Purpose

Project 3 focuses on the use of data structures such as lists, trees, and hash tables. You will also further develop and demonstrate your ability to write clear, intelligible, well-documented procedures, as well as test cases for your procedures.

Overview

Scenario

Background: the character in this scenario, Nim Chimpsky, is real. You can find out more about him at

http://en.wikipedia.org/wiki/Nim_Chimpsky

In brief, Nim was a chimpanzee who was the subject of a research experiment at Columbia University. The aim of the experiment was to see if chimpanzees could acquire sign language, and in particular whether they could in some sense acquire the “grammar” or “syntax” of a human language. The results were mixed—Nim did not, apparently, manage to learn to construct even simple sentences in sign language—see the Wikipedia article for more details.

Nim Chimpsky is depressed. His life’s work has been to learn sign language, but things haven’t turned out as he thought they would. Initially, progress was quick: he learned around 125 signs (for crucial things such as “Nim”, “eats”, “oranges” and “tickles”) in just a few years. He could even construct sequences of signs which he thought were quite profound. Recently, he thought he had made his crowning achievement, when he signed his longest sentence to date:

“Give orange me give eat orange me eat orange give me eat orange give me you”

Much to his disappointment, however, linguists—particularly the linguists at MIT—were not impressed. They claimed that Nim couldn’t acquire “syntax”, whatever that might mean, and thus couldn’t construct grammatical sentences.

Nim decides to find out more about what the linguists are looking for. He starts reading Syntactic Structures, an early book by Noam Chomsky, which was highly influential in linguistics. In it Chomsky describes formal systems that can be used to assign structures to sentences in a language, and to distinguish which sentences are vs. aren’t grammatical in a language. Nim realises that if he could implement one of these systems in Scheme, he would in effect have a grammar checker—i.e., a program that would tell him whether or not a
particular sentence he signed was grammatical. That way he could filter out the ungrammatical sentences that he produced, leaving just the grammatical ones, thereby impressing the linguists after all.

Your first task in this project will be to produce a grammar checker—or “parser”—for Nim. Later we’ll see that this parser will produce structures for English sentences that are useful in another application, translation between languages.

**Context-Free Grammars**

One of the big ideas in *Syntactic Structures* was to use context-free grammars in the study of language. A context-free grammar is a formal system that specifies which sentences are or aren’t grammatical in a language.

In this project, we’ll define a context-free grammar (CFG) to consist of the following parts:

- A *vocabulary*, or a set of *words* in the language. For example, a simple CFG might have the vocabulary \{Nim, Noam, likes, knows, respects, fears\}.
- A set of *non-terminal* symbols. For example, a simple CFG might have the non-terminals \(S, \ NP, \ VP, \ V\). We’ll soon see what role the non-terminals play.
- A *lexicon*. The lexicon is a set of *pairs*, where each pair consists of a non-terminal and a word. For example, we might have the following lexicon:
  - NP, Nim
  - NP, Noam
  - V, likes
  - V, knows
  - V, respects
  - V, fears

  From a linguistic standpoint, this particular lexicon says that the words *Nim* and *Noam* have the category NP, which we will use to stand for “noun”. The words *likes, knows, respects* and *fears* all have the category V, which stands for “verb”.
- A set of *rules*. Each rule is of the form
  \( X \rightarrow Y Z \)

  where \(X, \ Y, \) and \(Z\) are all non-terminal symbols. For example, our simple CFG might have the rules

  \[
  S \rightarrow NP \ VP \\
  VP \rightarrow V \ NP
  \]

  The intuition behind the *rules* in the grammar is as follows. The symbols \(S\) and \(VP\) stand for “sentence” and “verb-phrase” respectively. The first rule,
$S \rightarrow NP \ VP$

states that a sentence can be formed by a noun ($NP$) followed by a verb-phrase ($VP$). The second rule,

$VP \rightarrow V \ NP$

states that a verb-phrase can be formed by a verb ($V$) followed by a noun ($NP$).

