1.17 *Extended example:* A language processor

Since the earliest days of modern computing machinery, users have put computers to the task of processing text. For example, in 1964 an analysis of word frequencies was used to determine authorship of various essays in *The Federalist*. Progress in these fields is evident today in the grammatical advice your word processor gives you and in online tools for automatically translating the text of websites.

In this section we will see how useful ML datatypes, lists, tuples, and options are for describing the structure of a human language. We will write a program which will take a sentence in a (very small) subset of English, analyze it grammatically, and make transformations on it (such as changing the mood or tense).

First, we define the parts of speech included in our language. We represent these with datatypes, each with a `string` for extra information. The string is the word itself; the datatype is used to indicate the part of speech. (In big code examples like this, we omit the prompts and responses of the interpreter.)

```
datatype noun = Noun of string;
datatype article = Art of string;
datatype adjective = Adj of string;
datatype preposition = Prep of string;
datatype transitiveVerb = TV of string;
datatype intransitiveVerb = IV of string;
datatype linkingVerb = LV of string;
datatype adverb = Adv of string;
```

For review, a *transitive verb* is a verb that requires a direct object. In "The man hit the ball," *hit* is a transitive verb and *the ball* is its direct object. *Intransitive verbs* do not take direct objects: "The dog ran." *Linking verbs* are complemented by an adjective: "The woman felt smart."

So much for individual words. We next define phrases, also using datatypes. We will require a noun phrase to have an article. An adjective is optional, which makes a perfect use of an option type.

```
datatype nounPhrase = NounPhrase of (article * adjective option * noun);
```

Earlier we defined transitive, intransitive, and linking verbs as distinct parts of speech, for convenience. We can comprehend all of these under the idea of a verb phrase.

```
datatype verbPhrase = TVP of (transitiveVerb * nounPhrase)
                    | IVP of (intransitiveVerb)
                    | LVP of (linkingVerb * adjective);
```

The predicate of a sentence is the main verb phrase, or the sentence itself without its subject. In our language, we take a verb phrase and add an optional adverb, such as "*happily* greeted the unicorn."

```
datatype predicate = Predicate of (adverb option * verbPhrase);
```

A prepositional phrase is a preposition plus the preposition's object, which is a noun phrase.

```
datatype prepPhrase = PrepPhrase of (preposition * nounPhrase);
```

Finally, a sentence is a noun phrase (the subject), a predicate, and optionally a prepositional phrase modifying the predicate.

```
datatype sentence = Sentence of (nounPhrase * predicate *
                                 prepPhrase option);
```

The big dog quickly chased a red ball through the bright field.

parsing

Now for the difficult talk of *parsing* a sentence. To parse is to analyze the syntax (that is, grammar or structure) of a string in a language. Because we are still somewhat limited by the about of ML we have learned in this chapter, we will also use the code for parsing to define the (very small) vocabulary for our language—this would more naturally be separated into its own step.

Ultimately we want a function `parseSentence` that takes a string and returns a sentence. We will make a set of helper functions first: `parseArticle`, `parseAdjective`, and others to identify words and their parts of speech; then `parseNounPhrase`, `parseVerbPhrase` and others to parse larger portions; finally `parseSentence`.

The parsing function (except for `parseSentence`) will operate under the following assumptions:

- They are given a list of words—that is, their parameter has type string list.
- The first portion of words on that list together constitute the part of speech or phrase that the parsing function is looking for.
- The parsing function "consumes" those words.
- The parsing function returns two things: the datatype value that it parses for and the rest of the list it was given, after consuming the words making up the portion of the sentence it was parsing.

That all means this. We would call `parseNounPhrase` with a list like `["the", "big", "dog", "quickly", "chased", "a", "red", "ball", "through", "the", "bright", "field"]`. The first three items in that list constitute a noun phrase, and we would not call this function on a list that did not start with one. The function will return a tuple, the first item being a nounPhrase, in this case

```
NounPhrase(Art("the"), SOME Adj("big"), Noun("dog"))
```

The other item in the tuple will be the rest of the list: `["quickly", "chased", "a", "red", "ball", "through", "the", "bright", "field"]`.

Thus each function will process a part of the list and will return the result of the processing and the remainder of the list still needing processing.

