For this homework, you will be using \texttt{ISEngine2.tcl}. This engine is similar to \texttt{ISEngine.tcl}, except that it allows the user to keep the turn for multiple utterances. Simply check the box right before the ‘Send’ button if you want to make another utterance in the same turn.

The reason for allowing the user to make multiple utterances in the same turn is to allow the user to supply extra information, for which the user has not yet been prompted. In this homework, you will start from the base set of rules that you started from last time (\texttt{ISAgent.tcl}). Do not start from the agent rules that you handed in for the last homework.

**Question 1: User supplying extra information**

Add support for extra answers. You should only need to add a single understanding rule for processing extra answers. If you get a response that you can otherwise not process, you should check if it is the answer to something on the agenda. In checking the agenda, you should start with the most recent entry, to prefer its stack-like nature, but do not limit yourself to just the top item.

One way to do this is to add a procedure that you call in the precondition of the understanding rule to see which item on the agenda matches the user response. You can add an extra variable to the information state, called Temp, to temporarily hold this value.

Note that you do not need to worry about the ‘from’ versus the ‘to’ account. Just match against whichever is the most recent item added to the agenda.

Below is an interaction that the user should be able to have with the system.

\begin{verbatim}
System: query TransType
User : transfer
User : checking
User : 75
System: query To
User : savings
System: bye
\end{verbatim}

**Writeup 1.1** Hand in the code that you changed in ISAgent.tcl and indicate where it needs to be placed relative to the other understanding rules.

**Writeup 1.2** For the above dialogue, hand in a trace of the rules that are applied, as you did in the previous homework. Show the values for Agenda, QUD, TransType, From, To, and Amount after each rule is applied. For rule types, you can simply use ‘d’ for deliberate, ‘a’ for action, and ‘u’ for understanding rules. Also, show the agenda with the most top of the agenda at the end (as we implement it in Tcl). Show the information as a grid, as illustrated below. I have filled in the first few rows for you.
Writeup 1.3  In the Trindi paper you were given to read, the GoDiS system uses a different way to deal with the users giving extra information. Explain what they do, in terms of what rules would be executed and what variables in the information state are involved. Explain how this differs from what you did.

Question 2: Confirmations

Add the capability for the system to confirm the values that given by the user. You will do this from the base code in ISAgent.tcl, rather than building on to your answer from the previous question. This way you will not have to worry about interactions between confirmations and extra answers.

You should add the following variables to your information state: confirmed(TransType), confirmed(To), confirmed(From), and confirmed(Amount), and they should have an initial value of "", which means 'no'. These variables should either have a value of 'yes' or '""'.

Remember from the previous homework that Tcl has problems accessing indices of associate arrays that look like associative array accesses.

```
set is(confirmed(TransType)) yes
puts $is(confirmed(TransType))  # == won't work
puts $is([list confirmed(TransType)]) # == works!
```

Change your deliberate rules for From, To, and Amount so that they only adopt their goals if TransType is confirmed.

```
Change the **other-answer** understanding rule so that it adds a goal to the agenda of `confirm Slot Value`, where *Slot* is the variable name, and *Value* is the value of it that is being confirmed.\(^1\)

Add an action rule, **action-confirm**, for confirming that is similar to the action rule **ask**.

Also, add a rule, **confirm**, for understanding confirms (again that parallels **ask-self**). Add two rules for understanding the answer to the confirmation, one for understanding ‘yes’ (**confirm-answer-yes**), and one for understanding ‘no’ (**confirm-answer-no**).

Note, to get the message “confirm From savings” (or something similar) to work, you might have to enclose it using the list command, rather than quotes, or curly brackets.

Using these rules, you should be able to have a dialogue as shown below:

