SAND REPORT

SAND2001-3093
Unlimited Release
Printed November 2001

Visual Structure Language

Philip L. Campbell, Juan Espinoza Jr.

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2001-3093
Unlimited Release
Printed November 2001

Visual Structure Language

Philip L. Campbell
Networked Systems Survivability & Assurance Department

Juan Espinoza Jr.
Cryptography & Information Systems Surety Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87175-0785
{plcampb, jespino}@sandia.gov

Abstract

In this paper we describe a new language, Visual Structure Language (VSL), designed to describe the structure of a program and explain its pieces. This new language is built on top of a general-purpose language, such as C. The language consists of three extensions: explanations, nesting, and arcs. Explanations are comments explicitly associated with code segments. These explanations can be nested. And arcs can be inserted between explanations to show data- or control-flow.

The value of VSL is that it enables a developer to better control a code. The developer can represent the structure via nested explanations, using arcs to indicate the flow of data and control. The explanations provide a "second opinion" about the code so that at any level, the developer can confirm that the code operates as it is intended to do.

We believe that VSL enables a programmer to use in a computer language the same model—a hierarchy of components—that they use in their heads when they conceptualize systems.

(This work was developed as part of a Laboratory Directed Research and Development project, the details of which are elsewhere. [3])

Keywords: visual languages, software tools, software analysis, software assessment, software testing, analysis of programs, debugging, program understanding, reverse engineering, software maintenance.

Table of Contents

List of Figures

1 Introduction

Programmers work with systems. When asked to describe such a system, a programmer usually wants to draw boxes—programmers feel a need to create a visual representation of the system. The boxes represent components and usually there are about half a dozen of them. The programmer then draws arrows between the boxes, representing data- and/or control-flow. When asked about a given box, the programmer answers by repeating the process, drawing another set of boxes. At some point in this decomposition boxes are created that cannot be represented by another set of boxes. The contents of these lowest-level boxes is source code.

The language presented in this paper, the Visual Structure Language (VSL), is intended to increase program understanding by enabling a programmer to use in a computer language the same model—a hierarchy of components—that they use in their heads when they conceptualize systems.

The Visual Structure Language (VSL) consists of three extensions—explanations, nesting, and arcs—to a text-based language, such as C, Java, or C++. VSL provides a way to show the structure of a program, hence the name.

In the next section we introduce each of the three extensions. In the subsequent section we show how VSL might be related to other visual languages. In the final sections we present our conclusions and discuss status and future work.

2 VSL

This section introduces each of VSL's three extensions—explanations, nesting, and arcs.

2.1 Extension 1: Explanations

VSL explanations are comments that are explicitly associated by the user with sections of code. An implementation of VSL associates text with a section of code, perhaps specified by beginning and ending line and column numbers in a given file. Since the user depends on the system to create the holder of an explanation, the system can impose two particularly important constraints upon the use of explanations.

First, the system can force explanation to cover an entire semantic structure and prevent explanations from straddling semantic structure. For example, the system can force an explanation to cover an entire factor, term, expression, statement, function, or file, and never part of a factor, or part of a term and part of an expression, and so on. The system can prevent an explanation from covering only parts of two adjacent factors, say. Instead, the system could force the user to cover both factors completely or else provide two separate explanations. As a result, explanations can always map to syntactic structure. This makes this structure more visible, making the code easier to understand.

Second, the system can prompt the user to review an explanation whenever the code associated with that explanation is changed. This provides an automated way for the user to compare the new code with the previous explanation and remove disparities that may have been introduced with the change in code. As a result, the explanations have a better likelihood of remaining current. (To help, an implementation could use some visual flag, such as a different color, to identify explanations that are older than the code with which they are associated.)[1]

Depending upon the capabilities of an implementation, the user could choose to view the explanations instead of the code—that is, to have the explanations appear in a window over the screen area where the code would otherwise be displayed—or the user could choose to view the code instead of the explanations, or the user could choose to view both the code and its associated explanation, if there is one, at the same time in some arrangement that is implementation dependent.

In order to provide some precision to the above description, we will present an extended example of the use of explanations.

1. As we will see in Section 2.2, explanations can be nested. Depending upon the implementation, a code change for a nested explanation could trigger requests for review of explanations at each higher level.