Given a grammar, we can draw a parse tree for any sentence that is grammatical under the grammar. For example, the parse tree for *Nim likes Noam* is the following:

```
S
   NP   VP
     Nim   V   NP
        likes   Noam
```

At the top of the parse tree we see the symbol $S$, which means that *Nim likes Noam* is a sentence according to the grammar. The sentence $S$ breaks down into two parts, an $NP$ followed by a $VP$. Note that this way of breaking down a sentence is allowed because the grammar has the rule $S \rightarrow NP \ VP$. The $VP$ then breaks down into a $V$ followed by an $NP$; similarly, this is allowed because the rule $VP \rightarrow V \ NP$ is in the grammar. Finally, we are allowed to associate $NP$ with *Nim*, $V$ with *likes*, and $NP$ with *Noam*, because all of these entries are allowed in the lexicon in the grammar.

To summarize, a parse tree for a sentence under a grammar has the following properties:

- The label at the top of the tree is $S$ (for sentence).
- Each time a node in the tree expands to two child nodes, for example in

```
S
   NP   VP
     Nim   V   NP
        likes   Noam
```

the split is allowed by a rule in the grammar. For example, in this case the rule $S \rightarrow NP \ VP$ allows this split.

- At the leaves of the tree we see words paired with non-terminals that are allowed by the lexicon. For example, at the bottom of the parse tree shown above we see the three entries allowed by the lexicon:

```
NP  V  NP
|  |  |
Nim likes Noam
```

A crucial point is the following: *given a CFG, a sentence is only grammatical if it has at least one parse tree.*

In particular, note that some sentences do not have a parse tree under the grammar shown above, and hence are not grammatical sentences, at least according to our simple grammar. For example, it’s impossible to find a well-formed parse tree for the string *Nim likes*. On the other hand, it is possible to find parses for several other sentences, such as *Nim likes Nim*, *Noam respects Nim*, *Nim fears Noam*, and so on.
Building a Parser

A major focus of this project will be building a parser. A parser is a function (parse sentence grammar) which takes a sentence and a context-free grammar as its two arguments. It returns #f if there is no parse for the sentence under the grammar. Otherwise it returns a parse tree for the sentence. Parse trees in this project will be represented as box-and-pointer structures: for example, the tree

```
S
  NP   VP
    Nim  V  NP
          likes Noam
```

would be represented as the structure

```
(S (NP Nim) (VP (V likes) (NP Noam)))
```

(Note that the words and non-terminal symbols in the tree are all symbols in Scheme; for example, the above list structure could be created by evaluating `'(S (NP Nim) (VP (V likes) (NP Noam)))`.)

Problem 1: Building a Lexicon

In this problem we’ll make use of hash tables to represent a lexicon for a context-free grammar. You’ll see hash tables in lecture, here is a brief reminder. A hash table is a data structure to associating a “key” with a “value”; usually it is a very efficient data structure. The underlying idea is that associated with a hash table is a hash function. When you apply the hash function to the key, you get a value that tells you exactly where in the data structure to go. Thus, you can either “put” things into a hash table or “get” them out of a hash table very quickly.

To make a hash-table, we’ve provided a function make-hash. For example, you can use

```
(define new-table (make-hash))
```

to associate the name new-table with a new hash table. You can then use the function (hash-put! table key value) to add an entry to the hash table. The argument table specifies the hash table to which we’d like to add an element; the argument key is a list of symbols, or just a symbol, that represents the “key” we’d like to add to the hash table; the argument value is a value we’d like to associate with the key. For example, we could add three entries to our hash table as follows:

```
(hash-put! new-table '(X) 4)
(hash-put! new-table '(Y) (list 3 4))
(hash-put! new-table 'X 5)
```
Finally, the function \( (\text{hash-get table key}) \) retrieves the value associated with a particular key. If the key is not found in the table, the symbol \texttt{none} is returned. For example:

\[
\begin{align*}
(\text{hash-get new-table '(X)}) & \;\rightarrow\; 4 \\
(\text{hash-get new-table '(Y)}) & \;\rightarrow\; (3 \ 4) \\
(\text{hash-get new-table 'X}) & \;\rightarrow\; 5 \\
(\text{hash-get new-table '(Z)}) & \;\rightarrow\; \texttt{none}
\end{align*}
\]

Note that we use \( ;\rightarrow \) to show the return value of the different procedure calls. In the first three cases we retrieve the values previously added to the table. In the final case, the symbol \texttt{none} is returned, because the key is not found in the table.

