Problems for Week Nine

Context-Free Grammars

Here's some practice problems to help you get comfortable designing CFGs. There are a number of patterns that come up over the course of these problems, and we hope that by the time you've finished working through them you have a deeper understanding of how CFGs work!

i. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ has no a's or has no b's $\}$. Write a CFG for L.

ii. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ has at least one a and at least one b $\}$. Write a CFG for L.

iii. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ \mathtt{a}^n\mathtt{b}\mathtt{a}^n \mid n \in \mathbb{N} \}$. Write a CFG for L.

iv. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ \mathtt{a}^n\mathtt{b}^{2n} \mid n \in \mathbb{N} \}$. Write a CFG for L.

v. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ \mathtt{a}^n\mathtt{b}^m \mid n, m \in \mathbb{N}$ and $n \le m \le 5n \}$. Write a CFG for L.

vi. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ \mathtt{a}^n\mathtt{b}^m \mid n, m \in \mathbb{N}$ and $n \ne m \}$. Write a CFG for L.

vii. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and let $L = \{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^p \mid n, m, p \in \mathbb{N}$ and $n = m$ or $n = p \}$. Write a CFG for L.

viii. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$. Write a CFG for L^*, the Kleene closure of L.

ix. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{\mathtt{a}^n\mathtt{b}^m \mid n, m \in \mathbb{N}$ and either $n{=}2m$ or $m{=}2n \}$. Write a CFG for L.

x. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$. Write a CFG for $\overline{L}$, the complement of L.

Turing Machines

Although much of our discussion of Turing machines takes place at a high level, it's still instructive to try to design Turing machines at the level of individual states.

i. Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a palindrome $\}$ (recall that a palindrome is a string that's the same when read forwards and backwards). Draw a state-transition diagram of a TM for L.

ii. Draw the state-transition diagram for a TM whose language is $\{ \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N} \}$.

The Story So Far

From the “lava diagram” in lecture, you probably noticed that

$$\mathbf{REG} \subsetneq \mathbf{R} \subsetneq \mathbf{RE}$$

Here, **REG** is the class of all regular languages, **R** is the class of all decidable languages, and **RE** is the class of all recognizable languages.

i. Show that $\mathbf{REG} \subseteq \mathbf{R}$. To do so, briefly explain how to directly turn a DFA for a language L into a decider for L.

ii. Show that $\mathbf{R} \subseteq \mathbf{RE}$. *(Hint: What's the definition of* ***R****? What's the definition of* ***RE****? Expand out the requisite terms and see what you find.)*

Closure Properties of R

This question explores various closure properties of **R**. Because **R** corresponds to decidable problems, languages in **R** are precisely the languages for which you can write a method

```
bool inL(string w)
```

such that

- for any string $w \in L$, calling `inL(`w`)` returns true.
- for any string $w \notin L$, calling `inL(`w`)` returns false.

This means that we can reason about closure properties of the decidable languages by writing actual pieces of code.

i. Let L_1 and L_2 be decidable languages over the same alphabet Σ. Prove that $L_1 \cup L_2$ is also decidable. To do so, suppose that you have methods `inL1` and `inL2` matching the above conditions, then show how to write a method `inL1uL2` with the appropriate properties. Then, briefly justify why your construction is correct.

ii. Repeat problem (i), except proving that the **R** languages are closed under concatenation.

Decidable Languages

All regular languages are decidable, but below is a purported proof that the regular language described by the regular expression `a*b` is undecidable:

Theorem: `a*b` is undecidable.

Proof: By contradiction; assume `a*b` is decidable. Let D be a decider for it. Consider what happens when we run D on a string of infinitely many `a`'s followed by a `b` and on a string of infinitely many `a`'s. Let's call this first string x and the second string y. Since D is a decider, it halts on all inputs, and therefore cannot run for an infinitely long time. Therefore, D must halt before reading the last character of x and the last character of y. Because x and y are the same except for their last character, we see that D must have the same behavior when run on x and when run on y. If D accepts x, then D also accepts y, but y is not in the language `a*b`. Otherwise, D rejects x, but x is in the language `a*b`. Both cases contradict the fact that D is a decider for `a*b`. We have reached a contradiction, so our assumption must have been wrong. Thus `a*b` is undecidable. ■

What's wrong with this proof?

