Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **7** Due: **27/Oct/98** 6.034 Item # **12**

PLEASE WRITE YOUR TA'S NAME ON YOUR PAPER.

Goals

- To learn about program-writing programs, in particular, a program that writes semantic-transition-tree programs.
- To learn about relational database operations.
- To observe how easy/difficult it is to extend semantic-transition-tree grammars while using the same relational database.
- To observe how easy/difficult it is to reorient a semantic-transition-tree grammars from a relational database to another database.
- To think about how easy/difficult it is to reorient the linguistic knowledge represented in a semantic-transition-tree grammar for use in other applications.

Background on Transition Tree Grammars

You recall, from Chapter 29, that semantic transition trees are often described using graph diagrams, but the same semantic transition trees can be expressed in the form of definitions that use a parenthesized syntax instead of diagrams.

In the simplest definitions, only symbols appear. The following, for example, defines a procedure that succeeds only if the words *I like apples* appear:

```
(define sample1
  (make-tree '(i like apples)))
> (sample1 '(i like apples))
(#t ())
```

```
> (sample1 '(i like apples a lot))
(#t (a lot))
> (sample1 '(i like candy a lot))
#f
```

Note also that when successful, the procedure returns a list consisting of `#t` and the words that remain after eliminating the matching words.

In slightly more complex definitions, branches appear in order to account for distinct word sequence paths. The following example, for instance, defines a procedure that succeeds if either the words *I like apples* or the words *I like pears* appear:

```
(define sample2
  (make-tree '(branch (i like apples)
                      (i like pears))))
> (sample2 '(i like apples))
(#t ())
> (sample2 '(i like pears))
(#t ())
> (sample2 '(i like apples a lot))
(#t (a lot))
> (sample2 '(i like pears a lot))
(#t (a lot))
> (sample2 '(i like candy a lot))
#f
```

You can further arrange for a procedure to return the value of an expression if it succeeds. The following, for example, defines a procedure that returns `'success`, along with a list of remaining words, when it succeeds:

```
(define sample3
  (make-tree '(i like apples rtn 'success)))
> (sample3 '(i like apples))
(success ())
> (sample3 '(i like apples a lot))
(success (a lot))
> (sample3 '(i like candy a lot))
#f
```

Next, you can arrange for a procedure to return the value of an expression if it succeeds, but only if *all* words are matched. The following, for example, defines a procedure that returns `'success`, but only if there are no more words.

```
(define sample4
  (make-tree '(i like apples if-end-rtn 'success)))
> (sample4 '(i like apples))
(success ())
> (sample4 '(i like apples a lot))
#f
> (sample4 '(i like candy a lot))
#f
```

Next, you can arrange to call subroutines that define subtrees. The following, for example, defines a procedure that succeeds if the first two words match and the `fruit` procedure matches the third word.

```
(define sample5
  (make-tree '(i like (fruit))))
(define fruit
  (make-tree '(branch (apples rtn 'branch1) (pears rtn 'branch2))))
> (sample5 '(i like apples))
(#t ())
> (sample5 '(i like pears))
(#t ())
> (sample5 '(i like apples a lot))
(#t (a lot))
> (sample5 '(i like pears a lot))
(#t (a lot))
> (sample5 '(i like candy a lot))
#f
```

Finally, by adding an arrow, `-->`, and a variable name, you can arrange to call subroutines and use the values they produce. The following, for example, defines a procedure that succeeds if the first two words match and the `fruit` procedure matches the third word. But even more usefully, the variable `x` is bound to the result produced by the `fruit` procedure, and the value is used in producing a value for the `sample6` procedure.

```
(define sample6
  (make-tree '(i like (fruit --> x) rtn `(i took ,x))))
(define fruit
  (make-tree '(branch (apples rtn 'branch1) (pears rtn 'branch2))))
> (sample6 '(i like apples))
((i took branch1) ())
> (sample6 '(i like pears))
((i took branch2) ())
> (sample6 '(i like apples a lot))
((i took branch1) (a lot))
> (sample6 '(i like pears a lot))
```

```
((i took branch2) (a lot))
> (sample6 '(i like candy a lot))
#f
```

Generally, however, values are assigned so as to build up and evaluate database calls involving `select` or `project` procedures.

Code

You will make use of the following files, which you can find in the directory `/mit/6.034/ps7/`:

- `stt.scm`: this file contains a reader that translates strings into lists of symbols and an interpreter that translates (relatively) transparent specifications for semantic transition tree grammars into not-particularly-transparent, deeply-nested Scheme programs.
- `grammar.scm`: a basic grammar, in the form of semantic transition trees, that translates questions and commands into calls to relational database procedures.
- `rdb.scm`: this file contains a relational database stub; that is, it contains procedures—such as `db-select` and `db-project`—that mimic those of a relational database system.
- `blocks.scm`: a database describing some blocks.
- `books.scm`: a database describing a book inventory.