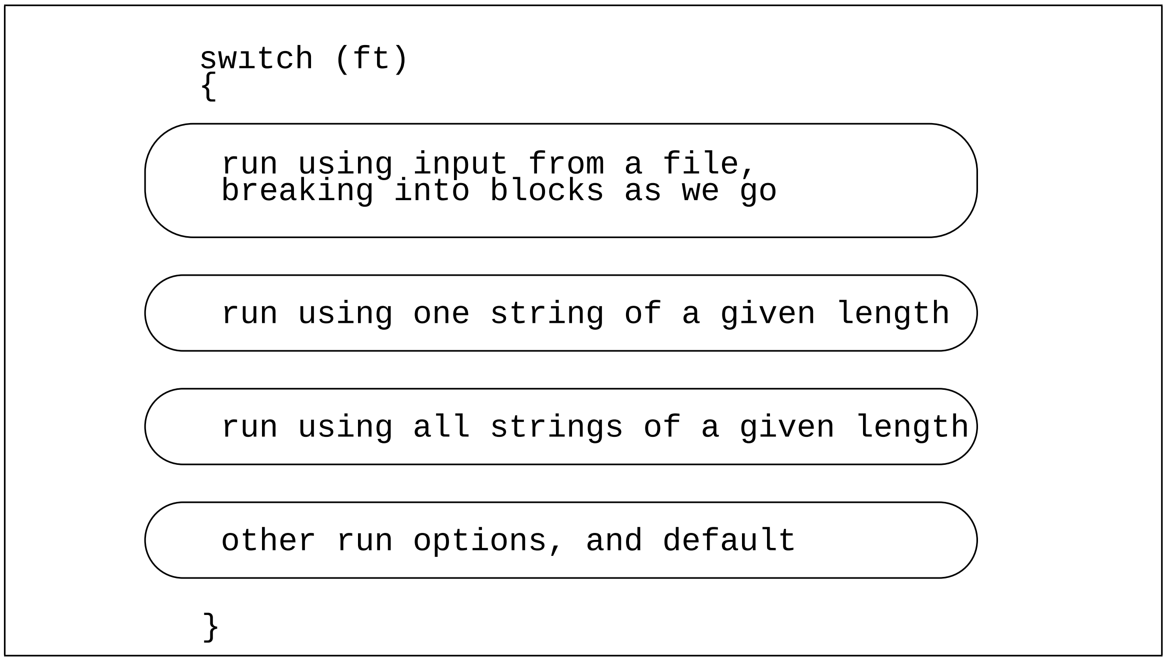
Suppose we have the SWITCH statement shown in Figure 1.

Figure 1. A SWITCH Statement

```
switch ( ft )
{
     case ( DRIVER ):
          main_driver ( argc, argv );
          break;
     case ( SINGLE_STRING ):
          run_single_string ();
          break;
     case ( ALL_STRINGS ):
          run_all_strings ();
          break;
     case ( RANDOM_STRINGS ):
          run_random_strings ();
          break;
     case ( EXAMPLE_0 ):
          run_example_0 ();
          break;
     case ( EXAMPLE_1 ):
          run_example_1 ();
          break;
     default:
          print_usage ( argc, argv );
          printf("fatal: unrecognized
function\n");
          exit(1);
}
```

We could associate explanations with it, as shown in Figure 2. We have chosen, in this and subsequent Figures, to show the explanations visually covering the associated code.

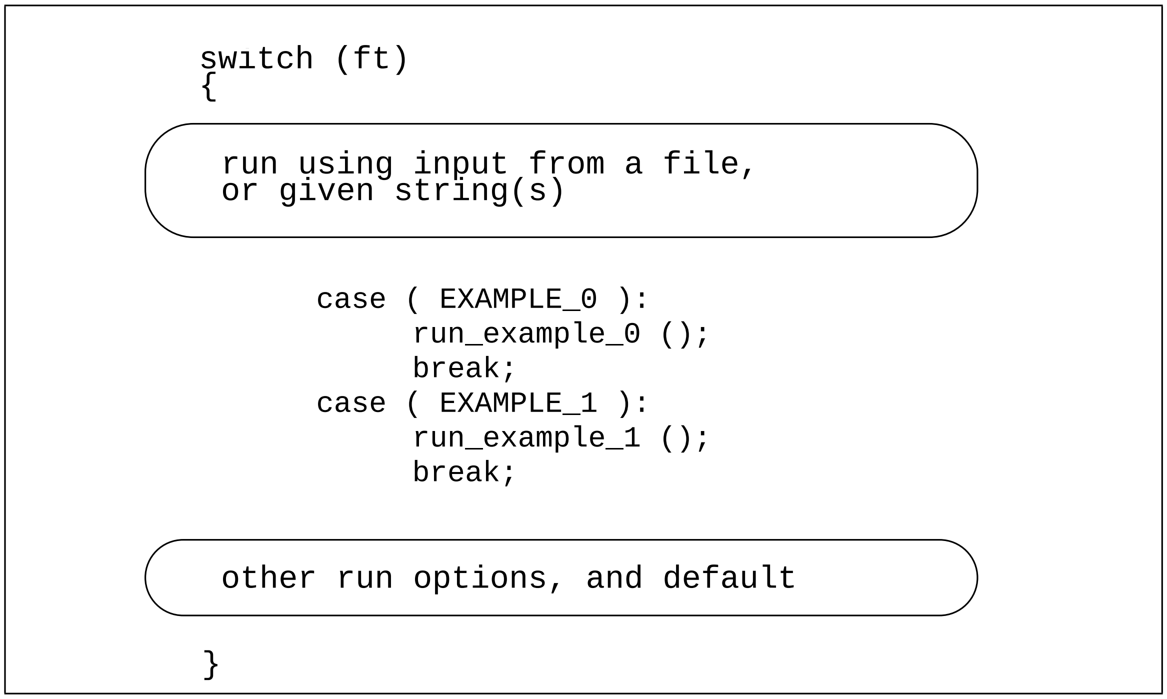
Figure 2. SWITCH Statement from Figure 1, with Explanations



```
switch (ft)
{
    run using input from a file,
    breaking into blocks as we go

    run using one string of a given length

    run using all strings of a given length

    other run options, and default
}
```