The function `parseArticle` is simple enough, but it will fail if the list does not have an article at the beginning.

```
fun parseArticle("the"::rest) = (Art("the"), rest)
  | parseArticle("a"::rest) = (Art("a"), rest);
```

We use `parseAdjective` to specify which adjectives are in our vocabulary (you can add to this easily if you like). Since adjectives are optional, the first item in the tuple is adjective option, and if the list does not begin with one of our recognized adjectives, then we return `NONE` and the unchanged list.

```
fun parseAdjective("big"::rest) = (SOME (Adj("big")), rest)
  | parseAdjective("bright"::rest) = (SOME (Adj("bright")), rest)
  | parseAdjective("fast"::rest) = (SOME (Adj("fast")), rest)
  | parseAdjective("beautiful"::rest) = (SOME (Adj("beautiful")), rest)
  | parseAdjective("smart"::rest) = (SOME (Adj("smart")), rest)
  | parseAdjective("red"::rest) = (SOME (Adj("red")), rest)
  | parseAdjective("smelly"::rest) = (SOME (Adj("smelly")), rest)
  | parseAdjective(wordList) = (NONE, wordList);
```

Parsing nouns, prepositions, and adverbs are similar, adverbs being optional:

```
fun parseNoun("man"::rest) = (Noun("man"), rest)
  | parseNoun("woman"::rest) = (Noun("woman"), rest)
  | parseNoun("dog"::rest) = (Noun("dog"), rest)
  | parseNoun("unicorn"::rest) = (Noun("unicorn"), rest)
  | parseNoun("ball"::rest) = (Noun("ball"), rest)
  | parseNoun("field"::rest) = (Noun("field"), rest)
  | parseNoun("flea"::rest) = (Noun("flea"), rest)
  | parseNoun("tree"::rest) = (Noun("tree"), rest)
  | parseNoun("sky"::rest) = (Noun("sky"), rest);

fun parsePreposition("in"::rest) = (Prep("in"), rest)
  | parsePreposition("on"::rest) = (Prep("on"), rest)
  | parsePreposition("through"::rest) = (Prep("through"), rest)
  | parsePreposition("with"::rest) = (Prep("with"), rest);

fun parseAdverb("quickly"::rest) = (SOME (Adv("quickly")), rest)
  | parseAdverb("slowly"::rest) = (SOME (Adv("slowly")), rest)
  | parseAdverb("dreamily"::rest) = (SOME (Adv("dreamily")), rest)
  | parseAdverb("happily"::rest) = (SOME (Adv("happily")), rest)
  | parseAdverb(wordList) = (NONE, wordList);
```

Now to parse a complete noun phrase, we grab an article, an adjective, and a noun, in sequence. Each of these will generate a new rest-of-the-list, which we feed into the next function. The last rest-of-the-list must be returned with the nounPhrase.

```
fun parseNounPhrase(wordList) =
    let val (art, rest1) = parseArticle(wordList);
        val (adj, rest2) = parseAdjective(rest1);
        val (nn, rest3) = parseNoun(rest2);
    in
        (NounPhrase(art, adj, nn), rest3)
    end;
```

For parsing verbs, we need to do something different depending on what kind of verb it is. Our `parseVerbPhrase` function will grab the verb itself and determine whether it is transitive, intransitive, or linking. It will then pass the verb and what is left of the word list to appropriate helper functions. For transitive verbs, we need a helper function to grab a noun phrase and package the verb and direct object together; for linking verbs, the helper function must grab and package a predicate adjective.

```
fun parseTransVerb(vb, wordList) =
    let val (dirObj, rest) = parseNounPhrase(wordList);
    in (TVP(vb, dirObj), rest)
    end;

fun parseLinkingVerb(vb, wordList) =
    let val (adj, rest) = parseAdjective(wordList);
    in (LVP(vb, valOf(adj)), rest)
    end;
```