Warm Up

Using the system

You should first review a 6.034 blocks-world example. You don't have to hand anything in to prove that you did this, but it will help you warm up for the problems (especially the next one!). Load all files in the same order given above, except for the books database. Then evaluate `(run)` to activate the sentence interpreter, and type the following sentences, surrounded by parentheses, as shown. (You can evaluate `(stop)` to halt the sentence interpreter.)

```
(Show me the blocks)
(List the bricks)
(What are the wedges)
(Count the balls)
(How many bricks are there)
(What are the colors of the bricks)
(List the small red bricks)
(Rank the bricks by color)
```

Once everything is working on a relational database stub such as the one you use in this problem, you could, of course, redirect database queries to a real relational database if you wanted to (but that is *not* required for credit in this problem set!).

Observing the translator

Try printing the results of a few translations, such as suggested by the following examples. Note what `translate-tree` does as it does its job.

```
(pp (translate-tree '(a nice day)))
(pp (translate-tree '(a (word) day if-end-rtn #t)))
(pp (translate-tree '(a (word --> x) day if-end-rtn x)))
(pp (translate-tree '(branch (nice) (wonderful))))
(pp (translate-tree '(branch (nice rtn 'nice)  (wonderful rtn 'wonderful))))
```

Problem 1

Next, while still using the blocks database, we explore the power of the current grammar and interpreter.

Part A

Try the following query:

```
(Count the the blocks)
```

Describe the output of the system and, referring to `grammar.scm` and the procedure `object-description`, explain why this behavior occurs.

Part B

Ben Bitdiddle suggests the following modifications to `main` and `object-description` to change the behavior described above:

```
;;;; MAIN TREE
(define main
  (make-tree
    '(branch
      ;; COUNT THE BLOCKS
      (count (determiner) (object-description --> o)
if-end-rtn (db-call `(db-report-count (db-count-items ,o))))
      ;; HOW MANY BLOCKS ARE THERE
...
etc.

(define object-description
  ;; Consumes, for example, A BIG BLACK HEAVY BLOCK
  (make-tree
    '(branch ((object-description --> o) rtn o)
     ((attribute-value-list --> a-v-l)
      (object-description --> o)
      rtn `(db-select ,o '(with ,@a-v-l)))
     ((object --> o) rtn o))))
```

Alyssa P. Hacker tells Ben that this will only make matters worse when the system tries to parse `count the the blocks`. Briefly, explain why Alyssa is correct.

Part C

Now try the following query:

```
(Show me all the blocks)
```

Briefly describe what happens and describe what property of the control structure of the transition tree interpreter produces this particular behavior. Then explicitly show how to modify one procedure in `grammar.scm` to correct the problem with the above query. (It is not necessary to modify the `main` procedure.)

Problem 2

You next turn to the electronic commerce business. Specifically, you decide to sell books on the net. You'd like to include a natural language search engine so that customers can easily find their favorite books and authors, and so forth. Accordingly, you want to see how well your existing natural language system will work on a new database. You load a small example books database, `books.scm`, and continue.

Part A

Adapt the translator, the grammar, or the database procedures such that the language interface handles the book database. Show and characterize the changes you have made, indicating which files you have altered and the nature of your changes (e.g. substituting names for names, modifying a program, supplementing a grammar). Be sure you can handle the following questions and show us the results you obtain from these questions:

```
(Show me the books)
(List the paperback books)
(Rank the computer science books by format)
(What are the Addison-Wesley books)
(Count the fiction books)
(What are the categories of Winstons books)
```

Part B

Of course, a typical book shopper would ask other questions than these. Show how to modify `grammar.scm` to handle the following kinds of questions, demonstrating them on the books database.

```
(Count the sci-fi books)
(What are the topics of the Addison-Wesley books)
(List the science books by Winston)
(What are the books about science)
(What are the fiction books published by Ace)
```

Note: it is much better to try to modify something other than the `main` procedure.

Problem 3

You consider that eventually you'd like to "personalize" the interaction between the human shopper and the database query system.

Considering the previous part, and making use of the semantic transition-tree machinery to return answers in response to queries and statements, alter `main` and `grammar.scm` to implement a *short* Eliza-like program that recognizes a few patterns that would be useful in mimicking the conversational behavior of an on-line book-selling search agent.

Show us your changes and a sample dialog. Discuss in a few sentences the differences between *recognizing* and *generating* English sentences.