Lambda Expressions

Question 1: Lambda Expressions

In homework 3, you added parallel semantic rule interpretation to a bottom-up parser. In this homework, you will be adding interpretation of lambda expressions, so that lambda expressions can be used for expressing the semantics of intermediate constituents. Listed below is a set of syntactic and semantic rules.

We can use this grammar to parse and interpret sentences like "transfer 10 from peter ’s checking to paul ’s savings." Note that we are explicitly separately the possessive “’s” from its noun. This is because the “’s” has a distinct semantic meaning, and thus we can use compositional semantics to build the meaning of phrases like “peter’s.” By separating the possessive markers in the input, we skip adding in a tokenizer. (Note that English tokenization is much easier than languages like Chinese, which do not use white spaces at all between words.)

Using the above grammar with your parser, you should get the following semantics:

do(transfer(10,X,Y),and(apply(lambda(X,and(apply(lambda(X,owner(X,peter)),X),apply(lambda(X,type(X,checking)),X))),X),apply(lambda(X,and(apply(lambda(X,owner(X,paul)),X),apply(lambda(X,type(X,savings)),X))),Y)))

Writeup 1.1. Marks 1 Verify that you are indeed getting the above. If not, you probably still have a bug in your hw3 code. Hand in a copy of your parse code and its semantic interpretation of “transfer 10 from peter ’s checking to paul ’s saving”.

Question 2: Lambda Expressions in Tcl format

To make the code that processes the lambda expressions simpler, we will change the style that we use for writing the semantics. Rather than put round brackets after the predicate name and at the end of the arguments, we will put Tcl curly brackets before the predicate name and at the end of the arguments. Also, to simplify substitution of the lambda variables with fresh copies, we will put angle brackets around them. Now, transform the semantics for each grammar rule accordingly. The output from parsing the sentence should now be as follows.

do(transfer(10,X,Y),and(apply(lambda(X,and(apply(lambda(X,owner(X,peter)),X),apply(lambda(X,type(X,checking)),X))),X),apply(lambda(X,and(apply(lambda(X,owner(X,paul)),X),apply(lambda(X,type(X,savings)),X))),Y)))
do {transfer 10 X Y} {and {apply {lambda <X> {and {apply {lambda <X> {owner <X> peter}} <X>}} {apply {lambda <X> {type <X> checking}} <X>}}} X} {apply {lambda <X> {and {apply {lambda <X> {owner <X> paul}} <X>}} {apply {lambda <X> {type <X> savings}} <X>}}} Y}

Note the following:

• The X and the Y in the transfer command should not have angled brackets around them, as they are part of the semantic representation language, which might have different conventions for variable names.

• The apply should have two arguments, the lambda expression and the value. The entire lambda expression should be inside of list brackets. Check this carefully now. If you are missing them, and you have stuff like {apply lambda <X> {type <X> checking}}, it is because of the regsub routine. It will allow you to substitute a list of things for the single item. But the list structure is not preserved. Try out the following.

regsub <0> {apply <0> <X>} {lambda <Y> {type <Y> checking}} sem

Note how it is flattening out the list structure. All that you have to do is enclose your semantics (in your rule definition) in an extra set of curly brackets. We need to do this because we are co-opting Tcl's lists to hold our predicate argument structures.

Make sure that you are getting EXACTLY the above. Do not proceed until you do.

Writeup 2.2. Marks 2

Hand in your grammar rules with the revised semantics and a copy of the resulting semantic interpretation of “transfer 10 from peter's checking to paul's saving”.

Question 3: Fresh Variables

One problem we have is that all of our rules use the same lambda expression variable, namely X. We need to make sure that when we apply a semantic rule, we rewrite any variables with a fresh variable name.

The following code should be added to the end of addrule, right before ::numrules is incremented. Make sure you understand what the code is doing.

# now find the lambda variables in the semantics
set lambdavars {}
set rest $newSem
while {$rest != {}} {
    set top [lindex $rest 0]
    set rest [lrange $rest 1 end]
    if {([llength $top] > 1 || [string index $top 0] == "\{") {
        set rest [concat $rest $top]
    } elseif {$top == "lambda"} {
        lappend lambdavars [lindex $rest 0]
        set rest [lrange $rest 1 end]
    }
}
if {$lambdavars != {}} {
    puts " Lambda variables are: $lambdavars"
}
set ::rulelambdavars($::numrules) $lambdavars

Now, in your bottom-up parser, before you use the semantics from a grammar rules, take a copy of the semantics for the grammar rule and rewrite the lambda variables in it with fresh variables. To get a fresh variable, call the following procedure.
proc gensym {} {
    if {!info exists ::gensymindex} {
        set ::gensymindex -1
    }
    incr ::gensymindex
    return "<$_::gensymindex>"
}

Your output should something like this:
do {transfer 10 X Y} {and {apply {lambda <2> {and {apply {lambda <1> {owner <1> peter}}} <2>}} {apply {lambda <3> {type <3> checking}}} <2>}}} X {apply {lambda <5> {and {apply {lambda <4> {owner <4> paul}}} <5>}} {apply {lambda <6> {type <6> savings}}} <5>}}} Y}

Note that the specific names of your variables will not be the same, for example, you might have <256> instead of <2>.

Hint: make sure you use the -all option on regsub.

Now that you are putting in fresh variables, you might notice that the number of interpretations you are searching over has dramatically increased. This is because two semantic interpretations might really be the same, but might use different variable names in the scope of the lambda expressions. The domain grammars that we are considering are simple enough that there will only be a single semantic interpretation for any syntactic form. So you can change the pruning so that you prune a parse-semantics if the syntax has already been seen.