```
System: query TransType
User: transfer
System: confirm TransType transfer
User: yes
System: query Amount
User: 75
System: confirm Amount 75
User: no
System: query Amount
User: 50
System: confirm Amount 50
User: yes
System: query From
User: checking
System: confirm From checking
User: yes
System: query To
User: checking
System: confirm To checking
User: no
System: query To
User: savings
System: confirm To savings
User: yes
System: bye
```

**Writeup 2.4** Hand in all of the rules that you changed or modified. This should be 8 rules.

**Writeup 2.5** Complete the table below, which is for the first half of the dialogue given above. Show the values for Agenda, QUD, TransType, confirm(TransType), Amount, and confirm(Amount) after each rule is applied.

---
\(^1\)You could argue that it might be best to not add this confirm goal to the agenda in the understanding rule, but to instead use a separate deliberation rule, or to use a separate class of grounding rules to adopt these goals. You could also argue that until the value is confirmed, the original question should stay on the QUD. However, we will just keep things simple here.
<table>
<thead>
<tr>
<th>Agenda</th>
<th>QUD</th>
<th>TransType</th>
<th>Value</th>
<th>Amount</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d setup</td>
<td>a ask</td>
<td>S query TransType</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u ask-self</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U transfer</td>
<td></td>
<td>u other-answer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a confirm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S confirm TransType transfer</td>
<td></td>
<td>u confirm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U yes</td>
<td></td>
<td>u confirm-answer-yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d transfer-to</td>
<td></td>
<td>d transfer-from</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d transfer-amount</td>
<td></td>
<td>a ask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S query Amount</td>
<td></td>
<td>u ask-self</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U 75</td>
<td></td>
<td>u other-answer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a confirm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U no</td>
<td></td>
<td>u confirm-answer-no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d transfer-amount</td>
<td></td>
<td>a ask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S query Amount</td>
<td></td>
<td>u ask-self</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U 50</td>
<td></td>
<td>u other-answer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a confirm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S confirm Amount 50</td>
<td></td>
<td>u confirm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U yes</td>
<td></td>
<td>u confirm-answer-yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 3: Used-Car Application

For the used-car application from the earlier homework, implement it using the Information State approach. For this, you will not change ISEngine.tcl. You will just make up a new version is ISAgent.tcl.

Just get the rudiments of it to work, similar to how the base code of the banking application works (system asks each slot in order, and user responds). In this version though, do not use an Agenda. Instead, have an action selection rule for each parameter, and have that rule make sure that nothing is already on the QUD, and that the parameter is not known yet. Note that the order of these action selection rules will determine the order in which the parameters are asked.²

- Allow the user to answer ‘DontCare’ if it does not care about the value. You should set the value of the slot to ‘DontCare’ so that you do not ask the user for this slot value again.
- Add code so that it stops asking questions when there is less than 5 cars, and instead summarizes what cars are left. You can use the following code to summarize what cars are left.

    proc SummarizeCarInfo {cars} {
        set temp ""
        foreach car $cars {
            set stats [set ::[set car]]
            set newstats {}
            foreach {s f} $stats {
                # lets just include the parameters that the user
                # has not specified.
                if {::isA($s) == "" || ::isA($s) == "DontCare"} {
                    lappend newstats $s $f
                }
            }
            lappend temp "$car: $newstats"
        }
        return $temp
    }

- After the cars have been summarized, you should end the conversation with the bye action. One way to do this is to have the summary action rule set an IS variable that indicates that a summary was done, and then have all of the other action rules check this in their preconditions.³
- If the response to a question results in no cars, remove that value and warn the user, and let the user pick a different value for that question (which might be ‘DontCare’).

Writeup 3.6 Hand in your Agent code, along with two dialogues (a transcript of what was said and the update rules applied) that demonstrate what your application can do. Make sure you explain in a paragraph or two how your rules handle the 4 new behaviors listed above.

²The reason why we are not using an agenda and deliberation rules is to make your system definition easier to use in the upcoming homework assignments, in which we apply Reinforcement Learning to choose the system’s action.
³This is stepping a bit beyond what an action selection rule should be doing, as action rules should only set NextMove and HaveTurn, and should not be setting other variables.