Labs and Applets for

The Most Complex Machine

March 2000
(Software updated June 2004)

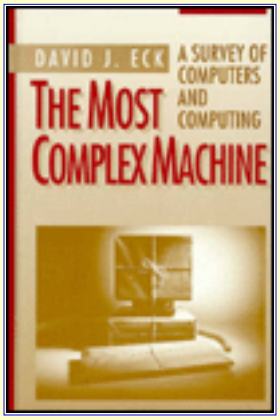
This PDF file contains material from the web site at

http://math.hws.edu/TMCM/java/.

The PDF version can be read and printed using Adobe Acrobat Reader. It is provided mainly to allow easy printing of material from the web site.

Most of the original Web pages that are reproduced here contained Java applets. In this PDF version, the places where the applets should appear are marked with a note stating that Java is not available.

David Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY 14456 USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/



Labs and Applets for The Most Complex Machine

(David Eck, May 1998, March 2000, and June 2004)

ON THIS PAGE YOU'LL FIND a set of lab worksheets and Java applets that are meant to help people learn about computer science. They were written for use with my introductory computer science textbook, but they can also be used independently of that text. The labs and applets are free for personal use. In addition, the applets can be freely used for non-commercial purposes, including courses that do not use my textbook. I ask that teachers use the labs as an official part of a course only if they adopt my textbook for that course (but I will consider giving permission for other uses). Again, the applets -- including the informational page about each applet at the bottom of this page -- are free for any non-commercial use, including use in any course.

Note about software update in June 2004: The Java applets were written using the original version of Java. While they have generally continued to work with more recent Java releases, a few incompatibilities have crept in. I June 2004, I fixed the known problems, so that the applets should now work with all versions of Java (at least for the time being). I have also taken the opportunity to repackage the applets in "executable jar files" that can be run as stand-alone applications, outside of a web browser. See the download page for more information.

The text for which I wrote the applets and labs is The Most Complex Machine: A Survey of Computers and Computing. It is an introductory survey of a big chunk of computer science. You can learn more about it at its home page, http://math.hws.edu/TMCM.html. For more information about me (David Eck) and my other projects, see my home page at http://math.hws.edu/eck/index.html. You can send me email at eck@hws.edu.

Later on this page, after the labs, you'll find links to more general information about each of the seven applets that are used in the labs. For some of the applets, a set of tutorial examples is included. You don't need to read this material to do the labs, since the lab worksheets include instructions for using the applets.

The labs and applets are available for **downloading**. So is **source code** for the applets. The applets are also available as Java applications, which can be run without a Web browser. For more details, see the "downloading and information page." You'll also find information there about how to use the applets on your own Web pages.

The Labs

Introductory Lab: The Web, Java, and DataReps.

This lab is mainly an introduction to the use of the World Wide Web and to the idea of Java applets. A simple applet, DataReps, serves as an example of an applet. It also serves to demonstrate how several different types of data are represented in a computer.

xLogicCircuits Lab 1: Logic Circuits.

Explores logic circuits created out of AND, OR and NOT gates. The relationship between

circuits and Boolean algebra is also covered.

xLogicCircuits Lab 2: Memory Circuits.

Shows how circuits that contain feedback loops can be used as memory circuits, and how a RAM (random access memory) can be constructed and used.

xComputer Lab 1: Introduction to xComputer.

Introduces the xComputer, a simple model computer, and investigates how it operates in a fetch-and-execute cycle to carry out machine language instructions stored in its memory.

xComputer Lab 2: Assembly Language Programming.

Covers assembly language programming for the xComputer, including labels and indirect addressing.

xComputer Lab 3: Subroutines.

Introduces the idea of a subroutine and shows how subroutines can be implemented "by hand" in the assembly language of xComputer, even though that language does not offer direct support for subroutines.

xTuringMachine Lab: Introduction to Turing Machines.

This lab is meant to illustrate the basic operation of Turing machines and to show that even the extremely simple operations performed by Turing machines are sufficient for performing complex computations.

Publishing on the Web.

This lab will cover some of the basics of Web publishing, concentrating on the "Composer" utility in Netscape Communicator. This lab is not closely related to The Most Complex Machine, and it does not use any applets. However, it does sort of fit in with the theme of "real computers" and their impact on society, which is covered in Chapter 5 of the text. (This lab is somewhat specific to Hobart and William Smith Colleges.)

xTurtle Lab 1: Introduction to Programming.

Covers the basics of the xTurtle programming language, including loops, if statements, variables, and built-in turtle graphics commands.

xTurtle Lab 2: Thinking about Programs.

Investigates how preconditions and postconditions can be used to help develop working programs that perform complex tasks. Also introduces the idea of subroutines.

xTurtle Lab 3: Subroutines and Recursion.

Continues with subroutines in general and recursive subroutines in particular. Recursion is used to produce nifty pictures.

xSortLab Lab: Sorting and the Analysis of Algorithms.

Uses the xSortLab applet to investigate several different algorithms for sorting lists of numbers.

xTurtle Lab 4: Multiprocessing.

Shows how multiprocessing can be used to divide a large problem into several subtasks that can be executed in parallel. Some examples of communication between parallel processes are also given.

xModels Lab 1: Two-D Graphics and Animation.

Introduces a scene-description language for creating still images and multi-frame animations. Shows how hierarchical, geometric models are used in computer graphics. In this lab, only two-dimensional images are covered.

xModels Lab 2: Adding the Third Dimension.

Extends the ideas covered in the previous lab to three dimensions. Also covers "lathing" and "extrusion," two operations for producing three-dimensional objects.

The Applets

DataReps

A small applet that shows how the same 32 bits stored in the memory of a computer can represent different things, depending on how they are interpreted. It is related to material covered in Chapter 1, Section 1 of The Most Complex Machine.

xLogicCircuits

Lets you create simulated logic circuits, like those discussed in Chapter 2, by dragging AND gates, OR gates, and other components onto a circuit board and drawing connections between them. You can turn the inputs of your circuits on and off, to see how the circuits behave.

xComputer

An implementation of the model computer developed in Chapter 3. You can write assembly language programs for that computer and watch as the computer executes them step-by-step.

xTuringMachine

Lets you create Turing machines and watch as they move back and forth along a "tape," reading and modifying its contents. Turing machines are covered in Chapter 4.

xTurtle

Lets you write and execute programs written in the xTurtle programming language, which is used as an example in Chapters 6, 7, and 10.

xSortLab

Lets you watch several sorting algorithms in action and measure their performance. This applet is related to material on the analysis of algorithms that is covered in Chapter 9.

xModels

Does geometric modeling and computer animation, as discussed in Chapter 11. You can write "scene descriptions" and then "render" the resulting images or animations as wireframe models.

David Eck (*eck@hws.edu*)

Labs last updated 28 May 1998
Page design updated 23 March 2000
Software updated June 2004

TMCM Labs and Applets
Downloading and Information Page

This page contains links for downloading a set of labs and applets that were written by David Eck for use with his introductory computer science textbook, The Most Complex Machine. The labs and applets are also available on line at http://math.hws.edu/TMCM/java/. Information about using this material is in the `README` files for the downloads, which are also available below for reading on line.

In addition to the archives, a "pdf" version of the labs and applet information is available, at the bottom of this page.

Note that many of the downloads are available in two formats: ZIP archives (with file names ending in .zip) and TAR.GZ archives (with file names ending in .tar.gz). The contents are identical, except that text files in ZIP archives are in Windows/DOS format while text files in TAR.GZ archives are in Linux/UNIX/MacOSX format. For most purposes, it doesn't matter which archive you use, as long as you can unpack it.

ZIP archives can be used directly in Windows XP. In any version of Windows, you can unpack a ZIP archive using WinZip (available from www.winzip.com) or Aladdin Expander (available from www.aladdinsys.com). You Web browser might already be configured to unpack the archive when you download it.

In MacOS X, your Web browser should unpack any archive that you download. If not, you can use Stuffit Expander (available from www.aladdinsys.com). If you are still using MacOS 9.0 or earlier, see the bottom of this page.

On Linux/UNIX, you should be able to unpack a TAR.GZ archive named *archive.tar.gz* with the command **`tar zxf archive.tar.gz`** or with the two commands `gunzip archive.tar.gz` followed by `tar xf archive.tar.gz`. This requires that you have gzip software installed.

Download the Entire Web Site

Use one of the following links to download a complete archive of the TMCM Labs and Applets Web site. You are welcome to post an unmodified copy of this material on your own Web site. You can also use it on your own computer. However, when you use the applets on your own computer, the applets on the web pages in this archive will probably **not** be able to read the example files that they are supposed to read. This is a security feature of applets, but it can be annoying at times. To deal with this problem, you might also want to download the "Lab and Tutorial Examples" archives below.

http://math.hws.edu/TMCM/java/download/tmcm-java-web-site.zip (for Windows)
http://math.hws.edu/TMCM/java/download/tmcm-java-web-site.tar.gz (for Linux/UNIX)

You can take a look at the README file for this archive. The README file explains in detail how to use the applets on your own web pages and how to run the applets as stand-alone applications.

Applet Jar Files (Software)

The seven applets are packaged as "jar files." These files can be run as standalone applications, and they can be used for putting the applets on your own web pages. The jar files are part of the complete web archive that you can download in the previous section of this page. See the README file from that archive for complete information about how to use them. You can also download individual jar files using the following links:

DataReps.jar xLogicCircuits.jar xComputer.jar

xTuringMachine.jar xTurtle.jar xSortLab.jar

xModels.jar

On some computers, you can run one of these files simply by double-clicking on it. If you have a recent version of Java from Sun Microsystems, you can try commands of the following form on the command line:

```
java  -jar  xLogicCircuits.jar
```

If you are using Microsoft's version of Java in Internet Explorer in Windows, then you can use the "jview" command in a command window to run the programs. The command takes a form such as:

```
jview  -cp  xLogicCircuits.jar  tmcm.xLogicCircuitsFrame
```

Here, "tmcm.xLogicCircuitsFrame" is the name of the Java class within the jar file that actually defines the programs. Similar names are used in the other jar files.

Lab and Tutorial Examples

Here are two archives that make it easy to run the Labs examples and Information/Tutorial examples outside of a Web browser. One advantage of this is that you will be able to load and save files.

The Labs examples and applets (from the "The Labs" section of the main page):

http://math.hws.edu/TMCM/java/download/TMCM_Labs.zip (for Windows)
http://math.hws.edu/TMCM/java/download/TMCM_Labs.tar.gz (for Linux/UNIX)

You can view the README file from the Labs archive for more information.

The Information/Tutorial examples and applets (from the "The Applets" section of the main page):

http://math.hws.edu/TMCM/java/download/TMCM_Applet_Tutorials.zip (for Windows)
http://math.hws.edu/TMCM/java/download/TMCM_Applet_Tutorials.tar.gz (for Linux/UNIX)

You can view the README file from this archive for more information.

PDF File for Printing

The following PDF file contains copies of all the lab worksheets and applet information pages from http://math.hws.edu/TMCM/java/. (Except that where an applet should appear on a page, you'll just see a note that Java is not available.) This file is provided primarily to make it easy to produce print outs. You can read it using Adobe Acrobat Reader

http://math.hws.edu/TMCM/java/download/tmcm-java-web-site.pdf

If you click on the above link, your browser might use a PDF plugin to let you see the contents of the file. If it does not have the plugin, it should let you download the file. If you do have the plugin and still want to download the file, try right-clicking or Control-clicking the link.

Java Source Code

The Java source code for the applets can be browsed on-line and is included in the complete web site archive that can be downloaded at the top of the page. However, for convenience, you can also download the source code separately using one of the following links:

http://math.hws.edu/TMCM/java/download/tmcm_source_code.zip (for Windows)
http://math.hws.edu/TMCM/java/download/tmcm_source_code.tar.gz (for Linux/UNIX)

You can view the README file from this archive. Note that this code was written for version 1.0 of Java and uses many features that should not appear in modern Java code. It was not written with the intent of publishing it, and it has almost no comments. I have made the source code available because a number of people have requested it.

For Users of MacOS 9 (or Earlier)

If are still running MacOS 9, or earlier, on an old PowerMac computer, you cannot and will never be able to use versions of Java newer than version 1.1. The changes that were made to this web site in June 2004 are irrelevant to you. You might want to use the following Macintosh-format archives of the old version of this site:

The complete Web site from March 2000:

http://math.hws.edu/TMCM/java/download/tmcm-java-web-site.sit.hqx (for Macintosh)

The Labs and Tutorials examples in a single archive from March 2000:

http://math.hws.edu/TMCM/java/download/tmcm-java-apps.sit.hqx (for Macintosh)

The Java source code files from March 2000:

http://math.hws.edu/TMCM/java/download/tmcm-java-source.sit.hqx (for Macintosh)

David Eck (eck@hws.edu), June 2004

Labs for The Most Complex Machine

Introductory Lab: The Web, Java, and DataReps

THIS IS THE FIRST in a set of lab worksheets meant to be used with the introductory computer science textbook, The Most Complex Machine. The lab worksheets are written for use on the World-Wide Web, and they make use of software written in the form of applets. Applets are computer programs written in a new programming language called Java. All this is explained in the lab, which acts as an introduction to the Web and an orientation to the way applets will be used in the rest of the lab.

As part of the lab, you will use an applet called "DataReps" to help you learn about how different types of data can be represented using binary numbers. This material is related to Section 1.1 of the text.

(For a full list of labs and applets, see the index page.)

This lab includes the following sections:

The World-Wide Web

The Internet consists of millions of computers around the world, linked together by a network so that they -- and their users -- can communicate and interact. In the last few years, the Internet has become a common part of everyday life for many people. The Internet provides a number of useful services, including e-mail, USENET news groups, and the World-Wide Web. This section of the lab is a brief introduction to the Web.

The World-Wide Web, also known as the WWW or simply as the Web, consists of "pages" of information stored on computers all around the world. These pages are available to anyone with a connection to the Internet. They viewed with a Web browser such as Netscape or Internet Explorer. A page can contain text, pictures, sounds, three-D graphics, movies, applets, and even interactive features such as fill-in forms. Most important, a page can contain links to other pages. When you click on a link, the Web browser will fetch the page that it refers to and display it to you. So, it's pretty easy to use the Web: just point your

mouse at a link, and click! Here, for example, are some links to pages you might want to visit:

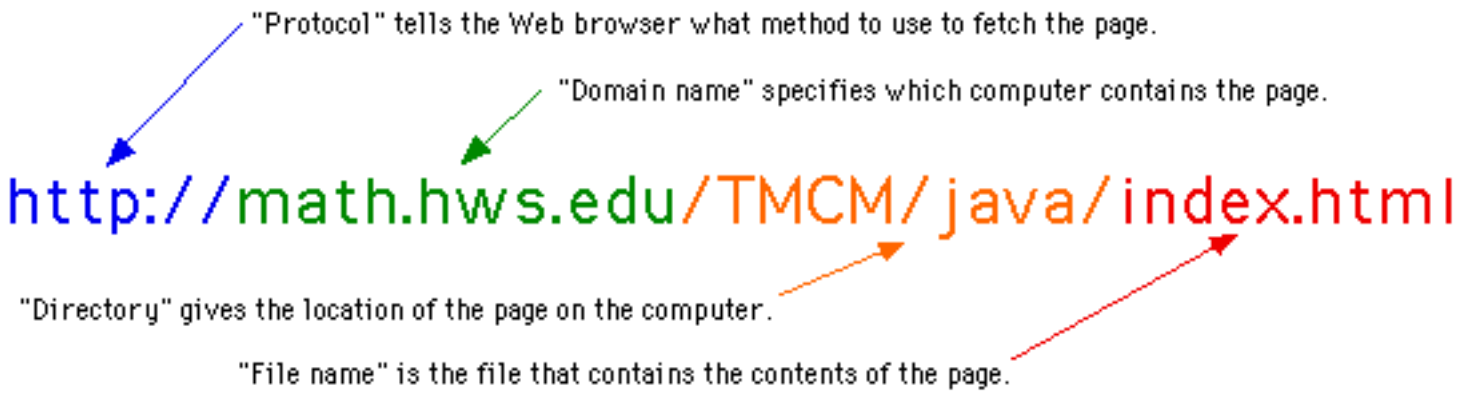
- ABC News, Cable Network News, and the New York Times
- The Whitehouse, the Senate, and the House of Representatives
- The Nation Magazine, for some left-wing political opinion
- Discover, Nature, and Scientific American science magazines
- Sports Illustrated and ESPN
- The Web Museum, featuring great art
- Views of the Solar System
- List of Colleges and Universities
- amazon.com, a place to buy books
- Map Blast, a free map-drawing utility
- Blue Mountain Arts, where you can send free virtual greeting cards
- Dilbert comics

The Web is huge, and it has information on almost anything you can think of. There are millions of computers on the Internet. Each of those computers can run a "Web site" and publish Web pages. No one controls this; no one has to authorize it (at least not yet). In fact, you can publish your own information on the Web. One of the later labs will deal with this.

URL's and All That

Every page of information on the Web is identified by a URL (Uniform Resource Locator). Other resources, such as pictures and sounds, are also identified by URL's. When you are viewing a page with a Web browser, the URL of that page is usually displayed in a box near the top of the browser's display window. If you know the URL for a page, you can go directly to that page by entering the URL in that box (and pressing return).

A typical URL is http://math.hws.edu/TMCM/java/index.html. This is the URL for a page that describes all the labs and applets that I have written for use with The Most Complex Machine. This URL has several meaningful parts:



The HyperText Transfer Protocol (HTTP) is the most common method used for communication on the Web. Another common protocol is File Transfer Protocol (FTP), an older method for transferring files from one computer to another. You might also run across some other protocols in URL's.

A domain name, such as math.hws.edu, identifies a particular computer on the Internet. Most of the computers that are used as "servers" of data on the Web have domain names that begin with "www", such as www.whitehouse.gov. You can often read some information about a computer from its domain name. The computer named math.hws.edu is in the Mathematics Department ("math") at Hobart and William Smith Colleges ("hws"), which is an educational institution ("edu"). The last part of the domain name, such as "gov" or "edu" is called the top-level domain. Top-level domains include:

- COM for commercial purposes
- EDU for educational institutions
- GOV for government computers
- MIL for the military
- ORG for other organizations
- NET for certain Internet services

These domains are usually used by computers in the United States. Computers in other countries generally use two-letter country codes for their top-level domains. For example, a domain name ending in "it" indicates a computer in Italy, and "ca" is used by Canadian computers.

Many companies, organizations, and institutions have "home pages" on the Web. If you know something about domain names, you can often guess the URL used by a given company, organization, or institution. For example, you might guess that the home page of the United States Senate is http://www.senate.gov or that the Coca-Cola corporation has a home page at http://www.cocacola.com. (When you use a URL that omits the directory and file name, you will usually get the home page, or index page, from the specified computer.)

Searching the Web

Because there is so much information on the Web, finding what you want can be a problem. There are several utilities that can help you to find things on the Web. First of all, there are "hierarchical indices" that list Web sites according to category. One of the largest of these indices is Yahoo.

Another way to find things on the Web is to use a "search engine." A search engine consists of an index of millions of Web pages and a program for searching the index. (The index is made by a program that constantly downloads Web pages and adds their contents to the index. **No** index can include all the data on the Web because the Web grows so quickly. Also, some of the data in an index will be out of date because people change or delete their Web pages.)

To use an index, all you have to do is type some words into a box and click on a button. (You can do more advanced searches, but most search engines allow you to do simple searches in this way.) You'll get back a list of Web pages that contain the words you entered. Here, for example, is a simple interface to the Alta Vista search engine. To try it, click in the input box, type some words, and the click the Submit button:

Search the Web

and Display the Results in Standard Form

Submit

There are many search engines, including Alta Vista, WebCrawler, Lycos, Excite, and Infoseek. You will have to use at least one of these search engines in order to do some of the exercises at the end of the lab. To get the most out of a search engine, you should read its help or instructions page.

Java and Applets

The example of the search engines shows that a Web page can be more than just a passive collection of links. A Web page can also be interactive. The Alta Vista search engine uses a type of interaction known as a form (or "fill-in form"). You enter some data in the form and click on a submit button. Your Web browser sends your data to another computer, which responds to your data by sending a new page for your Web browser to display.

Web pages can also provide other types of interactivity, without involving a second computer. One of the new technologies on the Web is Java, a programming language that can be used to write applets, which are small programs that run on a Web page. Many Java applets are more decorative than useful, like this "Moiré" applet:

(Sorry, your browser doesn't do Java!)

(A Moiré pattern is formed by the "interference" between two similar patterns. In this

applet, the basic pattern consists of lines radiating out from a command center. There are two copies of this pattern. One is fixed, while the other drifts about.)

Even this decorative applet allows some interaction. If you click-and-drag your mouse on the Moiré applet, you can control the motion of the pattern. (To "click-and-drag" means to press the mouse button and then move the mouse, while holding down the button.) If you shift-click on the applet, you can stop and restart its motion. (To "shift-click" means to hold down the shift key while you click the mouse button.) If you are using a slow computer, you might want to turn off the Moiré applet, so that it doesn't take computer processing time away from other things going on on this page.

Each of the labs for The Most Complex Machine uses an applet to help you learn something about computer science. In order to make it more convenient to use the applets and read the labs at the same time, the applets are set up to run in separate windows. The lab worksheet contains a button that you can click to launch the applet. In most of the labs, this button is close to the beginning of the lab, where it will be easy to find. (The button is itself a small applet that runs on the Web page.)

In this lab, you will use a fairly simple applet called "DataReps". You will use this applet to learn how the same binary number can be used to represent different types of data. To launch the applet, click on this button:

(Sorry, your browser doesn't do Java!)

The window that opens when you click this button will probably be marked with some kind of warning, to alert you to the fact that the window was created by an applet. For example, on my computer, there is a warning bar like this one along the bottom of windows opened by applets:

Why the warning? A Java applet is a program that you have downloaded from the Internet. Whenever you download a program, there is a danger that the program is malicious -- that it will try to damage your computer or steal information from you. A great deal of attention has been paid to making Java applets secure, that is to making sure that they can't damage your computer or access private information stored on your computer. However, nothing can stop you from entering private information, such as a password, into an applet. The warning on applet windows is there to stop malicious applets from tricking you into entering such information. For example, without the warning, the applet might imitate a window from a program that has a legitimate need for the information.

Data Representations

You'll be using the "DataReps" applet, which you launched above, in some of the exercises at the end of the lab. For now, you should read about it and experiment with it to see what it does.

This applet lets you type in a data value. You can select the type of data you want to enter by clicking on one of the five radio buttons. Just type your data into the input box at the top of the applet, and press return. You can also click on the 8-by-4 grid of "big pixels" at the center of the applet. The applet takes the data you enter, and it converts that data into a 32-bit binary number. (It has to do this in order to store it!) It then takes that same binary number and interprets it in six different ways. The six interpretations are: a binary number, an integer, a hexadecimal number, a real number, a string of four characters, and an eight-by-four grid of pixels. You should remember that you see the same string of thirty-two bits interpreted in different ways. You should also remember that the same bit-patterns could also be interpreted in an endless variety of additional ways: as a bar of music, or the chemical ingredients in a bar of soap, or your tab at your favorite bar, or....)

Here is a short explanation of each of the six data displays. You should try entering various types of values in the applet to see how they are represented as binary numbers.

Binary

This is the most direct display of the 32 bit binary number, showing a zero or one to represent each individual bit. The displayed binary number shows the full 32 bits, including any leading zeros. The computer stores the zeros, even though you don't ordinarily include leading zeros when you write a number.

Base-ten Integer

A binary number can be interpreted as a normal positive integer (0, 1, 2, 3, 4,...) written in the "base ten". Base ten is the usual way of writing numbers, using the digits 0 through 9. See Section 1.1 of The Most Complex Machine. With 32 bits, you can represent 2^{32} different numbers. Usually, you want to use both positive and negative numbers. The scheme for representing negative numbers is a bit strange. It is explained in Subsection 2.2.3 of the text. Using 32 bits, the integers from -2147483648 to 2147483647 can be represented.

Hexadecimal

It is difficult (for humans) to read long strings of zeros and ones. Hexadecimal numbers are a kind of shorthand for writing such strings. A hexadecimal number is written using the sixteen "hexadecimal digits" 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Each hexadecimal digit stands for four bits. So 0 represents 0000, 1 represents 0001, 2 represents 0010, ..., E represents 1110, and F represents 1111. We could also say that the hexadecimal digit A stands for the base-ten number 10 (ten), B stands for 11 (eleven), C stands for 12, D for 13, E for 14, and F for 15. A hexadecimal number is really just a number written in the base sixteen, just as ordinary integers are in the base ten and binary numbers are in the base two. You should be able to translate between hexadecimal and binary by hand. (But of course, you could use the applet instead.)

Real Number

Real numbers are numbers that can contain decimal points, like 3.14159 or -234.5, or 12.0. They can also be written using "scientific notation." For example, 2.15e12 is a way of writing 2.15 times 10^{12}. The representation used in computers for real numbers is very complicated. And it allows some strange possibilities, such as INF

and -INF, which stand for infinity and minus infinity. There are also NAN's. NAN stands for "not a number." NAN's are used to represent the results of illegal operations such as taking a square root of a negative number. Note that the **integer** 17 and the **real number** 17 have completely different representations in the computer, even though they are the same number mathematically. All-in-all, it's probably best not to worry about the internal representation of real numbers. I include this data type here for completeness, since real numbers are so important.

ASCII Text

Characters can be encoded using ASCII code, as explained in Section 1.1 of the text. Each possible character is assigned a code that is one byte (that is, eight bits) long. With 32 bits, you can represents 4 characters in ASCII code. Not every possible byte represents an ordinary, printable character. The applet shows other bytes in the form <#n>, where n is the base-ten number corresponding to the byte. For example, the byte 00000111, which is equivalent to 7 in base ten, is shown as <#7>.

Pixels

At the center of the applet, you will see an 8-by-4 grid of little squares. Each of these thirty-two squares corresponds to one bit in the binary number. You should think of these squares as being very big pixels. Each pixel can be either black or white. One bit specifies the color of one pixel -- 0 for white or 1 for black. This is how two-color graphical images can be represented by binary numbers. Again, see Section 1.1 of the text. In the applet, you can change the color of a pixel by clicking on it.

Exercises

Exercise 1: For this exercise, your goal is to use a search engine such as Alta Vista or WebCrawler to find an interesting page on the World Wide Web. Pick a topic that interests you. Think up some terms related to that topic, and search for pages containing those terms. Pick out one you find interesting. Don't settle for some boring generic page like ESPN Sports or Apple Computer Inc! Write a short paragraph saying what your topic was and how you went about doing the search. Also include the URL for the page that you find.

Exercise 2: Search the Web to find the poem written by the Greek poet Sappho about her daughter Cleis. How did you go about finding the poem? Where did you find it?

Exercise 3: Starting from one of the links given above, find the radius, in kilometers, of the planet Jupiter. How did you go about finding it?

Exercise 4: Guess the URL of the home page of each of the following. Explain your reasoning. (Check the Web to see whether you are right.)

- The IBM corporation
- The FBI, an agency of the US government
- Harvard University
- The SPCA, an animal-rights organization

Exercise 5: Pick out one or two of the following phrases. Each phrase is a fragment of a reasonably well-known quotation. Search the Web for uses of the phrase. (Use the Alta Vista advanced search; enter the phrase **in quotes** into the text-input box. This will **not** work with the regular Alta Vista search.) Try to find the complete phrase and the original source of the phrase. Also, try to find a few interesting variations that people have used on their Web pages.

- "a truth universally acknowledged"
- "yes, Virginia, there is"
- "on the shoulders of giants"

Exercise 6: In addition to working on some of the above exercises, you should spend some time "surfing" the World-Wide Web. Write a short essay describing your experiences with the Web and speculating on its possible impact and importance.

Exercise 7: Use the "DataReps" applet to find the following. In each case, indicate briefly what you did with the applet to answer the question.

- Find the ASCII code of the upper case letter X.
- Find the character that has an ASCII code equal to 63.
- What real number has the same binary representation as the hexadecimal number 4228AE14?
- What real number has the same binary representation as the four-letter word "Fred"?
- What binary number represents the base-10 number -3?

Exercise 8: Enter the following base-10 integers into the "DataReps" applet: 1, 2, 4, 8, 16, 32, 64. Describe the corresponding pixel representation of these numbers. (The pixel representation is displayed in the center of the applet). What pattern do you see? Why does this pattern occur? What can you say about the binary representation of these numbers?

Exercise 9: Enter a four-letter word such as "TIME" into the "DataReps" applet. (Select "ASCII Text" as the input type, type the word into the applets input box, and press return.) Consider the pixel representation of the word, which is displayed in the center of the applet. Play with the pixels in the third column of pixels from the left. Turn them on and off by clicking on them, and observe what happens to each letter in the word. What happens? What does this tell you about the ASCII coding of letters? (What is the meaning of the third bit in that encoding?)

Exercise 10: This final exercise is meant to be a longer essay question. You should try to show your understanding of the way data is represented in a computer, and an appreciation for the fact that meaning depends on context and convention.

It would be legal to input 1000 into the Data Representation Applet as either a binary number, a base-ten integer, a hexadecimal number, a real number, or as ASCII text. In each case, the input is represented differently -- as a different binary number. How is it possible that five different binary numbers can all represent "1000"? What is going on here? How can the computer keep all the different meanings straight?

This is one of a series of labs written to be used with The Most Complex Machine: A Survey of Computers and Computing, an introductory computer science textbook by David Eck. For the most part, the labs are also useful on their own, and they can be freely used and distributed for private, non-commercial purposes. However, they should not be used as a formal part of a course unless The Most Complex Machine is also adopted for use in that course.

--*David Eck* (*eck@hws.edu*), *Summer 1997*

Labs for The Most Complex Machine

xLogicCircuits Lab 1: Logic Circuits

IT IS POSSIBLE IN THEORY to construct a computer entirely out of transistors (although in practice, other types of basic components are also used). Of course, in the process of assembling a computer, individual transistors are first assembled into relatively simple circuits, which are then assembled into more complex circuits, and so on. The first step in this process is to build logic gates, which are circuits that compute basic logical operations such as AND, OR, and NOT. In fact, once AND, OR, and NOT gates are available, a computer could be assembled entirely from such gates. In this lab you will work with simulated circuits made up of AND, OR and NOT gates. You will be able to build such circuits and see how they operate. And you will see how simpler circuits can be combined to produce more complex circuits.

This lab covers some of the same material as Chapter 2 in The Most Complex Machine. The lab is self-contained, but many of the ideas covered here are covered in more depth in the text, and it would be useful for you to read Chapter 2 before doing the lab.

This lab includes the following sections:

The lab uses an applet called "xLogicCircuits." Start the lab by clicking this button to launch the xLogicCircuits applet in its own window:

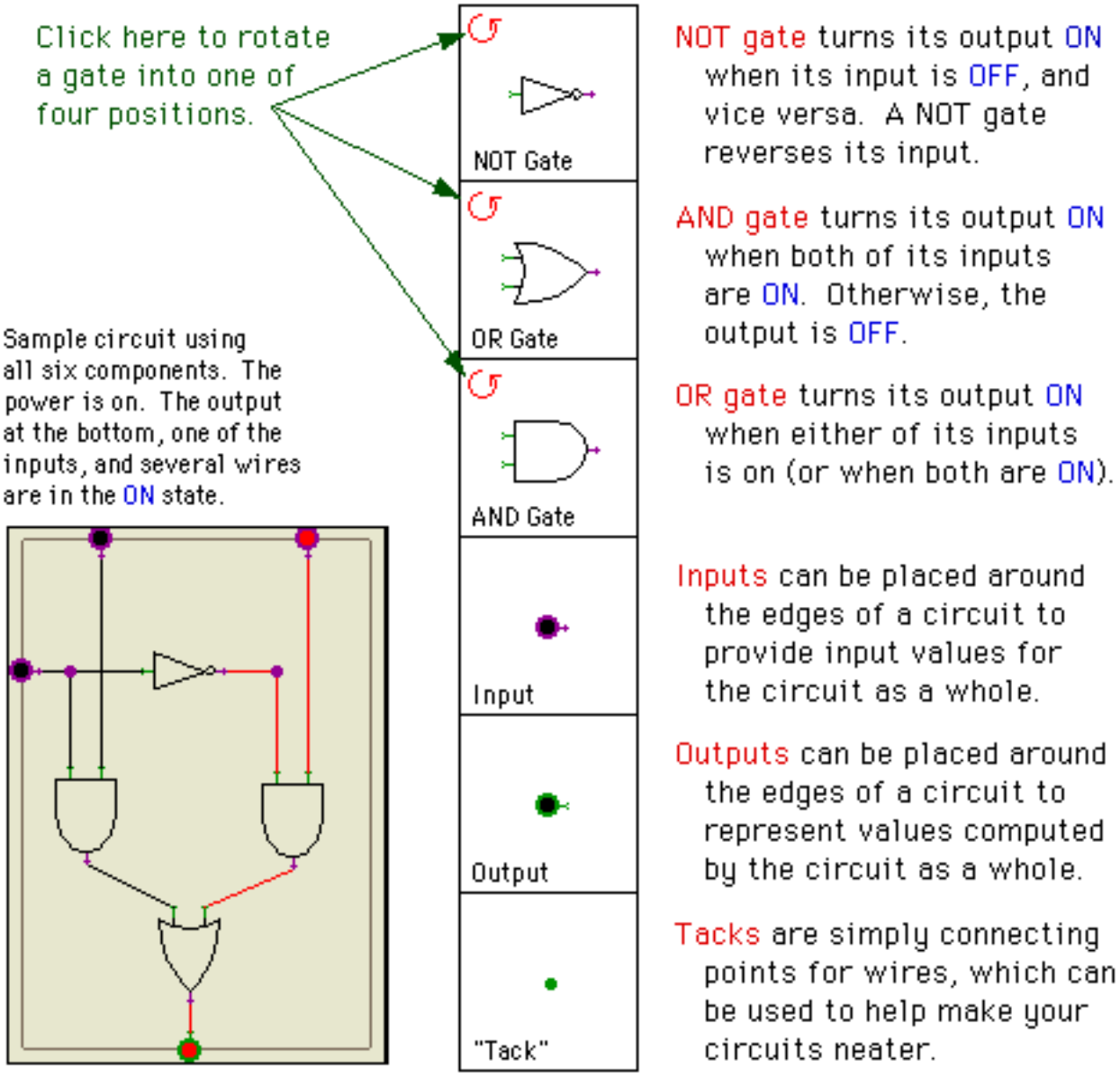
(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Logic and Circuits

A logic gate is a simple circuit with one or two inputs and one output. The inputs and outputs can be either ON or OFF, and the value of a gate's output is completely determined by the values of its inputs (with the proviso that when one of the inputs is changed, it takes some small amount of time for the output to change in response). Each gate does a simple computation. Circuits that do complex computations can be built by connecting outputs of some gates to inputs of others. In fact, an entire computer can be built in this way.

In the xLogicCircuits applet, circuits are constructed from AND gates, OR gates, and NOT gates. Each type of gate has a different rule for computing its output value. Circuits are laid out on a circuit board. Besides gates, the circuit board can contain Inputs, Outputs, and Tacks. Later, we'll see that circuits can also contain other circuits. All these components can be interconnected by wires. To the left of the circuit board in the applet is a pallette. The pallette contains components available to be used on the circuit board. You can't usually see all the components at once, but there is a scroll bar that allows you to scroll through all the components on the pallette. The following illustration shows the part of the pallette that contains the six standard components, along with some comments and a small sample circuit:



(One thing you should note: Wires cannot connect to each other except at Tacks. Just because two wires cross each other on the circuit board does not mean that they are connected. That is, no signal will propagate from one of the wires to another. Wires can only carry signals between components such as gates, Tacks, Inputs, and Outputs.)

The applet that you launched above should start up showing a sample circuit called "Basic

Gates." At the top of the circuit board are an AND gate, an OR gate, and a NOT gate. The gates are connected to some Inputs and Outputs. A more complicated circuit built from several gates occupies the bottom of the circuit board.

To see how the circuit works, you have to turn on the power. Power to the circuit board is turned on and off using the "Power" checkbox below the circuit board. The power is ON when the box is checked. **Click on the Power checkbox now to turn on the power.** (Why does the wire leading from the NOT gate come on when you do this?) When the power is on, you have control over the Inputs on the circuit board: you can turn an input ON and OFF by clicking on it. The circuit does the rest: signals from the Inputs propagate along wires, through gates and other components, and to the Outputs of the circuit. **Try it** with the sample circuit. If you have a problem, make sure the power is on and that you are clicking on an Input, not an Output.

You should check that the AND, OR, and NOT gates at the top of the circuit board have the expected behavior when you turn their inputs ON and OFF. You can also investigate the circuit in the bottom half of the logic board. Below the circuit board, to the left of the Power switch, you'll find a pop-up menu that can be used to control the speed at which signals propagate through the circuit. The speed is ordinarily set to "Fast." You can use the pop-up menu to change the speed to "Moderate" or "Slow" if you want to watch the circuit in slow motion. (For the most part, though, you probably want to leave the speed set to Fast.)

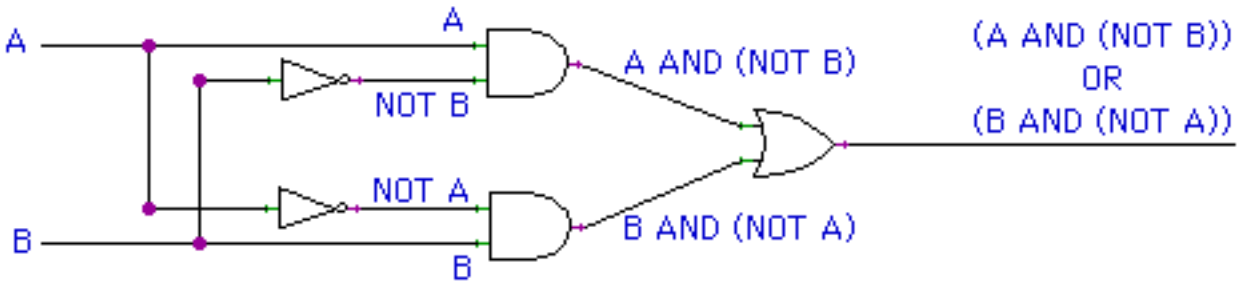
Logic gates and logic circuits are associated with mathematical logic, which is the study of the computations that can be done with the logical values **true** and **false** and with the logical operators **and**, **or**, and **not**. This association comes about when we think of ON as representing **true** and OFF as representing **false**. In that case, AND, OR, and NOT gates do the same computations as the operators **and**, **or**, and **not**.

Mathematical logic uses Boolean algebra, in which the letters A, B, C, and so on, are used to represent logical values. Letters are combined using the logical operators **and**, **or**, and **not**. For example,

(A **and** C) **or** (B **and** (**not** C))

is an expression of Boolean algebra. As soon as the letters in an expression are assigned values **true** or **false**, the value of the entire expression can be computed.

Every expression of boolean algebra corresponds to a logic circuit. The letters used in the expression are represented by the Inputs to the circuit. Each wire in the circuit represents some part of the expression. A gate takes the values from its input wires and combines them with the appropriate word -- **and**, **or**, or **not** -- to produce the label on its output wire. The final output of the whole circuit represents the expression as a whole. For example, consider the sample circuit from the applet. If the inputs are labeled A and B, then the wires in the circuit can be labeled as follows:



The circuit as a whole corresponds to the final output expression, (A **and** (**not** B)) **or** (B **and** (**not** A)). This expression in turn serves as a blueprint for the circuit. You can use it as a guide for building the circuit. The expression given earlier, (A **and** C) **or** (B **and** (**not** C)), corresponds to another sample circuit shown in the illustration above -- provided you label the inputs appropriately.

To sum up, given any expression of Boolean algebra, a circuit can be built to compute that expression. Conversely, any output of a logic circuit that does not contain a "feedback loop" can be described by a Boolean algebra expression. This is a powerful association that is useful in understanding and designing logic circuits. (Note: Feedback occurs when the output of a gate is connected through one or more other components back to an input of the same gate. Circuits with feedback are not covered in this lab. However, they have important uses that are covered in the next lab.)

Building Circuits

You can build your own circuits in the xLogicCircuits applet. Click on the "Iconify" button at the bottom of the applet. This will put away the "Basic Gates" circuit, by turning it into an icon on the pallette. You'll have a clear circuit board to work on. As an exercise, try to make a copy of the sample circuit shown above, which corresponds to the Boolean expression

(A **and** C) **or** (B **and** (**not** C)).

To add a component to your circuit, click on the component in the pallette, hold down the mouse button, and use the mouse to drag the component onto the circuit board. Make sure you drag it completely onto the board. If you want a gate that is facing in a different direction, you have to rotate the gate in the pallette **before** you drag it onto the circuit board.

Once some components are on the board, you can draw wires between them using the mouse. Every wire goes from a source to a destination. To draw a wire, move the mouse over the source, click and hold the mouse button, move the mouse to the destination, and release the button. You **must** draw the wire from source to destination, not the reverse. If you release the mouse button when the wire is not over a legal destination, no wire will be drawn. When there are two possible destinations in one component -- such as the two inputs of an AND or OR gate -- make sure that you get the wire connected to the right one.

Circuit Inputs are valid sources for wires. So are Tacks. So are the **outputs** of gates. Valid destinations include circuit Outputs, inputs of gates, and Tacks. You can draw as many

wires as you want **from** a source, but you can only draw **one** wire to a destination. (This makes sense because when the circuit is running, a destination takes its value from the single wire that leads to it. On the other hand, the value of a source can be sent to any number of wires that lead from it.)

Once a component is on the board, you can still move it to a new position, but you have to **drag it using the right mouse button**. Alternatively -- if you have a one-button mouse, for example -- you can drag a component by **holding down the control key** as you first press the mouse button on it.

You can **delete** components and wires that you've added by mistake. Just click on the component or wire to hilite it. Then click on the "Delete" button at the bottom of the applet. The hilited item will be deleted from the circuit board. If you delete a component that has wires attached, the attached wires will also be deleted along with the component.

If you delete an item or modify the circuit in some other way, you get one chance to change your mind. You can click on the "Undo" button to undo one operation. Only the most recent operation can be undone in this way.

There is one shortcut that you might find useful, if you like using Tacks. You can insert a Tack into an existing wire by double-clicking on the wire. If you double-click and hold the mouse down on the second click, you can drag the tack to a different position. (However, some browsers might not support double-clicks.)

After you build the practice circuit, you can clear the screen, since you won't need that circuit again in the rest of the lab. However, you'll get more practice building circuits in the Exercises at the end of the lab.

Complex Circuits and Subcircuits

In order to have circuits that display structured complexity, it is important to be able to build on previous work when designing new circuits. Once a circuit has been designed and saved, it should be possible to use that circuit as a component in a more complex circuit. A lot of the power of xLogicCircuits comes from the ability to use circuits as components in other circuits. Circuits used in this way are called subcircuits. A circuit that has been saved as an icon in the pallette can simply be dragged into another circuit. (More exactly, a **copy** of the circuit is created and is added to the circuit board. The copy is a separate circuit; editing the original will not change the copy.) This ability to build on previous work is essential for creating complex circuits.

You can open a circuit from the pallette to see what's inside or to edit it. Just click on the icon to hilite it, and then click on the "Enlarge" button. The icon will be removed from the pallette and the circuit will appear on the circuit board. At the same time, any circuit that was previously on the circuit board will be iconified and placed on the pallette. You should also be able to enlarge a circuit just by double-clicking on it. (By the way, you can change the name of the circuit on the circuit board by editing the text-input box at the top of the applet. This box contains the name that appears on the iconified circuit.)

The xLogicCircuits applet should have loaded several subcircuits for the pallette. One of these circuits is called "Two or More". **Open this circut now.** The circuit has three inputs. It turns its output ON whenever at least two of its inputs are ON. Try it. (Click on the inputs to turn them ON and OFF -- and don't forget to turn the Power on first.)

As a simple exercise in building circuits from subcircuits, use the "Two or More" circuit as part of a "At Most One" circuit. You want to build a circuit with three inputs that will turn on its output whenever zero or one of its inputs is on. Notice that this is just the opposite behavior from the "Two or More" circuit. That is, "At Most One" is ON whenever "Two or More" is **not** ON. This "logical" description shows that the "At Most One" circuit can be built from a NOT gate and a copy of the "Two or More" circuit. Begin by re-Iconifying the "Two or More" circuit, then drag a NOT gate and a copy of "Two or More" onto the empty circuit board. Add Inputs, Outputs, and wires as appropriate, then test your circuit to make sure that it works. If you like, you can give it a name and turn it into an icon.

