Using *LINK* to Tour Discrete Mathematics (DRAFT)

Jonathan Berry

July 6, 1998

Contents

.70.9in .30.9in .20.1in

1 Foreword

This document is a condensed version of a forthcoming book which will present concepts in discrete mathematics with examples and computations rather than theorems and proofs. It is written so that the first few sections should accessible to college students in non-quantitative majors, high school students, and perhaps even junior high school students. Teachers and professors should be able to use the later sections to extend the software with their own animations. If you are a student, get ready to do some math by drawing pictures; you'll see no lists of equations or algebraic word problems here! This material is all about problem solving, but of a different sort.

To the discrete mathematics researcher: I assure you that learning how to use the software described herein will be a worthwhile experience. Please have patience with the informal and basic writing style, perhaps skimming out the important points. I have intentionally avoided using the familiar notation associated with sets and graphs to keep the material accessible to less experienced readers. Read about theorems and proofs in a graph theory book; this document is about visualization, computation, and experimentation.

2 Introduction

This document will teach you about a kind of mathematics called *discrete mathematics*, which you can think of for now as the mathematics of objects that you can look at and count. For example, if we are talking about discrete mathematics, we could talk about a basket of apples, but we would not talk about 3.14159 apples. That word: *mathematics*! It might well send chills down your spine or immediately make you drowsy, but the “mathematics” you will see here won't be about equations, numbers, or algebra. In fact, it may be like nothing you've seen before. You'll draw a lot of pictures and see some neat animations. If you're not careful, you might even find yourself having fun. “Fun doing math!” you say? “Never!” Well, we will see.

You will also learn how to use a software system called *LINK* to play around with the concepts presented here. You can read about the history of the *LINK* project on its web page:

`http://dimacs.rutgers.edu/~berryj/LINK.html`

but that may bore you to tears, so we won't worry about that here. *LINK* makes it possible to draw nice pictures to illustrate concepts, so let's get started right now. If you need to set up *LINK* first, instructions can be found in Section 10. Hopefully, though, the system has been set up for you, somebody has shown you how to run it, and you are all ready to go.

Figure 1: *LINK*'s main window

3 The Graphical User Interface

When you run *LINK*, at least two additional windows should be created. These are the *main window*, shown in Figure 1, a *graph window* (also called a *graph view*) shown in Figure 2, and if you are running Windows, a DOS window which presents you with an `STk>` prompt (in Unix, the `STk>` prompt will appear in the original window).

Figure 3 gives a listing of some shorthand notation that we'll be using to describe mouse and keyboard actions. For example, “clicking the left mouse button” will be denoted simply as *left*, and “clicking the right mouse button while holding down the `Ctrl` key” will simply be written as *Ctrl-right*.

3.1 Creating, moving, and deleting graphics

The graph window will be our playground for viewing and tinkering with objects called *graphs*. A graph is a set of objects called *vertices* and a set of *edges*, each of which is itself a collection of *vertices*. Intuitively, we can think of a vertex as a dot, and an edge as a line which connects two dots, but later we will see that it can be important to think of edges as sets. You may have heard the word *graph* used differently before; don't worry about other possible meanings here. Let's build a graph with *LINK*. Clicking in the graph window with the right mouse button (*right*) creates vertices. Clicking on them with *left*, then selecting `delete` from the `Edit` menu (or simply typing `x`) deletes them. Go ahead and create three vertices, then try moving

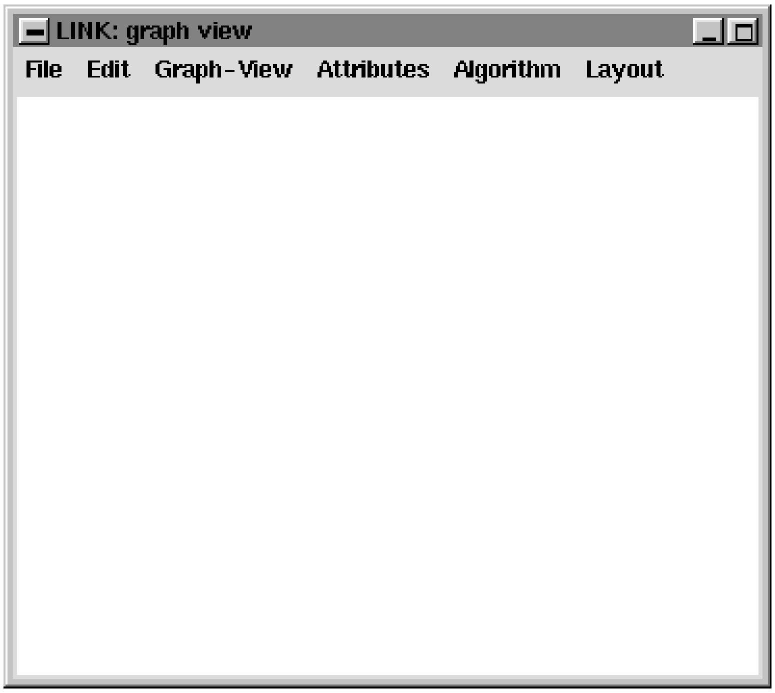

Figure 2: A *LINK* graph window

them around by clicking on them with *left*, holding the button down, and dragging the mouse. "This is mathematics??" I hear you ask. You bet it is!

The vertices of a graph can be used to represent objects, while *edges* represent relationships between objects. For example, we could think of the three vertices you have created as three athletes, while an edge between two athletes indicates that they have competed against one another before. To create an edge between two of the vertices, click on one of them with *right*, then release the mouse button and move the mouse pointer around. Note that a line will emanate from the source vertex and follow the mouse pointer. To anchor the edge to a destination vertex, move the mouse pointer on top of it and click with *right*. A thickened, labeled edge should appear between the two vertices.

Sometimes drawings of graphs can become quite cluttered, and in fact, some people spend great amounts of time trying to find good ways to draw graphs. For this document, however, we won't worry about graphs big enough to require such advanced ideas. If edges start to overlap one another, or otherwise do not look nice, you can stretch them into curved shapes by clicking with *left* on the *edge label*, or the piece of text drawn with the edge to identify it.

If during your explorations with *LINK* your screen becomes cluttered with graph windows, simply close them as you would close any other window. Also, if you want to wipe a graph window clean and start from scratch, choose `select all` from the `Edit` menu, then hit the `x` key to delete everything.

Exercises

1. Use *LINK*'s graphical interface to build the graph shown in the left side of Figure 4.

Notation	Explanation
main window	The small window that comes up when LINK is run
graph window	A window in which graphs can be drawn.
graph view	The same as Graph Window.
graph menu	The menu at the top of a graph window.
left	Click with the leftmost mouse button
right	Click with the rightmost mouse button
shift-left	Click the left mouse button while holding the shift key down
ctrl-left	Click the left mouse button while holding the ctrl key down

Figure 3: Basic GUI notation

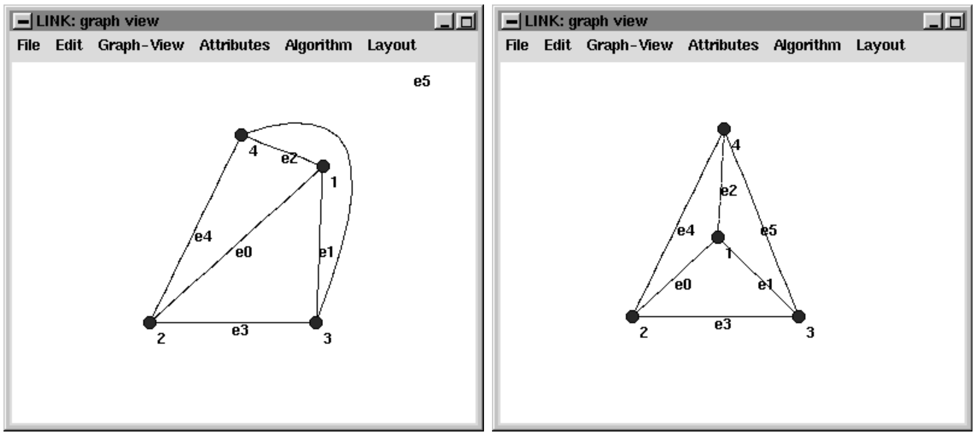

Figure 4: Exercises 1 and 2

2. Prove to yourself (by moving vertices around) that this graph is the same graph as the one shown in the right side of the figure.

3. Imagine the following situation: you are a zoo-keeper at a very small zoo, and you would like to house five types of animal: lion, tiger, antelope, emu, and chicken. Use *LINK* to draw a graph which models this situation. Hint: the animals will correspond to vertices, and possible conflicts between animals (if one would eat the other!) can be represented as edges.

4. Using the graph created in the previous exercise, can you determine the minimum number of separate habitats necessary to keep all of the animals safe? Hint: If two animals can be in the same habitat, then the vertices which correspond to those animals will have no edge between them. Try to group vertices which are not connected into the same group, then count the number of groups you have constructed.

5. Suppose that several courses plan to hold final examinations on the same day. The French exam must be held 10:00 a.m. - 1:00 p.m., the Algebra exam from 12:00 p.m. - 3:00 p.m., the Biology exam from 9:00 a.m. - 12:00 p.m., the Social Studies exam from 1:00 p.m. - 4:00 p.m., and the English exam

from 3:00 p.m. - 6:00 p.m. Model this situation with a graph and use it to determine the minimum number of classrooms necessary to hold all of the exams. Hint: an edge will represent a time conflict.

3.2 Some basic definitions

The first few pages of a book on "Graph Theory" will likely be a dizzying sequence of definitions. Since many readers of this document will have no idea of what "Graph Theory" is, and many others will know graph theory so well that they do not need to see the definitions, we will skip that routine here. This document is about computation and experimentation, not theory, so we will need only a few definitions. As stated above, a *graph* is a set of objects called *vertices* and a set of *edges*, each of which is itself a collection of *vertices*.

3.2.1 Vertex and edge definitions

To see the vertices of an edge, click on it (either the line or its label) with *left* in a graph window, then choose `select edge vertices` from the `Edit` menu Without any further clicking in the graph window, choose `colors`, then *green* from the `Attributes` menu to turn the edge and its vertices green. Two vertices connected by an edge are called *adjacent* vertices. Notice that when we choose an option from the `Attributes` menu, only certain vertices and edges are affected. A vertex or edge is called *selected* if it has been clicked on with *left*, or if it has been identified by one of the `Edit` menu options. Selected vertices and edges can be identified visually in various ways. For example, vertices might have a thick black ring around them, and edges might turn into mere outlines of their former selves. The system can be customized, so the appearance of selected objects may vary. Clicking in the white (background) area of a graph window *unselects* any selected objects within.

A given vertex may have zero or more edges coming out of it; these are called its *incident edges*. You can show the incident edges of a vertex in orange by selecting it, then choosing `select incident edges` from the `Edit` menu, then choosing `colors`, then *orange* from the `Attributes` menu. The number of incident edges of a vertex is called its *degree*. For example, Vertex 1 in Figure 4 has degree 3 (in fact, all of the vertices in that example do!). We'll do plenty with degrees later on when we start computing.

3.2.2 Directed graphs

Recall that we could think of the vertices of the graphs of Section 3.1 as athletes, tennis players perhaps, and edges as indications of whether a given pair of players have played one another. If we are really going to use this analogy, there is something bothersome about it. Imagine yourself as the referee of a tennis tournament eager to record the results. Seeing a pair of sweaty, racquet-toting young athletes approach, you might ask, "What are your names and who won?" Imagine your indignation if one of them answered, "I'm Sally Jones, she is Sue Smith, and I can tell you that we did in fact just play a match."

"But who *won*?!" you repeat. Sue Smith might answer, "I'm sorry sir, but we don't know. We did indeed play each other." This is most unsatisfactory, but edges as we know them so far simply answer *yes/no* questions about two objects: are they related to each other in some way (such as having played each other in tennis) or not?

In order to represent situations such as our tennis tournament, we can use *directed graphs*, also called *digraphs*. These types of graphs have edges which are *ordered sequences* of vertices rather than sets of vertices. Intuitively, we can think of these edges as being arrows pointing from one vertex to another rather than simply being connections between vertices. For an example, let's make a digraph to represent the following situation: Sue has beaten Sally, Sally has beaten Molly, and Molly has beaten Sue. From *LINK*'s *main* *window*, choose `Graph Windows`, then `Graph`, then `Directed`, then `No multiple Edges`. Don't worry about "multiple edges" yet.

