www.logofoundation.org

Conversations with Logo
(as overheard by Michael Tempel)

PART I

Person: I'm having some trouble with my Logo: program.

Logo: What seems to be the problem?

Person: Well, I'm trying to position the turtle at some random place on the screen.

Logo: That's easy enough. Go ahead.

Person: **`setpos [random 80 random 80]`**

Logo: *setpos doesn't like [random 80 random 80] as input*

Person: Why not?

Logo: Because **setpos** likes a list of two numbers as input. It uses the first number to set the turtle's x-coordinate and the second number to set the y-coordinate.

Person: I know that. Sometimes it works. For example,

```
setpos [50 50]
```

There. The turtle moved up and to the right.

Logo: Of course.

Person: Of course. So when I say **setpos [random 80 random 80]** I'm giving **setpos** a list of two numbers, only I'm asking **random** to pick them for me. Right?

Logo: Wrong. You're giving **setpos** a list of four words, not two numbers.

Person: Huh? What four words?

Logo: **random**, **80**, **random** and **80**

Person: But that's not what I mean. I want each **random 80** to report a number. They're in a list because of the brackets so **setpos** should be happy.

Logo: Well that's not how I do things, but if you want, I can change the way I interpret what you say.

Person: Can you?

Logo: Sure. Try your random position setting now.

Person: `setpos [random 80 random 80]`

Logo: There. Is that what you wanted?

Person: Yes! The turtle moved and you didn't complain. Thanks.

Logo: By the way, what's your program about?

Person: It's a tutorial to explain to people how you work.

Logo: That's a good thing to do. Lots of folks don't seem too clear about what I'm doing. Can you show me some of it?

Person: Sure. It begins like this

```
to tutor
print [Forward 50 draws a line.]
print [Try it for yourself.]
end
```

Logo: OK, let's see it work.

Person: `tutor`

Logo: *I don't know how to draws in tutor*

Person: Wait a minute! What's going on here. This worked earlier today! Now you're complaining that you don't know how to draws! And why did the turtle draw that line on the screen?

Logo: What did you want to happen?

Person: I just wanted that text to appear on the screen. That's why I used the **print** command.

Logo: Oh. Well, that's what used to happen, but you asked me to change things so that **setpos** would work the way you want it to.

Person: I didn't ask you to mess with **print**!

Logo: Well, I can't do it both ways. We'll have to decide.

Person: Do what both ways? I'm confused.

Logo: I can either evaluate what's inside a bracketed list or not. I never used to, which is why **setpos [random 80 random 80]** didn't work. Even though **random** is the name of a procedure, when it's in a bracketed list I just take it literally as a word. I don't run the procedure. This makes **setpos** unhappy. On the other hand, that seems to be what you want when you use **print**. When I evaluated what was inside the brackets I asked the turtle to draw a line because the first two words in the list were **forward 50**; they make a perfectly good command. Then, I found a word that wasn't the name of a procedure so I complained. Would you prefer that I just take all those words literally?

Person: Well, I suppose so. But wait a minute! Sometimes you do run procedures in bracketed lists. What about when I say **repeat 4 [forward 50 right 90]**? You draw a square.

Logo: I don't run what's inside the brackets, **repeat** does. That's her job. She runs lists. **Print** handles lists differently. He puts their contents on the screen. I don't do anything to bracketed lists. I just make sure that they're delivered to the procedures they're intended for.

Person: OK. But I still have to set the turtle at random positions for my program to work.

Logo: It can be done. You could . . .

Person: I know! I could write a procedure, let's call it **setxy**, that takes two numbers as inputs. So, I could say **setxy 50 50** instead of **setpos [50 50]**. Then **setxy random 80 random 80** should work. Since the **randoms** aren't in brackets, they'll be run. Each **random** will report a number to **setxy**. Right?

Logo: Right. In fact there are some versions of Logo: that have a **setxy** primitive that works just that way.

Person: OK, here's my procedure

```
to setxy :x :y
setpos [:x :y]
end
```

Let me try it with some actual numbers first before trying it with **random**.

