Model Answers

This document contains an explanation on how to address question 2b from the Sample Questions document (example1.pdf).

Question 2.b

This question deals with grammars that have left-recursion.

Recursion - to define something in terms of itself - is an important part of grammars. For example, the production $S \to \texttt{(}\ S\ \texttt{)}$ uses recursion: the definition of S uses S again.

We say that a grammar is (immediate) left recursive if one of its non-terminals is defined by rules that have the following form:

$$\begin{array}{lcl} X & \to & X\,\texttt{a} \\ X & \to & \texttt{b} \end{array}$$

where non-terminal X is defined recursively by the first production. We say that it is left recursive because X is the left-most term in one of its definitions, namely, X `a` - unlike $S \to \texttt{(}\ S\ \texttt{)}$, where S is the second term of the production.

A recursive descent parser (check lecture notes) is defined by mutually-recursive procedures, one per non-terminal or production of the grammar. The structure of a recursive descent parser closely resembles the grammar it implements. For example, the procedure that implements $S \to \texttt{(}\ S\ \texttt{)}$ will look like:

```
void S() {
   match(''('') S() match('')'');
}
```

Note that `S()` is recursive - it calls itself. We can see that the recursive descent parser implementing a left recursive grammar will run into problems. For example, check the possible implementation of the first production of the left recursive grammar introduced above:

```
void X() {
    X() a();
}
```

It's an infinite loop!! Therefore, in order to use a recursive descent parser we need to remove left-recursion from the grammar.

Next, we describe the general rule to rewrite a left recursive grammar such that left recursion is removed and the "meaning" of the grammar is preserved i.e. the language it defines must be the same!

We start by deriving the language defined by the general left recursive grammar introduced above. We show three derivations:

$$\begin{array}{l} X \to \texttt{b} \\ X \to X\texttt{a} \to X\texttt{aa} \to X\texttt{aaa} \to \texttt{baaa} \\ X \to X\texttt{a} \to X\texttt{aa} \to \ldots \to X\texttt{a...a} \to \texttt{ba...a} \end{array}$$

We can see that the language defined by the grammar is: `b, ba, baa, baaa, ba...a`. We can express the same language with the following rules:

$$\begin{array}{lcl} X & \to & \texttt{b}\,X' \\ X' & \to & \texttt{a}\,X' \\ X' & \to & \end{array}$$

where the last rule generates the empty sequence (check that it actually defines the same language!). The new grammar introduces a new non-terminal X', which is defined recursively but making sure that X' does not show up as the left-most element of the production.
Fact: The new grammar defines a general rule to remove left recursion.

We are given the following grammar:

$$\begin{array}{lccl}
(1) & E & \rightarrow & E \texttt{+} T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & T \texttt{*} F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
(6) & F & \rightarrow & \texttt{integer}
\end{array}$$

Where do we get left recursion? Non-terminals E and T are left recursive due to productions (1) and (3). If you apply the general rule to the productions that define E i.e. (1) and (2), we get the following:

$$\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \texttt{+}\ T\ E' \\
E' & \rightarrow &
\end{array}$$

You have to do the same with the productions that define T i.e. (3) and (4). What about F?