8 Property Lists

In the first volume of this series, I wrote a procedure named `french` that translates words from English to French using a dictionary list like this:

```
[[book livre] [computer ordinateur] [window fenetre]]
```

This technique works fine with a short word list. But suppose we wanted to undertake a serious translation project, and suppose we wanted to be able to translate English words into several foreign languages. (You can buy hand-held machines these days with little keyboards and display panels that do exactly that.) `Butfirst`ing through a list of tens of thousands of words would be pretty slow, and setting up the lists in the first place would be very difficult and error-prone.

If we were just dealing with English and French, one solution would be to set up many variables, with each an English word as its *name* and the corresponding French word as its *value:*

```
make "paper "papier
make "chair "chaise
make "computer "ordinateur
make "book "livre
make "window "fenetre
```

Once we've done this, the procedure to translate from English to French is just `thing`:

```
? print thing "book
livre
```

The advantage of this technique is that it's easy to correct a mistake in the translation; you just have to assign a new value to the variable for the one word that is in error, instead of trying to edit a huge list.

But we can't quite use this technique for more than one language. We could create variables whose names contained both the English word and the target language:

```
make "book.french "livre
make "book.spanish "libro

to spanish :word
output thing word :word ".spanish
end
```

This is a perfectly workable technique but a little messy. Many variables will be needed. A compromise might be to collect all the translations for a single English word into one list:

```
make "book [livre libro buch libro liber]

to spanish :word
output item 2 thing :word
end
```

Naming Properties

The only thing wrong with this technique is that we have to remember the correct order of the foreign languages. This can be particularly tricky because some of the words are the same, or almost the same, from one language to another. And if we happen not to know the translation of a particular word in a particular language, we have to take some pains to leave a gap in the list. Instead we could use a list that tells us the languages as well as the translated words:

```
[French livre Spanish libro German buch Italian libro Latin liber]
```

A list in this form is called a *property list.* The odd-numbered members of the list are property *names,* and the even-numbered members are the corresponding property *values.*

You can see that this solution is a very flexible one. We can add a new language to the list later, without confusing old procedures that expect a particular length of list. If we don't know the translation for a particular word in a particular language, we can just leave it out. The order of the properties in the list doesn't matter, so we don't have to

remember it. The properties need not all be uniform in their significance; we could, for example, give `book` a property whose name is `part.of.speech` and whose value is `noun`.

To make this work, Berkeley Logo (along with several other dialects) has procedures to create, remove, and examine properties. The command `pprop` (Put PROPerty) takes three inputs; the first two must be words, and the third can be any datum. The first input is the name of a property list; the second is the name of a property; the third is the value of that property. The effect of `pprop` is to add the new property to the named list. (If there was already a property with the given name, its old value is replaced by the new value.) The command `remprop` (REMove PROPerty) takes two inputs, which must be words: the name of a property list and the name of a property in the list. The effect of `remprop` is to remove the property (name and value) from the list. The operation `gprop` (Get PROPerty) also takes two words as inputs, the name of a property list and the name of a property in the list. The output from `gprop` is the value of the named property. (If there is no such property in the list, `gprop` outputs the empty list.)

```
? print gprop "book "German
buch
```

Writing Property List Procedures in Logo

It would be possible to write Logo procedures that would use ordinary variables to hold property lists, which would work just like the ones I've described. Since Berkeley Logo provides property lists as a primitive capability, you won't need to load these into your computer, but later parts of the discussion will make more sense if you understand how they work. Here they are:

```
to pprop :list :name :value
if not namep :list [make :list []]
make :list pprop1 :name :value thing :list
end

to pprop1 :name :value :oldlist
if emptyp :oldlist [output list :name :value]
if equalp :name first :oldlist ~
   [output fput :name (fput :value (butfirst butfirst :oldlist))]
output fput (first :oldlist) ~
            (fput (first butfirst :oldlist)
                  (pprop1 :name :value (butfirst butfirst :oldlist)))
end
```

```
to remprop :list :name
if not namep :list [make :list []]
make :list remprop1 :name thing :list
end

to remprop1 :name each
```

Note that the input called `list` in each of these procedures is not a property list itself but the *name* of a property list. That's why each of the superprocedures evaluates

to pass down as an input to its subprocedure.