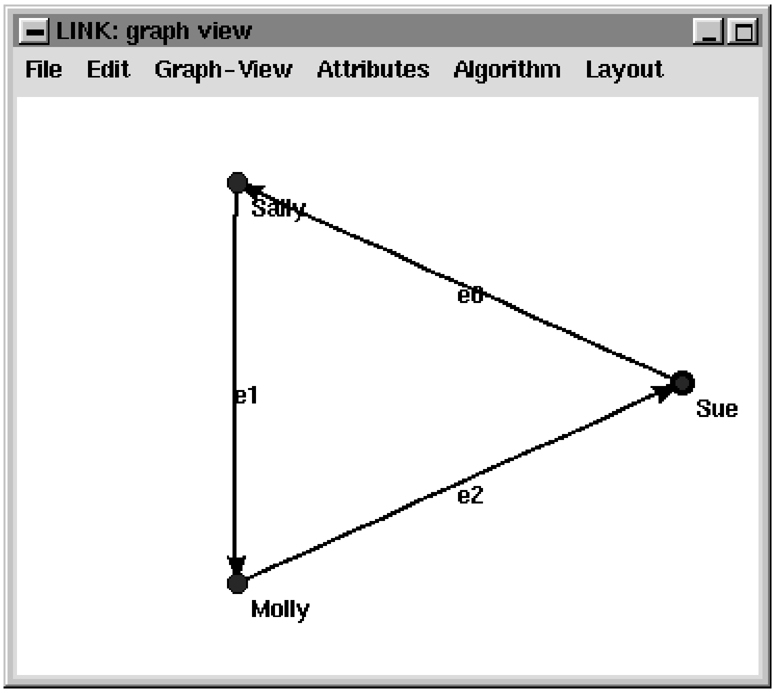

Figure 5: A digraph representing our small tennis tournament

Once you see a new empty graph window, create three vertices (remember – *right* button), then try to create an edge from Vertex 1 (let's say that's Sue) to Vertex 2 (that'll be Sally). Notice that this edge has an arrow on it. There is a little bit more information in this picture than in the previous ones: not only are 1 and 2 related in some way, but 1 comes "before" 2. Assuming that 3 represents Molly, add the remaining two edges to complete our picture of the tennis tournament. Once you've done that, choose `circular` from the `Layout` menu to arrange your digraph neatly. It would be nice to change the labels of the vertices so that they show the correspondence with our athletes, so let's do it. First, select a vertex using *left*. Let's assume you selected Vertex 1. From the `Attributes` menu, select `labels`, then `Set`. A window should appear in which you can type a name. Left click in that window, then type *Sue*, then click `ok`. Repeat this process with the other vertices, giving them proper names. Your picture should then look like Figure 5.

What are the *incident edges* of Vertex 1 (Sue)? Why, $e0$ and $e2$, just as before. What is the degree of Vertex 1? It is 2, and this number is found just as in the previous examples – count the number of edges incident to Vertex 1. With directed graphs, however, we can ask some trickier questions. For example, what do you think the terms *in-incident edges* and *out-incident edges* should mean? Rather than give definitions, I'll send you on an errand: select Vertex 1 in your digraph window, then select `in-incident edges` from the `Edit` menu. Which edges, if any, were selected? How are they related to Vertex 1? Color these edges orange. Now select Vertex 1 again and repeat the experiment with *out-incident edges*, this time coloring them blue. If you don't understand these terms yet, add a few more vertices and edges, then repeat the experiment until you understand.

The `Edit` menu has some other options involving the *neighbors*, *in-neighbors*, and *out-neighbors* of a vertex. Rather than defining these terms, I'll leave it to you to experiment with them and learn them by example. Try clicking on various vertices, then selecting one of these `Edit` menu options. Once you figure out what in-incident and out-incident edges are, you will understand the *in-degree* of a vertex, which is simply its number of in-incident edges, and the *out-degree*, which is the number of out-incident edges.

Exercises

1. Give written definitions of the *neighbors*, *in-neighbors*, and *out-neighbors* of a vertex.

2. Give written definitions of the *in-incident edges*, and *out-incident edges* of a vertex.

3. Create a graph that has 7 vertices and 12 edges, with the restriction that 6 of the 7 vertices have degree 3, while the last vertex has degree 6.

4. Create a digraph with 5 vertices and 6 edges where one vertex has 0 in-incident edges and 3 out-incident edges, one vertex has 3 in-incident edges and 0 out-incident edges, and each of the remaining 3 vertices has 1 in-incident edge and 1 out-incident edge.

3.2.3 Hypergraphs

Perhaps you are getting bored with these definitions. Maybe you are thinking, "he promised no barrage of definitions, yet I feel barraged." Well. For this section, just for a change of pace, I will *attempt* to write in a more "avant-garde" style. [1]

Math. A lonely subject for somebody like Spoyle, who had never been "good" at it. All these years of math. Pages of problems and equations he could never do. Struggle. But graphs. There hasn't been even one equation yet. Thinking that he really didn't like equations. The smart kids always seemed to get them. What would Spoyle ever need to solve an equation for anyway? Cop-out. He wanted very much to be able to solve them. Just to feel good if nothing more. Perhaps with these graphs... Who was he trying to kid? Still. Incident edges. He had seen them. Vertex neighbors. Spoyle had seen them too. Hope.

Hypergraphs. Impressive name, yet intimidating. Hyperactive graphs? Spoyle's friend Jic Nugget had been hyperactive as a child. He had a hard time picturing a "hyperactive graph" though – what would it mean? Actually, activity has nothing to do with it. In normal graphs, edges are pairs of vertices. Spolyle had seen and understood this. In a hypergraph, however, edges are groups of one or more vertices. "So you could have an edge with only one vertex?" Yes. "An edge with three vertices?" Yes. He could kind of understand this, but what would they look like?

Menus. Spoyle selected `Graph Windows` *from the main window. Hypergraph. It was there. He selected it. Undirected? Directed? Mixed? Too much. Forgetting about the others for now. He selected* `Undirected`. *Multiple edges or not? Since Spoyle had no idea what a multiple edge was, he selected* `No multiple edges`. *New empty window.* Right. *There was a vertex.* Right, *then* right, right, right. *Four more. Spreading them out. Now for one of these fancy edges. Actually, they were called* hyperedges. *Clicking on Vertex 1,* right, *then moving the mouse pointer over Vertex 2, holding the* shift *key down,* shift-right. *Moving to Vertex 4,* right, *this time without the shift key. There it was: a hyperedge. Linking vertices 1, 2, and 4. Funny; it looked like two normal edges, each having the same name. Selecting one of the two labels with* left. *Now choosing* `color`, *then red from the* `Attributes` *menu. Both of the edge segments were red.*

[1] In particular, my interpretation of the style of E. Annie Pollux, author of the Pulitzer Prize-winning novel *The Shipping News*.

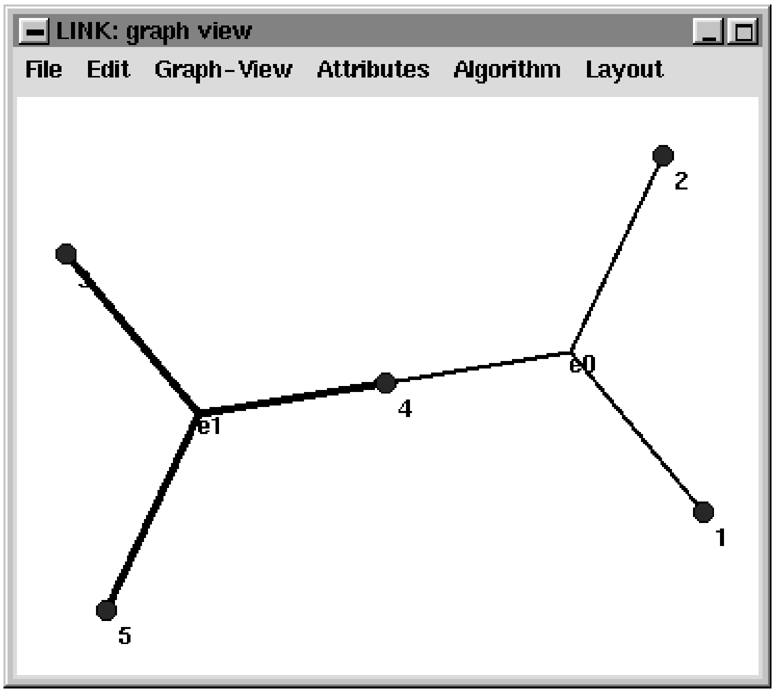

Figure 6: A hypergraph with two hyperedges, each with three vertices

"So a hyperedge is like a thread passing through a bunch of cotton balls," Spoyle thought. This concept wasn't that upsetting after all. He created another hyperedge through vertices 3, 4, and 5. For some reason, he went to the `Layout` *menu and selected* `Layouts`, *then* `Spring`. *Pretty. Just to see the difference, he selected hyperedge e1 and chose* `Edge Width` *4 from the* `Attributes` *menu. Thick. His picture looked just like the one in Figure 6.*

Exercises

1. Suppose that the relationship "in the same family" applies to blood relatives as well as immediate in-laws (such as your aunt's husband). Create a hypergraph which represents your family's ties with some other family.

2. Suppose three people went to the grocery store, and each bought a carton of orange juice, the first person bought a gallon of milk, the second person bought cookies and vegetable oil, and the third bought a bottle of wine. Create a hypergraph to model this situation.

3.3 Mixed graphs

Returning to the dry world of mathematics books and software manuals, let's consider a different type of graph: the *mixed graph*. A graph (or *undirected graph*) has edges which relate objects so that you can answer the yes/no question: are a given pair of objects related? We saw in Section 3.2.2 that *directed graphs* tell us not only if objects are related, but the order in which they are related (for example, who won and lost a tennis match). Another way to say this is that undirected graphs contain *undirected edges*, while directed graphs contain *directed edges*. A *mixed graph* is a graph which may contain both directed edges and undirected edges at the same time. For example, suppose once more that you are the scorekeeper at a tennis tournament. You might know that certain matches have been played, but only have the results for a few of them. You could represent this situation using a mixed graph. In fact, let's do it. Take the simple example we used before, in which Sue, Sally, and Molly all played each other. Suppose we only know the results of one match: Sue beat Sally.

From the `File` menu of a graph window or from the `Graph Windows` menu of the main window, create a new mixed graph with no multiple edges. Add three vertices (in the same way as before), then add an edge (as before) from Vertex 1 (Sue) to Vertex 2 (Sally). The edge will be directed. To insert an undirected edge, click on a vertex with *right*, then let go of the mouse button, move over another vertex, and click on it with *ctrl-right*.

3.4 Cloning, subgraphs, and the File menu

3.4.1 Cloning and subgraphs

To get started, let's create a *complete graph*, a graph which every pair of vertices is connected by an edge. From the `File` menu of a graph window, select `Generate`, then `Graph`, then `Undirected`, then `Complete Graph`. A small window in which you can specify the number of vertices should pop up. The default value is 5, and that will be fine for our purposes, so click the `ok` button in that window.

Once you see the graph appear, select `Clone` from the `File` menu. Another graph window will appear with a copy of our complete graph. [2] In your original graph window, select Vertices 1 and 2 in one of the following ways: either click on the background and drag a rectangular box around these vertices, or click on one, then *shift-left* on the other. The shift key lets you add vertices and edges to the current selection. From the `Edit` menu, try selecting `Collapse Selected(s)`. You should get an error message saying that the graph must be "binary" and allow "multiple edges." Well, the graph is binary (a *binary graph* is a normal graph as opposed to a hypergraph. Can you guess why the name *binary* is used?); the problem is that it is of a type that doesn't allow "multiple edges." We'll see what these are in a second.

Repeat the experiment of selecting Vertices 1 and 2 in the other (cloned) graph window. Now try to select `collapse selected(s)` from the `Edit` window. [3] Suddenly you should see only four vertices instead of five. Furthermore, some of the edge labels seem blurred, as if two labels are sitting on top of one another. In fact, they are! Click on one of the blurred edge labels with *left*, then, holding the button down, drag it a short distance. So that is what *multiple edges* are! They are simply two or more edges connecting the same vertices. If you separate the multiple edges, your graph should look like that in Figure 7. These edges are actually individuals; you can see by coloring Edge $e0$ blue and Edge $e1$ red. A graph that contains multiple edges is called a *multigraph*, while one that does not, and does not have any edge from a vertex to itself, is called a *simple graph*.

If you have colored $e0$ and $e1$ as instructed, you are ready to see how Vertices 1 and 2 disappeared. Select the new vertex (the one which appeared when 1 and 2 disappeared), then select `expand selected` From the `Edit` menu. Vertices 1 and 2 return, Edge $e0$ is still blue, and Edge $e1$ is still red. When we collapsed Vertices 1 and 2 together, it was as if the edge between them $e4$ shrunk down to nothing, Vertices 1 and 2

[2] A complete graph with n vertices is sometimes called K_n. A professor of mine once told the following joke: "How do you insult somebody with graphs? – Call him a complete graph on 9 vertices!" If you get it, I am impressed. You see, you would be calling him a K_9, or in other words, a dog! O.K., the joke flops, but I thought he was a great professor anyway.

[3] You could achieve the same effect by simply typing "s," the "keyboard shortcut." This will be true whenever you see a letter in parentheses to the right of a menu option.

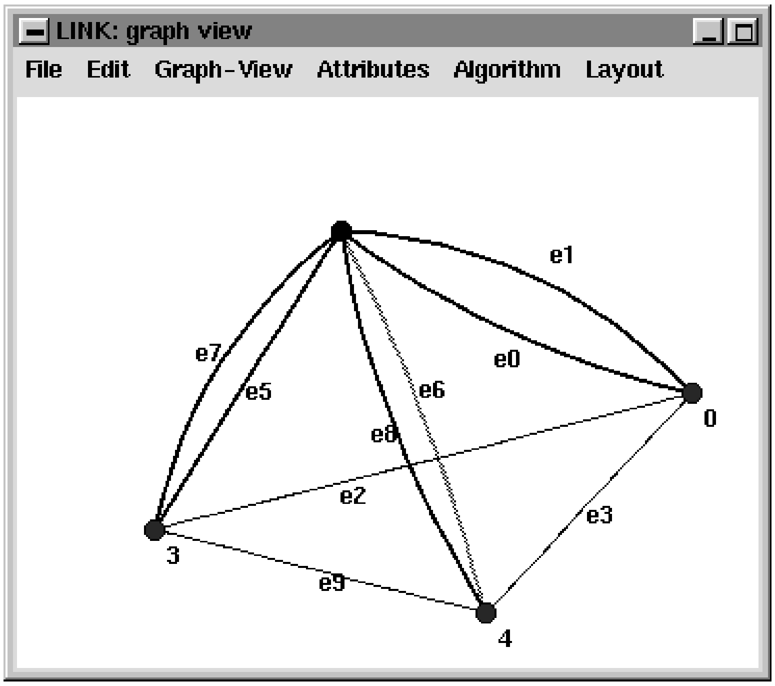

Figure 7: A *multigraph* obtained by collapsing Vertices 1 and 2 in K_5.

lay down on top of each other, and the other edges just followed along. This process is known as *collapsing* a set of vertices or an edge, or *identifying* the endpoints of an edge.