There are, of course, many different possible explanations for any given piece of code. The explanations shown in Figure 2 are only one of those possible explanations. We could associate different explanations, such as those shown in Figure 3.

Figure 3. Explanations and code, from Figure 1 and Figure 2



```
switch (ft)
{
  run using input from a file,
  or given string(s)

        case ( EXAMPLE_0 ):
             run_example_0 ();
             break;
        case ( EXAMPLE_1 ):
             run_example_1 ();
             break;

  other run options, and default

}
```

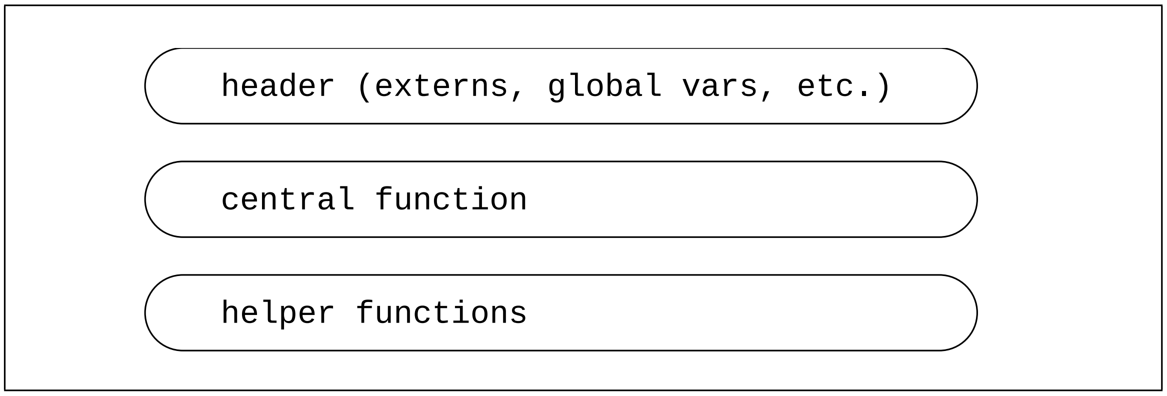
Note how the “break” statements shown in Figure 1 obscure the SWITCH statement. The syntax of C requires these statements but they get in the way of understanding. Using explanations assuages this problem.

The writing and maintenance of explanations cannot be automated. It is the user’s responsibility to write each explanation so that it reflects the intent of the code to which the explanation is associated.

2.2 Extension 2: Nesting

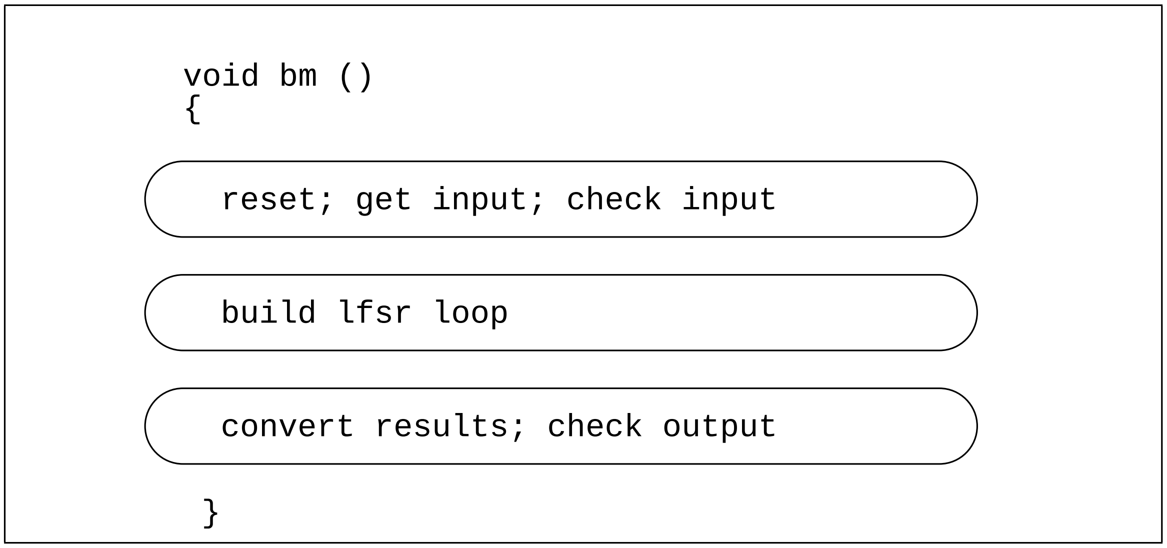
The second VSL extension is nesting. The explanations introduced in Section 2.1 can be nested to an arbitrary depth. A nesting can include code and or explanation(s). For example, a C source code file is shown in Figure 4.