In our context-free grammar we will represent a lexicon as a list of the following form:

\[
\begin{align*}
(\text{define simple-lexicon ' ( (NP Nim)
(NP Noam)
(V likes)
(V knows)
(V reveres)
(V fears)))
\end{align*}
\]

We’d like to design a function \( (\text{lexicon->hash lexicon}) \) that takes a lexicon as input, and returns a hash table as its value. The hash table should contain each entry in the lexicon as a key, paired with the value \#t. For example, we should be able to do the following:

\[
\begin{align*}
(\text{define lex-hash (lexicon->hash simple-lexicon)}) \\
(\text{hash-get lex-hash '(NP Nim)}) & \;\rightarrow\; \#t \\
(\text{hash-get lex-hash '(NP Noam)}) & \;\rightarrow\; \#t \\
(\text{hash-get lex-hash '(V knows)}) & \;\rightarrow\; \#t \\
(\text{hash-get lex-hash '(V Nim)}) & \;\rightarrow\; \texttt{none} \\
(\text{hash-get lex-hash '(X Y)}) & \;\rightarrow\; \texttt{none}
\end{align*}
\]

Note that in the last two cases \texttt{none} is returned, because neither \( '(V Nim) \) or \( '(X Y) \) are seen in \texttt{simple-lexicon}. We’ve already defined the following code in the attached code for the project:

\[
\begin{align*}
(\text{define (lexicon->hash lexicon)}
\let ((m (make-hash)))
(\text{lexicon->hash-helper m lexicon)))
\end{align*}
\]

**Question 1**: Define the code for \( \text{lexicon->hash-helper} \) so that \( \text{lexicon->hash} \) has the desired behavior. You should test your code with the \texttt{lexicon} supplied in the code associated with this project (look for \( (\text{define lexicon ...}) \) in the code).
Problem 2: Representing Rules in the Grammar

The rules in a context-free grammar will be represented as a list of the following form (this particular set of rules is given in the code for the project):

```
(define rules '(
    (S NP VP)
    (VP V NP)
    (VP V3 ADJP)
    (V3 V3 NOT)
    (VP ADVP VP)
    (VP VP PP)
    (PP P NP)
    (SBAR COMP S)
    (VP V2 SBAR)
    (V2 V2 ADVP)
    (V2 V2 PP)
    (VP V3 PP)
    (VP VP SBAR2)
    (SBAR2 COMP2 S)
    (NP D N)
))
```

Each rule in the grammar is a list of three elements, for example \((S \ NP \ VP)\). In this case the list \((S \ NP \ VP)\) represents the rule \(S \rightarrow NP \ VP\); the other rules are represented in a similar way.

Again, we'd like to define a function \((rules->hash \ rules)\) that takes a list of rules in this format, and returns a hash table. The hash table associates each symbol in the grammar—for example, \(S\), \(VP\), or \(V3\)—with a list of rules that have that symbol as the left-hand-side of the rule. For example, we should see the following behavior:

```
(define rules-hash (rules->hash rules))
(hash-get rules-hash 'S ) ;-> ((S NP VP))
(hash-get rules-hash 'VP ) ;-> ((VP V NP)
  (VP V3 ADJP)
  (VP ADVP VP)
  (VP VP PP)
  (VP V2 SBAR)
  (VP V3 PP)
  (VP VP SBAR2))
```

**Question 2**: Write a function \((get-all-lhs \ rules)\) which returns a list of all non-terminals that are seen on the left-hand side of a rule. For example, we should have

```
(get-all-lhs rules) ;-> (S VP V3 PP SBAR V2 SBAR2 NP)
```
Note that each non-terminal is seen exactly once in the return value. Your function should use map, filter, or foldr. In addition, you may find useful the function unique, which is provided in the attached code for the project.