Self-Reference

Self-reference is one of the trickier topics from the tail end of the quarter. This series of questions explores self-reference through a series of questions about a variety of different programs.

i. What does the following program do?

```
int main() {
    string input = getInput();
    string me = mySource();

    if (input == me) accept();
    else reject();
}
```

In the proof from lecture we did that A_{TM} is undecidable, we began by assuming that A_{TM} was decidable. That meant there must be some decider for A_{TM} , which, in software, we represent as a method

```
bool willAccept(string program, string input)
```

that takes as input the source code of a program and an input string, then returns true if the specified program will accept the specified input and returns false otherwise.

ii. Consider the following program:

```
int main() {
    string input = getInput();

    if (input == "") accept();
    else if (input[0] == input[input.length() - 1]) accept();
    else reject();
}
```

What happens if we run the above program with input `abba`? Why?

(Continued on the next page)

iii. Let `p1` be a string containing the source of the above program. What will happen if we call `willAccept(p1, "abba")`? Why?

iv. Consider this program:

```
int main() {
    string input = getInput();
    string target = "";

    while (target != input) target += "a";
    accept();
}
```

What happens if we run the above program with input `abba`? Why?

v. Let `p2` be a string containing the source of the above program. What will happen if we call `willAccept(p2, "abba")`? Why?

Self-reference for decidability can be a tricky topic. If you haven't yet done so, you should pause and go read the Guide to Self-Reference on the course website.

Let's look at the self-referential program we wrote in lecture that we used to show A_{TM} was undecidable:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

Let's go look at this code in some more detail.

vi. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the program accepts `abba`, then the program does not accept `abba`.

vii. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the does not accept `abba`, then it does accept `abba`.

viii. Explain why your answers to parts (vi) and (vii) collectively result in a contradiction that shows that A_{TM} is undecidable.

(Continued on the next page)

In the Guide to Self-Reference, we showed another self-referential program we could have written that would also help us see that A_{TM} is undecidable:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        while (true) {
            // Do nothing
        }
    } else {
        accept();
    }
}
```

Let's go look at this code in some more detail.

ix. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the program accepts `abba`, then it does not accept `abba`.

x. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the program does not accept `abba`, then it does accept `abba`.

xi. Explain why your answers to parts (ix) and (x) collectively result in a contradiction that shows that A_{TM} is undecidable.

In the Guide to Self-Reference, we showed a third self-referential program related to A_{TM} :

```
int main() {
    string me = mySource();
    string input = getInput();

    if (willAccept(me, input)) {
        accept();
    } else {
        reject();
    }
}
```

Let's go look at this code in some more detail.

xii. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the program accepts `abba`, then it does accept `abba`.

xiii. Suppose we feed the string `abba` as input to the above program. Explain why if `willAccept` says that the program does not accept `abba`, then it does not accept `abba`.

xiv. Explain why your answers to parts (xii) and (xiii) do ***not*** prove that A_{TM} is decidable.

Self-Reference and Decidability

Consider the language $L = \{ \langle M \rangle \mid M \text{ is a TM that accepts at least one string } \}$. This language is undecidable. Let's go see why this is.

i. Suppose for the sake of contradiction that $L \in \mathbf{R}$. This means that we could write a function

```
bool acceptsAtLeastOneString(string program)
```

that accepts as input the source code of a program, then returns true if the program accepts at least one string and returns false otherwise. Write a self-referential program that uses this function to obtain a contradiction. As a hint, recall the general template for these sorts of programs: have the program ask whether it accepts at least one string, then have it do the opposite of whatever it determines it's supposed to do.

ii. Formalize your reasoning from part (ii) by writing a formal proof that $L \notin \mathbf{R}$. To do so, follow the proof template from lecture: assume that $L \in \mathbf{R}$, describe what that assumption entails, write a program that causes a contradiction, then explain why you get a contradiction in all cases.