A helper function for intransitive verbs is unnecessary—`pareseVerbPhrase` can handle them on the spot. Notice that our verbs are assumed to be simple past tense.

```
fun parseVerbPhrase("chased"::rest) =
                parseTransVerb(TV("chased"), rest)
  | parseVerbPhrase("saw"::rest) =
                parseTransVerb(TV("saw"), rest)
  | parseVerbPhrase("greeted"::rest) =
                parseTransVerb(TV("greeted"), rest)
  | parseVerbPhrase("bit"::rest) =
                parseTransVerb(TV("bit"), rest)
  | parseVerbPhrase("loved"::rest) =
                parseTransVerb(TV("loved"), rest)
```

```
  | parseVerbPhrase("ran"::rest) = (IVP(IV("ran")), rest)
  | parseVerbPhrase("slept"::rest) = (IVP(IV("slept")), rest)
  | parseVerbPhrase("sang"::rest) = (IVP(IV("sang")), rest)
  | parseVerbPhrase("was"::rest) =
              parseLinkingVerb(LV("was"), rest)
  | parseVerbPhrase("felt"::rest) =
              parseLinkingVerb(LV("felt"), rest)
  | parseVerbPhrase("seemed"::rest) =
              parseLinkingVerb(LV("seemed"), rest);
```

By now you should be able to understand `parsePredicate` and `parsePrepPhrase` for yourself, though `parseSentence` will require some explanation.

```
fun parsePredicate(wordList) =
    let val (adv, rest1) = parseAdverb(wordList);
        val (vPh, rest2) = parseVerbPhrase(rest1);
    in (Predicate(adv, vPh), rest2)
    end;

fun parsePrepPhrase(wordList) =
    let val (prep, rest1) = parsePreposition(wordList);
        val (nPh, rest2) = parseNounPhrase(rest1);
    in (PrepPhrase(prep, nPh), rest2)
    end;

fun parseSentence(message) =
   let val wordList = String.tokens(fn(x) =>
                               not(Char.isAlpha(x)))(message);
       val (subj, rest1) = parseNounPhrase(wordList);
       val (pred, rest2) = parsePredicate(rest1);
    in
      case rest2 of
        [] => Sentence(subj, pred, NONE)
      | prPh => Sentence(subj, pred,
                         SOME(#1(parsePrepPhrase(prPh))))
    end;
```

The part about `String.tokens(fn(x)` ... uses more ML than we have covered so far. All you need to know is that it turns a string containing a sentence into a list of strings, each string holding one word. It discards spaces and punctuation. Then we parse the subject and the predicate. What is left is the list `rest2`. If there is no prepositional phrase at the end, then `rest2` is empty, and the sentence is `Sentence(subj, pred, NONE)`—the first pattern in the case expressions handles this. Notice that the second pattern in the case expression effectively renames `rest2`

to `prPh` (as well as guaranteeing that it is not empty). The call to `parsePrepPhrase` returns a tuple—the prepPhrase and the (now empty) list—of which we want only the first. We package that into an option (`SOME(#1(...))`), and the sentence is complete.

```
- val dog = parseSentence("the big dog quickly chased a red ball
= through the bright field");


val dog =
  Sentence
    (NounPhrase (Art #,SOME #,Noun #),Predicate (SOME #,TVP #),
     SOME (PrepPhrase (#,#))) : sentence
```

Unfortunately, since the ML interpreter abbreviates long responses like this, we still have little to see for all the effort we have put in. Although parsing itself is an interesting problem, the fact is that so far we have only built values for our datatypes—we have not yet implemented any operations on the datatypes themselves.

To see how powerful this system is, we will use it to transform the sentences. The sentences that can be parsed so far must be declarative, meaning the sentences state a simple fact. In English, interrogative sentences (for example, "why" questions) require the verb to change form:

Why did the big dog quickly chase a red ball through the bright field?

Chased became *did chase*, and the two words fall to different places in the sentence. To make a function `makeInterrogative` that will print the sentence after transforming it to a "why" question, we first make helper functions to transform individual verbs. Note that these must return tuples, string * string:

```
fun interrogativeT(TV("chased")) = ("did", "chase")
  | interrogativeT(TV("saw")) = ("did", "see")
  | interrogativeT(TV("greeted")) = ("did", "greet")
  | interrogativeT(TV("bit")) = ("did", "bite")
  | interrogativeT(TV("loved")) = ("did", "love");

fun interrogativeI(IV("ran")) = ("did", "run")
  | interrogativeI(IV("slept")) = ("did", "sleep")
  | interrogativeI(IV("sang")) = ("did", "sing");

fun interrogativeL(LV("was")) = ("was", "")
  | interrogativeL(LV("felt")) = ("did", "feel")
  | interrogativeL(LV("seemed")) = ("did", "seem");
```