**Question 3**: Next, write a function (get-rules lhs rules). The first argument, lhs, is a non-terminal symbol such as S or VP. The second argument, rules, is a list of rules, as in the example above. The function should return a list of rules associated with the symbol lhs. For example:

```lisp
(get-rules 'S rules) ;-> ((S NP VP))
(get-rules 'VP rules) ;-> ((VP V NP)
  (VP V3 ADJP)
  (VP ADVP VP)
  (VP VP PP)
  (VP V2 SBAR)
  (VP V3 PP)
  (VP VP SBAR2))
```

**Note**: your function should make use of map, filter or foldr

**Question 4**: Finally, write the function (rules->hash rules), which should make use of the functions get-all-lhs and get-rules.

Note: The code supplied with the project includes definitions of make-grammar, grammar-rules, and grammar-lexicon. These functions make use of lexicon->hash and rules->hash. Take a look at the code to see how these functions work. Here’s one example of how to use these functions:

```lisp
(define g (make-grammar rules lexicon))
(grammarm-rules g 'S) ;-> ((S NP VP))
(grammarm-lexicon g 'NP 'Nim) ;-> #t
(grammarm-lexicon g 'V 'Nim) ;-> #f
```

**Problem 3: Building a Parser**

We’ll now build a parser. As a reminder, a parser will try to find a tree that identifies a non-terminal with each non-leaf of the tree using the rules of the grammar. In other words, the leaves of the tree should be elements of the lexicon, appropriately tagged with a non-terminal associated with the word, and each intermediate node represents a legal rule of the grammar.

A central function that we’ll need to define is (find-tree nt i j s g). The arguments of find-tree are as follows. nt is a non-terminal symbol, for example S or VP. The argument s is a sentence in list form, for example (Nim likes Noam). Finally, g is a grammar, as constructed by make-grammar. The arguments i and j are both integers such that 1 ≤ i ≤ j ≤ n, where n is the number of words in the sentence s.

The function find-tree works as follows. If a parse tree can be found that spans words i to j inclusive in the sentence s, with root node nt, then it should return such a parse tree. Otherwise, it should return #f.
As one example, if we evaluate `(find-tree 'S 1 3 '(Nim likes Noam) g)` then this should return the value

(S (NP Nim) (VP (V likes) (NP Noam)))

If we evaluate `(find-tree 'VP 2 3 '(Nim likes Noam) g)`, we should get the value

(VP (V likes) (NP Noam))

If we evaluate `(find-tree 'V 2 2 '(Nim likes Noam) g)` then this should return the value

(V likes)

(note that in this case we have \(i = j = 2\), and we therefore search to see if the entry \((V \ \text{likes})\) is in the lexicon, and then construct a parse tree for the single word \(\text{likes}\)). Finally, if we evaluate `(find-tree 'VP 1 3 s g)` the value \#f should be returned, as there is no way of finding a parse with root node \(\text{VP}\) spanning words 1 to 3 in the sentence.

The function `find-tree` is defined as follows:

```scheme
(define (find-tree nt i j s g)
  (if (= i j)
      (leaf nt i s g)
      (find-tree-helper i j i s (grammar-rules g nt) g)))
```

We have the following definition for `(leaf nt i s g)`:  

```scheme
(define (leaf nt i s g)
  (let ((word (list-ref s (- i 1))))
    (if (grammar-lexicon g nt word)
        (list nt word)
        #f)))
```

Note that `find-tree` makes a call to `leaf` in the case that \(i\) and \(j\) take the same value. In this case `leaf` returns \#f if the non-terminal `nt` cannot be paired with the \(i\)'th word in the sentence, and returns a small parse tree `(list nt word)` otherwise.
The case when \( i \neq j \) is more complicated. The function \( \text{find-tree-helper} \) is currently defined as follows:

\[
\begin{align*}
\text{(define (find-tree-helper } i \ j \ k \ s \ \text{rules } g) \\
\quad \text{(cond } ((\text{null? rules}) \ #f) \ \\
\quad \quad (\text{=} k \ j) \ (\text{find-tree-helper } i \ j \ i \ s \ \text{(cdr rules) } g)) \\
\quad \text{(else (or (find-tree-prod (car rules) } i \ j \ k \ s \ g) \\
\quad \quad \quad \quad \text{(find-tree-helper } i \ j \ (+ \ k \ 1) \ s \ \text{rules } g))))
\end{align*}
\]

\[
\begin{align*}
\text{(define (find-tree-prod } \text{rule } i \ j \ k \ s \ g) \\
\quad \text{(display (list } \text{rule } i \ j \ k) \\
\quad \quad \text{(newline)} \\
\quad \quad \#f)
\end{align*}
\]