Exercises

1. Suppose we select four of the five vertices in a K_5, then select `collapse selected` from the `Edit` menu. How many vertices and edges does the result have?
2. In general, suppose we collapse $n-1$ vertices in a K_n, how many vertices and edges are in the result?
3. Generate a K_4, then collapse three of the four vertices. Select `Simple Graph Clone` from the `File` menu. Can you explain what happened?

When we collapse vertices and edges, what happens to the objects that we don't see anymore (at least until we expand them)? They are still there, just hiding. The objects which reappear when we expand are called an *induced subgraph* of a graph. To see an example of an induced subgraph, generate a K_5, select three of its vertices, then choose `induced subgraph(i)` from the `Edit` menu. You will get a new graph window containing a copy of all of the selected vertices and the edges that they "induce," i.e., all edges that join only selected vertices.

3.4.2 Saving and loading graphs

If you spend a lot of time creating a graph, you will probably want to save it so that when you come back some other day, you can just load it back in without having to recreate it. The `File` menu has options on it for saving and loading graphs. Files containing graphs in *LINK*'s format end in ".g" and can be saved and loaded using the `Save Graph` and `Load Graph` options, respectively. *It is possible to name a graph file with an extension other than ".g," but such files won't be shown in the graph loading window. It if really is necessary to name a graph file with some other extension, it can then be loaded by clicking into its directory, then typing the filename at the end.*

Try creating a graph, then saving it and loading it again. When you select `Save` from the `File` menu, a small window should come up. Replace the text in that window with the name of your graph file and make sure to keep the `.g` file extension. Some examples of good filenames are `graph1.g`, and `digraph2.g`. After saving the graph, select `Load` from the `File` menu, and the file you saved should be displayed. Click on it, then click *ok* to load it.

The `File` menu also offers you the choices of saving graphs in *DIMACS* format or as *Postscript*. Ignore the DIMACS option for now. Saving a graph as *Postscript* is a good way to print out a copy of your graph on paper. Unfortunately, this option only works on Unix at the moment. If you are using a Windows machine, you will have to use an image manipulation package like LView to save and print images.

3.5 Refreshing, redrawing, and resizing

If you resize a graph window, the area in which the graph itself is displayed does not change. In order to redraw the graph to use the (presumably bigger) area, select `redraw` from the `Graph-view` menu. If you try this with a large enough graph, you will see *LINK*'s Achilles heel; it is slow. The redrawing process may take some time, so be patient. *A common error for* LINK *users is to resize the window, but forget to select* `redraw`. *Some strange behavior will result from this omission!*

Suppose that you change colors of the edges and vertices, then want to restore them to their former neat appearance. In fact, let's mess them up on purpose to achieve just this need. Try selecting one of the *Depth-First Search*es from the `Algorithms` menu. When you do this, something called the *animation controller* should appear at the bottom of the graph window. If you can't see it, try making the window a little taller. Press the *run* button and see what happens. An algorithm is running, but you don't need to understand it now. Of course, it might be fun to experiment more and figure it out for yourself. Press the *done* button, then go to the `Graph-View` menu and select `refresh graphics`. The colors and labels should return to their initial values.

3.6 Graph operations

In advanced mathematics, there is much talk of *operations*. In fact, the important field of *abstract algebra* explores operations like addition and multiplication without even using numbers! There are several operations defined on graphs that we can see using *LINK*. One of the most fundamental of these is called graph *complement*, or complementation. The complement of a graph is another graph with the same vertices, but a different set of edges: any pair of vertices that is *not* connected by an edge in the original graph *is* connected by an edge in the complement graph. For example, consider the graph on the left side of Figure 8 and its complement on the right side of the same figure.

Try creating a graph in *LINK* with 5 vertices and 4 edges. Can you tell how many edges will be in the complement? Make a guess (or prove it to yourself with a logical argument), then click in the graph window with *left*, then select `Complement` from the `Graph Operations` menu of the main window. A message window should appear instructing you to click in a graph window, then click 'ok.' Do so, and a new graph window should appear containing the complement of the original graph. Two graphs which are complements of one another are called *complementary* graphs.

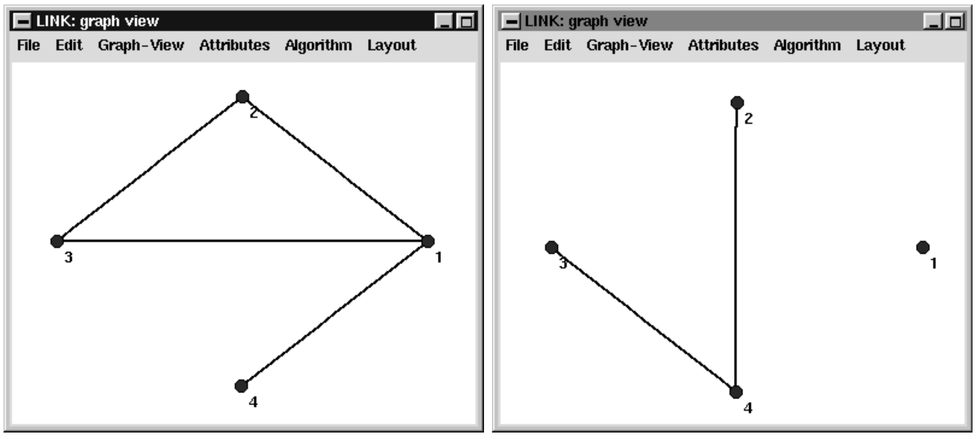

Figure 8: Complementary graphs

The complement operation is known as a *unary* operation since it only deals with one graph. An operation that deals with two graphs rather than one (or for that matter two numbers, two apples, etc.) is called a *binary* operation. The simplest binary graph operation is *graph sum*, or the "addition" of graphs. The *sum* of two graphs is itself a graph. It is constructed by combining the vertices and edges of the two original graphs. Only one copy of a given vertex will make it into the sum. For example, consider the graphs in Figures 9. The two graphs on the left side of the figure were created using *LINK*'s graphical interface, then

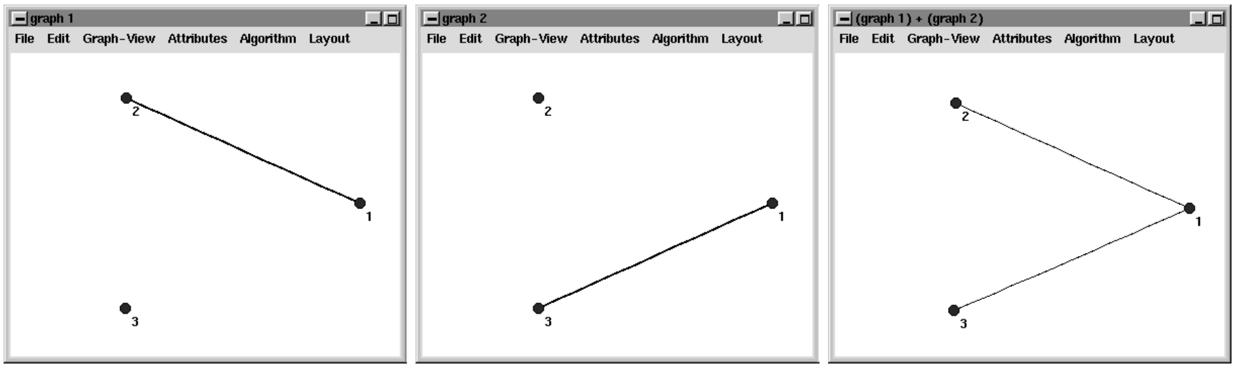

Figure 9: The sum of the left two graphs is the graph on the right

the graph sum was created by selecting `Sum` from the `Graph Operations` menu on the main window.

It is also possible to take the *product* of two graphs, though this operation is more complicated. Consider the example in Figure 10: graph 1 has two vertices and one edge, while graph 2 has three vertices and two edges. In order to construct the product, which again is itself a graph, we make a copy of graph 1 for each vertex in graph 2. In this case, we make three copies of graph 1. Now comes the tricky part: since vertex 1 is adjacent to vertex 2 in graph 2, we must connect the copies of graph 1 representing these vertices in the product graph. Look at the Figure to see how these connections are made. Notice that the vertex labels in the product graph are actually pairs of numbers. For example, the lowermost two vertices are labeled

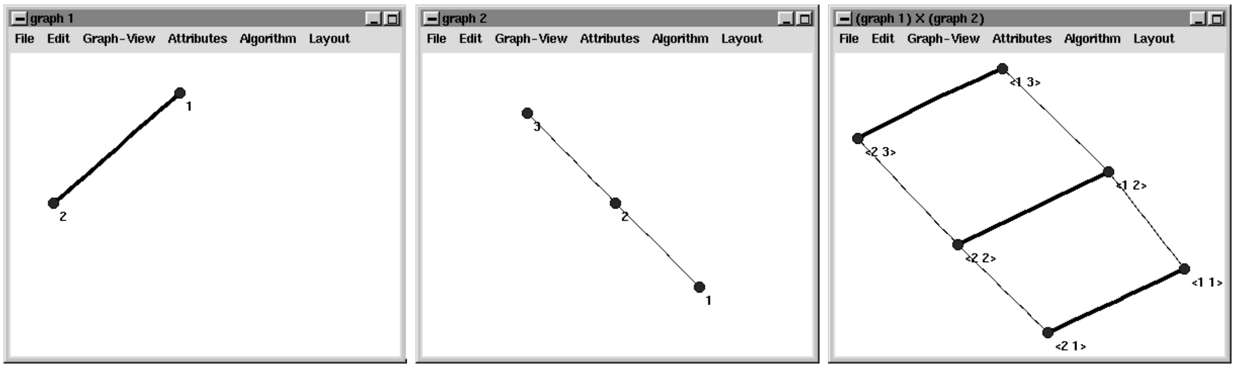

Figure 10: The product of the left two graphs is the graph on the right

$< 2, 1 >$ and $< 1, 1 >$. Within each label, the first number represents the vertex number inherited from graph 1. The second number in each pair indicates which copy of graph 1 in which the vertex resides. The two vertices we have singled out are vertices 1 and 2 in the first copy of graph 1.

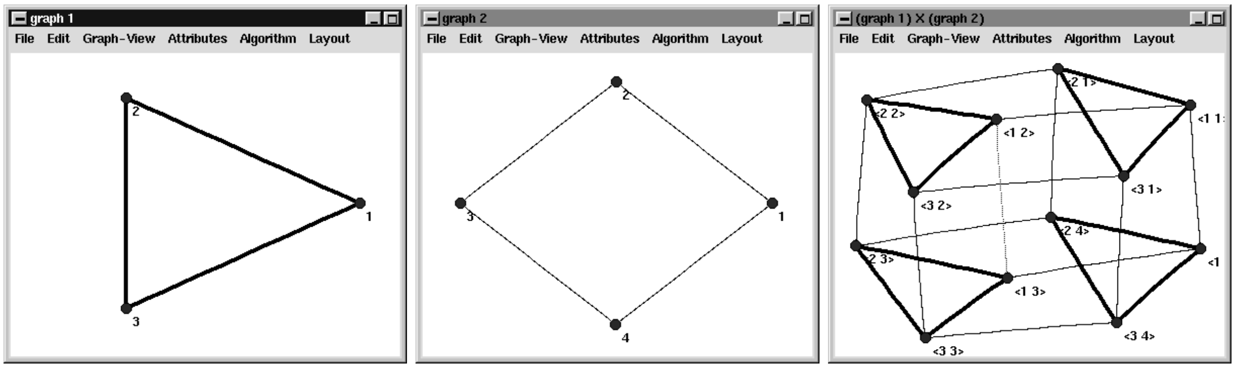

Figure 11: The product of the left two graphs is the graph on the right

Once you understand this vertex naming scheme, it is possible to understand how the connections between the copies of graph 1 are made. To see the connection scheme, create graphs 1 and 2 in *LINK*, then create the product graph with the `Product` option of the `Graph Operations` menu on the main window. Once the graph comes up, choose `Spring` from the `Layout` menu. Next, *collapse* (see Page 10) the vertices in each copy of graph 1 into one vertex. Voila! If you've done it right, you are looking at something that looks very much like graph 2. Try expanding the collapsed subgraphs one by one. Notice that connections are made between vertices representing *the same* vertex in graph 1.

When writing in formal mathematical language, the definitions of the graph operations described above can be expressed so accurately and concisely that they would all fit on about three lines of text. Unfortunately, this conciseness (and experienced mathematicians would say "beauty!") is not possible without introducing a notation which requires a high level of mathematical reading skill. I will refrain from making formal definitions here to avoid this, but I feel confident that you can understand the concepts by experimenting with them in *LINK*. I leave you with Figure 11 to begin your experiments.

Exercises:

1. What sort of graph results from adding a graph to its complement?
2. What sort of graph results from adding a graph to itself?
3. Describe the product of a graph with itself.
4. If a graph has n vertices and m edges, then how many edges does its complement have?
5. Describe two graphs whose product is one of the original graphs.
6. Describe a graph whose product with itself is itself.

4 Scheme basics

You have just had an introduction to some basic graph terminology and the use of *LINK*'s graphical interface. If you are familiar with other graphical interfaces, such as those of popular word processing and spreadsheet packages, you'll no doubt have noticed that *LINK*'s interface appears to be much simpler and more limited. Yes and no. The interface as you see it now is indeed simple. However, *LINK* is written so that you will be able to customize the interface yourself if you finish reading this book. Even better, *LINK* incorporates a standardized *command language* in which you can program repetitive operations to occur automatically.