Next, open the "4-Bit Adder" sample circuit. You'll see that it contains several copies of a subcircuit called "Adder." It's possible to look inside one of these circuits: Just click on the adder circuit to hilite it, and then click the "Enlarge" button. This does **not** remove the main circuit from the board -- it just lets you see an enlarged part of it. When you shrink the subcircuit back down to its original size, the main circuit is still there. In this case, you'll see that an "Adder" circuit contains two "Half Adder" subcircuits, which you can enlarge in their turn, if you want.

Circuits and Arithmetic

The "4-Bit Adder" circuit is an example of a logic circuit that can work with binary numbers. Circuits can work with binary numbers as soon as you think of ON as representing the binary value 1 (one) and OFF as representing the value 0 (zero). The "4-Bit Adder" can add two 4-bit binary numbers to give a five digit result. Here are some examples of adding 4-bit binary numbers:

```
 1011      1111      1111      1010      0111      0001
 0110      0001      1111      0101      1010      0011
-----     -----     -----     -----     -----     -----
10001     10000     11110     01111     10001     00100
```

The answer has 5 bits because there can be a carry from the left-most column. Each of the four "Adder" circuits in the "4-Bit Adder" handles one of the columns in the sum. You should test the "4-Bit Adder" to see that it gets the right answers for the above sums. The two four-bit numbers that are to be added are put on the eight Inputs at the top of "4-Bit Adder". The sum appears on the outputs at the bottom, with the fifth bit -- the final carry -- appearing on the output on the right. You should observe that it takes some time after you set the inputs for the circuits to perform its computations.

Exercises

Exercise 1: One of the examples in this lab was the circuit corresponding to the expression

(A **and** (**not** B)) **or** (B **and** (**not** A)).

This circuit is ON if **exactly one** of its inputs is on. Another way to describe the output is to say that it is ON if "one or the other of the inputs is on, but not both of the inputs are on." This description corresponds to the Boolean expression

(A **or** B) **and** (**not** ((A **and** B)).

Build a circuit corresponding to the second expression, and check that it gives the same output as the first circuit for every possible combination of inputs.

Exercise 2: When you checked "every possible combination of inputs" for the circuit in Exercise 1, how many combinations did you have to check? If you wanted to check that the "Two or More" example circuit works correctly for every possible combination of inputs, how many combinations would you have to check? Why? If you wanted to check that the "4-Bit Adder" gives the correct answer for each possible set of inputs, how many inputs are there to check? Why?

Exercise 3: Consider the following three Boolean algebra expressions:

(A **and** B **and** C) **or** (**not** B)

(**not** ((**not** A) **and** (**not** B)))

(**not** (A **or** B)) **or** (A **and** B)

For each expression, build a logic circuit that computes the value of that expression. Write a paragraph that explains the method that you apply when you build circuits from expressions. (One note: To build a circuit for an expression of the form (X **and** Y **and** Z), you should insert some extra parentheses, which don't change the answer. Think of the expression as ((X **and** Y) **and** Z), and build the circuit using two AND gates.)

Exercise 4: Given a logic circuit that does not contain any feedback loops, it is possible to find a Boolean algebra expression that describes each output of that circuit. Open the circuit called "For Ex. 4", which was one of the sample circuits in the applet's pallette. This circuit has four inputs and three outputs. Assuming that the inputs are called A, B, C, and D, find the expression that corresponds to each of the three outputs. Also write a paragraph that discusses the procedure that you apply to find the Boolean expression for the output of a circuit.

Exercise 5: Consider the following input/output table for a circuit with two inputs and one output. The table gives the desired output of the circuit for each possible combination of inputs.

Input 1	Input 2	Output

ON	ON	ON
ON	OFF	ON
OFF	ON	OFF
OFF	OFF	ON

Construct a circuit that displays the specified behavior. You have to build **one** circuit that satisfies **all four** rows of the table. Section 2.1 of The Most Complex Machines gives a general method for constructing a circuit specified by an input/output table. You can apply that method, or you can just try to reason logically about what the table says. Write a paragraph discussing how you found your circuit.

Exercise 6: One of the examples in this lab was a circuit called "Two or More", which checks whether at least two of its inputs are on. Consider the problem of finding a similar circuit with four inputs. The output should be on if any two (or more) of the inputs are on. A circuit that does this can be described by the Boolean expression:

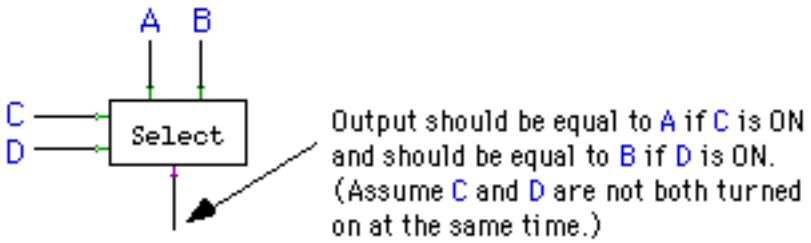
(A **and** (B **or** C **or** D)) **or** (B **and** (C **or** D)) **or** (C **and** D)

Use this expression to construct a "Two or More" circuit with four inputs. Try to understand where this expression comes from. Why does it make sense? (Hint: Think of two cases, one case where the input A is ON, and the other case where the input A is OFF.) Write a paragraph explaining this. The form of this expression can be extended to handle circuits with any number of inputs. Write down a logical expression that describes a circuit with **five** inputs that turns on its output whenever two or more of the inputs are on.

Exercise 7: "The structure of the 4-Bit Adder circuit reflects the structure of the compution it is designed to perform." In what sense is this true? What does it mean? How does this relate to problem-solving in general?

Exercise 8: Write a short essay (of several paragraphs) that explains how subcircuits are used in the construction of complex circuits and why the ability to make and use subcircuits in this way is so important.

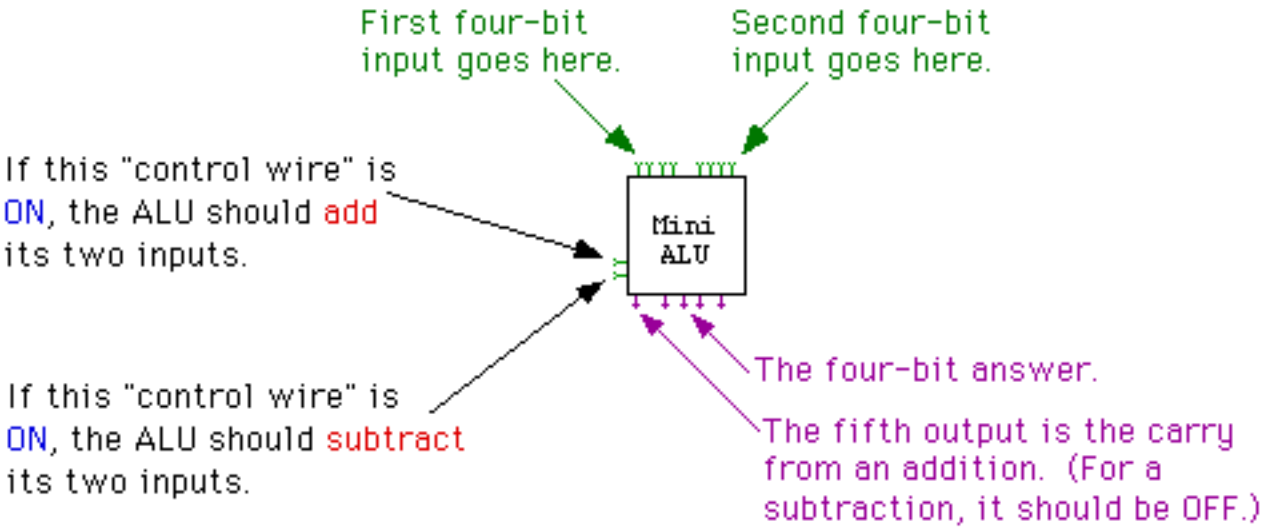
Exercise 9: Build a "Select" circuit, as shown in this illustration:



The circuit has two inputs, A and B, at the top. It also has two inputs, C and D, on the left, which serve as control wires. (The only thing that makes an input of a circuit a control wire is that the designer of the circuit says it is, but in general control wires are thought of as controlling the circuit in some way.) The control wires determine which of the inputs, A or B, gets to the output. In order to do this exercise, you should think "logically." That is, try to

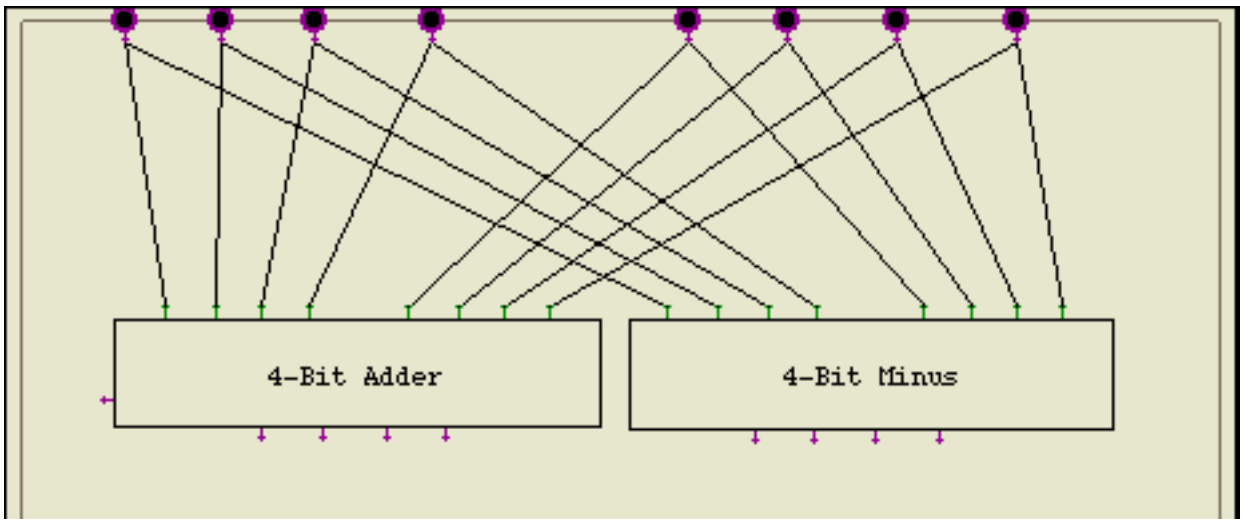
describe the output of the circuit using a Boolean expression involving A, B, C, and D. Then use that expression as a blueprint for the circuit. **Test your circuit** and save it for use in Exercise 10.

Exercise 10: For this exercise, you should build a "Mini ALU" that can do either addition or subtraction of four-bit binary numbers. An Arithmetic Logic Unit, or ALU, is the part of a computer that does the basic arithmetic and logical computations. It takes two binary numbers and computes some output. The interesting thing is that an ALU can perform several different operations. It has control wires to tell it which operation to perform. You will build an ALU that can perform either addition or subtraction of four-bit binary numbers. It has two control wires. Turning on one of these will make it do an addition; turning on the other will make it do a subtraction. You should construct the circuit as specified by this illustration:



Now, an interesting thing about an ALU is that it actually performs all the computations that it knows how to do. The control wires just control which of the answers get to the outputs of the ALU. To make your "Mini ALU," you can start with the "4-Bit Adder" and "4-Bit Minus" circuits, which were provided to you in the applet's palette. (The four-bit subtraction circuit has only four outputs, since for subtraction the carry bit from the leftmost adder does not provide any useful information. You don't have to worry about how the Minus circuit works -- you don't even have to understand how negative numbers are represented in binary.)

Start by placing a "4-Bit Adder" and a "4-Bit Minus" circuit on an empty circuit board, along with the eight inputs at the top of the circuit. These can be connected as shown:



(Note: To change the size and shape of a subcircuit, click the circuit to hilite it. When a circuit is hilited, it is surrounded by a rectangle with a little square handle in each corner. You can click-and-drag one of these handles to adjust the size of the circuit.)

All you have to do is construct the rest of the circuit so that the control wires can control whether the answer from the "4-Bit Adder" or the answer from the "4-Bit Minus" gets through to the Outputs of the ALU. One way to do this is to use four copies of the "Select" circuit that you built for Exercise 9.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xLogicCircuits Lab 2: Memory Circuits

THIS LAB CONTINUES THE STUDY OF CIRCUITS built from logic gates, which was begun in the previous lab. That lab showed how circuits can be built to perform arithmetic and logical computations with binary numbers. Such computations are one of the major functions of computers. But computers also need at least two other abilities. They need memory -- the ability to store and retrieve data. And they need control -- the ability to control what data is stored where, which computations are performed, and in what order. A program specifies a series of computations to be performed by a computer; the computer stores program and data in its memory, and the computer executes the program under its own control, without any further direction from its user or programmer. You already have some idea how circuits can perform computations. In this lab, you'll see how logic gates can be used to build memory circuits that can store binary numbers (which are used to represent both programs and data). As for control functions, they can also be implemented with gates and wires. There were some hints of this in the previous lab, and you'll see more in this lab. However, the full mystery of how a computer can execute a program all on its own will not be solved until future labs.

The material in this lab is also covered in Sections 2.3 and 3.1 of The Most Complex Machine. Not everything in the book is repeated here. As usual, you will find it useful to read the book before doing the lab. You also **definitely** need to do the previous lab before this one.

This lab includes the following sections:

You'll be using the xLogicCircuits applet in this lab. Start by clicking the button to launch the applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Circuits that Remember

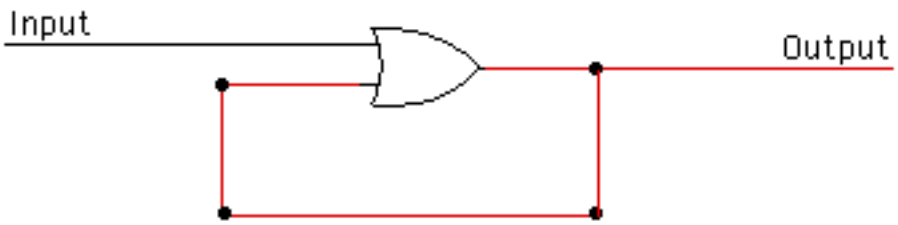
You've seen that a circuit that does not have any feedback loops simply takes the values from its inputs and computes an output value based on those inputs. Changing the values on the input wires will change the output values -- after just a very short delay for the signal to pass through the circuit. Such circuits have no "memory." The output is computed based on the current inputs, and anything that happened to the circuit in the **past** has no effect.

In this lab, we are interested in circuits that have some memory of what happened to them in the past. That is, the output of the circuit is not based solely on the current inputs. It can also depend on inputs that were given to the circuit in the past. Memory circuits include feedback loops. A feedback loop occurs when the output from a gate is connected back to an input of the same gate -- possibly through one or more other gates. Such a loop allows **previous** inputs to affect **current** outputs. This is exactly what we need for memory circuits.

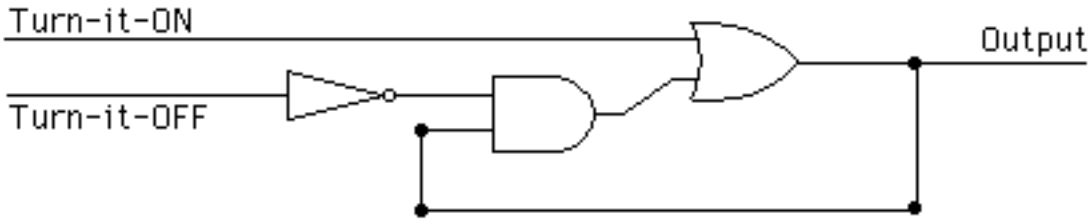
The xLogicCircuits applet is set up to load several examples. You should see three sample circuits on the circuit board. Each of these is a simple circuit containing a feedback loop. You should **turn on the power** and experiment with these circuits.

The first circuit consists of a NOT gate, with its output connected back to its input. What happens to this circuit when you turn on the power? This is not what I would call a memory circuit! It shows that not every circuit that contains a feedback loop can properly be called a memory circuit. (In fact, building memory circuits is a pretty touchy affair.) However, even this simple example of a feedback loop turns out to be useful and interesting, as you'll see in some of the exercises at the end of the lab.

The second example on the circuit board is an OR gate whose output is fed back to one of its inputs:



The other input to the OR gate is under your control. As soon as you turn this input on, the feedback loop turns on and the output of the circuit comes on. After that, you can turn the input off and on as much as you want. The output stays on (until you turn off the power to the circuit). The circuit "remembers" that its input has been turned on sometime in the past. This is interesting, but it would be nice to have a way of turning the circuit off. The third example on the circuit board shows how this can be done:



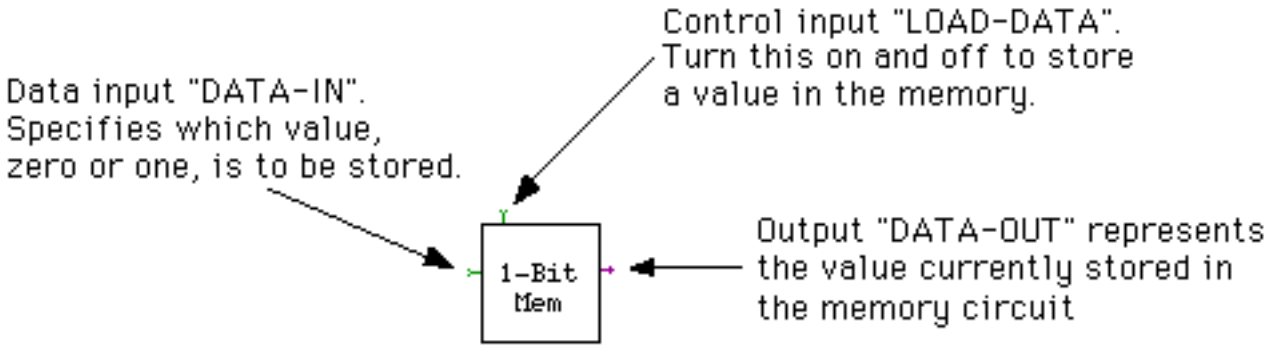
This circuit also contains an OR gate in a feedback loop, but now an AND gate has been inserted into the loop. If you turn the input labeled "Turn-it-ON" on and then off, the feedback loop and the output of the circuit will come ON. If you turn the other input, labeled "Turn-it-OFF," on and then off, the loop and output will go OFF. This circuit remembers which of its inputs was most recently turned on and off. Since we want to work with binary numbers, we think of ON as representing one and OFF as representing zero. We say that we store one in the circuit by turning the top input on and off, and that we store zero in the circuit by turning the lower input on and off. The circuit stores the value zero or one. You can check which value it is storing just by looking at its output. You can set the value that it is storing by manipulating its inputs. This simple memory circuit is the basic building block for most of what you will see in this lab.

In the previous lab, you saw that it is often convenient to think of some inputs to a circuit of as carrying data into the circuit, while other inputs are used to control the circuit. The circuit itself doesn't really make any distinction between two different types of inputs, but it useful for designers of circuits to distinguish between data inputs and control inputs depending on what functions the inputs serve in their circuit design. All the circuits that you see in the rest of this lab follow the following convention:

- **Circuit outputs are on the right edge of the circuit.**
- **Data inputs are on the left edge of the circuit.**
- **Control inputs are on the top and bottom edges of the circuit.**

In the simple memory circuit discussed above, it is not really clear to me whether the two input wires are data inputs or control inputs. For this reason -- and also because it will fit better into the design of other circuits -- we will use a modified version of this memory circuit for the rest of the lab. The basic memory circuit that we will use is in the example called "1-Bit Mem". You'll find this circuit in the scrolling pallette in the xLogicCircuits applet. (Remember that you can use the circuits in the pallette as building blocks in other circuits simply by dragging them onto the circuit board. You can also see inside any iconified circuit; just click on the circuit to hilite it and then click the "Enlarge" button.)

The one-bit memory circuit can store one bit, either zero or one. You won't need to understand the inside of this circuit, but it important that you understand how it is used. The circuit has one data input, one control input, and one output. I refer to these as DATA-IN, LOAD-DATA, and DATA-OUT:



To store a value in the circuit, turn the DATA-IN wire ON or OFF to represent the value (ON for one or OFF for zero). Turn LOAD-DATA on and off. (You have to do this slowly enough to allow the circuit time to react.) When you turn LOAD-DATA on, the value from DATA-IN will flow into the circuit. When you turn LOAD-DATA off, the value in the circuit will be "locked" and cannot change until the next time LOAD-DATA is turned on. You can always check what value is stored in the circuit by looking at its output wire, DATA-OUT.

Open the "1-Bit Mem" circuit, turn on the power, and work with it to make sure that you know how to use it.

Registers

With the basic one-bit memory as a starting point, you can build more complex memory circuits. Just as you can line up several "Adder" circuits to get a multibit addition circuit, you can combine several one-bit memory circuits to get a multibit memory circuit. As one of the exercises at the end of the lab, you will construct a 4-bit memory circuit. A 4-bit memory circuit can store a 4-bit binary number. Similarly, you can build memory circuits to store 8-bit binary numbers, 16-bit numbers, or any number of bits. Memory circuits of this type are important components in the central processing unit of a computer. A memory circuit that is used in the central processing unit to store a binary number is called a register. Multibit memory circuits can also be used in the main memory of a computer. Recall that main memory contains a sequence of numbered locations. Each location stores a binary number, so each location can be a simple multibit memory circuit.

For some purposes, a more sophisticated type of multibit memory is needed. As an example, look at the "Count Reg" circuit, which you will find in the pallette of the xLogicCircuits applet. This circuit is a register that **counts** in binary. Turn its control wire on and off several times (allowing, as always, enough time for the circuit to respond). As you keep turning the control wire ON and OFF, the three outputs of the circuit will cycle through the values 000, 001, 010, 011, 100, 101, 110, 111, and back to 000. (Read the outputs from bottom to top.) If you convert these binary numbers to decimal numbers, the circuit is counting from zero to seven. A count register of this sort could be used in a CPU to count off the steps in a computation, for example.

The count register is made from three "Flip Flop" circuits. It is not important for you to

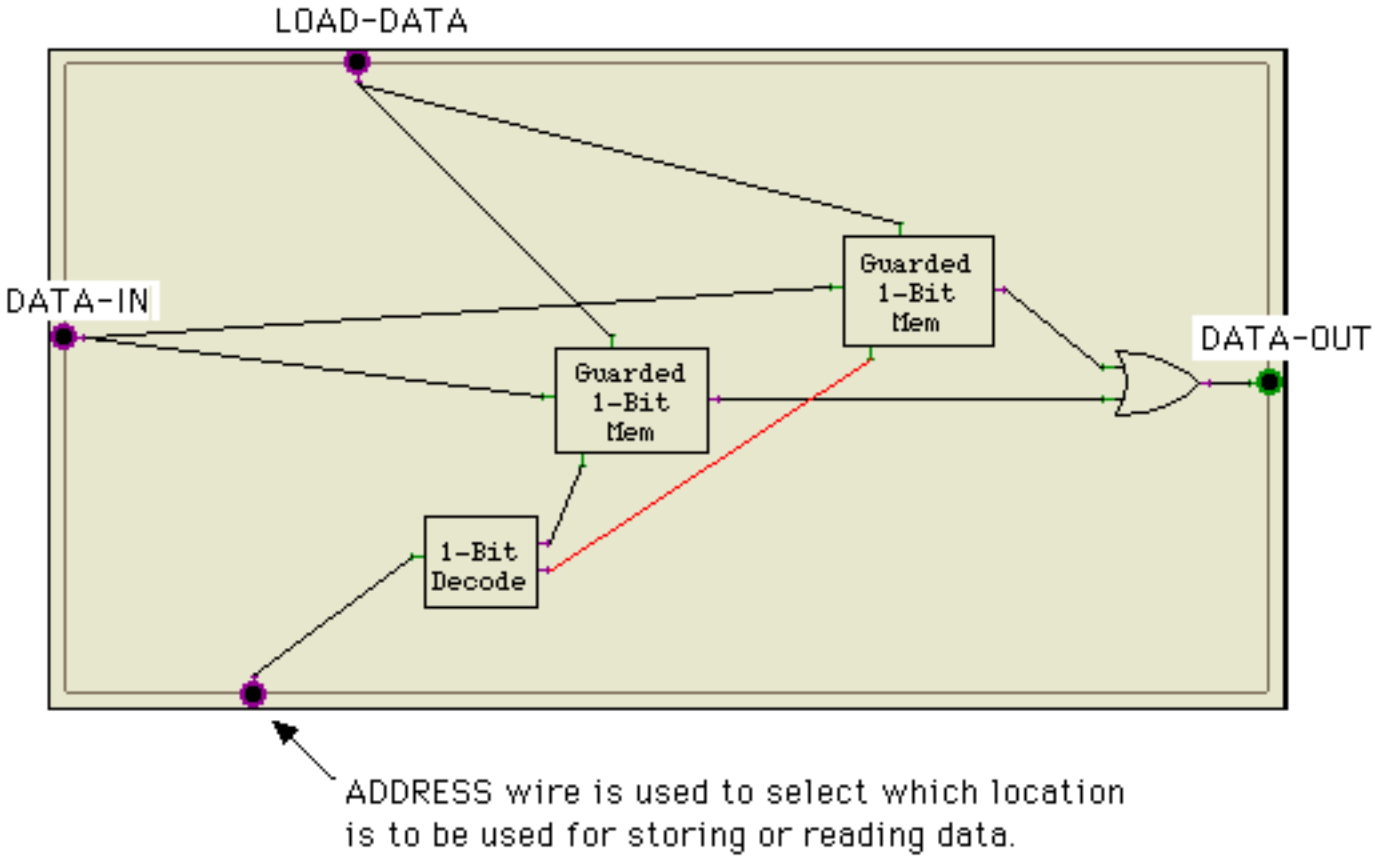
understand how a flip-flop works, but if you look inside, you'll see that it is made from two interconnected one-bit memories of the type you have seen above. In fact, a flip-flop is itself a kind one-bit memory, but it can do things that the simpler one-bit memory cannot -- such as count. The count register is used in one of the exercises at the end of the lab, but mainly it is here to show you that registers can be made to do more than just store numbers.

Random Access Memory

A random access memory, or RAM, is a memory circuit that can hold several different binary numbers. Each binary number is stored in a separate location. The locations are numbered 0, 1, 2, 3, and so on. The number of a location is called its address. Every computer has a RAM, which it uses as a main memory where it stores the data and programs that it is working with. A RAM can have any given number of locations. (In a typical computer, the RAM has several million locations.) The binary numbers that are stored in a RAM can have any given number of bits. (In a typical modern computer, each location holds an eight-bit binary number.)

In this section, you'll see how a very simple RAM can be constructed. The RAM you will look at has only two locations, and each location holds just a single bit. The exercises at the end of the lab will investigate how larger and more useful RAM's can be built.

The sample RAM is named "2-Bit RAM". It is one of the sample circuits loaded by the xLogicCircuits applet. Find it in the applet's pallette and open it. You'll see a circuit with one data input, two control inputs, and one output:



The ADDRESS wire picks out one of the two locations in the RAM. When the ADDRESS wire is OFF, one location is used (the location "with address zero"). When the ADDRESS wire is ON, the other location is used (the location "with address one"). The DATA-OUT wire shows the contents of the selected location. Turning the LOAD-DATA wire on and off stores a value in the selected location. The DATA-IN wire specifies the value that is to be stored.

For example, suppose you want to store a 1 in location 0. Then you should:

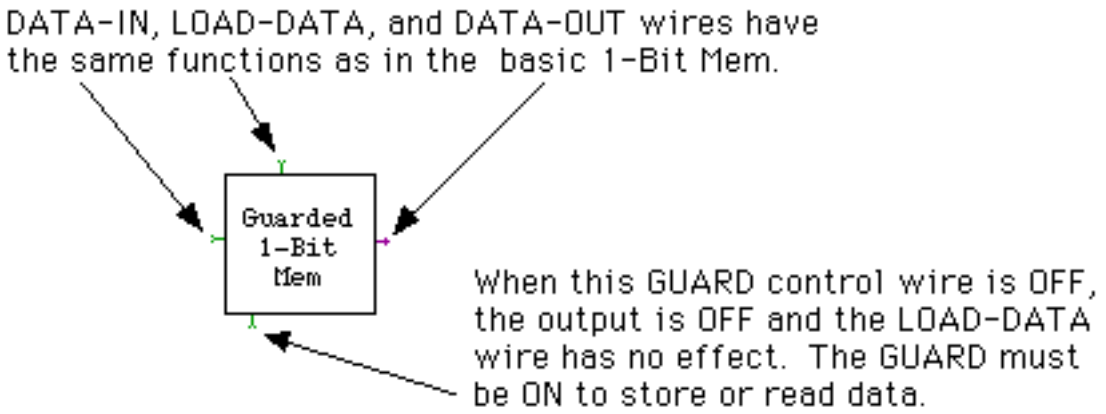
- Turn the DATA-IN wire ON, representing the value 1.
- Turn the ADDRESS wire OFF, representing the address of location 0.
- Turn the LOAD-DATA wire ON.
- Wait long enough for the signals to propagate through the circuit.
- Turn the LOAD-DATA wire OFF.

To read the value stored in location 0, all you have to do is turn the ADDRESS wire OFF to represent the address of the location you want to read. The value stored in location 0 will appear on the DATA-OUT wire.

If you want to work with location 1 instead of with location 0, all you have to do is turn the ADDRESS wire ON.

A real RAM should have more than one ADDRESS wire. Every combination of values that can be put on the ADDRESS wires specifies a different address. For example, if there are three address wires, they can be set to any of the eight combinations OFF-OFF-OFF, OFF-OFF-ON, OFF-ON-OFF, OFF-ON-ON, ON-OFF-OFF, ON-OFF-ON, ON-ON-OFF, and ON-ON-ON. These combinations represent the binary numbers 000, 001, 010, 011, 100, 101, 110, and 111. In ordinary decimal notation, they represent 0, 1, 2, 3, 4, 5, 6, and 7. So, a RAM with three ADDRESS wires can have eight locations, numbered 0 through 7. Twenty ADDRESS wires are enough to specify over a million locations, and thirty ADDRESS wires can specify over a billion.

You should spend some time understanding how to use the "2-Bit RAM" and how it works. Each of the two locations in the RAM is a "Guarded 1-Bit Mem" circuit. This circuit is similar to the basic "1-Bit Mem" circuit that you looked at above, but it has an addition control input, called the GUARD:



When the GUARD of a location is ON, that location is selected. When it is selected, it outputs its stored value on its DATA-OUT wire, and its LOAD-DATA wire can be used to store a new value in it. When the GUARD is OFF, the "Guarded 1-Bit Mem" is inert. It ignores its LOAD-DATA wire and leaves its DATA-OUT wire turned OFF.

The "1-Bit Decode" is used to make sure that exactly one of the two locations is selected at any given time. The ADDRESS wire is used as input to the Decode circuit. The Decode circuit has two outputs. When the ADDRESS wire is OFF, the Decode circuit turns its lower output ON, and this in turn selects one of the "Guarded 1-Bit Mem" circuits. When the ADDRESS wire is ON, the Decode circuit turns its upper output ON, and this in turn selects the other "Guarded 1-Bit Mem" circuit. Thus, the Decode circuit decodes the ADDRESS to decide which location to select.

Finally, the OR gate is necessary so that both locations will be able to send their output to the DATA-OUT wire of the RAM. You can't connect two wires to one output. Using the OR gate allows either of the "Guarded 1-Bit Mem" circuits to turn on the output.

Exercises

Exercise 1: A clock in a computer is a component that "ticks" by turning its output wire on and off. This regular ticking can be used as a signal by other components in the computer. Open the sample circuit "Clock" in the applet that you launched above. This example is a clock that will start ticking (by turning its output on and off) as soon as you turn the power off. However, you can stop the ticking by turning on the clock's control wire at the top of the circuit. Turning this control wire off will restart the clock. Write a few paragraphs explaining exactly how this circuit works. Why does the clock tick? What process does the clock go through as it turns its output on, then off, then back on? What is the role of the OR gate? (**Note**: The "Tacks" in the feedback loop serve to slow the ticking down, because in this simulated circuit, it takes a bit of time for a signal to propagate from one Tack, along a wire, to the next Tack. In a real circuit, the length of the feedback loop could be used to control how long a signal takes to circle the loop.)

Exercise 2: This is a continuation of Exercise 1. As an example of using the signal from a clock to drive another component, build a circuit in which the output from a "Clock" circuit is connected to the control wire of a "Count Reg" circuit. The circuit you build should have

three outputs, which are connected to the outputs from the Count Register. It should also have one control input, which is connected to the Clock's control wire. When you turn the power on, the ticking of the Clock will make the Count Register count, and it will do so as long as the control wire is turned off. This circuit just counts up to 7 before it goes back to zero. How would you make a circuit that counts up to 15 before it goes back to zero?

Exercise 3: One of the sample circuits, "1-Bit Mem", is capable of storing a one-bit binary number. Construct a four-bit memory that can store a four-bit binary number. It can be built using four copies of "1-Bit Mem". Your circuit should have four DATA-IN wires and four DATA-OUT wires. However, it should have only one LOAD-DATA control wire. You want to load data from the DATA-IN wires into all the "1-Bit Mem" circuits at the same time. This will be done by turning the single LOAD-DATA wire on and off.

Exercise 4: This continues Exercise 3. Explain in detail how you would store the binary number 1011 in the four-bit memory that you constructed for Exercise 3. Also, explain carefully what it **means** to say that this value is "stored" in the memory.

Exercise 5: This exercise uses the four-bit memory that you built for Exercise 3. One of the examples in this lab was a "2-Bit RAM" circuit that can store two one-bit numbers. It stores one number in each of two locations, and an ADDRESS wire is used to determine which of the two locations is in use. Suppose that you want to store a **four-bit** number in each of two locations. The circuit would be very similar to the "2-Bit RAM" but it would have four Inputs and four Outputs. It would still have just one LOAD-DATA control wire and one ADDRESS wire to select between the two locations. For this exercise, you should build such a circuit. You can start with your four-bit memory. Use it to build a "Guarded 4-Bit Mem" modeled on the "Guarded 1-Bit Mem" from the "2-Bit RAM" circuit. Then use two copies of the "Guarded 4-Bit Mem" to construct a RAM with two locations. Test your circuit by storing a different four-bit number in each location. Explain how you can check that your circuit has stored the numbers correctly.

Exercise 6: The "2-Bit RAM" sample circuit uses a "1-Bit Decode" circuit. This circuit has one input and two outputs. It "decodes" its input by turning one of its outputs on when its input is ON and by turning the other output on if the input is OFF. Similar decoder circuits with more inputs are also useful. For this exercise, build a two-bit decoder circuit. Your circuit should have two Inputs and four Outputs. The two inputs can have any of the following pairs of values: 00, 01, 10, or 11. Each of these pairs corresponds to one of the Outputs. The decoder circuit should turn on a different output for each pair of inputs. That is, if the inputs have the value 00 (both OFF), then the circuit should turn on its first Output; if the Inputs have the value 01, it should turn on its second Output; and so on. So at any given time, exactly one of the outputs will be ON. (This is a fairly simple circuit. Begin by finding a Boolean algebra expression for each of the outputs.)

Exercise 7: This is a continuation of Exercise 6. The two-bit decoder circuit that you built for Exercise 6 can be used as a component in a four-location RAM. The four locations in the RAM will be "Guarded 1-Bit Mem" circuits. The RAM will have two ADDRESS wires, which connect to the two inputs of the decoder. The four outputs from the decoder should connect to the GUARD wires of the four "Guarded 1-Bit Mem" circuits. This allows the

values on the ADDRESS wires to be used to select one of the four locations. For this exercise, build such a four-location RAM. (As in the original "2-Bit RAM" example, each location should hold just one bit.)

Exercise 8: Based on your work in Exercises 5 and 7, explain how it would, in principle, be possible to build a RAM containing any given of locations, with any given number of bits in each location.

Exercise 9: Write a short essay explaining what you learned in this lab about constructing complex circuits from subcircuits.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xComputer Lab 1: Introduction to xComputer

THIS LAB INTRODUCES the xComputer applet, which simulates a simple model computer (which is also called xComputer). The model computer is discussed in Chapter 3 of The Most Complex Machine. The xComputer consists of a Central Processing Unit (CPU) and a main memory that holds 1024 sixteen-bit binary numbers. The CPU contains an Arithmetic-Logic Unit (ALU) for performing basic arithmetic and logical computations. It also contains eight registers, which hold binary numbers that are being used directly in the CPU's computations, a Control circuit, which is responsible for supervising the computations that the CPU performs, and a clock, which drives the whole operation of the computer by turning its single output wire on and off.

The xComputer applet that you will use in this lab lets you load programs and data into the memory of the simulated xComputer. You can then watch while those programs are executed, and you can observe how numbers stored in the computer change as a program runs. The applet displays only the registers and main memory. You have to take the control circuit, ALU, and clock on faith.

This lab contains basic information about xComputer and its machine language. It demonstrates how instructions are fetched from memory and executed by the CPU. It will also explain the features of the xComputer applet and the process of programming the xComputer. The next lab will cover the programming process in more detail.

You would find it useful to read through Chapter 3 of the text before doing this lab. Chapter 3 is rather technical, and you might find that you need to work through both this lab and that chapter before you really understand either of them.

This lab includes the following sections:

Start by clicking this button to launch the xComputer applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

The xComputer Applet

The xComputer applet is divided into three sections. The right-hand third of the applet represents the main memory of the simulated computer. This section of the applet shows the 1024 locations in xComputer's memory. These locations are numbered from 0 to 1023. Each line in the memory shows a location number (in blue) and the value stored in that location. When the program first starts up, the memory contains only zeros. The scroll bar can be used to view any part of memory.

The "Control" section of the applet is used to interact with and control the xComputer. For example, you can use text-input boxes and buttons in the Control section to enter programs and data into the xComputer's memory. Once a program has been loaded into memory, you can tell the xComputer to execute it. You'll learn about controlling the xComputer as you work through the lab.

The "Registers" section of the applet shows the xComputer's eight registers. Remember that a register is simply a memory unit in the CPU that holds data being used directly in the CPU's computations. Each register plays a particular role in the execution of programs by the CPU. These roles are described in detail in the text and will be illustrated during the course of this lab, but here for your reference is a brief summary:

- The X and Y registers hold two sixteen-bit binary numbers that are used as input by the ALU. For example, when the CPU needs to add two numbers, it must put them into the X and Y registers so that the ALU can be used to add them.
- The AC register is the accumulator. It is the CPU's "working memory" for its calculations. When the ALU is used to compute a result, that result is stored in the AC. For example, if the numbers in the X and Y registers are added, then the answer will appear in the AC. Also, data can be moved from main memory into the AC and from the AC into main memory.
- The FLAG register stores the "carry-out" bit produced when the ALU adds two binary numbers. Also, when the ALU performs a shift-left or shift-right operation, the extra bit that is shifted off the end of the number is stored in the FLAG register.
- The ADDR register specifies a location in main memory. The CPU often needs to read values from memory or write values to memory. Only one location in memory is accessible at any given time. The ADDR register specifies that location. So, for example, if the CPU needs to read the value in location 375, it must first store 375 into the ADDR register. (If you turn on the "Autoscroll" checkbox beneath the memory display, then the memory will automatically be scrolled to the location indicated by the ADDR register every time the value in that register changes.)
- The PC register is the program counter. The CPU executes a program by fetching instructions one-by-one from memory and executing them. (This is called the fetch-and-execute cycle.) The PC specifies the location in memory that holds the next instruction to be executed.
- The IR is the instruction register. When the CPU fetches a program instruction from main memory, this is where it puts it. The IR holds that instruction while it is being executed.

- The COUNT register counts off the steps in a fetch-and-execute cycle. It takes the CPU several steps to fetch and execute an instruction. When COUNT is 1, it does step 1; when COUNT is 2, it does step 2; and so forth. The last step is always to reset COUNT to 0, to get ready to start the next fetch-and-execute cycle. This is easier to understand after you see it in action. Remember that as the COUNT register counts 0, 1, 2,..., just **one** machine language program is being executed.

You will learn how the xComputer works by giving it a short program and watching it execute that program. A program is a sequence of assembly language instructions. You have to enter the instructions into the xComputer's memory. But first, make sure that the "addr" input box in the Control area of the applet contains a zero. The number in this box specifies the address in memory where the instruction that you type will be stored. Then, type the following instructions into the "data" input box. Press return after each instruction (or, equivalently, click the "Data to Memory" button):

```
lod-c 17
add-c 105
sto 10
hlt
```

When you press return, an instruction is translated into a machine language instruction and is stored in the xComputer's memory. (You'll actually see a number in the memory, rather than the assembly language instruction that you typed. For example, the instruction "lod-c 17" is represented by the number 25617.) Note that when you press return, the number in the "addr" input box is automatically incremented by one, to get ready for storing the next instruction that you type in the next memory location.

The first instruction of this program, lod-c 17, tells the xComputer to load the constant 17 into the accumulator. The second instruction, add-c 105 tells the computer to add the constant 105 to whatever number is in the accumulator and to put the result back into the accumulator. The sto 10 makes the computer copy the contents of the accumulator into memory location 10. And the final instruction, hlt, tells the computer to halt. The net effect is that the program adds 105 to 17 and stores the answer in location 10. After the program halts, you can look in memory at location 10 to find the answer.

Now, how do you make the computer **run** the program? All the computer ever does is fetch instructions from memory and execute them. The value stored in the PC register tells the computer which memory location to go to to get the next instruction. Before running the program, you have to make sure that it will begin with the first instruction of the program, which is in memory location zero. This means that the PC register should contain a zero. If this is not the case, you can put a zero into the PC by clicking the "Set PC = 0" button in the Control section of the applet. (Zero is the most common starting value for the PC, but if you want to start at a different location, you can type the address of that location into the "addr" input box and click the "Addr to PC" button.)

Once you have checked the value of the PC register, you can just click on the "Run" button to run the program. You should think of this as turning on the xComputer's clock. As soon as you do this, the clock stars ticking, the value in the COUNT register starts changing, and

the fetch-and-execute cycle proceeds. This will continue until the computer executes a HLT instruction, or until you stop it.

You should **Try this now**. Make sure that the PC contains a zero. Then, click on the "Run" button. If you have done everything correctly, the program will run. You will see things happening, although you will probably not really understand them at this point. But you will notice that the instructions in the program appear one by one in the IR register as they are executed. Eventually, the HLT instruction will be executed and the computer will stop running. The correct answer to the computation, 122, will be in memory location 10. This gives you the general idea of how programs are executed by xComputer. Your goal in the rest of the lab is to understand the details.

After running the program once, you should run it again. First, **reset the PC to zero**. This time, instead of using the "Run" button, use the "Cycle" button. Clicking on "Cycle" makes the computer run, but only until the value in the COUNT register is 2. At that point, the computer has just loaded a new instruction into the IR and is about to execute that instruction. When you click on "Cycle" again, the computer will execute that instruction and fetch the next instruction. Thus, the "Cycle" button lets you step through a program one instruction at a time. Try it! Keep clicking on "Cycle" until you get to the HLT instruction.

Finally, you can run the program one more time, this time with the "Step" button. Clicking on the "Step" button makes the computer perform a single step in the fetch-and-execute cycle. You have to click on it several times just to execute one instruction in the program. Once again, you should **reset the PC to zero**. Then, click through your program with the "Step" button until the HLT is executed.

Writing Programs for xComputer

It should be clear that entering a long program into xComputer by typing it into the "data" box would be very tedious and error-prone. If you accidentally leave out one instruction, for example, you might have to retype most of the program! Fortunately, the xComputer applet lets you type a complete program in a separate text-input area and then translate the whole program at once and store it in xComputer's memory. This has three advantages: You can edit the program in the window, for example by inserting a new instruction. You can (if the configuration of your Web browser permits it) save the program in a file, so that you'll never have to retype it again. And you can use labels in your program. Labels are a powerful programming technique; they are described in the Postscript to Chapter 3 in the text. They are not covered in this lab, but they will be an important part of the next lab.

If you want to type a new program, just click on the "New Program" button in the Control area of the applet. Alternatively, you can select "[New]" from the pop-up menu at the very top of the applet. The computer display will disappear and will be replaced by a text-input area when you can type your program. Enter the following program into that text area:

```
lod-c 1
sto 12
lod 12
```

```
inc
sto 12
jmp 2
```

This program **counts**. It starts by putting the number 1 into memory location 12, and then it adds one to the number in that location over and over, forever. (You'll see this in action in a moment.) There are several new instructions here. Lod 12 tells xComputer to copy the number from memory location 12 into the accumulator. (Note how this differs from lod-c 12, which puts the number 12 itself into the AC, rather than the number stored in memory location 12.) The inc instruction adds one to the value in the accumulator. And jmp 2 is a jump instruction that sends the computer back to location 2.