Writeup 3.3. Marks 2
Hand in your revised parse procedure, and highlight the lines of code that you changed or added. Also, hand in your semantic interpretation of “transfer 10 from peter ’s checking to paul ’s saving”.

Question 4: Simplifying Apply

In our previous output, we have a lot of applies that can be simplified down. Below, is what you should be able to simplify your output to.
do {transfer 10 X Y} {and {and {owner X peter} {type X checking}}} {and {owner Y paul} {type Y savings}}}

Write a procedure called simplify that will take an expression, and simplify it. This will be difficult. The easiest way to do this is as a recursive routine, starting at the bottom, and working up. To get you started, you can use the following code.

proc simplify {expr} {
    # if you see something of form {apply {lambda X Expr} Var}, simplify it
    # base case: just a token
    if {([llength $expr] == 1 && [string index $expr 0] != "}") {
        return $expr
    }

    # weird case: a list embedded inside of a list, like {{a b}}
    if {([llength $expr] == 1) {
        set newexpr [string range $expr 1 [expr [string length $expr] - 2]]
        set newexpr [simplify $newexpr]
        set newexpr "$newexpr"
        return $newexpr
    }
set newexpr [lindex $expr 0]
foreach i [lrange $expr 1 end] {
    lappend newexpr [simplify $i]
}
set expr $newexpr

# at this point, we have simplified everything below
# so check if we have an ‘apply lambdaexpr value’
# where lambdaexpr is of form ‘lambda var expr’
# if so, substitute var in expr for value, and output the new expr

# YOUR CODE HERE
}

Execute this routine after each time you create the semantics for a constituent. This way, when you run your program, you will see the simplified semantics for each edge in your chart.

**Writeup 4.4. Marks 3**  Hand in the rest of your code for the simplify routine.

**Question 5: Your Own Semantics**

For the home automation world, we want to change it so that the room is optional. For instance, a person can say “turn on light in the kitchen” or turn off music.” Below is a simple grammar.

```
addrule S    {turn State DeviceDesc}
addrule State on
addrule State off
addrule DeviceDesc Device
addrule Device heat
addrule Device music
addrule Device light
addrule DeviceDesc {DeviceDesc in the Room}
addrule Room kitchen
addrule Room {living room}
```

For the first example sentence, we want the semantics to be:
do {turn on X} {and {type X light} {inroom X kitchen}}}

For the second example, we want the semantics to be: do {turn off X} {type X music}

**Writeup 5.5. Marks 2**  Add the semantics for each rule.

**Running ASR and TTS from a Tcl Script**

Download the tcl code class04form.tcl. The tcl code sets up the speech recognizer, tts engine, a grammar for the speech recognizer, and a phonetic dictionary. It also has the definition of a form (in the style of class04) for the home automation task at the beginning of the file. The main loop is called Converse. It calls DoPrompt and GetResponse. Converse sets what slot to work on, and DoPrompt calls the TTS engine to say that prompt, and GetResponse calls the speech recognizer to get the reponse from the user, using the grammar appropriate for the current slot. As the recognizer can be called with a different grammar each time, the code in GetResponse is a bit complicated as it needs to initialize a new speech recognizer each time, and destroy it after its use.
Question 6: Familiarizing Yourself with the Code

There is a lot of tricky code in here, which was hidden from you in RAD. Look over the code to try to understand it. Look over the code to try to understand it.

Writeup 6.6. Marks 1  Run the program. It will ask you for a device type, such as heat, light, or stereo. You can exit form the program by pressing the kill window button in the upper right corner. When you press this, what code is run?

Writeup 6.7. Marks 1  The procedure GetResponse calls the speech recognizer. It calls the recognizer with the grammar and a second piece of knowledge. Explain what the second piece of knowledge is in a sentence. If you are not sure, use puts to output the value of that variable so that you can understand what it is.

Writeup 6.8. Marks 1  How is the procedure Converse called? What tcl/tk code is invoking it, and when is it being invoked?

Writeup 6.9. Marks 1  The interaction between GetResponse and Accumulate is complicated. Briefly explain the purpose of the “vwait ::RecordingDone” command in Converse.

Writeup 6.10. Marks 2  What did you change so that system would not wait as long after the user spoke? What value worked best? What adverse effects happen at higher and lower values?

Form-Based Dialogue Management

Question 7: Basic Functionality

Add functionality to Converse so that it will prompt for each slot in the form. It should check what values have not been filled in yet using the global variable ::values (which is an associative array, as we defined it in the class notes). When there are no other values to be filled in, the system should not prompt for any devices. Instead, it should exit the Prompt-and-Response loop, and be ready for the user to press the ‘start to speak’ button again. You also need to change GetResponse so that it saves the value that the user said. This will give you the basic functionality of a form-filling dialogue manager.

Writeup 7.11. Marks 3  Hand-in your code for Converse and GetResponse.

Question 8: Repeat, Start-over, Quit

Add functionality to allows the user to say ‘quit’ at any point. Note that the word ‘quit’ should be automatically added to the grammar for each slot, rather than forcing the dialogue designer to add it to the grammar of each slot.

Add functionality so that the user can say ‘repeat’ at any point.

Add functionality so that the user can say ‘start over’ at any point.

In adding this functionality, only change the code in GetResponse.

Writeup 8.12. Marks 4  Hand-in the code for GetResponse and highlight the code that you changed.
Question 9: Simple Summary

Add functionality to Converse so that after the user fills in all of the slots, the system will say all of the values of the slots to the user. Make sure that this is domain dependent. The code that you add should not have the actual slot names hardcoded in.

Writeup 9.13 2 Hand in your code for Converse, and highlight which lines you changed.