Figure 4. A C File, showing only explanations



These explanations help us understand the organization of this file (or at least what the developer at one point intended the organization to be). We see the structure almost immediately: the file consists of three parts—header, central function, and helper functions. Given these parts, our next step in understanding the file would probably be to look at that central function, since this is where the work appears to be done, as shown in Figure 5.

Figure 5. Expansion of “central function” explanation from Figure 4



```
void bm ()
{
  reset; get input; check input
  build lfsr loop
  convert results; check output
}
```

Figure 5 shows a mixture of code and explanation. We note that this central function consists of three semantic parts. From the explanations, it is apparent that the “build lfsr loop” is the heart of the function, so we expand that, as shown in Figure 6.

Figure 6. Code for “build lfsr loop” explanation in Figure 5

```
while ( N < s_length )
{
     for ( i = 1, k = 0 ; i <= L ; i++ )
          k += c[i] * s[N-i];
     d = (s[N] + k) % 2;
     if ( d == 1 )
     {
          bound = N+1;
          p = N - m;
          ... <--code not shown
     }
     N++;
}
```

Without explanations, the code in Figure 6 does not make much sense, at least not without some work. However, we know that it is the heart of this program, so we know that our time is fruitfully spent right here.

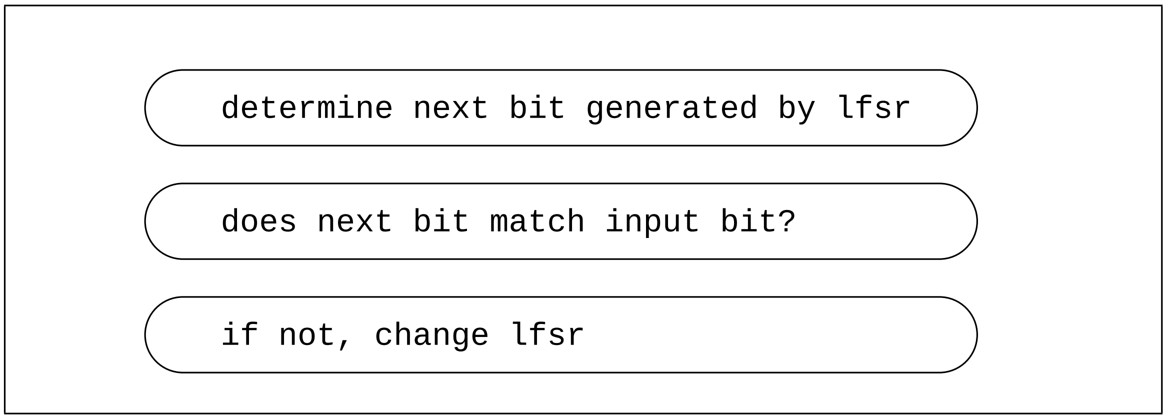
To start with, we would like to see what the FOR loop is supposed to do, and what the “d = (s[N] ...” statement is intended to do. But as it is, we are at a loss. Figure 7 shows the same code, but this time covered by one explanation for the entire while loop.

Figure 7. Explanation for “while statement” from Figure 6

```
in this while statement we build an
lfsr, one cell at a time, as we
consider each input in the input
string; as long as the bits generated
by our under-construction lfsr match
the bits in the input string, then
we are fine, but if there is a
discrepancy, then we have to change
the lfsr, possibly increasing its
length.
```

We are making headway, thanks to that explanation. We now know the basic structure of the loop. We are ready to delve deeper into the explanation in Figure 7. As we do so, we see nested explanations, as shown in Figure 8.

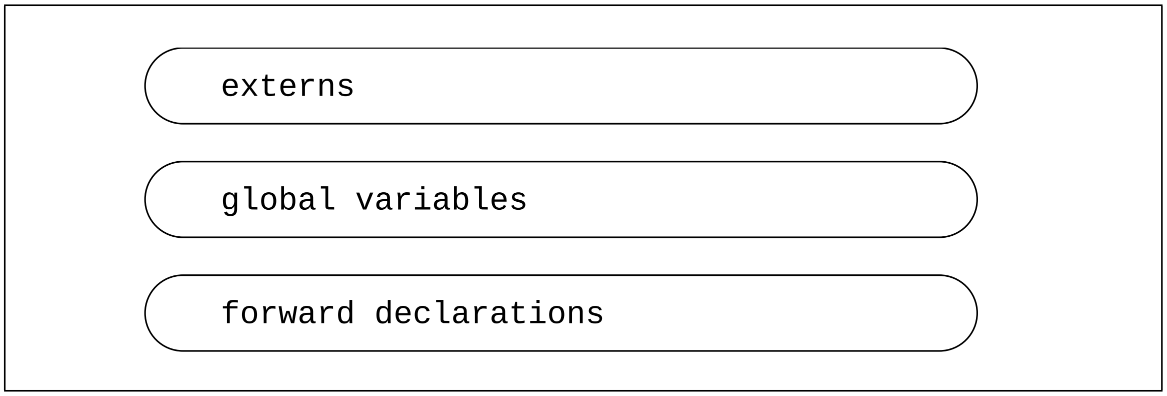
Figure 8. Explanations for statements inside the “build lfsr loop” shown in Figure 7


determine next bit generated by lfsr
does next bit match input bit?
if not, change lfsr

