Bull, G. L. (2020). Exploring language: The lourney from Logo to Snap!
Contemporary Issues in Technology and Teacher Education, 20(3), 577-582.

Exploring Language: The Journey from Logo to Snap!

Glen L. Bull
University of Virginia

As Paul Goldenberg notes in his reflection on the republication of *Exploring Language with Logo* as a seminal paper in the *CITE Journal*, at the time the original work was published, Logo was the only computing language with linguistic features that enabled children to explore language and linguistics in a playful way. Today there are many other choices. *Snap!,* a direct descendent of Logo, is one of the better choices for this type of exploration.

Snap! encompasses the list processing features of *Logo* described in *Exploring Language with Logo*. It can be used to explore language in the same way as *Logo*. However, because it is a block-based coding system with a web-based graphical user interface, students can devote more of their time to exploration of language and less time dealing with programming trivia that does not directly advance language learning.

In the sections that follow, the original functions first described in *Exploring Language with Logo* are compared with modern-day implementations of the same concepts in *Snap!*

Lists of Words

Modern educational programming languages like *Snap!* build upon the foundation of *Logo*. In *Logo* a list of words is enclosed in brackets:

```
[Sandy Dale Dana Chris]
```

This notation was the most convenient way of expressing a list of words in an era prior to graphical user interfaces. In *Snap!* a similar list is expressed in this way:

There are two benefits made possible by the graphical interface. The first is that the word *list* at the beginning of the list immediately lets a novice know that this sequence is a list. The second benefit is that typographical errors are reduced; it is no longer possible to create a list with mismatched brackets. This advance reduces the time that students spend debugging programming errors, enabling them to focus on exploring language.

Randomly Picking a Word

Many of the language explorations such as creation of computer-generated poetry involve the process of randomly picking a word from a list. In *Logo* a procedure to pick a word from a list would be written in this way:

```
To Pick :Object
 Output Item (1 + Random Count :Object) :Object
End
```

Once this tool was created, it could be used to pick a word from a list. (The *question mark* before the command *Pick* is the *Logo* prompt.) In this example, *Pick* has randomly picked *Dale* from the list:

```
? Pick [Sandy Dale Dana Chris]
  Dale
```

In *Snap!* a code block that can pick an item from a list is built into the language. It is no longer necessary to create this tool. In this example, the name *Chris* has been randomly picked from the list:

Snap! includes the language processing capabilities of *Logo* refined in a way that is more accessible to users. There is less to learn about computers before diving into exploration of language. This takes advantage of the capabilities offered by modern graphical user interfaces.

In *Logo* the notation made it necessary to enclose a grouped sequence of words (such as *loves to walk*) as a sublist, in another set of brackets. This example illustrates the way in which a sublist within a larger list might be indicated by a set of inner brackets within the outer brackets.

```
[cheats [loves to walk] yells]
```

In *Snap!* both the inner set of brackets and the outer set of brackets disappear altogether. Something like this is occurring within the bowels of the computer, but the user no longer has to manage these housekeeping details. This again allows the focus to be placed on language exploration rather than programming trivia.

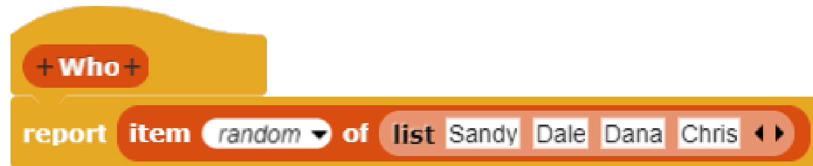
The Grammar of Gossip

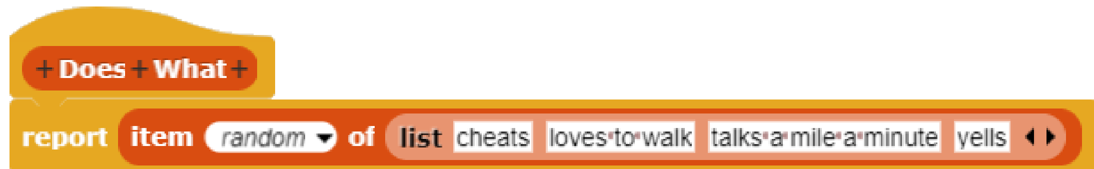
Goldenberg and Feurzeig conceived of an initial grammar for gossip as consisting of a *person* (Who) and an *action* (DoesWhat). These procedures could be written in *Logo* in the following way:

```
To Who
   Output Pick [Sandy Dale Dana Chris]
End

To DoesWhat
   Output Pick [cheats [loves to walk] [talks a
 mile a minute] yells]
End
```

In *Snap!* comparable procedures could be written in this way:





The **Who** and **Does What** code blocks can be clicked to generate gossip.

