain why not. If it is ambiguous, explain how you would resolve this ambiguity, and rewrite the production rules of G (except for $S \to X$; leave it as is), using as few production rules as possible, such that G becomes unambiguous.

The grammar is ambiguous, as it is possible to generate more than one left-most derivation tree for the given input string S. Here are two possible left-most derivations (that would result in different derivation trees) to demonstrate this ambiguity:

$$S \to X \to X * X \to \text{int} * X \to \text{int} * X + X \to \text{int} * \text{int} + X \to \text{int} * \text{int} + X + X \to$$
$$\text{int} * \text{int} + (X) + X \to \text{int} * \text{int} + (X * X) + X \to \text{int} * \text{int} + (\text{int} * X) + X \to$$
$$\text{int} * \text{int} + (\text{int} * \text{int}) + X \to \text{int} * \text{int} + (\text{int} * \text{int}) + \text{int}$$

$$S \rightarrow X \rightarrow X + X \rightarrow X \ * \ X + X \rightarrow \text{int} \ * \ X + X \rightarrow \text{int} \ * \ \text{int} + X \rightarrow \text{int} \ * \ \text{int} + X + X \rightarrow$$
$$\text{int} \ * \ \text{int} + (X) + X \rightarrow \text{int} \ * \ \text{int} + (X \ * \ X) + X \rightarrow \text{int} \ * \ \text{int} + (\text{int} \ * \ X) + X \rightarrow$$
$$\text{int} \ * \ \text{int} + (\text{int} \ * \ \text{int}) + X \rightarrow \text{int} \ * \ \text{int} + (\text{int} \ * \ \text{int}) + \text{int}$$

There are many ways to eliminate the ambiguity. One such way is to implement precedence of multiplication over addition in the evaluation of an expression. Doing so results in the following grammar:

$$S \rightarrow X$$
$$X \rightarrow Y + X \mid Y$$
$$Y \rightarrow \text{int} \mid (X) \mid \text{int} * Y \mid (X) * Y$$

Note that while the addition rule for X ($X \rightarrow Y + X$) is always chosen first in the derivation (since input S contains addition, $X \rightarrow Y + X$ is guaranteed to be chosen), all of Y will be evaluated separately and all of X will be evaluated separately; those two evaluations are then added together, as desired by order of operations. Essentially, the rules for Y enforce that multiplication happens before addition. See the left-most derivation below for input S based on the reconstructed grammar:

$$S \rightarrow X \rightarrow Y + X \rightarrow \text{int} \ * \ Y + X \rightarrow \text{int} \ * \ \text{int} + X \rightarrow \text{int} \ * \ \text{int} + Y + X \rightarrow$$
$$\text{int} \ * \ \text{int} + (X) + X \rightarrow \text{int} \ * \ \text{int} + (Y) + X \rightarrow \text{int} \ * \ \text{int} + (\text{int} \ * \ Y) + X \rightarrow$$
$$\text{int} \ * \ \text{int} + (\text{int} \ * \ \text{int}) + X \rightarrow \text{int} \ * \ \text{int} + (\text{int} \ * \ \text{int}) + Y \rightarrow \text{int} \ * \ \text{int} + (\text{int} \ * \ \text{int}) + \text{int}$$

Here, there is only one choice of substitution at any derivation step. You should see this is the case with right-most derivation as well. Putting these together, you can reason that there is no longer any ambiguity in the grammar!

There are other possibilities for how to rewrite the grammar, but this is just one example. Note that we would ideally like to reconstruct a grammar (such as the one in this example) that does not include left-recursion, but we did not penalize students who did so.

5. **(25 Points)** Consider a simple markup language L that uses tags (similar to those in HTML and XML). To create tags, L supports the following tokens as terminal states: $\{<, >, /, =, word\}$.

   The following list provides the criteria for this language:

   - Every tag begins with $<$ and ends with $>$.
   - A tag may be an opening tag or a closing tag.
   - Inside an opening tag, the first token that follows $<$ is a word ($word$) representing the name of the tag. The tag also contains an optional list of *attributes*, which are pairs of words connected by $=$ (*i.e.*, $word = word$).
   - In a closing tag, the first token that follows $<$ is $/$, followed by the name of the tag (again, $word$). There are no attributes in the closing tag.
   - Every opening tag must be paired with a matching closing tag.
   - Any number of words or tags may appear between an opening tag and a closing tag.