After typing this program, click on the "Translate" button that is located below the text area, on the left end of a row of buttons. If the program contains some error, an error message will be displayed. If you've typed the program correctly, it will be translated into machine language and stored in the xComputer's memory. The text area will be replaced by the computer display, and the computer will be ready to run the program. Click on the "Run" button to run the program and see how it operates. You can watch as the PC counts off the instructions in the program. You will see the assembly language instructions themselves as they are loaded into the IR. And you can observe that the value in memory location 12 changes from 1 to 2 to 3 to 4 and so on. This program will run forever, if you let it.

You will be working with this little program throughout most of the remainder of the lab. Your objective is to understand how xComputer operates and to appreciate the fetch-and-execute cycle.

Controlling Speed and Memory Display Style

As you let the counting program run, you can try varying the speed at which the computer executes instructions by changing the setting on the speed pop-up menu (located just below the "Run" button. The lower speeds allow you to watch what is happening in more detail. The higher speeds allow the computer to get more done. At the very highest speed, the registers are not displayed, so that the computer can run as quickly as possible, without updating the register display all the time.

When you have had enough of this, **stop the program** and experiment with the memory display style pop-up menu, which is located just above the scrolling memory display. This menu allows you to select how you would like to view the contents of main memory. (There is also a similar pop-up menu for setting the register display style.) Of course, the actual contents of memory are binary numbers, but a binary number can mean many things, depending on how it is interpreted. When the applet first starts up, it is set to display the contents of memory as ordinary decimal integers in the range -32768 to 32767. This is only one possible interpretation of the binary numbers that are stored in memory. Using the display style pop-up menu, you can select from six different interpretations:

- The Instructions display shows the contents of each memory location as an assembly language instruction. In this display style, you should see the original counting

program in memory locations 0 through 5. Most of the other locations contain Add 0, which just happens to be the assembly language instruction encoded by the 16-bit binary number 0000000000000000. (Since not every 16-bit binary number corresponds to a legitimate assembly-language instruction, you might see some funny things in this display style.)

- The Integers and Unsigned Ints displays show ordinary decimal integers. The difference is that signed 16-bit integers are in the range -32768 to 32767, while unsigned 16-bit integers are in the range 0 to 65535. (In either case, there are 2^{16} different possible values -- it's just a question of how they are interpreted. See Subsection 2.2.3 in the text.)
- The Binary display shows a 16-bit binary number in each memory location; this display style is closest to the actual physical contents of the memory.
- The ASCII display interprets each sixteen-bit number in memory as made up of two eight-bit ASCII character codes, and shows the two characters. Some eight-bit binary numbers do not represent visible ASCII characters. These numbers are shown in the form <#N>, where N is the number, in decimal form. Thus, for example, the 16-bit binary number 0000000000000000 is shown in ASCII display style as <#0><#0>.
- The Graphics display is very different from the others. It shows the **entire** memory at once. Each bit in memory -- all 16 times 1024 of them -- is represented by one pixel on the screen. That pixel is white if the bit is zero and is black if the bit is one. If you choose the Graphics display now, the memory will be almost entirely white, except for a few black dots at the top that represent the program you entered into memory.

You can try out the various memory display styles. You'll be using them in Exercise 1 at the end of the lab. I should note that when you enter information into Memory using the "data" input box in the Control area of the applet, you can type the information in several of the above display styles, as well as in assembly language. You can, for example, enter ordinary numbers in the range -32768 to 65535. You can enter a binary number, but you must precede it by the letter B. For example: B1011010111. Finally, you can enter one or two ASCII characters, but you must precede them by a quote mark. For example: '#1 or 'A. You will need to do this for Exercise 1.

There is another option in the pop-up menu: "Control Wires". This display style doesn't show memory at all. If you select it, the computer's memory display will be replaced by a list of control wires. These control wires are the key to understanding how the xComputer works. The basic idea is that turning control wires on and off makes things happen in the computer. They are turned on and off by a Control Circuit, and they control the operation of other components of the CPU. Each control wire has a function. Turning that wire on causes something to happen, such as moving a number from main memory into the AC register or adding the numbers in the X and Y registers and putting the answer into the AC. Executing a program is just a matter of turning the right wires on and off in the right sequence.

The "Control Wires" display lets you see what wires are turned on during each step in the execution of an instruction. Try it with the instruction lod-c 17, by doing the following: **First**, enter the instruction "lod-c 17" into some memory location, and set the PC to the

address of that location. **Next**, set the display style to "Control Wires". **Then**, use the "Step" button to go through the fetch-and-execute cycle one step at a time. Here's what you will see:

- First click on the "Step" button: COUNT becomes 1, indicating that the first step in the fetch and execute cycle is being performed. The Load-addr-from-PC control wire is turned on, and the value in the PC register is copied into the ADDR register. (The PC register tells which memory location holds the next instruction; that location number must be copied into the ADDR register so that the computer can read that instruction from memory.)
- Second click: COUNT becomes 2. The Load-IR-from-Memory control wire is turned on, and an instruction is copied from memory into the IR. (The ADDR register determines which instruction is read.) In this case, the instruction is Lod-c 17.
- Third click: COUNT becomes 3. The Increment-PC control wire is turned on, and the value in the PC register is incremented by 1. Ordinarily, this prepares the PC for the **next** fetch-and-execute cycle. This completes the "fetch" portion of the fetch-and-execute cycle. The remaining steps in the cycle depend on the particular instruction that is begin executed (in this case, lod-c 17).
- Fourth click: COUNT becomes 4. The Load-AC-from-IR control wire is turned on. The data part of the instruction in the IR register, is copied into the accumulator. In this case, the value is 17. This is the only step necessary to execute the lod-c 17 instruction.
- Fifth click: COUNT becomes 5, but only briefly. The Set-COUNT-to-Zero control wire is turned on and immediately the value of COUNT is reset to 0. One fetch-and-execute cycle is over. (On the next click, COUNT would become 1 again, and the next cycle would begin.

As you click on the "Step" button in this exercise, you are actually simulating the role of the xComputer's clock. Each click has the same effect as one tick of the clock, and you are driving the computation at your leisure in the same way as the ticking of the clock usually drives the computer with its regular ticking.

Count and Store

For the last part of the lab, consider the following program, "CountAndStore". This program should have been automatically loaded by the xComputer applet, so you won't have to type it in. Just select it from the pop-up menu at the top of the applet, and click on the "Translate" button to store it in the xComputer's memory. Note that in an xComputer program, anything that comes after a semicolon on a line is a comment, which is meant for human readers. Comments are ignored by the computer.

```
lod-c 1     ; Start with a 1 in location 12
sto 12
```

```
lod 12      ; This instruction is stored in location 2
inc
sto 13      ; This instruction is stored in location 4

lod 2       ; Add 1 to the number in location 2
inc
sto 2

lod 4       ; Add 1 to the number in location 4
inc
sto 4

jmp 2       ; Go back to the instruction in location 2
```

This program is similar to the simple counting program that you looked at earlier in the lab, except that the number for the second sto command has been changed, and six new instructions have been inserted before the jmp command. The instructions "lod 2, inc, sto 2" add 1 to the number stored in memory location 2. But if you look at what's stored in that location, you'll find the instruction lod 12, which is part of the program. This seems odd. What happens when you "add 1" to an **instruction**?

Remember that machine language instructions are really just numbers. There is no problem with adding 1 to a number. However, the meaning of the instruction represented by the number is changed. If you add 1 to the number that encodes "lod 12," the meaning of the answer is "lod13." If you want to understand exactly why this is true, look at the binary representations of the machine language instructions. (The details are given in Section 3.2 of the text, if you want to check there.)

Run this program and see what it does. To fully appreciate this program, you should run it at "Fastest Speed" with the memory display set to "Graphics". You can watch as the memory is gradually filled with numbers.

Exercises

Exercise 1: You can use the xComputer to translate from one type of data to another by entering it in one form in the "data" input box and viewing it in memory in another form. Use this method to do the following conversions, and explain briefly how you do each part:

1. Find the ASCII code for the character #. (The ASCII code is the **integer** that represents this ASCII character.)
2. Find the character whose ASCII code is 99.
3. Find the binary representation of -233.
4. Find the unsigned integer that has the same binary representation as the signed integer -233.

5. Find the unsigned integer that represents the assembly language instruction sto 1023.
6. Now, Add 1 to the number that represents sto 1023 and find the assembly language instruction represented by the resulting number. Why do you get a completely different instruction? (Note: Do the addition yourself; you don't have to program the computer to do it!)

Exercise 2: In reality, the memory of a computer contains only binary numbers. Machine language, in particular, consists of binary numbers. Translate the counting program, which is repeated below, into the binary numbers of machine language, and write a paragraph or two explaining **why** computers use binary numbers instead of something more readable.

```
lod-c 17
sto 12
lod 12
inc
sto 12
jmp 2
```

Exercise 3: Write an assembly language program that computes 34 - 17 + 103 - 12. The instruction for subtracting a constant from the accumulator is sub-c.

Exercise 4: Earlier in the lab, you were asked to step through the execution of the instruction lod-c 17, which tells the computer to load the number 17 into the accumulator. The instruction lod 17 tells the computer to copy the contents of memory location 17 into the accumulator. Use xComputer to watch as this instruction is executed step-by-step, just as you did above for the lod-c 17. Enter a lod 17 instruction into memory location zero, and **reset the PC to zero**. Set the display style to "Control Wires". Then use the "Step" button to step through the fetch-and-execute cycle as the lod instruction is executed. Write down what happens during each step. Carefully explain the purpose of each step in the execute phase of the cycle (steps 4 and later). What differences do you find between the execution of a lod-c instruction and the execution of a lod instruction? How can the differences be explained in terms of what the instructions do?

Exercise 5: Use the "Step" button to trace the execution of an add-c instruction and of an add instruction. (See Exercise 4 for detailed instructions about how to do this.) Record the control wires that are on during each step in the execute part of the fetch and execute cycle, that is for steps number 4 and later, and carefully explain the purpose of each of those steps. What differences do you observe between these two instructions? How can the differences be explained in terms of what the instructions do?

Exercise 6: Carefully explain why the first three steps of the fetch-and-execute cycle are always the same, and why they have nothing to do with the contents of the instruction register.

Exercise 7: Modify the first counting program used in this lab so that it will count just from one to sixteen, stopping when it reaches sixteen. (The program is repeated below.) To do this, each time through the loop, you need to test whether the number is sixteen. If it is, jump to a HLT instruction at the end of the program. Testing whether a number is sixteen

requires two steps: First, subtract 16 from the number, and then test whether the answer is zero. Use a JMZ instruction to test whether the answer is zero. (This is like a JMP instruction, except that the jump only occurs if the number in the AC is zero.) Write a paragraph explaining how your program works.

```
lod-c 1    ; This is the original counting program.
sto 12
lod 12
inc
sto 12
jmp 2
hlt        ; Add a halt instruction at the end.
```

Exercise 8: Describe what is done by the "CountAndStore" program, which you encountered earlier in the lab. Discuss **how** it works in some detail. To do a good job on this exercise, you will have to step through several executions of the loop in the program and study how it works.

Exercise 9: Discuss what you learn from the "CountAndStore" program about "data" and "instructions" and the relationship between them. (The memory of a computer can hold both data and instructions. How does the computer distinguish between them? Does it?)

Exercise 10: Run the "CountAndStore" program at "Fastest Speed" with the display set to "Graphics." If you let the program run long enough, it will halt. How is this possible, since the program contains no HLT instruction? To figure this out, you'll need to do some detective work after the program ends. Look at what has happened to the program in the computer's memory. Try to be as explicit and complete with your explanation as possible. This is not an easy question. (Hint: something you did in Exercise 1 turns out to be relevant here.)

--*David Eck* (*eck@hws.edu*), *Summer 1997*

Labs for The Most Complex Machine

xComputer Lab 2: Assembly Language Programming

THE MACHINE LANGUAGE FOR xComputer consists of thirty-one different instructions. Each instruction performs a very simple task. Nevertheless, very complex programs can be built up from these instructions. The previous lab introduced the xComputer applet and the basic xComputer machine language instructions. In this lab, you will learn more about programming the xComputer. Hopefully, you'll begin to appreciate how complex programs can be composed from very simple instructions.

Machine language consists of binary numbers, but it would be almost impossible for people to program if they had to write programs directly in binary. Instead, programmers use assembly language or high-level language. The programs they write in these languages are translated by assemblers and compilers into machine language. You'll use a high-level language called "xTurtle" in later labs. In this lab and the next, you'll use assembly language.

Assembly language is closely related to machine language, but has several features that make it much easier to use. You've already seen that assembly language uses meaningful instruction names, such as `ADD-C`, instead of numerical instruction codes. In this lab, you'll see how memory locations and data values can also be referred to by name rather than by number. Such names are called labels. You'll also learn about a few new machine language instructions, which use indirect addressing.

The material in this lab is based on Chapter 3 of The Most Complex Machine. Labels are introduced in the postscript to that chapter.

This lab includes the following sections:

Start by clicking this button to launch the xComputer applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Labels and the Assembler

The previous lab introduced the xComputer applet and the most basic aspects of xComputer's assembly language. Here is an example repeated from that lab. This program simply counts forever by adding 1 to location 12 over and over in an infinite loop:

```
LOD-C 1
STO 12
LOD 12
INC
STO 12
JMP 2
```

The numbers 12 and 2 in this program are addresses of memory locations. While this short program is not terribly difficult to understand, it can be extremely tedious and error prone to keep track of the large number of numerical addresses that would be used by a complex program. For this reason, assembly language allows you to give names to memory locations, so that you can refer to the locations with meaningful names rather than meaningless numbers. Names used in this way are called labels.

To give a name to a memory location in an assembly language program, all you have to do is put the name, followed by a colon, before the contents of the memory location. For example, the line

```
Loop:   LOD 12
```

in a program would give the name "Loop" to the memory location that contains the `LOD 12` instruction. Elsewhere in the program, you can use the word "Loop" to refer to that location, instead of using the numerical address. For example, you could jump to that location with the command `JMP Loop`.

Memory locations can be use to hold either **instructions** or **data**. Labels are useful in both situations. If "Num" is the label of a location that is used to hold data, then it would make sense to use "Num" in data-manipulation commands such as `LOD Num`, `STO Num`, and `ADD Num`. Labels for instructions, on the other hand, are mostly used in jump commands.

Here is a version of the counting program that uses labels. The name "Loop" is used for the instruction that begins the loop, and "Count" is used for the location that contains the value of the counter:

```
          LOD-C 1       ; Set Count equal to 1
          STO Count
Loop:     LOD Count     ; Add 1 to Count
          INC
          STO Count
          JMP Loop      ; Jump back to start of loop

          @12
Count:    data          ; Location to be used for counting
```

This example introduces two other features of xComputer's assembly language: "@" and "data". The word "data" is used as a place-holder for a memory location that is going to be used by the

program to contain a data value. When the assembler translates the program into machine language and loads it into memory, it replaces "data" with a zero. (Actually, you could just use a 0 in the program, but "data" is more descriptive of your intentions.) The line "`@12`" is not translated into machine language. It is a directive to the assembler telling the computer to store the next item in location 12. In this case, it means that the Count will be stored in location 12. Without the "`@12`", it would be stored in location 6, the next sequential location after the `JMP Loop` instruction. If there were additional items after Count, they would be stored in locations 13, 14, and so on -- until the next "@" directive.

This program is one of the sample programs loaded by the xComputer applet that you launched above. Select the program "SimpleCounter.txt" from the pop-up menu at the top of the xComputer window. Load it into xComputer's memory by clicking on the "Translate" button. Once it is loaded, you can use the "Run," "Step," or "Cycle" button to execute the program. If you need more information about using the xComputer applet, please see the previous lab.

Loops and Decisions

Complex programs can be constructed using loops, decisions, and subroutines. All of these things become easier to use when labels are available. Subroutines will be introduced in the next lab. In this lab, you will work with loops and decisions.

Loops, of course, are implemented with jump commands, when the computer jumps back to a previous location in the program. Decisions are implemented with conditional jump instructions such as `JMZ` and `JMN`. When the computer executes one of theses instructions, it decides whether or not to jump, based on the current circumstances. When the computer encounters the instruction `JMZ Loc`, it checks the accumulator register. If the number in the accumulator is zero, then the computer jumps to location "Loc." Otherwise, the computer simply proceeds on to the next statement following the JMZ. The `JMN` instruction is similar, except that it checks whether the number in the accumulator is negative. Another conditional jump instruction, `JMF`, tests the value in the Flag register. It is described in one of the exercises at the end of the lab.

There are two sample programs for you to look at that make effective use of loops and decisions. Each of these programs is used in one of the exercises at the end of the lab.

The sample program "ThreeNPlusOne.txt" computes a "3N+1 sequence." (This is a problem that you will see several times in these labs.) Given a positive integer N, the program applies the rule: "If N is even, then replace N by N/2; if N is odd, then replace it by 3N+1." It applies this rule over and over until the number N becomes equal to 1. For example, if the starting value of N is 7, then the program generates the sequence of values: 7, 22, 11, 34, 17, 52, 26, 13, and so on.

Run the "ThreeNPlusOne.txt" program. To see the numbers as they are generated, watch memory location number 17, which contains the successive values of N starting with 7. You will probably want to bump the speed up to "Fast." You can run the program again with any other starting value of N. Modify the value of N in the assembly language program, and then reload it into memory with the "Translate" button. (You could also change the number in location 17 directly and then run the program. If you do this, don't forget to click the "Set PC=0" button to

reset the Program Counter to zero.)

You should also read the program and note how it uses the five labels, "NextN," "Odd," "Even," "Done," and "N." (The label "Odd" is defined but is never referred to in the program. It is really just there for human readers.)

Another sample program, "MultiplyByAdding.txt" adds two numbers by adding one of the numbers to itself over and over. The program is set up to multiply 13 by 7. Read the program and try it out. Use it to multiply some other pairs of numbers.

Make sure that you understand these programs and the general ideas of loops and decisions. You will need to understand them to complete the exercises at the end of the lab.

Indirect Addressing

The next sample program is "ListSum.txt." This program illustrates a common type of processing where the computer processes a sequence of consecutive locations in memory. In "ListSum.txt", the computer adds up the numbers stored in consecutive locations. It stops when it gets to a location that contains a zero. In the exercises, you will write two programs based on this one. Before doing those exercises, you should read the sample program, run it, and try to understand how it works.

The "ListSum.txt" program uses indirect addressing to access numbers in the list. Indirect addressing is used in several assembly language instructions, including `LOD-I`, `STO-I`, `ADD-I`, and `SUB-I`. Recall that `LOD-C XXX` tells the computer to load the number `XXX` into the accumulator. `LOD XXX`, on the other hand, tells the computer to copy the contents of memory location `XXX` into the accumulator. This is known as direct addressing. In the `LOD-I XXX` instruction, `XXX` is the address of a memory location, but not of the memory location containing the data. Instead, memory location `XXX` contains another address, and that address specifies the location whose contents are to be loaded into the accumulator. For example, if location 17 contains the number 42, then `LOD-I 17` will load the contents of memory location 42 into the accumulator.

Admittedly, this is confusing, but it turns out to be just what we need to do list processing. In fact, in that context, its not all that confusing after all. Consider the following program outline:

```
                .
                .   ; Set up
                .

               LOD ListStart ; Load Loc with the starting
                             ;          location  of the list.
               STO Loc

Loop:          LOD-I Loc

                .
                .   ; Process the number.
                .
```

```
                LOD Loc        ; Add 1 to Loc.
                INC
                STO Loc

                JMP Loop       ; Return to the start of the loop.

Loc:            data           ; Location of next number
                               ;                 to be processed.

ListStart:      183            ; List of data to be processed.
                72
                902
                164
                 .
                 .
                 .
```

The first time `LOD-I Loc` is executed, it loads 183, the first number in the list, into the accumulator, since the value of Loc is ListStart. After processing the 183, the program adds 1 to the number in Loc and jumps back to the start of the loop. The value stored in Loc is now the address of the second number in the list, 72. So now when `LOD-I Loc` is executed, it loads 72 into the accumulator. The next time through the loop, it will load 902, then 164, and so on. So, each time through the loop a different location is processed, even though the instructions in the loop don't change.

"Loc" is said to be a pointer since the value stored in Loc is not really the number we are interested in. Instead, the value in Loc indicates where to go to find the number we want. It "points" to that value. Pointers and indirect addressing are used in various ways, even in high-level languages. In the next lab, you'll see how useful indirect addressing can be in a jump instruction.

Dancing Bits

There is another sample program that I've provided mostly for your amusement, but also because watching it run might just give you a better appreciation of what a computer is really doing as it computes. Select the sample program "PowersOfThree.txt," from the pop-up menu in the xComputer Window. Read the comments at the beginning of the file, and then load the program into xComputer's memory by clicking on the "Translate" button. Set the display style to "Graphics" and the run speed to "Fastest," and run the program. You will see the bits in xComputer's memory dance as a non-trivial computation is performed.

Exercises

Exercise 1: The sample program "SimpleCounter.txt" is a copy of the program that counts forever, which was discussed earlier in this lab. Modify this program so that it counts to 16 and

then halts. (This was also an exercise in the previous lab, except that in that lab, you didn't use labels.) Add a HLT statement after the JMP statement. Use the name "Done" as a name for that statement. Jump to Done when the Count reaches 16.

Exercise 2: Here is a copy of the "CountAndStore" program that was used in the previous lab. (A copy was also loaded as one of the sample programs by the xComputer applet on this page.) Rewrite this program to use labels, instead of numerical addresses, for locations 2 and 4. Do you think the program is easier to understand with labels?

```
lod-c 1     ; Start with a 1 in location 12
sto 12

lod 12      ; This instruction is stored in location 2
inc
sto 13      ; This instruction is stored in location 4

lod 2       ; Add 1 to the number in location 2
inc
sto 2

lod 4       ; Add 1 to the number in location 4
inc
sto 4

jmp 2       ; Go back to the instruction in location 2
```

Exercise 3: Modify the "ThreeNPlusOne.txt" program so that it counts the number of steps it takes for N to become equal to 1. To do this, add another labeled location at the end of the program. Call it "Count". Change the program so that it starts by storing a zero in location Count. Each time through the loop, it should add 1 to Count. When the program ends, the value in Count is the number of times the program has gone through the loop. This is also the number of steps that were computed in the sequence. How many steps are there in the sequence that begins with N=7? How about N=27? Be sure to add comments to your program to explain how Count is being used.

Exercise 4: The assembly language instruction SHR shifts the number in the Accumulator one bit to the right. That is, each bit in the binary number is moved over one bit-position to the right. The leftmost bit-position, which would be left empty, is filled in with a zero. The rightmost bit, which has no other place to go, is placed into the Flag register. A JMF instruction can be used to test the contents of the Flag register. Suppose XXX is a label. If the Flag register contains a one, then JMF XXX causes a jump to location XXX. If the Flag register contains a zero, then JMF XXX has no effect.

Write a program that counts the number of 1's in a binary number. When the program begins, the number should be stored in a memory location labeled Num. There should also be a memory location named Count for counting the number of 1's in Num. The program begins by storing a zero in Count. It then goes into a loop that shifts Num one bit to the right. If the number that

"falls off the end" into the Flag register is a 1, then the program should add 1 to `Count`. The loop continues until the value of `Num` becomes zero. At that point, `Count` contains the number of 1's that were in the original number.

Exercise 5: Just as it is possible to multiply by adding over and over, it is possible to divide by subtracting over and over. Suppose you want to know how many times N1 goes into N2. Start with N2 and subtract N1 repeatedly until the answer is less than N1. The number of subtractions you performed is the number of times that N1 goes into N2. For example, you can compute that 4 goes into 14 three times by computing 14 - 4 - 4 - 4 = 2 . (The number 2 that you end up with here is the remainder when 14 is divided by 4.)

Write a program to compute how many times a number N1 goes into another number N2. Your program will be somewhat similar to the sample program "MultiplyByAdding.txt." You still need a loop, and you still need to count how many times that loop is executed. However, the set-up before the loop, the action performed in the loop, and the test for ending the loop are different. Note that to test whether A < B, you can subtract B from A and test whether the result is negative. In the language of xComputer, you can use a `JMN` instruction to test whether a number is negative.

Exercise 6: The sample program "ListSum.txt" adds up a list of numbers. Modify this program so that instead of computing the sum of the numbers in the list, it will find the largest number in the list. Change the name of the memory location "Sum" to "Max". As the program runs, this memory location will hold the largest number seen so far in the list. When the program ends and the whole list has been examined, Max will hold the largest number in the entire list. (You can compare two numbers by subtracting one from the other, and using a `JMN` instruction to test whether the answer is negative.)

Exercise 7: Write another list-processing program that makes a copy of a list. You can model your program on "ListSum.txt." However, instead of adding up the numbers in the list, it should copy them to another part of memory. Use a label named "Copy" to indicate the location in memory where the copy of the list should begin.

Exercise 8: The "CountAndStore" program in Exercise 1 is a self-modifying program. That is, the machine language instructions in locations 2 and 4 change as the program is executed. Another way to write the program would be to use indirect addressing. Use a memory location labeled "Loc" to keep track of which location the program is currently working on. Use `LOD-I Loc` and `STO-I Loc` to load and store values in that memory location. To move on to the next consecutive memory location, add one to `Loc`.

Exercise 9: Write an essay explaining in detail how indirect addressing is used in the program you wrote for one of the exercises above (Exercise 5, Exercise 6, or Exercise 7).

Exercise 10: Write an essay explaining how labels are used in assembly language programming and why they are so important. Give examples of things that can be done with labels that would be much harder to do without them.

---

course.

--David Eck (eck@hws.edu), Summer 1998

Labs for The Most Complex Machine

xComputerLab 3: Subroutines

A SUBROUTINE IS A SET OF INSTRUCTIONS for performing some task, chunked together and thought of as a unit. Like loops and decisions, subroutines are useful in the construction of complex programs. The machine language for xComputer does not provide direct support for subroutines. But then again, it doesn't really provide direct support for loops and decisions, which must be implemented by the programmer with jump and conditional jump instructions. Similarly, it is possible to implement subroutines using jump instructions. They are not as easy, as neat, or as safe as subroutines in a high-level language, but they can still be a useful tool. Furthermore, by working with subroutines on xComputer, you'll get to see some of the details of how subroutines can be implemented on a very low level. (You should understand, though, that the machine languages of real computers do provide more support for subroutines than what is covered here.)

Before doing this lab, you should be very familiar with the xComputer applet and with assembly language programming for xComputer, as covered in xComputerLab 1 and xComputer Lab 2. The material in this lab is not covered in The Most Complex Machine.

This lab includes the following sections:

Start by clicking this button to launch the xComputer applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

There and Back Again

The idea of subroutines is simple enough. A subroutine is just a sequence of instructions that performs some specific task. Whenever a program needs to perform that task, it can call the subroutine to do so. The subroutine only has to be written once, and once it is written, you can forget about the details of how it works. If the same task needs to be performed in another program, then it can simply be copied from one program to another using cut-and-paste . So the work done on writing the subroutine doesn't have to be repeated over and over. The subroutine can be reused. In fact, real computers have large libraries of subroutines that are available for use by programs. The complex programs that are used on modern computers would be

extremely difficult to write, if these libraries of pre-written subroutines were not available.

In xComputer's assembly language, "calling" a subroutine means jumping to the first instruction in the subroutine, using a JMP instruction. The execution of the subroutine will end with a jump back to the same point in the program from which the subroutine was called, so that the program can pick up where it left off before calling the subroutine. This is known as returning from the subroutine. (Other computers provide special commands for calling and returning from subroutines.)

There is more to it than a few jump instructions, though. For one thing, if the subroutine is to be reusable in a meaningful sense, it must be possible to call the subroutine from many different places in a program. If this is the case, how does the computer know what point in the program to return to when the subroutine ends? The answer is that the return point has to be recorded somewhere before the subroutine is called. The address in memory to which the computer is supposed to return after the subroutine ends is called the return address. Before jumping to the start of the subroutine, the program must store the return address in a place where the subroutine can find it. When the subroutine has finished performing its assigned task, it ends with a jump back to the return address. If, for example, the return address has been stored in a memory location labeled "`RetAddr`", then the subroutine can finish with the statement:

```
JMP-I  RetAddr
```

In the language of xComputer, `JMP-I` is an indirect jump instruction, which uses indirect addressing. It tells the computer to jump to the location whose address is stored in `RetAddr`. In this case, that will be the return address for the subroutine.

All of the subroutines that you will work with in this lab use return addresses in the same way. A memory location in the subroutine is reserved for holding the return address. Before jumping to the beginning of the subroutine, the program will save the appropriate address in that memory location. The subroutine ends with an indirect jump instruction to the return address.

As a first example, look at the sample program "MultiplyBySeven.txt" This program uses a very simple subroutine whose task is to multiply a number by seven. The subroutine, which is named `Times7`, is at the end of the program. It begins with the lines:

```
RetAddr:  data       ; The return address for the subroutine
                     ;     must be stored in this location
                     ;     before thesubroutine is called.

Times7:   STO num_t  ; STARTING POINT OF SUBROUTINE.
```

The first line reserves a memory location to hold the return address. The "main program," which uses the subroutine, stores the return address in this memory location. It then jumps to the location "Times7", which is where the instructions for the subroutine begin. The last instruction in the subroutine is "`JMP-I RetAddr`" which returns control back to the main program.

The return address is not the only item of information that the program has to send to the subroutine. The task of the subroutine is to multiply a number by seven. The main program has to tell the subroutine **which** number to multiply by seven. This information is said to be a parameter of the subroutine. Similarly, the subroutine has to get its answer -- the result of multiplying the parameter value by seven -- back to the main program. This answer is called the

return value of the subroutine. In "MultiplyBySeven.txt," the program puts the parameter value in the accumulator before calling the subroutine. The subroutine knows to look for it there. Before it jumps back to the main program, the subroutine puts its return value in the accumulator. The main program knows to look for it there. Passing parameter values and return values back and forth in a register, such as the accumulator, is a very simple and efficient method of communication between a subroutine and the rest of a program. In the next section, we'll look at other methods of communication.

You should read the "MultiplyBySeven.txt" sample program and be sure you understand it. Load it into memory with the "Translate" button and execute it. You might want to execute it by hand with the "Cycle" button so that you can follow in detail how it works. Modify the program so that it computes 103*7 instead of 34*7. (Would you have thought of multiplying a number by seven using the method in this subroutine? Of course, a major point about subroutines is that when you are using a subroutine that someone else has written for you, you don't really care so much **how** it performs its task, as long as it does it correctly. You use the subroutine as a "black box.")

Parameters and Local Names

It is not always possible to pass parameter values in registers. In xComputer, for example, there is only register (the accumulator) that can be used for parameter-passing. But some subroutines require two or more parameters. The solution is to use reserved memory locations for the parameter values, just as is done for the subroutine's return address. Similarly, a return value from a subroutine can be placed in a reserved memory location rather than in a register. This method is a little more difficult than using registers, but it is also more flexible.

Look at the sample program "MultiplyTwoNumbers.txt." This sample program includes a subroutine that can multiply any two numbers. The numbers that are to be multiplied are parameters to the subroutine, and the product of the two numbers is its return value. The memory locations labeled `N1`, `N2`, and `Answer` are used to hold the two parameter values and the return value. These locations can be found at the beginning of the subroutine, along with a memory location to hold the return address. Before it calls the subroutine, the main program must load the two numbers that it wants to multiply into `N1` and `N2`. When the subroutine ends, the main program can get the answer by loading the contents of memory location `Answer` You should read the program, try it out, and make sure that you understand all this. The program contains a list of detailed instructions for using the subroutine. (Note that you don't have to understand the method that the subroutine uses for multiplying the numbers. In fact, it's a fairly complex procedure.)

By the way, when you read the "Multiply" subroutine, you'll notice that it uses nine different labeled memory locations. Five of these -- `ret_addr`, `N1`, `N2`, `Answer`, and `Multiply` -- are used for communication with the main program. The other four are part of the internal working of the subroutine. Ideally, the main program wouldn't have to know about them at all, because the main program is only interested in the task performed by the subroutine, not in its internal workings. These labels are called local names, since they are meant to be used only "locally" inside the subroutine. Unfortunately, in the simple assembly language of xComputer, it is not possible to actually "hide" these names from the main program, and you have to be careful

not to use the same name for a different purpose in the main program. In my sample subroutines, I have tried to use local names that are not likely to occur elsewhere in the program, such as `loop_m` and `done_m`. In real programming languages, local names are actually invisible to the rest of the program, so there is no possibility of a conflict.

Passing Pointers

The final sample program, "ListSumSubroutine.txt," illustrates one more aspect of parameter passing. The subroutine in this example is meant to add up an entire list of numbers. There is no limit placed on the number of items in the list. How is it possible to pass a potentially limitless number of parameters to the subroutine?

The solution is that the numbers in the list are not passed to the subroutine at all! Instead, the main program tells the subroutine **where in memory** to look for the list. There is only one parameter: the address of the starting location of the list. This address is said to be a pointer to the list.

In the "ListSumSubroutine.txt" example, the main program stores a pointer to the list in the memory location labeled "ListStart." The subroutine then accesses the numbers in the list using indirect addressing (in a `LOD-I` instruction). This is a nice example that demonstrates once again the usefulness of pointers and indirect addressing.

Reality Check

It's actually kind of crazy to try to write subroutines for xComputer. The limited variety of machine language instructions for xComputer makes it very hard to express the idea of a subroutine in that language. Not surprisingly, real computers have special-purpose machine language instructions for working with subroutines.

The first thing that a machine language needs is a pair of instructions for calling a subroutine and for returning from a subroutine. These instructions might be called jump-to-subroutine and return-from-subroutine. The jump-to-subroutine instruction would automatically save a return address and then jump to the starting point of the subroutine. The computer could figure out the return address on its own -- instead of leaving it up to the programmer -- by looking at the value in the Program Counter register. (The Program Counter holds the address of the **next** instruction after the one that is currently being executed, and that's exactly the point that the subroutine should jump back to.) The return-from-subroutine instruction would get the return address that was previously saved by jump-to-subroutine and jump back to that address. These two instructions would make it unnecessary for a programmer to even think about return addresses.

Real computers also have a more systematic way of dealing with parameters. An area of memory called the stack is used to hold the parameters for **all** subroutines. In fact, the stack also holds return addresses and data values used internally by subroutines. The stack is just a list of values. When a subroutine is called, the parameters and return address for the subroutine are added to the end of the list. When the subroutine ends, the return address and parameters are removed from the stack. The jump-to-subroutine instruction stores the return address on the stack, and return-from-subroutine removes it from the stack when it's time for the subroutine to

end. Typically, a computer has a register called the Stack Pointer to keep track of how big the stack currently is. And machine language typically includes instructions called push and pop to add items from the stack and to remove items from the end of the stack.

When one subroutine calls another, all the data for the second subroutine is simply added to the stack, on "top" of the data that is already there. When the second subroutine ends, its data is removed from the stack, but the data for the first subroutine is still there so that the first subroutine can simply pick up where it left off. The whole system is really rather elegant.

Maybe it's not so crazy to write a few subroutines for xComputer after all, since doing everything by hand can help you understand what really goes on when a subroutine is called. And it can also help you appreciate the elegance of more sophisticated computers and programming languages.

Exercises

Exercise 1: The main program in the "MultiplyTwoNumbers.txt" sample program is as follows:

```
        lod-c 13          ; Set up to call the subroutine with
        sto N1            ;   N1 = 13, N2 = 56, and ret_addr = back.
        lod-c 56
        sto N2
        lod-c back
        sto ret_addr
        jmp Multiply      ; Call the subroutine.

  back: lod Answer        ; When the subroutine ends, it returns
                          ;    control to this location, and the
                          ;    product of N1 and N2 is in Answer.
                          ;    This LOD instruction puts the answer
                          ;    in the accumulator.

        hlt               ; Terminate the program
                          ;                  by halting the computer.
```

Carefully explain each instruction in this program. Explain exactly what each of the first 8 instructions has to do with calling the subroutine.

Exercise 2: Write a main program that uses the `Multiply` subroutine in "MultiplyTwoNumbers.txt" to compute the product 5 * 23 * 17. Do not modify the subroutine. The main program should call the subroutine **twice**.

Exercise 3: Write a subroutine to add three numbers. (This is a pretty silly thing to do, but the point is to demonstrate that you understand the basic concepts involved.) Your subroutine should have three parameters and a return value (the three numbers to be added and their sum). Write a main program that uses your subroutine to add 17, 42, and 105. Your program will be very similar to the sample program "MultiplyTwoNumbers.txt." Make sure to include comments in your program!

Exercise 4: Read the Reality Check section above. Why can't you express the jump-to-subroutine and return-from-subroutine operations in the language of xComputer? What do these instructions need to do that can't be expressed in that language? What sort of modifications would have to be made to xComputer to add them to xComputer's machine language?

Exercise 5: The sample program "ListSumSubroutine.txt" uses a subroutine to add up a list of numbers. Suppose you would like to multiply the numbers instead. To do this, copy the multiplication subroutine from the "MultiplyTwoNumbers.txt" sample program, and paste it onto the end of "ListSumSubroutine.txt." Modify the `ListSum` subroutine so that instead of adding two numbers, it uses the `Multiply` subroutine to multiply. You have to replace the instruction "`add Sum`" with several instructions that set up a call to the `Multiply` subroutine. And you'll need to make up a return address label for the subroutine to jump back to. Once you've made this modification, the program will compute the product 1*2*3*4*5*6*7 instead of the sum 1+2+3+4+5+6+7. (You might want to change the name of the subroutine from `ListSum` to `ListMul`, and change the name of the memory location `Sum` to `Product`. This will require that you also change the names in the main program.)

Exercise 6: The sample file "PrimesAndRemainders.txt" defines two subroutines. One of the subroutines can be used to find the remainder when one integer is divided into another. The other subroutine can be used to determine whether a number is prime. The file does not contain a main program. If you want to use one or both of the subroutines, you can add a main program at the beginning of the file.

You should read the comments on the two subroutines to find out how to use them. Then write two programs that use the subroutines. The first program should use the "Remainder" subroutine to compute the remainder when 609 is divided by 81. The second program should use the "PrimeTest" subroutine to determine whether or not 51 is prime. Note that you do **not** have to understand how the subroutines work. You just need to know how to call them and pass the proper parameters to them.

Exercise 7: This exercise, like the previous one, involves writing a main program for the "PrimesAndRemainders.txt" example. However, this exercise is much more challenging. Write a main program that makes a list of prime numbers. The program should use the "PrimeTest" subroutine to test whether each of the numbers 2, 3, 4, 5, 6, and so on, is prime. Each number that is found to be prime should be added to a list. You can write a program that runs in an infinite loop.

Use a location named "p" to store the number that you are checking. Start by storing a 2 in p. In a loop, you should call PrimeTest to see whether p is prime. If p is prime, then add it to the list. In any case, you should then add 1 to p and jump back to the start of the loop to test the next value of p.

Making a list of primes means storing primes in consecutive memory locations. Use a location named "`list`" to point to the end of the list. That is, the value of `list` is the location where you want to store the next prime that you find. Let's say that you want the list of primes to begin at location 50. At the beginning of the program, you should store a 50 in `list`. When you find a prime number, you can add it to the end of the list with a `STO-I list` command. Then you should add 1 to the value of list to get ready for the next number.

If you run your program at high speed, you can watch it store the numbers 2, 3, 5, 7, 11, 13, and so on in memory. You might want to watch the program in graphics mode so that you can watch the activity in the main program and in the two subroutines.

Exercise 8: Write a short essay discussing how subroutines can make it easier to design and write complex programs. (Your answer should show that you understand that they can do more than save you typing!) In your essay, use some of the work you did in this lab for examples.

--David Eck (eck@hws.edu), Summer 1998

Labs for The Most Complex Machine

xTuringMachine Lab: Introduction to Turing Machines

TURING MACHINES are extremely simple calculating devices. A Turning machine remembers only one number, called its state. It moves back and forth along an infinite tape, scanning and writing symbols and changing its state. Its action at a given step in the calculation is based on only two factors: its current state number and the symbol that it is currently scanning on the tape. It continues in this way until it enters a special state called the halt state. In spite of their simplicity, Turing machines can perform any calculation that can be performed by any computer. In fact, certain individual Turing machines, called universal Turing machines, can actually execute arbitrary programs, just as a computer can. You won't see any universal Turing machines in this lab, but you will experiment with Turing machines that can perform non-trivial calculations.

Turing machines are covered in Chapter 4 of The Most Complex Machine. Although the lab is mostly self-contained, it would be useful for you to have some familiarity with Turing machines before beginning the lab. Especially important is the idea that a Turing machine is described by a table of rules that specify what action the machine will take for each combination of state and scanned symbol. The action takes the form of writing a new (or the same) symbol to the current square, moving either left or right on the tape, and entering a new (or the same) state.

This lab includes the following sections:

In this lab, you will work with an applet called xTuringMachine. Start by clicking this button to launch the applet in its own window:

(Sorry, your browser doesn't do Java!)

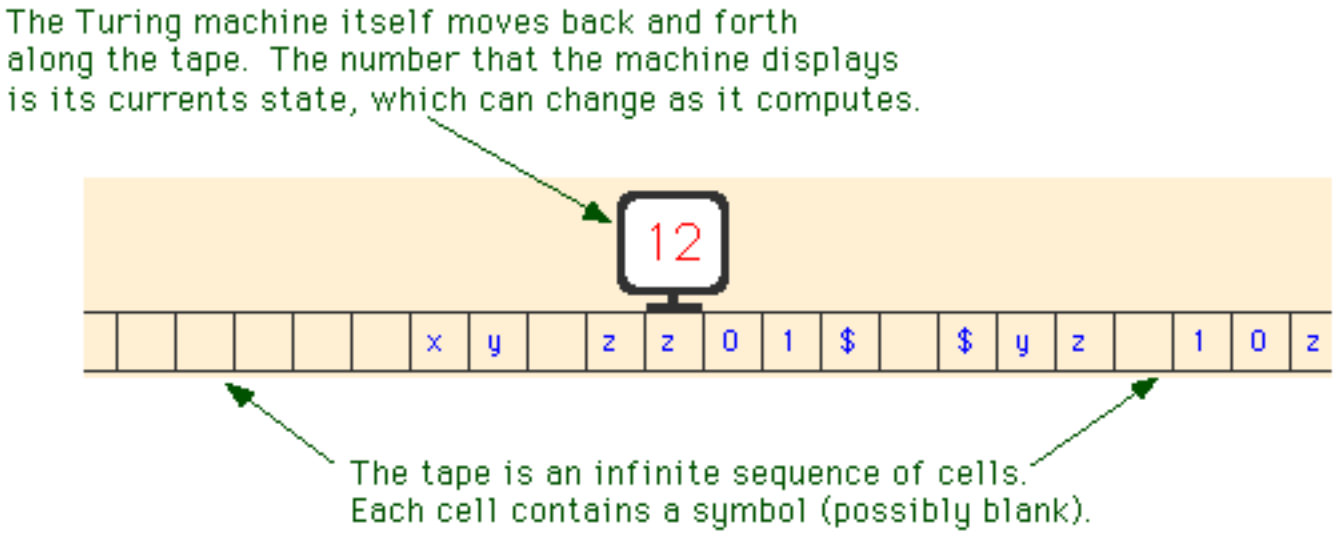
(For a full list of labs and applets, see the index page.)

Using the Applet

The xTuringMachine applet can simulate Turing machines with up to twenty-five states. The states are numbered from 0 to 24. There is also the special halt state, which is denoted by "h". The Turing machines in this applet are restricted to using the following symbols: the letters x, y, and z; the binary digits 0 and 1; the dollar sign, $; and the blank space (often written as #). We often think of the machines as working with "binary numbers" made up of 0's and 1's or with "words" made up of x's, y's, and z's. But remember that the **meaning** of a symbol has no effect on any calculation performed by a Turing machine; all the machine does is follow its rules.

At the very top of the xTuringMachine applet is a pop-up menu that you can use to select from among the machines that the applet knows about. The applet is set up to load several sample machines when it starts up. (Later, you'll see how to construct new machines from scratch.) You'll work with the first of these sample machines, "Change01toXY," to help you learn how to use the applet.

Just below the pop-up menu is the Turing machine itself and its tape:



Below the machine, on the left, is a set of controls. Use the "Run" and "Step"buttons to control the computation of the machine. If you click on the "Step" button, the Turing machine will perform one step in its computation. If you click on "Run," the machine will compute until you stop it or until it enters the halt state. You can control the speed of a running machine with the Speed pop-up menu, which is just above the run button. (You might want to stick with the "Step" button at first, so that you can follow each step of the computation in detail.)

To do one step in its computation, the Turing machine considers the **state that it is in** and the **symbol that it is reading** in the cell where it is located. Based on this information, it will (1) **write** a new symbol in the current cell; (2) **move** one cell to the left or to the right; and (3) change to a **new state**. (Note, however, that the "new symbol" that the machine writes can actually be the same as the old symbol, and that the "new state" can be the same as the old state.) The machine bases its action on the table of rules that is shown in the lower

right part of the applet.