As we delve deeper into these explanations we will, at some point, reach the code itself. But hopefully by then we will have in mind the context and structure that give meaning to that code and allow us to determine whether or not the code is “correct,” i.e., matches the explanations.

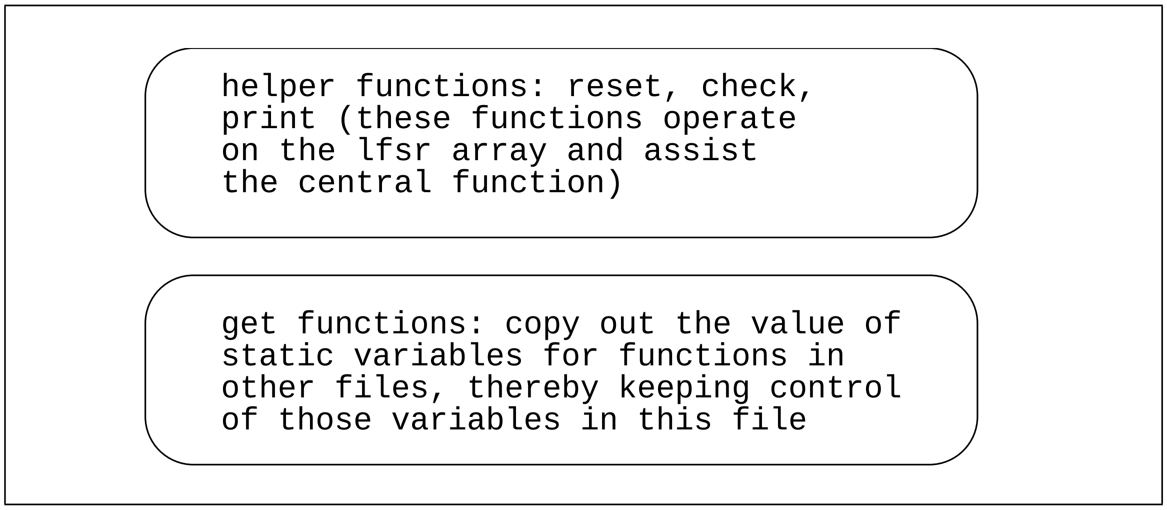
Meanwhile, let us return to Figure 4 and expand the “header” explanation, as shown in Figure 9.

Figure 9. Expansion of the “header” explanation, from Figure 4



And now let us expand the “helper functions” explanation from Figure 4, as shown in Figure 10.

Figure 10. Expansion of the “helper functions” explanation, from Figure 4



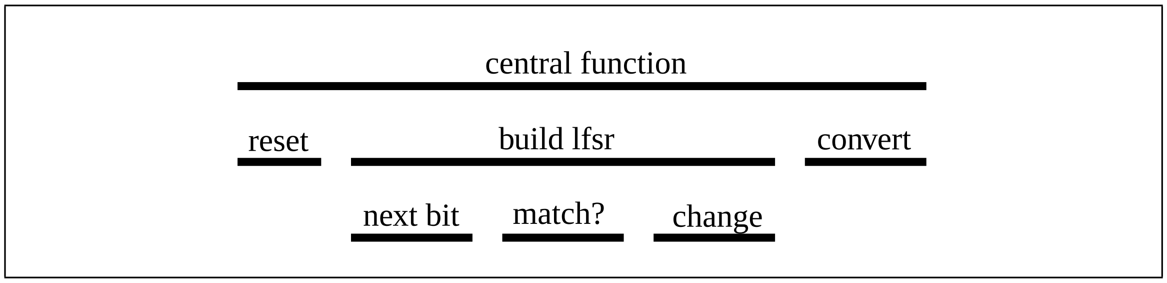
```
helper functions: reset, check,
print (these functions operate
on the lfsr array and assist
the central function)
```

```
get functions: copy out the value of
static variables for functions in
other files, thereby keeping control
of those variables in this file
```

We have seen enough from these two lower-level explanations to understand now what the code here is supposed to do. We are in a position to determine whether or not the code performs as the explanations indicate it is supposed to.

There is a final twist to explanations that can provide additional program understanding. Explanations enable an implementation to provide a “side” view that allows the user to see the layering of explanations and thereby visually understand the complexity of the code. For example, Figure 11 shows a side view of the “central function” part of Figure 4, giving us a way to see the complexity of the code.

