1

Wordplay

Sengen: A Sentence Generator

SENGEN makes up English sentences similar to the following ones:

```
PECULIAR BIRDS HATE JUMPING DOGS
FAT WORMS HATE PECULIAR WORMS
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS
FAT GEESE BITE JUMPING CATS
```

One of the questions you might ask is this: Does SENGEN make up sentences the way we do or the way we did when we first learned to talk or write? Another question you might ask is: What relationship does SENGEN have to understanding grammar? The first question is open to research and speculation. The second might be an easier one to answer. Often when I first discuss this project with children, they do not relate the programming process to the learning of grammar. Later as they use their programs, the children frequently exclaim: "So this is why they call words nouns and verbs!" They also begin to appreciate formal systems. Studying grammar by generating sentences that obey certain rules requires the programmer to become aware of rules as well as of their exceptions.

Since this program seems to make sensible sentences without knowing very much about grammar, children often develop an appreciation for cleverness. For example, SENGEN doesn't know that some words are singular and some are plural or that singular subjects should be matched with singular verbs; it does not know about verb tenses or pronomial relations. Its apparent intelligence comes from the programmer's choice of words and categories.

In the following examples, the nouns and verbs are all plurals and the verbs are all in the present tense.

SENGEN builds sentences from vocabulary lists of nouns, verbs, adjectives, connectives, and so on. It then assembles its selections according to some rule of grammar.

Making the Program

One strategy in making a program might be to concentrate on developing a random sentence generator that outputs only a verb. For example:

```
Go.
Run.
```

By Cynthia Solomon.

To do this a procedure is needed to blindly (randomly) pick out a selection from a list of possibilities.

Let's make up a list of verbs and then make a procedure to select a word from the list. In this example, the procedure VERBS outputs the vocabulary list.

```
TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE]
END
```

Whenever VERBS is called, it outputs that list.

```
PR VERBS
EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE

PR FIRST VERBS
EAT

PR LAST BL VERBS
TRIP
```

What we now want is a procedure that will randomly choose one of the items in this list. Here is the plan for this task: use a number obtained from RANDOM to point to an item in the given list of choices. Then get that item from the list. PICK does this and outputs the selection.

```
TO PICK :LIST
OP SELECT  RANDOM COUNT :LIST  :LIST
END
```

PICK's input is a vocabulary list. PICK calls SELECT, giving it the list and a number indicating which item in the list SELECT is to output.

There is a slight problem. RANDOM outputs a number from 0 up to but not including its input number. Thus its output in PICK is always one less than the length of the list. We can fix that by adding 1 to RANDOM's output.

```
TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END
```

SELECT carries out its job recursively. When its input number is one, it outputs the first item from its input list. Otherwise, SELECT subtracts one from its input number and takes away the first item from its input list and continues the process until the item is found.

Here is what SELECT looks like.

```
TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END
```

Now we can try PICK.

```
PR PICK VERBS
SCARE

PR PICK VERBS
EAT
```

We could try it on different lists:

```
PR PICK [1 2 3 4]
3

PR PICK [A B C D]
A
```

PICK seems to work.

Let's make a procedure that outputs just a verb.

```
TO VERB
OP PICK VERBS
END

PR VERB
BITE
```

Now we can move on to building a sentence by first making a one-word sentence.

```
TO SEN
PR VERB
SEN
END

SEN
LAUGH AT
SCARE
LAUGH AT
EAT
```

and so on.

Our attempt at making a one-word sentence fails because of the verbs in the verb list. Only EAT can be used without an object. So if we want to make grammatical one-word sentences, we have to restrict our choice of verbs.

Now let's make a sentence with a subject and an object. Let's follow the pattern already set up for verbs and make two operations NOUNS and NOUN. NOUNS outputs a list of nouns.

```
TO NOUNS
OP [BOYS [DOGS AND CATS] PUPPIES [SIAMESE FIGHTING FISH]
GEESE BIRDS GIRLS [GUINEA PIGS][MICE AND GERBILS] WORMS
TEACHERS DONKEYS CLOWNS [BASEBALL PLAYERS]]
END
```

NOUN outputs one of the items from NOUNS.

```
TO NOUN
OP PICK NOUNS
END

PR NOUN
CLOWNS
```

All we need to do to make a sentence is the following:

```
PR (SE NOUN VERB NOUN)
SIAMESE FIGHTING FISH SCARE BOYS
```

Imagine we had miscategorized the vocabulary and NOUNS could output a list like

```
RED LAUGHING TORTOISE BOY
```

We might then get sentences like

```
RED SCARE LAUGHING
BOY EAT TORTOISE
```

This kind of bug is typical of the kind people run into when they first do this project. Usually, when people confront their bugs, they begin to appreciate rules of grammar and the fantastic power we derive from categorizing words.

We can now make a procedure that outputs a sentence.

```
TO SEN
OP (SE NOUN VERB NOUN)
END
```

```
PR SEN
BASEBALL PLAYERS EAT DONKEYS
```

SENGEN can print this output and continue the process.

```
TO SENGEN
PR SEN
PR []
SENGEN
END
```

Extensions

One extension is to add adjectives to the sentences.

```
TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR JUMPING]
END
```

```
TO ADJECTIVE
OP PICK ADJECTIVES
END
```

Edit SEN.

```
TO SEN
PR (SE ADJECTIVE NOUN VERB NOUN)
END
```

The sentences are getting more complicated, so it is time to introduce additional categories like NOUNPHRASE and VERBPHRASE. For example:

```
TO NOUNPHRASE
OP (SE ADJECTIVE NOUN)
END

TO VERBPHRASE
OP (SE VERB NOUN)
END

TO SEN
OP (SE NOUNPHRASE VERBPHRASE)
END
```

Another possibility is to link two simple sentences by using connectives:

```
TO CONNECTS
OP [BUT AND [EVEN THOUGH]]
END

TO CONNECT
OP PICK CONNECTS
END
```

Finally, you change SENGEN to include the new sentence:

```
TO SENGEN
PR (SE SEN CONNECT SEN)
PR []
SENGEN
END
```

```
SENGEN
FAT DOGS HATE DONKEYS EVEN THOUGH
      JUMPING BASEBALL PLAYERS SCARE BOYS

RED CATS EAT WORMS EVEN THOUGH
      FUNNY BUNNY BASEBALL PLAYERS LOVE BOYS

FAT MICE AND GERBILS SCARE TEACHERS
      AND FAT CLOWNS BITE MICE AND GERBILS
```

PROGRAM LISTING

```
TO SENGEN
PR (SE SEN CONNECT SEN)
PR []
SENGEN
END

TO SEN
OP (SE NOUNPHRASE VERBPHRASE)
END

TO CONNECTS
OP [BUT AND [EVEN THOUGH]]
END

TO CONNECT
OP PICK CONNECTS
END

TO NOUNPHRASE
OP (SE ADJECTIVE NOUN)
END
```

```
TO NOUN
OP PICK NOUNS
END

TO NOUNS
OP [BOYS [DOGS AND CATS] PUPPIES ▸
     [SIAMESE FIGHTING FISH] GEESE ▸
     BIRDS GIRLS [GUINEA PIGS][MICE ▸
     AND GERBILS] WORMS TEACHERS ▸
     DONKEYS CLOWNS [BASEBALL ▸
     PLAYERS]]
END

TO VERB
OP PICK VERBS
END

TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] ▸
     TRIP BITE]
END

TO ADJECTIVE
OP PICK ADJECTIVES
END

TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR ▸
     JUMPING]
END

TO VERBPHRASE
OP (SE VERB NOUN)
END

TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END

TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END
```

Argue

ARGUE carries on a dialogue with you. When you run ARGUE, it expects you to type a statement in the form I LOVE LEMONS or I HATE DOGS. ARGUE comes back with contrary statements. For example, if you make the statement I HATE DOGS, the program types

```
I LOVE DOGS
I HATE CATS
```

If it doesn't already know the opposite of a word, it asks you. For example, if you type I LOVE LEMONS and ARGUE does not know the opposite of LEMONS, it types

```
I HATE LEMONS
WHAT IS THE OPPOSITE OF LEMONS?
```

If you tell it ORANGES, it will type

```
I LOVE ORANGES
```

Here is a sample dialogue.

Program by Danny Hillis; write-up by Margaret Minsky.

```
ARGUE
-->I LOVE SALT
I HATE SALT
I LOVE PEPPER

-->I HATE CATS
I LOVE CATS
I HATE DOGS

-->I HATE DOGS
I LOVE DOGS
I HATE CATS

-->I LOVE LEMONS
I HATE LEMONS
WHAT IS THE OPPOSITE OF LEMONS?ORANGES
I LOVE ORANGES

-->
```

ARGUE *Can Reply to Your Statements*

When you run ARGUE, it types an arrow to let you know that it is ready for you to type your statement, then calls ARGUEWITH. ARGUEWITH is given the statement you type as its input. ARGUE is recursive so this process continues.

```
TO ARGUE
TYPE [\-\-\>]
ARGUEWITH RL
ARGUE
END
```

ARGUEWITH prints two responses to your statement. First, it turns around your statement; if you say that you *love* something, ARGUEWITH says that it *hates* it, and if you say you *hate* something, ARGUEWITH says that it *loves* it. Second, it makes a statement about the opposite of the object you mentioned.

```
TO ARGUEWITH :STATEMENT
PRINT ( SE "I LOVE.HATE SECOND :STATEMENT LAST :STATEMENT )
PRINT ( SE "I SECOND :STATEMENT OPPOSITE LAST :STATEMENT )
END
```

The procedure LOVE.HATE sees whether its input is "LOVE or "HATE and outputs the other one.

```
TO LOVE.HATE :WORD
IF :WORD = "LOVE [OP "HATE]
IF :WORD = "HATE [OP "LOVE]
END
```

The ARGUEWITH procedure works only with statements in the form I LOVE *something* or I HATE *something* because it assumes that the second word in your statement is LOVE or HATE and that the last word in your statement is something whose opposite it can find.

ARGUEWITH uses SECOND to grab the second word in a sentence.

```
TO SECOND :LIST
OP FIRST BF :LIST
END
```

The OPPOSITE procedure is the real guts of the ARGUE program. It takes a word as its input and outputs the opposite of that word.

The Program Keeps Track of Opposites

How does the program know that *pepper* is the opposite of *salt*? Somehow, the ARGUE program has to have this information stored. We use variables to hold this information. For example, :SALT is PEPPER, :CATS is DOGS. This is how we have chosen to store the facts the program "knows." We call this a *data base*. You can look at the data base for the ARGUE program by looking at all the variables in the workspace. Try:

```
PONS
MAKE "PEPPER "SALT
MAKE "SALT "PEPPER
MAKE "DOGS "CATS
MAKE "CATS "DOGS
MAKE "LIFE "MARRIAGE
MAKE "MARRIAGE "LIFE
MAKE "DARK "LIGHT
MAKE "LIGHT "DARK
```

These variables are loaded into the workspace with the ARGUE program.*

To find out the opposite of something, for example DARK, we can say

```
PR :DARK
LIGHT
```

or

```
PR THING "DARK
LIGHT
```

What if we want to find out the opposite of LIGHT? There is no easy way to find out it is DARK unless we have another variable named LIGHT, with value DARK. So we can say

```
PR THING "LIGHT
DARK
```

We have set up a convention in our data base that we always put in both parts of a pair. That way, we don't end up in the funny situation where it

*If you type in the procedures and there are no variables in the workspace, ARGUE will create these variables when it asks you for the opposites of things.

is easy to find out that the opposite of ROUGH is SMOOTH, but impossible to find out what the opposite of SMOOTH is. Our mental concept of opposite is that it "goes both ways," so we make our data base reflect that.

How the OPPOSITE *Procedure Works*

With this kind of data base we can write a procedure to output the opposite of something. Here is a possible first version of the OPPOSITE procedure:

```
TO OPPOSITE :OBJECT
OP THING :OBJECT
END
```

This is a good example of needing to use THING rather than dots(:). The word of which OPPOSITE is trying to find the value is whatever :OBJECT is. For example, if :OBJECT is the word SALT, then the program is trying to find :SALT. It must do this indirectly by using THING :OBJECT.

This first version of OPPOSITE has a problem. It only works for words that are already in the data base. If you make a statement like I LOVE SUNSETS and there is no variable named SUNSETS, then this OPPOSITE procedure will get an error. To solve this problem, we use NAMEP to check for the existence of a variable named by :OBJECT. In this example :OBJECT is the word SUNSETS; the program checks whether there is already a variable named SUNSETS. If there isn't, you'd like the program to learn the opposite of SUNSETS and put it in the data base. Then it can go ahead and argue with you about sunsets. The procedure LEARNOPP does this. OPPOSITE calls LEARNOPP when it needs to.

```
TO OPPOSITE :OBJECT
IF NAMEP :OBJECT [OP THING :OBJECT]
PRINT ( SE [WHAT IS THE OPPOSITE OF] :OBJECT "? )
LEARNOPP :OBJECT FIRST RL
OP THING :OBJECT
END

TO LEARNOPP :OBJECT :OPP
MAKE :OBJECT :OPP
MAKE :OPP :OBJECT
END
```

When OPPOSITE tries to find the opposite of a word that is not in the data base, it asks the user for the opposite. After the user types the opposite, OPPOSITE passes both the problem word and its opposite to LEARNOPP. LEARNOPP puts that pair of words in the data base.

Now ARGUE *Can Argue Pretty Well*

So ARGUE can keep going as it adds new words to its data base.

```
ARGUE
-->I HATE PEPPER
I LOVE PEPPER
I HATE SALT
-->I LOVE SUNSETS
I HATE SUNSETS
WHAT IS THE OPPOSITE OF SUNSETS?SUNRISES
I LOVE SUNRISES
-->I LOVE SUNRISES
I HATE SUNRISES
I LOVE SUNSETS
```

and so on.

If we look at the data base after this, we can see what has been added.

```
PONS
MAKE "SUNRISES "SUNSETS
MAKE "SUNSETS "SUNRISES
MAKE "PEPPER "SALT
```

and so on.

In order for the program to "remember" this data base, these variables must be saved by SAVEing this workspace on a diskette.

SUGGESTIONS

The ARGUE program assumes that the sentences you type in are going to be exactly in the form

```
I LOVE something
```

or

```
I HATE something
```

If they are not, an error occurs and the program stops. You could improve the program so that it checks for the right kinds of sentences and asks you to retype them if there are problems.

Maybe it could know about more emotion words such as DESIRE, LIKE, DISLIKE, DESPISE, DETEST.

If you try:

```
I LOVE GREEN PEAS
```

the program will say:

```
I HATE PEAS
```

and ask you for the opposite of PEAS. It will ignore the GREEN. You might make a better arguing program that tries to figure out if there is an adjective and finds its opposite, so it would do something sensible like

```
ARGUE
-->I LOVE GREEN PEAS
I HATE GREEN PEAS
I LOVE RED PEAS
```

ARGUE doesn't have any mechanism for dealing with single objects described by more than one word, like ICE CREAM. Perhaps a special way to type these in might be added.

You might want to look at the Madlibs and Sengen projects for more ideas that have to do with taking apart and putting together sentences. You might want to look at the Animal Game project for an example of a program with a different kind of data base that also appears to learn some simple things.

PROGRAM LISTING

```
TO ARGUE
TYPE [\-\-\>]
ARGUEWITH RL
ARGUE
END

TO ARGUEWITH :STATEMENT
PRINT ( SE "I LOVE.HATE SECOND ▸
     :STATEMENT LAST :STATEMENT )
PRINT ( SE "I SECOND :STATEMENT ▸
     OPPOSITE LAST :STATEMENT )
END

TO LOVE.HATE :WORD
IF :WORD = "LOVE [OP "HATE]
IF :WORD = "HATE [OP "LOVE]
END

TO SECOND :LIST
OP FIRST BF :LIST
END

TO OPPOSITE :OBJECT
IF NAMEP :OBJECT [OP THING :OBJECT]
PRINT ( SE [WHAT IS THE OPPOSITE OF] ▸
     :OBJECT "? )
LEARNOPP :OBJECT FIRST RL
OP THING :OBJECT
END

TO LEARNOPP :OBJECT :OPP
MAKE :OBJECT :OPP
MAKE :OPP :OBJECT
END

MAKE "PEPPER "SALT
MAKE "SALT "PEPPER
MAKE "DOGS "CATS
MAKE "CATS "DOGS
MAKE "LIFE "MARRIAGE
MAKE "MARRIAGE "LIFE
MAKE "DARK "LIGHT
MAKE "LIGHT "DARK
MAKE "SUNRISES "SUNSETS
MAKE "SUNSETS "SUNRISES
```

Animal Game

The animal game is a little like twenty questions: you think of an animal, and the game tries to guess it by asking yes-or-no questions.*

What makes the game interesting is that it learns new animals. When it can't guess your animal, it asks you to teach it the animal and its distinguishing characteristic. By learning new questions and new animals, the game gets "smarter."