For example, look at the table of rules for the sample machine, "Change01toXY." The first row of the table says "If the machine is in state 0 and if the symbol in the current square is # (that is, a blank), then the machine will write a # in the square, move one square to the right, and change to state 0." All the rules for a Turing machine are of this general form. Note that in this case, the symbol it writes in the square is the same as the symbol that was already there; this is just a fancy way of saying that it doesn't change the contents of the square. Similarly when the machine "changes to state 0" in this case, it doesn't really change its state; its new state is actually same state that it was already in.

Step through the computation of the "Change01toXY" machine until it enters its halt state. The machine moves along the tape changing any 0 it finds to an x and changing any 1 to a y. What makes it halt? What would happen if there were no $ on the tape? And, by the way, what happens when the machine encounters the edge of the applet window? You should also try running the machine with the "Run" button.

Note that when a Turing machine halts, it displays an "h" as its current state, and the "Step" button changes to "Reset." Clicking "Reset" will reset the state to zero, so the machine will be ready to start a new computation. By convention, a Turing machine always begins its computation in state zero.

Before you go on to the rest of the lab, there are a few more things you should know about. First, you can use the mouse to drag the Turing machine to a new position on its tape. You can also drag the tape. If you want to drag the tape and the machine together, use the right mouse button instead of the left button, or hold down the Control key as you begin to drag.

Second, and more important, you should know how to change the state of the machine and the contents of its tape. You can click on the Turing machine to hilite it. You'll see a bright blue-green outline, and the blue rectangle below the machine will display a "palette" showing the possible states of the machine. To change the state of the machine, you can either type the state or use the mouse to click on the state in the palette. Editing the tape is similar. Click on any cell to hilite it. The palette displays the symbols that the cell can contain. You can type a symbol or click on it in the palette. When you do this, the hilite will move to the next cell on the tape. This makes it easy to type a string of symbols onto the tape.

The "input palette" for symbols.

The "input palette" for states.

Try making a new input tape for the "Change01toXY" machine. Move the machine to the beginning of the input. Make sure that the machine is in state 0. Then run the machine on

your new input.

A More Interesting Machine

As another example, select the sample machine "FindDoubleX" from the pop-up menu at the top of the applet. The purpose of this machine is to move to the right along its tape, until it finds two x's in a row; it then halts on the leftmost of those two x's. The machine you looked at in the previous section had only a single numbered state, state 0. The "FindDoubleX" machine has two states, number 0 and number 1. As this machine runs, you will see it changing between these two states. Try it! Use the "Step" button to step through the computation.

Although its states are completely meaningless to the machine, from our human point of view we can assign a kind of meaning to each state. In state 0, this machine is "moving to the right searching for an x." In state 1, it "has found one x and needs to check the next square to see whether there is another x there." In state 1, after checking the next square, it halts if it finds an x there and returns to state 0 if not.

One might say that the state number **counts** the number of x's in a row that the machine has encountered. In state 0 it has encountered zero x's in a row; in state 1, it has encountered one x in a row. You will need to understand this in order to do one of the exercises at the end of the lab. You will also need to know about editing the rule table. This is covered in the next section of the lab.

The complete table of rules for the "FindDoubleX" machine looks like this:

In State	Reading	Write	Move	New State
0	x	x	R	1
0	other	same	R	0
1	x	x	L	h
1	other	same	R	0

The entries other under "Reading" and same under "Write" need some explanation. The word "other" is used here to indicate a default rule. This rule is used when the machine is in the state specified in the "In State" column, and no other rule applies. For example, suppose that the "FindDoubleX" machine is in state 0. If it happens to be reading an x, it will follow the first rule in the table, which tells it to write an x, move right, and change to state 1. However, if it is in state 0 and reads any **other** symbol, then it will apply the second rule in the table. The word "same" under the "Write" column in that rule tells the machine to write the same character that it read. Without this default rule, the machine would need six separate rules to tell it what to do when it is in state 0 and it reads one of the symbols y, z, 0, 1, $, or blank.

For an example of a more complex word-processing Turing machine, you can try out the

sample machine called "CopyXYZ." This machine will make a copy of a string of x's, y's, and z's. Try it out!

Making New Machines

In this part of the lab, you will learn how to construct new machines in the xTuringMachine applet. To begin a new machine, select "[New]" from the pop-up menu at the top of the applet. This will give you an empty rule table that you can fill in to define the machine you want.

The xTuringMachine applet does not allow you to simply type in a rule. Instead, it has procedures for adding a new rule to the table and for modifying rules that are already in the table. The type of editing that you can do is similar to what you already know about setting the Turing machine's state and changing the contents of its tape.

New rules are added to the rule table using the "Rule Maker" that is located just above the table of rules. The Rule Maker has a set of five boxes where you create the rule and a "Make Rule" button that you can click to add the rule to the table:

You can edit any of the five items in the Rule Maker. Just click on the item that you want to change. The item will be hilited. In the above picture, the second item, "other," is hilited. The blue rectangular palette will display the values that you can legally put in the hilited spot. You can either type the value you want or click on it in the palette. (Note that "other" is represented in the palette by a "*". If you want to enter the value "other," you have to type * or click on it.) Once you've set up the rule you want in the Rule Maker, you can either click the "Make Rule" button or press the Return key to add it to the table of rules. The rule has no effect on the Turing machine until you add it to the table.

A newly added rule will be displayed in the table in red. The rule shown in red is selected. You can delete the selected rule from the table by clicking on the "Delete Rule" button. You can select any rule in the table by clicking on the rule.

Once a rule has been added to the table, you can edit the last three columns in the rule. Click on the item you want to change, and edit it in the usual way. Note that the last three columns of the table specify the action that the Turing machine will take when it is in the specified state and reading the specified symbol. You are only allowed to change the action part of the rule, once it is in the table. Often, the easiest way to create a table of rules is to quickly create a bunch of rules without worrying about the action specified in each rule. You can then edit the action parts of all the rules in the table.

There are lots of things in the xTuringMachine applet that you can edit. You can use the arrow keys and the tab key to move among the various editable items. Often, this is quicker than using the mouse.

You will notice that sometimes the "Make Rule" button changes into a "Replace" button. This will happen whenever the first two items in the Rule Maker match the first two items in an existing rule. If you click on the "Replace" button, the rule in the Rule Maker will replace the rule in the table.

Before you do the exercises at the end of the lab, you should get some practice at creating and editing a table of rules. Here is a table of rules for a Turing machine that "Nudges" a string of x's, y's and z's one square to the left. The machine must be started on the leftmost symbol in the string, and it will only work if there are a couple of blank squares surrounding the string:

In State	Reading	Write	Move	New State
0	#	#	L	5
0	x	#	L	1
0	y	#	L	2
0	z	#	L	3
1	#	x	R	4
2	#	y	R	4
3	#	z	R	4
4	#	#	R	0
5	#	#	L	6
6	#	#	R	h
6	other	same	L	6

You should make a copy of this machine by adding each of the above rules to the applet's rule table. Begin by selecting "[New]" from the pop-up menu at the top of the applet, if you haven't done so already. You should also type an input string of x's, y's, and z's onto the Turing machine's tape, and move the machine to the leftmost symbol in the input. Then you can start making the rules, one-by-one, and adding them to the table. When you are all done, you should have a machine that will perform as advertized.

One more feature of the applet deserves to be mentioned here: Suppose that you click the "Step" or "Run" button, and the Turing machine finds itself in a situation that is not covered by any rule in the rule table. In this case, the machine will stop and will display the message "No Rule Defined!" It will also set up the Rule Maker with the its current state and the symbol that it is reading, so that it is all set for defining the missing rule. It's possible to define a machine using this feature. Start with an empty table of rules. Click "Step." The machine will protest. You can define the rule, and click "Step" again. You can proceed in this way until the whole rule table has been defined. However, you have to be careful to make sure that you have in fact covered all the situations that might arise.

Binary Arithmetic

The operations of incrementing (adding one to) or decrementing (subtracting one from) a binary number are simple enough to be done easily by Turing machines.

The algorithm for adding one can be described as follows: If the digit on the tape is a zero, then simply change that digit to a one. If the digit is a one, change it to a zero, move left, and apply the same procedure at that position. (This is like "carrying" a one to the next column.) Finally, to add one to a blank space, simply change that blank to a one. (This can occur when a one is carried beyond the leftmost digit of the number; the blank should be treated just like a zero.) For example, 110_2 + 1_2 = 111_2, 1011_2 + 1_2 = 1100_2, and 11_2 + 1_2 = 100_2.

The sample Turing machine "Increment" is a simple Turing machine for incrementing a binary number (This same machine can be found in Figure 4.2 of The Most Complex Machine.) The "Increment" machine should be started on the rightmost digit of a binary number. It will add one to the number, and then it will halt on the rightmost digit of the answer. Try it out! Every time you click the "Run" button, the machine will add one to the number on its tape.

With somewhat more work, it is possible to make a Turing machine that adds two binary numbers. The sample machine "AddBinaryNumbers" does this in the ordinary way, by adding corresponding digits in the numbers one-by-one, starting at the right and working left. The numbers to be added must be next to each other on the tape, and they must be separated by a single blank space. The machine must be started on the right end of the second number. Try running this machine on the sample input, and then try giving it several other pairs of numbers to add. Note that as the machine adds the second number to the first, it replaces 0's and 1's with x's and y's. When it finishes, it erases the second number from the tape and changes all the x's and y's back to 0's and 1's.

The final sample machine, "MultiplyByAdding," multiplies two binary numbers. It does not do this by the usual multiplication algorithm. Two numbers can be multiplied by repeatedly adding the first number to itself. The second number tells how many times the addition is to be performed. The "MultiplyByAdding" machine works in this way. Each time it adds the first number to itself, it subtracts one from the second number. When the second number reaches zero, the process is finished. At that point, the machine erases the two original input numbers. The number remaining on the tape at the end of this process is the product of the two inputs. You won't have to understand this machine in detail, but it's interesting to see how a relatively complex computation can be performed by a Turing machine.

Exercises

Exercise 1: Create a Turing machine that will move to the right until it finds a $. Then it will erase everything on the tape between that $ and the next $ to the right. It will halt when it gets to the second $. The $'s themselves should not be erased. You can do this with a fairly simple machine that uses only two states, 0 and 1. (Note that if the machine is started

on a tape that does not contain two $'s to the right of the machine's starting position, then the machine will never halt.)

Exercise 2: One of the examples in the lab is a Turing machine called "FindDoubleX." This machine moves to the right until it comes to two x's in a row. Then it halts on the first of the two x's. Create a new machine that will move to the right until it finds **three** x's in a row. It should halt when it finds a group of three consecutive x's. (Ideally, it should move back to the first of the three x's and halt there.)

Exercise 3: This exercise assumes that you have done Exercise 2. Given any sequence of x's, y's, and z's, describe how you could construct a machine that will move to the right until it finds the given sequence. For example, if the sequence is xyzzyx, it should move to the right until it has found the symbols x, y, z, z, y, and x in consecutive squares, and then it should halt. What is the minimum number of states that such a machine would need (assuming that you don't care which square it halts on)? Why?

Exercise 4: Construct a Turing machine to do the following: Assume that the machine is started on a tape that contains nothing but a string of $'s. The machine is started on the left end of this string. The purpose of the machine is to multiply the length of the string by 3. For example, if it is started on a string of seven $'s, it should halt with twenty-one $'s on the tape. If it is started on a string that contains just one $, it should halt with three $'s on the tape. Here is one way that the machine might operate: Change one of the $'s to an x, then go to the end of the string and write two more x's. Go back and process the next $ in the same way. Continue until all the $'s have been processed. Then change all the x's to $'s.

Exercise 5: Construct a Turing machine to do the following: Assume that the machine is started on a tape that contains nothing but a string of $'s. The machine is started on the left end of this string. The purpose of the machine is to divide the length of the string by 3. (Throw away any extra fractional part, so that 17 divided by 3 would be 5). For example, if the machine is started on a string of twenty-one $'s, it should halt with seven $'s on the tape. If it is started with ten $'s on the tape, it should halt with three $'s on the tape. And if it is started with five $'s on the tape, then it should halt with one $ on the tape.

Exercise 6: Modify the sample machine "Increment" so that instead of halting after it adds one to its input, it enters into state 0. (All you have to do is change the "New State" in one rule from h to 0.) With this modification, you have a **counting** machine. It will continue to add one to the number on the tape over and over, as long as you let it. Run the counting machine at "Fastest" speed, and time how long it takes to count from 0 up to 1000000000 (in binary). The number 1000000000 has nine zeros and so is equivalent to 2^9, or 512, in base 10. Based on your answer, compute approximately how long the machine would take to count from 0 to 1000000000000000, that is from 0 to 2^{15}. Explain how you computed your answer and why the method that you used is valid.

Exercise 7: Construct a Turing machine to do the following: Assume that the tape contains a binary number, and that the machine is started on the right-hand end of the number. The machine should write a string of $'s, where the number of $'s is given by the binary number that was initially on the tape. For example, if the number on the tape was 10110, which is 22

in binary, then the machine should halt with a string of twenty-two $'s on the tape. In order to construct this machine, you will have to understand something of how the sample machine "MultiplyByAdding" works. In that machine, the problem was to repeat the addition operation a specified number if times. In this exercise, the problem is to repeat the operation "add a $ to the string" a specified number of times.

Exercise 8: Construct a Turing machine to do the following: Assume that the machine is started on a tape that contains nothing but a sequence of x's and y's. (The machine must work for **any** such sequence.) The machine is started on the left end of this sequence. The purpose of the machine is to separate the x's from the y's. For example, given the input xxyxyyxyxx, it will change the tape to read xxxxxxyyyy. The output string does not have to be in the same place on the tape as the input string, but it should be the only thing on the tape when the machine halts. There are many ways to make such a machine.

Exercise 9: Write a description of the algorithm that is used by the sample machine "AddBinaryNumbers" to add two binary numbers on its tape. Your object is to express an understanding of the process used. A description that is on too high a level -- such as "It adds the numbers digit-by-digit from the right" -- doesn't really explain the process. A very low-level description doesn't provide any understanding of the goals or purpose of the actions taken.

Exercise 10: The Turing machines you worked with in this lab can use only the symbols $, 0, 1, x, y, z, and blank. But this is an arbitrary limitation imposed by the xTuringMachine applet. In fact, a Turing machine could be built to use any finite number of different symbols. But no matter how many symbols a Turing Machine might use, the same computation could be done by a machine that uses only the symbols 0, 1, and blank. Why is this true? (Think about the way data is represented in a real computer.) If this is the case, then, why bother with Turing Machines that use more than the minimum number of symbols?

Exercise 11: I have claimed that Turing machines can do any computation that can be done by any computer. What is your reaction to this claim? Do you believe it? What evidence is there to support this claim? What reasons might someone have for doubting it? Write a short essay discussing your answers to these questions.

Exercise 12: Write a short essay comparing Turing machines as computational devices with the model computer, xComputer, that you worked with in previous labs.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

Web Publishing Lab:
Creating Web Pages with Netscape Composer

THIS LAB IS A BREAK FROM THE USUAL run of Java applets and exercises. You've used Web pages in the previous labs. In this lab, you'll learn something about how Web pages are created, and you will publish a page of your own on the Web.

Note: Parts of this lab are specific to students in CPSC 100, Fall 1997, at Hobart and William Smith Colleges. The entire lab assumes that you are working with Netscape Communicator, version 4.0 (or, presumably, later), which includes a component called Netscape Composer for creating Web pages. The use of HTML in email, as described in the last section of the lab, is possible in Netscape Communicator for Windows but might not be available in versions of Netscape that run on other platforms. Other details will be different on other platforms as well.

The only **exercise** for this lab is to produce a Web page. Your page should include headlines, lists, links, colors, graphics, and at least one table. You might want to create a personal home page with information about yourself. You might make a page with information on some selected topic. Or you might want to be more creative: maybe a work of Web-based art?

The lab includes the following sections:

What is HTML?

The pages that are displayed by a Web browser program such as Netscape are written in a special language called HTML, or HyperText Markup Language. (The word hypertext refers to documents that can contain links to other documents; the use of such links is the most distinctive feature of HTML and of the Web.) An HTML document is a plain text file. It contains the text that you see on the page, along with special commands called tags that tell the browser how to display the text and what else to put on the page besides text. For example, text can be displayed in bold face by enclosing it between the tags and . If you want the words "This is important" to appear in bold face on the page, like this:

This is important

then you would put

```
<b>This is important</b>
```

into your HTML document. Similarly, a link to David Eck's home page, which has a URL of http://math.hws.edu/eck/, could appear in an HTML document as

```
<a href="http://math.hws.edu/eck/">Eck's Home Page</a>
```

and this would appear in a Web browser as the link

Eck's Home Page

You can see the HTML source code for any Web page. Most Web browsers have a command that will display the HTML source of the page that you are currently viewing. In Netscape, you can use the "Page Source" command under the "View" menu. **Try this command now, to see the source for this page!**

HTML started out a few years ago as a relatively simple and easy-to-use language. As time has gone by and as newer versions of HTML have been introduced, it has become much more cluttered and complicated. This has made it possible to produce fancier Web pages, but it has made it more difficult for ordinary people to produce state-of-the-art pages just by typing in a few HTML commands. Fortunately, however, it has become possible for people to produce Web pages without knowing any HTML at all! Programs have been written that allow you to create Web pages in a WYSIWYG ("What You See Is What You Get") environment. These programs are very similar to word-processing programs, and in fact you will find that many word-processors now include HTML authoring capabilities.

Netscape Communicator 4.0 includes an HTML authoring component called "Composer." With Composer, you can create Web pages without ever seeing any HTML codes. They will still be there in the background, of course, and you can use the "View Page Source" command to see them if you want. Composer is not a complete tool for HTML authoring -- it does not give you access to many of HTML's advanced features. However, it is likely to be perfectly adequate for most people's needs, and it is a good place for anyone to start.

Netscape Configuration

You can use Composer to create a Web page, and you can view that page on your own computer. However, in that case, you'll be the only one in the world who can see it. Suppose you want to publish your page, so that it can be seen by any of the tens of millions of uses of the Web? In that case, you have to place your page on a Web server -- a computer that is connected to the Internet and that is running a program that lets it "serve" Web pages to other computers on the Net. If your computer is permanently connected to the Internet, you might be able to run your own personal Web server. Otherwise, you can still hope to find a friendly Web server where you can locate your pages.

For students at Hobart and William Smith Colleges, there are several Web servers where you can publish your Web pages. The official Web pages of the Colleges are on a server called www.hws.edu. If you want to be serious about maintaining a presence on the Web, you should get an account on this server and publish your Web pages there. (See the Computer Services page for information.) You can also set up Web pages in your email account on the campus VAX, hws3.hws.edu. (I am **not** going to have you use this account for this lab -- as I would have liked -- because I can't get Netscape Composer to work properly with the VAX's somewhat out-of-fashion VMS operating system.)

However, for this lab, I have set up a special account for you to use on one of the computers belonging to the Department of Mathematics and Computer Science. The name of the machine is escher.hws.edu. Your account on this machine will be deleted at the end of the Fall term, so if you want to keep your Web pages around, please talk to me about moving them to another machine before the end of the term.

You will have to configure Netscape Composer so that it knows about this account. To do this, choose the "Preferences" command from the "Edit" menu. A dialog box will appear. From the list on the left edge of this dialog box, choose the item "Publish," which is listed under "Composer." (You might have to

click on a small plus sign or triangle next to the word "Composer" in order to see "Publish.") In the Publish dialog box, make sure that the two boxes under "Links and Images" are checked. Then, fill in the following information under "Default Publishing Information", substituting your own username for "username":

```
     Publish to (FTP or HTTP):
ftp://username@escher.hws.edu/export/home/username/www/
     Browse to (HTTP):          http://escher.hws.edu/~username/
```

Then click the OK button. This will allow Netscape Composer to find your account on escher.hws.edu, so that it can "upload" your pages into that account. You will need the password for the account later, when you have a page ready to go.

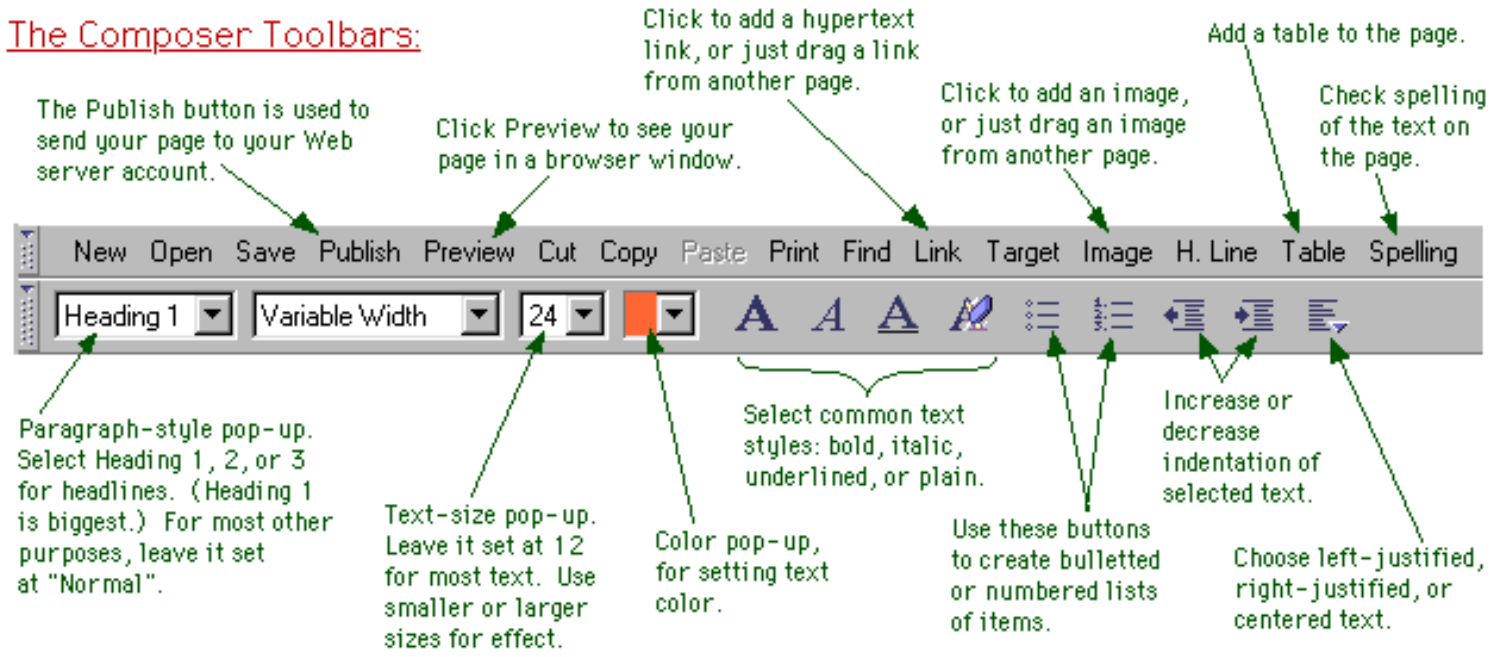
Note: Netscape Communicator includes an on-line help facility that has lots of information about Composer and HTML, as well as about all of Communicator's other features. To access this information, choose "Help Contents" from the "Help" menu. To get information about Composer in particular, select the link to "Creating and Editing Web Pages" in the help window that appears.

Making Your First Page

There are several ways to start up Netscape Composer:

- From Netscape's "File" menu, select the "New" submenu, and then select "Blank Page";
- Click on the small Composer icon in the bottom right corner of a Netscape browser window;
- Select "Page Composer" from the Netscape menu.

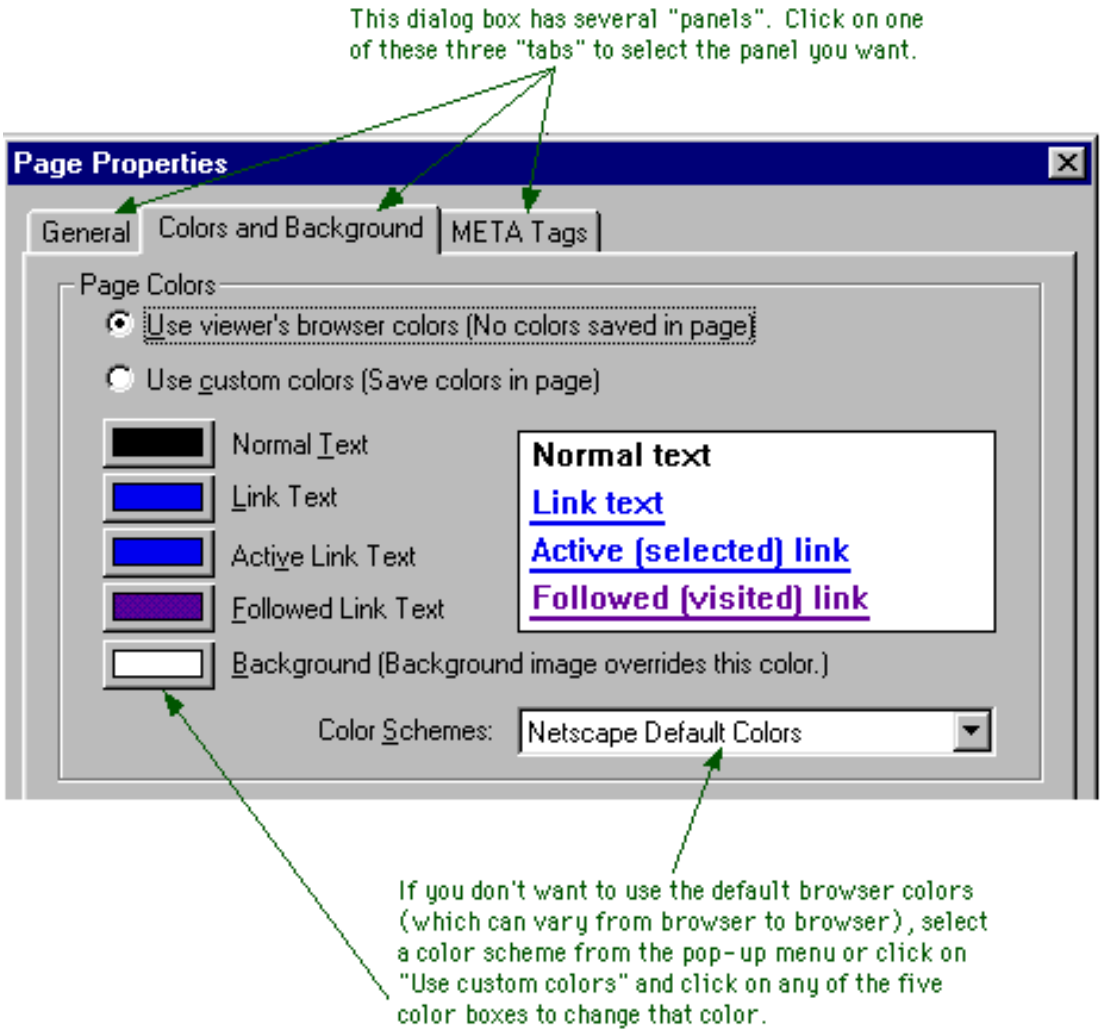
Use any one of these methods to open a window where you can create your page. There will be a large area where you can type your page and several toolbars along the top of the window. One toolbar is used to set the style of the text you type. You can have italic text, big text, colored text, centered text, and so on. The other toolbar is used to add HTML features such as graphics, links, and tables to the page. (The toolbar commands are also available in "Insert" and "Format" menus, in case you prefer using menus.)



I suggest that you start by typing some text for your page. Start with a headline for the page, on a line by itself. Press return a couple of times, and type in a paragraph or two of introductory text. (As with a word-processor, you should press return only at the end of a paragraph; lines within a paragraph will be

wrapped automatically to fit in the window where the page is displayed.) Once you've gotten some text on the page, you can go back and add formatting.

First of all, you should try changing the color of the text and of the page background. To do this, you need to access a "Page Properties" dialog box. Click on the page with the **right** mouse button. A pop-up menu will appear. Choose "Page Properties" from the pop-up menu. (Note that on a Macintosh, you can click-and-hold the mouse button to get the same pop-up menu.) This dialog box has several panels. You should be looking at a panel called "Colors and Background".



Each page has five associated colors: one for the page background, one for regular text, and three different colors that are used to display links. You can select the colors for your page from the "Color Scheme" pop-up menu. (Alternatively, you can set each color individually.)

Now, make some changes to the text. If you have a page heading on the first line of text, hilite it, and then select "Heading 1" from the pop-up menu at the left end of the toolbar. Center the heading by selecting centered text from the pop-up menu at the right end. You might also use the color pop-up menu in the toolbar to change the color of the heading. You can make similar changes to any text on your page.

Dividing lines can help to organize a page. Add one to your page by clicking on the "H. Line" button. Once the line appears on your page, you can drag its edges to change its size, and you can right-click on it to get a pop-up dialog with various options.

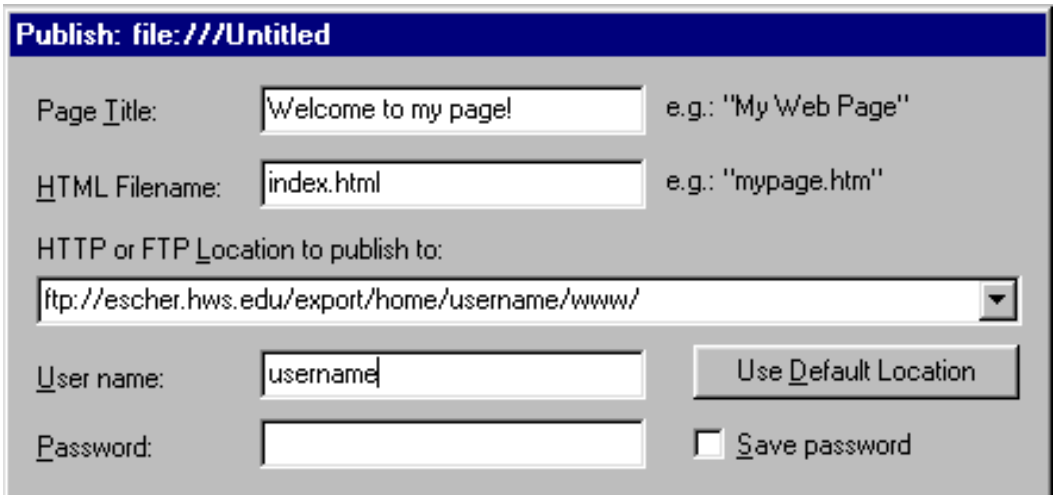
To make a hypertext link on your page, use the "Link" button in the toolbar. You will get a dialog box where you can type in the URL of the page to which you want to link. You can also enter the text of the link that you want to appear on the page. (Alternatively, you could hilite the link text before clicking the

Link button.) Remember that an easy way to get the URL for a page is to go to that page in the Netscape browser and then Cut-and-Paste the URL from the Location box at the top of the browser window. Once the link is created on the page, you can right-click on it to get a pop-up dialog with various options.

You should play with the basic tools mentioned in this section until you get your page into reasonable shape. As a final step, you will probably want to run a spell-checker on your page by clicking on the "Spelling" button in the toolbar. You will then be ready to publish your page.

Publishing Your Page

To publish a page that you have created in Netscape Composer, click on the "Publish" button in the toolbar. This will bring up a dialog box that looks something like this:



The Location and Username should already be filled in with the configuration information that you entered earlier. You can fill in the password for your account, or you can leave it blank. If you leave it blank, Netscape will ask you for it later. If you are working with a newly created page, then the Page Title and HTML Filename boxes will be blank. You should fill in these two boxes. Ordinarily, the HTML Filename for your "home page" should be index.html. The Page Title can be anything you want -- this is the title that will appear in the title bar of a browser window when the page is viewed.

Once you have set up all this information, click the OK button, and Netscape will attempt to send your page to the Web server. If it succeeds, you will be able to view the page in a regular browser window (and so will anyone else on the Internet who knows its URL).

Once your page has been published, you can continue to make changes in the Composer window. For those changes to take effect, you will have to have to upload the page again by clicking on the Publish button.

If you want to return to edit your Web page at a later time, all you have to do is start up Netscape again, load your page into the browser window, and then choose "Edit Page" from the "File" menu. This will open a Composer window that you can use to edit the page and publish the new version. (In fact, you can do this with **any** Web page that you would like to use as a starting point for a page of your own.)

Adding Some Frills

After you get the hang of using Composer, you will want to try out some other HTML features. You will certainly want to add some graphics images to your page. This is very easy to do in Composer, since you can drag images from another page into the page you are creating. I've made a page of sample images that

you can use. You can get images from any page in the same way (although you should, of course, try to be careful about copyright issues). You can also drag images from one place to another on the page you are creating.

When you drag an image into your Web page, it will appear wherever the blinking cursor is currently located. Web browsers treat images much like big characters, and you can have an image in the middle of a paragraph. You can even include an image in a link; someone viewing the page will be able to follow the link by clicking on the image. There are several options for image placement and other image properties. To see a dialog box containing these options, just click on the image with the right mouse button.

When you publish your page, the images on the page will be uploaded to the server along with the page itself. Each image is stored in its own individual file, not in the file that contains the HTML source. The HTML source file just contains the name of the image file.

You can organize items on your page by using lists and tables. Lists are fairly simple. A list consists of a sequence of indented items. Generally, the items are short, often consisting of a single line. Each item is preceded by a marker, such as a bullet or a number. An easy way to make a list is to type all the items, pressing return after each one. Then hilite the items and click on one of the two list-creation buttons in the toolbar.

Tables are more complicated. A table is made up of rows and columns. By default, tables are surrounded by a thin border. The table can have a different background color from the rest of the page, and in fact each row and each individual cell can have its own background color. You can set one cell to span several rows and/or several columns. In this way, you can make sophisticated layouts, and there are still more options that I have not even mentioned here.

To insert a table into your page, click the "Table" button in the toolbar. You will see a dialog box that allows you to set several options such as the background color and the number of rows and columns. You might want to set some of this, but you can always go back and change the table's properties later. When you click the Insert button, a table is added to your page.

To type into a cell, click in the cell to move the blinking cursor into it. You can edit the text in tables, just like you would any text. You can also drag an image into a cell. You can move the cursor from one cell to the next by pressing the tab key. If the cursor is located in the bottom right cell when you press the tab key, then a new row of cells will be added to the table.

As you must by now be able to guess, you can set the table's properties by right-clicking on the table. The pop-up menu that appears when you do this also contains commands for adding a row or adding a column to the table. If you select "Table Info" from this menu, you will get a dialog box with three panels, one for setting Table Properties, one for setting Row Properties, and one for setting Cell Properties. You can use the Row Properties panel, for example, to set the background color for an entire row. Have some fun playing with all the options!

As a finishing touch, you might want to add an email link to your page. The URL for such an email link contains an email address. Someone who views the page can send an email message to that address by clicking on the link (assuming, of course, that they have properly configured their browser for email). Here, for example, is an email link to me:

Send me mail!

The URL for this link is "mailto:eck@hws.edu", where "eck@hws.edu" is my email address. Substitute your own email address in your own email link.

HTML Email

Now that you have learned something about creating HTML pages, you might be interested to know that if you use Netscape Communicator for email, then you can send and receive email messages written in HTML. When you compose an email message in one of Netscape's email windows, you can use the full range of HTML composition tools. You can also drag images and links onto your page, just as can when you are creating a Web page.

When you send the message, you have the option of sending it in HTML format only or in a combination form that includes both HTML and plain text versions of the message. If the recipient of the message reads it with Netscape, they will see the message just as you created it, with colors, active links, graphics, and whatever else you put on the page. Unfortunately, recipients who don't use Netscape will only see the plain text version or the nasty HTML source code.

Before you can experiment with using Netscape for Email, you will have to configure it. I have a page on using Netscape Email, if you are interested.

--David Eck (eck@hws.edu), October 1997

Labs for The Most Complex Machine

xTurtle Lab 1: Introduction to Programming

THIS LAB is an introduction to a high-level programming language called xTurtle. This language was created to be used with The Most Complex Machine, but it is in the mainstream of high-level languages, along with Pascal, Ada and C. It incorporates some ideas common to all these languages, such as variables, assignment statements, loops, if statements and subroutines. (You should find that you are already familiar with the basic ideas because of your work in previous labs.) The xTurtle language also contains special-purpose commands for doing turtle graphics. These commands can be used to draw pictures on the computer screen. In this lab, you will learn about the basic xTurtle commands, about loops and if statements, and about variables. Future labs will cover programming in more detail, including the use of subroutines.

This lab covers some of the same material as Chapter 6 of The Most Complex Machine. The lab is meant as a self-contained introduction to this material, but it would still be useful to read Chapter 6 before doing the lab.

This lab includes the following sections:

Start by clicking this button to launch xTurtle in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Basic xTurtle Commands

When the xTurtle applet first starts up, it displays a white drawing area on the left, with a strip of controls on the right. There is a "turtle" in the center of the drawing area, represented as a small black triangle. The turtle has a position and a heading. Its heading is the direction it is facing, given as a number of degrees between -180 and 180. The turtle has a heading of zero when it is facing to the right, a heading of 90 when it is facing upwards towards the top of the screen, and a heading of -90 when it is facing downwards. Its position is given by two numbers: an xcoord, or horizontal coordinate, and a ycoord, or vertical coordinate.

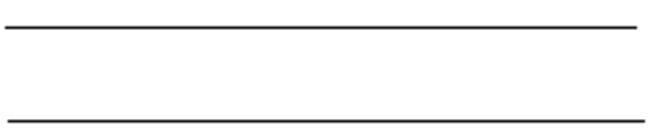
The drawing area of the applet includes a twenty-by-twenty square in which the turtle can move and draw. This square has horizontal coordinates from -10 on the left to 10 on the right, and it has vertical coordinates from -10 at the bottom to 10 at the top. Because the drawing area is unlikely to be exactly square, the coordinates for the entire drawing area probably extend beyond the range -10 to 10 in either the horizontal or vertical direction.

The turtle starts out in the center of the screen -- at the point (0,0) -- facing to the right. It can obey commands such as "forward(5)," which tells it to move forward five units, and "turn(120)," which tells it to rotate in place through an angle of 120 degrees. (It turns counterclockwise if the number of degrees in positive and clockwise if the number of degrees is negative.) The number in parentheses is called a parameter for the command; you can substitute any number you want. The parameter in a "forward" command tells the turtle **how far** to move forward, while the parameter in the "turn" command tells it **how many degrees** to turn.

The turtle can draw a line as it moves. You can think of it as dragging a pen that draws as the turtle moves. The command "PenUp" tells the turtle to "raise the pen." While the pen is raised, the turtle will move without drawing anything. The command "PenDown" tells the turtle to lower the pen again.

Just below the drawing area of the applet are a text-input box and a button labeled "Do It". You can type commands for the turtle in the text-input box. When you press return or click on the "Do It" button, the turtle will carry out the command or commands that you typed. You can type in several commands at once, or you can type in one command at a time, pressing return after each command. Note also that after a command is executed, the contents of the text-input box are hilited, so that as soon as you start typing, the previous command will be erased and replaced with what you type. And finally, note that you can change the speed at which the turtle follows a sequence of commands by changing the setting of the Speed pop-up menu, which is one of the controls located to the right of the drawing area. (The speed is initially set to "Fast".)

As an exercise, you should try to make the turtle draw two separate, parallel lines, like this:



If you make a mistake, you can use the command "clear" to clear the screen and the command "home" to return the turtle to its original position and orientation (at the center of the screen, facing right).

The turtle can execute a number of other commands, in addition to forward, turn, PenUp, PenDown, clear, and home. Here are a few more basic commands. In this list, x and y are parameters. You can replace a parameter with a number when you use the command.

- back(x) tells the turtle to back up x units, that is, to move x units in the direction **opposite** to its current heading. For example, back(3) tells the turtle to back up three units. Negative numbers are allowed as parameters for both forward and back.

Back(x) is provided only as a convenient shorthand for forward(-x).

- face(x) makes the turtle turn to a heading of x degrees from heading zero. For example, face(90) points the turtle straight up, face(-90) points it straight down, and face(180) points it to the left. Note the distinction between turn and face: turn specifies a change in direction from the current heading, while face specifies a new heading without any reference to whatever the old direction might have been.
- moveTo(x,y) tells the turtle to move from wherever it happens to be, to the point with coordinates x and y.
- move(x,y) is related to moveTo(x,y) in the same way that turn(x) is related to face(x). That is, while moveTo(x,y) says "move from the current location, whatever it is, to the point with coordinates (x,y)," move(x,y) says "move x units horizontally and y units vertically **from the current location**." Note that these commands do not depend upon or change the heading of the turtle.
- circle(x) draws a circle of radius x. You should think of the turtle moving in a circle starting from its current position and returning to that position at the end. Note that the turtle position is **on** the circle. If x is positive, the turtle curves to its left as it draws the circle, and the center of the circle is x units to the left of the original turtle position. If x is negative, the turtle curves to the right, and the center of the circle is to the right of the original position.
- arc(x,y) draws part of a circle of radius x. A full circle would be 360 degrees; arc(x,y) draws an arc of y degrees. As with circle(x), the turtle curves to the left if x is positive and to the right if x is negative. If y is negative then the turtle will "back up" along an arc. Note that the turtle changes position and heading as it draws.
- HideTurtle and ShowTurtle make the turtle (the small black triangle) invisible and visible. The turtle can still draw while it is invisible. (There is a check-box labeled "No Turtles" in the control strip to the right of the drawing area. When this box is checked, the turtle will always be invisible, **regardless** of whether the program uses any HideTurtle or ShowTurtle commands.)

Draw some pictures using these basic commands. Here are a few things you can try, for example:

- circle(7) circle(5) circle(3) circle(1)
- arc(5,90) arc(-5,-90) arc(5,90) arc(-5,-90)
- forward(5) turn(120) forward(5) turn(120) forward(5)
- forward(7) turn(90) forward(5) back(10)

Color

Unless you tell it to do otherwise, the turtle will draw everything in red. However, you can tell it to change its drawing color to any other color. The turtle understands the following basic color commands: red, blue, green, cyan, magenta, yellow, black, white, and gray. After it executes one of these commands, it will draw in the specified color until it comes to

another color-change command. For example, the following sequence of commands will draw a triangle with each side in a different color:

```
green forward(5) turn(120)
blue  forward(5) turn(120)
cyan  forward(5) turn(120)
```

Besides these basic color commands, there are two commands that can be used to specify any drawing color that the computer is capable of displaying. The commands are rgb(x,y,z) and hsb(x,y,z), where x, y, and z are parameters that can have any value in the range 0.0 to 1.0. To understand these commands, you need to know something about color. (But you can safely skip the details, if you want.)

Any color can be specified as some combination of the primary colors, red, green, and blue. In the command rgb(x,y,z), the parameters x, y, and z specify the amount of red, green, and blue in the color. For example, a value of zero for x indicates that the color is to contain no red at all, and a value of one for x means that the color contains the maximum possible amount of red. So, rgb(0,0,0) represents black, rgb(1,0,0) represents bright red, and rgb(0.5,0,0) is a darker red. Some other examples: rgb(0.8,0.8,0.8) is a very light gray, rgb(1,0.6,0.6) is pink, and rgb(0,0.4,0.4) is a dark blue-green.

The command hsb(x,y,z) uses an alternative method of specifying a color. In this command, x, y, and z represent hue, saturation, and brightness. The hue is the basic color: As x ranges from zero to one, the hue ranges through the spectrum from red through orange, yellow, green blue, violet, and back to red. The meaning of the brightness parameter is pretty clear, with a value of one representing the brightest color of a given shade. The saturation can be thought of as follows: A saturation value of one gives the purest possible version of a color. Decreasing the saturation from one towards zero is like mixing paint of that color with gray paint of equal brightness. The basic color remains the same, but it becomes "diluted."

You'll find some examples of using the rgb and hsb commands later in the lab.

Writing xTurtle Programs

The power of a computer comes from its ability to execute programs. The xTurtle applet allows you to write and run programs. Programs can include all the basic commands described above. They can also include loops, decisions, subroutines, and other features.

The applet can be configured to load one or more programs when it starts up. The applet that you launched with the button above should have loaded several sample programs that you will use in this lab. You can also write new programs from scratch. You can select among the programs that the applet knows about using the pop-up menu at the very top of the applet. You can begin a new program by clicking on the "New Program" button, or by choosing "[New]" from the pop-up menu. (There are also buttons for loading programs from files and for saving programs to files; however, the configuration of your Web browser might prevent these buttons from functioning.)

One of the sample programs for this lab is called "Necklace". Select this program from the

pop-up menu at the top of the xTurtle applet. This program contains an example of a **loop**. The program itself reads:

```
PenUp
moveTo(0,-5)
PenDown
LOOP
    arc(5,20)
    circle(-0.5)
    EXIT IF heading = 0
END LOOP
```

In addition to these commands, the program contains a lot of comments. A comment is anything enclosed between braces, { and }. Comments are meant for human readers of the program and are completely ignored by the computer. You should read the comments on the sample program to help you understand what it does and how it works.