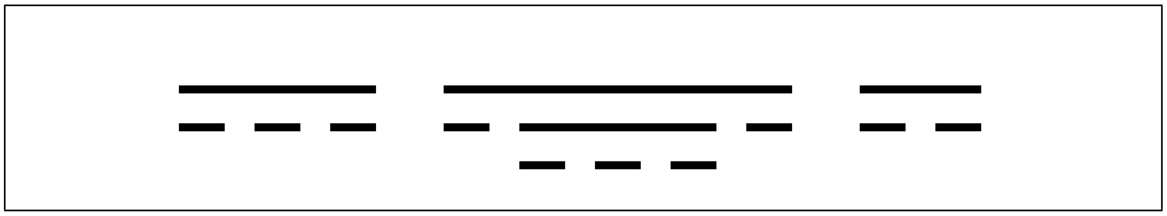
Figure 11. Side view of explanations shown in Figure 4 through Figure 8



If we remove the text from Figure 11, we can expand the scope of the side view to include

Figure 4 through Figure 10, as shown in Figure 12.

Figure 12. Side view of explanations shown in Figure 4 through Figure 10



The presentation of the side view, as shown in Figure 12, can be augmented in a number of ways. For example, the thickness of lines could indicate the depth of explanations below. The length of lines could indicate the number of lines of code covered by the explanation.

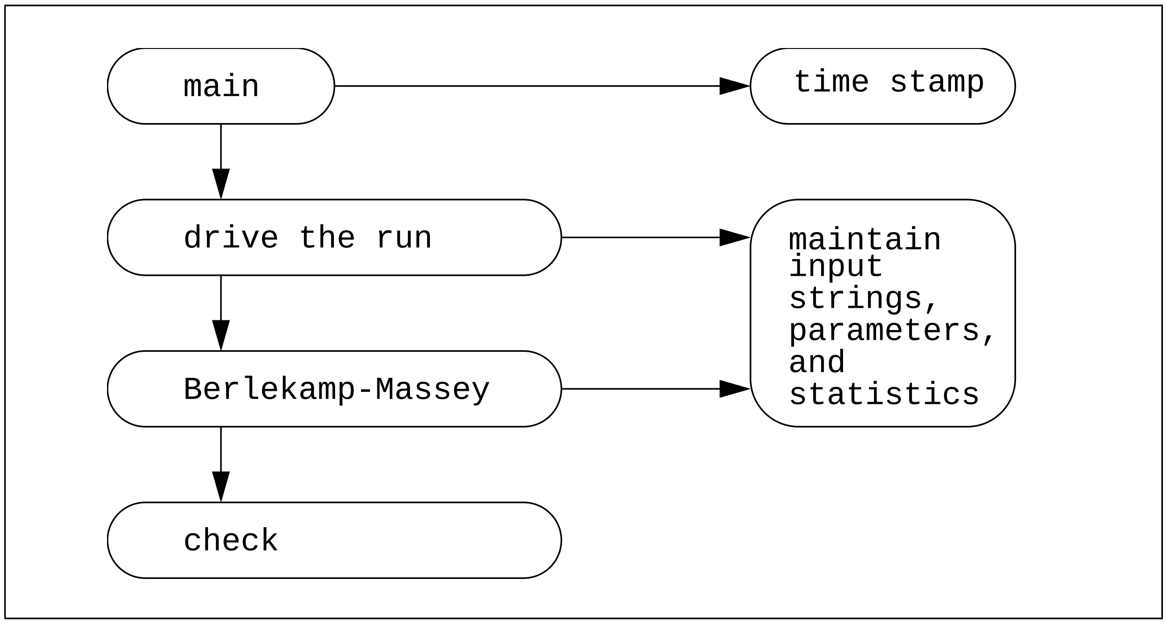
One way to think of nesting is that it provides anonymous higher-level structures. The historical progression of programming languages begins with machine language. In the first major step up from machine language, we graduate to assembly language. Even though each statement in assembly maps to one statement in machine language, assembly is still considered a step up because it uses mnemonics. Mnemonics enable the human reader to understand the code significantly faster than when it is presented in machine language. The next major step is to so-called higher-level languages. These languages are characterized by a one-to-many relationship between the number of statements in the higher-level language and the number of statements in the corresponding machine language statements. This condensation of machine language statements in higher-level languages enables the human reader to understand significantly more code (i.e., more of the machine language code) with the same effort. In addition higher-level languages can provide constructs, such as WHILE statements, that increase the human reader's understanding of the code.

Subsequent to the advent of higher-level languages, researchers have been looking for increasingly higher-level constructs. C uses the file, for example. Within a file the programmer can declare functions or variables to be invisible to code outside the file ("static"). But there is no structure in C that is at a higher-level than the file. The nesting in VSL continues where the file in C leaves off and it continues for as high as the programmer needs it to go.