*This animal game is a popular computer game. It first appeared about ten years ago. Since then many people have implemented it in various computer languages. This Logo program was inspired by Bernard Greenberg's unpublished LISP textbook.

By William Weinreb.

Here's a sample dialogue between the computer and a person playing the animal game. Everything the user types is boldface.

```
?ANIMALGAME
PICK AN ANIMAL, ANY ANIMAL
IS IT FURRY?
YES
HERE'S MY GUESS:  IS IT A CAT?
NO
I GIVE UP.  WHAT IS IT?
A DOG
PLEASE TYPE IN A QUESTION
WHOSE ANSWER IS 'YES' FOR A DOG
AND 'NO' FOR A CAT
DOES IT BARK?
DO YOU WANT TO PLAY AGAIN?
YES
PICK AN ANIMAL, ANY ANIMAL
IS IT FURRY?
MAYBE
PLEASE ANSWER YES OR NO
IS IT FURRY?
YES
DOES IT BARK?
YES
HERE'S MY GUESS:  IS IT A DOG?
YES
I WIN!
I WIN!
DO YOU WANT TO PLAY AGAIN?
NO
?
```

The player's secret animal is "dog."

Here's where the game gets smarter.

The player's secret animal is "dog" again.

Here's where the game asks the question it just learned!

Knowledge Grows on Trees

Below is a diagram of the knowledge the game might have after someone has played it a few times. We call the diagram a *tree*, because it looks something like an upside-down tree.

The tree is made of questions and animal names. Each question has a "yes branch" and a "no branch." Each branch either leads to a question or ends at an animal name.

By drawing what the game knows in the form of a tree, we can get a more vivid picture of how the game works. For example, we can think of the game as exploring the tree from its top. It always starts at the IS

IT FURRY? question. Its goal is to climb down the branches to an animal name. The animal it finally reaches is the one it guesses.

Let's play an imaginary game and trace the game's progress on the tree. Our secret animal is "mouse."

The game's first question is always the question at the tree's top: IS IT FURRY? Since a mouse is furry, we answer yes.

The game follows IS IT FURRY?'s yes branch to the DOES IT BARK? question. From DOES IT BARK?, the game can descend to either of the furry animals, DOG or CAT, but it can no longer reach the unfurry animal, FROG. By descending IS IT FURRY?'s yes branch, the game has narrowed down its possible guesses to furry animals.

The game now asks the question DOES IT BARK?. A mouse does not bark, so we answer no.

The game follows DOES IT BARK?'s no branch to the animal name CAT. When the game reaches an animal name, it guesses that animal. Here, of course, the game's guess is wrong. To improve its chances of guessing right the next time, the game learns the player's secret animal. Before we look at the learning process, let's examine how the game represents its knowledge as lists.

Making Trees with Logo Lists

Consider the very simple tree below. Here we represent it as a list.

```
[[IS IT FURRY?] CAT FROG]
```

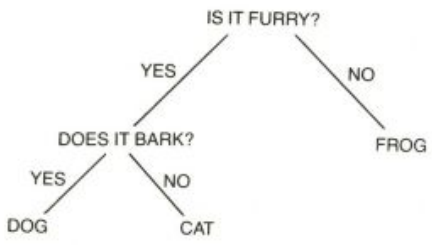
The tree is a list of three elements: a question, the question's yes branch, and the question's no branch. In this case, the question is [IS IT FURRY?], its yes branch is CAT, and its no branch is FROG.

Both branches of the left tree below are animal names. Sometimes, as we've seen, a branch does not lead directly to an animal name but to another question that has its own two branches; it leads, that is, to another tree or *subtree*.

For example, look now at the slightly more complicated tree. Here it is represented as a list.

```
[[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```

This slightly more complicated tree is also a list of three elements: a question, its yes branch, and its no branch. The question is [IS IT FURRY?]; its yes branch is the subtree [[DOES IT BARK?] CAT DOG]; its no branch is the animal name FROG.



Examining Trees

We can write procedures that look at each of a tree's three parts. Sometimes we want to look at a subtree of a tree. Since a subtree is itself a tree, these procedures work on subtrees too. The procedures all expect a list of three elements as input.

```
TO QUESTION :TREE
OP FIRST :TREE
END

TO YES.BRANCH :TREE
OP FIRST BF :TREE
END

TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END
```

Here's an example of how they work.

```
?MAKE 'SAMPLE [[IS IT FURRY?] [[DOES
   IT BARK?] DOG CAT] FROG]
?SHOW QUESTION :SAMPLE
[IS IT FURRY?]
?SHOW YES.BRANCH :SAMPLE
[[DOES IT BARK?] DOG CAT]
?SHOW NO.BRANCH :SAMPLE
FROG
?SHOW NO.BRANCH YES.BRANCH :SAMPLE
CAT
?PR COUNT :SAMPLE
3
?PR COUNT YES.BRANCH :SAMPLE
3
```

Exploring the Game's Knowledge

The animal game's first task is to begin at the tree's top and follow branches to a guess. The procedure that does this is called EXPLORE.

```
TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE
 [EXPLORE YES.BRANCH :TREE]
 [EXPLORE NO.BRANCH :TREE]
END
```

The first line, IF WORDP :TREE [FINISH.UP :TREE STOP], means that if :TREE is a word—that is, an animal name—EXPLORE calls FINISH.UP with the animal name as input and STOPs. If :TREE is not a word, it's a subtree, so EXPLORE follows either its yes branch or its no branch.

Here are two paths EXPLORE can take if its first input is [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]:

```
EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```
The player answers "yes" to IS IT FURRY?
```
   EXPLORE [[DOES IT BARK?] DOG CAT]
```
   The player answers "no" to DOES IT BARK?
```
      EXPLORE "CAT
```
      EXPLORE calls FINISH.UP with CAT as input and STOPs
```
   EXPLORE STOPs
EXPLORE STOPs
EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```
The player answers "no" to IS IT FURRY?
```
   EXPLORE "FROG
```
   EXPLORE calls FINISH.UP with FROG as input and STOPs
```
EXPLORE STOPs
```

No matter what path EXPLORE takes, it always ends at an animal name, which it passes to FINISH.UP.

Guessing and Learning

Guessing

When EXPLORE calls FINISH.UP, the game is ready to guess that FINISH.UP's input (:BEAST) is your animal. FINISH.UP calls GUESS to do the actual guessing. If GUESS outputs TRUE, the game's guess is right, and BRAG is called. If GUESS outputs FALSE, the game's guess is wrong, and LEARN is called.

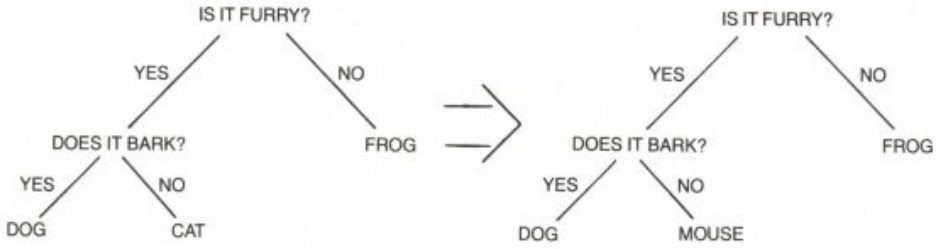
```
TO FINISH.UP :BEAST
IF GUESS :BEAST [BRAG] [LEARN :BEAST [] []]
END

TO GUESS :BEAST
OP YESP (SE  [IS IT] A.OR.AN :BEAST [?])
END

TO BRAG
PR [I WIN!]
PR [I WIN!]
END
```

Learning

LEARN *Adds to the Game's Knowledge*

How does the animal game get smarter? Let's review the imaginary game we played earlier. Our secret animal was "mouse," and the game guessed CAT. Obviously, if the game had guessed "mouse" instead of CAT, it would have won. We might want to change the game so that, from now on, it will guess MOUSE whenever it would have guessed CAT.

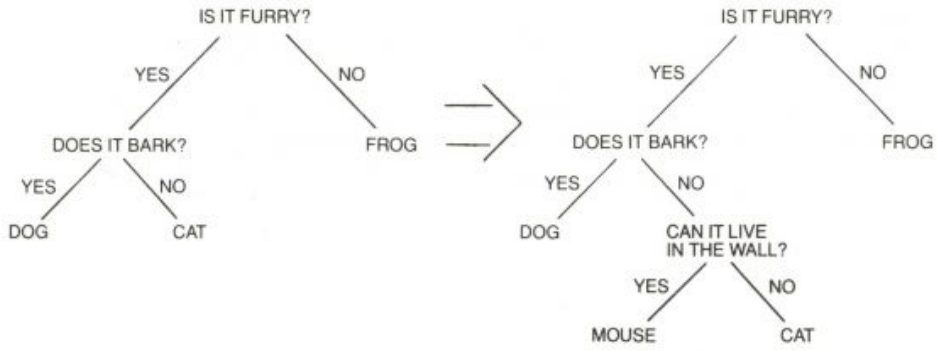
Look at the tree below. To make the game guess MOUSE instead of CAT, we could remove CAT (the wrong guess) from the tree and put MOUSE (the right guess) in its place.



Has the game learned? Not really. We've added a new animal to its knowledge, but we've also subtracted one.

If we want the game's knowledge to include both MOUSE and CAT, we must teach the game a new question, such as CAN IT LIVE IN THE WALL? We also teach it that if a player answers "yes" to the new question, it should guess MOUSE, and if a player answers "no," it should guess CAT.

The next tree shows the result of adding a new animal and a new question to the game's tree. Instead of replacing CAT with MOUSE, we replace CAT with a *new subtree.* The subtree—like all trees—consists of a question ([CAN IT LIVE IN THE WALL?]), a yes branch (MOUSE), and a no branch (CAT).



Building a New Subtree

GET.RIGHT.GUESS and GET.NEW.QUESTION get parts for a new subtree.

```
TO GET.RIGHT.GUESS
PR [I GIVE UP.  WHAT IS IT?]
OP LAST RL
END

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END
```

Adding to the Game's "Tree of Knowledge"

The game's entire "tree of knowledge" is stored in the global variable BIGTREE. For the game to get smarter, the new subtree must be added to :BIGTREE. LEARN and ALTER are the main procedures that do this. ALTER uses 3LIST, which outputs a list of its three inputs.

```
TO LEARN :WRONG.GUESS :RIGHT.GUESS :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER
          :BIGTREE
          :NEW.QUESTION
          :RIGHT.GUESS
          :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS
  [OP 3LIST :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS]
IF WORDP :TREE
  [OP :TREE]
OP 3LIST (QUESTION :TREE)
         (ALTER YES.BRANCH :TREE :NEW.QUESTION
             :RIGHT.GUESS :WRONG.GUESS)
         (ALTER NO.BRANCH :TREE :NEW.QUESTION
             :RIGHT.GUESS :WRONG.GUESS))
END
```

Let's recall how LEARN is called. EXPLORE climbs down to an animal name and passes the animal to FINISH.UP. FINISH.UP calls GUESS to guess the animal. If the guess is right, BRAG is called. If the guess is wrong, LEARN is called.

LEARN has three inputs. When it is called, :WRONG.GUESS is the animal the game guessed, and :RIGHT.GUESS and :NEW.QUESTION are empty lists.

LEARN calls GET.RIGHT.GUESS to get the player's secret animal and stores this animal in :RIGHT.GUESS. It calls GET.NEW.QUESTION to get the player's new yes-or-no question and stores it in :NEW.QUESTION. Then LEARN makes BIGTREE the output from ALTER.

ALTER's four inputs are the game's current "tree of knowledge" and the three parts for the new subtree. ALTER looks through the game's current tree, finds the animal the game guessed, and replaces this wrong guess with the new subtree. It then outputs a new, enlarged "tree of knowledge" to LEARN.

Here's a sample set of inputs to ALTER.

```
:TREE           [[IS IT FURRY?]
                DOES IT BARK?] DOG CAT] FROG]

:NEW.QUESTON    [CAN IT LIVE IN THE WALL?]

:RIGHT.GUESS    MOUSE

:WRONG.GUESS    CAT
```

The following display traces how ALTER works with the preceding inputs. The only input traced is :TREE, since the other inputs are the same each time ALTER is called recursively.

```
ALTER [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
   ALTER [[DOES IT BARK?] DOG CAT]
      ALTER "DOG
      ALTER outputs "DOG
      ALTER "CAT
      ALTER outputs [[CAN IT LIVE IN THE WALL?] MOUSE CAT]
   ALTER outputs [[DOES IT BARK?] DOG [[CAN
                              IT LIVE...?] MOUSE CAT]]
   ALTER "FROG
   ALTER outputs "FROG
ALTER outputs [[IS IT FURRY?] [[DOES IT BARK?] DOG [[CAN
                              IT LIVE...?] MOUSE CAT] FROG]
```

Starting the Game

You begin each session with the animal game by typing ANIMALGAME. This procedure checks whether the game knows anything yet. If no variable named :BIGTREE exists in your workspace, the game knows no questions or animals, so MAKETREE creates a "tree of knowledge" and puts it in :BIGTREE.

PLAY prompts you to think of a secret animal; calls EXPLORE with :BIGTREE as input; and, when the game is over, asks if you'd like to play again.

```
TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?] CAT FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] [PLAY]
END
```

Remember that every time you play the animal game and it loses, :BIGTREE gets "bigger." And the bigger the game's "tree of knowledge," the smarter the game appears to be.

Since :BIGTREE is a global variable, it remains in your workspace after you've finished a session with the animal game (that is, after you answer "no" to the game's question, DO YOU WANT TO PLAY AGAIN?). If you save this workspace, :BIGTREE will be saved as well. At another session, you could make the game's knowledge even bigger.

If you ever want to *erase* the game's knowledge, stop playing the game and call MAKETREE. MAKETREE causes the game to forget everything it has ever learned.

Other Procedures Used by the Game

All these procedures were mentioned earlier but we did not look at how they work.

The input to A.OR.AN should be an animal name. Its output is the animal name preceded by an appropriate article—either "a" or "an."

```
TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] [OP SE "AN :ANIMAL]
OP SE "A :ANIMAL
END
```

YESP and COMPLAIN get a yes-or-no answer to a question. The question is the input to YESP.

```
TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP :ANS [NO])
  [COMPLAIN OP YESP :QUESTION]
OP EQUALP :ANS [YES]
END

TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END
```

Here's an example.

```
?PR YESP [IS IT FURRY?]
IS IT FURRY?
SORT OF
PLEASE ANSWER YES OR NO
IS IT FURRY?
YES
TRUE
?
```

3LIST outputs a list of its three inputs.

```
TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END
```

SUGGESTIONS

You can play this game with exotic animal names such as armadillo, gnu, gazelle, iguana. You could even use fantastic animals like centaurs or pushme-pullyous. Some people say that it's most fun to play it with the names of your friends!

PROGRAM LISTING

```
TO QUESTION :TREE
OP FIRST :TREE
END

TO YES.BRANCH :TREE
OP FIRST BF :TREE
END

TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END

TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE [EXPLORE ►
     YES.BRANCH :TREE] [EXPLORE ►
     NO.BRANCH :TREE]
END

TO FINISH.UP :BEAST
IF GUESS :BEAST [BRAG] [LEARN :BEAST ►
     [] []]
END

TO GUESS :BEAST
OP YESP (SE  [IS IT] A.OR.AN :BEAST ►
     [?])
END

TO BRAG
PR [I WIN!]
PR [I WIN!]
END

TO GET.RIGHT.GUESS
PR [I GIVE UP.  WHAT IS IT?]
OP LAST RL
END

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] ►
     :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END

TO LEARN :WRONG.GUESS :RIGHT.GUESS ►
     :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER :BIGTREE ►
     :NEW.QUESTION :RIGHT.GUESS ►
     :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION ►
     :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS [OP 3LIST ►
     :NEW.QUESTION :RIGHT.GUESS ►
     :WRONG.GUESS]
IF WORDP :TREE [OP :TREE]
OP 3LIST (QUESTION :TREE) (ALTER ►
     YES.BRANCH :TREE :NEW.QUESTION ►
     :RIGHT.GUESS :WRONG.GUESS) (ALTER ►
     NO.BRANCH :TREE :NEW.QUESTION ►
     :RIGHT.GUESS :WRONG.GUESS))
END

TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?] CAT ►
     FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] ►
     [PLAY]
END

TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] ►
     [OP SE "AN :ANIMAL]
OP SE "A :ANIMAL
END
```

```
TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP ▸
     :ANS [NO]) [COMPLAIN OP YESP ▸
     :QUESTION]
OP EQUALP :ANS [YES]
END
```

```
TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END

TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END

MAKE "BIGTREE [[IS IT FURRY?] CAT ▸
     FROG]
```

Dictionary

The idea for this project came about while I was hiking with some friends. During our climb up the mountain, we tried to stump each other by asking the meaning of unusual words. I began to think about developing a dictionary project using Logo.

I wanted to be able to do several things with my dictionary:

- Add a new word and its definition.
- Print the definition of a word.
- Remove a word and its definition.
- Print the entire dictionary.

The Dictionary

My first task was to decide how to store the words. I decided that the dictionary would be a list of entries. Each entry would be a list composed of a word and its definition. Here are two examples.