In an xTurtle program, a loop consists of the word loop, then a sequence of instructions, then the words end loop. One of the instructions must be an exit statement, which gives a condition for ending the loop. In the sample program, the statement "exit if heading = 0" includes the condition "heading = 0. Since the heading is the direction that the turtle is facing, this condition is true when the turtle is facing in the direction zero (that is, directly towards the right).

When a loop is executed, the computer will execute the statements in the loop repeatedly. Each time the exit statement is executed, the computer tests the condition specified by that statement. If the condition is satisfied, the computer jumps out of the loop. (Ordinarily, after exiting from a loop, the computer jumps to the statement that follows the end loop. In this example, there is no further statement after the loop, so the program ends when the loop ends.)

To run a program in the xTurtle program, select it from the pop-up menu if necessary, so it is visible on the screen. Then click on the "Run Program" button. If there are no errors in the program, the computer will switch back to the drawing area and execute the program. You can control the rate of execution with the speed pop-up menu. You can pause the execution with the "Pause" button, and you can terminate it permanently with the "Stop" button. After the program has been executed, you can run it again by clicking the "Run Program" button. Note that every time you run a program, the turtle starts out in its initial configuration: at the center of the drawing area, facing right, with the pen down, and with the drawing color set to red.

Run the Necklace program, and try to understand how it works. As another example, click on the "New Program" button and then type in and run the following program. (Instead of typing it, you might want to be clever and use cut-and-paste.)

```
DECLARE hue
hue := 0
LOOP
   hsb(hue,1,1)
```

```
        forward(5)
        back(5)
        turn(360/100)
        hue := hue + 0.01
        exit if hue = 1
    END LOOP
```

This program uses a variable named "hue". (You can use any names you want for variables, as long as you avoid words that already have a special meaning, such as "loop" and "if".) A variable is just a memory location which has been give a name and which can be used to store a value. In xTurtle, you give a name to a memory location with a declare statement. Once you have declared a variable, you can store a value in it with an assignment statement, which has the form

```
<variable-name>  :=  <value>
```

The operator := is called the assignment operator. It tells the computer to calculate the value on the right and to store it into the variable on the left. The value on the right can be given as a number, as another variable, or as a mathematical formula. For example:

```
hue := hue + 0.01
x := 17
newAmount := oldAmount
cost := length * width * costPerSquareFoot
```

Next, we turn to an example that introduces the if statement. This is the sample program "RandomWalk", which should already be loaded into the xTurtle applet. Select the "RandomWalk" program from the pop-up menu and run the program several times. This program makes the turtle do a "random walk" in which it repeatedly moves in a randomly chosen direction. Read the program and the comments on it, and try to understand how it works.

In particular, look at the if statement in the Random Walk program. An if statement is used to decide among alternative courses of action. An if statement begins with the word if and ends with the words end if. (The "end if" here is not a separate command; it is merely a required syntactic marker to mark the end of the if statement. It is very different from an "exit if" statement, and you should try not to confuse them.) The exact rules for using if statements are rather complicated, and are covered in The Most Complex Machine and in the detailed on-line information about xTurtle. However, you should be able to get the basic idea by looking at the example in the sample program.

Interacting with the User

Any real programming language needs to provide some way for a program to communicate with the person who is using the program. The xTurtle programming language provides only minimal support for input and output, but what it provides is enough for a program to have a simple dialog with the user.

There are two commands for output (sending information from the computer to the user), and one command for input (getting information from the user into the computer). All of these commands use strings. A string is sequences of characters enclosed in quotes, such as: `"Hello"`. The command

```
DrawText("Hello")
```

will print the string `Hello` in the drawing area at the current turtle position. (Note that the quotes are **not** of the string that is displayed. The quotes are just there in the program to tell the computer that this is a string.)

The command

```
TellUser("Hello")
```

will display {\tt Hello} in a little green-colored box in the center of the display area. There will also be a button labeled "OK". The user reads the string and then clicks on the OK button to get rid of the box. The TellUser command has no permanent effect on the picture in the drawing area.

Finally, there is the more complicated input command, AskUser. This command allows the user to enter a number; the number entered by the user will be stored in a variable. The variable must, of course, be declared before it can be used in this command. For example, assuming that a variable named "betAmount" has been declared, the command

```
AskUser("How much do you want to bet?", betAmount)
```

will display a box containing the string "How much do you want to bet?" along with a text-input box where the user can enter a response. The number entered by the user will be stored in the variable betAmount, and the program can then use that number by referring to the variable.

All of the input/output commands have a nice feature that allows you to include the value of a variable in a string. If a string includes the special character #, then that # must be followed by the name of a variable. When the string is displayed, the # and the name will be replaced by the value of the variable. For example, if betAmount and winnings are variables with the values 25 and 75, then

```
TellUser("You bet #betAmount dollars; you win $#winnings.")
```

would display the string: `You bet 25 dollars; you win $75.`

All of this is illustrated in the sample program "InputOutputExample". You should select this program from the pop-up menu, read it, run it, and try to understand it.

Exercises

Exercise 1: Find a sequence of xTurtle commands to draw each of the following: a square with a circle inside it; a pair of parallel lines connected by half-circles at each end; a plus sign. The pictures should look something like this:

Exercise 2: Find a sequence of xTurtle commands that will draw your initials. Draw each initial in a different color.

Exercise 3: For this exercise, experiment with the "Necklace" sample program. The original program draws a large circle of radius 5, made up of a sequence of 20-degree arcs. Between each pair of arcs, another small circle is added as decoration. You can try changing the size of the arcs in the large circle. For example, change the "20" to "3" and you'll get a decoration every 3 degrees instead of every 20 degrees. You could also change the radius in the arc command. You can try using a different decoration. For example, try changing circle(-0.5) to forward(5) back(5). You could be more creative. Try using a triangle or a square as decoration. (Whatever commands you use to draw the decoration, you should be sure that they return the turtle to the same position and heading as when it starts.) Try to find the prettiest variation on the "necklace" theme that you can find.

Exercise 4: The following program was used as an example earlier in the lab. It draws 100 lines radiating out from a central point. Each line is drawn with a different "hue,", and the colors of the lines range throughout the entire spectrum.

```
DECLARE hue
hue := 0
LOOP
   hsb(hue,1,1)
   forward(5)
   back(5)
   turn(360/100)
   hue := hue + 0.01
   exit if hue = 1
END LOOP
```

How would you modify this program so that, instead of changing the hue from one line to the next, you change the saturation instead? What does the resulting picture look like? (Try it and find out!)

Exercise 5: The word random can be used in a program to represent a random value in the range from 0.0 to 1.0. (Random is actually a function with no parameters, but it acts like a variable that has a different value each time it is used.) What happens if you substitute the command rgb(random,random,random) for the command hsb(hue,1,1) in the program from Exercise 4? Why? Explain carefully what the modified program does.

Exercise 6: Modify the "RandomWalk" sample program so that each line is a different, randomly chosen color.

Exercise 7: In the "RandomWalk" sample program, the computer chooses one of the **four** directions 0, 90, -90 and 180 at random. Modify the program so that it chooses one of the

three directions 0, 120 and -120 instead. It should have an equal chance of choosing any of these directions. Make sure that you test your program!

Exercise 8: Write a program that can draw a square in any of three different colors. It should let the user of the program decide which color to use. Ask the user to "Enter 0 for red, 1 for blue, and 2 for green." If the user enters an invalid response, you should display an error message instead of a square.

Exercise 9: With what you have learned in this lab, you can write a simple guessing game program (which will use none of the graphical capabilities of xTurtle). Write a program in which the computer chooses a random integer between 1 and 100, and the user tries to guess the number. Each time the user makes a guess, the computer should (honestly) tell the user "Sorry, your guess is too high," "Sorry, your guess is too low" or "You got it." Although this program does something completely different from the random walk sample program, nevertheless it is similar in general outline. In particular, you will use an if statement inside a loop. Use a loop to allow for repeated guesses. The loop will end when the user guesses correctly. Your program can begin the following three lines, before the loop:

```
DECLARE answer
DECLARE guess
answer := randomInt(100)
```

Your program should include comments. Like the sample programs, it should use indentation to show the structure of the program. (The "Indent" button can be used to automatically indent a program; this feature is also useful for finding certain types of errors in a program, such as a missing end if.)

Exercise 10: This exercise assumes that you have completed Exercise 9 successfully. Improve the program from that exercise so that after each game, it will give the user the option of playing again. You will need to add **another** loop to the program, containing the loop that already exists. You will also need to know a new command. The YesOrNo command can be used to ask the user a yes-or-no question. Specifically, the command

```
YesOrNo("Do you want to play another game?", response)
```

will ask the user the given question. It will store a zero in the variable response if the user answers no and will store a one in that variable if the user answers yes. The outer loop should continue until the value of the response is zero.

Exercise 11: You have probably already discovered that the computer can display error messages if you try to run a program that contains an error. Errors that the computer can find before actually running the program are called syntax errors. The following program contains several syntax errors. Find each error and explain what is wrong in each case. (You can type in the program and let the computer find the errors.)

```
DECLARE length
length = 8
LOOP
   forward length
   turn(90)
```

```
      length := length - 1
      EXIT IF length equals 0
   END LOOP
```

Some errors can only be found when a program is running. For example, what error occurs when the following program is run? (Type it in and find out!)

```
DECLARE sum, count
sum := 0
count := 0
LOOP
   sum := sum + 1/count
   count := count + 1
   EXIT IF count = 10
END LOOP
DrawText("The sum is #sum.")
```

There is a third -- and much worse -- type of error, which occurs when the program gives an incorrect result but the computer gives you no warning of the fact that the answer is wrong. Give an example of such an error. Explain why the computer cannot detect such "wrong-answer errors."

Exercise 12: This exercise assumes that you are familiar with the "xComputer" model computer, which is used in some other labs. Write a short essay comparing the assembly language of xComputer with the high-level programming language, xTurtle. For example, you could: compare the way loops are constructed in each language; compare labels in assembly language to variables in xTurtle; and compare the way computations are done in assembly language with the way they are done by assignment statements in xTurtle.

--David Eck (eck@hws.edu), Summer 1997

xTurtle Info

The xTurtle Applet lets you program in a simple programming language, which is also called xTurtle. This page contains descriptions of the language and of the applet. This is a rather long file -- about 42K. Also available on separate pages are some tutorial examples.

I invented the xTurtle programming language to use as an example in The Most Complex Machine, a book that surveys the field of computer science. Since programming is only one of the topics in the book, I wanted a reasonably simple language, but one that would have the major features of a typical programming language. (By "typical" here, I really mean "Algol-like" or "Pascal-like", for those of you who know what that means.) These features include variables, assignment statements, loops, if statements, recursive subroutines, and even some multitasking. However, xTurtle does not include objects or data structures of any type.

The xTurtle applet is based on a similar program that I wrote for Macintosh computers. That program was one of several that I wrote for use with The Most Complex Machine. All the Macintosh programs are available for downloading. I am in the process of porting all the Macintosh programs to Java.

The xTurtle Applet

This section is a meant as a brief user manual for the xTurtle applet. The next section is a description of the programming language.

When the xTurtle applet first starts, it displays a large white graphics area, with a black, triangular "turtle" in the center. This turtle can move around in the graphics area. Usually, it draws a line as it moves. The turtle can respond to a number of commands, such as forward(5), which tells the turtle to move forward 5 units, and turn(45), which tells the turtle to rotate counterclockwise through an angle of 45 degrees. The turtle draws red lines, unless you tell it to use a different color.

The position of the turtle is given by a pair of coordinates, (x,y). Values of x and y between -10 and 10 are guaranteed to be visible in the graphics area. (Since the graphics area is not exactly square, the actual range of visible values can be larger in one direction.) The turtle can move outside the visible region, and will just keep drawing happily, even though you can't see what it's doing. The command home moves the turtle back to the center of the screen.

Below the graphics area is an input-box where you can type in commands for the turtle. When you press return, the turtle will carry out the commands or tell you that there was an error in your input. Clicking on the button named "Do It!" is equivalent to pressing return. If an error message is displayed, you can make it go away by clicking on it. It will also go away the next time you press return or click the "Do It!" button.

There are no restrictions on what you can type in the command-input box. You can even put a whole subroutine definition there. However, for anything that takes more than a few words, you'll want to write a program. If you want to create a new program, click on the "New

Program" button, at the top-right corner of the applet. The graphics area will be replaced by a text-input area where you can type your program. To run the program, just click on the "Run Program" command. If the program is correct, it will be compiled and executed. If there is an error, you'll get an error message. Again, you can make the error message go away, if you want, by clicking on it. (If there is an error, the computer will move the blinking cursor to the position in the program where it noticed the error. The browser should scroll the text, if necessary, to show the cursor. Unfortunately, not all browsers do this.)

Once the program has finished executing, you can run it again by clicking the "Run Program" button again. Before running a program, the computer always clears the screen and restores the turtle to its initial state (at the point (0,0), facing right, with pen down, and drawing color set to red). If you've just run a program that defines subroutines or variables, you can use them in any commands that you type into the command-input box. However, they will be cleared out of memory if you run another program.

At the top left corner of the applet is a pop-up menu that you can use to switch among the graphics display and any of the programs that the applet knows about. This includes programs you write, but it can also include programs that the applet loads automatically when it starts up. (If you know about <applet> tags: Applet parameters can be used to specify URLs of programs to be loaded when the applet starts up. However, I don't want to go into the details here.) This pop-up menu includes an entry for the graphics display. It also includes a special item called "[New]". Selecting "[New]" is equivalent to clicking the "New Program" button.

Finally, here is some information about each of the other buttons and other widgets that appear on the right-hand edge of the applet:

- Pause/Resume Button: This button is active while a program is running. Click it to pause the program. The name changes to "Resume", and you can then click on it to restart the program.
- Stop Button: This button is active while a program is running or paused. Click on it to stop the program permanently.
- Clear Button: If the graphics area is displayed, this button will clear it and restore the turtle to its starting point. If a program is displayed, it will erase it. (There is no way to undo this!)
- Indent Button: This button is available when a program is displayed. It will indent the lines of the program to show the program's structure.
- Load Button: Click on this button to load an xTurtle program from a text file on your hard drive. Note that this is likely to fail because of security restrictions on applets.
- Save Button: Click on this button to save the program to a file on your hard drive. Note that this is likely to fail because of security restrictions on applets.
- Speed Pop-up Menu: This menu, which is originally set to "Fast", controls the speed at which the turtle draws. (A delay is inserted after each turtle motion such as drawing a line, drawing a circle, or turning.)
- No Turtles Checkbox: When this box is checked, the little black triangular turtles are invisible. (By the way, this can speed up a program significantly, especially in a multitasking program where there are a lot of turtles.)
- Lock Step Checkbox: This box only has an effect on multitasking programs. When it is

unchecked, each turtle gets to execute a small random number of commands before control passes to the next turtle. When the box is checked, multiple turtles get to execute commands in strict alternation, one after the other. This often looks nicer, but the random version is a more realistic simulation of multitasking.

One thing that the applet **doesn't** have is the ability to print. This should be in a future version (although with the same security restrictions as loading and saving files).

The xTurtle Language

What follows is somewhat informal, but complete, specification of the xTurtle programming language, as implemented in the xTurtle applet. For many purposes, you might find the tutorial examples more useful. You might want to look at them first, in any case.

Program Structure

A program can contain comments. A comment begins with { and ends with }. Comments can be nested. Comments are for human readers only. They are ignored when the program is run.

The layout of a program on the page is ignored. You can have more than one command on a line, and you can split commands over several lines. You can't have spaces in the middle of words. (The command PenUp, for example, **cannot** be written as "Pen Up".) No distinction is made between upper and lower case letters. (So you can write PenUp, penUp, penup, or even PeNUp, and they will all mean the same to the computer.

A program consists of a sequence of one or more of the following: statements, variable declarations, subroutine declarations, function declarations. Subroutine and function declarations **cannot** be nested inside one another. (To provide for mutual recursion among subroutines and functions, they can be "predeclared." This will be covered below.) The program is executed from beginning to end. Declarations are much like statements in that you can think of them as being executed. That is, you can't use a variable, subroutine, or function until it has been declared.

Identifiers and Reserved Words

Certain words are reserved for special purposes in the xTurtle language. Reserved words cannot be used as names for variables, subroutines, and functions. Note that since upper and lower case are equivalent, reserving the word "declare" also reserved "DECLARE", "Declare", and so on. Reserved words in xTurtle include:

- Built-in subroutine names (listed below)
- Built-in function names (listed below)
- Predefined read-only variables (listed below)
- Logical operators: and, or, and not
- Reserved words for declarations: declare, import, end, endfunction, endsub, function, predeclare, ref, and sub
- Reserved words for control statements: else, end, endif, endgrab, endloop, exit, exitif, exitunless, grab, if, or, orif, loop, return, then, and unless

(Combined words like "endif" and "endfunction" are redundant, since they can all be written equivalently as two words: For example, "end if". Probably, it was a mistake to include these combinations in the language, but there they are. I will not mention them again; instead, I will always use two separate words.)

Variable names, subroutine names, and function names are collectively known as identifiers. You can make up your own names for the variables and routines that you declare, as long as you don't use reserved words and as long as you don't try to reuse a name in the same program. Identifiers must begin with a letter. They can contain letters, digits from 0 to 9, and the underscore character (_). They cannot contain spaces or other white space. They can be of any length.

Variables and Expressions

Before a variable can be used, it must be declared. This is done using a variable declaration, which consists of the reserved word, declare, followed by the names of one or more variables that are being declared. Variables in the list should be separated by commas. For example:

```
DECLARE InterestRate
DECLARE x, y, row, column
```

Variable declarations cannot be nested inside other statements, such as loops. They can occur in subroutines and functions (where they are used to create "local variables," as discussed below).

In xTurtle, the value of a variable must be a real number such as 42, 3.14159, -1, or 12.3e-12. The last example used scientific notation, which is legal in an xTurtle program. (The notation 12.3e-12 means 12.3 times 10 raised to the power -12. Very large and small numbers are written using scientific notation.) When a variable is first declared, it has a special value called "not-a-number" It is illegal to use such a value in a computation, and doing so will result in an error that will crash your program.

Variables and numbers can be used in mathematical expressions such as (1 + InterestRate) * Principal. Expressions can include the usual arithmetic operators plus (+), minus (-), times (*), divide (/), and exponentiation (^). Expressions can also include built-in functions and user defined function. For example: sin(2*angle+30). Parentheses are always required around the arguments of a function. (In the case of a function that takes no argument, a set of empty parentheses is optional.) The predefined functions are:

- The usual trigonometric functions: sin(x), cos(x), tan(x), sec(x), csc(x), cot(x). The arguments for these functions are measured in degrees.
- Some inverse trigonometric functions: arcsin(x), arctan(x), arccos(x).
- The exponential function exp(x), meaning e^x.
- The natural logarithm ln(x).
- The square root function sqrt(x).
- The absolute value function abs(x).
- The function round(x), which rounds its argument to the nearest integer.
- The function trunc(x), which truncates its argument by dropping any digits that follow

the decimal point.

- A random integer function, randomInt(x), which returns a random integer in the range from 1 to x, inclusive.
- A random real number function, random() (or just random without the parentheses), which returns a random number in the range from 0.0 to 1.0, including 0.0 but not including 1.0.

Note that the names of these predefined functions are reserved words.

There are a few reserved words in xTurtle that act like pre-defined read-only variables. These read-only variables contain information about the current state of the turtle. You can inspect the values of these variables, but you can't use assignment statements to change their values. The read-only variables are:

- forkNumber -- used to distinguish among multiple turtles when doing multitasking, as described below.
- heading -- the direction in which the turtle is facing, given in degrees between -180 and 180, where an angle of zero means that the turtle is facing to the right and positive angles are measured counterclockwise.
- isDrawing -- has a value of 1 or 0, depending on whether the turtle's pen is down or up. If the pen is up, the turtle doesn't draw when it moves.
- isVisible -- has a value of 1 or 0, depending on whether or not the turtle has been hidden with a HideTurtle command.
- xcoord -- gives the current x coordinate of the turtle.
- ycoord -- gives the current y coordinate of the turtle.

Assignment Statements

The value of a variable can be changed by using an assignment statement. An assignment statement takes the form

```
<variable> := <expression>
```

where `<variable>` is any declared variable (or the name of a parameter in a user-defined subroutine or function) and `<expression>` can be a number, a variable, or any expression created using operators and functions, as described above. Here are some example assignment statements:

```
Rate := 0.7
y := 3*x^2 - 2*x + 1
count := count + 1
r := exp(theta)
```

Built-in Subroutines

The xTurtle language includes many predefined subroutines. Most of the predefined subroutines are turtle graphics commands which move or turn the turtle or affect its state. Two predefined subroutines related to multitasking, Fork(n) and KillProcess, are discussed later. The others built-in subroutines are:

- forward(x) -- moves the turtle forward by a distance x along the direction in which it is currently facing. (If x is negative, the turtle actually moves backward.) Whether it draws a line, and what color that line is, depends on the current state of the turtle. (This same proviso applies to all the commands that move the turtle, and I will not repeat it.)
- back(x)-- moves the turtle backwards by a distance x along the direction in which it is currently facing. (If x is negative, the turtle actually moves forward.)
- moveTo(x,y) -- moves the turtle from its current position to the point with coordinates (x,y).
- move(dx,dy) -- moves the turtle from its current position to a point that is dx units away horizontally and dy units away vertically.
- turn(dA) -- rotates the turtle from its current heading through an angle of dA degrees. If dA is positive, the rotation is counterclockwise; if dA is negative, the angle is clockwise.
- face(A) -- rotates the turtle to a heading of A degrees from the zero position. (In the zero position, the turtle faces to the right.)
- Circle(r) -- draws a circle of radius r. The current turtle position is on the circumference of the circle. The turtle's heading is tangent to the circle. If r is positive, the circle lies on the turtle's left. If r is negative, the turtle lies on the turtle's right. The effect of Circle(r) is exactly the same as the effect of Arc(r,360). Note that the turtle's position and heading do not change.
- Arc(r,A) -- Draws an arc of a circle of radius r. The arc subtends an angle of A. The arc starts at the current position and heading of the turtle. The turtle ends up at the other end of the arc. If r and A are both positive, then the turtle curves forward and towards its left. If r is negative and A is positive, the turtle curves forward and towards its right. Negative angles move the turtle backwards.
- PenUp and PenDown -- used to raise and lower the turtle's pen. If the pen is up, it does not draw anything. This affects all the movement commands described above.
- HideTurtle and ShowTurtle -- HideTurtle makes the turtle invisible. ShowTurtle makes it visible again. It continues to draw while it is invisible, provided its pen is down. (If the applet's "No Turtle" checkbox is checked, the turtle will not be seen, no matter what is done with HideTurtle and ShowTurtle.)
- Halt -- Halts the program.
- Commands for changing the drawing color to one of eleven named colors: red, green, blue, cyan, yellow, magenta, black, darkGray, gray, lightGray, and white.
- rgb(x,y,z) -- Changes the drawing color, using an RGB color specification. The parameters x, y, and z must be in the range 0.0 to 1.0. They specify the amount of red, green, and blue, respectively, in the desired color. A value of 1.0 represents the maximum possible amount of a color. For example, rgb(0,0,0) is black, rgb(1,1,1) is white, and rgb(1,0.5,0.5) is pink.
- hsb(x,y,z) -- Changes the drawing color using an HSB specification. The parameters x, y, and z must be in the range 0.0 to 1.0. They specify the hue, saturation, and brightness respectively, of the desired color. Setting y and z equal to 1 gives bright, saturated colors. As x ranges from 0 to 1, the resulting hues cover the entire spectrum.

Input/Output

The input/output commands in xTurtle are pretty rudimentary. However, it is possible to display messages to the user and to read numbers input by the user. There are four built-in subroutines for doing such input and output. These subroutines are special in that they use strings. A string is a sequence of characters to be displayed to the user. (A string cannot include an end-of-line.) For example, the command

```
TellUser("Hello World!")
```

will display the characters

```
Hello World!
```

to the user. Note that in the program, the string is enclosed in double quotes, but that the quote characters are not part of the string that is displayed to the user. There are rules for converting the string in the program to the string to be displayed to the user: To display a quote character to the user, you have to use two quote characters in the string in the program. For example,

```
TellUser("I said, ""Stop!""")
```

will display

```
I said "Stop!"
```

The value of a variable can be included in the displayed string. To do this, you have to include a # character followed by the variable name in the string in the program. This works for declared variables as well as for the predefined read-only variables. For example,

```
TellUser("The turtle is at (#xcoord,#ycoord).")
```

will tell the user the current position of the turtle. That is, when the string is displayed, the value of xcoord will be substituted for #xcoord in the string, and similarly for ycoord. (If you want to display an actual # character, you have to write it as ## in the program.)

The TellUser command pops up a box to display its string. The user must click on an OK button to dismiss this box, and the program waits for the user to do so. The display is not changed.

There is also a command for displaying a string in the graphics display area. This DrawText command writes the string at the current turtle position in the current drawing color. The string is drawn even if the turtle's pen is currently up. After drawing the string, the turtle moves to a point just below its original position, so that the output of successive DrawText commands will line up neatly one under the other. DrawText has one parameter specifying the string to be drawn. For example,

```
DrawText("Hello World!")
```

There are commands for doing input: AskUser and YesOrNo. Each of these input commands has two parameters. The first parameter is a string to be displayed to the user. This string is meant to prompt the user for a response. The second parameter is the name of a variable where the user's response is to be stored. For AskUser, the use can type in any real number as a response. For YesOrNo, the user is given the choice of responding yes or no. If the user says yes, the value 1 is stored in the variable; if the user says no, the value 0 is stored. For both of these commands, the computer pops up a box to display the string and get the user's response. The program waits until the user responds. The display is not changed. Here are some

examples of using these two subroutines:

```
AskUser("What is the interest rate?", rate)
AskUser("Enter a number less than #max", x)
YesOrNo("Do you want to play again?", response)
```

Note that the strings used in these commands can include the values of variables, just like the strings in TellUser and DrawText.

Logical Expressions

LOOP statements and IF statements (described below) use logical expressions to test whether or not some specified condition is true. A logical expression is a formula that can be either true or false. Basic logical expressions are formed by comparing numerical values using the relational operators =, <, >, <=, >=, and <>. (The last three of these mean "is less than or equal to", "is greater than or equal to", and " is not equal to", respectively.)

Basic logical expressions can be combined into more complex expressions using the logical operators and, or, and not. (These can also be written as single characters: &, |, and ~.)

In the absence of parenthethes, the precedence ordering for operators in xTurtle, from highest to lowest, is:

```
NOT
AND
OR
relational operators
^
* and /
+ and -
```

LOOP Statement

To repeat a sequence of statements in xTurtle, use a LOOP statement, which consists of the reserved word loop, followed by the statements to be repeated, followed by end loop. One of the statements in the loop **must** be some sort of EXIT statement, which causes the loop to terminate (either unconditionally or conditionally) and transfers control to the statement that follows the loop. Statements can be nested. An EXIT statement always exits from the innermost enclosing loop. There are three forms of the EXIT statement:

```
EXIT
EXIT IF <condition>
EXIT UNLESS <condition>
```

The plain EXIT statement exits the loop unconditionally, and would ordinarily be used inside an IF statement that is nested inside the loop. In the other two forms of the EXIT statement, the `<condition>` is a logical expression, as defined above. An EXIT IF statement exits its loop if its condition is true; an EXIT UNLESS statement exits its loop if its condition is false.

Here are two simple example programs that use loops:

```
DECLARE ct                    DECLARE length
```

```
ct := 0
LOOP
   forward(1)
   turn(45)
   ct := ct + 1
   EXIT IF ct = 8
END LOOP
```

```
LOOP
   EXIT IF 1=2  { loop forever! }
   length := 7*random
                { 0 <= length < 7 }
   hsb(random,1,1)
   forward(length)
   back(length)
   face(360*random)
END LOOP
```

IF Statement

An IF statement is used to choose one of several alternative courses of actions. An IF statement always starts with a test of the form

```
IF <condition> THEN
```

and ends with

```
END IF
```

The end if is not an independent statement. It simply marks the end of the IF statement. Between the if and the end if, there are lots of options. Here are some examples that exhibit the options, with comments that explain what they mean:

```
IF d >= 0 THEN   { Simple choice:
                       do the following statements or skip them }
   r1 := (-b - sqrt(d))/(2*a)
   r2 := (-b + sqrt(d))/(2*a)
END IF
```

```
IF n/2 = trunc(n/2) THEN  { Branch: if condition is true, do the }
    n := n/2              {   statements between THEN and ELSE;  }
ELSE                      {   if the condition is false, do the  }
    n := 3*n+1            {   statements between ELSE and END IF }
END IF
```

```
IF grade > 90 THEN          { Multiway Branch: Each condition       }
   TellUser("Grade is A")   { is tested in turn.  As soon as one is }
OR IF grade > 80 THEN       { found that is true, the statements    }
   TellUser("Grade is B")   { following that condition's THEN are   }
OR IF grade > 70 THEN       { executed, and then the computer jumps }
   TellUser("Grade is C")   { out of the IF statement to whatever   }
OR IF grade > 60 THEN       { statement follows the END IF.  If     }
   TellUser("Grade is D")   { none of the conditions are true,      }
ELSE                        { then the statements between ELSE and  }
   TellUser("Grade is F")   { END IF are executed.  The ELSE part is}
```

```
END IF                          { optional.  If it is absent and if all }
                                { the conditions are false, then none   }
                                { of the statements within the IF       }
                                { statement are executed.               }
```

User-defined Subroutines

The xTurtle language has built-in subroutines like PenUp and moveTo(x,y). It is possible to define new subroutines in a program. A subroutine has a name and, optionally, a list of parameters. Once a subroutine has been defined, it can be called by giving its name and -- if it has a parameter list -- a list of values to be used for its parameters. A subroutine consists of a list of statements and variable declarations. When the subroutine is called, all the statements within the subroutine are executed.

Variables declared within a subroutine are called "local variables" for that subroutine. They are not visible from outside the subroutine and are deleted from memory when the subroutine ends. Variables that are not defined within a subroutine are called "global variables." A subroutine does **not** have automatic access to global variables. However, it is possible to give a subroutine access to global variables by explicitly "importing" them into the subroutine. This is done with an IMPORT statement, which consists of the reserved word import followed by the names of one or more previously declared global variables (separated by commas). IMPORT statements can only occur inside subroutine and function definitions.

A subroutine definition starts with the word sub, followed by the subroutine name, followed optionally by a list of parameters. The parameter list is just a list of parameter names, separated by commas. A parameter name can be optionally preceded by the reserved word ref, which indicates that the parameter is to be passed by reference. (This is discussed below.) The word sub, the subroutine name, and the parameter list make up the "subroutine header." Following the header come the statements that make up the subroutine. Finally, end sub is used to mark the end of the subroutine. Here are two sample subroutines:

```
SUB polygon(N,side)
   DECLARE count
   count := 0
   LOOP
      forward(side)
      turn(360/N)
      count := count + 1
      EXIT IF count = N
   END LOOP
END SUB
```

```
SUB UpdateAmount(ref amount)
   IMPORT InterestRate
   DECLARE interest
   interest :=
       InterestRate * amount
   amount := amount + interest
END SUB
```

When a subroutine is called, one parameter value must be provided for each parameter listed in the subroutine definition. The parameter values in the subroutine call statement are called "actual parameters." Parameters can be passed by value or by reference, as indicated by the absence or presence of the reserved word ref in the subroutine definition. When a parameter is passed by reference, the subroutine can change the value of an actual parameter that is provided to it when the subroutine is called. (The actual parameter for a ref parameter must be

a name; it cannot be a constant or a complex expression.) This is illustrated by the UpdateAmount example given above.

It is possible to exit from a subroutine at any point by using a RETURN statement, which consists simply of the word return. RETURN statements can only occur in subroutines. When the computer executes a RETURN statement, it exits immediately from the subroutine.

A subroutine can call itself. This is called "recursion." It is also possible for one subroutine to call another which in turn calls the first subroutine. Longer loops of subroutine calls are possible. This is called "mutual recursion." Because subroutines must be declared before they are used, a special syntax is required to make mutually recursive subroutines possible. One of the subroutines must be "predeclared". This is done by giving the reserved word predeclare, followed by the subroutine heading. The rest of the subroutine is omitted. Predeclaring a subroutine allows it to be called by other subroutines. A full definition of the predeclared subroutine must be given later in the program. The full definition includes another copy of the subroutine header.

User-defined Functions

A function is very similar to a subroutine, except that it computes and returns a value. A function in xTurtle is defined in the same way as a subroutine, with the word function substituted for the word sub. The only other difference in the definition is that a function must include a RETURN statement that specifies the value to be returned by the subroutine. A RETURN statement in a function takes the form

```
return <value>
```

where `<value>` is a constant, variable, or formula specifying the value to be returned. Here are two sample functions:

```
FUNCTION NextN(num)
   IF num/2 = round(n/2) THEN
      return num/2
   ELSE
      return 3 * num + 1
   END IF
END FUNCTION
```

```
FUNCTION UpdateAmount(amount)
  IMPORT Rate
  DECLARE Interest
  Interest := amount * Rate
  DECLARE newAmount
  newAmount := amount + Interest
  return newAmount
END FUNCTION
```

User-defined functions are used in the same way as built-in functions such as sin(x). Functions can have ref parameters, they can be recursive, and they can be predeclared.

Multitasking

In "parallel processing," several processes are going on at the same time. "Multitasking" is a way of simulating parallel processing by giving a little bit of execution time to one process, then a little bit to another process, and so on. Multitasking can be done in xTurtle by using the fork statement. Fork is a subroutine that takes a single parameter, specifying the number of processes to be created. This number must be between 1 and 100. Conceptually, the command fork(N) splits a turtle into N different turtles. Each of the turtles then proceeds to execute the following statements independently. Any variables that exist before the fork are shared by all

the turtles. However, if a variable declaration statement occurs after the fork, each turtle will create its own copy of the variable.

Each process created in a fork command continues executing until either: it reaches the end of the program, or it executes a KillProcess command, or it finishes the subroutine or function in which the fork command ocurred. Note how forks in subroutines are handled: Any processes that are created inside the subroutine end before the subroutine returns. When the subroutine returns, only the original process that called the subroutine is still running.

In fact, the command fork(N) actually creates N "child processes." The original parent process goes to sleep until all the child processes have ended. Then the original parent awakens. The state of the turtle in the awakened parent process is the same as it was before the child processes were created. That is, the turtle still has the same heading, position, visibility, pen state, and drawing color as it did at the moment when the fork statement was executed. If the fork command occurred inside a subroutine, the subroutine does not return until **all** the child processes have ended, and then it is actually the original parent process that returns. (The same sort of thing happens if you use a fork command in the xTurtle applet's command-input box.)

The processes that are created by a fork command are identical, except for one thing: Each process has a different value for the read-only variable forkNumber. The fork numbers for the processes created by the command fork(N) range from 1 to N. The different values for forkNumber allow the different processes to do different things. Note that it is certainly possible to have two or more forks in a row. A process only remembers the fork number from the most recent fork command that it has executed.

Grab Statement

After a fork command has been executed, the processes that it creates can communicate by setting or checking the values of shared variables (variables that were declared before the fork command). This form of communication has the problem of "mutual exclusion" -- making sure that only one process at a time has access to the shared variable. It is up to the programmer to enforce mutual exclusion. In xTurtle, this can be done using the grab statement, which takes the form

```
GRAB <variable> THEN
    .
    .  { any statements -- except fork, exit, return }
    .
END GRAB
```

The `<variable>` in a GRAB statement must be a global variable. If you want to use a GRAB statement inside a subroutine, you will have to use an imported global variable. The point here is that only one process at a time is allowed to grab a given variable. If a process tries to grab a variable and another process has already grabbed it, then the second process has to wait until the first process exits from its GRAB statement.

There is a variation of the grab statement that has an ELSE part. (I have never found it actually useful.) It has the form:

```
GRAB <variable> THEN
      { some statements }
```

```
ELSE
      { more statements }
END GRAB
```

In this case, if the GRAB fails, the computer does not wait. Instead, it executes the ELSE part of the grab statement. The ELSE part can include fork, exit, and return statement.

David Eck (eck@hws.edu), June 1997

Labs for The Most Complex Machine

xTurtle Lab 2: Thinking about Programs

THIS LAB CONTINUES THE STUDY of programming, which was begun in the previous lab. The emphasis here is on how a complex program can be developed to perform a specified task. An organized approach to programming is necessary for all but the most simple programs. Complex tasks can be broken down into simpler tasks, and complex programs can be built up out of simple components. The problem is how to determine what components are needed and how to piece them together.

Before beginning this lab, it would be useful to be familiar with the material in Chapter 6 of The Most Complex Machine, especially Section 6.3. In particular, this lab uses the ideas of preconditions and postconditions. This lab also uses the "nested squares" example from Section 6.3. You'll also find an introduction to subroutines in this lab. Subroutines are covered in Chapter 7 of the text.

This lab includes the following sections:

Start by clicking this button to launch xTurtle in its own window:

(Sorry, your browser doesn't do Java!)

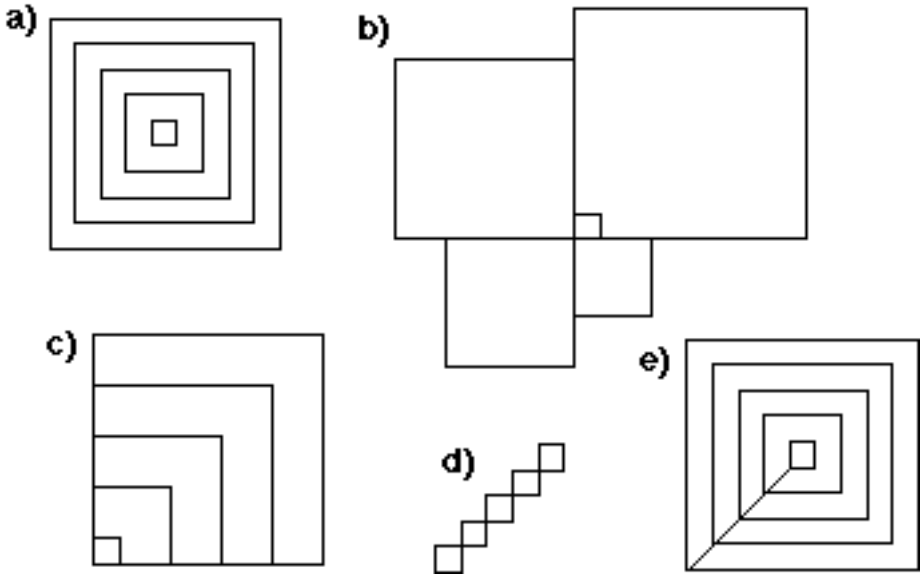
(For a full list of labs and applets, see the index page.)

Preconditions

The xTurtle applet that you have launched should have loaded a sample program called "NestedSquares". Select this program using the pop-up menu at the top of the applet. Read the program and the comments, and run the program by clicking on the "Run Program" button. For this example, and for much of the lab, I suggest that you **use the speed pop-up menu to reduce the speed at which programs are executed**, so that you will better understand what is going on.

As explained in the text, the key to getting this program correct was making sure that the preconditions for drawing each square were set up properly. A precondition is something that must be true at a certain point in a program, **if** the program to continue correctly from that point.

The following picture, taken directly from Figure 6.9 in the text, shows the correctly drawn squares and the results of five incorrect attempts to draw them. For each of the incorrect versions, the error can be traced to the fact that one or more preconditions was not met in the program that produced that picture:

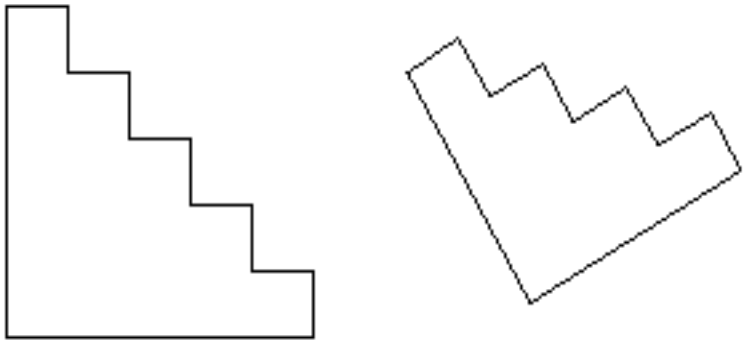


In the "NestedSquares" program, there are several statements near the end of the loop that are responsible for making sure that the required preconditions are met. You should try to understand why each of these statements is required. You will work with this example in Exercise 1 at the end of the lab.

Postconditions

Preconditions are things that must be true at a given point in a program for that program to continue correctly. A postcondition is something that is **actually** true at a given point in the program, because of what has been done by the program so far. A common way for programmers to think about programs is to ask, "At this point in the program, do the postconditions from what comes before match up with the preconditions for what is done next." In addition, the effect of a program **as a whole** can be thought of as a set of one or more postcondition for the entire program. The postconditions of a program are things that are true after the program has been executed. That is, they are things that are accomplished by the program. For an xTurtle program, the postconditions of the program include the picture that has been drawn on the screen.

The following picture shows a simple "staircase" with 5 steps and another staircase, with 4 steps, leaning at a 30 degree angle:



Suppose that you want a program to draw such staircases. Let's say that the number of steps in the staircase will be input by the user. You will have to use a loop to draw the steps, since when you are writing the program, you don't know how many steps there will be. Each execution of the loop will draw one of the staircase's steps. Before drawing each step, the turtle must be facing in the right direction; this is a precondition. After drawing the step, the turtle has changed direction; this is a postcondition. You have to include commands that will provide "splicing" from the actual postcondition to the desired precondition. After the loop, you will still have to draw the two long sides of the staircase. To get them into the correct positions and orientations, you will have to think about the **postconditions** that hold after the loop has been executed and how they match up with the **preconditions** for drawing the lines. Exercise 5 at the end of the lab asks you to write this program.

Subroutines

A subroutine is -- more or less -- a small program made into a black box and given a name. Some subroutines, such as forward and PenUp are predefined; others are written by a programmer as part of a larger program. Subroutines are an essential tool for organizing complex tasks.

Most subroutines have parameters such as the 9 in forward(9) or the 30 in turn(30). Parameters allow subroutines to receive information from the rest of the program or to send information back. Suppose that we want to turn the staircase program described above into a subroutine. Then it would no longer make sense to get the input from the user, since that would greatly limit the generality of the subroutine. Instead, the number of steps would probably be provided as a parameter. From the "point of view of the subroutine," the parameter is like input coming from "somewhere outside," just as input from the user comes from outside the program.

A subroutine definition begins with the word SUB and ends with the word END SUB. Just after the word SUB comes the name of the subroutine and (optionally) a list of one or more parameter names. The subroutine name and the parameters form the interface of the subroutine; everything from there up until the END SUB is the implementation.

The sample program "SpiralsSubroutine" defines a subroutine named spiral. Select this program from the pop-up menu at the top of the xTurtle applet, and read the program and comments. When you click the "Run Program" button to run this program, it will look like

nothing has happened! But in fact, the effect of the program is to define the subroutine. Ordinarily, the computer has no idea what the word "spiral" means, but once the computer compiles the subroutine definition, it will then understand commands like spiral(61) and spiral(89). Such commands can be added to the program after the subroutine definition, or they can be entered into the text-input box below the drawing area in the xTurtle applet. Try it. Some of the pictures you can make are rather pretty!

Exercises

Exercise 1: Consider each of the pictures **b)**, **c)**, **d)**, and **e)** in the nested squares illustration, shown above. For each of these incorrect versions, determine what small change in the program "NestedSquares" would produce that picture. In each case, it's a question of removing one or more statements from the correct program, so that one or more of the required preconditions are not met. In each case, determine which statement to remove and what precondition or preconditions are unmet in the resulting program.

Exercise 2: Select the sample program "Quadratic" from the pop-up menu at the top of the xTurtle applet. When you run this program, it will ask you to input three numbers, A, B, and C. It will then compute and display the solutions to the quadratic equation $A*x^2 + B*x + C = 0$. If you run the program and enter the values 1, 1, and -1 for A, B, and C, it will work fine. However, if you enter 1, 1, and 1 as the values of A, B, and C, the program will crash. This crash can be traced to the fact that at a certain point in the program, there is a precondition that might not be satisfied. If it is not satisfied, an error occurs and the program crashes. What precondition is not properly checked by the program? (It has something to do with the square root function. Recall that in some cases, the quadratic equation has no solutions.) Modify the program so that it does not crash when the input values fail to meet the precondition. Instead of crashing, the program should display an error message and halt.

Exercise 3: This is a continuation of Exercise 2. The "Quadratic" sample program actually exhibits another precondition, which is violated if A equals zero. Where does this precondition occur in the program, and what exactly is the problem with having A=0?