Combining Words to Form Sentences

In *Logo* the command `Sentence` is used to put lists of words together in sentences.

```
Sentence Who DoesWhat
```

In *Snap!* the code block **Join** is used to join text strings together.

As more complex sentences are created, more inputs to *Sentence* are needed. For example, the structure:

```
Who DoesWhat Who
```

might generate sentences like:

```
Dale looks for Dana
```

Because *Logo* is a text-based language, the scope of an expression was described with parentheses, as it is in the arithmetic statement $(3 + 7) \times 5$, which indicates that the scope of + applies to 3 and 7, not to 3 and the product of 7 and 5. When a *Logo* function could take a varying number of inputs, the parentheses indicated that scope.

```
(Sentence Who DoesWhat Who)
```

This led to lots of confusion about the distinction between brackets (used to designate lists of words) and parentheses (used to include more than two inputs in this instance). While this notation may have been logical from a computer programmer's viewpoint, remembering the distinction placed an additional burden on students' memory that served as a distraction from the goal of learning language.

In *Snap!* additional input slots can be created by clicking the right arrow at the right end of the Join code block.

Eliminating the need for both brackets and parentheses through use of the affordances of a graphical user interface also eliminates the need to remember the distinction between the parentheses and bracket notation.

The graphical interface also makes it possible to create more English-like structures. In Logo (and most other text-based programming languages such as Python, JavaScript, etc.) the name of a procedure is limited to a single word. Therefore, the two words of the English phrase *does what* must be combined into the single name `DoesWhat` when this term is

used to define a Logo procedure. In *Snap!* the more English-like structure **Does What** can be retained when the function is defined.

The Gossip Procedure

In *Logo* the master gossip procedure might be created in this way:

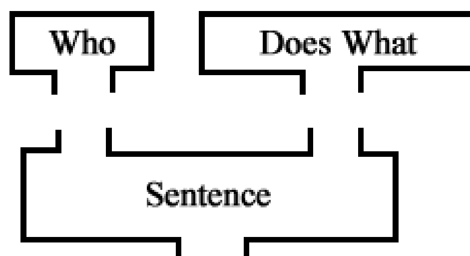
```
To Gossip
   Output Sentence Who DoesWhat
End
```

In *Snap!* the equivalent procedure could be constructed in this manner.

The master gossip procedure can be used to create sentences such as this.

Computational Thinking and Function Machines

The concept of functions exists in many symbolic systems, including mathematical, logical, and linguistic systems. In programming, a function can be described as a computer procedure that returns a value. For example, the function **Who** returns the name of a person and the function **Does What** returns an action. These values, in turn, can be combined to form sentences. The concept of function machines is introduced in *Exploring Language with Logo*. These machines are depicted in the form of diagrams in which the output of one function can serve as the input to another function. In this example, **Who** and **Does What** serve as inputs to the function **Sentence**.



This is the point at which the intersection of computational thinking and linguistic thinking occurs. An algorithm is a description of a rule or a process. For example, to form a plural in English:

1. add the consonant "s"
   - unless … in which case, add "es"
2. unless a word ends in "y"; then add "ies"
   - unless, as in "boy," you need to take another condition into account
3. unless….

Humans have limited capabilities for execution of algorithms. Our working memory is limited, and we have limited capacity for perfect execution of repetitive tasks. On the other hand, computers excel at perfect execution of a process if the process can be described. An algorithm is a recipe for execution of a process that the computer can follow. The function machines described in *Exploring Language with Logo* are graphical expressions of algorithms.

The key computational thinking concepts of *algorithms* and *abstraction* work hand in hand. Once an algorithm has been tested and has proven to be effective, it can be encapsulated in the form of an abstraction. The function **Who** is an abstraction that conceals the messy details (**Item Random List [Sandy Dale Dana Chris]**) that underlie its operation. This enables human programmers to focus on its functionality and protects them from the necessity of remembering the underlying complexity. This makes it possible to write much more complicated programs than otherwise would be possible.

Summary

Advances in technology now make it easier than ever to undertake such explorations. In contrast, the underlying pedagogical goals for such explorations have not changed at all.

One of the joys of exploring language with a computer is that this process makes it possible for unexpected results to occur. In fact, creation of a program of any degree of complexity almost guarantees that unanticipated results will occur, surprising the programmer. It is at this moment of unexpected discovery that the opportunity for learning occurs in a way that could not occur in the absence of a computer.

Contemporary Issues in Technology and Teacher Education is an online journal. All text, tables, and figures in the print version of this article are exact representations of the original. However, the original article may also include video and audio files, which can be accessed online at http://www.citejournal.org