```
[ICE [FROZEN WATER]]
```

or

```
[HAT [COVERING FOR HEAD]]
```

I named the dictionary ENTRY.LIST. Here's how I created it.

```
MAKE "ENTRY.LIST [[EGREGIOUS [CONSPICUOUSLY BAD]]
  [PROSY [COMMONPLACE]]
  [AUTO-DA-FE [BURNING OF A HERETIC]]]
```

Using the Dictionary

When you type DICTIONARY, the following is printed on your screen:

By Susan Cotten.

```
WELCOME TO THE DICTIONARY.

HERE ARE THE COMMANDS:

TYPE A - TO ADD NEW ENTRY
TYPE D - TO PRINT DEFINITION OF WORD
TYPE P - TO PRINT DICTIONARY
TYPE Q - TO QUIT
TYPE R - TO REMOVE ENTRY
TYPE ? - TO PRINT COMMANDS

TYPE COMMAND.
>
```

DICTIONARY calls INIT, which checks to see if you already have a dictionary. If you do not, INIT creates one.

```
TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS:]
DO.CHOICE "?
END

TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST [MAKE "ENTRY.LIST
  [[EGREGIOUS [CONSPICUOUSLY BAD]]
  [PROSY [COMMONPLACE]]
  [[AUTO-DA-FE] [BURNING OF A HERETIC]]]]
END
```

DO.CHOICE has the job of figuring out whether the character you type matches one of the expected commands. If there is no match or if you type ?, DO.CHOICE prints the list of possible choices.

```
TO DO.CHOICE :LTR
PR []
IF EQUALP "A :LTR [ADD.ENTRY]
IF EQUALP "D :LTR [PRINT.DEFINITION]
IF EQUALP "P :LTR [PRINT.DICTIONARY]
IF EQUALP "Q :LTR [STOP]
IF EQUALP "R :LTR [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">
MAKE "LTR RC
PR :LTR
DO.CHOICE :LTR
END
```

```
TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END
```

Adding a New Word and Definition

To add a word, you type A while running DICTIONARY. Here's an example of what happens.

```
TYPE NEW WORD.
FLUMP
TYPE DEFINITION OF NEW WORD.
DROP OR MOVE HEAVILY

TYPE COMMAND
>
```

If you try to add a word that is already in the dictionary, this happens:

```
TYPE NEW WORD
EGREGIOUS
EGREGIOUS IS ALREADY IN DICTIONARY.
```

ADD.ENTRY is the procedure that lets you add a new entry to the dictionary.

```
TO ADD.ENTRY
PR [TYPE NEW WORD.]
ADD.ENTRY1 FIRST RL
END
```

ADD.ENTRY1 calls GET.ENTRY to see if the word you want to add is already in the dictionary. If the word is not in the dictionary, then the word and its definition become a new entry.

```
TO ADD.ENTRY1 :WRD
IF NOT EMPTYP GET.ENTRY :WRD :ENTRY.LIST
  [PR SE :WRD [IS ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END
```

GET.ENTRY has the task of finding an entry in the dictionary. It does this by attempting to match an input word with the first word in each entry.

```
TO GET.ENTRY :WRD :LST
IF EMPTYP :LST [OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END
```

MAKE.ENTRY adds a new entry to the dictionary.

```
TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY :ENTRY.LIST
END
```

Printing the Definition of a Word

This is what happens when you type D.

```
TYPE WORD WHOSE DEFINITION
YOU WANT PRINTED.
EGREGIOUS
[CONSPICUOUSLY BAD]
```

PRINT.DEFINITION calls PRINT.DEF1 to print out the definition of a word.

```
TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END

TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD :ENTRY.LIST
END
```

PRINT.DEF1 then calls PRINT.DEF2 with the word to be defined and its entry in the dictionary. If the entry is in the dictionary, PRINT.DEF2 prints the definition.

```
TO PRINT.DEF2 :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN DICTIONARY.] STOP]
PR BF :LST
END
```

Removing an Entry from the Dictionary

To remove an entry, you type R. Here is an example.

```
TYPE WORD
YOU WANT TO REMOVE.
FLUMP
```

REMOVE.ENTRY uses REMOVE to output a dictionary, minus the unwanted entry.

```
TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL :ENTRY.LIST
END
```

```
TO REMOVE :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF :LST]
OP FPUT FIRST :LST REMOVE :WRD BF :LST
END
```

Printing the Dictionary

Here's what happens when you type P. I've added some words that I thought were interesting to the dictionary.

```
IMPUISSANT
[WEAK; IMPOTENT]

ACCRETIVE
[ADDING IN GROWTH]

DENTILOQUY
[THE ACT OR HABIT OF SPEAKING WITH TEETH CLOSED]

CENOSITY
[FILTHINESS; SQUALOR]

DELIQUESCE
[TO MELT AWAY]

FETOR
[STRONG OFFENSIVE SMELL]

BRUMAL
[INDICATIVE OF OR OCCURRING IN WINTER]

**TYPE ANY CHARACTER
TO SEE MORE**
```

Note: At this point you press any key to see the next seven (or remaining) entries.

```
EGREGIOUS
[CONSPICUOUSLY BAD]

PROSY
[COMMONPLACE]

AUTO - DA - FE
[BURNING OF A HERETIC]
```

The procedures PRINT.DICTIONARY, FORMAT, and PRINT.ENTRY work together to print ENTRY.LIST in an easy-to-read format. There is room on the screen for seven entries. FORMAT counts the number of entries. When the screen is full, FORMAT pauses and waits until you type any character before printing the next seven or remaining entries.

```
TO PRINT.DICTIONARY
FORMAT Ø :ENTRY.LIST
END

TO FORMAT :CTR :LST
IF EMPTYP :LST [STOP]
IF :CTR = 7
  [PR [**TYPE ANY CHARACTER]
   PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF :LST
END

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END
```

PROGRAM LISTING

```
TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS:]
DO.CHOICE "?
END

TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST [MAKE ►
     "ENTRY.LIST [[EGREGIOUS ►
     [CONSPICUOUSLY BAD]] [PROSY
     [COMMONPLACE]] [[AUTO-DA-FE] ►
     [BURNING OF A HERETIC]]]]
END

TO DO.CHOICE :LTR
PR []
IF EQUALP "A :LTR [ADD.ENTRY]
IF EQUALP "D :LTR [PRINT.DEFINITION]
IF EQUALP "P :LTR [PRINT.DICTIONARY]
IF EQUALP "Q :LTR [STOP]
IF EQUALP "R :LTR [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] ►
     [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">
MAKE "LTR RC
PR :LTR
DO.CHOICE :LTR
END

TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF ►
     WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END

TO ADD.ENTRY
PR [TYPE NEW WORD.]
ADD.ENTRY1 FIRST RL
END

TO ADD.ENTRY1 :WRD
IF NOT EMPTYP GET.ENTRY :WRD ►
     :ENTRY.LIST [PR SE :WRD [IS ►
     ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END

TO GET.ENTRY :WRD :LST
IF EMPTYP :LST [OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP ►
     FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END
```

```
TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY ▸
     :ENTRY.LIST
END

TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END

TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD ▸
     :ENTRY.LIST
END

TO PRINT.DEF2 :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN ▸
     DICTIONARY.] STOP]
PR BF :LST
END

TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL ▸
     :ENTRY.LIST
END

TO REMOVE :WRD :LST
IF EMPTYP :LST [PR SE :WRD [IS NOT IN ▸
     DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF ▸
     :LST]
OP FPUT FIRST :LST REMOVE :WRD BF :LST
END
```

```
TO PRINT.DICTIONARY
FORMAT 0 :ENTRY.LIST
END

TO FORMAT :CTR :LST
IF EMPTYP :LST [STOP]
IF :CTR = 7 [PR [**TYPE ANY CHARACTER] ▸
     PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF ▸
     :LST
END

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END

MAKE "ENTRY.LIST [[IMPUISSANT [WEAK; ▸
     IMPOTENT]] [ACCRETIVE [ADDING IN ▸
     GROWTH]] [DENTILOQUY [THE ACT OR ▸
     HABIT OF SPEAKING WITH TEETH ▸
     CLOSED]] [CENOSITY [FILTHINESS; ▸
     SQUALOR]] [DELIQUESCE [TO MELT ▸
     AWAY]] [FETOR [STRONG OFFENSIVE ▸
     SMELL]] [BRUMAL [INDICATIVE OF OR ▸
     OCCURRING IN WINTER]] [EGREGIOUS ▸
     [CONSPICUOUSLY BAD]] [PROSY ▸
     [COMMONPLACE]] [[AUTO - DA - FE] ▸
     [BURNING OF A HERETIC]]]
```

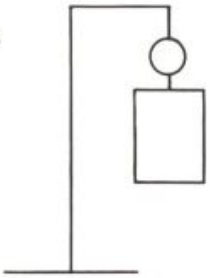
Hangman

HANGMAN is based on the popular two-person pencil-and-paper game in which one player thinks up a secret word and the other player tries to discover what the word is by guessing what letters are in the word. A gallows is drawn, and for each incorrect guess, part of a stick figure is added to the drawing. The player who is guessing wins the game by guessing the entire word before the stick figure is completed.

By Brian Harvey.

In this version, the program chooses the secret word and you do the guessing. At each turn, you can guess either a single letter or the entire word.

Here is a picture of a game in progress.



```
-A--E-
GUESSES: E T A O I
YOUR GUESS?
```

The secret word is shown as -A--E-. This means that it has six letters, two of which have been guessed. You have made five guesses. A and E were correct. The others, T, O, and I, were wrong. Because of these three wrong guesses, the program has drawn the head, neck, and body of the person being hanged. If you make more wrong guesses, the program will draw the person's arms and legs.

I like this program because it combines text processing with graphics. The top-level procedure divides the program into two parts: setting up and playing the game.

```
TO HANGMAN
SETUP
PLAY
END
```

Setting Up

SETUP has two jobs; it gives initial values to certain variables, including the secret word, and it draws the gallows.

```
TO SETUP
MAKE "MYWORD PICKWORD
MAKE "GUESSES []
MAKE "WON "FALSE
MAKE "SPACES "
REPEAT 18 [MAKE "SPACES WORD :SPACES CHAR 32]
MAKE "GOTTEN 0
MAKE "TRIES 7
CT
GALLOWS
END
```

SETUP uses two main subprocedures, one to pick the secret word and one to draw the gallows. PICKWORD outputs the secret word, which SETUP remembers in the global variable MYWORD. To choose the word from a list of possible words, PICKWORD uses the procedures PICK and ITEM, which appear as examples in the *Atari Logo Reference Manual.*

```
TO PICKWORD
OP PICK [POTSTICKER COMPUTER IRAQ GAZEBO THRUSH STYLE FOILED
        SWARM ZEBRA AWFUL WILY YELLOW BARKED STOIC]
END
```

GALLOWS positions a turtle for drawing the gallows, sets the pen down, and uses GALL1 to do the actual drawing. The reason to make GALL1 a subprocedure is that it will be used again, with the eraser down, to erase the gallows if you win by guessing the word.

```
TO GALLOWS
TELL [0 1 2 3]
CS HT
TELL 0
PU SETPOS [-40 -60]
RT 90
PD
GALL1
END

TO GALL1
FD 80
BK 40
LT 90
FD 170
RT 90
FD 60
RT 90
FD 20
END
```

Variables Created by SETUP

The variable MYWORD is one of several that are used throughout the hangman program to keep track of the progress of the game. For example, the program must remember what letters have been guessed and how many wrong guesses are allowed before you lose. Several of these variables are given their initial values by SETUP.

MYWORD	The secret word.
GUESSES	A list of the letters you have guessed.
WON	TRUE if you win by guessing the word or the last missing letter in it.
SPACES	A word of eighteen spaces, which is typed to erase messages from the program in the text part of the screen.
GOTTEN	The number of letters in the secret word that you have guessed correctly. (If a letter occurs more than once in the secret word, the number of letters guessed correctly may be more than the number of correct guesses you have made, because one correct guess may reveal several letters in the word.)
TRIES	The number of incorrect guesses remaining before you lose.

Playing the Game

The central part of the hangman program is the procedure PLAY and its subprocedure GETGUESS, which is called each time you make a guess.

```
TO PLAY
IF :TRIES=0 [LOSE STOP]
GETGUESS
IF :WON [SETCURSOR [0 23] STOP]
PLAY
END

TO GETGUESS
DISPLAY
MAKE "GUESS FIRST RL
CLEARMESSAGE
IF (COUNT :GUESS) > 1 [TESTWORD STOP]
IF MEMBERP :GUESS :GUESSES [ALREADY GETGUESS STOP]
MAKE "GUESSES SE :GUESSES :GUESS
IF MEMP :GUESS :MYWORD [FIXGOT :GUESS :MYWORD] [BADTRY]
IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
END
```

PLAY calls GETGUESS repeatedly, checking between times to see if you've won (the variable WON made TRUE) or lost (no more TRIES left). GETGUESS uses several subprocedures to display the current state of the game, read a guess from the keyboard, and test the guess. A guess can be either a single letter or the entire word. These cases are distinguished by checking the COUNT of the guess; if it's more than one letter, the procedure TESTWORD is used to compare the guess to the secret word. Otherwise, the program checks if you have already guessed the letter; if not, it checks to see if the guessed letter is actually in the word. If the letter is in the word, FIXGOT is called to update the number of letters correctly guessed. If not, BADTRY draws another piece of the body under the gallows.

Keeping Track of the Text Screen

The text part of the screen in the middle of a game might look like this:

```
-A-E--                YOU GUESSED THAT!
GUESSES: E T A
YOUR GUESS? _
```

In the top left corner is the display of the secret word, with some letters already guessed and the others indicated by hyphens. In the top right corner is the *message area.* You have just repeated a guess already made, and the program has complained about it. The next line shows the list of letters already guessed. The third line invites you to make another guess, and the cursor is positioned for reading that guess.

The message area is maintained by the procedure SAY. Two subprocedures of GETGUESS show simple examples of how SAY is used:

```
TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END
```

```
TO ALREADY
SAY [YOU GUESSED THAT!]
END

TO CLEARMESSAGE
SAY :SPACES
END
```

(The underlining in this listing represents inverse-video characters on the screen.) The CLEARMESSAGE procedure types spaces into the message area, erasing any leftover messages. The procedure ALREADY is called by GETGUESS if you repeat a previous guess.

The rest of the text screen, apart from the message area, is maintained by the DISPLAY procedure:

```
TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END
```

For each letter of the secret word, DISWORD looks in the list of letters already guessed. If this letter has been guessed, DISWORD types it. If not, DISWORD types a hyphen.

```
TO DISWORD :WORD
IF EMPTYP :WORD [STOP]
IF MEMBERP FIRST :WORD :GUESSES [TYPE FIRST :WORD] [TYPE "-]
DISWORD BF :WORD
END
```

When You Guess a Letter

When you guess a letter (that hasn't been guessed already), GETGUESS calls either FIXGOT or BADTRY depending on whether the guess is correct or incorrect. To test the correctness of the guess, GETGUESS uses MEMP, which is like the primitive MEMBERP except that it checks whether a letter is an element of a word, instead of whether a word is an element of a list.

```
TO MEMP :LETTER :WORD
IF EMPTYP :WORD [OP "FALSE]
IF EQUALP :LETTER FIRST :WORD [OP "TRUE]
OP MEMP :LETTER BF :WORD
END
```

(Actually, MEMP would work equally well testing for membership in a list, like MEMBERP, but we need it only to check for membership in a word.)

If the guess is correct, the task of FIXGOT is to calculate a new value

for the variable GOTTEN, which counts the number of correctly guessed letters in the secret word. We can't just add 1 to GOTTEN, because the letter you guessed may appear more than once in the secret word. For example, if the secret word is "thrush" and you guess H, FIXGOT must add 2 to GOTTEN. So FIXGOT must examine each letter of the secret word.