Since our transformed sentences are going to be printed as regular strings, we need some helper functions to undo the parsing of parts of the sentence, that is, to take values in our data types and turn them back into strings.

```
fun printNounPhrase(NounPhrase(Art(a), adj, Noun(n))) =
    a ^ " " ^ (case adj of SOME(Adj(aa)) => aa ^ " "
                         | NONE => "") ^ n;

fun printPrepPhrase(SOME(PrepPhrase(Prep(p), nPh))) =
      " " ^ p ^ " " ^ printNounPhrase(nPh)
   | printPrepPhrase(NONE) = "";

fun printAdverb(SOME(Adv(aa))) =  " " ^ aa
   | printAdverb(NONE) = "";
```

Now for the real work. The function `makeInterrogative` uses pattern matching based on the the kind of verb phrase in the predicate.

```
fun makeInterrogative(Sentence(subj, Predicate(adv, TVP(v, dObj)),
                               prPh)) =
    let val (v1, v2) = interrogativeT(v)
    in
      "why " ^ v1 ^ " " ^ printNounPhrase(subj) ^ " " ^ v2
      ^ " " ^ printNounPhrase(dObj) ^
      printAdverb(adv) ^ printPrepPhrase(prPh) ^ "?"
    end
  | makeInterrogative(Sentence(subj, Predicate(adv, IVP(v)),
                               prPh)) =
    let val (v1, v2) = interrogativeI(v)
    in
      "why " ^ v1 ^ " " ^ printNounPhrase(subj) ^ " " ^ v2 ^
      printAdverb(adv) ^ printPrepPhrase(prPh) ^ "?"
    end
  | makeInterrogative(Sentence(subj, Predicate(adv, LVP(v, Adj(a))),
                               prPh)) =
    let val (v1, v2) = interrogativeL(v)
    in
      "why " ^ v1 ^ " " ^ printNounPhrase(subj) ^ " " ^ v2 ^
      printAdverb(adv) ^
       " " ^ a ^
      printPrepPhrase(prPh) ^ "?"
     end;

- makeInterrogative(dog);
```

```
val it =
  "why did the big dog chase a red ball quickly through the
bright field?"
  : string

- makeInterrogative(parseSentence("the beautiful woman felt
= smart"));

val it = "why did the beautiful woman feel smart?" : string

- makeInterrogative(parseSentence("the flea dreamily slept on
= the smelly unicorn"));

val it = "why did the flea sleep dreamily on the smelly
unicorn?" : string
```

Project

1.A Write a function `diagram` (and appropriate helper functions, say `diagramNP`, `diagramVP`, etc) that takes a `sentence` and produces a string that shows the structure of the sentence in a readable format, equivalent to diagramming a sentence. The diagram of our sample sentence would be something like `(dog, the, big)(chased:(ball, a, red), quickly(through(field, the, bright)))`.

1.B Following the pattern for `makeInterrogative`, write a function `makeImperative` (and appropriate helper functions) that transforms and prints the sentence into the imperative mood. The verb should be in an appropriate place, and the subject should be vocative. Our sample sentence would be transformed into something like, *Chase a red ball quickly through the bright field, o big dog!"*

1.C Add to the datatypes and parsing functions so that our system will allow present active participles such as *chasing*. Participial phrases have the grammatical value of adjectives. They also need to be differentiated by whether the participle is of a transitive, intransitive, or linking verb, and have a direct object or predicate adjective if appropriate. Once present active participles are implemented, our language has the imperfect tense for free, since sentences can be constructed like *The big dog was chasing the red ball.*

1.D So far in our system, prepositional phrases have been only with an adverbial value, meaning they modify the verb phrase (*through the bright field* describes the action of chasing). However, prepositional phrases can also have an adjectival value, where they would modify a noun: "The dog *with the smart flea* chased the ball." Modify the datatypes and parsing functions so that prepositional phrases can follow a noun and modify it. One thing that makes this difficult is that if a prepositional phrase occurs at the end of a sentence that has a transitive verb, it is ambiguous whether it modifies the direct object or the verb. You can solve this by differentiating between prepositions that form adverbial prepositional phrases (such as *through*) and those that form adjective prepositional phrases (such as *with*).