Exercise 4: This is a continuation of Exercises 2 and 3. Another way to deal with a precondition is to write a loop that can only end when the precondition is satisfied. Modify the "Quadratic" program so that the input from the user is read in a loop that can only end if the user has entered legal values for A, B, and C. After the loop, the program can safely compute and print the solutions to the equation.

Exercise 5: Write a program that can draw staircases, like those shown in the picture above. The program you write must meet the following requirements:

1. The user will be asked to specify the number of steps.
2. Each step is one unit high and one unit wide.
3. The orientation of the staircase will depend on the starting orientation of the turtle, as shown in the second example in the picture. This means that you should draw the staircase using only forward, back, and turn commands. Do **not** use face or moveTo.

4. After the staircase is drawn, the position and heading of the turtle will be the same as they were when the drawing begins. This is a postcondition for the program as a whole.

Start your program with the following three lines:

```
DECLARE NumberOfSteps
AskUser("How many steps?", NumberOfSteps)
DECLARE count
```

The variable named count should be used as a counting variable in a loop to count the number of steps that have been drawn. You should think about preconditions and postconditions as you write the program. Include comments in your program that discuss specific preconditions and postconditions for various parts of the program, and explain how they were used -- or could have been used -- in developing the program. It is a bit easier to write the program if you start drawing the steps at the top of the staircase.

Exercise 6: Convert the program you wrote for Exercise 5 into a subroutine. **Start by removing the first two lines of the program**, which were given to you in Exercise 5. Replace them with:

```
SUB stairs(NumberOfSteps)
```

Add the line

```
END SUB
```

at the end of your program. These two steps turn your staircase-drawing program into a staircase-drawing subroutine. After running the modified program, you will be able to use commands like stairs(5) to draw a staircase with five steps and turn(30) stairs(4) to draw a tilted staircase with four steps. To make a more interesting picture, add the following lines at the end of your modified program, after the definition of the subroutine, and then run the program:

```
LOOP
   stairs(3)
   stairs(5)
   stairs(7)
   turn(30)
   EXIT IF heading = 0
END LOOP
```

What picture is drawn by one execution of the loop in this program? Why? (You will only get the correct picture if your solution to Exercise 5 meets the fourth requirement imposed on the program in that exercise.) What picture is drawn by the program as a whole? Why?

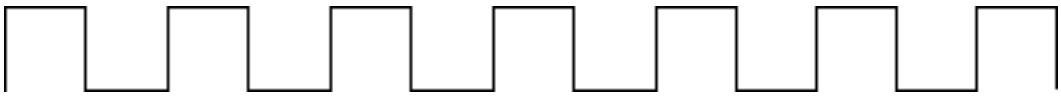
Exercise 7: Here are some questions about the subroutine you wrote for Exercise 6:

- Why does the single line SUB stairs(NumberOfStairs) replace the first two lines of the program from Exercise 5? Why does it make sense for the **variable**, NumberOfStairs, in the program to become a **parameter** in the subroutine?

- In the subroutine, why don't you ask the user for the number of steps to draw? Why is it better to use a parameter?
- The variable "count" in the program from Exercise 5 becomes a local variable in the subroutine. Why? Why is it a local variable instead of a parameter?
- What did you learn from Exercise 6 about subroutines and their use in complex programs?

Exercise 8: Explain carefully why running the sample program "SpiralsSubroutine" does not produce any output or have any visible effect. What exactly does the program do when it is executed? What is the point of it, if it doesn't do anything?

Exercise 9: Write a program that will draw pictures like the following, where the number of "bumps" is input by the user. (In this picture, there are seven bumps.) Note that the number of vertical lines is one more than the number of horizontal lines.



Exercise 10: Convert your program from Exercise 9 into a subroutine in which the number of bumps is specified by a parameter. How would your subroutine be used to produce a picture with seven bumps, like the one shown above?

Exercise 11: The contract of a subroutine is defined to be everything you need to know about a subroutine in order to use it correctly. This includes the name of the subroutine and its list of parameters. It also includes specifications of what must be true before the subroutine is called and what will be true after it finishes execution. These specifications are the preconditions and the postconditions of the subroutine. In an xTurtle subroutine, the preconditions and postconditions usually include statements about the position and orientation of the turtle and about what is drawn on the screen. Here are two square-drawing subroutines. Describe the contract of each subroutine. Include complete specifications of the preconditions and postconditions for each subroutine.

```
SUB square(r,g,b,size)
   rgb(r,g,b)
   forward(size)
   turn(90)
   forward(size)
   turn(90)
   forward(size)
   turn(90)
   forward(size)
   turn(90)
END SUB
```

```
SUB square(x,y,size)
   PenUp
   moveTo(x,y)
   PenDown
   move(size,0)
   move(0,size)
   move(-size,0)
   move(0,-size)
END SUB
```

---

adopted for use in that course.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xTurtle Lab 3: Subroutines and Recursion

SUBROUTINES WERE INTRODUCED in the previous lab. This lab will continue the study of subroutines. The lab concentrates on the idea of a subroutine as a black box and on recursive subroutines that call themselves, either directly or indirectly.

You should be familiar with the material from Chapter 7 of The Most Complex Machine, especially with the material on recursive subroutines from Section 3. The Koch curve and the binary tree introduced in that section will be used in the lab.

This lab includes the following sections:

Start by clicking this button to launch xTurtle in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Black Boxes

You are familiar with the idea of a subroutine as a black box. When you use predefined subroutines such as forward and moveTo, you don't need to know exactly how they work. All you need to understand is how to use them and what they will do. User-defined subroutines can also be used as black boxes, provided that someone else has written them for you.

The xTurtle applet that you launched for use in this lab is set up to load a sample program called "SymmetrySubs" This program contains the definitions of six subroutines for drawing symmetric pictures. These subroutines are meant to be used in the same way as the usual drawing subroutines such as forward and circle. For example, one of the subroutines defined in "SymmetrySubs" is multiForward. This subroutine is similar to the built-in forward command, except that instead of just drawing one line, it draws eight lines in a symmetrical pattern. The "SymmetrySubs" program defines the following six subroutines, each of which draws a symmetric pattern: multiForward, multiBack, multiMoveTo, multiMove, multiCircle, and multiArc.

To see how this works, select the "SymmetrySubs" program from the pop-up menu at the top of the xTurtle applet, and use the "Run Program" button to run the program. Nothing

appears on the screen, since all the program does is define some subroutines. However, once the subroutines are defined, you can use them as commands, just as you would use any of xTurtle's built-in commands.

As an example, type the following commands into the text-input box below the xTurtle drawing area, pressing return after you enter each command:

```
multiArc(5,40)
turn(-40)
multiforward(3)
multicircle(2)
```

Also try some other commands. If you want to make a more complicated picture, go back to the "SymmetrySubs" program and add your commands at the **end** of that program (after the definitions of all the subroutines). For example, add the following commands to the end of the program, and then hit the "Run Program" button:

```
LOOP
    multiforward(0.5)
    face(randomInt(360))
    EXIT IF 1=2
END LOOP
```

You will have to end the program with the "Stop" button. (The statement "exit if 1=2" in this program is a fancy way of saying "Never exit.")

In the previous part of the lab, you used several subroutines as black boxes, without having to understand what went on inside the box. But you should remember that "not having to know what's inside" is only part of the black box story! When you write a subroutine **yourself**, you are working **inside** the box. While you are writing the subroutine, you can concentrate on making it perform the specific task it is designed to do, without worrying for the moment exactly what role it will play in a complete program. From the point of view of a programmer trying to design a complex program, subroutines are a tool for breaking a complex problem down into smaller, more manageable subproblems.

Recursive Trees and Recursive Walks

A recursive subroutine is one that calls itself. A recursive subroutine is a black box that uses **itself** as a black box. Section 7.3 in the text introduces recursive subroutines using the example of a binary tree. The program for this example is in the Sample program "BinaryTrees". Select this program from the pop-up menu at the top of the xTurtle applet and run it. Nothing will happen, since the file only defines some subroutines. The main subroutine defined in the file is called TestTree. If you enter this into the text-input box beneath the drawing area of the applet, you will be asked to specify a complexity level. The computer will draw a tree with the complexity that you specify. Try this, for example, for a complexity level of 5.

A binary tree of complexity zero is defined to be a single straight line segment. A binary

tree of complexity greater than zero is defined to consists of a "trunk," which is just a line segment, with two "branches" attached to it. Each of the branches is a binary tree, which is smaller than the complete tree and which has smaller complexity than the complete tree. This is a recursive definition because we are saying that a tree contains pieces which are themselves trees. Because binary trees are defined recursively, they can be drawn by a recursive subroutine. You should try to understand the definition of the Tree subroutine.

A second example of recursion is contained in the sample program "KochCurves." This example is also taken from Section 7.3 in The Most Complex Machine. A Koch curve is a way of getting from one point to another -- with a lot of detours. To help you understand this, run the sample program "KochCurves." After you have run the program, you can use the command TestKoch in the xTurtle applet's text-input box. When you do so, you will be asked to specify a complexity level for the Koch curve. You should try complexity levels of 0, 1, 2, 3, and 4.

A Koch curve of complexity 0 is defined to be a straight line segment. A Koch curve of complexity 1 is a line segment with a "bump" or "detour." The complexity-one curve is made up of four line segments, but you should think of each line segment a Koch curve of complexity zero. A Koch curve of complexity 2 is obtained from the curve of complexity 1 by adding a detour to each line segment in the curve. You should look at a Koch curve of complexity 2 as being made up of four smaller pieces, where each piece is a Koch curve of complexity 1. More generally, a Koch curve of complexity N is made up of four smaller Koch curves of complexity N-1. Once again, this is a recursive definition, and the subroutine that draws Koch curves is a recursive subroutine. Try to understand how the pictures you see are produced by the Koch subroutine.

The "KochCurves" program also defines a subroutine named "Snowflake" which draws "Koch snowflakes." A Koch snowflake is made by joining three Koch curves together at their endpoints, producing a symmetric, snowflake-like picture. Try it!

Exercises

Exercise 1: Add the following lines to the **end** of the "SymmetrySubs" sample program, after all the subroutines. Run the program a few time, to see what it does. The write a short essay explaining exactly how the program works and why it produces the pictures that it does. (Try running this at fastest speed, with the turtle turned off.)

```
declare x
x := 0
loop
   face(360*random)
   hsb(x,1,1)
   multiforward(0.4)
   x := x + 0.005
   if x > 1 then
      x := 0
```

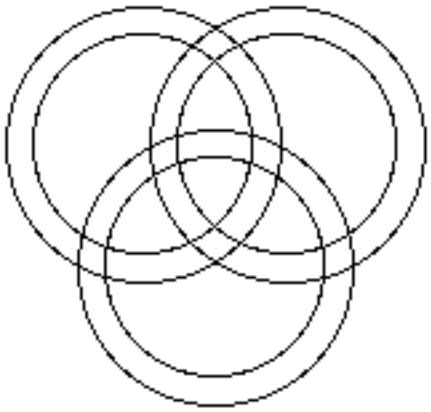
```
        end if
        exit if 1 = 2
    end loop
```

Exercise 2: The mathematics used in the subroutines defined in "SymmetrySubs" is not trivial. How much did you need to know about this mathematics to use the subroutines? What point about subroutines does this illustrate?

Exercise 3: The Speed pop-up menu in the xTurtle applet works by inserting delays betweens commands. Since circle is a single command, a complete circle is drawn instantaneously, no matter what the setting of the speed menu. This disappointed one of my students, who wanted to be able to watch the circles being drawn. Write a subroutine

```
SUB SlowCircle(radius)
```

that will draw a circle by drawing 60 arcs, where each arc covers 6 degrees. Then use your subroutine to (slowly) draw a picture like the following:



Exercise 4: Modify your subroutine from Exercise 3 so that it draws each of the arcs of the circle in a different color. The color of an arc can be set to hsb(hue,1,1). At the beginning, the value of hue should be zero. After drawing each arc, it should be increased by 1/60.

Exercise 5: A subroutine such as the one you wrote for Exercise 4 can be used in many different programs. What did you have to know about those programs in order to write the subroutine? What does this illustrate about black boxes?

Exercise 6: How many straight line segments are there in a Koch curve of complexity 2? You can use the TestKoch subroutine to draw the curve and them count the line segments. You could do the same for complexity 3, and maybe for complexity 4. But what about complexity 10 or 100? There are two many line segments to count. However, it is possible to predict the number of segments for any complexity, if you think about how Koch curves are created. The question you should ask yourself is, When the complexity is increased by one, what happens to the number of line segments? Try to figure out the pattern by looking at the number of line segments in curves of complexity 1, 2, 3, and 4 and by thinking about what happens as you go from one curve to the next. Try to find a formula that gives the number of line segments in a Koch curve of any given complexity.

Exercise 7: This exercise asks you to find the number of line segments in a binary tree of a

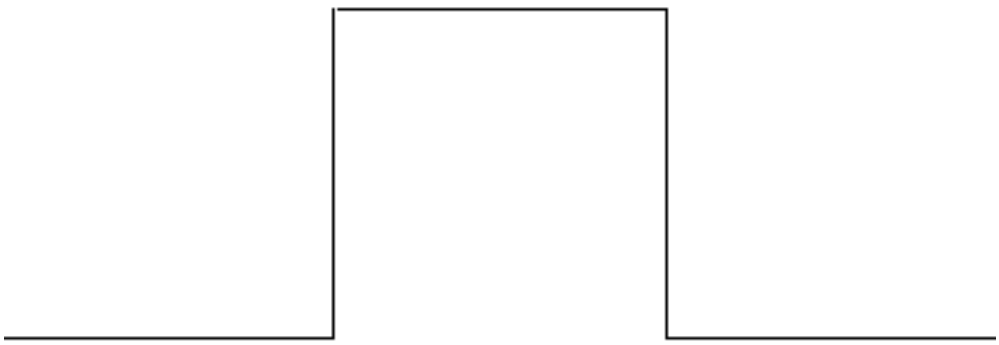
given complexity. It is similar to the previous exercise, but it's harder to find a formula in this case. Run the "BinaryTrees" example program, and then use the TestTree subroutine to draw trees of complexity 0, 1, 2, 3, 4, and 5. For each of these trees, count the number of straight line segments that it contains. For example, in a tree of complexity 1, the number of line segments is 3 -- each branch is a single line, and the trunk is the third line. You should try to find a formula that gives the number of line segments for **any** given complexity. You might not be able to find a formula that gives the number directly, but you should at least be able to find a formula that tells how the number of line segments changes as you go from one complexity level to the next.

One way to approach this problem is first to determine how many **new** lines are added to the tree when you go from a tree of complexity N to one of complexity N+1. Then use that to figure out the total number of line segments. Another approach is to "think recursively": Remember that a tree of complexity N+1 is made up of a trunk plus two trees of complexity N.

Exercise 8: The text notes that you can add randomness to a Koch curve by deciding randomly whether to detour to the left (using turns of 60, -120 and 60) or to the right (using turns of -60, 120, and -60 instead). Make this change to the program "KochCurves" sample program, and try it out. In the subroutine, you can declare a variable named x (for example) and set x := RandomInt(2). Use the value of x to decide whether to detour to the left or to the right. (When you have made the change, the SnowFlake subroutine will produce a "Koch Island" instead. Try it!)

Exercise 9: The idea of "detouring" used in making Koch curves can be used to make other interesting fractal pictures. In a Koch curve, the idea is to replace a straight line with a line containing a "triangular detour," like this:

Suppose that a "square detour" were used instead, looking like this:



What would the resulting picture look like, for higher degrees of complexity? Find out by rewriting the subroutine Koch to use square detours instead of triangular detours.

Exercise 10: For this exercise, you will write a recursive subroutine that displays an element of randomness. The subroutine will be called "mountain", because it draws pictures that look a bit like a mountain range. Here are two pictures produced by the subroutine:

Each of these pictures was produced with the commands:

```
clear penUp moveTo(-7,0) penDown mountain(7,0,10)
```

The command mountain(x,y,c) should move the turtle along a jagged path from its current position to the point (x,y). The amount of jaggedness is specified by the "complexity," c. If the third parameter, c, is zero, then mountain(x,y,c) simply draws a straight line from the current position to (x,y). If c > 0, then mountain(x,y,c) will choose a random point, (x1,y1), somewhere between the current position and (x,y). It will draw a "mountain" of complexity c-1 to the point (x1,y1) and from there it will draw a second mountain curve of complexity c-1 to the point (x,y). The trick is to choose the intermediate point (x1,y1). This can be done by finding the midpoint between the current position and (x,y), and then moving the y coordinate of that midpoint up or down by a random amount. This computation can be done as follows, recalling that the current position of the turtle is given as (xcoord,ycoord):

```
x1 := (xcoord + x) / 2
y1 := (ycoord + y) / 2
y1 := y1 + (random - 0.5) * (xcoord - x)
```

Try to put all this together into a definition of mountain(x,y,complexity). Remember to declare x1 and y1 at the beginning of your subroutine.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xSortLab Lab: Sorting and the Analysis of Algorithms

ONE OF THE MOST COMMON OPERATIONS performed by computers is that of sorting a list of items. An example of this would be sorting a list of names into alphabetical order. This lab deals with two natural questions: How can sorting be done? And how can it be done efficiently?

The second question is one that would be asked in a field of study called the analysis of algorithms. Recall that an algorithm is an unambiguous, step-by-step procedure for solving a problem, which is guaranteed to terminate after a finite number of steps. For a given problem, there are generally many different algorithms for solving it. Some algorithms are more efficient than others, in that less time or memory is required to execute them. The analysis of algorithms studies time and memory requirements of algorithms and the way those requirements depend on the number of items being processed. In this lab, you'll look at the time requirements of various sorting algorithms.

In the lab, you will see five remarkably different algorithms for sorting a list. Each algorithm solves the sorting problem in a different way. You will see how each algorithm works, and you will also see that some sorting algorithms are much more efficient than others.

Sorting and the analysis of algorithms are discussed as one example in Section 9.3 of The Most Complex Machine. Some of the material from that section is repeated in this lab. However, some background material and motivation is not repeated here, and it would be worthwhile for you to read Section 9.3 before doing the lab.

This lab includes the following sections:

You'll be using an applet called xSortLab. Start by clicking this button to launch the applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

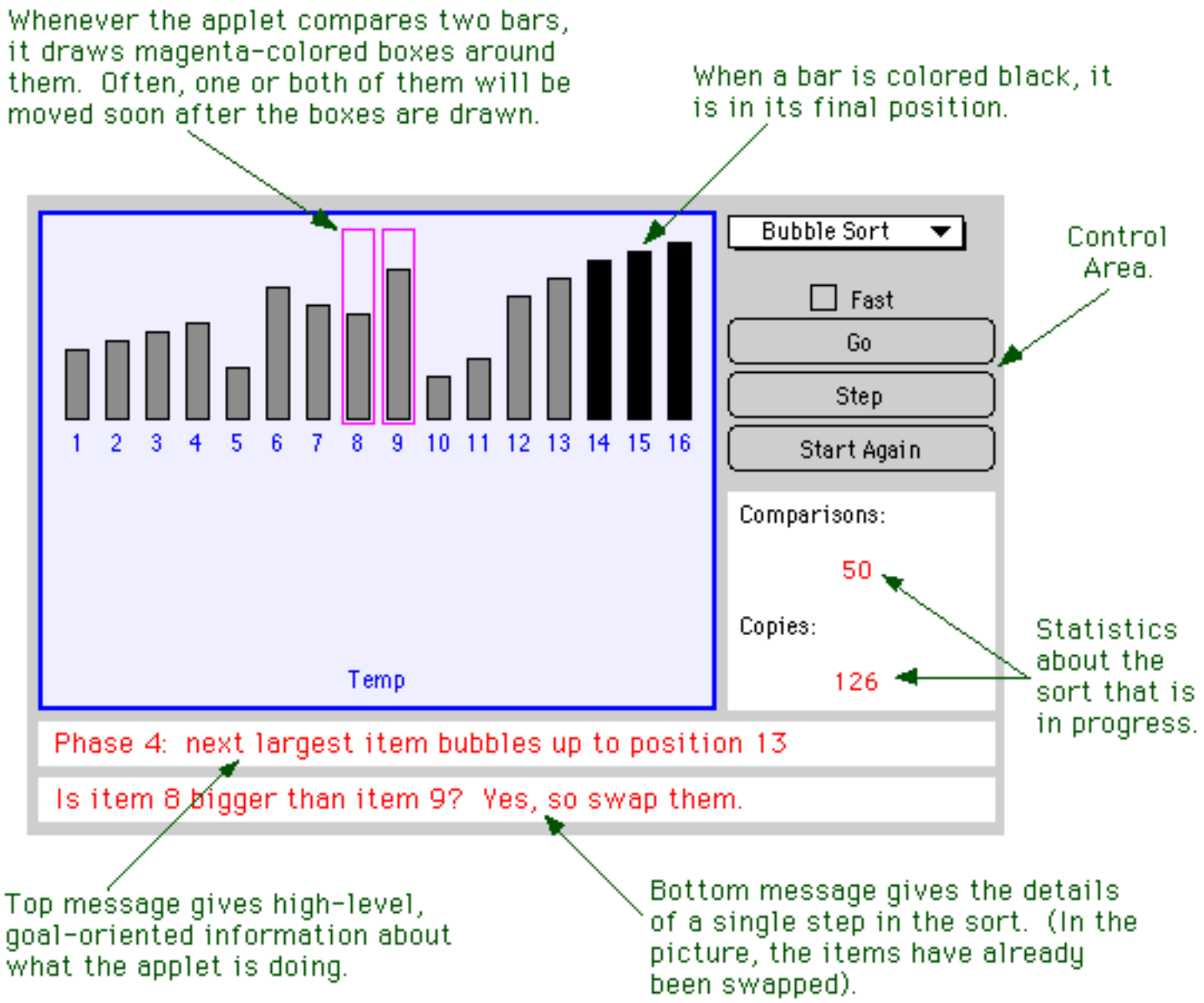
Watching Bubble Sort

The xSortLab applet has two operating modes: "Visual Sort" and "Timed Sort." There is a pop-up menu at the very top of the applet that can be used to select one of these modes. The pop-up menu also contains an entry for a "Log" which records statistics about sorts that have been performed by the applet. When the applet first starts up, it is in Visual Sort mode, and you will use this mode for the first part of the lab. In this mode, you can watch as the applet sorts sixteen bars into order of increasing height. Later in the lab, you'll be using the Timed Sort mode.

The applet can perform five different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and QuickSort. When it fist starts up, it is set to use Bubble Sort. There is a pop-up menu near the upper right corner of the applet that you can use to select the sorting method that you want to work with. Below the pop-up menu are three buttons and a checkbox that can be used to control the applet. Click on the "Go" button to let the applet perform the sort automatically, without further intervention from you. Click on the "Step" button to perform just one step in the sort. The "Start Again" button lets you start a new sort, with a randomly arranged set of bars. (The applet will also start a new sort if you select a new sorting method from the pop-up menu.)

All the sorting algorithms that you will look at use two basic operations: compare two items to see which is largest, and copy an item from one place to another. Sorting consists of these two operations performed over and over (plus some "bookkeeping," such as keeping track of which step in the sort the computer is currently performing). Sometimes, the program has to exchange, or swap, two items. It takes three copy operations to perform a swap: First, copy the first item to a special location called "Temp." Second, copy the second item into the first location. And third, copy the item from "Temp" into the first location. As the xSortLab applet performs a sort, it will tell you how may comparison and copy operations it has done so far. This information is displayed in the lower right section of the applet.

Your first task in the lab is to understand Bubble Sort, using the xSortLab applet that you launched above. The basic idea of Bubble Sort is to compare two neighboring items and, if they are in the wrong order, swap them. The computer moves through the list comparing and swapping. At the end of one pass, the largest item will have "bubbled up" to the last position in the list. On the second pass, the next-to-largest item moves into next-to-last position, and so forth. Here is a picture of what the applet will look like when it is partway through the sort. I've added some comments to help you understand what you see:



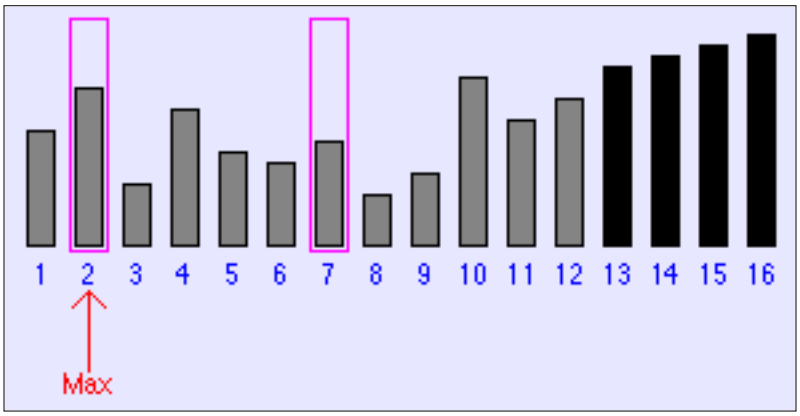
Use the "Step" and "Go" buttons in the xSortLab Applet to execute Bubble Sort. Check the "Fast" checkbox when you want to speed things up. (This option might also help you to get the broader picture of how the algorithm operates.) You should try stepping slowly through the algorithm with the "Step" button, reading each message in detail. It will also be useful for you to watch the algorithm run itself at "Fast" speed. At this speed, the applet only displays the upper message, which tells you what it is trying to accomplish in each major phase of the algorithm. In Bubble Sort, each major phase moves one item into its final position at the end of the list.

The learn how the Bubble Sort algorithm works, you will have to pay attention to what you see and think about it. You have control over the applet. Use your best judgement about how to proceed. In the exercises at the end of the lab, you will be asked to apply Bubble Sort by hand. You should make sure you understand it well enough to do this.

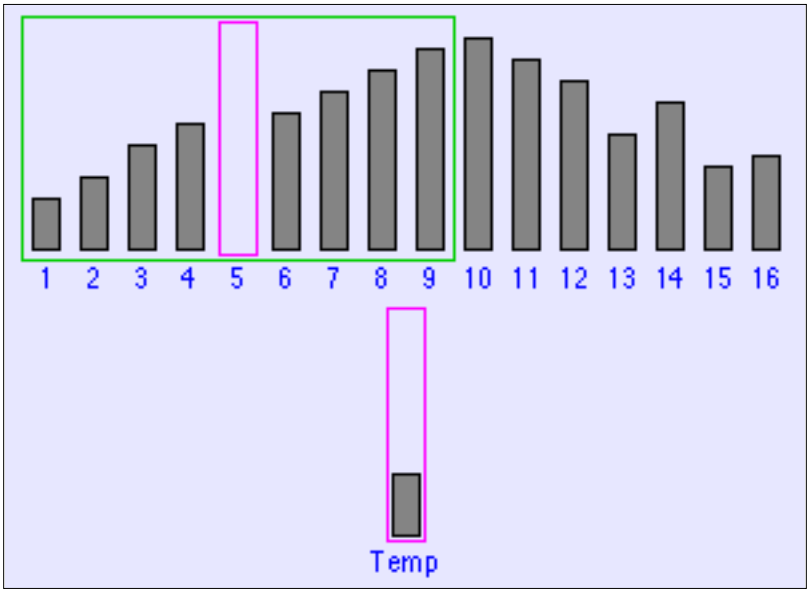
Other Sorting Algorithms

You can go on to learn about the four other sorting algorithms (or as many of them as you care to look at). The five sorting algorithms can be divided into two groups. Bubble Sort, Selection Sort and Insertion Sort are fairly straightforward, but they are relatively inefficient except for small lists. Merge Sort and QuickSort are more complicated, but also much faster for large lists. QuickSort is, on average, the fastest. Bubble Sort is the slowest. Bubble Sort is often the first -- and sometimes the only -- sorting method that students learn. Here are brief descriptions of the four remaining algorithms, with some information about what you see when xSortLab executes them:

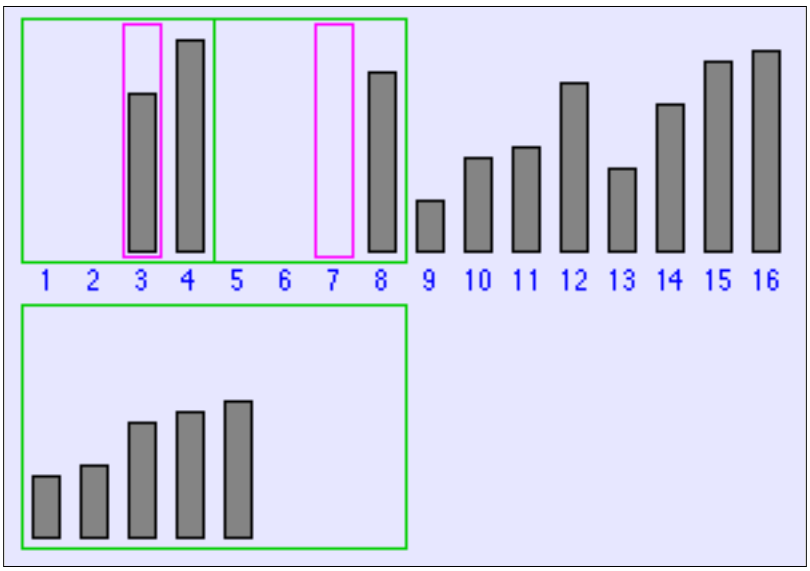
Selection Sort is probably the easiest sorting method to understand. In each major phase of the algorithm, the next largest item is found and moved into position at the end of the list. In the picture below, the four black bars have already been moved into position, and the algorithm is in the middle of the next phase. It is important to remember that the program can't just "look at all the bars and pick the biggest one" in one step, as you can. It is restricted to comparing two items at a time. To find the largest item in a list, the computer moves through the list one item at a time, keeping track of the largest item it has seen so far. After looking at all the available items, it knows which is the largest item overall. It moves the largest item into the next available spot at the end of the list by swapping it with the item that is currently occupying that spot. In the picture, during the current phase of the sort, the computer has looked at items number 1 through 7. An arrow labeled "Max" is pointing to item number 2, since that is the largest item that the computer has seen so far during this phase of the sort. The magenta-colored boxes indicate that the computer has just compared items 2 and 7. The next step will be for it to compare items 2 and 8.



In **Insertion Sort**, the basic idea is to take a sorted list and to insert a new item into its proper position in the list. The length of the sorted list grows by one every time a new item is inserted. You can start with a list containing just one item. Then you can insert the remaining items, one at a time, into the list. At the end of this process, all the items have been sorted. In the picture below, the items that are enclosed in a green box have been sorted into increasing size. Each major phase of the sort inserts one new item into this list and increases the size of the box by one. The problem is to determine where in the list the new item should be inserted. You can use the applet to see how it all works.



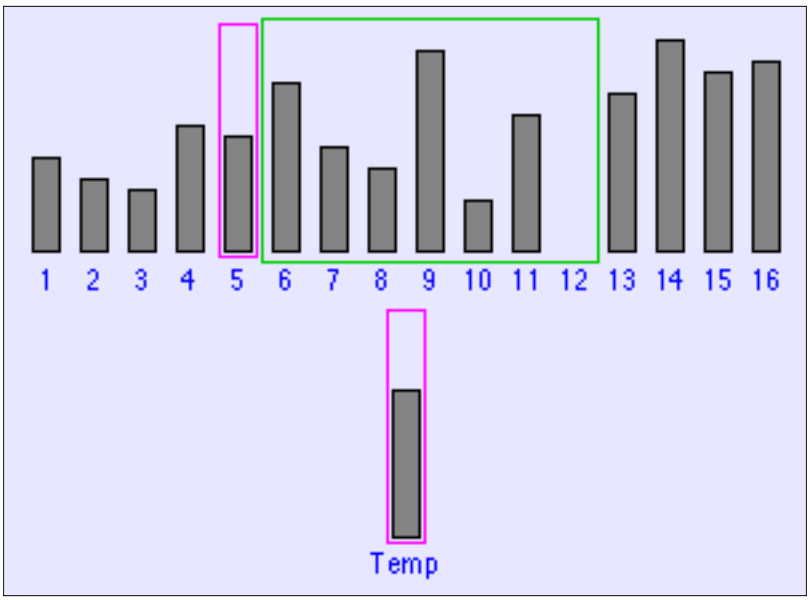
Merge Sort uses the idea of "merging" two sorted lists into one longer sorted list. To start, think of single items as being sorted lists of length one. In the first phase of the sort, these lists of length one are paired off and are merged to form sorted lists of length two. In the next phase of the sort, the lists of length two are merged to form lists of length four. Then lists of length four are merged into lists of length eight. This continues until all the items have been merged into one long sorted list. In the picture below, lists of length four are being merged into lists of length eight. The two lists that are being merged are in the top row, enclosed inside a pair of green boxes. The merged list that is being formed is in the bottom row, surrounded by a bigger green box. In the picture, items 3 and 7 have just been compared. Since item 7 was smaller, it has been moved to the bottom list.



QuickSort is the most complicated of the available sorting methods. The main idea in QuickSort is an operation called "QuickSortStep," which works like this: Remove one item from the list. Then divide the remaining items into two parts: items bigger than the removed item and items smaller than the removed items. Move all the smaller items to the beginning

of the list and all the bigger items to the end, leaving a gap in the middle. The item that was removed at the beginning is placed in the gap. Since all the items to the left are smaller and all the items to the right are bigger, the item that was placed into the gap is in its correct, final position. In the xSortLab applet, the item is colored black to indicate that it will not move again during the rest of the algorithm. The black item divides the list into two parts; each of these parts still has to be sorted. This is done by applying QuickSortStep recursively to each part. The great cleverness of QuickSortStep is in the efficient way in which it divides the list into smaller items and bigger items -- but that is easier seen than described.

The picture below shows QuickSortStep being applied to the original list of sixteen items. An item has been removed from the list and placed in "Temp." A box encloses items that have not yet been compared to Temp; as far as the program knows, Temp could end up in any of the locations enclosed by the box. Items to the left of the box are known to be smaller than Temp. Items to the right are known to be larger. Each step shrinks the box by one location, moving an item from that location to the other side of the box if necessary. In the end, there is only one location left in the box, and that is where Temp belongs.



Using the Timed Sort Mode

Now that you understand how some sorting algorithms work, the next step is to investigate how efficiently lists of items can be sorted. In this part of the lab, you will use the "Timed Sort" mode of the xSortLab applet. Select this mode from the pop-up menu at the very top of the applet (which you launched above). In the Timed Sort mode, the computer works behind the scene to sort arrays of randomly generated numbers. An array is just a numbered list of items; the size of the array refers to the number of items in the list. As the computer sorts the arrays, it displays various statistics in the large green area in the center of the applet. The statistics are updated about twice every second.

At the top of the applet, there is a text-input box where you can type the size of the arrays to be sorted. There is also a box where you can type the **number** of arrays to sort. The computer will create the number of arrays that you specify. Each array will have the size

that you specify. The computer fills all the arrays with random numbers. Then it sorts each array, one after the other. The reason for using more than one array is that for small arrays, the time it takes to sort a single array is a tiny fraction of a second. Since xSortLab can't measure very small time intervals accurately, the only way to get an accurate idea of the sorting time for small arrays is to sort a lot of arrays, measure the time it takes to sort them all, and divide the total time by the number of arrays. For larger array sizes, using more than one array is not so important (but it might still give you a more accurate measurement).

At the bottom of the applet, you will see a pop-up menu that can be used to select the sorting algorithm that is to be applied to the arrays. There is a also a "Start Sorting" button. Once you have selected the sorting method and set up the size of the arrays and the number of arrays, click on the "Start Sorting" button to begin. (If you are dealing with a large number of items, there will be a noticeable time interval before the computer starts sorting. The pause occurs while the computer is filling the arrays with random numbers.) When you click the "Start Sorting" button, the name of the button will change to "Abort." Use the "Abort" button if you want to terminate the sorting operation before the computer finishes.

The xSortLab applet displays several different statistics about the sorts its does. For this lab, you will only need the "Approximate Compute Time." This is different from the "Elapsed Time" because as it is computing the applet allows some time -- about 20% of the time available -- for other activities, such as redrawing the screen. It is only the time actually devoted to sorting that you are interested in. The compute time is approximate because it is possible for your browser or other programs on your computer to steal time from the applet. The applet might incorrectly include this time in the compute time it reports. However, if you are not doing anything else with your computer at the same time that the applet is sorting, then the reported time should be reasonably accurate. The computer measures time in units of 1/1000 of a second, and this also limits the accuracy of measurements. In particular, you should not try to use measurements that are less than, say, 0.1 seconds. Ideally, you should adjust the number of arrays so that the compute time is at least a couple of seconds.

Your task in this part of the lab is to gather timing statistics about each of the five available sorting methods. You want to measure how long it takes each method to sort arrays of various sizes. You will need this data for the exercises at the end of the lab, so you should record the data as you work. For each experiment that you do, record the array size, the number of arrays, and the compute time that is reported by the applet.

You should apply each method to arrays of at least the following sizes: 5, 10, 100, and 1000. In addition, you should apply Merge Sort and QuickSort to arrays of size 10000. You might also want to try Merge Sort and QuickSort on arrays with 100000 items. And you might try Bubble Sort, Selection Sort, and Insertion Sort with 10000 items. For the largest array sizes, it will be good enough to sort a single array. For the smaller array sizes, you will have to set the number of arrays to be rather large to get a decent measurement. Don't be afraid to sort 10000, or even 100000, arrays of size 5. Use your judgement. (If the arrays require more memory than your computer has available, you should just get an error message. However, this has **crashed** many systems that I tried it on.)

N^2 and N*log(N)

In the previous section of the lab, you measured the computation times for the five sorting algorithms on arrays of different sizes. You probably noticed that QuickSort and Merge Sort are much faster than the other three algorithms, except for very small arrays. This final section of the lab gives a more rigorous mathematical form to this observation.

Some sorting algorithms exhibit what is called running time on the order of N squared. They are also called, more briefly, N^2 algorithms. The N here is the number of items being sorted. To say that an algorithm is of the order of N^2 means that the running time of the algorithm for an array of N items is given approximately by $K*N^2$, where K is some constant number. (The approximation tends to become more exact as N gets bigger.) Different algorithms have different values for K. Furthermore, if the same algorithm is run on different computers, each computer will give a different value of K. (Saying that K is a "constant" means that for a given algorithm running on a given computer, there is one value of K that will work for **any** array size N.)

Bubble Sort is an example of an N^2 algorithm. You should be able to use the data you collected in the previous section of the lab to calculate a K-value for Bubble Sort running on your computer. For example, suppose that it took 8.205 seconds to sort 5 arrays of size 1000. Here N, the array size, is 1000. Let T be the time it took to sort one array of size 1000. T can be computed by dividing the total time, 8.205, by the number of arrays, 5. This gives T = 1.641 seconds. Now, T is supposed to be given approximately by the formula

$$T = K * N^2$$

We know that N is 1000 and T is 1.641, so the equation becomes

$$1.641 = K * 1000^2$$

You can solve this for K to get

$$K = 1.641/(1000^2) = 0.000001641$$

More generally, a value for K can be calculated from the formula K = T/(N*N). Of course, this is only supposed to be an approximation. But if you repeat the experiment several times with different array sizes, and calculate a value for K from each experiment, you should get values of K that are fairly close to one another. If this does not happen, then the algorithm you are looking at is probably not an N^2 algorithm. (Remember, though, that there is room for some "experimental error" because of the difficulty of measuring compute time. Also, keep in mind that the values calculated for K tend to get more accurate as N increases. A very small value of N might give a poor approximation for K.)

Not every sorting algorithm is an N^2 algorithm. Some of them are N*log(N) algorithms. This means that the running time of the algorithm is given approximately by K*N*log(N), where N is the size of the array and K is a constant. The function log(N) is the logarithm of N. You can compute this function using the "log" button on your calculator. (There are

actually many different logarithm functions, and you can use any of them as long as you are consistent. If you use a differernt logarithm function, you'll just get a different value for K. The log function on your calculator is almost surely the "common" or "base-10" logarithm, and that's the one that I will use in this lab.) You don't need to know anything about logarithms, except that when N is large, N*log(N) is much smaller than N^2. This means that for large values of N, an N*log(N) algorithm will run much faster than an N^2 algorithm. Based on this, you can probably guess which of the five algorithms are N^2 algorithms and which are N*log(N) algorithms.

Of course, you can use the measurements you made in the previous section of the lab to compute an approximate value for K. If T is the time it takes to sort an array of size N, then an approximate value for K is given by the formula:

$$K = T / (N*\log(N))$$

You will do some calculations of this kind in the exercises below.

Exercises

Exercise 1: Each phase of Insertion Sort inserts an item into its correct location in a sorted list. Describe in detail how this insertion is accomplished. Try to give an **algorithm** -- an unambiguous, step-by-step procedure -- for inserting a new item into a sorted list. You can get the information you need by running the xSortLab applet and seeing how it performs this task.

Exercise 2: Suppose that Bubble Sort is applied to the following list of numbers. Show what the list will look like after each phase in the sort:

73 21 15 83 66 7 19 18

Exercise 3: Suppose that Selection Sort is applied to the list of numbers given in Exercise 2. Show what the list will look like after each phase in the sort:

Exercise 4: Suppose that Merge Sort is applied to the following list of numbers. Show what the list will look like after each phase in the sort:

73 21 15 83 66 7 19 18 21 44 58 11 91 82 44 39

Exercise 5: Suppose you have a list of names. You could sort the list into alphabetical order either by first name or by last name. Suppose that the list is already sorted by last name. For example:

```
Phil  Doe
Jane  Doe
Fred  Doe
Bill  Jones
Jane  Jones
Mary  Smith
```

```
Fred  Smith
Jane  Smith
```

Now, suppose that you take this list and sort it into alphabetical order by **first name**. In the new list, Jane Doe, Jane Jones, and Jane Smith will be grouped together. The question is, will they be in that order. That is, will people with the **same** first name still be in alphabetical order by last name? For some algorithms, the answer is yes. Other algorithms, however, can mess up the relative order of a group of people with the same first name. A sorting algorithm that will always preserve the order of people with the same first name is said to be stable. Is Selection Sort a stable algorithm? Why or why not? How about Bubble Sort? Why or why not? (You might want to apple Bubble Sort and Selection Sort to the above list to see what happens.)

Exercise 6: Consider the data that you collected on the compute time of the five sorting algorithms. For arrays of size 100, how do the algorithms rank according to speed? How about for arrays of size 5? The two orderings should be very different. How can this be? If one algorithm is faster than another for arrays of size 5, why shouldn't it be faster for arrays of size 100?

Exercise 7: Selection Sort is an N^2 algorithm. You should be able to use data you collected in this lab to compute the value of K in the equation $T = K*N^2$ for Selection Sort running on your computer. (Recall that N here is the array size and that T is the time it takes to sort one array of size N. Also remember that the equation is only approximately true.) Make a table showing the following information for each array size, N, for which you collected data: The array size (N); the number of arrays you sorted; the total computation time to sort the arrays; the computation time for one array (T); and the computed value T/(N*N) which is an approximation for K.

Based on the data in your table, what is your best guess for the actual value of K? Explain your reasoning.

For your convenience, here is an applet that you can use to do the computation. Enter the array size, the number of arrays, and the total computation time in the three boxes at the top. The applet will calculate the computation time per array and the value of T/(N*N). It will do the calculation when you click the "Compute" button and whenever you press return while typing in one of the boxes.

(Sorry, your browser doesn't do Java!)

Exercise 8: QuickSort is an N*log(N) sorting algorithm. You should be able to use data you collected in this lab to compute the value of K in the equation T = K*N*log(N) for QuickSort running on your computer. Make a table just like you did for Exercise 7, but use the value of T/(N*log(N)) instead of T/(N*N). You can use the above applet to do the calculation. Based on the data in your table, what is your best guess for the actual value of K? Explain your reasoning.

Exercise 9: Is Insertion Sort an N^2 or an N*log(N) algorithm. You can probably guess, but how can you be sure? From the data you collected in the lab, compute both T/(N*N) and

T/(N*log(N)) for various values of the array size N. Based on the answers, you should be able to tell whether Insertion Sort is an N^2 or an N*log(N) algorithm. Which is it? Explain your reasoning. Do the same thing for Merge Sort.

Exercise 10: In Exercise 7, you determined the value of K for selection sort in the equation $T = K*N^2$. Now that you know the value of K, you can use the equation to predict the running time T for any array size N -- even for arrays that are two big for you to experiment with. Use the equation to predict the value of T when N is equal to one billion. (One billion is 1000000000 or 10^9.) This is the amount of time that your computer would take to sort one billion items using Selection Sort. The equation gives T in terms of seconds, but you should convert your answer to years.

Similarly, you can use the equation T = K*N*log(N) and the value of K that you found in Exercise 8 to predict the running time T for QuickSort for arrays of any size N. What is the prediction for the running time of QuickSort on an array of one billion items? Give your answer in terms of hours.