```
TO FIXGOT :GUESS :WORD
IF EMPTYP :WORD [STOP]
IF EQUALP :GUESS FIRST :WORD [MAKE "GOTTEN :GOTTEN+1]
FIXGOT :GUESS BF :WORD
END
```

Note that FIXGOT does not actually display the newly guessed letters on the screen. This will be done by DISPLAY the next time through GETGUESS.

What Happens on a Wrong Guess

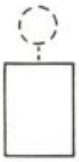
If the guess is incorrect, BADTRY is called to count down the number of turns until you lose and to draw part of the body under the gallows:

```
TO BADTRY
RUN SE WORD "DRAW :TRIES []
MAKE "TRIES :TRIES-1
END
```

The RUN command is used to select a subprocedure to draw the appropriate part of the body, based on the number of tries remaining. For example, the variable TRIES is initially 7, and the procedure DRAW7 draws a head. DRAW6 draws the neck, DRAW5 the torso, DRAW4 and DRAW3 the arms, and DRAW2 and DRAW1 the legs:

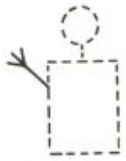
```
TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END
```

```
TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END
```

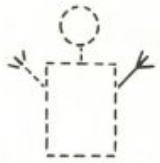


```
TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END
```

```
TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END
```

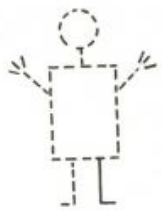


```
TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END

TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END
```



```
TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END
```

```
TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END
```



None of the procedures DRAW1, DRAW2, and so forth, assume that the turtle is at any particular position. This is because if you win, the program will erase the gallows and then finish drawing in the body, so any of these

procedures might be called with the turtle at the end of the gallows, rather than at the end of the previous body part.

When You Guess a Word

We have looked at the procedures that deal with a guess of a single letter. You may also guess the entire word; if so, the GETGUESS procedure calls TESTWORD.

```
TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY [NOPE!] BADTRY]
END
```

An incorrect guess of the entire word is handled by BADTRY, just like an incorrect guess of a letter. But if you guess the entire word correctly, there is no need to call FIXGOT. We can simply call WIN, because you have won the game.

When You Lose the Game

We have now looked at all the procedures involved in playing the game, up to the point of winning or losing. The case of losing is easier to understand. You lose by running out of tries. This means that the entire body has already been drawn.

```
TO LOSE
SAY [YOU LOSE!!  SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END
```

```
TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END
```

```
TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END
```

```
TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END
```

The program tells you what the secret word was, moves the cursor down to the last screen line, and fills in the already-drawn head with a frowning face. When the program stops, Logo will print its prompt on the last line without obscuring what is written in the text area. (LOSE is called only by PLAY, which then stops, returning to HANGMAN, which stops. So when LOSE stops, the entire program is done.)

When You Win the Game

What if you win? In this case, the body is not yet entirely drawn. We want to erase the gallows, finish drawing the body, notify the winner, and stop the program.

```
TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END
```

UNGALL is like GALLOWS except that it draws the gallows in PE (penerase). FINISH calls each of the yet-undone drawing procedures (DRAW1, etc.) to

finish drawing the body. And SMILE draws the same mouth as FROWN, but right-side up.

Unlike LOSE, the WIN procedure can be called from two places in the program: TESTWORD and GETGUESS. Because these places are deeper in the chain of subprocedures, we must set the variable WON so that the PLAY procedure can test it, to know when to stop the game program.

Utilities

To complete the listing of procedures used in this project, here are the utility procedures PICK and ITEM:

```
TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END
```

PROGRAM LISTING

```
TO HANGMAN
SETUP
PLAY
END

TO SETUP
MAKE "MYWORD PICKWORD
MAKE "GUESSES []
MAKE "WON "FALSE
MAKE "SPACES ""
REPEAT 18 [MAKE "SPACES WORD :SPACES ►
     CHAR 32]
MAKE "GOTTEN 0
MAKE "TRIES 7
CT
GALLOWS
END

TO GALLOWS
TELL [0 1 2 3]
CS HT
TELL 0
PU SETPOS [-40 -60]
RT 90
PD
GALL1
END

TO GALL1
FD 80
BK 40
LT 90
FD 170
RT 90
FD 60
RT 90
FD 20
END

TO PLAY
IF :TRIES=0 [LOSE STOP]
GETGUESS
IF :WON [SETCURSOR [0 23] STOP]
PLAY
END

TO GETGUESS
DISPLAY
MAKE "GUESS FIRST RL
CLEARMESSAGE
IF (COUNT :GUESS) > 1 [TESTWORD STOP]
IF MEMBERP :GUESS :GUESSES [ALREADY ►
     GETGUESS STOP]
MAKE "GUESSES SE :GUESSES :GUESS
IF MEMP :GUESS :MYWORD [FIXGOT :GUESS ►
     :MYWORD] [BADTRY]
IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
END
```

```
TO PICKWORD
OP PICK [POTSTICKER COMPUTER IRAQ ▸
     GAZEBO THRUSH STYLE FOILED SWARM ▸
     ZEBRA AWFUL WILY YELLOW BARKED ▸
     STOIC]
END

TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END

TO ALREADY
SAY [YOU GUESSED THAT!]
END

TO CLEARMESSAGE
SAY :SPACES
END

TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END

TO DISWORD :WORD
IF EMPTYP :WORD [STOP]
IF MEMBERP FIRST :WORD :GUESSES [TYPE ▸
     FIRST :WORD] [TYPE "-]
DISWORD BF :WORD
END

TO MEMP :LETTER :WORD
IF EMPTYP :WORD [OP "FALSE]
IF EQUALP :LETTER FIRST :WORD [OP ▸
     "TRUE]
OP MEMP :LETTER BF :WORD
END

TO FIXGOT :GUESS :WORD
IF EMPTYP :WORD [STOP]
IF EQUALP :GUESS FIRST :WORD [MAKE ▸
     "GOTTEN :GOTTEN+1]
FIXGOT :GUESS BF :WORD
END

TO BADTRY
RUN SE WORD "DRAW :TRIES []
MAKE "TRIES :TRIES-1
END

TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END

TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END

TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END

TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END

TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END

TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END
```

```
TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END

TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END

TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY ►
     [NOPE!] BADTRY]
END

TO LOSE
SAY [YOU LOSE!!  SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END

TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END

TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END

TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END

TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END

TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END
```

Math: A Sentence Generator

When we think of computers making up sentences, we most often think of them making up English or French sentences. We rarely think of them making up math sentences. This project is about developing a math sentence generator. It is set in the context of developing an interactive program. A sentence is made up in the form `3 + X = 5` and the user is asked `WHAT IS X?`.

The first example involves only addition sentences. Then the program is modified to include multiplication, subtraction, and division. Later the program is changed once more to vary the form of the math sentences and keep track of the number of times the user responds to the same question.

I boldface what the user types.

```
?MATH
6 + X = 7
WHAT IS X? 1
RIGHT

7 + X = 16
WHAT IS X? 5
NOPE, X IS 9

7 + X = 10
WHAT IS X? 3
RIGHT
```

As the example shows, `MATH` makes addition sentences of the form `2 + X = 3` and not of the form `X + 2 = 3`. Later we will change `MATH` so that it uses both forms.

`MATH` randomly chooses two of the integers to be used in the math sentence. `ADD` then presents the addition problem and checks on your answer. The numbers `MATH` chooses are less than ten, but you can easily adjust the procedure and make the numbers larger.

```
TO MATH
ADD RANDOM 10 RANDOM 10
PR []
MATH
END

TO ADD :NUM1 :NUM2
PR (SE :NUM1 [+ X =] :NUM1 + :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

In the addition sentences, the value of `X` is `:NUM2`, which is the second input to `ADD`. The sum of the two inputs is computed by `ADD`.

There are different ways to expand this program. You could design the

---

By Cynthia Solomon.

program so that it gives you three chances to get the answer right. You could expand the program so that it gives you problems in subtraction, division, and multiplication. You could make it keep track of the number of problems you do and the number you respond correctly to. You might decide to help the user. Some of these suggestions are explored in the next section.

Making MATH *Subtract, Multiply, and Divide*

One way to extend MATH is to make three more procedures, SUBTRACT, MULTIPLY, and DIVIDE.

```
TO SUBTRACT :NUM1 :NUM2
PR (SE :NUM1 [- X =] :NUM1 - :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END

TO MULTIPLY :NUM1 :NUM2
PR (SE :NUM1 [* X =] :NUM1 * :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END

TO DIVIDE :NUM1 :NUM2
PR (SE :NUM1 [/ X =] :NUM1 / :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

Try these procedures to see if there are any bugs. Modifying MATH is a good way to try these new procedures.

```
TO MATH
ADD RANDOM 10 RANDOM 10
SUBTRACT RANDOM 10 RANDOM 10
MULTIPLY RANDOM 10 RANDOM 10
DIVIDE RANDOM 10 RANDOM 10
PR []
MATH
END
```

```
MATH
4 + X = 7
WHAT IS X? 3
RIGHT
3 - X = 4
WHAT IS X? 1
NOPE, X IS -1
6 * X = 18
WHAT IS X? 3
RIGHT
3 / X = 1
WHAT IS X? 3
RIGHT
```

What do you think? The program seems to work, but there are some possible problems. For example, in the subtraction sentences X might have a negative value. Perhaps you want to use this program without negative numbers for answers. We can adjust SUBTRACT so that the value of X is always positive.

Notice that the sentences are of the form 3 - X = 2. The form X - 3 = 4 might be easier to solve, and so you might want to make sentences in that form.

There is a potential bug with multiplication and division. For example, division by 0 will cause Logo to stop the program and print out an error message. Attempts to divide by 0 must be prevented. One way to make sure of this is to add one to the random number used as DIVIDE's second input. Multiplication by 0 can cause a different sort of problem when you try to figure out what 0 * X is.

Although the preceding examples do not show X being a fractional number like .5, it is possible. You might want to guard against that happening. Since the sentences are generated by the program, we can make sure that the computation is performed so that X is always a whole number.

In the next section MATH is extended to include some of these ideas. The procedures are rewritten. A new procedure is introduced called ANSWER. It is used by ADD, MULTIPLY, SUBTRACT, and DIVIDE to print out the sentence and get the user's response to what X is.

Extending MATH

In this section, the first extensions to MATH guard against multiplication or division by 0 and give the user three chances to figure out what X is. All math sentences are still written in the form 3 + X = 5 and expect integer answers. The program generates two random numbers and then computes a third. Here is an example of the program in action.

```
?MATH
HERE ARE SOME MATH PROBLEMS.
8 + X  = 13
WHAT IS X?
```

Now if you type 5, Logo responds:

```
RIGHT ON
```

If you type anything else, Logo responds:

```
TRY AGAIN

8 + X = 13
WHAT IS X?
```

You are given three tries to get the answer. If you are still wrong, Logo responds:

```
NOPE, X IS 5
```

MATH has MATH1 present sentences in subtraction, multiplication, and division as well as addition.

```
TO MATH
TS
CT
PR [HERE ARE SOME MATH PROBLEMS.]
MATH1
END

TO MATH1
ADD RANDOM 11 RANDOM 11
SUBTRACT RANDOM 11 RANDOM 11
MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
DIVIDE 1 + RANDOM 10 1 + RANDOM 10
PR [] WAIT 60
MATH1
END
```

After ADD computes :RESULT, ANSWER takes over the job of printing out the sentence and checking the user's response.

```
TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
ANSWER [+ X =] 1
END
```

ADD gives ANSWER the form [+ X =] as its first input. The second input represents the number of times the user responds to the question WHAT IS X?. :NUM1, :NUM2, and :RESULT are used by ANSWER's subprocedures ANSWER1 and GETINP. The variables are not given as inputs to ANSWER or its subprocedures. As far as these procedures are concerned, these are global variables. The value of X is still :NUM2.

ANSWER prints the mathematical sentence with the help of ANSWER1. After the sentence is printed, ANSWER asks for the value of X. It then turns the job over to GETINP along with the user's response.

```
TO ANSWER :PHRASE :TIMES
PR []
ANSWER1 :PHRASE
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END
```

(\ is the way to quote special characters like *space*. ANSWER prints two spaces after the question mark.) GETINP plays an important role. It determines what to do next. If :INP is empty, GETINP assumes this is the user's signal to do something else and so calls MATH1. If :INP is not the same as :NUM2, then GETINP calls ANSWER adding 1 to :TIMES, unless this is the user's third try. On the third try GETINP gives the answer.

```
TO GETINP :INP
IF EMPTYP :INP [MATH1 STOP]
IF :NUM2 = FIRST :INP [PR [RIGHT ON] STOP]
IF :TIMES = 3 [PR SE [NOPE, X IS] :NUM2  STOP]
PR [TRY AGAIN]
ANSWER :PHRASE :TIMES + 1
END
```

The MULTIPLY procedure is similar to ADD in structure.

```
TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
ANSWER [* X =] 1
END
```

A couple of tricks are used here so that ANSWER will work for MULTIPLY, DIVIDE, and SUBTRACT. :NUM2 is always the value of X. :NUM1 is always on the left side of the equals sign and :RESULT is always on the right of the equals sign. What does change is which of these numbers are inputs to a procedure and which are computed in the procedure. For example, SUBTRACT computes the value of NUM1 while :RESULT and :NUM2 are inputs. But the value of X is still :NUM2.

```
TO SUBTRACT :RESULT :NUM2
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END
```

DIVIDE makes sure that the value of X is always an integer by shifting the role of its first input, which becomes :RESULT. DIVIDE is given :RESULT and computes :NUM1.

```
TO DIVIDE :RESULT :NUM2
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END
```

Extensions

There are many modifications you might want to make to this kind of program. The modification I chose is to allow sentences to be in either of two forms.

```
3 + X = 5
X + 3 = 5
```

The changed procedures follow. Notice that the decision as to which form to use is based on whether RANDOM 2 outputs 0 or 1.

```
TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
ANSWER [+ X =] 1
END
```

```
TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
ANSWER [* X =] 1
END
```

The new forms for ADD and MULTIPLY are

```
X + 3 = 5
X * 3 = 6
```

where the value of X is still :NUM2.

When SUBTRACT generates a sentence in the form 6 - X = 2, its inputs, :RESULT and :NUM2, are added together to be :NUM1. In the example 6 - X = 2, :NUM1 is 6 and :RESULT is 2.

When the sentence is in the form X - 4 = 2, then SUB2 computes :NUM2 by adding the inputs :RESULT and :NUM1. In this case :RESULT is 2 and :NUM1 is 4.

```
TO SUBTRACT :RESULT :NUM2
IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

TO SUB2 :RESULT :NUM1
MAKE "NUM2 :RESULT + :NUM1
ANSWER [X -] 1
END
```

DIVIDE computes :NUM1 as :RESULT * :NUM2 when the form is 6 / X = 2. DIVIDE computes :NUM2 as :NUM1 * :RESULT when the form is X / 3 = 2.

```
TO DIVIDE :RESULT :NUM2
IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

TO DIV2 :RESULT :NUM1
MAKE "NUM2 :RESULT * :NUM1
ANSWER [X /] 1
END
```

ANSWER needed to be changed as well.

```
TO ANSWER :PHRASE :TIMES
PR []
IF "X = FIRST :PHRASE [ANSWER2 :PHRASE] [ANSWER1 :PHRASE]
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END

TO ANSWER2 :PHRASE
PR (SE :PHRASE :NUM1 "= :RESULT)
END
```

PROGRAM LISTING

```
TO MATH
TS
CT
PR [HERE ARE SOME MATH PROBLEMS.]
MATH1
END

TO MATH1
ADD RANDOM 11 RANDOM 11
SUBTRACT RANDOM 11 RANDOM 11
MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
DIVIDE 1 + RANDOM 10 1 + RANDOM 10
PR [] WAIT 60
MATH1
END

TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
ANSWER [+ X =] 1
END

TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
ANSWER [* X =] 1
END

TO SUBTRACT :RESULT :NUM2
IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 ▸
     STOP]
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

TO SUB2 :RESULT :NUM1
MAKE "NUM2 :RESULT + :NUM1
ANSWER [X -] 1
END

TO DIVIDE :RESULT :NUM2
IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 ▸
     STOP]
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

TO DIV2 :RESULT :NUM1
MAKE "NUM2 :RESULT * :NUM1
ANSWER [X /] 1
END

TO ANSWER :PHRASE :TIMES
PR []
IF "X = FIRST :PHRASE [ANSWER2 ▸
     :PHRASE] [ANSWER1 :PHRASE]
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END

TO ANSWER2 :PHRASE
PR (SE :PHRASE :NUM1 "= :RESULT)
END

TO GETINP":INP
IF EMPTYP :INP [MATH1 STOP]
IF :NUM2 = FIRST :INP [PR [RIGHT ON] ▸
     STOP]
IF :TIMES = 3 [PR SE [NOPE, X IS] ▸
     :NUM2  STOP]
PR [TRY AGAIN]
ANSWER :PHRASE :TIMES + 1
END
```

Number Speller

```
?PRINT SPELL 1427
ONE THOUSAND FOUR HUNDRED TWENTY SEVEN
?
```

This program takes a whole number as input and outputs the number spelled out in words.

The general idea is to divide the number into groups of three digits. For example, the number 1234567890 is 1 billion, 234 million, 567 thousand, 890. For each such group we must spell out its three-digit number and also find the word (like "million") that indicates the position of that group in the entire number.

Spelling a Group of Three

Let's start by writing a procedure, SPELL.GROUP, that spells out a number of up to three digits.