Logo: Go ahead.

Person: `setxy 50 50`

Logo: *setpos doesn't like [:x :y] as input in setxy*

Person: Gimme a break! What's the problem now?

Logo: It's the same problem. Your procedure handed **setpos** a list of two words that aren't numbers. **Setpos** needs a list of two numbers as input. He's very picky about these

things.

Person: I gave him a list of two numbers, the value of x and the value of y. Those were numbers because I used numbers as inputs to **setxy**. Right?

Logo: Wrong. **Setpos** got a list of two words as input. The first word was **:x** and the second word was **:y**. Each of them is a two character word which isn't a number. Since these words were in a bracketed list I left them alone.

Person: I see. So, there are two rules here. You don't run procedures if their names are inside a bracketed list and you don't find the values of variables if they're inside a bracketed list. Right?

Logo: Actually there's only one rule, the first one. Did you know that you never really have to use **:** at all? It's just an abbreviation.

Person: I didn't know that. Can you explain?

Logo: Sure. If you say **name 0 "black**...

Person: Wait a minute. I don't know about **name**.

Logo: Then you must know about **make**.

Person: Yes.

Logo: Well it's the same. **name 0 "black** is the same as **make "black 0**.

Person: Why have both?

Logo: Some people prefer one form, some like the other. Some folks like to switch from one form to the other depending on the context in which the command is being used.

Person: Isn't switching confusing? Don't people get the order of the inputs mixed up.

Logo: Yes.

Person: Well, since I know about **make,** can we stick with that?

Logo: If we must. I prefer **name**.

Person: Why?

Logo: Well, because . . . Say, that's another discussion. Let's just use **make** for now. When you say **make "black 0**, you make the word black the name of the number 0. Here are two more examples

```
make "flavors [vanilla [chocolate chip] strawberry]
make "greeting "hi
```

Now **thing** is a procedure that can report an object if you give its name. For example, if

you want to print a greeting on the screen you could type

```
print thing "greeting
```

See, the word hi is on the screen.

Person: I see that, but I've never seen the primitive **thing** before. I think I can see what it does though. Let me try this

```
print thing "flavors
```

Sure, you put vanilla [chocolate chip] strawberry on the screen. That's what I thought. **Print** needs an input. **Thing** gets the object with the name we gave it as input and reports that object to **print**.

Logo: Right. Once you give something a name, you can ask **thing** to report what it is by giving **thing** its name as input. You're asking for the thing, or object, whose name is the specified word.

Person: But I always say **print :flavors**. Oh I get it. You said that is just an abbreviation. It's an abbreviation for **thing**!

Logo: Not quite. It's an abbreviation for **thing "**.

Person: Oh sure. We don't say **print : "flavors**.

But what about using in a procedure like this

```
to square :side
repeat 4 [forward :side right 90]
end
```

Logo: You could write that procedure as

```
to square :side
repeat 4 [forward thing "side right 90]
end
```

and it would work just as it did before.

Person: But what about **:side** after **to square**. Can you substitute

```
to square thing "side?
```

Logo: No. The title line is special. It's not a Logo: instruction, so using the reporter **thing** in that context isn't right. You use just the word side, but it actually doesn't matter much what punctuation you use. You could write

```
to square "side
```

or

```
to square side
```

Person: Wait. I thought that using **"** means the literal word and using no punctuation indicates that you want to run a procedure. Is **side** a procedure in your last example?

Logo: No. But, remember that the line beginning with **to** is not an instruction. When I see the word **to** I assume that the next word is the name of the procedure you want to define. Any words after that on the same line I assume to be the names of inputs to that procedure. I just go by the position of the words on the line. I don't really care about the punctuation.

Person: That seems uncharacteristically sloppy of you. You're usually so precise.

Logo: Yes. I suppose you're right. I guess I should settle on some punctuation for procedure title lines and stick to it. Most people use **:** before procedure input names. Maybe I should just go with that.