The problem with point and click graphical interfaces in general is that in order to use them, you have to be there, move your mouse, and click buttons. It is nice when you can plan in advance everything that should happen, instruct the system to do a whole series of things, then sit back and watch (or leave altogether and go do something else) while it works. *LINK* gives you this ability by offering the *Scheme* programming language in which to give your instructions. Scheme is a small language, and you will be able to use it after learning a very limited set of commands.

When you run *LINK*, you will see in the text window an `STk>` "prompt." This is where you will type your Scheme commands. The word *STk* refers to the particular version of Scheme that *LINK* uses: STk, by Erick Gallesio of the University of Nice, France. The next few sections will cover the basics of Scheme programming and tell you how to use this command language to control your graphs. It is an interesting fact that *anything that the graphical interface does or lets you do is really done using this command language*! Knowing the command language and how to use it to affect graphs on the screen will let you program algorithms, experiment with graphs efficiently, add new menu options, etc.

The subsequent sections contain an introduction to the aspects of Scheme that will be especially helpful in our exploration of discrete mathematics. Although you will learn to compute using part of the Scheme in this document, it no substitute for a true Scheme language reference. A complete description of the Scheme standard and the STk reference manual are available, respectively, at the following ftp addresses.

```
ftp://dimacs.rutgers.edu/pub/berryj/r4rs.ps
ftp://dimacs.rutgers.edu/pub/berryj/stk-manual.ps
```

4.1 Atoms and S-expressions

Run *LINK*, then let's write a Scheme program. This one won't be very intimidating; it will have only 1 symbol in it! This little program and its output is shown in Figure 12. The number 1 in the program is called an *atom*. Scheme *interprets* each command typed in response to its prompt and returns an answer, and

```
STk> 1
1
```

Figure 12: A Scheme program of one character and its output

in this case, the interpretation is simply to recognize that the user has typed a number, and the answer is simply the same number. There are other types of atoms, such as character strings, symbols, and variables, but we won't concern ourselves with these until a little later.

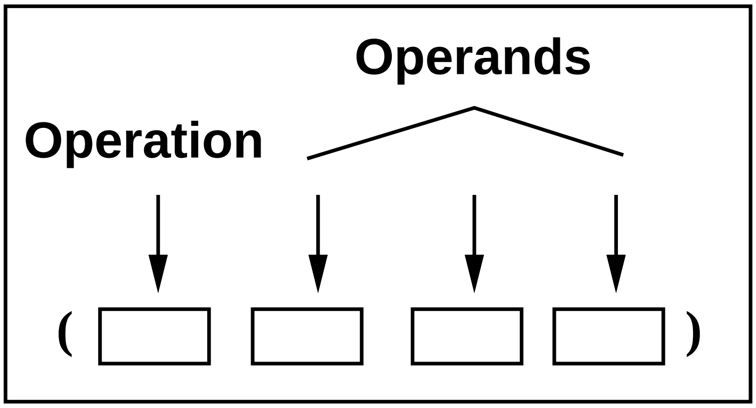

Figure 13: The interpretation of Scheme S-expressions (note that this example shows 3 operands, but in general, there can be any number of operands.

The other, more common Scheme expression is an *S-expression*, or in more familiar terms, an *expression* or *list*. Scheme lists are always surrounded in parentheses. Figure 14 shows a simple Scheme S-expression which computes the sum $1 + 1$ and returns the answer: 2. Note that there are spaces separating +, 1, and 1 (and in general, there are spaces separating all operations and operands from one another).

To understand how Scheme interprets an S-expression, consider Figure 13. The first element in any list is interpreted to be an *operation*, while all subsequent elements are interpreted as *operands*. For example, in the first expression in Figure 14, the operation is +, or addition, and the operands, are 1 and 1. The second expression in that figure has the same operation, and three operands: 1, 2, and 3.

If all Scheme programs had to be this simple, the language would not be very flexible. Of course, there is something more, but it is actually a very basic idea: an operand in a Scheme expression may itself be a Scheme expression! For example, consider Figure 15. The expression in the figure has **two** operands (not 4!). The first operand is itself a Scheme expression with 2 operands, as is the second operand. Studying this example will give you insight into the workings of the Scheme interpreter. When it interprets an expression, it *first* interprets the operands and gets a value for each one, *then* it finally applies the operation.

When expressions are located within other expressions, as in the preceding example, a computer scientist would probably refer to the situation as the *nesting* of expressions, while a mathematician would call this

```
STk>  (+ 1 1)
2

STk>  (+ 1 2 3)
6
```

Figure 14: Scheme expressions to compute $1 + 1$ and $1 + 2 + 3$

```
STk>  (+ (- 5 2) (+ 3 6))
12
```

Figure 15: The interpretation of nested Scheme expressions

the *composition* of expressions. You can think of an expression as a call to a function that returns some answer. Programming simply by writing expressions that have other expressions as operands is known as *functional programming.*

Exercises: Evaluate the following expressions in your head, then try typing them into *LINK.*

1. (+ 2 3 4 5)
2. (+ 2 (− 3 2) 4 5)
3. (+ 2 (+ 2 3 (− 4 5)))

4.2 Variables

As introduced in the previous section, functional programs consist only of function calls, though they may be nested within one another. This is a style of programming, also called a *paradigm.* Often, however, it is both more convenient and more efficient to break with this style and store temporary results in *variables.* You can think of a variable as a box which can hold a value. Variables are defined in Scheme using a function called *define.* Once in existence, variables are considered atoms in Scheme, which means that you can type a variable name (not in parentheses) at the `STk>` prompt and Scheme will return the value stored in that variable. You can change the value stored by a variable using *set!*, another Scheme function. Note that the exclamation point is in fact part of the name of the function; this is true of Scheme functions that change things.

Figure 16 shows an example of a variable definition. Notice the the *define* function returns the value `#[undefined]`. This is a special STk value used to indicate that you cannot nest calls to the *define* function within other functions. For example,

```
STk>   (define a 1)
#[undefined]

STk>   a
1

STk>   (set! a 3)
#[undefined]

STk>   a
3
```

Figure 16: Scheme expressions to define, evaluate, and set a variable

(define a (define b 2))

will result in an error since *(define b 2)* doesn't evaluate to be any value.

You can choose just about any name you like when defining a variable, as long as it doesn't have spaces in it. For example, you could name variables *billy-bob*, *rage*, or even

supercalifragilisticexpialidocious.

Good programming practice suggests, however, that you should give your variables names which give the reader some indication of the sort of value stored. For example, if you are storing the number of vertices in a graph, *num-vertices* would be a more descriptive name than *resign-Clinton-RESIGN!*. Of course, the latter would be perfectly acceptable to Scheme, as would *GO-HOME-Starr!*, but these variable names would probably not help a reader understand your code. Scheme doesn't distinguish upper-case from lower-case letters, so for example, *Num-Vertices* and *num-vertices* will be considered the same name.

Exercises: Break each of the following expressions into multiple expressions, storing intermediate results and the final result in variables.

1. (+ 2 (− 2 3) 4 5)
2. (+ 2 (− 2 3 (+ 4 5)))

4.3 Stopping interpretation with the quote

There are times when you do *not* want a list interpreted by Scheme as an S-expression. For example, you might want to construct a list of numbers, vertices, edges, graphs, etc. Scheme has a mechanism to *turn off* interpretation, or in other words, to accept a list as it is, rather than treating its first element as a function and computing something. This mechanism is the single quote character. Figure 17 gives a simple example which uses quote. The list (1 2 3) cannot be interpreted without an error since 1 is not an operation!

```
STk>    (define my-list '(1 2 3))
#[undefined]

STk>    my-list
(1 2 3)

STk>    (define my-list (1 2 3))

*** Error:
eval: bad function in : (1 2 3)
```

Figure 17: Using the quote to stop interpretation

4.4 Predicates

Knowing how lists and atoms are interpreted, how functions return values and can be nested, and how to define variables to store values, you already know how to compute a lot of results. However, there are many cases in which you will want to ask a yes/no question about something. For example, "Is a a list?" or "Is a the empty list?" or "Is a a number?" Functions which provide true/false answers to questions like this are called *predicates*. Figure 18 shows some of the most frequently used predicates in Scheme. Note that Scheme has special names for "true" and "false." They are denoted by `#t` and `#f`, respectively.

There are many more Scheme predicates, but these will suffice for our purposes. We are going to concentrate more on discrete mathematics than Scheme. One thing to note in the figure is that the *member* function is not a true predicate (that's why it doesn't end in a question mark). If the element specified is not in the list, then `#f` is returned, as we would expect. However, if the element *is* in the list, then the sub-list beginning at that element is returned rather than a simple `#t`. This information can be potentially more useful, and this behavior reveals something else about Scheme: it considers `#f` to be "false," and anything else (including but not limited to `#t`) to be "true."

4.5 Extracting elements and sub-lists

This short section will introduce you to two of the most notorious Scheme functions. Scheme is actually a dialect of the LISP language, a product of late 1950's research into artificial intelligence which remains useful today. LISP stands for "list processing," and its two most fundamental list operations are called *car* and *cdr* (pronounced "could-er"). Scheme has inherited these, and their meanings are as follows:

- *car*: Given a list, return its first *element*.
- *cdr*: Given a list, return a *list* containing all of its elements *except* the first element.

Consider Figure 19 as an example. Note that since car and cdr both return values, we can compose them to extract values from a list.

Exercises: Give compositions of calls to *car* and *cdr* to extract the following values from the corresponding lists. Check your results using *LINK*.

```
STk>  (define my-list '(1 2 3))
#[undefined]

STk>  (list? my-list)          ; is it a list?
#t

STk>  (null? my-list)          ; is it the empty list?
#f

STk>  (equal? my-list '(1 2 3)) ; do these have the same elements?
#t

STk>  (number? my-list)        ; is my-list a number?
#f

STk>  (number? 3)              ; is 3 a number?
#t
```

Figure 18: Frequently used Scheme predicates

1. Extract 2 from (1 2 3)
2. Extract 2 from (1 (2) 3)
3. Extract (2) from (1 (2) 3)
4. Extract ((2)3) from (1 (2) 3)
5. Extract 3 from (1 (2) 3)

4.6 Building lists

Section 4.5 introduced the car and cdr functions to extract elements from existing lists, but often we will want to construct new lists from existing elements. Three different Scheme functions are helpful in this regard: *cons*, *append*, and *list*. See Figure 20 for examples of the following definitions.

- The *cons* function takes two arguments, the second of which should be a list, and returns a new list identical to this list except with the first argument tacked onto the front as the new first element.
- The *append* function takes one or more lists and returns a new list containing the elements of all of the lists strung into one.
- The *list* function takes arguments of any type and returns a new list containing those arguments as elements.

```
STk>   (define my-list '(1 2 3 4))
#[undefined]

STk>   (car my-list)
1

STk>   (cdr my-list)
(2 3 4)

STk>   (car (cdr my-list))
2
```

Figure 19: The *car* and *cdr* functions

Exercises: Suppose that l is the list (1 2) and s is the list (a b). Show how to combine l, s, and individual numbers and letters using *cons*, *append*, and *list* to produce the following lists.
following values from the corresponding lists. Check your results using *LINK*.

1. ((1 2) 3)
2. (3 (1 2))
3. ((a b) (1 2))
4. (a b 1 2)
5. ((a b) 1 2)
6. ((a b 1 2) 1 2)

4.7 Loading STk(los) files

In order to customize the *LINK* interface and/or design animations to demonstrate algorithms or mathematical properties, it will be necessary to gather groups of commands into small programs. Unless they are very carefully conceived, programs (even small programs) are very unlikely to work the first time. It would be very frustrating to have to type ten lines of Scheme code into the interpreter repeatedly, but thankfully, this is not necessary. The scheme commands can be typed into a file using any text editor (e.g. *edit* or *notepad* in DOS/Windows). Note that if a word processor is used, the file should be saved as a *text* file.

The customary extensions to use for the filenames of these files of Scheme commands is either ".stk" or ".stklos." To load a file of commands, select `Load STk(los) File` from the file menu of the main window. If you get somewhat serious about extending the interface and want faster performance, you may load the file directly from the command line. Run *LINK* from the directory containing your file (suppose it is called *my-file.stk*), then simply type *(load "my-file.stklos")* to load your file.

```
STk>    (define l1 '(1 2 3))
#[undefined]

STk>    (define l2 '(a b c))
#[undefined]

STk>    (append l1 l2)
(1 2 3 a b c)

STk>    (list l1 l2)
((1 2 3) (a b c))

STk>    (cons l1 l2)
((1 2 3) a b c)

STk>    (cons 5 l1)
(5 1 2 3)
```

Figure 20: Building lists in Scheme

5 Basic *LINK* extensions to Scheme

```
STk>    (define g (graph '(1 2 3) '((1 2) (2 3))))
#[undefined]

STk>    (show-graph g)
#[<graph-view> 40244c10]
```

Figure 21: Defining and displaying a graph in *LINK*

In the past few sections, you have read about features of Scheme that are *standardized*. In other words, they will work in all correct versions of Scheme. The version of Scheme that *LINK* uses, in addition to conforming to the basic standard, allows new commands to be added to the interpreter. *LINK*'s interface consists of many such new commands, all designed to explore discrete mathematics. I will describe the most basic of these here, then go into more detail later.

```
STk>  (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk>  (describe gv)
#[<graph-view> 40244964] is an instance of class <graph-view>
Slots are:
     id = #[unbound]
     eid = #[unbound]
     parent = #[unbound]
     frame = #[unbound]
     graph-toplevel = #[<toplevel> 40240e6c]
     graph-canvas = #[<canvas> 401ef8e0]
     vertex-table = #<hash-table 40241274>
     edge-table = #<hash-table 402410ac>
     segment-table = #<hash-table 40241070>
     vertex-tag-table = #<hash-table 4024104c>
     edge-tag-table = #<hash-table 40241010>
     name = graph-view2159
     graph = #[<ubingraph*> 40241454]
     layout = circular
     xpixels = 385
     ypixels = 300
     title = "LINK: graph view"
#f

STk>  (title gv)
"LINK: graph view"

STk>  (set! (title gv) "Path of length 2")
#[undefined]
```

Figure 22: Examining a graph-view

5.1 Defining and viewing graphs

LINK is designed to manipulate objects from the study of discrete mathematics (a mathematician or computer scientist would term these *discrete structures*), so commands to build and manipulate these have been incorporated into the Scheme interpreter. The first example is the *graph* function, which can take various arguments, but at its simplest, takes two lists: a list of atoms (the vertices) and a list of lists of atoms (the edges). Figure 21 shows an example in which a graph of three vertices and two edges is created, then displayed. Notice that each of the two list operands of the *graph* function is quoted (why?). As shown in the figure, the list of edges is actually a list of sub-lists, each of which represents one edge. The *show-graph* function brings up a graph window containing the new graph and returns a *graph-view*, the name used in STk to describe one of *LINK*'s graph windows. In the next few paragraphs, I will show some of the things you can do if you store this value in a variable.