```
TO SPELL.GROUP :GROUP
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST :GROUP "HUNDRED
        SPELL.GROUP BF :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF :GROUP]
IF AND :GROUP>10 :GROUP<20 [OUTPUT TEEN :GROUP-10]
OUTPUT SE (IF :GROUP>9 [TENS FIRST :GROUP] [[]])
        (IF 0<LAST :GROUP [DIGIT LAST :GROUP] [[]])
END
```

```
?PRINT SPELL.GROUP 425
FOUR HUNDRED TWENTY FIVE
?
```

Subprocedures DIGIT, TEEN, and TENS select words corresponding to a particular digit in different positions. DIGIT selects words like "three"; TEEN words like "thirteen"; and TENS words like "thirty."

The first instruction in SPELL.GROUP deals with a nonzero hundreds digit of the group, if any. Next, a possible leading zero is eliminated from the group. Then the procedure recognizes the special case of a number greater than ten and less than twenty. These numbers are special because they are represented all in one word, like "thirteen." Other two-digit numbers are represented by one word for the tens digit and one for the ones digit, like "eighty seven." If the number isn't a teen, the procedure then deals with its tens digit and its ones digit separately.

A trick used in SPELL.GROUP looks like this:

```
IF predicate [ expression ] [[]]
```

Here is an example:

```
IF :GROUP>9 [TENS FIRST :GROUP] [[]]
```

By Brian Harvey.

If the predicate tested by IF is FALSE, the value of this expression is the empty list ([]), so it contributes nothing to the final result when combined with other things using SE.

SPELL.GROUP outputs the empty list, not the word ZERO, if its input is 0. This is okay because we want to say "zero" only if the entire number we're spelling is 0, not just one group. (Remember that the reason we wrote SPELL.GROUP for numbers up to three digits is that groups of three are the building blocks of larger numbers.) For example, the number 1000234 is spelled "one million two hundred thirty four," not "one million zero thousand two hundred thirty four." We'll have to remember to notice, later on, if the entire number we're spelling is 0.

Here are the procedures that select the words for each digit.

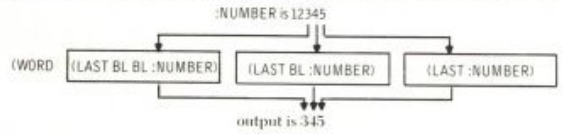
```
TO TENS :DIG
OUTPUT ITEM :DIG [TEN TWENTY THIRTY FORTY FIFTY
        SIXTY SEVENTY EIGHTY NINETY]
END

TO TEEN :DIG
OUTPUT ITEM :DIG [ELEVEN TWELVE THIRTEEN FOURTEEN FIFTEEN
        SIXTEEN SEVENTEEN EIGHTEEN NINETEEN]
END

TO DIGIT :DIG
OUTPUT ITEM :DIG [ONE TWO THREE FOUR FIVE SIX SEVEN
  EIGHT NINE]
END
```

These use the common subprocedure ITEM.

```
TO ITEM :NUM :STUFF
IF :NUM=1 [OP FIRST :STUFF]
OP ITEM :NUM-1 BF :STUFF
END
```

Spelling a Large Number

Now we have to divide a large number into groups of three, so that we can use SPELL.GROUP on each of the triads. One complication is that in dealing with very large numbers, we can't rely on Logo's arithmetic operations, because if we do, the numbers will be rounded off. Logo ordinarily handles numbers only up to ten digits without rounding. We'll use Logo's word-manipulation operations. For example, if we're spelling out the number 12345 and want to find the rightmost group, we'll do something like this:



In other words, we must treat a large number as a word that happens to be composed of digits instead of letters.

Note: In order to convince Logo not to round off numbers longer than ten digits, you have to type them in with a quotation mark like this:

```
PRINT SPELL "1234567890987654321
```

We can work up from SPELL.GROUP. One thing we need is a procedure to combine a spelled-out group with the name of its place in the complete number (thousand, million, etc.):

```
TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP :PLACE]
OP []
END
```

The test for :GROUP>0 is there to deal with cases like 1000234, where the entire thousands group should be omitted.

At this point, it's important to decide whether we are working on the number from left to right or from right to left. The most obvious thing is probably left to right, because that's the way we actually read numbers, starting with the leftmost group. That's the approach I took the first time I wrote this program. But it turns out to be much simpler to write the program if we start from the right. There are two reasons for this.

The first reason is this: suppose you see a long number like 123,456,-234,345,567,678,346,765,654,987. What is the name of the place associated with the leftmost group? To answer that question you have to count the groups, starting from the right. The 987 group is the ones group, the 654 group is the thousands group, the 765 group is the millions group, and so on. So in a sense we have to start from the right in order to know what to do with the 123 group on the left. The second reason is related to the first. Sometimes numbers are written with commas separating the groups. But in Logo we don't use commas inside numbers this way. Suppose you see a number like 1234567890987654321. What is the leftmost group? You might guess 123, but that would be true only if the number of digits in the entire number were a multiple of three. Actually, this number is 1 quintillion 234 quadrillion and so on. In order to know the number of digits in the leftmost group, we have to count off by threes from the right.

Working from right to left, the overall pattern of the program will be more or less like the following. I've written this in lower case to emphasize that it isn't a completed Logo procedure.

```
to spell.number :number
op se (spell.number butlast3 :number) (triad last3 :number)
end
```

Two things are missing from this partially written procedure. First, there is no *stop rule* to tell the procedure when it has reached the end (the leftmost end, that is) of the number. Second, we haven't provided for the place-name input to TRIAD. The solution to the first problem is that when the number of digits in the number we're spelling is three or fewer, we're down to the last group. The solution to the second problem involves providing a list of group place names as another input to this partly written procedure. Putting these things together results in two procedures.

```
TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [[] THOUSAND MILLION BILLION TRILLION
        QUADRILLION QUINTILLION]
END
```

```
TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF :PLACES)
        (TRIAD LAST3 :NUMBER FIRST :PLACES)
END
```

The top-level procedure, SPELL, recognizes the special case of the number 0. In its subprocedure SPELL1, two auxiliary procedures are used that we haven't written yet. LAST3 and BUTLAST3 are operations like LAST and BUTLAST, but they output (all but) the last three letters of a word instead of (all but) the last one. Here they are:

```
TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL :WORD) (LAST :WORD))
END

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END
```

SUGGESTIONS

- What do you have to do to make this program spell out numbers in a language other than English? The main thing, of course, is to change the lists of words in SPELL, DIGIT, TENS, and TEEN. But what *structural* differences are there in different languages? For example, in French there are no special names for 70 and 90. Instead, numbers are added to the names for 60 and 80. That is, 70 is "soixante-dix," or "sixty-ten"; 73 is "soixante-treize" or "sixty-thirteen." (This is true of French as spoken in France; the dialect of French spoken in Belgium *does* have special words for 70 and 90!)
- Can you modify the program to spell out numbers including a decimal fraction, so SPELL 3.14 will output [THREE AND FOURTEEN ONE-HUNDREDTHS]? What about exponential notation, so that SPELL 4E3 will output [FOUR THOUSAND]?
- What about translating to or from Roman numerals? In what ways would a program to do that be similar to this one? How would it be different?
- What about translating backward? That is, write a program that will accept a list of words representing a number and output the number.

PROGRAM LISTING

```
TO SPELL.GROUP :GROUP
IF :GROUP>99 [OUTPUT (SE DIGIT FIRST ▸
     :GROUP "HUNDRED SPELL.GROUP BF ▸
     :GROUP)]
IF 3=COUNT :GROUP [MAKE "GROUP BF ▸
     :GROUP]
IF AND :GROUP>10 :GROUP<20 [OUTPUT ▸
     TEEN :GROUP-10]
OUTPUT SE (IF :GROUP>9 [TENS FIRST ▸
     :GROUP] [[]]) (IF 0<LAST :GROUP ▸
     [DIGIT LAST :GROUP] [[]])
END

TO TENS :DIG
OUTPUT ITEM :DIG [TEN TWENTY THIRTY ▸
     FORTY FIFTY SIXTY SEVENTY EIGHTY ▸
     NINETY]
END
```

```
TO TEEN :DIG
OUTPUT ITEM :DIG [ELEVEN TWELVE ▸
     THIRTEEN FOURTEEN FIFTEEN SIXTEEN ▸
     SEVENTEEN EIGHTEEN NINETEEN]
END

TO DIGIT :DIG
OUTPUT ITEM :DIG [ONE TWO THREE FOUR ▸
     FIVE SIX SEVEN
  EIGHT NINE]
END

TO ITEM :NUM :STUFF
IF :NUM=1 [OP FIRST :STUFF]
OP ITEM :NUM-1 BF :STUFF
END

TO TRIAD :GROUP :PLACE
IF :GROUP>0 [OP SE SPELL.GROUP :GROUP ▸
     :PLACE]
OP []
END
```

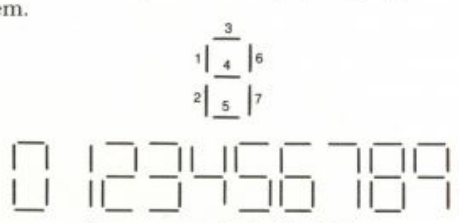
```
TO SPELL :NUMBER
IF :NUMBER=0 [OP [ZERO]]
OP SPELL1 :NUMBER [[] THOUSAND MILLION ▸
     BILLION TRILLION QUADRILLION ▸
     QUINTILLION]
END

TO SPELL1 :NUMBER :PLACES
IF (COUNT :NUMBER)<4 [OP TRIAD :NUMBER ▸
     FIRST :PLACES]
OP SE (SPELL1 BUTLAST3 :NUMBER BF ▸
     :PLACES) (TRIAD LAST3 :NUMBER ▸
     FIRST :PLACES)
END

TO LAST3 :WORD
OP (WORD (LAST BL BL :WORD) (LAST BL ▸
     :WORD) (LAST :WORD))
END

TO BUTLAST3 :WORD
OP BL BL BL :WORD
END
```

Drawing Letters

This project lets the turtle draw letters using a multiple-segment system like that of digital watches. It illustrates Logo's list processing capability and the use of RUN with program-generated Logo instructions. That is, instead of just carrying out procedures that were written ahead of time, this program actually assembles lists of Logo instructions and then carries out those instructions to draw the letters.

Drawing Letters in Segments

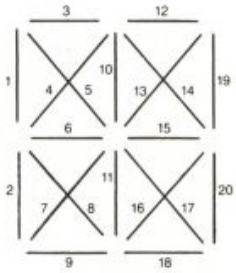
Digital watches, which only have to display digits, generally use a seven-segment system.



Seven-segment display for digits

By Brian Harvey

To display all the letters of the alphabet, I chose to use a twenty-segment system, illustrated below.

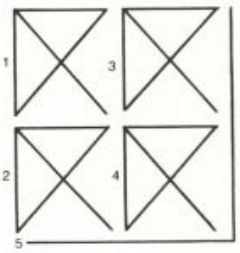


Twenty-segment display for letters

Of course, it would be possible to write a separate procedure for each letter, giving explicit turtle motion commands to shape the letter precisely. The advantage of the segment idea is that it makes it possible to write a single program, then design the individual letters very quickly. For example, after I had finished the letters of the alphabet, it was very easy for me to add the ten digits, even though I hadn't planned for them initially.

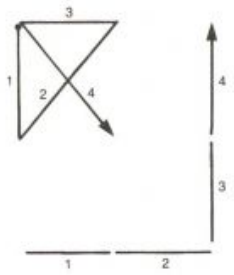
Twenty-segment digits

I could have written twenty procedures, one for each segment. Each would start from a "base" position, move the turtle to one end of the segment, draw the segment, and return to the base position. Then each letter could be described as a list of numbers, identifying the segments that are used to draw the letter. Instead, I chose to try to find some regularities in the way the segments are arranged. I divided the twenty segments into five groups of four each. In each group, the segments can be drawn in a single continuous path, without drawing any segment twice. (I would have liked to be able to draw the entire group of twenty segments continuously without duplication, but that's impossible.) Four of my five groups are identical in shape; the fifth is special.



Dividing twenty segments into five sets of four each

The five groups are numbered in a specific order. Within a group, the segments are also numbered in a specific order; this is shown in the next figure. The program is written so that it draws segments in this order. That is, to draw a letter, the program first draws the four segments that make up the arrow-shaped group in the top left corner. Then the program goes on



Order of segments within a group

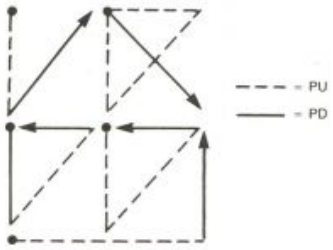
to the second group, the arrow-shaped one at the bottom left, and so on. Within each group, the program first draws segment 1, then 2, 3, and 4.

Not all segments are used in every letter, of course. Therefore, the turtle lifts its pen while tracing some of the segments. For example, consider this representation of the letter *A*.

```
MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
         [PU PU PD] [PU PUPD]]
```

The variable A contains a list of five lists. Each of these smaller lists corresponds to one of the five groups of segments. The first sublist is [PU PD]; this means that the turtle's pen should be up during the first segment and down during the second segment. (There could be up to four words in each sublist. In this case, since there are only two words, the program will stop tracing the first group of segments after the second segment in the group.) This figure shows how the program draws the letter *A*; compare it to the list just given.

The Letter-Drawing Procedures



Drawing an *A*

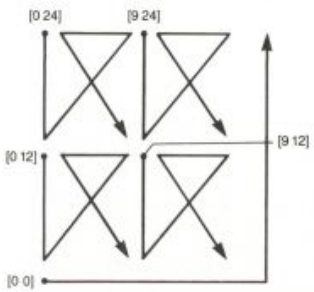
The procedure LETTER draws a letter. It takes two inputs. The first is a list like the one stored in the variable A; the second is a position, that is, a list of two numbers. The letter described by the list is drawn at the position. (Actually it is the lower left corner of the letter that is at the given position.) For example, if we have defined the variable A as just given, we could say

```
LETTER :A [0 0]
```

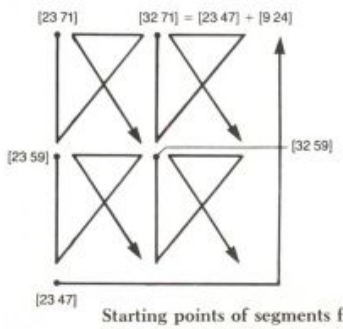
Here is the procedure:

```
TO LETTER :LET :POS
SEGMENTS :LET [ [[0 24] ARROW] [[0 12] ARROW]
        [[9 24] ARROW] [[9 12] ARROW] [[0 0] FINISH] ] :POS
END
```

The LETTER procedure uses a subprocedure SEGMENTS. The second input to SEGMENTS is a list that describes the overall layout of the groups of segments. Like the letter descriptions, it is a list containing five lists. But each of the five lists has only two elements: the starting position of the group of segments and the name of a procedure to draw the group of segments. This procedure is called ARROW for the first four groups and FINISH for the fifth group. The "position" of the beginning of the segment group is actually relative to the position of the letter as a whole, not an absolute screen position. For example, if the position of the letter is [23 47] and the relative position of the third segment group is [9 24], then the actual screen position for that group is [32 71].



Starting points of segments in relative coordinates



Starting points of segments for a letter drawn at POS=[23 47]

To know why the position numbers are what they are, you must know that I chose to base the segment lengths on a 3-4-5 right triangle. The horizontal segments are 9 turtle steps long, the vertical ones 12 steps long, and the diagonal ones 15 steps long. This conveniently makes all the FORWARD commands use whole-number inputs. It is also a reasonable shape for the overall letters.

The procedure SEGMENTS has three inputs. The third is the position of the letter. The first two are both five-element lists of lists. One is a letter description; the other is the overall layout description. The job of SEGMENTS is to match each element of the letter with the corresponding element of the description. It invokes the subprocedure SEGMENT with these sublists as inputs:

```
TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTYP :LET [STOP]
IF NOT EMPTYP FIRST :LET
        [SEGMENT FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END
```

Let's see how this works with a particular example. Suppose we ask Logo to draw the letter *A* with this instruction:

```
LETTER :A [23 47]
```

This ends up invoking SEGMENTS this way:

```
SEGMENTS [[PU PD] [PD PU PD] [PU PU PU PD]
                  [PU PU PD] [PU PU PD]]
         [ [[0 24] ARROW] [[0 12] ARROW]
           [[9 24] ARROW] [[9 12] ARROW] [[0 0] FINISH] ]
         [23 47]
```

Then SEGMENTS invokes SEGMENT five times.

```
SEGMENT [PU PD] [[0 24] ARROW] [23 47]
SEGMENT [PD PU PD] [[0 12] ARROW] [23 47]
SEGMENT [PU PU PU PD] [[9 24] ARROW] [23 47]
SEGMENT [PU PU PD] [[9 12] ARROW] [23 47]
SEGMENT [PU PU PD] [[0 0] FINISH] [23 47]
```

Each element of the list that is specific to the letter *A* (for example, [PU PD]) is matched with an element of the list that describes the layout of letters in general (for example, [[0 24] ARROW]).

Drawing Each Segment Group

Remember that each sublist of the template (the overall layout description) has two pieces: the relative position of the group and the name of the procedure that draws the group. SEGMENT first has to position the turtle, then invoke the correct procedure. To position the turtle, SEGMENT uses a subprocedure called ADDPOS, which adds two position lists just as we did a few paragraphs ago. Then it uses the RUN command to invoke the procedure ARROW or FINISH, as the case may be. These procedures take the letter

description sublist as input, so the procedure name must be linked with that list to form the Logo instruction for RUN.

