Model Answers

This document contains an an explnation on how to address question 1 from the Sample Questions document (example1.pdf).

More will follow soon.

Question 1.a

We are given the following grammar:

$$\begin{array}{lccc}
(1) & S & \rightarrow & S\,S \\
(2) & S & \rightarrow & \texttt{(}\ S\ \texttt{)} \\
(3) & S & \rightarrow & \texttt{a}
\end{array}$$

The first thing you need to do is to figure out the language described by the grammar. In our case, the grammar above describes strings made of `a`’s and balanced parentheses e.g. `a`, `a(a)`, `((aa))`, etc. String `((aa))` can be derived as follows:

$$S \xrightarrow{2} \texttt{(}S\texttt{)} \xrightarrow{2} \texttt{((}S\texttt{))} \xrightarrow{1} \texttt{((}\underline{S}\,S\texttt{))} \xrightarrow{3} \texttt{((}a\,S\texttt{))} \xrightarrow{3} \texttt{((}a\,a\texttt{))}$$

where non-terminal expansions (arrows) are labelled with the production’s number. We sometimes underline the non-terminal being expanded in order to improve readability.

The derivation shown above generates the following parse tree:

```
   S
   |
---|---
|  |  |
(  S  )
   |
---|---
|  |  |
(  S  )
   |
---|---
|     |
S     S
|     |
a     a
```

Going back to the question. A grammar is ambiguous if it can generate strings that can be represented by more than one parse tree. Therefore, it will be enough to show one string (from the language defined by the grammar) that can be represented by two different parse trees. Furthermore, the question asks for the shortest. We can clearly see that `a`, `aa` (lengths one and two) can only be represented by a single parse tree. What about strings of length 3? `(a)` is not ambiguos. What about `aaa`? we have at leats the following derivations:

$$\begin{array}{ll}
\text{Derivation 1} & S \xrightarrow{1} \underline{S}\,S \xrightarrow{1} S\,S\,S \xrightarrow{3} \texttt{a}\,S\,S \xrightarrow{3} \texttt{a a}\,S \xrightarrow{3} \texttt{a a a} \\
\text{Derivation 2} & S \xrightarrow{1} S\,\underline{S} \xrightarrow{1} S\,S\,S \xrightarrow{3} \texttt{a}\,S\,S \xrightarrow{3} \texttt{a a}\,S \xrightarrow{3} \texttt{a a a}
\end{array}$$

Note that even though the sema sequence of productions/rules was used, there were applied to different non-terminals i.e. left- and right-hand side of $S\ S$ in derivation1 and 2, respectively. The resulting parse trees are:

```
          S                    S
          |                    |
   --------------        -------------
   |            |        |           |
   S            S        S           S
   |            |        |           |
-------         a        a        -------
|     |                           |     |
a     a                           a     a
```

They are different and, therefore, the grammar is ambiguous.

Question 1b

You are asked to introduce the repeat-until statement to MiniJava. The syntax of the new statement is:

$$\text{Statement} \rightarrow \texttt{repeat}\ \text{Statement}\ \texttt{until(}\text{Exp}\texttt{)}$$

1. Abstract syntax removes all the information that is only used for parsing e.g. terminals, and keeps the parts that have semantic value. Examples of abstract syntax of the repeat-until statement follow:

   $$\text{RepeatUntil}(s, e) \qquad \text{RepeatStm}(\text{Stm}\ \mathit{body}, \text{Exp}\ \mathit{cond})$$

   where s and e represent an statement and expression, respectively.

2. The JavaCC code used to parse and create an AST for repeat-until is:

```
Statement RepeatUntil() :
{
  Statement s;
  Expression e;
}
{
  ''repeat'' s=Statement() ''until'' ''('' e=Expression() '')''
  { return new RepeatUntilStm(s,e); }
}
```

   Assumming that the classes `Statement` and `Expression` exist, as well as the respective non-terminals.

3. A RepeatUntil(s,e) statement is correctly types if:

   - Statement s is correctly typed.
   - Expression e is correctly typed.

- The type of e is bool.

Or, given function `TypeCheckStm`(s) that typechecks s, and `TypeCheckExp`(e) that type checks e and returns the resulting type:

- `TypeCheckStm`(s) ok
- `TypeCheckExp`(e) = bool

You can use a similar approach to express typechecking of the ShortIf expression introduced in the last coursework:

```
TypeCheckExp(e1?e2:e3) = t   where
                             TypeCheckExp(e1) = bool
                             TypeCheckExp(e2) = TypeCheckExp(e3) = t
```

4. Finally, you are asked to translate repeat-until into IR. In order to do this you need to have a clear understanding of repeat-until, which is straightforward: **execute statement s until expression e is true**.

   It can be expressed by the following control-flow graph:

```
              false
   .-----------------.
   |                 ^
   |                 |
Lstart ---> s ---> <e> ---> Lend
                     true
```

where Lstart and Lend are fresh labels (you can use the name you want) used to mark the beginning and end of the statement, and as targets of possible jumps. The IR translation of `repeat s until (x < 20)`, for example, where s stands for any statement, is as follows:

```
SEQ(LABEL(Lstart),
  SEQ(code(s),
    SEQ(CJUMP(LT,TEMP(x),CONST(20),Lend,Lbody),
      SEQ(LABEL(Lend)))))
```

where `code(s)` represents the code generated for s. As mentioned in the question, you do not need to generate specific code for the body - code(s) will be enough - but in order to write the CJUMP you need an example e.g. `(x < 20)`.