Here is the basic idea. If we try all the rules without success, we give up on parsing the sentence. Otherwise we try to split the sentence into two portions (based on \( k \)). If we can’t apply the first rule for any split of the sentence, then we try the next rule, starting with all of the words in one part of the split. Otherwise we see if the current rule works for the current split, or we try again with all the rules, and a shift in the split point.

Note that the code for \( \text{find-tree-prod} \) is a place-holder: the parser is not yet complete. Soon we’ll complete this code.

**Question 5** Try evaluating \( \text{(find-tree } \text{'S} \ 1 \ 3 \ \text{'(Nim likes Noam) } g) \), \( \text{(find-tree } \text{'S} \ 1 \ 4 \ \text{'(Nim likes the oranges) } g) \), and \( \text{(find-tree } \text{'VP} \ 1 \ 3 \ \text{'(Nim likes Noam) } g) \). Explain the behavior in each case.

We’ll now complete the code for \( \text{(find-tree-prod } \text{rule } i \ j \ k \ s \ g) \). To illustrate how this code should work, consider evaluating

\[
\text{(find-tree-prod } \text{'(S} \ \text{NP} \ \text{VP} \ i \ j \ k \ s \ g)
\]

where \( s \) is a sentence, and \( i, j, k \) are integers. The function should take the following steps:

- It should call \( \text{(find-tree } \text{'NP} \ i \ k \ s \ g) \) to see if it is possible to have an NP spanning words \( i \) to \( k \) in the sentence.
- It should call \( \text{(find-tree } \text{'VP} \ (+ \ k \ 1) \ j \ s \ g) \) to see if it is possible to have a VP spanning words \( k + 1 \) to \( j \) in the sentence.
- If both of the previous evaluations return parse trees, it should return a value that is equivalent to

\[
\text{(list } \text{'S} \ (\text{find-tree } \text{'NP} \ i \ k \ s \ g) \\
\quad \quad \text{(find-tree } \text{'VP} \ (+ \ k \ 1) \ j \ s \ g))
\]

On the other hand, if either of the calls to \( \text{find-tree} \) return \#f, the function should return \#f.

The intuition behind \( \text{(find-tree-prod } \text{rule } i \ j \ k \ s \ g) \) is as follows. The function searches for a parse tree that spans words \( i \) to \( j \) in the sentence. It does this by using \text{rule} at the top of the tree. In
general, rule is of the form \((X Y Z)\) where \(X\), \(Y\) and \(Z\) are non-terminals. The function tests whether it’s possible to find a parse tree spanning words \(i\) to \(k\) with \(Y\) at the root. It also tests whether it’s possible to find a parse tree spanning words \(k + 1\) to \(j\) with \(Z\) at the root. If both parse trees can be found, it “glues” them together to form a parse tree for words \(i\) to \(j\) with \(X\) at the root.

Note that \texttt{find-tree-prod} will make two calls to \texttt{find-tree}. We now see that the parser is a recursive procedure: evaluation of \texttt{find-tree} leads to evaluations of \texttt{find-tree-prod}, which in turn makes calls to \texttt{find-tree}. The base case for the recursion is the case in \texttt{find-tree} where \((= i j)\) is true.

**Question 6** Implement the code for \texttt{find-tree-prod}. The code for \texttt{find-tree} will then be complete. Make sure to try the functions \texttt{find-tree-prod} and \texttt{find-tree} on a few examples, for example try

\[
\text{try (define grammar (make-grammar rules lexicon))}
\text{(find-tree-prod '(S NP VP) 1 3 1 '(Nim likes Noam) grammar)}
\text{(find-tree 'S 1 3 '(Nim likes Noam) grammar)}
\]

**Problem 4: Making the Parser Efficient**

It turns out that our implementation of \texttt{find-tree} in problem 3 is rather inefficient. In fact, if the input sentence is of length \(n\), then the parser can take exponential time in \(n\) to parse a sentence. Fortunately we’ll see in this part of the project that the problem is fairly easy to fix.