```
TO SEGMENT :LETPART :TEMPPART :POS
PU
SETPOS ADDPOS :POS FIRST :TEMPPART
RUN LIST LAST :TEMPPART :LETPART
END
```

For example, the first use of SEGMENT in drawing the letter *A* in our example is

```
SEGMENT [PU PD] [[0 24] ARROW] [23 47]
```

This is equivalent to the following Logo instructions.

```
PU
SETPOS ADDPOS [23 47] [0 24]
RUN LIST "ARROW [PU PD]
```

This is, in turn, equivalent to

```
PU
SETPOS [23 71]
ARROW [PU PD]
```

The tricky (but exciting!) thing to understand here is that the instruction ARROW [PU PD] doesn't actually appear in any Logo procedure in this program. Instead, this instruction is put together as the program is run. SEGMENT combines the word ARROW (which it found in the template list) with the list [PU PD] (which it found in the letter description list) into one big list. It then uses the RUN command to interpret that list as a Logo instruction. We'll use the same trick again later.

The procedures ARROW and FINISH have to follow a certain path, setting the turtle's pen up or down between steps as specified in the letter description. They use a common subprocedure DRAW, which knows how to do that. One of the inputs to DRAW is the letter description sublist with the PU and PD commands; the other input is a list of four Logo instruction lists, one for each segment of the group.

```
TO ARROW :PENS
DRAW :PENS [[SETH 180 FD 12] [LT 143.13 FD 15]
        [LT 126.87 FD 9] [LT 126.87 FD 15]]
END

TO FINISH :PENS
DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT 90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
IF EMPTYP :PENS [STOP]
RUN FPUT FIRST :PENS FIRST :CMDS
DRAW BF :PENS BF :CMDS
END
```

Here is how this works out in our example with the letter *A*. The five invocations of SEGMENT listed earlier result in four invocations of ARROW and one of FINISH.

```
ARROW [PU PD]
ARROW [PD PU PD]
ARROW [PU PU PU PD]
ARROW [PU PU PD]
FINISH [PU PU PD]
```

We'll look at the first invocation of ARROW in more detail. ARROW invokes DRAW like this:

```
DRAW [PU PD] [[SETH 180 FD 12] [LT 143.13 FD 15]
        [LT 126.87 FD 9] [LT 126.87 FD 15]]
```

Just as SEGMENTS paired elements of its list inputs, so does DRAW. It ends up executing these Logo instructions:

```
RUN FPUT "PU [SETH 180 FD 12]
RUN FPUT "PD [LT 143.13 FD 15]
```

There might have been up to four of these RUN instructions, because there are four segments in an ARROW group, but in this case there were only two pen commands in the input list :PENS. If we look at what the RUN instructions actually do in this example, we see that the final effect is just as if the procedure contained these instructions:

```
PU SETH 180 FD 12
PD LT 143.13 FD 15
```

This is a straightforward series of turtle graphics commands. Again, though, it's important to understand that that series of commands is not actually part of any procedure. Instead, the commands were *generated* by the DRAW procedure by putting together pieces of its inputs.

Final Details

Here is ADDPOS, the subprocedure of SEGMENT that turns the relative position of a segment group into an absolute position:

```
TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2
       (LAST :POS1)+LAST :POS2
END
```

Finally, the procedure SAY takes an entire word as input and draws the letters in that word one by one. It's used like this:

```
SAY "HELLO [0 0]
```

and here it is.

```
TO SAY :WORD :POS
IF EMPTYP :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 0]
END
```

Here are the definitions for my letters:

```
MAKE "A [[PU PD] [PD PU PD] [PU PU PU PD]
        [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD PD]
        [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [PU PD] [] [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU PD]
        [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD]
          [PU PU PD] [PU PD PU PD] [PD PU PU PD]]
MAKE "R [[PD PU PD] [PD PU PD] [PU PU PD]
        [PU PU PD PD] [PU PU PU PD]]
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] [PU PD PD] [PD]]
MAKE "T [[PU PU PD] [] [PD PU PD] [PD]]
MAKE "U [[PD] [PD] [] [] [PD PD PD PD]]
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] [PU PU PU PD]]
MAKE "W [[PD] [PD PD] [] [PU PU PU PD] [PU PU PD PD]]
MAKE "X [[PU PU PU PD] [PU PD] [PU PD] [PU PU PU PD]]
MAKE "Y [[PU PU PU PD] [] [PU PD] [PD]]
MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] [] [PD PD]]
```

SUGGESTIONS

- Make up descriptions for the twenty-segment digits shown near the beginning of this write-up.
- The letter *L* is described very efficiently by this scheme; the turtle takes no unnecessary steps to draw it. The letter *A*, on the other hand, is not very efficiently described. Each `PU` in its description represents a step that the turtle takes without drawing anything; to draw six strokes, the turtle travels over fifteen segments. Can you work out a way to group the segments that makes more letters more efficient? (I don't have any secret answer to this; I haven't tried it myself.)
- Modify the procedures so that the size of the letters can be varied. You could have an input called `SIZE` and use `3*:SIZE` for the horizontal segments, and so forth.
- Modify the procedures so that the aspect ratio of the letters (the ratio of the vertical segment length to the horizontal segment length) is variable. This is much harder; in general, it requires using trigonometry.
- Make up descriptions for lower-case letters. This may require changing the whole arrangement of segments, since some lower-case let-

ters have *descenders.* That is, they extend below the baseline of the capital letters. These letters are *g, j, p, q,* and *y.* Manufacturers of computer terminals don't always use descenders for lower-case letters. Some avoid it by printing those letters higher than they should be; others just use SMALL CAPITALS instead of lower case.

- Without changing the letter descriptions, change the shapes embodied in the procedures ARROW and FINAL. See if you can invent an interesting new alphabet this way.
- Modify the procedures so that you can write words at an angle, not just horizontally across the screen.

PROGRAM LISTING

```
TO LETTER :LET :POS
SEGMENTS :LET [ [[0 24] ARROW] [[0 12] ▸
     ARROW] [[9 24] ARROW] [[9 12] ▸
     ARROW] [[0 0] FINISH] ] :POS
END

TO SEGMENTS :LET :TEMPLATE :POS
IF EMPTYP :LET [STOP]
IF NOT EMPTYP FIRST :LET [SEGMENT ▸
     FIRST :LET FIRST :TEMPLATE :POS]
SEGMENTS BF :LET BF :TEMPLATE :POS
END

TO SEGMENT :LETPART :TEMPPART :POS
PU
SETPOS ADDPOS :POS FIRST :TEMPPART
RUN LIST LAST :TEMPPART :LETPART
END

TO ARROW :PENS
DRAW :PENS [[SETH 180 FD 12] [LT ▸
     143.13 FD 15] [LT 126.87 FD 9] ▸
     [LT 126.87 FD 15]]
END

TO FINISH :PENS
DRAW :PENS [[SETH 90 FD 9] [FD 9] [LT ▸
     90 FD 12] [FD 12]]
END

TO DRAW :PENS :CMDS
IF EMPTYP :PENS [STOP]
RUN FPUT FIRST :PENS FIRST :CMDS
DRAW BF :PENS BF :CMDS
END

TO ADDPOS :POS1 :POS2
OUTPUT LIST (FIRST :POS1)+FIRST :POS2 ▸
     (LAST :POS1)+LAST :POS2
END

TO SAY :WORD :POS
IF EMPTYP :WORD [STOP]
LETTER THING FIRST :WORD :POS
SAY BF :WORD ADDPOS :POS [24 0]
END

MAKE "A [[PU PD] [PD PU PD] [PU PU PU ▸
     PD] [PU PU PD] [PU PU PD]]
MAKE "B [[PD PU PD] [PD PU PD] [PU PD ▸
     PD] [PU PU PU PD] [PD PD]]
MAKE "C [[PU PD] [PU PU PU PD] [PU PU ▸
     PD] [] [PU PD]]
MAKE "D [[PD PU PD] [PD] [PU PU PU PD] ▸
     [PU PD] [PD]]
MAKE "E [[PD PU PD] [PD PU PD] [PU PU ▸
     PD] [] [PD PD]]
MAKE "F [[PD PU PD] [PD PU PD] [PU PU ▸
     PD] []]
MAKE "G [[PD PU PD] [PD] [PU PU PD] ▸
     [PU PU PD] [PD PD PD]]
MAKE "H [[PD] [PD PU PD] [] [PU PU PD] ▸
     [PU PU PD PD]]
MAKE "I [[PU PU PD] [] [PD PU PD] [PD] ▸
     [PD PD]]
MAKE "J [[PU PU PD] [] [PD PU PD] [PD] ▸
     [PD]]
MAKE "K [[PD] [PD PU PD] [PU PD] [PU ▸
     PU PU PD]]
MAKE "L [[PD] [PD] [] [] [PD PD]]
MAKE "M [[PD PU PU PD] [PD] [PU PD] [] ▸
     [PU PU PD PD]]
MAKE "N [[PD PU PU PD] [PD] [] [PU PU ▸
     PU PD] [PU PU PD PD]]
MAKE "O [[PD PU PD] [PD] [PU PU PD] [] ▸
     [PD PD PD PD]]
MAKE "P [[PD PU PD] [PD PU PD] [PU PU ▸
     PD] [PU PU PD] [PU PU PU PD]]
MAKE "Q [[PD PU PD] [PD] [PU PU PD] ▸
     [PU PD PU PD] [PD PU PU PD]]
```

```
MAKE "R [[PD PU PD] [PD PU PD] [PU PU ▸
     PD] [PU PU PD PD] [PU PU PU PD]]
MAKE "S [[PU PD] [PU PU PD] [PU PU PD] ▸
     [PU PD PD] [PD]]
MAKE "T [[PU PU PD] [] [PD PU PD] ▸
     [PD]]
MAKE "U [[PD] [PD] [] [] [PD PD PD ▸
     PD]]
MAKE "V [[PD] [PU PU PU PD] [] [PU PD] ▸
     [PU PU PU PD]]
MAKE "W [[PD] [PD PD] [] [PU PU PU PD] ▸
     [PU PU PD PD]]
MAKE "X [[PU PU PU PD] [PU PD] [PU PD] ▸
     [PU PU PU PD]]
MAKE "Y [[PU PU PU PD] [] [PU PD] ▸
     [PD]]
MAKE "Z [[PU PU PD] [PU PD] [PU PD PD] ▸
     [] [PD PD]]
```

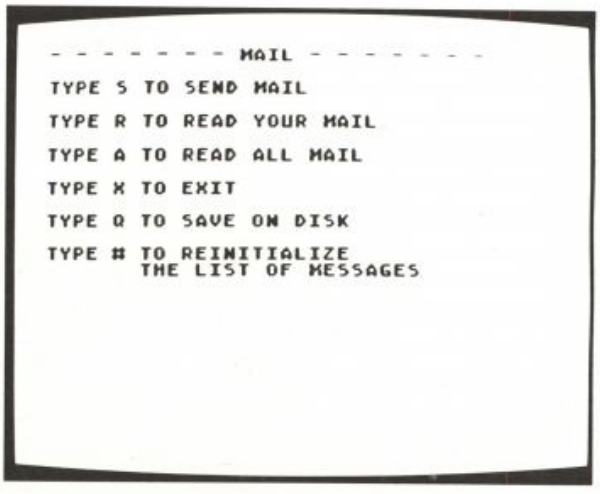
Mail

When I was a kid in school, my friends and I liked passing notes to each other. It was reflection on this experience that inspired me to write a mail program. In those olden days, the suspense was great as we waited to see if we could send messages from one side of the room to another without getting caught. With this modern method of letting Logo be the mail carrier, students today find different pleasures.

Using the Program

Since Logo has no mail system of its own, I decided to build one. The essential actions are sending and receiving mail. This project is just one example of an electronic mail system. :em.

The program assumes that you have a disk on which daily mail can be saved. For convenience, you should reserve one diskette specifically to hold the messages and the mail program.* To start the program, type `MAIL`. You will get a screen that looks like this:



*You may also change the mail program so that you can use a cassette recorder. Then you would save to the cassette instead of to the disk.

By Annette Dula.

Sending Mail

If you want to send mail, type S.

```
WHO IS THE MESSAGE FOR?
>LAUREN
WHO IS THE MESSAGE FROM?
>CYNTHIA
BEGIN TYPING YOUR MESSAGE.
PRESS RETURN AFTER EACH TYPED LINE.
TYPE . ON A SEPARATE LINE TO END.
>FOR WHAT CRIME WERE SACCO AND
>VANZETTI PUT TO DEATH?
>.
SEND IT? ( Y OR N )
```

First you are asked who the message is for. A prompt (>) appears, and you type in the name of the person to whom you want to send mail. You are then asked who the message is from, and you type in your name. Next you receive instructions. After you type in your message, you are asked if you really want to send it. If you do, you are informed that the message is in the mailbox.

Reading Your Mail

To read your mail, type R.

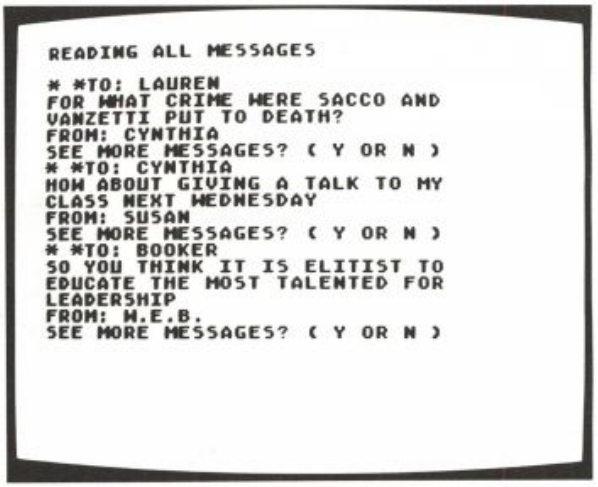
You are asked to type in your name. Once you do, your messages appear on the screen.

```
TYPE YOUR NAME TO SEE YOUR MESSAGES:
>LAUREN

LAUREN, HERE ARE YOUR MESSAGES!!!

TO: LAUREN
FOR WHAT CRIME WERE SACCO AND
VANZETTI PUT TO DEATH?
FROM: CYNTHIA

DELETE MESSAGE? ( Y OR N )
```

After reading each message, you are asked if you want to delete it. If you type Y, that message is deleted.

Reading All the Mail

If for some reason you want to read all the messages that have been written, type A.



After each message, you are asked if you want to see more messages. If you type Y, you see another message, otherwise you exit from reading all messages.

Other Commands

- Q Automatically saves all messages on the diskette. You are asked if the mail disk is in the drive. If you type Y, the program and all messages are saved on diskette; otherwise the program stops.
- X Stops the program.
- # Deletes all messages.

Structure of the Mail Program

The Data Base

All the messages are organized into one list named ALL.MESSAGES. For example, :ALL.MESSAGES might look like this:

```
[[[TO: JRD] [WHO GOT THE VOTE FIRST: BLACKS OR] [WOMEN]
  [FROM: DAVE]]
[[TO: LAUREN] [FOR WHAT CRIME WERE SACCO AND]
  [VANZETTI PUT TO DEATH] [FROM: CYNTHIA]]
```

```
[[TO: LISA][DID THEY EVER DECIDE WHETHER]
 [BLACK ENGLISH IS A LANGUAGE OR A DIALECT?]
 [FROM: MARGARET]]
[[TO: JAN] [I HEARD THAT THE BLUEBERRY CROP IN]
 [MAINE WILL BE GREAT THIS YEAR.]
 [BECAUSE OF THE ACID RAIN.]
 [DO YOU BELIEVE IT ?] [FROM: TOM]]
[[TO: BOOKER] [DO YOU AGREE WITH ME THAT THE]
 [TALENTED TENTH SHOULD RECEIVE]
 [EDUCATION FOR LEADERSHIP] [FROM: W.E.B.]]]
```

Each message is itself a list of lists.

The first message in this example is

```
[[TO: JRD]
 [WHO GOT THE VOTE FIRST: BLACKS OR]
 [WOMEN]
 [FROM: DAVE]]
```

This message contains four sublists:

The first is `[TO: JRD]`.
The second is `[WHO GOT THE VOTE FIRST: BLACKS OR]`.
The third is `[WOMEN]`.
The last is `[FROM: DAVE]`.

The word `TO` and the receiver's name make up the first list in each message, while the word `FROM` and the sender's name make up the last list in the message.

The Main Procedure

`MAIL` is the main procedure of the program. It displays the help text, gets a character command from the user, and checks to see if the command is valid. If it is, it calls the appropriate procedures to carry out the actions. These procedures are `SEND.MAIL`, `MY.MESSAGES`, `READ.ALL.MAIL`, `REMOVE.ALL.MESSAGES`, and `DISK.DUMP`.