   Examples of valid inputs for L:

   - $< word >< /word >$
   - $< word > word < /word >$
   - $< word >< word\ word = word > word\ word\ word < /word >< /word >$
   - $< word\ word = word\ word = word >< /word >$
   - $< word > word\ word < word\ word = word >< /word >< /word >$
   - $< word > word < word\ word = word > word < /word > word < /word >$

   Examples of invalid inputs for L:

   - $<>< / >$ (no presence of *word* following $<$ in the opening tag and following $/$ in the closing tag)
   - $< word >$ (no matching closing tag)
   - $< word\ word =>< /word >$ (incomplete attribute)
   - $< word >< /word\ word = word >$ (closing tag contains attributes)

   This problem has several edge cases, so to simplify the problem, we have these additional assumptions:

   - Every tag must either be an opening tag or a closing tag; there are no other possibilities.
   - Assume that there always exists a space between two words; therefore, *wordword* is equivalent to *word word*. You do not have to worry about spaces.
   - Include \$, the character to indicate the end of the token stream, in your answers wherever applicable.
   - *word* is a single terminal, just like *int*; treat it as one token, not four individual character tokens.
   - There can be words outside of a set of opening and closing tags, as long as this set of tags is nested within a set of outer tags encapsulating the words (see valid examples 5 and 6 above).
   - It is valid to have an ambiguous grammar, as long as it is correct.
   - You do not have to worry about whether the opening and closing tags are the same.
   - Only consider the criteria explicitly defined in this problem; do not make assumptions related to real markup languages.

(a) **(5 Points)** Construct a context-free grammar G for the language L.

Let S be the start state, O represent the non-terminal for the opening tag, C represent the non-terminal for the closing tag, A represent the non-terminal for any attributes, and I represent the non-terminal for any intermediate text/tags. The correct CFG G for the language L is as follows:

$$\begin{gathered} S \to OICS \mid \epsilon \\ O \to < word\ A > \\ C \to < /word > \\ A \to word = word\ A \mid \epsilon \\ I \to SI \mid word\ I \mid \epsilon \end{gathered}$$

(b) **(8 Points)** Write the $FIRST$ and $FOLLOW$ sets for G.

Here are the $FIRST$ sets for G:

$$\begin{gathered} FIRST(S) = \{<, \epsilon\} \\ FIRST(O) = \{<\} \\ FIRST(C) = \{<\} \\ FIRST(A) = \{word, \epsilon\} \\ FIRST(I) = \{word, <, \epsilon\} \\ FIRST(<) = \{<\} \\ FIRST(>) = \{>\} \\ FIRST(/) = \{/\} \\ FIRST(=) = \{=\} \\ FIRST(word) = \{word\} \end{gathered}$$

Here are the $FOLLOW$ sets for G:

$$\begin{gathered} FOLLOW(S) = \{word, <, \$\} \\ FOLLOW(O) = \{word, <\} \\ FOLLOW(C) = \{word, <, \$\} \\ FOLLOW(A) = \{>\} \\ FOLLOW(I) = \{<\} \\ FOLLOW(<) = \{word, /\} \\ FOLLOW(>) = \{word, <, \$\} \\ FOLLOW(/) = \{word\} \\ FOLLOW(=) = \{word\} \\ FOLLOW(word) = \{word, =, <, >\} \end{gathered}$$

(c) **(10 Points)** Show the $LL(1)$ parsing table for G.

Here is the $LL(1)$ parsing table for G:

Left-Most Non-Terminal	Next Input Token					
	$word$	$<$	$>$	$=$	$/$	$\$$
S	$S \to \epsilon$	$S \to OICS$, $S \to \epsilon$				$S \to \epsilon$
O		$O \to < word\ A >$				
C		$C \to < /word >$				
A	$A \to word = word\ A$		$A \to \epsilon$			
I	$I \to SI$, $I \to word\ I$	$I \to SI$, $I \to \epsilon$				

(d) **(2 Points)** Is G an $LL(1)$ grammar? Explain why or why not.

G is NOT an $LL(1)$ grammar because from the parsing table constructed in part (c), it is evident that there are a few scenarios in which multiple possible productions can be chosen for a next input token given a left-most non-terminal:

i. Next input token: $<$, left-most non-terminal: S, conflicting rules: $S \rightarrow OICS$, $S \rightarrow \epsilon$

ii. Next input token: $word$, left-most non-terminal: I, conflicting rules: $I \rightarrow SI$, $I \rightarrow word\ I$

iii. Next input token: $<$, left-most non-terminal: I, conflicting rules: $I \rightarrow SI$, $I \rightarrow \epsilon$

In an $LL(1)$ grammar, there can be no such conflicts.