In figure 22 we see that the return value of the *show-graph* function has been saved in a variable named *gv*. This variable now "holds" the graph window! We can manipulate it and see the changes on the screen. Recall that *set!* function from Page 17 can be used to reset values of existing variables. This function can also be used to change values which are *parts* of larger objects such as a graph-views. We won't go into how these larger objects are created yet, but it is easy to look at them once they have been created. The *describe* function shows the components of any variable, even a simple variable containing the number 1. Figure 22 shows the various components of a graph-view (components of an object are usually called *fields* or *members*, though STk calls them *Slots*). Don't worry about any of these right now except *title*. Any of these fields can be retrieved by treating it as a function within a Scheme list, with the object as the only operand. A good example of this is found in the figure: we can retrieve the title field of the graph view *gv* by typing the expression `(title gv)`. Furthermore, somewhat surprisingly, a field can be reset with the *set!* function. The expression `(title gv)` can be thought of as returning a variable (the title field of gv), so the expression `(set! (title gv) "hi")` resets the value of that variable. See the figure for an example then changes the title of the graph window.

When we describe the variable *gv*, we're told that it is an "instance" of a "class" called *graph-view*. Think of a *class* as a group of values that have something in common. For example, all of the blue jays currently alive in your home state is an example of a class, as is the set of integers between -100 and 100. An *instance* of a class is simply one of its members (for example, 3 is a member of the class of integers between -100 and 100). In Figure 22, the instance is identified as:

[<graph-view> 40244c34],

which indicates the class of this instance and a unique way to identify it. The terms *object* and *instance* usually mean the same thing when we are talking about computer programming. It can be difficult to decide exactly what should be in a class, and what information should be stored within each object. The process of doing this is called *object-oriented design* and is of major importance in the software industry.

```
STk>   (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk>   (define vs (vertices gv))
#[undefined]

STk>   (define v1 (car vs))
#[undefined]

STk>   (define v2 (car (cdr vs)))
#[undefined]

STk>   (set! (color v1) "green")
#[undefined]

STk>   (set! (color v2) "red")
#[undefined]
```

Figure 23: Retrieving and coloring the vertices of a graph-view

Of course, there are more exciting things to do than to change the title of a window. The vertices and edges of the graph displayed in a graph-view are readily accessible, and it is easy to change their *attributes* such as color, size, and label. The functions *vertices* and *edges* return lists of a graph's vertices and edges, and these in turn can be manipulated using *car* and *cdr*. For an example, see (and try!) Figure 23, which puts up a graph, then colors Vertex 1 green and Vertex 2 red.

5.2 Extracting vertices and edges from graphs

```
STk>   (define list1 '(a b c))
#[undefined]

STk>   (list-ref list1 0)
a

STk>   (list-ref list1 1)
b

STk>   (define g (graph '(1 2 3) '((1 2) (2 3))))
#[undefined]

STk>   (define v (vertex-ref g 0))
#[undefined]

STk>   (describe v)
#[<vertex*> 402cd148] is an instance of class <vertex*>
Slots are:
     val = 1
#f

STk>   (describe (edge-ref g 1))
#[<edge*> 402beb38] is an instance of class <edge*>
Slots are:
     val = {2 3}
#f
```

Figure 24: Extracting vertices and edges from a graph

Scheme provides us with what is sometimes a convenient way of extracting elements from a list: the function *list-ref* (short for "list reference"). *LINK*'s extensions to Scheme give us similar functions for extracting vertices and edges from a graph, called *vertex-ref* and *edge-ref*. Figure 24 shows these functions in action: the first argument is the list or graph, respectively, and the second argument is the "rank" or position of the element, vertex, or edge. Notice that these ranks start at 0, not 1. This can be confusing when the *names* of the vertices or edges do not match their rank. For example, in the figure, the vertex *named* "1" is the first vertex in the list, i.e., it has *rank* 0.

5.3 Graphics objects versus graph objects

Until Page 22, we had been considering vertices and edges to be objects that appear on the screen. Something happened in Figure 21 to change that view though. We defined a *graph* with the following line:

```
(define g (graph '(1 2 3) '((1 2) (2 3))))
```

Before we displayed the graph g with the next command, it lived only in *LINK*'s memory, and was not visible in picture form. It is true that we could have used the command *(describe g)* to see lists of its vertices and edges, but still, there would be no blue circles or black lines on the screen. It is important to understand that the the circles and lines which appear on the screen are simply drawings of the *real* vertices and edges, which exist only in our and *LINK*'s memories! Here is an interesting mathematical thought to ponder: the *real* objects we care about in this case are actually imaginary; the things we actually see on the screen are just pictures to help our imagination! The Greek philosopher Plato argued in this way that mathematical ideas are indeed "real." If the world (or for that matter, *all worlds*) were to end tomorrow, mathematical objects like graphs would still be "there."

Plato and deep thoughts aside, what does this have to do with discrete mathematics and using *LINK*? The answer is that it will soon become important to keep in mind the distinction between the vertices and edges of a *graph* and the circles and lines of a *graph-view*. From now on, we'll call the circles that appear on the screen *vertex-item*'s and we'll call the lines *edge-item*'s. In past sections, we have actually been manipulating these items and calling them vertices, but now we know more: each graphical "item" is actually a picture associated with a vertex or edge. Figure 25 shows how to examine a vertex-item and its associated vertex. Notice that when we describe the vertex-item, we see all sorts of information about its *appearance*, such as color, outline, font, etc. On the other hand, when we describe the vertex, we simply find out that its value is 1 and its class (see Page 5.1) is called *<vertex*>* (don't worry about the "*").

Just as there is a correspondence between *vertex-item*'s and *vertex* objects, there is a similar relationship between *edge-item*'s and *edge* objects. I will leave it to you to experiment with these for now, and we will see plenty of examples shortly.

Exercises: Using Scheme, create and display the following graphs, then assign different colors to at least two vertices and at least two edges.

1. `{[1 2 3] {{1 2} {2 3}}}`
2. `{[1 2 3 4 5] {{1 2} {2 5} {3 4} {1 5}}}`

5.4 Graph attributes and graphical attributes

One of the main reasons that the theory of graphs is such a popular research area is that many real-life problems can be modeled using graphs. For example, suppose that a traveling salesman without much money has to visit a bunch of cities. If he knows the cost of getting from any given city to any other, how does he decide the cheapest way to go? [4] The cities can be modeled by vertices, and the travel routes between them can be represented as edges. The cost of traveling from, say, New York to Chicago is an example of an *attribute*, an individual value or characteristic, of an edge. Vertices and other objects may have attributes as well, of course (simple example: the name of the city a vertex represents).

Fortunately, *LINK* makes it easy to get and set existing attributes, and to define new attributes of vertices and edges. In fact, you have already been setting attributes, in particular the color and labels of vertices and

[4] Surprisingly, nobody knows an efficient way to get the best solution to this seemingly simple problem!

```
STk>  (define gv (current-graph-view)) ; assumes you have clicked w/left in a graph-view
#[undefined]

STk>  (define vi (car (vertices gv)))
#[undefined]

STk>  (describe vi)
#[<vertex-item> 4029ab38] is an instance of class <vertex-item>
Slots are:
     id = #[Tk-command .v2158.v2159]
     eid = #[Tk-command .v2158.v2159]
     parent = #[<canvas> 402c2324]
     cid = vertex2161
     oval-item = #[<guarded-oval> 40276044]
     txt-item = #[<text-item> 40291eb0]
     graph-view = #[<graph-view> 401b356c]
     vertex = #[<vertex*> 4029f5d8]
     coords = (346.5 150.0)
     world-coords = (0.9 0.5)
     size = 10
     vertex-label = "1"
     tags = ("vertex" "vertex2161")
     color = "blue"
     font = "-Adobe-Helvetica-Bold-R-Normal--*-120-*-*-*-*-*-*"
     text-color = "black"
     stipple = ""
     width = 1
     outline = "black"
#f

STk>  (define v (vertex vi))
#[undefined]

STk>  (describe v)
#[<vertex*> 4029f5d8] is an instance of class <vertex*>
Slots are:
     val = 1
#f
```

Figure 25: Extracting a vertex-item and its associated vertex

edges. These are called *graphical attributes* since changes in them alter the state of a graphics object such as a vertex-item or edge-item (see Page 26), but there are many other attributes that come in very handy in solving problems.

In the next two sections, I will describe the differences between setting graph attributes and graphical attributes. As a quick "pre-summary," graphical attributes will be set using the `set!` function, while graph attribures will be set using the `set-attribute!` function. The primary difference between the two is that `set!` is used to change the appearance of some object in the graphical interface, while `set-attribute!` changes some attribute associated with a vertex or edge (we'll see an example in the next section) which may or may not result in any visual change.

```
STk>    (define g (graph '(1 2 3) '((1 2) (2 3) (3 1))))
#[undefined]

STk>    (define v (vertex-ref g 0))    ; vertex "1"
#[undefined]

STk>    (find-attribute 'mark v)
0

STk>    (set-attribute! 'mark 1 v)
#[undefined]

STk>    (find-attribute 'mark v)
1
```

Figure 26: Getting and setting an integer attribute

5.4.1 Graph attributes

Unless otherwise specified, attributes of vertices and edges in *LINK* will be integer values (whole numbers, not fractions or decimals). There are three functions to manipulate integer attributes:

- `find-attribute`
- `set-attribute!`
- `new-attribute!`

An example of an attribute which comes in handy in many different situations is *mark*, typically used to indicate whether something has happened or not at a given vertex (for example, whether the traveling salesman has visited yet). All vertices and edges can have their *mark* variable set to any integer value, and Figure 26 shows how to examine the mark attribute of a vertex, then change its value. Note that the arguments to the *set-attribute!* function are, in order, the *name* of the attribute to set, the *value* to give it, and the *object* for which the change is to take effect. What does the single quote in front of the attribute's name do? If you've forgotten, see Section 4.3.

Suppose that we next thought of another use for the *mark* attribute, and decided to remember whether or not the traveling salesman has visited using a *new* attribute. Figure 27 shows how to create the new attribute, then use it. What is the second argument in the *new-attribute!* function? It would seem that the name of the new attribute and the graph whose vertices and edges are to use it should suffice. The answer is that we also provide a *default value*, or a value for the attribute to store before anybody has actually set its value.

Of course, there are plenty of occasions where we might want to associate non-integer values with vertices or edges. In the case of our puzzled traveling salesman, the travel costs themselves are not integers (what if

```
STk>    (define g (graph '(1 2 3) '((1 2) (2 3) (3 1))))
#[undefined]

STk>    (define v (vertex-ref g 0))    ; vertex "1"
#[undefined]

STk>    (new-attribute! 'visited 0 g)
#[undefined]

STk>    (find-attribute 'visited v)
0

STk>    (set-attribute! 'visited 1 v)
#[undefined]

STk>    (find-attribute 'visited v)
1
```

Figure 27: Making a new integer attribute

the cost is $159.32?). To store non-integers such as this with a vertex or edge, you must specify the type of value to be *double* [5]

Figure 28 shows how we can add an attribute to our edges called "cost" and use it to store the cost of airplane trips between cities. Notice that vertices can be given names that aren't numbers after all!

If you want to store a text string with each vertex or edge, the procedure is similar, except the word *string* is substituted for *double*. See Figure 29 for an example. There is one more type of attribute available, but I will leave that as a mystery for now. It will come up when we address graph algorithms in a subsequent chapter.

Exercises: Let's model an instance of the traveling salesman problem. Suppose that the fictional air fares of Figure 30 are the cheapest our intrepid salesman can find. In parenthesis by each fare is a fictional airline (aal, bal, cal, dal, or eal) offering the air fare.

1. Create a graph or digraph in *LINK*to represent these fares.
2. Use the graphical interface to set the edge weights.
3. Use Scheme to change the *weight* attribute of the edge representing the Chicago to LA flight to be $400.00.
4. Add two new attributes to the graph to indicate the arrival and departure airlines flown (what data type will they be?).