2.3 Extension 3: Arcs

The third and final VSL extension is arcs. Arcs explicitly connect explanations, the code for which is presumed to be already connected by data- or control-flow. Arcs express the developer's understanding of this connection; arcs are not the result of automatic processing on the code by a compiler, for example. Arcs enable a component view of the code. Let's consider Figure 4 again. This Figure represents a file. There are nine other files in that program, comprising approximately 5,000 lines of code in all. Figure 13 shows all ten files, but does so via explanations connected with arcs.

Figure 13. Top level view of program files, with arcs



Note first of all that there are less than ten explanations in the Figure. This implies that some of the explanations cover multiple files. However, we know by the nature of explanations that no explanation covers part of one file and part of another.

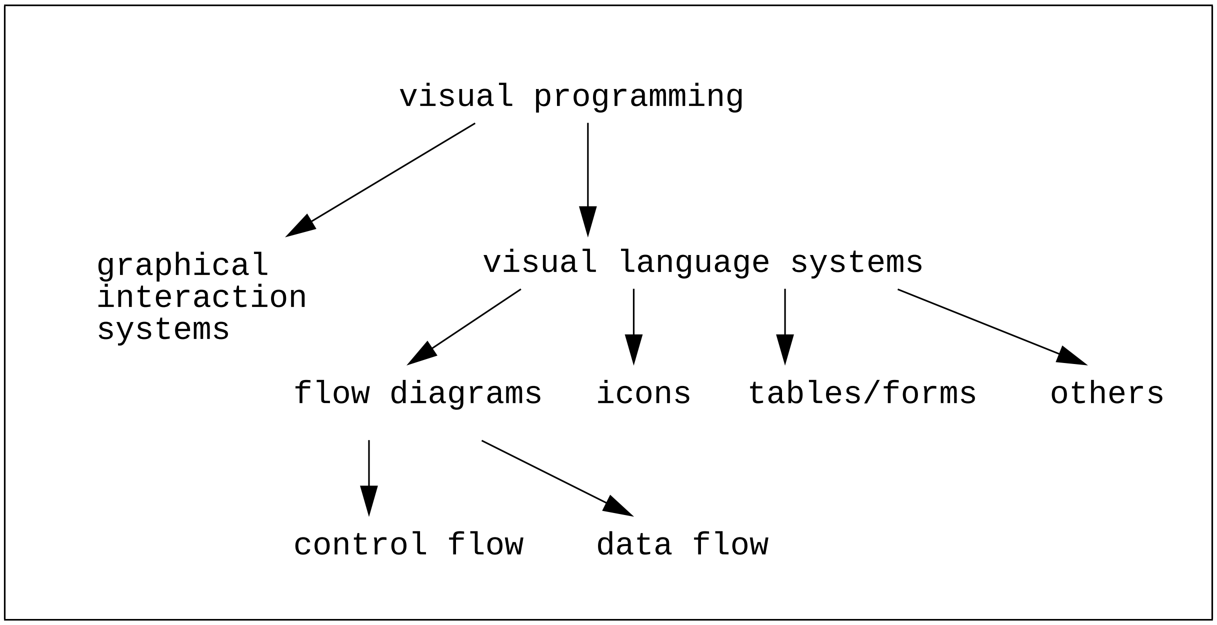
Figure 13 makes clear the high-level structure of these 5,000 lines of code. We see how the code associated with the different explanations is related (or at least intended to be related). We can now expand any of these explanations and pursue deeper levels. As we do so, more of the lower-level structure is revealed. However, since we already know the highest-level structure—given to us in Figure 13—we can more easily understand that lower-level structure. (If we expand the “Berlekamp-Massey” explanation we will, eventually come to the file represented in Figure 4.) We have not shown a difference between control- and data-flow in Figure 13, but a given implementation could provide this.

3 Related Work

We do not know of a language similar to VSL. However, inasmuch as VSL may be considered a “visual” language, we will compare and contrast it to other visual languages.

Unfortunately it is not clear what visual languages are or even if they are effective. Kiper et al.’s [4] recent paper provides a taxonomy of “visual computing,” the “visual programming” branch of which is shown Figure 14, is perhaps the best. [4] But VSL does not fit in this part of the