Comment on the answers to the two parts of this exercise.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xTurtle Lab 4: Multiprocessing

A CENTRAL PROCESSING UNIT EXECUTES A PROGRAM one step at a time, fetching each instruction from memory and executing it before going on to the next instruction. In many cases, though, a problem can be broken down into sub-problems that could be solved at the same time. In parallel processing, several CPUs work simultaneously on a problem, each one solving a different sub-problem. This is one of the major techniques for speeding up the execution of programs.

Even when only one processor is available, it is sometimes natural to break down a program into parts that can be executed simultaneously. Multitasking can be applied to divide the single processor's time among the various parts of the program. The program won't be executed any more quickly, but the use of parallel processing "abstractions" might make the program easier to write.

In this lab, you will use the multitasking capabilities of the xTurtle programming language. In this language, it is possible to split (or fork) a process into several processes that will all execute simultaneously and independently. Each process will have its own turtle visible on the screen, so you can actually see what is going on. Although you will be seeing only simulated parallel processing, it would be at least theoretically possible for each process to run on its own CPU.

The background material for this lab is covered in Sections 10.1 and 10.2 of The Most Complex Machine. It would be useful, but not essential, for you to read that material before working on the lab.

This lab includes the following sections:

Start by clicking this button to launch xTurtle in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Multiple Turtles

After launching xTurtle, select the first sample program, "Bugs.txt," from the pop-up menu at the top of the xTurtle window. Run the program by clicking on the "Run Program" button. You will see ten turtles wandering around on the screen. This is a very simple program, but it illustrates the basic multiprocessing command in xTurtle, the fork statement.

The statement `fork(10)` causes a single process to split into ten processes. Any commands in the program that follow the fork statement will be executed by each process independently and simultaneously. In the Bugs program, each of the ten processes goes into a loop that sends its turtle on a random walk, and so you see all ten turtles wandering aimlessly on the screen. Each turtle follows the same program, but each turtle chooses different random numbers and so each turtle follows its own random path. The first two exercises at the end of the lab ask you to add a few modifications to this program.

When a process is forked, all of the processes that are created start out in exactly the same state, with one small exception. The xTurtle language has a predefined variable named ForkNumber. This is a read-only variable; that is, you can test the value of this variable but you can't change its value. Each of the processes created by a fork statement gets its own value for ForkNumber. For the first process, the value of ForkNumber is 1, for the second the value is 2, and so forth. To test this, try executing the following commands. Type them on a single line in the input box at the bottom of the xTurtle window, and then click the "Do It" button:

```
fork(5)  TellUser("ForkNumber = #ForkNumber")
```

When the computer executes the TellUser command, it substitutes the actual value of ForkNumber for "`#ForkNumber`". Make sure you understand what happens. (You have to type both commands in the input box on one line. If you execute "`fork(5)`" by itself, two processes will be created, but both processes will "die" before you get a chance to type in the next command.)

In the sample program "ParallelSpectrum.txt", the ForkNumber is used in the program in several calculations. Each process does the same calculation, but since different processes have different values for ForkNumber, the result will be different in each process. For example, "`turn(4*ForkNumber)`" will make Turtle #1 turn 4 degrees, Turtle #2 turn 8 degrees, Turtle #3 turn 12 degrees, and so on.

The sample program "TwoTasks.txt" shows how ForkNumber can be used to make several turtles do several completely different tasks. When you run this program, two processes are created with a fork command. Both processes execute an IF statement of the form:

```
IF  ForkNumber = 1  THEN
   { do one thing }
ELSE
   { do something else }
END IF
```

The first turtle, with a ForkNumber of 1, does one task while the second turtle, with a ForkNumber of 2, does another, completely different task. You are asked to do something similar in Exercise 3 at the end of the lab.

Two other sample programs, "ParallelSnowflake.txt" and "Circles.txt," show that a program can contain more than one fork command. In these programs, the first fork command creates several processes. Each of these processes then goes on to execute a second fork command. When this happens, **each** process splits into several processes. You can see this clearly when you run "ParallelSnowflake.txt," which creates 216 processes with a sequence of three `fork(6)` commands.

The "Circles.txt" sample program illustrates another important fact. It shows how a declare statement works when it occurs after a fork. Every process will execute the declare statement separately, so that every process will have its **own** copy of the variable. Thus, the variable can be assigned different values in different processes. You will need to understand the "Circles.txt" program in order to do exercise 4 at the end of the lab.

On a separate web page, I've provided a set of six sample programs that use xTurtle multitasking to draw some interesting "tilings" of the plane. These are pictures that would be more difficult to draw without multitasking. If you like pretty pictures, you might want to take a look.

Scheduling

The operating system of a multitasking computer is in charge of scheduling all the processes. That is, it determines which process gets to run next and how long it will be permitted to run. The CPU, under the control of the operating system, will execute one process for a while, then switch to another, then to another, and so forth. In xTurtle, the scheduling is done by the xTurtle applet, but the objective is to simulate the way things are done in an actual multitasking computer system.

When you watch the execution of the "ParallelSnowflake.txt" or "Circles.txt" sample program, you'll see that the processes do not all run at exactly the same speed. As the system switches its attention from one process to another, there is some randomness in the amount of time that is spent on each process. The xTurtle program is set up to run this way by default, since it's the way real multitasking systems work.

However, the xTurtle applet has an option to run all the processes at exactly the same rate of speed. To turn this option on, click on the "Lock Step" checkbox in the bottom right corner of the xTurtle window. When this box is checked, all the turtles on the screen will move in "lock step." While this might not be an accurate simulation of multitasking, it can be pretty to watch. You might want to try executing some of the sample programs with this option turned on.

Shared Variables

In all the examples you have seen so far, the multiple processes are completely independent. The various turtles go about their business without interacting with the other turtles in any way. (This is not quite true, since the turtles have to share the same screen. You might have noticed that one turtle will sometimes wipe out the image of another turtle temporarily.)

Things can get more interesting when the processes have to communicate with each other. In xTurtle, processes communicate through shared variables. When a variable is declared **before** a fork command, there is only **one** copy of that variable, which is shared by all the processes. If any of those processes changes the value of the variable, then all the other processes can see the new value. This is the only form of communication between processes that can occur in xTurtle.

As explained in The Most Complex Machine, great care must be taken when shared variables are used for communication, so that one process does not change the value of a variable while another process is using that value. A process must obtain exclusive access to a shared variable while it is using that variable. This is the mutual exclusion problem. In xTurtle, the grab statement is provided to make mutual exclusion possible. A grab statement takes the form

```
GRAB <global variable name> THEN
   <statements>
END GRAB
```

Only one process at a time is allowed to "grab" a given variable. When a process comes to a grab statement, the computer checks to see whether another process has already grabbed the variable. If so, then the second process must wait until the first process releases its lock on the variable by finishing the execution of its grab statement. Only then is the second process allowed to grab the variable and execute the statements in its own grab statement. (Of course, this can cause big problems if a process grabs a variable and doesn't release it. Other processes that want to grab the variable will never get a chance to run at all.)

The statements inside a grab statement are called a critical region. As long as access to shared variables is confined to critical regions, processes can use the variables to communicate in relative safety.

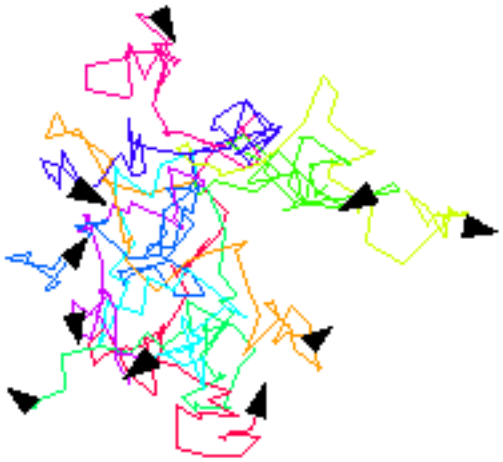
Even with the grab statement, communication among processes can still be very complicated. A relatively straightforward example can be found in the sample program "SynchronizedRandomWalk.txt." Select this program from the pop-up menu, read the comments, and run the program. You will see two turtles executing identical random walks. One of the turtles selects a random angle to be used in the random walk and records it in the shared variable, `angle`. The other turtle reads the value from the shared variable and uses it. A second shared variable, `control`, is used in a grab statement to control access to `angle`. Exercise 6 asks you to modify this program so that more than two processes are involved.

Another example of shared variables is given in the sample program "ThreeNPlusOneMax.txt." Read the comments in the program and run the program. It will take a few seconds to run, and you won't see anything happening until the end, when the program reports the value that it has computed. In Exercise 7, you will work with a similar example.

The ThreeNPlusOneMax program also illustrates what happens when a fork command is used inside a subroutine: At the end of the subroutine, all the processes are "rejoined" into one process before the subroutine terminates. After the subroutine finishes, there is only one process to carry on. The forking is part of the subroutine's black box; it has no effect outside the subroutine. In the sample program, there is only one process to execute the TellUser statement at the end of the program.

Exercises

Exercise 1: Modify the sample program "Bugs.txt" so that each bug will leave a trail behind it as it wanders randomly around on the screen. Each of the trails should be a different color. The screen should look something like this after the program has run for a while:



Exercise 2: In the sample program "Bugs," ten "bugs" wander around on the screen. Real bugs, though, are born and die. Add "birth" and "death" to the Bugs program. Add birth by programming a one-in-twenty-five chance that a bug will split in two, each time it moves. (You can program a one-in-twenty-five chance by checking whether RandomInt(25) is 1.) Similarly, there should be a 1-in-25 chance that the bug will die.

To program death, you will need to know about another built-in xTurtle command, KillProcess. When a process executes the command KillProcess, it dies. The turtle for that process disappears from the screen. (Note that this command differs from Halt. If any process executes a Halt command, the entire program halts and so **all** the turtles die.)

Exercise 3: The sample program "TwoGraphs.txt" draws the graphs of two functions, one after the other. Instead of a single turtle that draws the graphs one after the other, the program could use two turtles that draw the graphs at the same time. Modify the program so that it does this. You only need to do a bit of work on the half-dozen or so lines at the very

bottom of the program.

Exercise 4: Write a program that uses two fork(9) statements to draw a multiplication table like this one on the screen:

```
1   2   3   4   5   6   7   8   9
2   4   6   8  10  12  14  16  18
3   6   9  12  15  18  21  24  27
4   8  12  16  20  24  28  32  36
5  10  15  20  25  30  35  40  45
6  12  18  24  30  36  42  48  54
7  14  21  28  35  42  49  56  63
8  16  24  32  40  48  56  64  72
9  18  27  36  45  54  63  72  81
```

The entry in row number *row* and column number *col* is *row*col*. Your program will be similar in outline to the sample program, "Circles.txt." You'll need to use the DrawText command to write each number on the screen. Recall that if *num* is a variable, then the command DrawText("#num") will output the value of *num* at the current turtle position.

Exercise 5: A recursive subroutine for drawing binary trees was covered in xTurtle Lab 3. Binary trees can also be drawn using multitasking. Each "fork" in the tree is represented by a fork command in the program. Here is a recursive subroutine that uses multitasking to draw a binary tree:

```
SUB tree(size, complexity)
   IF complexity > 0 THEN
      forward( size/2 )
      fork( 2 )
      IF ForkNumber = 1 THEN
         turn( 45 )
      ELSE
         turn( -45 )
      END IF
      tree( size/2, complexity - 1 )
   END IF
END SUB
```

Implement this subroutine in a program that draws binary trees. Then modify the subroutine to add some color and an element of randomness. For example, instead of being always equal to two, the number of branches could be random. Instead of just dividing the size by 2, you could multiply it by a random amount. Try to make your program draw "bushes" that bear at least some resemblance to real bushes.

Exercise 6: Modify the sample program "SynchronizedRandomWalk.txt" so that instead of showing two turtles moving in identical random walks, it shows *five* turtles moving in identical random walks. Each of the turtles should start from a different position and should draw in a different color. The hard part of this exercise is passing off the "control" from one

turtle to the next. You have to make sure that the *control* goes through the values 1, 2, 3, 4, 5, and then back to 1 so that each turtle will get a chance to move. Just as in the original program, Turtle #1 should select the random angle. Each of the other four turtles should use the same angle.

Exercise 7: The Sample program "SumOfSquares.txt" is a **failed** attempt to write a program that computes the value of the sum

$$1^2 + 2^2 + 3^2 + ... + 25^2$$

Run the program several times. It will give a different answer each time. (If you've turned on the "Lock Step" checkbox, then you'll get the same answer each time, but it will still be an incorrect answer.) Use a grab statement to fix the program so that it gives the correct answer. See the "ThreeNPlusOneMax.txt" sample program for an example of using a grab statement for a similar purpose. Write a short essay explaining carefully what goes wrong when the grab statement is omitted and how adding the grab statement fixes the problem.

Exercise 8: Design your own parallel processing xTurtle program. Try to make a nice colorful design, either random or symmetric. Hopefully, it will be fun to watch as multiple turtles construct the picture.

Exercise 9: Is it really sometimes easier to write a program using parallel processing than to write a standard program to do the same task? Write a short essay explaining and defending your opinion.

--David Eck (eck@hws.edu), Summer 1998

Labs for The Most Complex Machine

Tiling Examples for xTurtle

THE COPY OF xTurtle on this page has six sample programs written by Kevin Mitchell. Each of these programs uses xTurtle's multitasking abilities to draw a tiling of the plane. After launching xTurtle, choose an example from the pop-up menu at the top of the window and click the "Run Program" button.

Click this button to launch xTurtle in its own window:

(Sorry, your browser doesn't do Java!)

Back to the Rest of the Lab

Labs for The Most Complex Machine

xModels Lab 1: Two-D Graphics and Animation

IMAGES ARE OFTEN CREATED ON COMPUTERS in a two step process. First, a geometric model of a scene is created, then the scene is rendered using realistic coloring and lighting effects. In this lab and the next, you will use an applet called xModels, which deals with the model-construction stage of image creation. This lab covers two-dimensional scenes, while the next lab moves on to the problem of working with three-dimensional objects.

Complex geometric models are built up out of simpler components which are scaled, rotated and positioned in the scene using geometric transformations. Simple geometric shapes like circles and lines are used as a starting point in the modeling process. These shapes can be combined to form more complex figures that can then be combined to form even more complex scenes.

Scenes constructed from objects in this way can be used in a natural way to produce animations. An animation is just a sequence of frames in which objects move slightly from one frame to the next. When the frames are quickly displayed one after the other, the viewer perceives objects in motion.

In this lab, you will use the xModels applet to create two-dimensional geometric models and simple animations. In the process, you will learn about various two-dimensional geometrical transformations. The lab is based on the material in Section 12.1 of The Most Complex Machine. Some of the questions at the end of Chapter 12 -- and their answers in the back of the book -- are also relevant

The lab includes the following sections:

Start by clicking this button to launch the xModels applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

Introducing xModels

The xModels applet that you launched above is set up to load several sample programs. Programs for xModels are actually scene descriptions written in a scene description language. A scene description consists of a list of objects in a scene, together with geometric transformations to be applied to the objects and attributes that affect the object's appearance. (In xModels, the only attribute that an object can have is a color.) A scene description can include definitions of objects to be used later, perhaps repeatedly, in the scene. Such object definitions are very much like subroutines in a programming language. Finally, there are a few special-purpose commands in the xModels scene description language that can be used to control such things as the background color and the number of frames in an animation.

When the applet first starts up, you should see a text area containing a scene description called "Pinwheel." It describes an animation showing a revolving, colored pinwheel displayed on a black background. Don't worry for now about the scene description itself. Your first task is to learn how to get xModels to display the scene.

To see the scene represented by a scene description in xModels, you have to render it by clicking the "RENDER!" button below the text area. When you click this button, the applet first checks the scene description in the text area for errors. Assuming that no errors are found, the applet will switch to a "Graphics" panel, where it will display the scene. **Try it now.** To the right of the displayed scene is a set of controls. You should familiarize yourself with the controls. Here is a brief description of each:

- A message at the top of the column of controls shows the number of the frame that is currently displayed. An animation consists of a sequence of frames, which are displayed one after another. The "Pinwheel" animation has 121 frames, which are numbered from 0 to 120. (If you get tired of watching this message flash, try clicking on it with your mouse. This might make it stop, depending on the version of Java that you are using.)
- The "New Program" button will give you a blank text area where you can write your own scene description from scratch. Choosing "[New]" from the pop-up menu at the very top of the applet will have the same effect.
- The "Show Program" button will take you back to the scene description that produced the image or animation displayed on the graphics screen.
- The next four buttons are used to control animations. If the scene description specifies a single image, all four buttons are disabled. The "Go" and "Pause" buttons can be used to start and stop an animation. If an animation has been paused, then the "Next Frame" and "Previous Frame" buttons can be used to move though the frames in the animation, one frame at a time.
- Next, there is a pop-up menu to control the speed at which animations are played back. The default is 10 frames/second, which should be OK for most purposes. (This menu represents only a **request** for a given frame rate. The computer might not be able to display frames as quickly as you request.)
- Finally, there is a pop-up menu that determines what happens when the computer

reaches the end of an animation. In the default setting, "Loop," the computer will return to the beginning of the animation and replay, and will keep doing this until you stop it. For many animations, the first frame is the same as the last, and the "Loop" option produces what looks like continuous motion. The second option in the menu is "Back-and-Forth," which makes the computer reverse direction at the end of the animation and play it backwards. The last option, "Once Through" makes the computer stop when it reaches the end of the animation.

After you have played with the "Pinwheel" animation for a bit, it's time to learn how to write a scene description of your own.

Basic Objects and Basic Transformations

To begin working with scene descriptions, click on the "New Program" button, or select "[New]" from the pop-up menu at the top of the applet. In this section, you will work with single images. Animating an image will be a simple extension of this.

To add an object to a scene, you just have to add the name of the object to the scene description. Every time you use the name, you get a new copy of the object. If you give only the name of the object, it will appear at a default size, position, and orientation. If you want it to have a different size, position, or orientation, you have to apply one or more transformations to the object. This is done simply by listing the transformations after the object name. For example, the word "square" will add a square to the scene. It will be a small square, one unit wide and one unit high, in the center of the scene. (The entire image always includes a 20-by-20 unit region, with x-coordinates from -10 to 10 and y-coordinates from -10 to 10. Since the image might not be exactly square, it can extend farther than this in one direction or the other.)

To get a bigger or smaller square, you can apply a scale transformation. To get a square that is five times as large as the default, you have to scale it by a factor of five. This is done by saying square scale 5. Try typing the following scene description into the empty text area. The render the scene and see what you get:

```
square
square scale 5
```

You should see **two** squares of different sizes. (By the way, if you make some sort of mistake in the program, an error message will be displayed above the text area. You can click on the message, if you like, to make it go away.)

Now, using transformations is not the most obvious way of doing things, but it turns out to be remarkably powerful and even, after some experience, intuitive. In addition to the scaling transformation, the geometric transformations that you can use are translation and rotation. Scaling changes an object's size. Translation moves it. Rotation pivots it about the point (0,0) or about some other specified point. (It is important to understand that you don't actually see the object moving. The applet applies the transformations to determine the size, position, and orientation of the object before it adds the object to the scene. You only see it

in its final position. Later, you'll see how to make an animation in which objects do seem to move, but even then each individual frame of the animation is constructed as described here.)

You've seen that the transformation scale 5 magnifies an object by a factor of 5. To shrink an object, you would use a scaling operation with a factor less than 1, as for example in square scale 0.5. The scale command can also be used with two numbers. The first number gives a horizontal scaling factor and the second a vertical scaling factor. For example, square scale 2,5 specifies a **rectangle** that is 2 units wide and 5 units tall. (By the way, the comma in "2,5" is optional. The xModels applet ignores commas. You can put them in, if you like, for human readability.)

The transformation rotate 45 pivots an object through an angle of 45 degrees in a counterclockwise direction about the point (0,0). A negative angle would rotate the object in a clockwise direction. It is possible to specify a different pivot point. For example, square rotate 45 about 0.5,0.5 would pivot the square about its upper right corner, (0.5,0.5), instead of about (0,0).

The transformation translate 3,7 moves an object 3 units to the right and 7 units up. Negative numbers can be used to move the object to the left and down. If the translate command is used with just one number, as in translate -5, it indicates horizontal movement to the left or right. For convenience, there are also commands xtranslate and ytranslate for moving an object horizontally or vertically only. For example, "square ytranslate 5" produces a square translated five units upwards.

You can apply a sequence of transformations to the same object, simply by listing them all after the object's name, in the order in which they are to be applied. For example,

```
square scale 3 rotate 30 translate 5,5
```

specifies a square that is first magnified by a factor of 3, then rotated through 30 degrees, then translated 5 units horizontally and 5 units vertically. The order in which the transformations are applied can make a difference. Switching the order can produce a very different picture.

There are other basic objects besides squares. A circle is a circle of diameter one, centered at (0,0). A line is a line of length one that extends from the point (-0.5,0) to (0.5,0). A polygon can be specified by listing its vertices. For example,

```
polygon 0,0 0,5 3,4
```

specifies a triangle with vertices at the points (0,0), (0,5) and (3,4). These few simple shapes are all that you have to work with.

To make things a little more interesting, you can add color to your scenes. The default drawing color is black. You can use a color-change command to change the drawing color. The drawing color that you specify remains in effect until it is changed by another color-change command. (That is, it does not just apply to the next object.) The color-change commands include the names for the standard colors: red, green, blue, cyan, magenta,

yellow, black, white, and gray. There is an rgb command that uses three numbers to specify the red, blue, and green components of a color. The numbers must be in the range zero to one. For example, "rgb 0.7 0.7 1.0" represents a light blue color. (There is also an hsb command that lets you specify the hue, saturation, and brightness levels of a color, but I won't discuss the details here.)

For example, here is a scene description for an image scene that contains several objects of different colors:

```
red
square scale 2 translate 5,5
cyan
circle scale 5,2 rotate 30
rgb 0.4 0.2 0.2
square scale 3 rotate 30 translate -5,5
blue
polygon 0,0 0,5 3,4 translate -7,-7
magenta
line scale 5 rotate 45 translate 5,-5
```

You can give your scene a background color with the background command. The word "background" must be followed by the color you want to use for a background. For example:

```
background gray
```

or

```
background rgb 1.0 0.8 0.8
```

The background command should occur at most once in a scene. It will have the same effect no matter where it occurs.

You'll find a copy of the above scene description, with a pink background, in the sample program "SimpleObjects". You can select this example from the pop-up menu at the top of the applet. You should render the scene to see what it looks like. You should also spend some time making modifications to the scene. Try changing the color of the objects or the background. Try changing the numbers for the transformations. Add some new objects and new transformations. You should try to understand how scenes are constructed from objects, transformations, and colors.

Animation

Let's face it, static scenes are not all that exciting. Animation makes things get a lot more interesting. To make an animation in xModels, you just have to do two things to your scene description: put an animate command at the beginning, and change some of the numbers in the scene description to number ranges. Here is a simple example, which can also be found in the sample scene description, "FirstAnimation":

```
animate 30

circle scale 1:5

red
polygon  -4, -3  4, -3  -5:5, 4

blue
square  scale 7:15, 1  rotate 0:60
```

The first line of this scene description is an animate command which specifies that 31 frames will be rendered. (There are 31 frames, not 30, because the number in the animate command gives the **length** of the animation, not the number of frames. There are 30 intervals between frames, so there are 31 frames.) The remaining lines are a standard scene description, except that in some cases number ranges, such as 1:5 or 0:60, appear instead of single numbers. Where such a range appears, the first value in the range is used in the first frame, the second value is used in the last frame, and intermediate values are used in intermediate frames. If a range is used with translate, the object moves during the animation; if it is used with scale, the object grows or shrinks; and if with rotate, the object rotates through a range of angles.

The "FirstAnimation" example shows a circle that grows from size 1 to size 5 over the course of the animation. It contains a red polygon in which the x-coordinate of one of the vertices ranges from -5 to 5. And it has a blue rectangle that both changes size and rotates during the animation. Try out the example, and make sure you understand it. Try modifying it and adding to it!

Computer animation uses the idea of key frames. The key frames in an animation are specified explicitly. Other frames, which bridge the gaps between key frames, are created by the computer by interpolating between the key frames. The sample animation given above has just two key frames, one at the beginning and one at the end. In a number range such as 1:5, the first number gives the value for the first key frame, and the second gives the value for the second key frame. The xModels applet computes intermediate values for the intermediate frames.

In xModels, you can create animations with more than two key frames. I call such animations segmented animations. You can specify how many frames there are in each segment, between the key frames. For example,

```
animate  15  30  10
```

specifies that there are **four** key frames. Therefore, there are **three** segments in the animation. The first segment contains 15 frames, the second contains 30, and the third contains 10. The key frames are frames number 0, 15, 45, and 55.

When you use a number range in a segmented animation, you must specify a number for each of the key frames. For example, you could use the number range 1:3:5:2 in an animation with three segments and four key frames. (Note that the number of colons is

always equal to the number of segments.) The number range 1:3:5:2 specifies a value of 1 for the first key frame, 3 for the second, 5 for the third, and 2 for the fourth and final key frame.

You can leave out some of the numbers in the middle of a number range (as long as you don't leave out the colons). For example, the number range 1:::2 provides a value of 1 for the first frame and a value of 2 for the last frame. The computer interpolates smoothly between these two values for all the intermediate frames.

For example, here is an animation that shows a small square moving along the inside edges of a big square. You might want to type this in -- or cut-and-paste it in -- and see how it works:

```
animate 15 15 15 15
square scale 12
red
square  scale 2  translate  -5:5:5:-5:-5  5:5:-5:-5:5
```

Defined Objects and Structured Complexity

It's a long way from simple geometric shapes to complex scenes. As usual, this complexity is handled by tackling it level-by-level, with reasonable jumps in complexity from one level to the next. In the xModels applet, new objects can be defined on one level that can then be used on higher levels. An object definition takes the form of the word define followed by a scene description enclosed in square brackets ("[" and "]"). For example:

```
define wheel [
    circle
    line
    line rotate 60
    line rotate 120
]
```

When a definition like this one occurs in a scene description, it does not immediately add anything to the overall scene. It just defines the word "wheel" for the computer as a new type of object made up of the specified parts. Once this definition has been made, a "wheel" can be used like any other object. You can apply the same transformations to wheels that apply to other objects. For example:

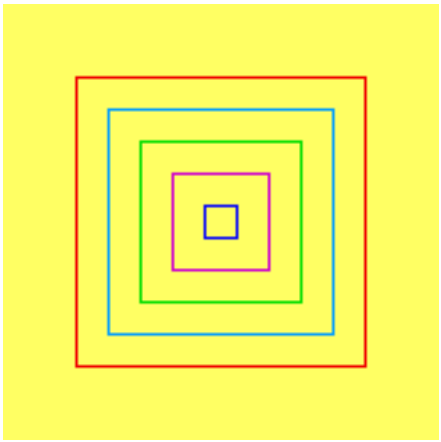
```
wheel scale 2 xtranslate -2.5
```

The word wheel becomes part of the language of xModels, on an equal basis with square and circle. It can even be used in the definitions of other objects!

The sample program named "Wagon" contains an example in which wheels are defined and used. Open the file, read it, and render it to see what it looks like. Note that the wheels on the wagon rotate. Another example is given in the sample program "Houses," which you should also look at.

Exercises

Exercise 1: Write a scene description for a still image that shows five squares of different colors nested inside one another, like this:



Use any colors you like (as long as you don't use the default color, white, for the background).

Exercise 2: This exercise builds on the nested squares that you created for Exercise 1. Turn your scene description into an animation in which each of the five squares rotates about its center. The squares should rotate at different speeds. Some of them should rotate clockwise and some should rotate counterclockwise. (The speed of rotation has to do with the range of angles through which the object rotates over the course of the animation.)

Exercise 3: The first scene you looked at in this lab was the "Pinwheel" example. The scene description for this example contains no comments. Explain what each line in this scene description does.

Exercise 4: Carefully explain the difference between

```
square scale 5,2 rotate 45
```

and

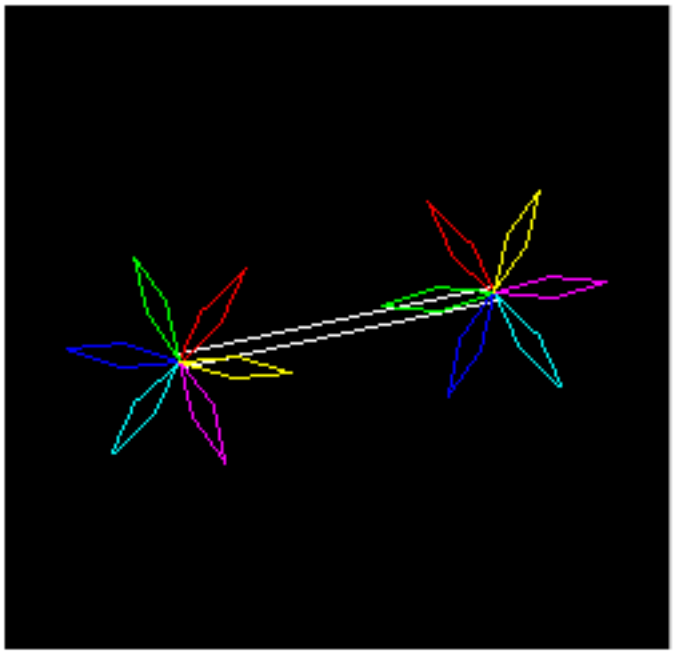
```
square rotate 45 scale 5,2
```

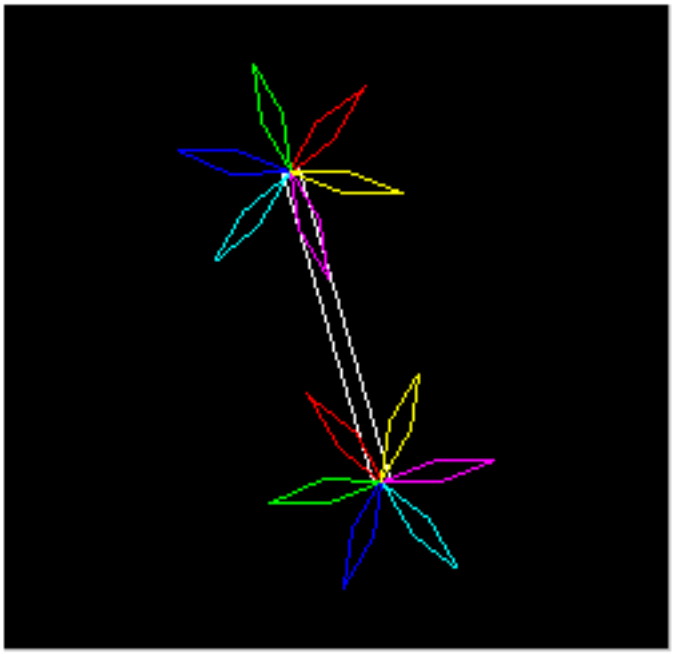
What image is produced by each command? Why are they different? How does the computer carry out each command?

Exercise 5: Use two polygons to make a large letter A, like this:

The triangle inside the top of the A is one polygon. All the other lines form another, larger polygon.

Exercise 6: This exercise builds on the "Pinwheel" example. Use the definition of the pinwheel object form that example. Make a "double_pinwheel" object consisting of a long rectangular bar with a **rotating** pinwheel at each end. Use your double_pinwheel object in an animation that shows a double pinwheel rotating about its center. Here are two frames from such an animation:





Exercise 7: The sample scene description called "Bounce" contains an animation in which a red circle seems to bounce back and forth between two edges on a square. Try it. Add a second circle bouncing between the other two edges. The second circle should be blue. This can be done by adding two short, simple lines at the end scene description (one for the color and one for the ball). Think about geometric transformations!

Once you have the two bouncing circles inside the square, make the square and the two circles into a single object by putting them into a "define" command. Make an animation in which the combination object rotates and changes size.

Exercise 8: Write an essay comparing object definitions in xModels with subroutine

definitions in a programming language such as xTurtle. How are they similar, and how are the different. (Among other things, you should discuss the fact that object definitions in xModels do not have parameters. What are the consequences of this?)

Exercise 9: Write an essay discussing how objects created by the "define" command can be used to create complex scenes. Why is the ability to define objects so important to dealing with complexity? (Keep in mind that once an object has been defined, it can be used as a component in another, more complex object. Your essay should discuss this fact.)

Exercise 10: Design your own animated scene using xModels. Try to be creative and/or aesthetic.

--David Eck (eck@hws.edu), Summer 1997

Labs for The Most Complex Machine

xModels Lab 2: Adding the Third Dimension

REAL OBJECTS EXIST in three dimensions, and realistic computer graphics must be able to deal with three-dimensional objects. In the previous lab, you used the xModels applet to create two-dimensional scenes and animations. However, the applet can also use three-dimensional objects and three-dimensional geometric transformations. In this lab, you will add the third dimension to your work with xModels.

Before starting this lab, you should make sure that you are familiar with using the xModels applet to write programs and to view scenes. You should understand the basics of the scene description language, including the use of the define command to create new objects. This material was covered in the previous lab and is not repeated here.

This lab includes the following sections:

Start by clicking this button to launch the xModels applet in its own window:

(Sorry, your browser doesn't do Java!)

(For a full list of labs and applets, see the index page.)

In the Direction of Z

Geometric models can be constructed in three dimensions, as well as in two. Even for three-dimensional models, of course, the image still has to be displayed on a two-dimensional computer screen, but the model of the scene exists, at least in our imagination and the computer's memory, in three dimensions. Once you understand the basic ideas of geometric modeling in two dimensions, the step up to three dimensions is not so hard. Just remember that in addition to the x and y-coordinates that you are used to, there is also a z-coordinate.

Think of z as measuring distance in front of the computer screen (or, if z is negative, behind it). The x-axis is a horizontal line on the computer screen. The y-axis is a vertical line on the screen. The x- and y-axes intersect at a point called the origin, which is at the center of xModels' display area. The z-axis is a line perpendicular to the screen. Like the other two axes, the z-axis goes through the origin. The positive direction of the z-axis points at you out

of the screen. Larger positive values are closer to you. Points behind the screen have negative z-values. Any point in three dimensions has three coordinates, giving its position with respect to the x, y, and z axes.

In the previous lab, you worked with two-dimensional objects that lay entirely in the xy-plane. In this lab, we add a z-coordinate, and you can work with objects that lie anywhere in xyz-space. The image that you see on the screen is obtained by projecting the three-dimensional objects onto the two dimensional computer screen. By default, objects appear as they would from the point (0,0,20), which is on the z-axis 20 units in front of the screen. (Any objects or parts of objects that are **behind** this point are not shown at all.) The number 20 is called the viewing distance. You can set a different viewing distance for a scene by using the viewDistance command in your scene description. For example, the command

viewDistance 50

sets the viewing distance to 50. This means that the objects are projected onto the xy-plane from the point (0,0,50). The viewDistance can only occur once in a scene description, and it will have the same effect no matter where it occurs.

(This description of viewing distance is somewhat misleading, since xModels will display the same region of the xy-plane, no matter what the viewing distance. You might expect to see a larger region of the xy-plane if you "back up," but this doesn't happen. Things will, however, look more squashed in the z-direction. Objects that actually lie in the xy-plane will look exactly the same no matter how the viewing distance is set.)

The xModels applet, which you launched above, is set up to load several sample programs. When it first starts up, it should be displaying a program called "Flaps." Click on the "RENDER!" button to see what this animation looks like. The object in the animation is made from eight rectangles, but the rectangles have been rotated out of the xy-plane so that the object as whole is three-dimensional. To understand the scene description, you'll have to read the next section of the lab. However, you might want to try adding a viewDistance command to the scene description. Try several different distance settings. In particular, you should try viewDistance 5, which places the viewing point **inside** the object. Try to understand what you see.

Three-Dimensional Objects and Transformations

The basic building blocs of three-dimensional models in the xModels applet still include line, circle, square and polygon, which start out as two-dimensional objects in the xy-plane but which can be translated or rotated out of that plane. (You saw an example of this in the "Flaps" program above.)

There are also a few three-dimensional basic objects in xModels: A cube is a one-by-one-by-one unit cube, centered at the point (0,0,0). A cone is a cone that just fits inside the unit cube, and a cylinder is a similarly-sized cylinder. These objects are pretty small, but they can of course be scaled. For example, if you want to see what a cone looks

like, try a scene description that contains the command:

cone scale 10

There is also a polygon_3D command, which constructs a polygon from a list of three-dimensional points instead of two-dimensional points. Later in the lab, you'll see two other commands, lathe and extrude, which can produce interesting objects.

Just as in two-dimensions, the geometric transformations in three dimensions include scaling, translation, and rotation.

The scale command can be used with one, two, or three parameters. With one parameter, as in scale 5, it magnifies an object equally in all three directions. If three parameters are given, they specify different scaling factors in each direction. For example,

```
cube scale 5 0.5 2
```

makes a rectangular solid that is 5 units long in the x-direction, 0.5 units high in the y-direction, and 2 units thick in the z-direction. If a scale has only two parameters, then the same scaling factor is used in the z-direction as in the y-direction. (This version is provided mostly for use in two dimensions, where any scaling in the z-direction has no effect in any case.)

The translate command can also be used with one, two, or three parameters. In its most general form, with all three parameters, it specifies motion in all three directions. With one parameter, it specifies movement in the x-direction only, and with two parameters, it specifies movement in the x-direction and y-direction only. There is a command ztranslate for moving an object in the z-direction only. For example, "square ztranslate 5" represents a square that has been moved 5 units forward out of the screen. (You already saw the analogous commands xtranslate and ytranslate in the previous lab.)

Scaling and translation are pretty much the same in three dimensions as in two, but rotation in three dimensions is another matter. In two dimensions, an object is rotated about a point. A three-dimensional object must be rotated about a **line**, like a top spinning about its axis. In xModels, there are three rotations commands, for rotation about the x-axis, the y-axis, and the z-axis, respectively. The commands are called xrotate, yrotate, and zrotate. (The zrotate command is actually the same as the plain old rotate.)

It is easiest to understand rotation in three dimensions by looking at examples. Enter the following simple scene description into a new program in xModels, and render it:

```
animate 60
square scale 10 yrotate 0:360
```

You will see a square rotating about its vertical axis. Note that the edge of the square that is farther from you looks shorter, as it should. Try the same scene with the square changed to a cube. Try using xrotate and zrotate instead of yrotate.

Next, try the following example, which illustrates a translation in the z direction followed by a rotation:

```
animate 60
square scale 5 ztranslate 8 yrotate 0:360
```

The ztranslate command moves the square 8 units forward towards you. The yrotate command then sends it circling away from you and back. Try using xtranslate instead of ztranslate. Try xrotate and zrotate in place of yrotate. Make sure that you understand what you are seeing.

Of course, a lot of the power of xModels comes from its ability to construct complex, hierarchical models using the define command. You can use define to define three dimensional objects, just as you used it in two dimensions. You'll work with some defined objects in the exercises at the end of the lab.

The example program "NestedSquares3D" uses several levels of object definitions to create what might be the ultimate "nested squares" example. Each square rotates on its own, but it also takes part in the rotation of all the squares in which it is nested. I find the result visually interesting, kind of like a mobile.

Lathing and Extrusion

Modeling real objects (like cars or faces) in 3D requires that they be approximated with large numbers of polygons, perhaps hundreds of polygons in one object. You won't want to do anything so complicated with xModels. But xModels does have two ways of producing certain types of complicated objects. The methods are called lathing and extrusion. These are standard operations in three-dimensional graphics. The idea is similar in each case: a specified figure is copied several times, and the vertices of the copies are connected with line segments. For lathing, the copies are obtained by rotating the original around the y-axis. For extrusion, the copies are obtained by translating the original in the z-direction. In xModels, the commands for performing lathing and extrusion are lathe and extrude.

The lathe command takes a sequence of points in the xy-plane and connects them with line segments. It then takes the resulting figure and makes several copies of it by rotating it around the y-axis. The first parameter of the lathe command indicates the number of copies. This is followed by a list of the x and y coordinates of the points in the xy-plane. This is easier to understand if you see an example. The command "lathe 12 3,3 7,-3" says that the line from the point (3,3) to the point (7,-3) is to be copied 12 times. The copies are evenly spread out around the y-axis. With 12 copies, they are spaced every 30 degrees. The endpoints of the 12 lines are then connected, giving an object that looks like a lampshade. To see it, type in the following scene description:

```
animate 60
lathe  12  3,3  7,-3
          yrotate 0:360
```

If you change the 12 to a 4, only 4 copies of the line will be made. The resulting object looks like a truncated pyramid. You could also try adding another point or two to the end of

the lathe command.

The extrude command is similar to lathe. It also takes a sequence of points in the xy-plane, connects them with line segments, copies the resulting figure a specified number of times, and then joins the vertices of the copies with more line segments. However in this case, the copies are translated in the z direction rather than rotated around the y-axis. The copies are stacked up in front of and behind the screen. Each copy is separated from the next by a distance of one unit. The simple example "extrude 10 -1,0 1,0" will create a ladder-like object with 10 rungs. (It will lie along the z-axis, so if you want to see it, you should apply an xrotate 90 to it.)

The sample program "LatheAndExtrude" contains several examples of lathing and extrusion. You should read the scene description and render it.

Exercises

Exercise 1: Make an animation showing three cubes of different colors nested inside one another. One cube should rotate about the x-axis, one about the y-axis, and one about the z-axis.

Exercise 2: The "Flaps" example program shows an abstract sort of paddle wheel that rotates about the y-axis. Paddle wheels are supposed to rotate vertically. Modify the "Flaps" example so that the "wheel" is vertical and rotates about the x-axis. This is a small modification, if you do it right! You don't have to modify the definitions of the "flap" or the "paddles" object.

Exercise 3: Define an object that consists of a large cube, ten units on a side, with a circle on each of its six faces. Each circle should be a different color. You will have to work to get all the circles properly oriented and positioned. You'll need some commands along the lines of "circle scale 8 xrotate 90 ytranslate 5". Make an animation that shows the entire object rotating as a whole.

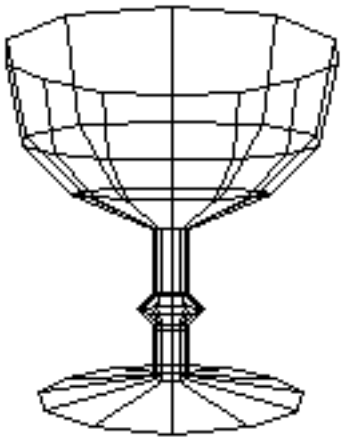
Exercise 4: Consider the following four animations. (The command "viewDistance infinity" indicates projection from a very large, effectively infinite distance. This is also called parallel projection.)

```
(1)   animate 60
      viewDistance 20  ; Note: This is the default value.
      cube scale 10 yrotate 0:360

(2)   animate 60
      viewDistance 100
      cube scale 10 yrotate 0:360

(3)   animate 60
      viewDistance infinity
      cube scale 10 yrotate 0:360
```

```
(4)   animate 60
      viewDistance 5
      cube scale 10 yrotate 0:360
```

Explain what is happening in each animation, and why each animation looks the way it does. Explain the differences among them.

Exercise 5: Use lathing to create a "goblet" shape, like this one:

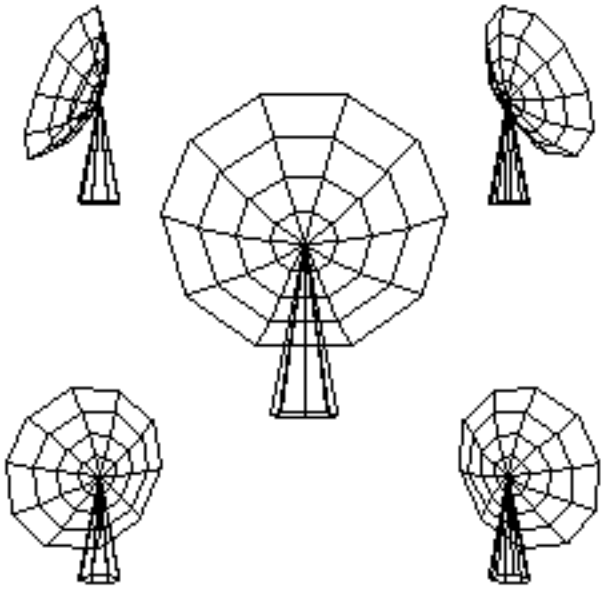


Exercise 6: The sample program "Wagon," which was also used in the previous lab, shows a simple two-dimensional wagon. Convert it into a three dimensional wagon. Use four wheels. Use a cube instead of a square as the starting point for the body of the wagon. Try out your wagon in an animation with the command

```
wagon xtranslate -10:10 yrotate 60 xrotate 15
```

You should see a wagon that looks like it's driving away from you up a hill.