```
TO MAIL
HELP
MAKE "CHAR RC
IF NOT MEMBERP :CHAR [R S A X Q #] [PR [] PR
  [!!!NOT A COMMAND.]]
IF :CHAR = "R [MY.MESSAGES]
IF :CHAR = "S [SEND.MAIL]
IF :CHAR = "A [READ.ALL.MAIL]
IF :CHAR = "X [STOP]
IF :CHAR = "Q [DISK.DUMP STOP]
IF :CHAR = "# [REMOVE.ALL.MESSAGES]
PR []
MAIL
END
```

`HELP` puts the menu of possible actions on the text screen.

```
TO HELP
WAIT 50
CT
PR [- - - - - - - MAIL - - - - - - -]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END
```

Sending Mail

SEND.MAIL is the main procedure for sending mail. First it asks for the name of the person who is to receive the message. It then asks for your name (the sender). You are then given instructions for typing the message. Finally you are given a chance to change your mind about sending it. If you decide that you want to send it, the message is included in the list of all messages.

```
TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE SE FPUT :ANS GET.MESSAGE
  [] LPUT :FROM []
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!!MESSAGE DELETED!!!!!] WAIT 50 STOP]
IF EMPTYP :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * *]
PR []
END
```

SEND.MAIL uses four subprocedures: RECEIVER'S.NAME, SENDER'S.NAME, GET.MESSAGE, and ADD.THE.MESSAGE.

RECEIVER'S.NAME outputs a sentence of the word TO: and the name of the person who is to receive the message. SENDER'S.NAME works similarly.

```
TO RECEIVER'S.NAME
TYPE ">
OP SE "TO: RL
END

TO SENDER'S.NAME
TYPE ">
OP SE "FROM: RL
END
```

GET.MESSAGE lets you type in a message, line by line. A "." typed on a separate line signals the completion of the message.

```
TO GET.MESSAGE :MSG
TYPE ">
MAKE "EACH.LINE RL
IF :EACH.LINE = [.] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END
```

In ADD.THE.MESSAGE, the typed message is added to :ALL.MESSAGES.

```
TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT :PRESENT.MESSAGE :ALL.MESSAGES
END
```

Reading Your Mail

MY.MESSAGES is the main procedure for reading your own messages. It gets your name and checks to see if you have any mail by calling CHECK.MY.MESSAGES.

```
TO MY.MESSAGES
CT
IF EMPTYP :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR MESSAGES]
TYPE ">
MAKE "ANS RL
IF EMPTYP :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [HERE ARE YOUR MESSAGES!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES 0
END
```

CHECK.MY.MESSAGES checks each message in :LIST to see if it is for you. CHECK.MY.MESSAGES takes three inputs. The first is :WHO, the name of the person (you) whose mail it is looking for. The second, :LIST, is the list of messages. The third, :COUNTER, is a message counter that is needed if you should decide to delete a message.

```
TO CHECK.MY.MESSAGES :WHO :LIST :COUNTER
PR []
IF EMPTYP :LIST [PR [* * * * * * THAT'S IT * * * * * *]
  STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST FIRST :LIST
  [PRINT.AND.DELETE FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + :COUNTER )
END
```

CHECK.MY.MESSAGES calls PRINT.AND.DELETE, which prints a message and asks if you want to delete it.

```
TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y [MAKE "ALL.MESSAGES DELETE :COUNTER :ALL.MESSAGES
  PR [] PR [!!!!!MESSAGE DELETED!!!!!]]
END
```

PRINT.AND.DELETE uses PRINT.MESSAGE and DELETE.

PRINT.MESSAGE prints a single message, consisting of the receiver's name, then the message, and finally the sender's name.

```
TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END
```

```
TO DELETE :N :LIST
IF EMPTYP :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF :LIST
END
```

Reading All the Mail

The main procedure for reading all the mail is READ.ALL.MAIL; it calls the message-printing procedure, PRINT.ALL.MESSAGES.

```
TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END
```

PRINT.ALL.MESSAGES prints each message and asks you if you want to see more messages.

```
TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTYP :ALL [PR [* * * * * NO MORE MESSAGES * * * * *]
  STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * *  YOU EXITED MAIL READER * * *]
  STOP]
PRINT.ALL.MESSAGES BF :ALL
END

TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END
```

Saving on the Diskette

DISK.DUMP saves on the disk. First it reminds you to put the disk in the drive.*

```
TO DISK.DUMP
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE?????? (Y OR N)]
IF RC = "Y [SAVE "D:MAIL]
END
```

Reinitializing the List of Messages

REMOVE.ALL.MESSAGES clears all messages from the list of messages.

```
TO REMOVE.ALL.MESSAGES
MAKE "ALL.MESSAGES []
END
```

PROGRAM LISTING

```
TO MAIL
HELP
MAKE "CHAR RC
IF NOT MEMBERP :CHAR [R S A X Q #] [PR ▸
     [] PR [!!!NOT A COMMAND.]]
IF :CHAR = "R [MY.MESSAGES]
IF :CHAR = "S [SEND.MAIL]
IF :CHAR = "A [READ.ALL.MAIL]
IF :CHAR = "X [STOP]
IF :CHAR = "Q [DISK.DUMP STOP]
IF :CHAR = "# [REMOVE.ALL.MESSAGES]
PR []
MAIL
END

TO REMOVE.ALL.MESSAGES
MAKE "ALL.MESSAGES []
END
```

*If you are using a cassette instead of a diskette, you must change the last instruction in DISK.DUMP so that it saves to a cassette: IF RC = "Y [SAVE "C:]

```
TO HELP
WAIT 50
CT
PR [- - - - - - - MAIL - - - - - - -]
PR []
PR [TYPE S TO SEND A MESSAGE]
PR []
PR [TYPE R TO READ YOUR MAIL]
PR []
PR [TYPE A TO READ ALL MAIL]
PR []
PR [TYPE X TO EXIT]
PR []
PR [TYPE Q TO SAVE ON DISK]
PR []
PR [TYPE # TO REINITIALIZE]
SETCURSOR [7 13]
PR [THE LIST OF MESSAGES]
END

TO SEND.MAIL
CT
PR []
PR [WHO IS THE MESSAGE FOR?]
MAKE "ANS RECEIVER'S.NAME
PR [WHO IS THE MESSAGE FROM?]
MAKE "FROM SENDER'S.NAME
PR [BEGIN TYPING YOUR MESSAGE.]
PR [PRESS RETURN AFTER EACH TYPED ►
     LINE.]
PR [TYPE . ON A SEPARATE LINE TO END.]
MAKE "PRESENT.MESSAGE SE FPUT :ANS ►
     GET.MESSAGE [] LPUT :FROM []
PR [SEND IT? ( Y OR N )]
IF RC = "N [PR [!!!!!MESSAGE ►
     DELETED!!!!!] WAIT 50 STOP]
IF EMPTYP :PRESENT.MESSAGE [STOP]
ADD.THE.MESSAGE :PRESENT.MESSAGE
PR []
PR [* * * * IT'S IN THE MAILBOX * * * ►
     *]
PR []
END

TO RECEIVER'S.NAME
TYPE ">
OP SE "TO: RL
END

TO SENDER'S.NAME
TYPE ">
OP SE "FROM: RL
END

TO GET.MESSAGE :MSG
TYPE ">
MAKE "EACH.LINE RL
IF :EACH.LINE = [.] [OP :MSG]
OP GET.MESSAGE LPUT :EACH.LINE :MSG
END

TO ADD.THE.MESSAGE :PRESENT.MESSAGE
MAKE "ALL.MESSAGES FPUT ►
     :PRESENT.MESSAGE :ALL.MESSAGES
END

TO MY.MESSAGES
CT
IF EMPTYP :ALL.MESSAGES [STOP]
PR [TYPE YOUR NAME TO SEE YOUR ►
     MESSAGES]
TYPE ">
MAKE "ANS RL
IF EMPTYP :ANS [MY.MESSAGES STOP]
PR []
PR SE WORD FIRST :ANS ", [HERE ARE ►
     YOUR MESSAGES!!!]
PR []
CHECK.MY.MESSAGES :ANS :ALL.MESSAGES 0
END

TO CHECK.MY.MESSAGES :WHO :LIST ►
     :COUNTER
PR []
IF EMPTYP :LIST [PR [* * * * * * ►
     THAT'S IT * * * * * *] STOP]
IF EQUALP FIRST :WHO FIRST BF FIRST ►
     FIRST :LIST [PRINT.AND.DELETE ►
     FIRST :LIST :COUNTER]
PR []
PR []
CHECK.MY.MESSAGES :WHO BF :LIST ( 1 + ►
     :COUNTER )
END

TO PRINT.AND.DELETE :MESSAGE :COUNTER
PRINT.MESSAGE :MESSAGE
PR []
TYPE [DELETE MESSAGE? ( Y OR N )]
IF RC = "Y [MAKE "ALL.MESSAGES DELETE ►
     :COUNTER :ALL.MESSAGES PR [] PR ►
     [!!!!!MESSAGE DELETED!!!!!]]
END
```

```
TO DELETE :N :LIST
IF EMPTYP :LIST [OP []]
IF :N = 0 [OP BF :LIST]
OP FPUT FIRST :LIST DELETE :N - 1 BF ▸
     :LIST
END

TO READ.ALL.MAIL
CT
PR [READING ALL MESSAGES]
PRINT.ALL.MESSAGES :ALL.MESSAGES
END

TO PRINT.ALL.MESSAGES :ALL
PR []
IF EMPTYP :ALL [PR [* * * * * NO MORE ▸
     MESSSAGES * * * * *] STOP]
TYPE [* *]
PRINT.MESSAGE FIRST :ALL
TYPE [READ MORE MESSAGES? (Y OR N)]
IF RC = "N [PR [] PR [* * *  YOU ▸
     EXITED MAIL READER * * *] STOP]
PRINT.ALL.MESSAGES BF :ALL
END

TO PRINT.MESSAGE :MSG
IF EMPTYP :MSG [STOP]
PR FIRST :MSG
PRINT.MESSAGE BF :MSG
END
```

```
TO DISK.DUMP
PR []
PR []
PR []
PR [IS THE DISK IN THE DRIVE?????? (Y ▸
     OR N)]
IF RC = "Y [SAVE "D:MAIL]
END
```

```
MAKE "ALL.MESSAGES [[[TO: LAUREN] [FOR ▸
     WHAT CRIME WERE SACCO AND] ▸
     [VANZETTI PUT TO DEATH?] [FROM: ▸
     CYNTHIA]] [[TO: CYNTHIA] [HOW ▸
     ABOUT GIVING A TALK TO MY] [CLASS ▸
     NEXT WEDNESDAY] [FROM: SUSAN]] ▸
     [[TO: BOOKER] [SO YOU THINK IT IS ▸
     ELITIST TO] [EDUCATE THE MOST ▸
     TALENTED FOR] [LEADERSHIP] [FROM: ▸
     W.E.B.]] [[TO: LAUREN] [I HEARD ▸
     THAT THE BLUEBERRY] [CROP IN ▸
     MAINE WILL BE GREAT] [NEXT YEAR ▸
     BECAUSE OF THE ACID RAIN.] [FROM: ▸
     TOM]] [[TO: TO LISA] [IS BLACK ▸
     ENGLISH CONSIDERED] [A LANGUAGE ▸
     OR A DIALECT?] [FROM: MARGARET]] ▸
     [[TO: JRD] [WHO GOT THE VOTE ▸
     FIRST] [BLACKS OR WOMEN?] [FROM: ▸
     DAVE]]]
```

Wordscram

A Word Guessing Game

WORDSCRAM picks a word, scrambles the letters, and shows you the scrambled version of the word. Your job is to guess the word. (In this sample game, the word is chosen from a list of thirty or forty technical Logo terms.) WORDSCRAM helps you by showing which letters in your guess are in the correct spot. You can also type HINT if you need a hint, or HELP if you want to give up. Here is a sample of WORDSCRAM in action.

By Keith Sharman.

```
?WORDSCRAM
WELCOME TO WORDSCRAM !
DO YOU WANT INSTRUCTIONS ? N
THINKING....

OK.  HERE IS YOUR SCRAMBLED WORD:
RSNRUCOEI

WHAT'S YOUR GUESS ?
RE                                  Two letters correct.

WHAT'S YOUR GUESS ?
RE
**

WHAT'S YOUR GUESS ?
HINT                                        Get a hint.
WELL...OK...TRY REC                 Computer responds.

WHAT'S YOUR GUESS ?
RECUSR                              S and R not correct.
****??

WHAT'S YOUR GUESS ?
RECURSION
*********

DOING GREAT !                       Got it!
DO YOU WANT ANOTHER WORD ? Y
THINKING....

OK. HERE IS YOUR SCRAMBLED WORD:
PUUTOT

WHAT'S YOUR GUESS ?
HELP                                I give up.

THE WORD WAS OUTPUT
DO YOU WANT ANOTHER WORD ? N

                                    Program ends.
```

Scrambling a Word

The heart of WORDSCRAM is SCRAMBLE. It takes a word as input and outputs a scrambled version of it. The strategy goes something like this. Let's say the word to scramble is "draw."

1. Pick a letter from the word at random.
2. To make sure that the letter does not get picked again, remove it from the word.
3. Join the letter just picked to the result of scrambling the remaining letters of the word. Continue until there are no more letters left.

Using the word "draw" as an example, we might get this result:

```
SCRAMBLE "DRAW
    W + SCRAMBLE "DRA
        R + SCRAMBLE "DA
            A + SCRAMBLE "D
                D + SCRAMBLE "
```

The assembled word is "wrad."

SCRAMBLE picks a letter from the word, then uses that letter in two ways: it removes the letter from the word (to get the input for the recursive invocation of SCRAMBLE), and it sticks the same letter back onto the beginning of the scrambled word. To make this work, after SCRAMBLE picks a letter, it invokes a subprocedure, SCRAMBLE1 ,whose second input is the letter to remove from the word.

```
TO SCRAMBLE :WORD
IF EMPTYP :WORD [OP "]
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER (SCRAMBLE REMOVE :LETTER :WORD)
END
```

Here is how SCRAMBLE and SCRAMBLE1 interact, in the same example we looked at before.

```
SCRAMBLE "DRAW
   SCRAMBLE1 "DRAW "W
      SCRAMBLE "DRA
         SCRAMBLE1 "DRA "R
            SCRAMBLE "DA
               SCRAMBLE1 "DA "A
                  SCRAMBLE "D
                     SCRAMBLE1 "D "D
                        SCRAMBLE "
                        SCRAMBLE outputs "
                     SCRAMBLE1 outputs "D which is WORD "D "
                  SCRAMBLE outputs "D
               SCRAMBLE1 outputs "AD which is WORD "A "D
            SCRAMBLE outputs "AD
         SCRAMBLE1 outputs "RAD which is WORD "R "AD
      SCRAMBLE outputs "RAD
   SCRAMBLE1 outputs "WRAD which is WORD "W "RAD
SCRAMBLE outputs "WRAD
```

Removing a Letter from a Word

REMOVE takes two inputs, a letter and a word. It compares the input letter with each letter of the input word. When it finds a matching letter, it outputs the word with that letter removed.

```
?PRINT REMOVE "U "RECURSION
RECRSION
?
```

REMOVE works by comparing the input letter with the first letter of the input word. If they match, then the BUTFIRST of the word is the output we want. Otherwise, the output is formed by joining the first letter of the input word with the result of REMOVEing the input letter from the rest of the word.

```
TO REMOVE :LETTER :WORD
IF :LETTER=FIRST :WORD [OP BF :WORD]          Send back the rest.
OP WORD FIRST :WORD REMOVE :LETTER BF :WORD
END
```

Here is how the preceding example (using RECURSION as the word) happens.