[5] If you take some computer science, you'll learn that computers actually *can't* store numbers like 3.14159... exactly. The term *double* is short for "double precision," a term used to refer to a way of storing decimal numbers fairly precisely.

```
STk>   (define g (graph '(boston nyc dc) '((boston dc) (dc nyc) (boston nyc))))
#[undefined]

STk>   (describe g)
#[<ubingraph*> 402c87c8] is an instance of class <ubingraph*>
Slots are:
     val = {[boston dc nyc]  {{boston dc} {boston nyc} {dc nyc}}}
#f

STk>   (new-double-attribute! 'cost 0.0 g)  ; default value is 0.0
#[undefined]

STk>   (find-double-attribute 'cost (edge-ref g 0))  ; boston-dc
0.0

STk>   (set-double-attribute! 'cost 132.22 (edge-ref g 1))  ; boston-nyc
#[undefined]

STk>   (set-double-attribute! 'cost 222.34 (edge-ref g 2))  ; dc-nyc
#[undefined]

STk>   (find-double-attribute 'cost (edge-ref g 1))  ; boston-nyc
132.22
```

Figure 28: Making and manipulating a "double" attribute

5. Figure out a cheap route through all of the cities for the salesman to fly, then set the arrival and departure attributes for each city.
6. Use Scheme to color each city into which the salesman flies on aal *red*.
7. Use Scheme to change the size of each city out of which the salesman flies on bal to be 20.

5.4.2 Graphical attributes

We saw back in Figure 23 how to change the color of vertices (and edges as well actually!). Given a graphical object such as a *graph-view*, a *vertex-item*, or an *edge-item*, the *describe* function shows a list of slots, or values, that are associated with that object. Values in individual slots can be extracted by using the slot name as a Scheme function. For example, it is a simple matter to obtain the color of a vertex-item stored in variable *vi*: type *(color vi)*. Furthermore, STk allows us to treat this expression itself as a variable, and therefore *set!* its value:*(set! (color vi) "red")*.

If a vertex has been displayed on the screen, then there is a (single) vertex-item that corresponds to it. Fortunately, given a vertex, you can retrieve the corresponding vertex-item simply by using the *vertex-item*

```
STk>  (define g (graph '(boston nyc dc) '((boston dc) (dc nyc) (boston nyc))))
#[undefined]

STk>  (describe g)
#[<ubingraph*> 402c87c8] is an instance of class <ubingraph*>
Slots are:
     val = {[boston dc nyc]  {{boston dc} {boston nyc} {dc nyc}}}
#f

STk>  (new-string-attribute! 'comment "it was ok" g)
#[undefined]

STk>  (find-string-attribute 'comment (edge-ref g 0))  ; boston-dc
"it was ok"

STk>  (set-string-attribute! 'comment "exhausting!" (edge-ref g 1))  ; boston-nyc
#[undefined]

STk>  (find-string-attribute 'comment (edge-ref g 1))  ; boston-nyc
"exhausting!"
```

Figure 29: Making and manipulating a “string” attribute

function. Figure 31 shows an example in which a vertex-item is retrieved into a variable called *vi*. At this point, any of the graphical attributes listed in the aftermath of *(describe vi)* can be changed.

Conversely, given a vertex-item (say it's called *vi*), the corresponding *vertex* is stored in the *vertex* slot, and can be retrieved in the same manner as any other slot value.

Why the distinction between graph attributes and graphical attributes? You may have developed the (correct) impression that graphical attributes are more convenient to set than graph attributes. This is because extra work went into *LINK* to allow that. There are indeed graph attributes called “color,” “width,” “size,” “x,” “y,” and “label,” but the corresponding graphical attributes (with their more pleasant access methods) can be used instead.

	Chicago	LA	Newark
Chicago		590.76 (aal)	411.23 (bal)
LA	570.34 (cal)		626.22 (bal)
Newark	300.29 (aal)	350.44 (bal)	

Figure 30: Traveling salesman exercise

```
STk>  (define gv (current-graph-view)) ; assumes you have clicked w/left in a graph-view
#[undefined]

STk>  (define vi (car (vertices gv)))
#[undefined]

STk>  (define v (vertex vi))
#[undefined]

STk>  (define vitem (vertex-item v)) ; vi and vitem are the same!
#[undefined]

STk>  (set! (size vitem) 20)        ; makes it bigger
#[undefined]

STk>  (size vi)
20
```

Figure 31: Converting from vertex-item to vertex and back again

6 More advanced Scheme programming

It is time for another playful interpretation of the writing style of another great. This time I will mimic James Fenimore Cooper, author of the leatherstocking series, which includes "Last of the Mohicans." Imagine that the setting is colonial America in the 1750's, except that there is a pentium computer, with *LINK* of course, sitting in a small clearing deep in the frontier forest.

Two men sat before the screen, and had been in such a position for many hours. Despite being apparently absorbed in this occupation, each in his own way maintained a posture which might have indicated a high level of vigilance, though this fact would only have been ascertained by the most astute observer. One of the adventurers was dark in complexion, silent in bearing, perfectly erect in posture, and immovable. His war paint would mean nothing to the reader, but in the time of our story, it clearly marked him as a chief of the noble Delaware tribe. The other, dressed in hunter's garb, was laughing in a silent way at the screen. After a moment, he turned to his dignified companion and said, "Well Sarpent, what think ye of this LINK *software?"*

"The Great Serpent is much amazed by the Bright Mystery (for so he had named LINK*)."*

"Why is that, Sarpent? I find the software not of much import, and of some annoyance."

"Hawk-eye has not understood the gifts of the Bright Mystery."

"Come, Sarpent, the graphical user intarface has a sartain attraction, but this Scheme language can be none other than a waste of time. I reason that Providence has freed me from any such consarn. It is not among my gifts to type at the keyboard, though I might boast that I have come to feel quite comfortable with the mouse."

"Nobody would say that Hawk-eye cannot sting the horse-fly with Killdeer [6] *or that he cannot create K_5 in the time that a drop falls from Niagara. The Great Serpent sees these things. But he also sees things that Hawk-eye does not."*

The woodsman was silent for a moment, and he took the time to scan the surrounding forest before answering. Slowly and softly, with some reproach, he retorted, "If I had not just heard these things, I would not recognize my oldest friend, Chingachgook, last of the Mohicans! To think that you would deny the greatest gift of one who has been called Deerslayer, Hawk-eye, and the Pathfinder! It is not boastfulness, but the statement of sartain fact to say that Providence has honored none with better eyes than mine."

"Hawk-eye still misunderstands the Great Serpent. The legend of Hawk-eye's gifts is known by all. But the Serpant can see things without using his eyes. Hawk-eye must listen to the Serpant to hear his explanation."

"Quite so, quite so. Please proceed, Sarpent."

"Hawk-eye dismisses the Scheme interface after seeing the beauty of the graphical interface and the pain of typing many commands?"

"That I do, Sarpent, that I do. I should rather we both be sent to the happy hunting grounds than type at this accursed keyboard all the day."

"The Great Serpent realizes, and Hawk-eye must as well, that Bright Mystery would only be a toy without the Great Scheme. When a thing can be done once, the same thing can be done many times."

"Aye, but Sarpent, is that not the problem? As I have related before, I do not include typing many commands from the keyboard among my gifts, though the mouse has proven friendly."

"The Serpent can see what must follow. The Great Scheme will allow us to type only one *command, but will perform feats of work like many armies of ants."*

"That would cause me to revise my opinion on the subject, Sarpent, yes, if it were true. Yet I am still not fond of typing even one *command."*

"The Great Serpent forsees that every command can be made into a mouse click instead, to honor the gifts of Hawk-eye."

"By the Creator, Sarpent, I hope ye're correct!"

"The Great Serpent forsees it."

Until now, our use of Scheme might be termed "pay as you go." That is, for every result we've obtained, we have had to type a command. Fortunately, Scheme is a complete computer language, designed to *automate* processes such as coloring red all vertices which have degree 3 or more. We will take this and other similar tasks as examples and develop solutions over the next couple of sections. A computer scientist would term what you about to learn *control structures*, because they control the flow of the many statements and alternatives that might occur in a long computation.

6.1 Making decisions

Perhaps the most fundamental control structure in any computer language is that which allows us to decide between two or more alternatives. Is a vertex red? Is its weight more that 300.05? Has our traveling salesman visited it yet or not? Expressions like these which evalute to "true" or "false" are called *boolean expressions.*

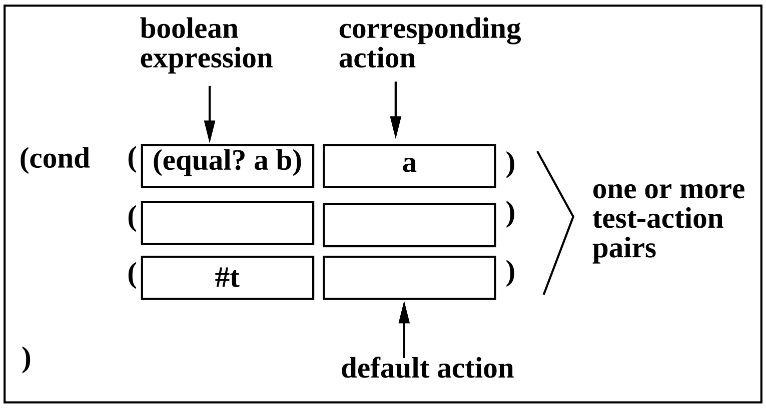

Figure 32: The format of a *cond* function call

```
STk>   (cond (  (> 2 3)  ”2 is greater than 3”)
             (  (equal? ”hi” ”bye”) ”hi is the same string as bye”)
             (  #t        ”neither of those things were true”))
"neither of those things were true"

STk>   (cond (  (< 2 3)  (+ 2 5))
             (  #t        ”2 is greater than or equal to 3”))
7
```

Figure 33: The cond statement

6.1.1 The *cond* statement

In Scheme, there is a command called *cond* which allows us to phrase our logic as follows: if expression 1 is true, then to action 1. Otherwise if expression 2 is true, do action 2. Otherwise, if expression 3 is true, do action 3, etc. If none of the expressions are true, then do a final action. The *cond* command has one argument for each test-action pair, each of which is itself a list. For example, consider the *cond* statements in Figure 33. In the first one, expression 1 is false, as is expression 2, but expression 3 (*#t*) is true (by definition!). In the second cond statement, both expressions are true, so the first action is returned, and the cond statement is finished before the second expression is even evaluated. Note that in the first cond statement, the action which “fires” simply prints a string, while in the second cond statement, the action computes something. The action in a test-action pair may be any Scheme expression.

```
STk>  (define a 4)
#[undefined]

STk>  (define b 12)
#[undefined]

STk>  (and (>= a 1) (<= a 10))
#t

STk>  (or (>= a 1) (<= a 10))
#t

STk>  (and (>= b 1) (<= b 10))
#f

STk>  (or (>= b 1) (<= b 10))
#t

STk>  (not (<= b 10))
#t
```

Figure 34: Using the logical operations *and*, *or*, and *not*

6.1.2 Boolean Expressions

As stated above, a boolean expression is one that evaluates to "true" or "false." The simplest boolean expressions are predicates (functions which return "true" or "false" – see Section 4.4) and arithmetic comparison operations such as $<$, $>$, $\leq$, $\geq$, and $=$. However, it is often convenient to combine these expressions to form more complex expressions. For example, we might want to ask whether or not a number is between 1 and 10. To ask this question, we really have to ask two questions:

- Is the number greater than or equal to 1?
- Is the number less than or equal to 10?

If the answers to both of these questions are "yes," then we know that the number is between 1 and 10. If either answer was "no," then we know it is not.

Scheme provides the three fundamental *logic operations* for the purpose of combining boolean expressions. These are *and*, *or*, and *not*. These work as follows:

- *and*: takes any number of arguments and returns `#f` if *any* of the argmuents evaluate to be `#f`, otherwise returns `#t`

[6] To those who have not read Cooper: Killdeer is Hawk-eye's rifle.

- *or*: takes any number of arguments and returns `#t` if *any* of the argmuents evaluate to be `#t`, otherwise returns `#f`
- *not*: takes one argument and returns `#t` if that argument evaluates to be `#f`, otherwise returns `#f`.

Figure 34 shows some examples which use these logical operators. Each expression in the figure could itself, in its entirety, be used as a boolean expression in a *cond* statement. We will see plenty of examples of this soon enough.

6.2 Defining functions

```
STk>  (define (make-big-and-red vert)
          (set! (color vert) "red")
          (set! (size vert) 20))
#[undefined]

STk>  (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk>  (make-big-and-red (car (vertices gv)))
#[undefined]

STk>  (make-big-and-red (car (cdr (vertices gv))))
#[undefined]
```

Figure 35: Defining a function

Typing sequences of commands into the Scheme interpreter can, as you may have found, be tiresome! It would be altogether unacceptable to have to type the *same* sequence of commands over and over again. When we have a situation in which this might happen, it is time to *define* a function. You already know how to *use* a function; you have been doing this all along (e.g. car, cdr, and, or, vertex-ref, etc.). Scheme also gives us the ability to define our own functions to accomplish repetitive tasks.

The example in Figure 35 shows how to define a function. Just as when we define variables, we use the *define* Scheme command. However, instead of giving a variable name as the first argument, we give a list. The latter contains as its first element the name to give the new function, and as subsequent arguments, the names of this new function's arguments. In the figure, we have defined a function called *make-big-and-red* which takes one argument, which we assume will be a vertex-item. The goal of the function will be to make the vertex-item big and red (no surprise there!), and it accomplishes this goal by executing two scheme commands in sequence. Once it is defined, we can call our new function repeatedly to make as many vertices as we want big and red.