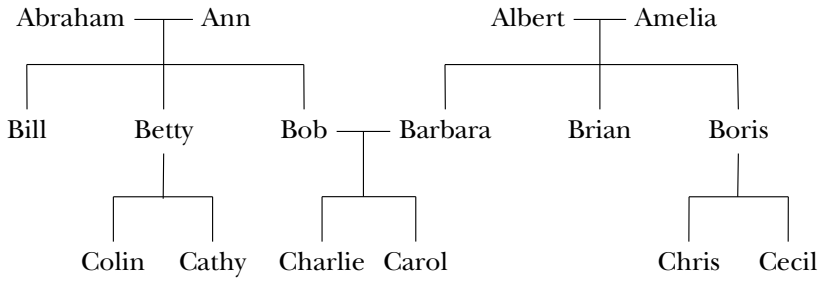
Property Lists Aren't Variables

The primitive procedures that support property lists in Berkeley Logo, however, do *not* use `thing` to find the property list. Just as the same word can independently name a procedure and a variable, a property list is a *third* kind of named entity, which is separate from the `thing` with the same name. For example, if we gave `book` the property list shown with a series of instructions like

So why is `gprop` permissive when all other Logo primitives are not? Well, the others were designed early in the history of the language when teachers were in charge at the design meetings. Property lists were added to Logo more recently; the implementors showed up one day and said, "Guess what? We've put in property lists." So they did it their way!

An Example: Family Trees

Here is an example program using property lists. My goal is to represent this family tree:



Each person will be represented as a property list, containing the properties `mother`, `father`, `kids`, and `sex`. The first two will have words (names) as their values, `kids` will be a list of names, and `sex` will be `male` or `female`. Note that this is only a partial family tree; we don't know the name of Betty's husband or Boris's wife. Here's how I'll enter all this information:

```
to family :mom :dad :girls :boys
catch "error [pprop :mom "sex "female]
catch "error [pprop :dad "sex "male]
foreach :girls [pprop ? "sex "female]
foreach :boys [pprop ? "sex "male]
localmake "kids sentence :girls :boys
catch "error [pprop :mom "kids :kids]
catch "error [pprop :dad "kids :kids]
foreach :kids [pprop ? "mother :mom   pprop ? "father :dad]
end

family "Ann "Abraham [Betty] [Bill Bob]
family "Amelia "Albert [Barbara] [Brian Boris]
family "Betty [] [Cathy] [Colin]
family "Barbara "Bob [Carol] [Charlie]
family [] "Boris [] [Chris Cecil]
```

The instructions that catch errors do so in case a family has an unknown mother or father, which is the case for two of the ones in our family tree.

Now the idea is to be able to get information out of the tree. The easy part is to get out the information that is there explicitly:

```
to mother :person
output gprop :person "mother
end

to kids :person
output gprop :person "kids
end

to sons :person
output filter [equalp (gprop ? "sex) "male] kids :person
end
```

Of course several more such procedures can be written along similar lines.

The more interesting challenge is to deduce information that is not explicitly in the property lists. The following procedures make use of the ones just defined and other obvious ones like `father`.

```
to grandfathers :person
output sentence (father father :person) (father mother :person)
end

to grandchildren :person
output map.se [gprop ? "kids] (kids :person)
end

to granddaughters :person
output justgirls grandchildren :person
end

to justgirls :people
output filter [equalp (gprop ? "sex) "female] :people
end

to aunts :person
output justgirls sentence (siblings mother :person) ~
                          (siblings father :person)
end
```

```
to cousins :person
output map.se [gprop ? "kids] sentence (siblings mother :person) ~
                                       (siblings father :person)
end

to siblings :person
local "parent
if emptyp :person [output []]
make "parent mother :person
if emptyp :parent [make "parent father :person]
output remove :person kids :parent
end
```

In writing `siblings`, I've been careful to have it output an empty list if its input is empty. That's because `aunts` and `cousins` may invoke `siblings` with an empty input if we're looking for the cousins of someone whose father or mother is unknown.

You'll find, if you try out these procedures, that similar care needs to be exercised in some of the "easy" procedures previously written. For example, `grandfathers` will give an error message if applied to a person whose mother *or* father is unknown, even if the other parent is known. One solution would be a more careful version of `father`:

```
to father :person
if emptyp :person [output []]
output gprop :person "father
end
```

The reason for choosing an empty list as output for a nonexistent person rather than an empty word is that the former just disappears when combined with other things using `sentence`, but an empty word stays in the resulting list. So `grandfathers`, for example, will output a list of length 1 if only one grandfather is known rather than a list with an empty word in addition to the known grandfather. Procedures like `cousins` also rely heavily on the flattening effect of `sentence`.

This is rather an artificial family tree because I've paid no attention to family names, and all the given names are unique. In practice, you wouldn't be able to assume that. Instead, each property list representing a person would have a name like `person26` and would include properties `familyname` and `givenname` or perhaps just a `name` property whose value would be a list. All the procedures like `father` and `cousins` would output lists of these funny `person26`-type names, and you'd need another procedure `realnames` that would extract the real names from the property lists of people in a list. But I thought it would be clearer to avoid that extra level of naming confusion here.