```
REMOVE "U "RECURSION
    REMOVE "U "ECURSION
        REMOVE "U "CURSION
            REMOVE "U "URSION
            REMOVE outputs "RSION
        REMOVE outputs "CRSION which is WORD "C "RSION
    REMOVE outputs "ECRSION which is WORD "E "CRSION
REMOVE outputs "RECRSION which is WORD "R "ECRSION
```

The remaining procedures in this program are straightforward and won't be explained in detail. You can look at the program listing to see what they are.

SUGGESTIONS

Here are a few ideas for changing WORDSCRAM.

- Change the list of words it knows.
- Tell the player how many guesses it took to get the word.
- After the player guesses the word, ask if she or he would like to see the definition of the word. Since WORDSCRAM's words are technical Logo terms, this would be an interesting way to learn about Logo.
- Add some new messages.
- Do some psychology experiments. Some words look very strange when scrambled. Does this "strangeness" vary from person to person? Some people are better at unscrambling words than others. Why? What sort of strategy do you apply to unscrambling a word? Does it resemble other problem-solving strategies you use?

PROGRAM LISTING

```
TO WORDSCRAM
TS
CT
PR [WELCOME TO WORDSCRAM !]
PR []
PR [DO YOU WANT THE INSTRUCTIONS ?]
IF GETANSWER RC [INSTRUCTIONS] [PR []]
PLAYGAME WIN.MESSAGES GETWORDS
END
```

SEE IF THE USER WANTS INSTRUCTIONS

```
TO GETANSWER :ANS
IF :ANS = "Y [TYPE "YES. OP "TRUE]
IF :ANS = "N [TYPE "NO. OP "FALSE]
PR [PLEASE ANSWER WITH Y OR N]
OP GETANSWER RC
END

TO INSTRUCTIONS
TS CT
PR (SE [FROM A LIST OF] COUNT GETWORDS [LOGO WORDS, THE])
PR [COMPUTER WILL PICK ONE AT RANDOM AND]
PR [SCRAMBLE IT FOR YOU. YOUR JOB IS]
PR [TO UNSCRAMBLE IT.]
PR []
PR [IT IS NOT NECESSARY TO GUESS THE WORD]
PR [ON THE FIRST TRY. THE COMPUTER WILL]
PR [TELL YOU WHICH LETTERS YOU HAVE IN]
PR [THE RIGHT POSITION BY PRINTING A]
PR [STAR UNDER EACH CORRECT LETTER. A]
PR [LETTER IN THE WRONG POSITION WILL]
PR [HAVE A ? UNDER IT.]
PR []
PR [IF YOU ARE REALLY STUCK, TYPE HINT]
PR [FOR A HINT OR HELP TO SEE THE WORD.]
PR []
PR [GOOD LUCK.]
PR []
TYPE [PRESS ANY KEY TO START...]
MAKE "DUMMY RC
END
```

STARTING THE GAME PLAY

```
TO PLAYGAME :WIN.MESSAGES :WORDS
CT
PR [THINKING ...]
PLAYGAME1 RANPICK :WORDS
IF ANOTHER? [PLAYGAME :WIN.MESSAGES :WORDS] [PR []]
END
```

```
TO PLAYGAME1 :WORD
MAKE "SCRAMBLED SCRAMBLE :WORD
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
MAKE "TOO.MANY.HINTS "FALSE
MAKE "GUESSED.WORDS SE FIRST :WORD []
GET
END
```

SCRAMBLING THE WORD

```
TO SCRAMBLE :WORD
IF :WORD = " [OP "]
OP SCRAMBLE1 :WORD RANPICK :WORD
END

TO SCRAMBLE1 :WORD :LETTER
OP WORD :LETTER ( SCRAMBLE REMOVE :LETTER :WORD )
END

TO REMOVE :LETTER :WORD
IF :LETTER = FIRST :WORD [OUTPUT BF :WORD]
OUTPUT WORD FIRST :WORD REMOVE :LETTER BF :WORD
END
```

GETTING THE USER'S GUESS

```
TO GET
PR []
PR [WHAT'S YOUR GUESS ?]
IF ( NOT ( ROW < 23 ) ) [REFRESH.SCREEN]
GETGUESS FIRST RL
END

TO ROW
OP .EXAMINE 171
END

TO REFRESH.SCREEN
SAVE.CURSOR
REDISPLAY
RESTORE.CURSOR
END

TO GETGUESS :GUESS
IF EMPTYP :GUESS [OP GETGUESS FIRST RL]
IF :GUESS = "HELP [SHOW.WORD STOP]
IF :GUESS = "HINT [HINT LAST :GUESSED.WORDS GET STOP]
ADDGUESS :GUESS COMPARE :GUESS :WORD
IF :GUESS = :WORD [PR [] PR RANPICK :WIN.MESSAGES ▸
       STOP] [GET]
END
```

CHECK THE GUESS FOR CORRECT AND INCORRECT LETTERS

```
TO ADDGUESS :GUESS
MAKE "GUESSED.WORDS LPUT :GUESS :GUESSED.WORDS
END

TO COMPARE :GUESS :CORRECT
IF ( OR ( :GUESS = " ) ( :CORRECT = " ) ) [PR [] STOP]
IF ( ( FIRST :GUESS ) = ( FIRST :CORRECT ) )    ▸
      [TYPE [*]] [TYPE [?]]
COMPARE BF :GUESS BF :CORRECT
END
```

HINT AND HELP

```
TO HINT :G
IF ( OR ( :G = BL :WORD ) ( :G = ( BL BL :WORD ) ) )    ▸
      [MAKE "TOO.MANY.HINTS "TRUE]
IF :TOO.MANY.HINTS [PR [YOU DON'T NEED A HINT!]    ▸
      PR [THINK SOME MORE.] STOP]
TYPE [WELL]
DODOTS ( 1 + RANDOM 7 )
TYPE [OK]
DODOTS ( 1 + RANDOM 5 )
PR SE [TRY] HINTWORD1 :G :WORD
END

TO HINTWORD1 :W1 :W2
IF EMPTYP :W2 [OP []]
IF EMPTYP :W1 [OP FIRST :W2]
IF NOT EQUALP FIRST :W1 FIRST :W2 [OP FIRST :W2]
OP WORD FIRST :W2 HINTWORD1 BF :W1 BF :W2
END

TO DODOTS :N
IF :N = 0 [STOP]
WAIT 5
TYPE [.]
DODOTS :N - 1
END

TO SHOW.WORD
PR []
PR SE [THE WORD WAS] :WORD
END
```

MISCELLANEOUS PROCEDURES

```
TO ITEM :N :OBJECT
IF :N = 1 [OUTPUT FIRST :OBJECT]
OUTPUT ITEM :N - 1 BF :OBJECT
END
```

```
TO RESTORE.CURSOR
SETCURSOR :CURSOR
END

TO REDISPLAY
SETCURSOR [0 0]
PR [THINKING.....]
PR []
PR [OK. HERE IS YOUR SCRAMBLED WORD:]
PR :SCRAMBLED
PR []
END

TO SAVE.CURSOR
PR []
MAKE "CURSOR LIST ( .EXAMINE 172 ) - 1 ( .EXAMINE 171 ) - 1
END

TO ANOTHER?
PR []
PR [DO YOU WANT ANOTHER WORD ?]
OP GETANSWER RC
END

TO RANPICK :L
OP ITEM ( 1 + RANDOM COUNT :L ) :L
END

TO GETWORDS
OP [CATALOG TURTLE FORWARD BACK LEFT RIGHT PROCEDURE    ▶
    INPUT RECURSION SETBG CIRCLE SQUARE LOGO GRAPHICS    ▶
    EDIT REPEAT POTS DEFINE COUNT HEADING MEMBERP NODES    ▶
    PADDLE DYNATURTLE INSTANT BUTFIRST PENDOWN PENUP    ▶
    PRODUCT RANDOM SETCURSOR SETPC WINDOW TOUCHING MAKE    ▶
    BUTLAST OUTPUT HIDETURTLE SQRT]
END

TO WIN.MESSAGES
OP [[HEY, YOU'RE PRETTY SMART] [WHAT A FLUKE!]    ▶
    [WE ALL GET LUCKY ONCE IN A WHILE!]    ▶
    [A GOLD STAR FOR YOU] [1 POINT FOR YOU!]    ▶
    [DOING GREAT!!] [KEEP UP THE GOOD WORK...]]
END
```

Madlibs™

This project plays the game of Madlibs.* The program asks for words or phrases with which to fill in the blanks in an already-prepared story. Then it prints the resulting story.

*"Madlibs" is a trademark of Price/Stern/Sloan.

By Brian Harvey; story template by Susan Cotten.

Here is an example of a story to be used with the program.

```
AT _____, _____ WAS _____ING DOWN THE
   time of day      person      way to move

STREET. _____ SPOTTED A _____ DIGGING IN A
            person              animal

GARBAGE CAN ACROSS THE STREET. _____ BEGAN TO
                                  person

_____ IN THE OPPOSITE DIRECTION BUT IT WAS TOO LATE.
way to move

THE _____ SAW _____. IT BEGAN TO CHASE
      animal        person

_____. _____ TRIPPED AND FELL. THE
  person   person

_____CAME UP BESIDE _____AND BEGAN TO WAG ITS
  animal                person

_____. _____ REALIZED THERE WAS NOTHING TO
 body part   person

FEAR. _____ REACHED OUT AND PATTED THE _____.
         person                        animal
```

Here is what happens when you use the program with this story.

```
?MADLIB :STORY1
TELL ME A TIME OF DAY
DUSK
TELL ME A PERSON'S NAME
URSULA
TELL ME A WAY TO MOVE
JUMP
TELL ME AN ANIMAL YOU FEAR
RAT
TELL ME A BODY PART
TOE
AT DUSK, URSULA WAS JUMPING DOWN THE STREET.
URSULA SPOTTED A RAT DIGGING IN A GARBAGE CAN ACROSS
THE STREET. URSULA BEGAN TO JUMP IN THE OPPOSITE
DIRECTION BUT IT WAS TOO LATE. THE RAT SAW URSULA. IT
BEGAN TO CHASE URSULA. URSULA TRIPPED AND FELL. THE
RAT CAME UP BESIDE URSULA AND BEGAN TO WAG ITS TOE.
URSULA REALIZED THAT THERE WAS NOTHING TO FEAR. URSULA
REACHED OUT AND PATTED THE RAT.
?
```

How a Story Is Represented

A story is represented as a list that contains words and lists (which we'll refer to as *sublists*). The sublists are the blanks of the story. Here is the list that represents the preceding example.

```
MAKE "STORY1 [AT * [HOUR TIME OF DAY]
, [PERSON PERSON'S NAME]
WAS * [MOTION WAY TO MOVE] ING DOWN THE STREET. [PERSON]
SPOTTED A [ANIMAL ANIMAL YOU FEAR] DIGGING IN A GARBAGE CAN
ACROSS THE STREET. [PERSON] BEGAN TO [MOTION] IN THE
OPPOSITE DIRECTION BUT IT WAS TOO LATE. THE [ANIMAL] SAW
* [PERSON] . IT BEGAN TO CHASE * [PERSON] . [PERSON] TRIPPED
AND FELL. THE [ANIMAL] CAME UP BESIDE [PERSON] AND BEGAN
TO WAG ITS * [ANATOMY BODY PART] . [PERSON] REALIZED THERE
WAS NOTHING TO FEAR. [PERSON] REACHED OUT AND PATTED
THE * [ANIMAL] .]
```

Each word or phrase that the user types to replace a blank is given a *name*, so that the program is able to remember it. The named phrase can be used to fill more than one blank. The sublist

```
[MOTION WAY TO MOVE]
```

signals the program to type

```
TELL ME A WAY TO MOVE
```

and to give what the user types the name `MOTION`. Later the sublist `[MOTION]` appears in `:STORY1` without the prompting phrase `WAY TO MOVE`. This signals the program to fill the blank with the word or phrase named `MOTION`, without asking for a new motion.

The Procedures

The top-level procedure is `MADLIB`.

```
TO MADLIB :STORY
PRINT FILL.IN :STORY
END
```

`MADLIB` invokes `FILL.IN` and prints its output, which is a story list with the blanks filled in.

The job of `FILL.IN` is to go through the story list, one element at a time. If an element is a word, that word itself should be part of the output. If the element is a list, it has to fill a blank. Here is the procedure.

```
TO FILL.IN :STORY
IF EMPTYP :STORY [OP []]
IF WORDP FIRST :STORY
   [OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTYP BF FIRST :STORY [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```

This procedure has the overall structure of a recursive operation that does something to every element of a list.

The first instruction is the end test for the input list being empty.

The next line checks for the case in which the first element of the list is a word. In that case, we want to put the word itself in the output.

If the first element isn't a word, it's a blank to be filled. There are two cases. If the list contains more than one word, like [MOTION WAY TO MOVE], that means that the user must be asked for a WAY TO MOVE to fill the blank. The name for what the user types is the first word of the list, MOTION. FILL.BLANK handles this interaction.

```
?SHOW FILL.IN [ [MOTION WAY TO MOVE] QUICKLY! ]
TELL ME A WAY TO MOVE
PERAMBULATE
[PERAMBULATE QUICKLY!]
?
```

If the first element is a list that has only one word, like [MOTION], then we use the word or phrase that was remembered under that name.

```
?SHOW FILL.IN [HELLO, [PERSON NAME];HOW IS [PERSON] TODAY?]
TELL ME A NAME
JONATHAN
[HELLO, JONATHAN ; HOW IS JONATHAN TODAY?]
?
```

The last line of FILL.IN provides the output for both kinds of sublists.

Filling Blanks by Asking Questions

FILL.BLANK has two tasks: it asks the user for a word or phrase, and it gives what the user types a name.

```
TO FILL.BLANK :BLANK
PR SE [TELL ME] ARTICLE BF :BLANK
MAKE FIRST :BLANK RL
END
```

By the way, this is a good example of the use of MAKE with a first input that is not a quoted word. The name of the variable we want to set is part of the story list and does not appear in the text of the procedure.

An elegant detail of FILL.BLANK is that it figures out whether to use A or AN in prompting for a word or phrase. Here is the subprocedure that does the figuring.

```
TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE "AN :PROMPT]
OP SE "A :PROMPT
END

TO VOWELP :LETTER
OP MEMBERP :LETTER [A E I O U]
END
```

Handling Punctuation

If a blank to be filled is the last thing in a sentence in the story, there is the problem of putting a punctuation mark at the end, without making it a separate word. For example, in our story we have a sentence that ends

```
SAW [PERSON] .
```

If the variable PERSON contains the word URSULA, we'd like the finished story to end

```
SAW URSULA.
```

But if we don't treat this as a special case, the period will be a word by itself:

```
SAW URSULA .
```

The solution I chose is to use an asterisk in the story to mean "take the next two elements in the list and combine them as one word." That's a slight simplification, though, because the next element may be an entire phrase, and only the last word of the phrase can be combined with the punctuation character that follows. The procedure that does the combining is this.

```
TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF :PUNCT
END
```

Here is a revised version of FILL.IN that uses PUNCTUATE.

```
TO FILL.IN :STORY
IF EMPTYP :STORY [OP []]
IF EQUALP FIRST :STORY "* [OP SE (PUNCTUATE FILL.IN
        (FPUT FIRST BF :STORY []) FIRST BF BF :STORY)
        FILL.IN BF BF BF :STORY]
IF WORDP FIRST :STORY
   [OP FPUT FIRST :STORY FILL.IN BF :STORY]
IF NOT EMPTYP BF FIRST :STORY [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN BF :STORY
END
```

PROGRAM LISTING

```
TO MADLIB :STORY
PRINT FILL.IN :STORY
END

TO FILL.IN :STORY
IF EMPTYP :STORY [OP []]
IF EQUALP FIRST :STORY "* [OP SE ▸
     (PUNCTUATE FILL.IN (FPUT FIRST BF ▸
     :STORY []) FIRST BF BF :STORY) ▸
     FILL.IN BF BF BF :STORY]
IF WORDP FIRST :STORY [OP FPUT FIRST ▸
     :STORY FILL.IN BF :STORY]
IF NOT EMPTYP BF FIRST :STORY ▸
     [FILL.BLANK FIRST :STORY]
OP SE THING FIRST FIRST :STORY FILL.IN ▸
     BF :STORY
END

TO FILL.BLANK :BLANK
PR SE [TELL ME] ARTICLE BF :BLANK
MAKE FIRST :BLANK RL
END

TO VOWELP :LETTER
OP MEMBERP :LETTER [A E I O U]
END
```

```
TO ARTICLE :PROMPT
IF VOWELP FIRST FIRST :PROMPT [OP SE ▸
     "AN :PROMPT]
OP SE "A :PROMPT
END

TO PUNCTUATE :STUFF :PUNCT
IF WORDP :STUFF [OP WORD :STUFF ▸
     :PUNCT]
OP SE BL :STUFF WORD LAST :STUFF ▸
     :PUNCT
END

MAKE "STORY1 [AT * [HOUR TIME OF DAY] ▸
     , [PERSON PERSON'S NAME] WAS * ▸
     [MOTION WAY TO MOVE] ING DOWN THE ▸
     STREET. [PERSON] SPOTTED A ▸
     [ANIMAL ANIMAL YOU FEAR] DIGGING ▸
     IN A GARBAGE CAN ACROSS THE ▸
     STREET. [PERSON] BEGAN TO ▸
     [MOTION] IN THE OPPOSITE ▸
     DIRECTION BUT IT WAS TOO LATE. ▸
     THE [ANIMAL] SAW * [PERSON] . IT ▸
     BEGAN TO CHASE * [PERSON] . ▸
     [PERSON] TRIPPED AND FELL. THE ▸
     [ANIMAL] CAME UP BESIDE [PERSON] ▸
     AND BEGAN TO WAG ITS * [ANATOMY ▸
     BODY PART] . [PERSON] REALIZED ▸
     THERE WAS NOTHING TO FEAR. ▸
     [PERSON] REACHED OUT AND PATTED ▸
     THE * [ANIMAL] .]
```