In the simple function we just defined, the user will not expect "an answer" since the function is supposed to do something (make a vertex big and red) rather than answer some question. However, our function actually *is* returning an answer. *All Scheme functions (including the ones we define) return the value*

```
STk> (define (big-and-red? vert)
       (and (equal? (size vert) 20)
            (equal? (color vert) "red")))
#[undefined]

STk> (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk> (make-big-and-red (car (vertices gv)))
#[undefined]

STk> (big-and-red? (car (vertices gv)))
#t

STk> (big-and-red? (car (cdr (vertices gv))))
#f
```

Figure 36: Defining a function that answers a question

returned by the last command they execute. So the `#[undefined]` values we get back from *make-big-and-red* are actually the values returned by the last command: `(set! (size vert) 20)`. Recall that the *set!* command returns `#[undefined]`.

The are many times, however, when we need to define functions which answer questions. A straightforward example would be to define a function that determines whether or not a given vertex is in fact big and red. Such an example is shown in Figure 36.

6.3 The *map* function

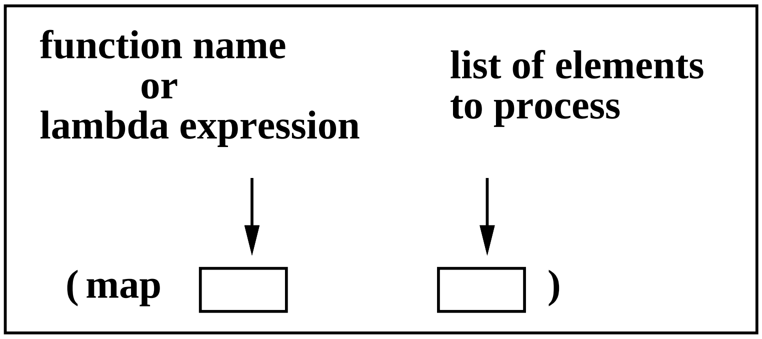

Figure 37: The format of a *map* function call

Knowing how to make decisions and define functions, we are ready to automate some otherwise tedious processes. For example, what if we had a graph with 20 vertices up on the screen, and we wanted to make *all* of the vertices big and red? With grim determination, we could extract each and every vertex from the list of vertices and call *make-big-and-red* for each one. There is a much better way, though! We can use Scheme's *map* function to do this all automatically. Figure 38 shows how. The *map* function, as we'll use it, takes two arguments: the name of a function, and a list. When *map* is executed, it extracts the elements of the list, one by one, and calls the function with the extracted element as an argument. For each element, the function computes a result, and map collects these results into a list and returns it. The standard way to refer to this process is to call it "mapping a function over a list."

```
STk> (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk> (map big-and-red? (vertices gv))
(#f #f #f)

STk> (map make-big-and-red (vertices gv))
(#[undefined] #[undefined] #[undefined])

STk> (map big-and-red? (vertices gv))
(#t #t #t)
```

Figure 38: An example use of *map*

In Figure 38, we map the function *big-and-red?* over the vertex list of a graph-view and obtain a list of `#f` answers, indicating that none of the vertices are big and red. Not to be detered, we map the function *make-big-and-red* over this same list, and make them all big and red. This call to map returns a list of `#[undefined]` values (why?). One more mapping of *big-and-red?* over the same list provides a list of evidence that they have all been changed.

6.4 The *lambda* function

The word "lambda" may excite some dread since it names a greek letter that, to many people, is quite unfamiliar. However, in Scheme, it has a straightforward explanation. Suppose that we want to do a sequence of operations to a each vertex in a long list of vertices. We know how to do this now: (1) define a function to do the sequence of operations, then (2) use *map* to apply this new function to each element of the list. In Scheme, *lambda* is simply a mechanism for skipping step (1).

Sometimes we know that the "function" or sequence of statements we would like to map over a list will never be used except in the very map statement we are about to write. Rather than defining the function, then using it (only this once!), we'll define something called a *lambda function*. This is a function that will only exist within the map statement, and will go away when it is finished. When we define a lambda function, the function name is *lambda*, and the arguments are a list containing a single element, and a sequence of zero or more Scheme expressions. Think of lambda as the "name" of the function, the list with one element as

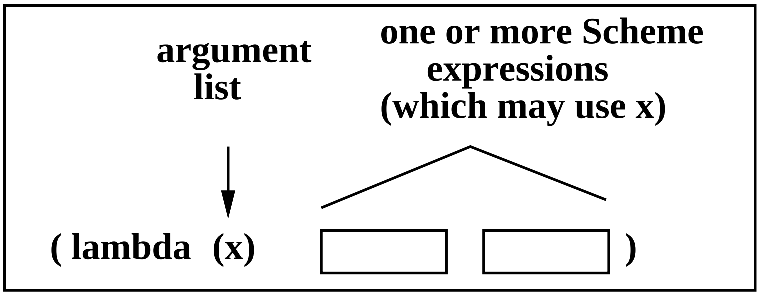

Figure 39: The format of a *lambda* function use

```
STk>   (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3)))))
#[undefined]

STk>   (map (lambda (x) (set! (color x) "red"))
     (vertices gv))
(#[undefined] #[undefined] #[undefined])
```

Figure 40: An example use of *lambda*

its argument, and the subsequent expressions as its body. Each time the *map* function extracts an element from its list, it calls our lambda function, passing the extracted element as an argument. As an example, consider the code in Figure 40, which colors all of the vertices in a graph-view red.

Exercises: Using the graphical interface, create an undirected graph with 10 vertices and 12 edges. Now write single Scheme to accomplish the following tasks:

1. Change the sizes of all vertices to 20.
2. Obtain a list containing the degrees of the vertices.
3. Set the *mark* attribute of all vertices to 0.
4. Change the color all vertices with degree at least 3 to be green.
5. Select all of the edges and choose "edge weights," then "random edge weights" from the attributes menu. Write a Scheme function which will take a graph G and a real number r as input, then alter all edges of G with weight greater than r by modifying their *width* attributes. Test your function, choosing r such that about half of the edges are affected.
6. Write a Scheme function which accepts a graph G as input, and labels each vertex "v" and each edge "e."

6.5 Iteration: the *do* function

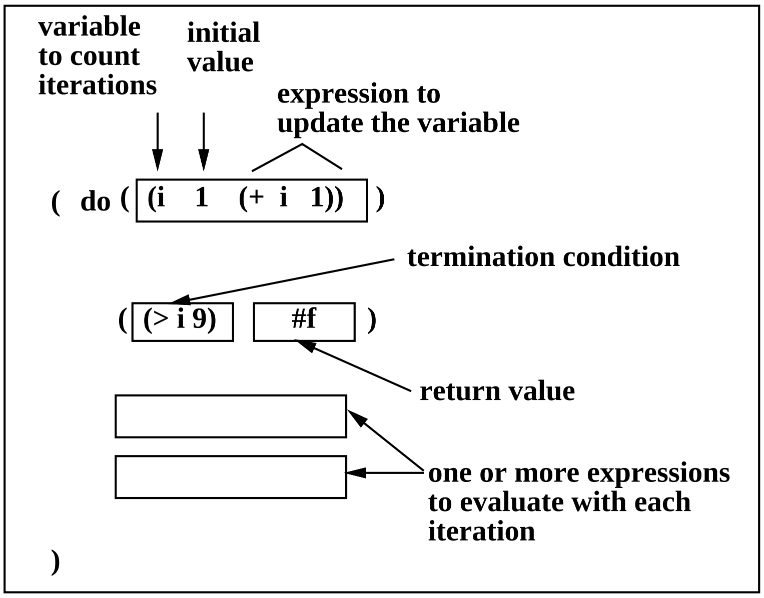

Figure 41: The format of a *do* function call

The *map* function is a good tool for doing something to each element in a list. However, it is not so good for processing a list of elements and computing a single result. For example, we might like to compute the number of vertices which have degree greater than 3, or maybe even build a list containing only these vertices. In this situation, we need to be able to process list elements one-by-one, rather than en masse. The natural solution to do exactly this is called *iteration*, and the construct I will describe is called a *do* statement. There is an alternative to iteration called *recursion*, but this is more complicated and will be introduced later.

```
STk>   (define sum 0)
#[undefined]

STk>   (do  ( (i 1 (+ i 1)) )
            ( (> i 10) sum  )
          (set! sum (+ sum i)))
55
```

Figure 42: Using iteration to sum the integers from 1 to 10

Figure 42 shows an example *do* function call. The variable i serves as a "counter" variable, which will control the number of iterations. In the example, i is initialized to 1, and is incremented by 1 after each iteration. The loop is terminated when the value of i becomes larger than 10, and the value of the variable *sum* is returned. Each time "through the loop," the statement(s) after the list containing the termination condition and return value are evaluated.

6.6 Local variables: the *let** function

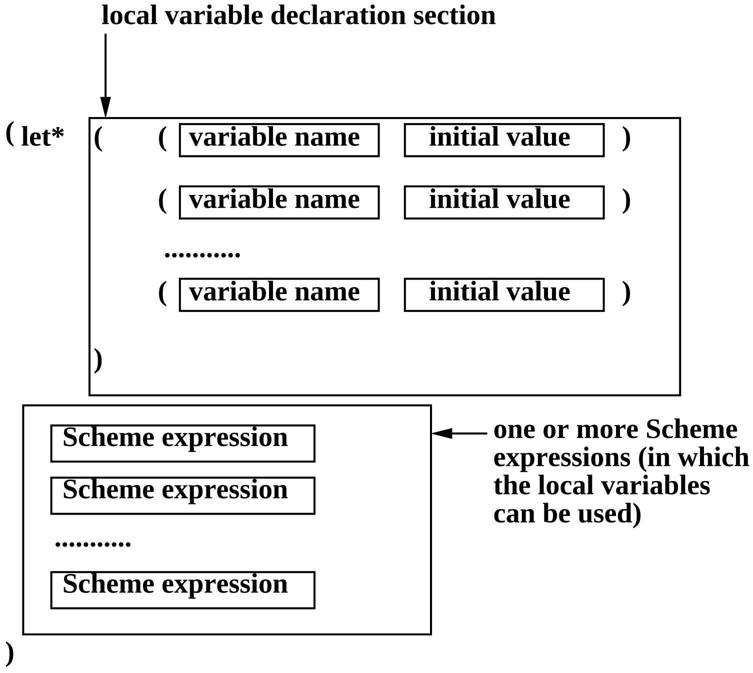

Figure 43: The format of a *let** function call

Functional programming, as an idea, is very clean and attractive. No variables are used, and everything fits together logically. The problem is that if no variables are used, then many things must be computed over and over again. It is often simply more efficient to define a variable to store a result, then use it repeatedly. We have been using this strategy in the preceding sections.

Given that variables can lead to faster computations, we might be tempted to use them at will and generate Scheme programs with dizzying numbers of variables. If we weren't careful, we might try by mistake to use the same variable for two different purposes. This can lead to very surprising results indeed. For example, if we defined a variable called s and put the value 3 in it, then called a function which redefined s to be the character string "Monty Python," the Scheme interpreter would get quite upset if we then tried to add 1 to s!

To control the proliferation of variables, Scheme includes a function called *let** (yes, the "*" is part of the function's name). This function gives us a little environment in which we can create variables meant to exist only in that environment, and which will go away when the *let** function is finished. Figure 43

```
STk>  (define a 1)
#[undefined]

STk>  (define b 2)
#[undefined]

STk>  (let*  ( (a 4)
             (b (+ a 1)))
        (describe a)
        (describe b))
4 is an integer.
5 is an integer.
#[undefined]

STk>  (describe a)
1 is an integer.
#[undefined]

STk>  (describe b)
2 is an integer.
#[undefined]
```

Figure 44: Defining local variables using let*

depicts the components of a call to *let**. The first argument is a list of variable names, along with associated initial values, while subsequent arguments are Scheme expressions to be evaluated. *When the let* function has evaluated its last expression, it returns the value returned by that expression, and all of the variables created within the let* disappear. Furthermore, the names of the variables created within the let* call may be the same as variables outside the call, and the latter are not changed.* For example, consider the variable a in Figure 44. The name "a" refers to two *different* variables: the one defined before the let*, and the one created within the let*. A variable defined using the *define* function is called a *global* variable since it can be used anywhere once it has been defined. On the other hand, variables created within a *let** function call are called *local* variables, since they only "live" within the let*.

7 Implementing discrete mathematical ideas in *LINK*

With some fundamentals of graph theory, the use of *LINK*'s graphical interface, and the use of Scheme to program new features, we are ready to try some larger examples.

7.1 Case study: equivalence relations

In Section 3.2.2, we saw that directed graphs (digraphs) model ordered relationships between objects. The example given there was a model of a tennis tournament, in which an edge from one vertex to another represented the fact that the player represented by the first vertex had beaten the player represented by the other vertex. If we allow an edge to point from a vertex back to itself (this is called a *self-loop*) and do not allow multiple edges, then a digraph is a picture of an abstract mathematical idea called a *relation*. A formal definition of a *relation* would require some notation and concepts which won't be covered here, so just think of a relation as a digraph.

Relations are useful in many different ways, and mathematicians are interested in many properties of relations, three of which are defined informally below. Any relation which satisfies all three of these properties is called an *equivalence relation*

- A relation (digraph) is *reflexive* if each vertex has a self-loop, i.e., every vertex points to itself.
- A relation (digraph) is *symmetric* if whenever a vertex u has an edge pointing to vertex v, the vertex v must have an edge pointing back to u.
- A relation (digraph) is *transitive* if whenever a vertex u has an edge pointing to vertex v and vertex v has an edge pointing to vertex w, the vertex u must also have an edge pointing to vertex w.