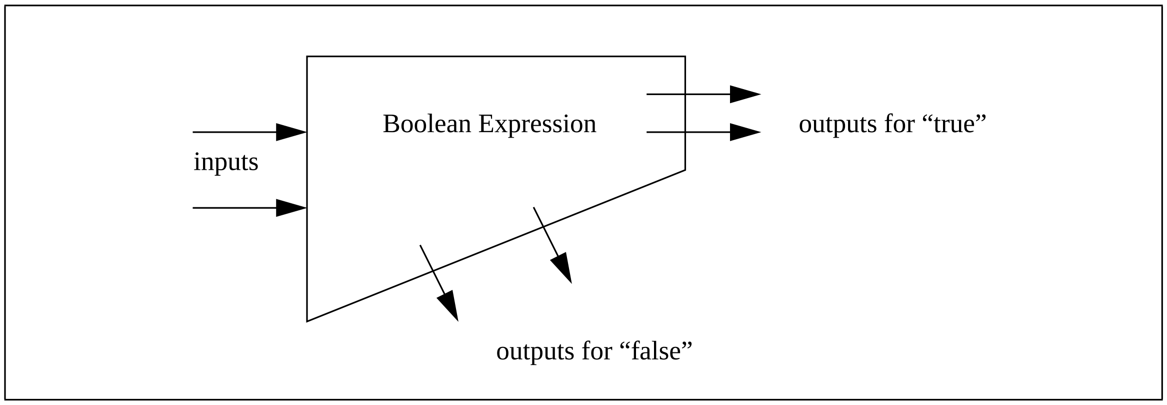
Figure 14. The “visual programming” branch of Kiper’s Taxonomy of visual languages [4]



taxonomy. There is another branch of the taxonomy, named “visualization,” that is at the same level as the “visual programming” branch shown in Figure 14. But VSL does not fit here either since VSL does not make a program visible. Rather, it enables the user to make visual the structure of the program—quite a different thing. Chang [2] [2]and, long ago now, Shu [5] [5]also grapple with the world of visual languages, but VSL does not fit easily in either of their taxonomies.

The “V” programming language, for example, is based on data-flow diagrams[1]. [1] In V, operations are replaced with visual symbols—a box with a “+” in it represents the addition operation, for example. Statements are likewise replaced with symbols—a conditional statement is replaced by a four sided figure, as shown in Figure 15. The rest of the language follows suit.

Figure 15. V conditional statement



Another visual language is LabVIEW. [6] LabVIEW [6]is similar to V in that it is also based on data-flow and has symbols for different kinds of statements.

Our general experience is that visual languages often trade one symbol system (text) for another (visual), without an obvious increase in control or expressiveness. Those familiar with assembly language programming note how baffling the code looks at first. With time the code becomes less baffling until finally the meaning of the code seems immediately evident. At that point the only thing that is still baffling is why it is confusing to newcomers. We suspect that a similar progression happens with text and visual languages.

Whitley [7] argues that the value of visual programming—whatever that may turn out to be—is still “unproven.” “This is so much the case that the empirical evidence problem warrants being called the biggest open problem in visual programming research,” Whitley writes. The field is trying to find solid ground. Empirical studies here would be helpful. Whitley concludes that “visual representations can improve human performance,” but there is not enough evidence yet to know the parameters of this improvement. We would have to agree that this same argument applies to VSL.

4 Discussion

We believe that VSL provides significant power for two reasons. First, VSL provides a “second opinion” for arbitrarily sized segments of code. This enables the developer to check that that particular segment of code functions as it is intended to do. Second, VSL provides a way to represent the organization of the code, using structures of arbitrary depth. This enables the developer to express the code’s organization so that the forest can always be visible, in spite of the trees. We believe that these two reasons provide significant power for code development and understanding.

We do not know of a language—visual or otherwise—that provides the capabilities of VSL.

5 Status and Future Work

At present VSL is a “paper” language: there is no prototype, let alone an implementation. Our next step is to build a prototype. With a prototype in hand we plan on running experiments to test the hypothesis that VSL increases program understanding.

References

[1] Mikhail Auguston, “The V Experimental Visual Programming Language. (draft) Part 1.” Technical Report NMSU-CSTR-9611, October 1996. 38 pages.

[2] Shi-Kuo Chang, “Visual Languages: A Tutorial and Survey.” IEEE Software. January 1987. pp. 29-39.

[3] Juan Espinoza, Philip L. Campbell, “Source Code Assurance Tool: LDRD Final Report.” SAND2001-3091. Sandia National Laboratories, Albuquerque, NM. Printed October 2001.

[4] James D. Kiper, Elizabeth Howard, Chuck Ames, “Criteria for Evaluation of Visual Programming Languages.” Journal of Visual Languages and Computing (1997) 8, 175-192.

[5] Nan C. Shu, Visual Programming. Van Nostrand Reinhold Company, New York. 1988. ISBN 0-442-28014-9.

[6] G. Michael Vose, Gregg Williams, “LabVIEW: Laboratory Virtual Instrument Engineering Workbench.” BYTE, September 1986. pp. 84-92.

[7] K. N. Whitley, “Visual Programming Languages and the Empirical Evidence For and Against.” Journal of Visual Languages and Computing (1997) 8, 109-142.

Distribution

1	MS	0188	LDRD Office, 1030
2		0785	J. Espinoza, 6514
1		0785	R. E. Trellue, 6514
2		0785	P. L. Campbell, 6516
1		0785	R. L. Hutchinson, 6516
1		0839	R. L. Craft, 16000
1		0899	Central Technical Files, 8945-1
2		0899	Technical Library, 9616
1		0612	Review & Approval Desk, 9612
			For DOE/OSTI