Person: Actually, I think I like the idea of using **"**.

Logo: Why?

Person: Well, when you say **make "green 2**, you're using the word green as the name of the number 2. Hmm... I think I see why **name** might be better than **make**. Let me reword that. When you say **name 2 "green**, you're using the word green as the name of the number 2. When you write a procedure with inputs you're really doing the same thing.

Logo: Except that the name is only used inside the procedure. It's local to that procedure. **Name** creates global names for use by any procedure.

Person: Yes, I know. When we say **square 50** we're implicitly saying

```
name 50 "side
```

for use in the procedure **square**. If **name** uses a quoted word as input, maybe the same should be true about the names of inputs to procedures.

Logo: Well that makes sense.

Person: That was a long digression. I wanted to know why **setpos [:x :y]** didn't work. You said that it was for the same reason that **setpos [random 80 random 80]** doesn't work. Oh I see! **setpos [:x :y]** is really **setpos [thing "x thing "y]**. The rule is that you don't run procedures that are inside brackets. It's the procedure **thing** that isn't being run.

Logo: Right. When you write **setpos [:x :y]** you just disguise the fact that you're using the procedure **thing**.

Person: OK. Well I think I understand all this, but I still need a way to set the turtle at random

positions. **Setpos** wants a list of two numbers so I guess I have to get those numbers from two **random** procedures first and then put them in a list.

Logo: That's right. You could use **list** to do that.

Person: Let's see. . . **list** puts together words or lists into a larger list. How about this:

```
setpos list random 80 random 80
```

Great! You moved the turtle and you didn't complain.

Logo: **Setpos** needs a list of two numbers as input. **List** needs two objects of any sort as inputs. The two **random** procedures each give **list** a number so **list** is happy. He puts the two numbers into a list and hands them to **setpos**. Can you fix your **setxy** procedure?

Person: I think so.

```
to setxy :x :y
setpos list :x :y
end
```

Logo: Now try it.

Person: **setxy 60 80**

It works! This has been very helpful. Thanks.

Logo: You're quite welcome. Come back again if you have any other problems.

Person: **goodbye**

Logo: *I don't know how to goodbye*

Person: Oh no!

```
to goodbye
print [See you again soon.]
end
```

goodbye

Logo: See you again soon.

PART II

Logo: Well, hello again. How've you been?

Person: Fine, thanks, but I have another Logo problem.

Logo: What's up?

Person: I'm trying to write a "guess my number" game. The program "thinks" of a number between 0 and 100 and we see how many turns the player needs to get it.

Logo: 50

Person: Huh?

Logo: My first guess is 50. Is that high, low, or right?

Person: Wait! We're not playing the game yet! I just want to tell you about my problem programming it in Logo:!

Logo: Oh. Too bad. I like to play that game.

Person: Well, if you help me get the program working we can play it all you want.

Logo: Great! What's your problem?

Person: Well, here's my program

```
to game
name random 101 "number
make "guesses 1
get.answers
print [Do you want to play again?]
name readlist "answer
ifelse :answer = "yes
[game]
[print "bye]
end

to get.answers
print [What's your guess?]
name readlist "answer
if answer = :number
[(print [Right! in] :guesses "guesses)
stop]
if answer > :number
[print [Too high]
make "guesses :guesses + 1
get.answers stop]
if answer < :number
```

```
[print [Too low]
make "guesses :guesses + 1
get.answers stop]
end
```

It randomly picks a number between 0 and 100. Then it asks for a guess. If you get it, you see a "that's right" message and the game is over. If not, then the program checks to see if your guess is high or low and tells you. That's where the problem is.

Logo: Let's try it.

Person: OK.

```
game
```

Logo: `What's your guess?`

Person: `23`

Logo: *< doesn't like [23] as input in get.answers*

Person: Why not?

Logo: Because [23] isn't a number. < can only compare numbers with each other.

Person: If it's not a number, then what is it?

Logo: It's a list.

Person: But I just typed in 23. That's a number.