To make the parser efficient, we’ll use a technique called memoization or dynamic programming. To illustrate this idea, let’s look at the fibonacci function again. The inefficient version of fibonacci that we saw earlier in the course is as follows:

\[
\begin{align*}
\text{(define (fib n)} \\
\text{  (cond ((=} \text{n 1) 1})} \\
\text{  ((=} \text{n 2) 1}) \\
\text{  (else (+ (fib (- n 1)) (fib (- n 2)))))}}
\end{align*}
\]

Recall that this is inefficient because it repeats a lot of computation. Now consider a new version of fibonacci:

\[
\begin{align*}
\text{(define (fibnew n)} \\
\text{  (fib-h n (make-hash)))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (fib-h n h)} \\
\text{  (cond ((=} \text{n 1) 1})} \\
\text{  ((=} \text{n 2) 1}) \\
\text{  (else (let ((value (hash-get h n)))} \\
\text{    (if (not (eq? value ’none))} \\
\text{      value} \\
\text{    (let ((newvalue (+ (fib-h (- n 1) h) (fib-h (- n 2) h))))} \\
\text{      (hash-put! h n newvalue) \\
\text{      newvalue))))))}}
\end{align*}
\]
The new function, \texttt{fibnew}, is actually very similar to our original definition of \texttt{fib}. The crucial difference is that it uses a hash table to store previously computed values. For \( n > 2 \), the first thing it does is check in the hash table whether this value has been previously stored. If it has been stored (i.e., \((\texttt{not (eq? value } \texttt{none)})\) evaluates to \#t) then it simply returns this value. Otherwise, it calculates \texttt{newvalue} using the usual recursive definition of fibonacci. The function is careful to store this value in the hash table, using \((\texttt{hash-put! h n newvalue})\), before returning \texttt{newvalue}.

Storing previously computed values in a table like this is called \textit{memoization}. The intuition here is that we save ourselves from repeated computation: if \((\texttt{fibnew n h})\) has been called in the past for a particular value of \( n \), then the \( n \)th fibonacci number will be stored in the hash table, and it can be looked up in \( \Theta(1) \) time. In fact, it can be shown that \texttt{fibnew} is now a linear time function, a drastic speed-up over the exponential run time of \texttt{fib}.

\textbf{Question 7} Build a version of the parser that uses memoization. You should fill in the code for \texttt{find-tree2} shown below. \texttt{find-tree2} should use the hash table to store parse trees for \((\texttt{nt}, \texttt{i}, \texttt{j})\) triples.

\begin{verbatim}
(define (parse2 sentence grammar)
  (find-tree2 'S 1 (length s) sentence grammar (make-hash)))

(define (find-tree2 nt i j sentence grammar hash)
  (if (= i j)
      (leaf nt i s g)
      YOUR-CODE-HERE))

Hint 1: You can store useful values in the hash table using

\((\texttt{hash-put! hash (list nt i j) tree})\)

to associate a parse tree \texttt{tree} with the arguments \texttt{nt}, \texttt{i} and \texttt{j}.

Hint 2: You’ll also need to come up with new definitions of \texttt{find-tree-helper} and \texttt{find-tree-prod}, which take a hash table as an additional argument. Here’s a new version of \texttt{find-tree-helper} which you can use:

\begin{verbatim}
(define (find-tree-helper2 i j k s rules g hash)
  (cond ((null? rules) #f)
        ((= k j) (find-tree-helper2 i j i s (cdr rules) g hash))
        (else (or (find-tree-prod2 (car rules) i j k s g hash)
                  (find-tree-helper2 i j (+ k 1) s rules g hash))))
\end{verbatim}

Note that this is very similar to \texttt{find-tree-helper}. The definition of \((\texttt{find-tree-prod2 rule i j k s g hash})\) is similar in that it involves very few changes to your code for \texttt{find-tree-prod}.