1.E The system presented here will crash if given any string that is not in the language. Modify the parsing functions so that an exception is thrown when input that is not parsable is given. The parsing functions should display an error message and return a default value.

5.3 Mutual recursion

Recall our language processing system from Section 1.17. Suppose we wanted to extend the system to accept sentences like *The unicorn knew that the big dog was smelly.* What is different about this?

The independent clause *the big dog was smelly*, which would be a sentence by itself, is imbedded in the larger sentence, and it is the direct object of the verb *knew*. What we want to introduce to the language is independent clauses (sentences or proposition prefixed with *that*) and verbs that take propositions as their direct object.

In theory, *that*- introduced propositions are full-fledged nouns, just as boolean-valued expressions are full-fledged expressions. However, since only a small number of verbs make sense with propositions as direct objects (The woman *proved* that a red dog ran through the field) or propositions as subjects (That a flea bit the unicorn *concerned* the man), we will treat those verbs specially. Also, for simplicity's sake, we restrict our discussion to verbs taking propositions as direct objects.

A sentence contains a predicate, which contains a verb phrase such as a transitive propositional-object verb phrase, which contains a sentence. A sentence is defined in terms of itself, but not directly: a sentence is defined in terms of a verb phrase, which is in turn defined in terms of a sentence. This is *mutual recursion*, where two or more things are defined in terms of each other.

mutual recursion

Consider two examples of pairs of simple functions that are mutually recursive. Suppose we want to cut down a list to make a new list containing every other element in the original list. For example, we would transform `[6, 3, 12, 8, 4, 2, 7]` to `[6, 12, 4, 7]`. We can decompose this process into two parts: taking an element to add to the new list and skipping an element. If we take, then we want to skip next time, and vice versa. (This example is from Ullman [28].)

```
- fun take([]) = []
=   | take(x::rest) = x::skip(rest)
= and skip([]) = []
=   | skip(y::rest) = take(rest);


val take = fn : 'a list -> 'a list
val skip = fn : 'a list -> 'a list

- take([6, 3, 12, 8, 4, 2, 7]);


val it = [6,12,4,7] : int list
```

There are two differences: the `fun` of the second function is replaced with `and`, and only the last function is terminated by a semicolon.

As a second example, we can rewrite `isEven` and `isOdd` to be mutually recursive:

```
- fun isEven(0) = true
=    | isEven(n) = isOdd(n-1)
= and isOdd(0) = false
=   | isOdd(n) = isEven(n-1);


val isEven = fn : int -> bool
val isOdd = fn : int -> bool

- isEven(17);


val it = false : bool

- isOdd(17);


val it = true : bool
```

Mutually recursive datatype also are joined together using `and`. Much of the code in this language example is the same as from Section 1.17, but first we need to add a datatype for transitive propositional-object verbs:

```
datatype transpoVerb = TPOV of string;
```

Next, we define phrases and sentences:

```
datatype nounPhrase = NounPhrase of
                              (article * adjective option * noun)
and verbPhrase = TVP of (transitiveVerb * nounPhrase)
                    | IVP of (intransitiveVerb)
                    | LVP of (linkingVerb * adjective)
                    | TPOVP of (transpoVerb * sentence)
and predicate = Predicate of (adverb option * verbPhrase)
and prepPhrase = PrepPhrase of (preposition * nounPhrase)
and sentence = Sentence of (nounPhrase * predicate *
                            prepPhrase option);
```

The parsing functions for articles, adjectives, nouns, prepositions, and adverbs can stay the same, and similarly for `parseNounPhrase`, `parsePrepPhrase`, `parseTransVerb`, and `parseLinkingVerb`. The function `parseTransPOVerb` is new, and the remaining parse functions (for verb phrases, predicates, and sentences) are reorganized below. The original version of the language processor was written before we had introduce `if`s, so all decision making was done by pattern-matching. We can simplify matters now by keeping a lists of different kinds of verbs:

```
val transitiveVerbs = ["chased", "saw", "bit", "loved"];
val intransitiveVerbs = ["ran", "slept", "sang"];
val linkingVerbs = ["was", "felt", "seemed"];
val transitivePOVerbs = ["knew", "believed", "proved"];
exception unknownVerb;
```