Logo: Yes, but **readlist** takes what you type and reports it as a list. In this case, you typed a single number, but **readlist** will read any combination of things and report it as a list. It's real flexible.

Person: Yeah, but I need to read 23 as a number.

Logo: Well, try **readnumber**.

Person: OK. I'll edit my **get.answers** procedure and change **readlist** to **readnumber**. There, now . . .

```
game
```

Logo: What's your guess?

Person: `23`

Logo: *I don't know how to readnumber in get.answers*

Person: But you told me to try **readnumber**!

Logo: Well sure, but you'll have to write it first.

Person: Thanks a lot! Where do I start?

Logo: What do you want **readnumber** to do?

Person: I want it to read what I type at the keyboard . . .

Logo: **Readlist** does that!

Person: I know, but I want it to read a number, not a list.

Logo: **Readlist** just reads what you type. It's not how it reads it that counts, it's how it reports it. You want a procedure that reads what you type and reports a number.

Person: Well I'm not quite sure what to do, but I'll get started. I'll use **readlist** . . .

```
to readnumber
output do.something.with readlist
end
```

Logo: Good start. Now **do.something.with** needs to turn the list into a number.

Person: Wait. The thing inside the list is a number. Can't I extract it?

Logo: Sure. You could use . . .

Person: **First**!

Logo: . . or **last**. If there's only one thing in the list it doesn't matter.

Person: OK. So . . .

```
to readnumber
output first readlist
end
```

Logo: Now try it.

Person: `game`

Logo: What's your guess?

Person: `50`

Logo: Too high

```
What's your guess?
```

Person: `25`

Logo: Too high

```
What's your guess?
```

Person: OK, let's stop the program. I see that it's working.

Logo: But I want to play more.

Person: Oh all right!

```
game
```

Logo:
```
What's your guess?
```

Person:
```
50
```

Logo:
```
Too low

What's your guess?
```

Person:
```
75
```

Logo:
```
Too high

What's your guess?
```

Person:
```
67
```

Logo:
```
Too high

What's your guess?
```

Person:
```
56
```

Logo:
```
Too low

What's your guess?
```

Person:
```
59
```

Logo:
```
Too low

What's your guess?
```

Person:
```
61
```

Logo:
```
Too high

What's your guess?
```

Person:
```
60
```

Logo:
```
Right! in 7 guesses
```

```
Do you want to play again?
```

Person: `yes`

Logo: `bye`

Person: Wait a minute! I said yes, I did want to play again. You said bye. What's wrong?

Logo: It's the same problem as before.

Person: Huh?

Logo: **"yes** doesn't equal **[yes]**

Person: Oh, I see, just like **23** doesn't equal **[23]**

Logo: That's right. You could fix it by . . .

Person: I know! I could change the end of **game** to

```
ifelse :answer = [yes]
[game]
[print "bye]
```

Logo: Sure. That'll work. Or instead of that change you could leave the **ifelse** command alone and change the previous line to use your **readnumber** procedure instead of **readlist**:

```
name readnumber "answer
ifelse :answer = "yes
[game]
[print "bye]
```

Person: But **"yes** isn't a number.

Logo: Well, your **readnumber** procedure is really a **readword** procedure. It reads any words, not just numbers. If **readlist** reports **[50]** then readnumber reports **50**. If **readlist** reports **[yes]**, then **readnumber** reports **"yes**.

Person: That makes sense. I should probably call the procedure **readword** instead of **readnumber**.

Logo: OK. Let's play the game some more!

Person: OK.

```
game
```

Logo: `What's your guess?`

Person: `50`

Logo: `Too low`

`What's your guess?`

Person: `75`

Logo: `Too low`

`What's your guess?`

Person: `87`

Logo: `Too high`

`What's your guess?`

Person: `81`

Logo: `Too high`

`What's your guess?`

Person: `78`

Logo: `Right! in 5 guesses`

`Do you want to play again?`

Person: `Sure! why not?`

Logo: `bye`

Person: oops!