Note: The good news is that the new version of \texttt{find-tree2} is not exponential in terms of run-time: in fact, it takes \( \Theta(n^3) \) time to parse a sentence with \( n \) words.
Problem 5: Testing the Parser

The lexicon in the grammar provided in the project has a number of words, *Nim*, *Noam*, *eats*, *likes*, ..., *although*. You should be able to write grammatical sentences which are parsed by the grammar and use any one of these words. For example, the sentences

- Nim likes Noam
- he fears Noam
- he knows Noam
- Nim eats oranges

make use of the words *Nim, likes, Noam, he, fears, knows, eats, oranges*.

**Question 8**  Give a sequence of sentences which can be parsed by the grammar, and which between them make use of all words in the lexicon. For each of your sentences, show the structure produced by the parser, for example *(S (NP Nim) (VP (V likes) (NP Noam)))*. 

Note: you may find useful the description of the different categories in the lexicon, which can be found in the comments in the code. We’ve deliberately left out some details in the explanation though—you should be able to figure out how to use the different words and rules.

The grammar covers a reasonable number of English sentences, but we should warn you that it is very naive from a linguistic standpoint! In the worst case, it will actually give parse trees for some sentences which aren’t grammatical. As one example, you’ll find that the sentence *Nim likes he* gets the parse tree *(S (NP Nim) (VP (V likes) (NP he)))* under the grammar.

**Question 9**  Give at least two other examples of sentences that are not grammatical, but do receive parses from the system. Neither of these sentences should use the word *he*.

Problem 6: Adding Non-Binary Rules

So far our grammar has made use of binary rules, such as \( S \rightarrow NP \ VP \), where the number of non-terminal symbols on the right-hand-side of each rule is exactly two. In some cases, it would be convenient to be able to write non-binary rules, where there are more than two non-terminals on the right-hand-side. As one example, we might want to add a rule

\[ VP \rightarrow V4 \ NP \ NP \]

to the grammar, along with a lexical entry *V4, gives*. This would allow our grammar to parse sentences such as *Nim gives Noam oranges*. As another example, we might add the rule

\[ VP \rightarrow V5 \ NP \ SBAR \]
to the grammar, along with a lexical entry V5, told. This would allow us to parse sentences such as Nim told Noam that he is happy.

Fortunately, there is a simple way to handle non-binary rules in the system. The idea will be to convert any non-binary rule to 2 or more binary rules. As one example, the rule

\[ VP \rightarrow V4 \text{ NP SBAR} \]

should be converted to the two rules

\[ VP \rightarrow V4-NP \text{ SBAR} \]
\[ V4-NP \rightarrow V4 \text{ NP} \]

As another example, the rule

\[ X \rightarrow A \text{ B C D E} \]

should be converted to the rules

\[ X \rightarrow A-B-C-D E \]
\[ A-B-C-D \rightarrow A-B-C \text{ D} \]
\[ A-B-C \rightarrow A-B \text{ C} \]
\[ A-B \rightarrow A \text{ B} \]

**Question 10** Write a function `(binarize-rule rule)` which takes a rule as its argument, and returns a list of one or more rules as its value. If the rule is binary, a list of a single rule should be returned. Non-binary rules should be transformed as shown in the examples above. For example, the function should have the following behavior (note that when a list of rules is returned, their particular order doesn’t matter, we’ve just shown one possible order below):

\[(binarize-rule '(S NP VP)) \rightarrow ((S NP VP))\]
\[(binarize-rule '(X A B C D E)) \rightarrow ((A-B A B) (A-B-C A-B C) (A-B-C-D A-B-C D) (X A-B-C-D E))\]

Note: we’ve provided a function `(make-nt x y)` in the code, which you may find useful. For example `(make-nt 'A 'B)` returns the symbol A-B.