For reasons to be explained soon, we also make a list of prepositions:

```
val prepositions = ["in", "on", "through", "with"];
```

Now, the remaining parsing code, all at once:

```
fun parseTransPOVerb(vb, wordList) =
    let val (dirObj, rest) = parseSentence(wordList);
    in (TPOVP(vb, dirObj), rest)
    end

and parseVerbPhrase(vb::rest) =
    if contains(vb, transitiveVerbs) then parseTransVerb(TV(vb), rest)
    else if contains(vb, intransitiveVerbs) then (IVP(IV(vb)), rest)
    else if contains(vb, linkingVerbs)
         then parseLinkingVerb(LV(vb), rest)
    else if contains(vb, transitivePOVerbs) andalso hd(rest) = "that"
         then parseTransPOVerb(TPOV(vb), tl(rest))
    else raise UnknownVerb

and parsePredicate(wordList) =
    let val (adv, rest1) = parseAdverb(wordList);
        val (vPh, rest2) = parseVerbPhrase(rest1);
    in (Predicate(adv, vPh), rest2)
    end

and parseSentence(wordList) =
    let val (subj, rest1) = parseNounPhrase(wordList);
       val (pred, rest2) = parsePredicate(rest1);
    in
      case rest2 of
        [] => (Sentence(subj, pred, NONE), [])
      | next::rest3 =>
         if contains(next, prepositions)
         then let val (prPh, rest4) = parsePrepPhrase(rest3);
              in (Sentence(subj, pred, SOME prPh), rest4)
              end
         else (Sentence(subj, pred, NONE), rest3)
    end
```

```
and parseString(message) =
    #1(parseSentence(String.tokens(fn(x) => not(Char.isAlpha(x)))
                     (message)));
```

We separated the old `parseSentence` (which took a sentence-string and chopped it up into a list of word-strings) into `parseSentence` and `parseString` because having subsentences means that we need to parse sentences from word lists.

In `parseSentence`, we needed to check whether `rest2` (the words following the predicate) began with a preposition and parse the prepositional phrase if it did. The sentence *The man believed the unicorn slept in the field* is technically ambiguous as to whether *in the field* describes the place of the unicorn's sleeping or the place of the man's believing. Our parser associates prepositional phrases with the nearest, smallest sentence, as would be our intuition in the given example.

Now the sentences in our system can be arbitrarily nested.

```
- parseString("the man happily believed that the woman knew that
= the man proved that the man loved the woman");

val it =  Sentence (NounPhrase (Art #,NONE,Noun #),
Predicate (SOME #, TPOVP #),NONE)  : sentence
```

Exercises

5.3.1 What is wrong with the following?

```
fun isEven(0) = true
  | isEven(n) = isOdd(n-1)
and isOdd(1) = true
  | isOdd(n) = isEven(n-1);
```

5.3.2 Suppose you wanted every other item in a list but starting with the second item, not the first. How could you achieve this with code we have already written (that is, without writing a new function)?

A *forest* is a set of trees. We can define a type `tree` where each node contains an `int` datum and an arbitrary number of children with

```
datatype tree = Node of int * forest
and forest = Forest of tree list;
```

So, a node's children are viewed as a forest of trees.

5.3.3 Write mutually recursive functions `sumTree` and `sumForest` that determine the sum of all the data in a tree or forest.

5.3.4 Write mutually recursive functions `heightTree` and `heightForest` that determine the height of a tree or forest.

5.3.5 Write mutually recursive functions `leavesTree` and `leavesForest` that determine the number of leaves in a tree or in a forest. Notice that a leaf is a tree node with an empty forest.

5.3.6 Extend the language processor by introducing relative clauses: "The man *who chased the unicorn* saw a dog *that was big*." Restrict the system to relative clauses where the relative pronoun is the subject of the clause (*the dog that chased the ball*, not the *the ball that the dog chased* or *the field in which the dog chased the ball*). Assume relative pronouns *who*, *that*, and *which*. The relative clause should be an optional part of a noun phrase and come last in the noun phrase. Relative clauses can be arbitrarily nested (*the man who loved the woman who walked with the unicorn that chased the dog slept in the bright field*).