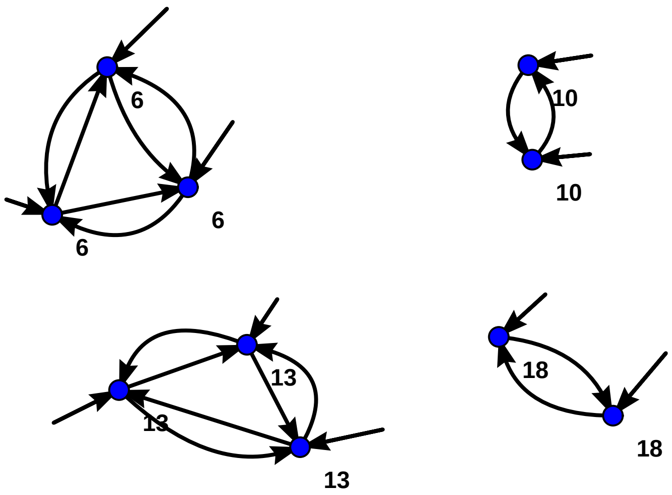

Figure 45: An equivalence relation.

7.2 An example

Think about modeling the following problem: there is a swim team for children from ages 6-18, and we want to group them into groups by age (6 year olds, 7 year olds, etc.). To model this using a digraph, we would model swimmers with vertices, and we would put a directed edge from swimmer 1 to swimmer 2 if, and only if, swimmer 1 is the same age as swimmer 2. Suppose that our team has three swimmers of age 6, two of age 10, three of age 13, and two of age 18. Figure 45 shows our model.

In this figure, each vertex is labeled not with the name of the swimmer it represents, but his or her age. The strange-looking arrows which point at each vertex seemingly from thin air are self-loops. They actually emerge from the same vertex to which they point, but limitations in *LINK*'s current drawing ability show them in their current form rather than a circular loop. Is the relation depicted in this digraph reflexive? Put a different way, does each swimmer have the same age as himself or herself? Yes! Is the relation symmetric? If I have the same age as you, do not you have the same age as I? In the digraph, testing for symmetry reduces to ensuring that whenever there is an edge going one direction, there is an edge going back in the original direction. A look at the figure should convince you that our example relation *is* symmetric, though you should think about the meaning of the self-loops in this context.

Is is a little more difficult to determine whether or not a digraph representing a relation is transitive. Here is the idea: whenever you can start at a vertex, say vertex 1, and follow two edges, arriving at another vertex, say vertex 2, then you must ensure that there is an edge from vertex 1 to vertex 2. Take a minute or two to try this on the example digraph we have been using. You won't find any violations, because it is, in fact, transitive. Since our relation is reflexive, symmetric, and transitive, it is an equivalence relation.

Exercises:

1. Draw a digraph representing a relation which is symmetric, but not reflexive.
2. Draw a digraph representing a relation which is reflexive, but not symmetric.
3. Draw a digraph representing a relation which is symmetric, but not transitive.
4. Draw a digraph representing a relation which is symmetric and transitive, but not reflexive.
5. Draw a digraph representing a relation which is reflexive and transitive, but not symmetric.

7.3 Creating animations using *LINK*

Suppose we want to create animations of the processes which check whether or not a digraph representing a relation is reflexive, symmetric, and transitive. A reasonable way to do this would be as follows.

- For the reflexivity test, highlight each vertex in turn by changing its color, then if it has a self-loop, jiggle that edge, otherwise color the vertex red.
- For the symmetry test, highlight each edge in turn by changing its color, then jiggle the edge going in the other direction if it is there. Otherwise, thicken the edge to show it as a violation.
- For the transitivity test, let the user click on a vertex, then change the color of each vertex to which it points to be green. After that, change the size of each vertex reachable from the original by following exactly two edges to be 20. Flash and jiggle any vertex that got resized, but not recolored – it shows a violation of the transitivity condition.

```
STk>   (define gv (show-graph (digraph '(1 2 3) '((1 1) (3 3)))))
#[undefined]

STk>   (define (reflexive gv)
     (not (member #f
             (map (lambda (x)
                    (set! (color x) "green")
                    (let ((e (is-edge (vertex x)
                                      (vertex x))))
                         (cond (e  (jiggle (edge-item e)) #t)
                               (#t (set! (color x) "red") #f))))
                 (vertices gv)))))
#[undefined]

STk>   (reflexive gv)
#f
```

Figure 46: A function to animate the reflexivity test

7.3.1 The animation to test reflexivity

For each vertex, we need to jiggle its self-loop, if it exists, so the first two questions are: how are we going to get each vertex, and once we have it, how do we find its self-loop and jiggle it? The two ways we know how to process a list of vertices systematically are to use the *map* function, or to iterate using the *do* function. For this example, let's use *map*. We only need to process each vertex once, and we don't need to keep track of any running result, like the highest degree vertex we've processed so far. With that issue settled, we need to know, given a vertex, how to find out if it has a self loop. There is a *LINK* function called *(is-edge v1 v2)* which, given two vertices v_1 and v_2 (note: vertices, not vertex-items. See Section 5.3), returns the edge from v_1 to v_2 if it exists.

We want to define a function that will do this animation, then return true (`#t`) if the relation represented by the digraph is reflexive, and false (`#f`) otherwise. Our function will take a *graph-view* as its argument, and extract its vertex-items using the *(vertices graph-view)* function. Next, it will *map* a *lambda* function over this list of vertex-items that will color the vertex green temporarily, determine whether there is a self-loop on that vertex, then, if so, jiggle it using the *(jiggle edge-item)* function. If there is no self-loop, the *lambda* function will return `#f` and color the vertex red, otherwise it will change the color of the vertex back to blue and return `#t`. Since the function is mapped over the list of all vertices, the result of the map command will be a list of `#t` and `#f` values. The *member* function (see Section 4.4) is then used to see if there are and `#f` answers, i.e., if the relation is not reflexive. The entire code for the reflexivity testing function is shown in Figure 46.

7.4 Adding menu options to *LINK*

The preceding sections have explained how to combine Scheme statements to develop new animations and procedures. Once these are working, it is straightforward to add new options to the graph menu so that users can see your work. Remember that the *LINK* interface itself is written in Scheme code, code that is readily available. The code which implements the menus of *graph-view* windows is found in a file called *graph-menu.stklos*. An outline for the process of adding a new menu option is shown below, and details will follow.

- Copy the file *graph-menu.stklos* from its directory (to be specified) into a directory of your own.
- Edit your copy of the file so that the menu is extended
- Instruct *LINK* to use your copy of the file instead of the original.

7.4.1 Copying the file

LINK's interface files are all located in a central directory, which you should be able to see. On a Windows machine, the standard place for this directory is

```
C:\link\1.3\stklos,
```

while in Unix it is

```
/usr/local/lib/LINK/1.3/stklos.
```

If the files are *not* in the standard place, check with your system administrator to see where *LINK* was installed.

Suppose that you are working on a Windows machine, and that you wish to put your *LINK*-related files in the directory `E:\my-files\link-code`. Once you have found the file *graph-menu.stklos*, copy it into your directory using either Windows Explorer or the following MS-DOS commands:

```
cd E:\my-files
mkdir E:\my-files\link-code
cd link-code
copy C:\link\1.3\stklos\graph-menu.stklos *.*
```

7.4.2 Creating the menu option

There are many ways in which to create a menu option which passes information to a function when calling it, but for this document, we will consider only the simplest menu options – those that call a function that takes at most argument: a graph-view. Your first task, therefore, will be to write such a function. For example, consider once more the function to test for reflexivity in a relation (Figure 46). This function *(reflexive gv)* takes one argument, a graph-view, so it is ready to be added as a new menu option.

Edit your copy of *graph-menu.stklos* using your favorite text editor and search for the text *"Algorithm"* (include the double quotes in your search string). You should be looking at the following text:

```
("Algorithm"
    ("Custom"
        (
```

```
        )
    )
    ("Searches"
    ...
```

You should recognize this text as that which appears in the `Algorithm` menu in any graph window. In order to add your new menu option, edit the text until it looks like this:

```
("Algorithm"
    ("Custom"
        (
             ("Reflexivity test" ,(lambda() (reflexive gv)))
        )
    )
    ("Searches"
    ...
```

You have just added a sublist describing the text of the menu option and what to do if it is selected. The only thing that should be mysterious about this addition is the comma, and I will leave it as a mystery for now. Simply put a comma in front of the lambda expression associated with each graph menu option.

7.4.3 Instructing *LINK* to use the new menu

When *LINK* is first run, it reads its interface from the standard directory. In this section, however, we've overridden the standard interface with a customized one. In order for our users to see the new option, our copy of the file *graph-menu.stklos* must be loaded into the interface. This is done using the `Load STk(los) File` option from the main window. *Note that any existing graph windows will not have the new menu option. Only graph windows created after the new file has been loaded will offer the new option.* It is possible to arrange things so that *LINK* automatically finds your file instead of the standard one when it first starts up, but you will need to understand environment variables, which are system-dependent. See Section 10.

8 Using sets, multisets, and sequences in *LINK*

The fundamental structure for storing data in Scheme is, of course, the list. There are collections of data, however, which are best stored in other ways. For example, a simple graph (see Section 3.4.1) cannot have more than one edge connecting the same vertices, while a multigraph can. How would we enforce this condition if a graph's edges were stored in a list? When adding an edge, we'd have to ensure that it wasn't already in the list before it was inserted.

Rather than forcing the user (or even the interface builder) to provide logic such as this, *LINK* provides several different types of collection as alternatives to using a list. These are listed below.

- *mset* (short for multiset) holds an unordered collection of elements, some of which may occur multiple times.
- *set* holds an unordered collection of unique elements
- *sequence* holds an ordered collection of elements some of which may occur multiple times.

```
STk>   (define g (digraph '(1 2 3) '((1 2) (2 3))))
#[undefined]

STk>   (define g-edges (edges g))
#[undefined]

STk>   (describe g-edges)
#[<mset<edge*>> 402c8aa8] is an instance of class <mset<edge*>>
Slots are:
     val = {<1 2> <2 3>}
#f

STk>   (describe (ref g-edges 0))
#[<edge*> 402bd1a4] is an instance of class <edge*>
Slots are:
     val = <1 2>
#f

STk>   (describe (vertices (ref g-edges 0)))
#[<sequence<vertex*>> 402bbe78] is an instance of class <sequence<vertex*>>
Slots are:
     val = <1 2>
#f
```

Figure 47: Using collection objects

Furthermore, these three "collection objects" are *templated*, which means that the type of element stored is part of the collection object's name. For example, a set of edge objects is refered to in STk as *set-edge*, while a sequence of vertex object is called a *sequence-vertex*. See Figure 47 for an example which shows the collection objects associated with a directed graphs. The *ref* command extracts elements of a collection just as edge-ref and vertex-ref extract edges and vertices from a graph.

There are many functions available to manipulate collections, including the set primitive operations (union, intersection, and difference) and basic combinatorial functions such as combinations, permutations, and powwer set operations. However, as this document is still in the early stages of development, most will not be described here. The *ref* function mentioned above is crucial since it can be used to extract individual elements, and we can convert collections into Scheme lists and vice versa. Figure 48 demonstrates this for an *mset*, a *set*, and a *sequence*.

STk> **(define g (digraph '(1 2 3) '((1 2) (2 3))))**
#[undefined]

STk> **(define g-edges (edges g))**
#[undefined]

STk> **(define g-edgelist (mset-edge– >list g-edges))**
#[undefined]

STk> **(map (lambda (e) (find-attribute 'width e)) g-edgelist)**
(1 1)

STk> **(define eset-copy (list->mset-edge g-edgelist))**
#[undefined]

Figure 48: Converting collection objects to and from lists

9 Binding mouse and keyboard events to *LINK* functions

Each graph-view window consist of several "slots," as you have seen whenever using the *describe* function. One of these slots, thus far unmentioned, is the "graph-toplevel." This is a window which encompasses both the familiar graph menu and the white area on which you've been manipulating graphs. The graph-toplevel window is retrieved from a graph-view *gv* using the *slot-ref* function as follows:

```
(define gt (slot-ref gv 'graph-toplevel))
```

Once the graph-toplevel has been retrieved in this way, it can be "bound" to react to mouse clicks and keystrokes. This is done with the *bind* function, and two examples are shown below. The first binds the "a" key to a function that prints a message, and the second binds the 1st button to display the current graph whenever *Alt-1* is clicked. Try them out!

```
(define gv (current-graph-view))
(define gt (slot-ref gv 'graph-toplevel))
(bind gt "<KeyPress-a>" (lambda () (display "hi")(newline)))
(bind gt "<Alt-1>" (lambda () (describe (graph gv))))
```

Other simple bindings are similar.

10 Setting up and running *LINK*

Before you can run *LINK*, you need to set several *environment variables.* These tell the program where to look for certain important files.

10.1 Windows 95 and NT

The necessary environment variable definitions are shown in Figure 49, which assumes that your *LINK* executable file is stored in the directory `C:\link\1.3`. Note that the `STK_LOAD_PATH` must be specified using forward slashes. These lines in Figure 49 should be added to your `C:\autoexec.bat` file so that they will always be set. Once you make the necessary changes, reboot your machine and you will be all set. When

```
set LINK_BASE=C:\link\1.3
set STK_LIBRARY=C:\link\1.3\stk
set STK_LOAD_PATH=C:\link\1.3\stklos
set PATH=C:\link\1.3;<...the rest of your path...>
```

Figure 49: Setting environment variables in Windows.

you are ready, execute `C:\link\1.3\Link.exe` to run the program. The application will take some time to load since it has to initialize many things. Before you run the program next, you might want to add it to your `Start` menu or create a desktop shortcut to it.

10.2 Unix

Unix supports several different command interpreters (shells), some of which have different syntaxes for defining environment variables. I'll assume that any Unix user will know how to set these variables. Follow the running directions in the distribution you have downloaded, or ask your system administrator (or whoever installed *LINK*) how to set them. Once everything is set up correctly, type `Link`.

Index