**Question 11** Now write a function `binarize-rule-list` that takes a list of rules which may include some non-binary rules, and returns a new list where all the rules are binary. For example, this function should have the following behavior:

\[(define rules ‘((S NP VP) (VP V5 NP SBAR)))\]
\[(binarize-rule-list rules) \rightarrow ((S NP VP) (V5-NP V5 NP) (VP V5-NP SBAR))\]
Problem 7: A Translation System

Nim is delighted that he now has a grammar checker. But he also realises that the parser can be used for several other purposes. In particular, he is now interested in building a system that can translate between his sign language and another language.

In English, the usual word order is “subject–verb–object” (SVO): i.e., the subject typically comes first, the verb second, and the object next. As one example, in *Nim likes Noam* we have *Nim* as the subject, *likes* as the verb, and *Noam* as the object.

Other languages, however, have systematically different word order. For example, we might want to translate from English into a language that has “subject–object–verb” (SOV) order. To see how this might work, consider the following parse tree:

```
S
 /    \
|     |
NP    VP
 |     |
| Nim |
    V   NP
     |   |
    likes oranges
```

To translate this into SOV word order, we would transform this tree to the following structure:

```
S
 /    \
|     |
NP    VP
 |     |
| Nim |
    NP   V
     |   |
    oranges likes
```

Note that this transformation involves simply swapping the order of the children under the *VP* node in the tree. As a more complicated example, consider the following sentence, which is in English (SVO) word order:

```
S
 /    \
|     |
NP    VP
 |     |
| Nim |
    V   SBAR
     |   |
    liked COMP S
     |   |
    that NP VP
         |   |
         NP   VP
          |   |
          he   NP
           |   |
           likes oranges
```

In this case we’d like to transform the tree to the following structure:
Again, the transformation involves reordering every VP node in the tree.

**Question 12** Write a function `(svo->sov tree)` which takes a tree and returns a new tree with all VP nodes reordered. For example, the function should have the following behaviour:

```scheme
(define x (parse '((Nim likes oranges) grammar)))
; x has the value (S (NP Nim) (VP (V likes) (NP oranges)))
(translate x) ; should evaluate to (S (NP Nim) (VP (NP oranges) (V likes)))
```

In some cases in translation, we’d also like to change the words in the parse tree’s sentence. Consider the following function:

```scheme
(define (eng->french word)
  (cond ((eq? word 'eats) 'mange)
        ((eq? word 'he) 'il)
        (else word)))
```

This is a function of type `symbol -> symbol`. We’d like to have a function `(translate-tree tree f)` which can be used in the following way:

```scheme
(translate-tree '(S (NP he) (VP (V eats) (NP oranges))) eng->french)
;-> (S (NP il) (VP (V mange) (NP oranges)))
```

The function `(translate-tree tree f)` returns a new parse tree, where the function `f` is applied to each of the words in the tree.

**Question 13** Write the function `translate-tree`.

To complete the translation system, we’d like to build a function `(parse-tree->sentence tree)` that maps a tree structure to the sequence of words at its leaves. For example, this function should have the following behavior:
Question 14 Write the code for (parse-tree->sentence tree).

Problem 8: More Tree Transformations

In the final part of the project, write at least one more function that transforms parse trees in some way that you think is interesting. Here are some examples of things you might try (both of these are pretty difficult!—it’s ok if you decide to try something simpler):

- Try writing a function (sentence->question tree) that takes a parse tree as input, and returns a new parse tree in question form as its return value. For example, if you evaluate

  (sentence->question (parse '(Nim eats oranges) grammar))

  you should get a new parse tree with the sentence does Nim eat oranges as the sentence in the parse tree.

- Nim is a big Star Wars fan: Chewbacca was one of his childhood heroes. He notices that Yoda speaks in an interesting way—here are some quotes:

  Always two there are, no more.
  Truly wonderful, the mind of a child is.
  Much to learn, you still have.

  As a first approximation, Yoda appears to take the object of any verb and move it to the front of the sentence. For example, Nim likes Noam would become Noam Nim likes.

  Nim would be delighted if he could emulate Yoda: for example, he’d like to be able to say things like

  oranges Nim likes
  clever Nim is because oranges he eats

  and so on. Write a function (sentence->yoda tree) which takes a parse-tree in English word order and returns a new tree with “Yoda-style” word order.