Exercise 7: The picture below shows a "radio telescope" from several different viewpoints. This telescope is made of two pieces. The base is a cone. You can use a cone object, or you can make it by lathing a single line segment. The dish of the telescope is not a cone, since it's cross section is curved. You can make the dish by lathing a curve consisting of several line segments connecting (0,0) to (4,2). The dish has to be rotated and translated into position after it is created. Define a telescope object as described. Then make an animation that shows three telescopes rotating back as forth (sweeping the skies for signs of new galaxies or extraterrestrial life).



Exercise 8: Use the xModels applet to make a three-dimensional image (not an animation) of your choice. Your scene description should include at least one object definition. You might, for example, try to make a house, a sailboat, a robot, or a space station.

Exercise 9: Use the xModels applet to make an animation of your choice. Try to be creative and/or aesthetic.

Exercise 10: Write an essay discussing the differences that you have observed between two-dimensional and three-dimensional graphics. Why is three-D graphics harder? What additional skills do you need in order to work in three dimensions? Is it worth the extra effort?

--David Eck (eck@hws.edu), Summer 1997

Data Representations Applet

What can you do with thirty-two bits? Computers use strings of bits to represent all the different types of data that they have to work with, so the answer must be that you can do a lot of different things. A given string of 32 bits can represent all kinds of things, depending on the context in which it is used. That is, the same bits can encode different things, depending on how they are interpreted.

This page contains an applet that lets you see different interpretations of the same 32-bit binary number. The applet lets you type in a data value. You can select the type of data you want to enter by clicking on one of the five radio buttons. Just type your data into the input box at the top of the applet, and press return. You can also click on the 8-by-4 grid of "big pixels" at the center of the applet. The various data representations are described below.

This applet was originally written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own.

For a list of other applets and for lab worksheets that use the applets, see the index page.

Sorry! Your browser doesn't do Java!

Data Types

The Data Representation Applet shows six different interpretations for the same string of thirty-two bits. The six interpretations are: a binary number, an integer, a hexadecimal number, a real number, a string of four characters and an eight-by-four grid of pixels. Here is a short explanation of each of the six data displays.

Binary

This is the most direct display of the 32 bit binary number, showing a zero or one to represent each individual bit.

Base-ten Integer

A binary number can be interpreted as a normal positive integer (0, 1, 2, 3, 4,...) written in the "base ten". With 32 bits, you can represent 2^{32} different numbers. Usually, you want to use both positive and negative numbers. The scheme for representing negative numbers is a bit strange. It is explained in Subsection 2.2.3 of The Most Complex Machine. Using 32 bits, the integers from -2147483648 to 2147483647 can be represented.

Hexadecimal

It is difficult (for humans) to read long strings of zeros and ones. Hexadecimal numbers are a kind of shorthand for writing such strings. A hexadecimal number is written using the sixteen "hexadecimal digits" 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E,

and F. Each hexadecimal digit stands for four bits. So 0 represents 0000, 1 represents 0001, 2 represents 0010, ..., E represents 1110, and F represents 1111. We could also say that in the base ten, the hexadecimal digit A stands for the number 10 (ten), B stands for 11 (eleven), C stands for 12, D for 13, E for 14, and F for 15. A 32-bit binary number can be expressed as an 8-digit hexadecimal number.

Real Number

Real numbers are numbers that can contain decimal points, like 3.14159 or -234.5, or 12.0. They can also be written using "scientific notation." For example, 2.15e12 is a way of writing 2.15 times 10^{12}. The representation used in computers for real numbers is very complicated. And it allows some strange possibilities, such as INF and -INF, which stand for infinity and minus infinity. There are also NAN's. NAN stands for "not a number." NAN's are used to represent the results of illegal operations such as taking a square root of a negative number. Note that the **integer** 17 and the **real number** 17 have completely different representations in the computer, even though they are the same number mathematically. I will not describe the representation of real numbers in detail.

ASCII Text

Characters can be encoded using ASCII code. Each possible character is assigned a code that is one byte (that is, eight bits) long. With 32 bits, you can represents 4 characters in ASCII code. Not every possible byte represents an ordinary, printable character. The applet shows other bytes in the form <#n>, where n is the base-ten number corresponding to the byte. For example, the byte 00000111, which is equivalent to 7 in base ten, is shown as <#7>.

Pixels

At the center of the applet, you will see an 8-by-4 grid of little squares. Each of these thirty-two squares corresponds to one byte in the binary number. You should think of these squares as being very big pixels. Each pixel can be either black or white. One bit specifies the color of one pixel -- 0 for white or 1 for black. This is how two-color graphical images can be represented by binary numbers. Again, see Section 1.1 of the text. In the applet, you can change the color of a pixel by clicking on it.

David Eck (eck@hws.edu), June 1997

The xLogicCircuits Applet

The applet on this page lets you build simulated logic circuits from AND, OR, and NOT gates and see how they behave. This is not a serious circuit design tool. It's an educational tool for learning the basics about logic circuits. In particular, only the "logical" behavior is simulated, **not** the electrical components from which real gates are made. The xLogicCircuits applet is one of several applets written by David Eck for use with his introductory computer science textbook The Most Complex Machine. (They can also be used independently of the book.) For a list of other applets and for lab worksheets that use the applets, see the index page.

The applet is set to load some sample circuits. You should see an XOR circuit with two inputs on the left, some gates and wires that compute the exclusive or of the inputs, and an output on the right. (The definition of XOR is that the output is ON if one of the inputs is ON and the other is OFF.) To run a circuit, you first have to turn the power on, using the "Power" checkbox at the bottom of the applet. Then you can click on the circuit's inputs to turn them on and off. The signals from the inputs propagate along the wires through the circuits.

More instructions and more details are given on this page below the applet.

(Java not available.)

Building Circuits

You can build circuits by dragging components from the scrolling pallette on the left of the applet. This pallette always contains at least the six standard components: NOT gate, OR gate, AND gate, Input, Output, and "Tack." The gates are components that do computations. An AND gate, for example, has two inputs and one output. It turns its output on if **both** of its inputs are on. An OR gate also has two inputs. It turns its output on if either one of its inputs is on (or if both are on). The NOT gate has one input and one output. It turns its output on if and only if its input is off. Inputs and Outputs can be placed anywhere along the outer boundary of the circuit. An Input represents an input for the circuit as a whole. When the power is on, you can turn Inputs on and off manually by clicking on them. An Output represents a value computed by the circuit. The values of Outputs cannot be set manually. A "Tack" is simply an attachment point for wires. You might want to use Tacks to help make your circuits neater.

Once some components have been dragged onto the circuit board, you can draw wires to connect them. Every wire leads from a source to a destination. To draw a wire, you have to click the mouse **first** on the source. Then hold the mouse button down while you move the mouse to the destination. When you release the mouse button, the wire will be added to the circuit. If you release the mouse button when the mouse is **not** over a valid destination, no wire will be drawn. Circuit Inputs are valid sources for wires. So are Tacks. So are the **outputs** of gates. Valid destinations include circuit Outputs, inputs of gates, and Tacks. You can draw as many wires as you want **from** a source, but you can only draw **one** wire to a

destination. (This makes sense because when the circuit is running, a destination takes its value from the single wire that leads to it. On the other hand, the value of a source can be sent to any number of wires that lead from it.)

When you are drawing wires, you'll notice some helpful visual clues. Sources generally have a purple sort of color. Destinations are green. A Tack, which can be both a source and a destination is green until a line has been drawn to that Tack; then the Tack turns purple. When you are drawing a wire and you move the mouse over a legal destination, the component will be hilited. (A hilited gate is draw in blue, a hilited Output is enclosed in a blue rectangle, and a hilited Tack is enclosed in a blue circle.) Note that some components, such as gates, have multiple inputs. When you move the mouse over a component with several available inputs, the wire will jump to the nearest one. Make sure that you get the wire attached to the one you want.

The gates in the pallette can be rotated into four different orientations. Just click on the little red, curved arrow on the left, above the gate. Once a gate has been dragged onto the circuit board, it can't be rotated. However, it can be resized. When the gate is hilited, a box appears around it with small drag handles in each corner. Drag one of these handles to change the size of a gate. (To hilite a gate, or any component or wire, just click on it.

Any component on the circuit board can be moved around. Just drag it by holding down the **right** mouse button. Alternatively (if you have a one-button mouse, for example) you can hold down the control key to drag a component.

Whenever a wire or component is hilited, you can delete it by clicking on the "Delete" button. If you delete a component that has some wires attached, those wires will also be deleted. There is an "Undo" button that you can use to get back something that you delete by mistake. (Note that the "Undo" button can only undo **one** operation.)

You can edit a circuit even while the power is on.

Subcircuits

In addition to the six standard components, the scrolling component pallette can also contain circuits that have been built from these basic components and iconified using the applet's "Iconify" button. These iconified circuits can be used as components in building other, more complex circuits. You can drag them onto the circuit board, just like other components. The applet on this page should have loaded three such iconified circuits: "4 Bit Adder," "Clock," and "One Bit Mem."

To see inside an iconified subcircuit, hilite it and then click on the "Enlarge" button. The circuit will expand to fill the circuit board. (You can also just double click on the subcircuit. However, not all browsers respond to double clicks.) When you expand a circuit icon from the pallette, its icon is removed from the pallette. The circuit that was on the board previously is iconified and moved to the pallette, unless it is empty -- in which case, it is discarded. There is only one circuit on the board at any given time. You can edit the circuit while it is on the circuit board, and the changes you make will be permanent even after you

re-iconify the circuit.

If the circuit on the board containes a subcircuit, you can enlarge that circuit to see what is inside it. This does **not** remove the main circuit from the board -- it just lets you see an enlarged part of it. When you shrink the subcircuit back down to its original size, the main circuit is still there. For an example, Enlarge the "4 Bit Adder" sample circuit from the pallette. You'll see that it contains four copies of a subcircuit called "Adder." If you hilite one of the "Adder" circuits and Enlarge it, you can see what's inside it. You'll also see a big red message, "Enlarged from 4 Bit Adder," to remind you that what you are looking at is a part of a larger circuit.

It is important to understand that when you drag a subcircuit onto the circuit board, what you get is a **copy** of the circuit from the pallette. If you edit the original circuit on the pallette, it **doesn't** affect any copies that have been made. If you edit one of the copies, it doesn't affect the other copies. Note in particular that it's easy to get two circuits that have the same name but that are different internally. (My advice, of course, is don't do it!)

By the way, a circuit can have multiple outputs. When you draw a wire from a subcircuit to another component, make sure that the wire starts from the output that you want.

Other Features

The "Clear" button can be used to clear all the components from the circuit board. If several circuits are stacked up on the board, it only affects the one on top. The Clear button does **not** remove circuits from the pallette. (If the circuit you Clear is an enlarged subcircuit that is part of another circuit, the Clear button does not remove the circuit Inputs and Outputs. This is on the theory that they are probably connected to other components in the containing circuit, and you probably don't want to delete all the connecting wires. However, you can still delete the Inputs and Outputs individually, if you insist.) **Note:** You can use the "Undo" button to undo a Clear, provided that you do so before you do any other operation.

The "Save" and "Load" buttons are for working with files. (Hoewever, it is likely that your browser's security policy will prevent you from using them.) The Save command saves the entire state of the applet, including all the circuits in the pallette and on the circuit board. When you Load a file, everything in the applet is thrown away and replaced by the data in the file. **Note** that the Load command is undoable. Pressing the Undo button immediately after loading a file will restore the previous data.

Above the applet you'll also find a text-input box labeled "Title:". This is name of the circuit currently visible on the circuit board. You can change the name by editing the contents of this text-input box.

At the left of the strip of controls below the applet is a Speed pop-up menu. This menu affects the speed at which signals propagate through the circuit. You might want to use the slower speeds to watch what is happening in more detail. (At the Fast speed, it is possible in some browsers that the circuit will not be drawn correctly. For example, the "Clock" sample circuit should blink very quickly when the speed is set to Fast, but your browser might not

allow applets to draw to the screen quickly enough to show this.)

Another interface note: You should be able to insert a tack into the middle of an existing wire by double-clicking on the wire. If you double-click and hold the mouse down on the second click, you can drag the tack to a different position. (Again, I note that some browsers might not support double-clicks.)

And one more: It is not possible to draw a wire directly from an output of a gate or circuit to an input of the same gate or subcircuit. However, there is nothing to stop you from making loops out of several components, as is done in two of the sample circuits.

About the Sample Circuits

The sample circuits loaded by the applet on this page are all fairly standard examples of logic circuits. The XOR circuit, which is visible on the circuit board when the applet starts up, is discussed above. The "4 Bit Adder" is a circuit that can be used to add two 4-bit binary numbers. When the numbers are put on the eight input wires at the top of the circuit, the sum -- which can have five bits -- appears on the five output wires on the bottom and left of the circuit. (The output on the left is the "carry" bit.)

The "Clock" circuit has one input and one output. If you turn on the power, it will start turning its output wire on and off. If you turn on the input, however, the clock will stop "ticking." It is worthwhile to watch this example with the speed set to Moderate or Slow. (You can slow down the rate of ticking by inserting more Tacks into the loop. This is because of the way signals are propagated through circuits. The simulated circuits in this applet use "discrete time." At each discrete moment of time, each component in a circuit gets the values from its input wires, and computes values for each of its output wires. Real circuits don't work this way. Nevertheless, the logical behavior of most simulated circuits is -- with a few exceptions -- the same as the behavior of real circuits.)

The "One Bit Mem" circuit is a memory circuit that remembers the value of one bit. To store a bit (ON or OFF), put the value to be stored on the lower of the two Inputs on the left of the circuit. Turn the upper Input ON; wait a while, until the circuit settles down; and turn the upper Input OFF. The value that you've stored will appear on the output wire. This value will remain stored in the circuit as long as the upper Input is OFF. (If you don't wait long enough before turning the upper Input OFF, the value will not be properly stored. In fact, strange things can happen that are different from what would happen in a real, electrical circuit.)

David Eck (eck@hws.edu), August 1997

The xComputer Applet

The applet at the bottom of this page -- assuming that you have a Java-enabled browser -- simulates the CPU and memory of a model computer called "xComputer." The applet lets you write assembly language programs for this computer and see how the computer executes them. The applet was written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own.

The applet below is set up to load some sample programs. These programs contain some basic instructions for using the applet. Select a program from the pop-up menu at the top of the screen. Read the comments in the program. Then click on the "Translate" button to translate the program into machine language and load it into the computer's memory. You will then be able to run the program or to step through it one step at a time. The sample programs are also available as plain text files: [1], [2], [3], [4].

This is a rather technical applet, and you will probably need to read the documentation to understand it. Documentation is available on the xComputer Info page.

For a list of other applets and for lab worksheets that use the applets, see the index page.

(Java not available.)

David Eck (eck@hws.edu), June 1997

```
; Example 1:  Basics

; This file contains a fairly simple program written in the
; assembly language of xComputer.  It illustrates a few basic
; assembly language instructions and the format of assembly
; language programs.  (Note that a semicolon (;) and anything
; that follows it on a line is a comment.)

; The program computes the sum of a list of numbers.  The
; list can only contain non-zero numbers.  A zero marks
; the end of the list.  The list of numbers is at the end
; of this file.  They are actually treated by the computer
; as part of the program:  A number on a line by itself
; is simply stored in memory.  An instruction, on the
; other hand, is translated into a number that represents
; that instruction in machine language.  It is that
; number that is actually stored in memory.

; To run a program, click the "Translate" button that
; appears in the applet below the program.  The program will be
; translated into assembly language and the computer screen
; will reappear, with the program in its memory.  The
; program probably appears in the form of numbers, but you
; can change the way the contents of memory are displayed by
; selecting a view style from the pop-up menu that is
; above the scrolling memory display.

; Once the program is in memory, you can run it by clicking
; on the "Run" button.  There is a Speed pop-up menu for
; selecting the speed at which the program runs.  Alternatively,
; you can use the "Step" or "Cycle" button to move through
; the execution of the program manually.

; If you want to run the program a second time, you should
; first click on the "Set PC=0" button.  The PC tells the
; computer where to find its next instruction.  It has
; to be set to zero to point to the start of the program.

; Each of the following instructions has a comment that says
; what it does:

lod-c 30  ;  Load the constant 30 into the AC register.
sto 25    ;  Store the value (30) from AC into location 25.

lod-c 0   ; Load 0 into the AC.
sto 26    ; Store the 0 into location 26.

lod-i 25  ; The value in location 25 is an address of some
          ;    memory location.  Get the value from that
          ;    address and put it into the AC.  (The "i"
          ;    indicates what is called indirect addressing.)

jmz 12    ; If the value in the AC is zero (indicating
          ;    end of the list of numbers) then jump
          ;    to location 12, where the program will halt.

add 26    ; Add the number in location 26 (the sum of the
          ;    previous numbers) to the AC (which contains
          ;    the next number from the list).
sto 26    ; Put the number from the AC into location 26.
```

```
lod 25    ; Load the number from location 25 into the AC.
inc       ; Add one to the number in the AC.
sto 25    ; Put the number from AC into location 25, so
          ;   location 25 now contains the location of the
          ;   next number in the list.

jmp 4     ; Jump to location 4, to get the next number
          ;    from the list.  NOTE:  Locations are numbered
          ;    starting from zero.

hlt       ; This is a halt instruction, which tells the
          ;    computer to stop execution.  (It is in
          ;    location number 12.)

@30  ; This is a special command that says that the following
     ; items are to be stored in memory starting "AT" location
     ; number 30.

26   ; These numbers will be in memory starting at
17   ;    location 30.  The computer will add them
-34  ;    and leave the answer in location 26
15
12
-23
-7
19
87
11
-73
21
0
```

```
; Example 2: Graphics

; This is a very simple program that is meant to demonstrate a
; "Graphics" memory display.  If you select "Graphics" from
; the pop-up menu above the scrolling memory display area of
; the xComputer, the scrolling display will be replaced by
; a rectangle that shows the entire contents of memory at
; once.  Each pixel in the rectangle represents one bit
; in memory.  The pixel is white if that bit is zero and
; is black if that bit is one.  The rectangle is 64 pixels
; wide, so each row of dots represents four 16-bit memory
; locations.  As the computer executes a program, you can
; watch the dots change as memory is modified.  This can
; be particularly nice if you set the Speed pop-up menu
; to "Fastest Speed".

; The program simply stores the numbers 1, 2, 3, ... in
; consecutive memory locations, starting at location 20.
; To run the program, click the "Translate" button located
; below this program.  Once the computer reappears, set the
; memory display pop-up menu to "Graphics".  The program will
; appear as a few dots at the top of the memory rectangle.
; Set the run speed to "Fastest Speed" and then click the
; "Run" button.  You should see memory fill up with a
; rather attractive pattern.

; (Note: This is a self-modifying program, an old-fashioned
; but cute idea.  The commands in locations 2 and 4 are
; changed as the program runs so that they load and store
; into different locations each time they are executed.)

lod-c 1   ; Put the starting number in location 20
sto 20

lod 20    ; Add 1 to the number in location 20 and put the
inc       ;     result into location 21
sto 21

lod 2     ; Modify the instruction in location 2 so it
inc       ;   loads from the next location.
sto 2

lod 4     ; Modify the instruction in location 4 so it
inc       ;   stores into the next location.
sto 4

jmp 2     ; Go back to the "LOD" instruction in
          ;     location 2.
```

```
; Example 3:  Labels

; Many assembly language instructions refer to
; addresses of memory locations.  This could force
; you to count instructions in order to find the
; address number of the location you want to refer to.
; Fortunately, there is a way around this:  Use labels
; to refer to memory locations.

; A label is just a name for a memory location.
; You define a label by writing the name of the
; label, followed by a colon (:) followed by
; the contents of the memory location.  The value
; of the label is the address of that location.
; You can use the label anywhere in the program
; where you could use a number.  For example,
; the command "JMP start" will jump to the
; location with label "start".

; The program in this file uses several labels,
; such as "loop," "doAdd," "N1," and "ANS."  Some
; of these label refer to instructions, and
; some of them refer to data.  Both of these
; uses are very common.

; Note, by the way, that the way this program is
; formatted isn't important, as long as there is
; at most one instruction or data value per line.
; Also, you should know that the computer doesn't
; distinguish between upper and lower case letters
; in instructions or in label names.


; This program multiplies two numbers, stored in
; locations "N1" and "N2".  The result is left in
; location number "ANS".  (How this works isn't
; important, but essentially, it is a loop that
; looks at the bits in N1.  When a bit is found that
; is 1, N2 is added to ANS.  In any case, each time
; through the loop, N2 is multiplied by 2.)  For this
; to give the correct answer, the answer must be in
; the range of numbers that can be represented
; using 16 bits.

; The program:

       lod-c 0     ; Start by putting a zero into "ANS"
       sto ANS

loop:  lod N1      ; If N1 is zero, the process is complete.
       jmz done

       shr         ; Otherwise, shift N1 one bit right.
       sto N1
       jmf doAdd   ; If the bit shifted off the end of N1
                   ;   was a one, jump to doAdd to add N2
                   ;   to the answer.

shift: lod N2      ; Multiply N2 by 2 by shifting it left.
       shl
       jmz done    ; If N2 is zero, we are done.
```

```
       sto N2

       jmp loop    ; Proceed to the next iteration.

doAdd: lod N2      ; This section adds N2 to ANS before
       add ANS     ;   doing the preceding shift operation.
       sto ANS
       jmp shift

done:  hlt         ; Halts the program.


@20  ; This says that when the program is loaded, the
     ;   following item is to be at location 20.  Thus,
     ;   N1 will be 20, N2 will be 21, and ANS will be 22.

N1:   13    ; The number 13 is stored in a location named "N1".
N2:   56    ; 56 is in a location named "N2".  These are the
            ;    numbers that will be multiplied; change them
            ;    to any values you like.

ANS:  0     ; "ANS" is the name for a memory location that
            ;    will hold the product of N1 and N2 when
            ;    the program ends.
```

```
; Example 4: 3N+1 sequences

; This file doesn't illustrate anything in particular about
; the xComputer.  It's just that I really like the 3N+1
; problem.

; Starting from any positive integer N, the "3N+1 sequence"
; for N is computed as follows:  If N is 1, then stop; the
; sequence is complete.  Otherwise, if N is even then divide
; N by 2.  Otherwise (if N is odd), multiply N by three and
; add 1.  The question is whether this sequence terminates
; for ALL starting values N.  The answer is not known at
; this time.

; This program computes 3N+1 sequences for various values
; of N, starting from 1.  For each sequence, it counts
; the number of terms in the sequence.  The values are
; stored in memory in successive memory locations.

; Run this at "Fastest" speed, and watch it in graphics
; mode to see the random-looking series of sequence
; lengths that are generated.  Or watch locations
; 42, 43, and 44 (labeled by num, N and ct), which are
; where all the computational action takes place.


       lod-c 1    ; Let num = 1.  "Num" is the starting
       sto num    ;   value for the current sequence

       lod-c 100  ; 100 is location in memory where
       sto loc    ;   first answer is to be stored.

loop1: lod num    ; "Loop1" computes one sequence; begin by
       sto N      ;    initializing N to the starting value
                  ;    for the sequence.

       lod-c 1    ; "Ct" keeps track of the number of terms
       sto ct     ;     in the sequence; start counting at 1.

loop2: lod N      ; "Loop2" computes one term in the sequence.
       dec        ; Test if N=1 by subtracting 1 from it and
       jmz next   ;    testing if the answer is 0.  If so, this
                  ;    sequence is complete; jump to "next" to
                  ;    get ready for the next sequence.
       and-c 1    ; Compute bitwise logical AND of 1 with N-1.
       jmz odd    ; If the answer is 0, N is odd; jump to
                  ;    location "odd" to handle that case.
       lod N      ; Otherwise, N is even; divide N by 2 by
       shr        ;    shifting it right, and putting the
       sto N      ;    result back into N.  Then jump to
       jmp count  ;    "count" where this term in the sequence
                  ;    is counted.
odd:   lod N      ; If N is odd, multiply it by 3 by adding it
       add N      ;    it to itself twice.  Then add 1.
       jmf error  ;    If any of these additions produces a
       add N      ;    result greater than 65535, the FLAG
       jmf error  ;    register is set.  This indicates an
       add-c 1    ;    error: "Number too large for this
       jmf error  ;    computer".  Jump to "error" if the
       sto N      ;    FLAG is set.
```

```
count: lod ct     ; Count this term in the sequence by
       inc        ;    incrementing the value of ct.
       sto ct
       jmp loop2  ; Return to start of "loop2" to do the
                  ;    next term in the sequence.

next:  lod ct     ; The 3N+1 sequence for the current starting
       sto-i loc  ;    value, num, is complete.  Store the
       lod loc    ;    number of terms in the sequence in the
       inc        ;    location given by the value of loc,
       sto loc    ;    then add 1 to loc.
       lod num    ; Also add 1 to num to give the starting
       inc        ;    value for the next sequence.
       sto num    ;
       jmp loop1  ; Jump to "loop1" to do the next sequence.

error: lod-c 0    ; A term in the current sequence has
       dec        ; exceeded 65535.  Store -1 (computed
       sto ct     ; as zero minus one) in ct and jump to
       jmp next   ; "next" to get ready for the next sequence.

num:   0   ; starting value of sequence
ct:    0   ; number of terms in the sequence
N:     0   ; current value of N in sequence
loc:   0   ; address where answer is to be stored

5#    0    ; Put 5 extra zeros in memory, just to leave space
```

xComputer Info

A computer stores programs and data in its main memory (or "RAM"). It has a central processing unit (or "CPU") that fetches instructions from memory, one-by-one, and executes each instruction. This is called the fetch-and-execute cycle. A single fetch-and-execute cycle is itself made up of small steps. Each of these little steps performs a very simple operation, such as copying data from one location inside the CPU to another, adding two numbers, or moving information between main memory and the CPU. Each step is accomplished by turning control wires on and off. These control wires are attached to the main memory and to various components in the CPU. A control circuit turns the control wires on and off, based on just a few pieces of information, including the instruction that is being executed and the value of a counter that counts off the little steps of each fetch-and-execute cycle. All the values that CPU is currently working with are stored in registers, which are small memory units contained within the CPU. The whole process is driven by a clock. Each time the clock ticks, one step of a fetch-and-execute cycle is performed.

That's a very brief description of the fundamental operation of a computer. With some extensions, it is true of all existing computers. The xComputer applet simulates this fundamental operation for a computer with a 1024-location RAM and eight registers. Like any computer, the xComputer has a certain set of machine language instructions that it understands. Machine language instructions are actually binary numbers and are not really meant to be read or written by humans. Programs for the xComputer are therefore usually written in assembly language. An assembly language program must be translated into machine language before it can be executed by a computer. The assembly language for xComputer has 31 different instructions, which are listed below.

The xComputer and its assembly language are discussed in detail in Chapter 3 of The Most Complex Machine. This page does not cover all the details, but it should have enough information to enable you to use the applet and even to write some assembly language programs.

The xComputer Applet

The point of the xComputer Applet is to illustrate the step-by-step operation of a computer as it executes a program. When the applet starts up, it displays three sections: A "Control" area, a set of "Registers" and a "Memory." The Memory is a scrolling list occupying the right-hand third of the applet. It displays 1024 memory locations, numbered from 0 to 1023. Each location contains a 16-bit binary number that can be interpreted either as data or as a machine language instruction. The Registers section of the applet shows eight registers that are components in the xComputer's Central Processing Unit. The registers are explained below. Finally, the Controls are for interacting with and controlling the xComputer.

The applet has another mode in which it displays a text-input area where you can write and edit assembly language programs. (Information about the assembly language of xComputer is

given below.) To start a new assembly language program, just click on the "New Program" button in the Control area of the applet. Alternatively, choose "[New]" from the pop-up menu at the very top of the applet. There is also a "Load File" button that can be used to load a program that has been previously saved to a file; however, you can only do this if your browser permits applets to access files. Programs that have been created or loaded are listed in the pop-up menu at the top of the applet. You can view any program by selecting it from this menu. You can return to main computer screen by selecting "Computer" from this menu.

Suppose that you are looking at an assembly language program in programming mode. Before xComputer can do anything with that program, it must be translated into machine language and put into xComputer's main memory. When the applet is in programming mode, there is a "Translate" button at the bottom of the applet that will attempt to do this. If an error is found in the program, an error message will be displayed. (You can rid of this message by clicking on it, if you like.) If the program contains no errors, it will be put into xComputer's memory and the applet will switch to the main computer screen. You can then run the program or step through it, as described below.

Short programs can also be entered directly into memory from the Control area of the xComputer screen. Information that you want to put into memory -- instructions or data -- can be entered into the text-input box labeled "data:". When you press return or click the "Data to Memory" button, the data is stored in memory. The location in memory where it is stored is specified by a number in the "addr:" text-input box. You can only enter data for one memory location at a time. Each time you enter data, the number in the "addr" box is automatically incremented by one. This lets you store information into successive memory locations simply by typing values and pressing return after each one.

The question remains, exactly what sort of thing can you enter into main memory? The first thing you need to understand is that what is really in each memory location is a 16-bit binary number. Any other form of information must be represented in this form. The same binary number can represent different values, depending on how it is interpreted. You can enter information in the "data" box in any of the following ways:

- Integer -- any whole number in the range -32768 to 32767.
- Unsigned integer -- any whole number in the range from 0 to 65535. (Thus, you can actually use any number from -32768 to 65535. However, numbers from -32768 to -1 and from 32768 to 65535 are ambiguously represented by the same binary numbers.)
- Assembly language instruction -- any legal assembly language instruction for xComputer, such as "LOD-C 17". (See the list below.)
- ASCII characters -- a single quote, followed by any typeable character or a pair of characters (except for carriage return). The quote just marks this as ASCII data. For example: 'DE
- Hexadecimal number -- a dollar sign ($) followed by from one to four hexadecimal digits. For example: $A73D
- Binary number -- a B (upper or lower case) followed by from 1 to 16 zeros and ones. For example: B110010111010

When you type any of these things in the "data" box and press return, it is translated into a

binary number and put into memory at the specified address. (If there is some error in what you type, you'll get a small error message above the "addr" box.) So, a binary number in memory can represent several different things. The xComputer applet allows you to view the contents of memory in several different forms. Select the view that you want from the pop-up menu above the memory display. The "Instructions", "Integers", "Unsigned Ints", "Binary", and "ASCII" display styles correspond to some of the data types listed above. (In the ASCII display style, two characters are shown in each memory location. Non-printing characters are shown as ASCII code numbers enclosed between < and >. For example, <#17> represents the character with ASCII code number 17.) The "Graphics" display shows all of memory at once, as a rectangle full of black and white dots. Each dot represents one bit in memory. A black dot represents a one, and a white dot represents a zero. The rectangle is 64 bits wide, with each row of dots representing four memory locations of 16 bits each. The "Control Wires" display style actually doesn't have anything to do with memory at all. I'll discuss it below.

For example, here is a short assembly language program that adds the numbers 105 and 17, stores the answer into memory location 10, and then halts:

```
lod-c 105
add-c 17
sto 10
hlt
```

You could enter this program directly into memory by typing the lines, one at a time, into the "data" box. Note that what you **see** in memory depends on the setting of the memory display style. You might want to try entering this program, and use it as an example as you read the next section.

Running a Program

A program consists of a series of instructions stored in memory. The computer fetches instructions one-by-one and executes them. The program counter register (or "PC") tells the computer which address to go to in memory for the next instruction. When you want to run a program, you should always first check that the value in the PC register is the address of the location that contains the first instruction of the program. You can set the value in the PC to zero using the "Set PC=0" button. To set the PC to some other value, type the value into the "addr" box and then click on the "Addr To PC" button. **This is important.** A common, frustrating mistake when trying to run a program on xComputer is simply to forget to tell xComputer where in memory the program is located! (Note: If you use the "Translate" button in programming mode to put a program into memory, the PC will automatically be set to zero at the same time. However, after you run the program once, you have to reset the PC manually if you want to run it again.)

Once you have set the PC, there are three different ways to run the program:

- Click the "Step" button. This performs one of the several small steps that make up each fetch-and-execute cycle. You have to click on this between five and ten times, depending on the instruction, to execute each instruction.

- Click the "Cycle" button. This is meant to perform one complete fetch-and-execute cycle. More exactly, it performs "step" operations until the value in the COUNT register is two. At that point, a new instruction has just been loaded into the IR register. Clicking the "Cycle" button again will execute that instruction.
- Click the "Run" button. This makes the computer execute instructions continually, like a real computer. The "Run" button changes into a "Stop" button, which you can use to stop the computer. It will also stop if it executes a HLT instruction. The speed at which the computer runs is determined by a pop-up menu just below the run button. At the "Fastest Speed", the register display is turned off, so the computer can run as quickly as possible. This speed is especially useful with the "Graphics" memory display.

Registers and Control Wires

The xComputer has eight registers. A register is a memory unit that holds one binary number. Different registers holds different numbers of bits. Each of the registers has a role to play in fetching and executing instructions. Here is a short description of the purpose of each register:

- ADDR register: The address register is a 16-bit register that holds the address of a location in memory. Whenever data is read from or written to memory, this is the address that is used. (If you turn on the "Autoscroll" checkbox, below the scrolling memory display, then any time the value in ADDR changes, the memory will be scrolled to show that address at the bottom of the display.)
- PC register: The program counter is a 10-bit register that contains the address in memory of the next program instruction that is scheduled to be executed. The PC is ordinarily incremented by 1 during each fetch-and-execute cycle. Its value can be also be changed by the execution of a jump instruction, which tells the computer to jump to a different location in the program and continue execution from there.
- IR register: The instruction register is a 16-bit register that holds a program instruction while it is being executed. This is where an instruction is put when it is fetched from memory.
- COUNT register: This is a 4-bit register that counts off the steps in each fetch-and-execute cycle. At the beginning of each step, its value is incremented by one. The last step of the cycle resets this register to zero, so that the next cycle can begin.
- AC register: The accumulator is a 16-bit register that holds a number that is being used in the current calculation. When a number is loaded from memory, it is put in the AC. When a number is "added", it is added to the value currently in the AC, and the result is put back into the AC. Etc.
- FLAG register: This is a 1-bit register that can give extra information about a calculation. For example, when two 16-bit numbers are added, the final "carry" into the 17-th column is stored in the FLAG register. When a shift operation is performed on the AC, the extra bit that is shifted off the end is placed into the FLAG register.
- X and Y registers: These are 16-bit registers that hold numbers that are to be used in a calculation. For example, when two numbers are to be added, they are placed into X

and Y. (The Y register is also used as a temporary storage place in a few cases.)

The X and Y registers are connected to the inputs of an Arithmetic-Logic Unit, or "ALU", which does all the arithmetic and logical calculations in the computer. The outputs of the ALU are connected to the AC and to the FLAG register. (The ALU is not shown in the xComputer applet.)

The components of the computer -- including the main memory, the registers, the clock, and the ALU -- are controlled by turning wires on and off. These wires are connected to various components of the computer, and they control the operation of those components. It is these " control wires" that make the steps of the fetch-and-execute cycle happen. You can see a list of the control wires in xComputer by selecting the "Control Wire" option from the memory display pop-up menu. The wires that are currently turned on are shown in red. As a program is executed, the wires that are on change during each step of each fetch-and-execute cycle. You can watch how they change in order to learn how each step is accomplished.

For example, in step #1 of each fetch-and-execute cycle, the control wire named "Load-ADDR-from_PC" is turned on. This causes the number stored in the PC -- which is the location of the instruction that is to be fetched -- to be copied into the ADDR register -- where it sets up the main memory for reading from that location. The purposes of most of the wires are clear from their names. (The seven wires at the top of the list, starting with "Select-Add" are connected to the ALU. The ALU can perform several different calculations. The Select wires are used to tell it which calculation it should do.)

The Assembly Language of xComputer

An assembly language program is simply a way of specifying a sequence of 16-bit binary numbers to be stored in the computer's memory. As such, it can include any of the data items described above: Assembly language instructions, numbers in the range -32768 to 65536, hexadecimal numbers (up to four digits, preceded by $), binary numbers (up to 16 bits, preceded by B or b), and ASCII characters (one or two characters, preceded by a single left quote mark).

The legal instructions are listed below. An instruction consists of a two- or three-character code, such as LOD, OR, and HLT. Since upper and lower case letters are not distinguished, these could also be written as lod, or, and hlt. In some cases, this instruction code can be followed by an addressing mode, indicated by "-C" for "constant" addressing mode and by "-I" for "indirect" addressing mode. The addressing mode indicates how the data for the instruction is to be used. For example, ADD-C 17 indicates that the constant, 17, is to be added to the number in the accumulator, while ADD 17 indicates that a number is to be read from memory location 17 and that number is to be added to the number in the accumulator.

As you can see, the data for the instruction simply follows the instruction. It must be on the same line. You can't split instructions over two lines, and you can't have more than one instruction on a line. Not all instructions need data. If you provide data for an instruction that doesn't need it, it is legal, but the data will be ignored when the instruction is executed. The data for an instruction is a 10-bit binary number. It can be given in any of the following

forms:

- a number between 0 and 1023,
- a binary number between B0 and B1111111111,
- a hexadecimal number between $0 and $3FF,
- a single ASCII character, preceded by a left single quote mark,
- a label name.

The last possiblilty -- a label name -- brings us to a whole new aspect of assembly language. An assembly language program can contain more than just a sequence of items representing 16-bit numbers. It can contain other things to make programming easier by letting the computer do more of the work. A label is a name that stands for a number. A label represents a 10-bit binary value and can appear anywhere in the program where such a value could be used, that is, as the data part of an instruction or as a stand-alone item on a line by itself. When the program is translated, the label is replaced by the number it represents. A label is given a value by using it to label one of the 16-bit items that make up the program. The label name must be followed by a colon (:) and the item that it labels. The value of the label is the address of the location in memory that contains that item. For example, the following program adds up all the numbers from 1 to 50:

```
            lod-c 1      ; Initialize number to contain 1.
            sto number
            lod-c 0      ; Initialize sum to contain 0.
            sto sum
      next: lod sum      ; Add current value of number to sum.
            add number
            sto sum
            lod number   ; Add one to the value of number.
            inc
            sto number
            sub-c 51     ; Subtract 51 from the number,
                         ;                   which is still in AC.
            jmz done     ; If the answer is zero, jump to "done".
            jmp next     ; Otherwise, jump to "next",
                         ;            to continue the computation.
      done: hlt          ; Halt.
       sum: 0            ; (The zeros are place-holders to reserve
    number: 0            ;    memory locations for sum and number.)
```

In this program, next, done, sum, and number are labels. Next and sum refer to locations that hold instructions. Sum and number refer to locations that hold data for the program. The programmer can work with the instructions and data without having to work out the actual location numbers. (This program also illustrates comments. Anything on a line after a semicolon (;) is treated as a comment and is ignored by the computer.)

There are a few more things you can do in an assembly language program. A program item can be preceded by a number followed by a #. This is a repetition count and is the same as typing the item the specified number of times. For example, "25# 17" puts a 17 in each of the

next 25 memory locations. "4# SHL" is equivalent to four SHL instructions in a row.

Ordinarily, a program is loaded into consecutive memory locations starting at location 0. However, you can specify where loading is to take place by using the character @ followed by an address. For example, "@100" specifies that the next item is to go into memory location 100. (Items following that one will then go into location 101, 102, etc.) You might use this feature to put "subroutines" at specific points in memory.

Finally, you can store a string of ASCII characters into consecutive memory locations by using a string. A string is just a series of characters enclosed between double quotes, such as "Hello World!". When the computer encounters a string in a program, it stores the characters in consecutive memory locations, one per location.

List of Assembly Language Instructions

Here is a complete list of the assembly language instructions for xComputer. In this listing, "X" is data for the instruction; it must translate into a 10-bit binary number.

- ADD X -- Add the number in memory location X to the AC
- ADD-C X -- Add the number X to the AC
- ADD-I X -- Let Y be the contents of memory location X, and add the number in location Y to the AC
- SUB X -- Subtract the number in memory location X from the AC
- SUB-C X -- Subtract the number X from the AC
- SUB-I X --Let Y be the contents of memory location X, and subtract the number in location Y from the AC
- AND X -- Bitwise AND the number in memory location X with the AC
- AND-C X -- Bitwise AND the number X with the AC
- AND-I X -- Let Y be the contents of memory location X, and bitwise AND the number in location Y with the AC
- OR X -- Bitwise OR the number in memory location X with the AC
- OR-C X -- Bitwise OR the number X with the AC
- OR-I X Let Y be the contents of memory location X, and bitwise OR the number in location Y with the AC
- NOT -- Apply a bitwise NOT to the AC
- INC -- Add 1 to the AC
- DEC -- Subtract 1 from the AC
- SHL -- Shift the AC left one bit
- SHR -- Shift the AC right one bit
- LOD X -- Load the number in location X into the AC
- LOD-C X -- Load the number X into the AC
- LOD-I X -- Let Y be the contents of memory location X, and load the number from

location Y into the AC

- STO X -- Store the value in AC into memory location X
- STO-I X -- Let Y be the contents of memory location X, and store the value in AC into location Y
- JMP X -- Jump to location X (that is, store X into the PC, so that the next instruction will be loaded from X)
- JMP-I X -- Let Y be the contents of memory location X, and jump to location Y
- JMZ X -- If the value in the AC is zero, then jump to location X
- JMZ-I X -- If the value in the AC is zero, then let Y be the contents of memory location X, and jump to location Y
- JMN X -- If the value in the AC is negative, then jump to location X
- JMN-I X --If the value in the AC is negative, then let Y be the contents of memory location X, and jump to location Y
- JMF X -- If the value in the FLAG register is one, then jump to location X
- JMF-I X -- If the value in the FLAG register is one, then let Y be the contents of memory location X, and jump to location Y
- HLT -- Halt. That is, stop the xComputer by turning on the Stop-Clock control wire

---

David Eck (eck@hws.edu),
June 1997; minor editing May 1998

The xTuringMachine Applet

TURING MACHINES are very simple computational devices. A Turing machine has an infinitely long tape, divided into cells. Each cell can be blank or can contain a symbol chosen from some fixed finite list. The Turing machine moves along the tape reading and writing symbols. It has an internal state, which can be either the halt state or an integer between zero and some specified maximum value. When a Turing machine enters the halt state, it stops computing. Although Turing machines are very simple, any computation that can be done by any computer can also be done by some Turing machine.

The action that a Turing machine takes depends only on its state and on the symbol displayed in the cell where the machine is currently located. Given this information, the Turing machine takes three actions: It writes a symbol to the cell (possibly the same one that is already there); it moves one cell to the left or one cell to the right; and it sets its internal state (possibly to the same state that it is currently in). The Turing machine has a table of rules that tells it what to do for various combinations of its current state and the symbol it reads from the current cell.

The xTuringMachine applet is designed to show Turing machines in action. The Turing machines that it works with have a maximum of 25 states, and they can only use the symbols 0, 1, x, y, z, and $. Nevertheless, they can do some non-trivial computations. The applet is set up to load several sample machines. More information about the applet can be found below.

This applet was originally written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own.

For a list of other applets and for lab worksheets that use the applets, see the index page.

Sorry! Your browser doesn't do Java!

About the Applet

A pop-up menu at the very top of the applet can be used to select machines that were loaded by the applet at start-up or that have been created by the user. Selecting "[New]", the first item in the menu, will create a new, empty machine.

Just below the applet is the machine itself, sitting on an infinite tape. The tape is divided into cells, and each cell either contains a symbol or is blank. The machine itself sits over one of the cells and displays its current state. By convention, the machine starts out in state zero. When it is in the halt state, it displays an "h". (This area of the screen is also used to display messages; the machine will be there when you dismiss the message.)

On the left side of the applet below the machine is a set of controls. The first control is a pop-up menu that controls the speed of the applet when it computes. The "Run" button makes the machine start computing; when you click it, it changes into a "Stop" button. The "Step" button makes the machine perform one step in its computation. The "Clear Tape" button does just that. The "Delete Rule" button can be used to delete one rule from the rule table. (It is only active when a rule has been selected. The selected rule is shown in red; you can select a rule by clicking on it.) The "Load File" and "Save" buttons are used to work with files. Your browser might not permit you to use them.

The table of rules is in the lower right section of the applet. Each rule tells the machine what to do if it is in a certain state and if it reads a certain symbol. The "Move" column tells the machine which direction to move: "L" for left and "R" for right. You can edit the "Write," "Move," and "New State" columns.

The columns labeled "Reading" and "Write" can contain the symbols $, 0, 1, x, y, and z. They can also contain the character "#", which is used to represent a blank cell. The "Reading" column might also contain "other", which represents a default value that means "any other symbol for which no explicit rule is given." If the "Reading" column contains "other", then the "Write" column can contain "same", which tells the machine to write the same character that it read.

Just above the rule table is a "rule maker" with a "Make Rule" button that can be used to add new rules to the table. The blue rectangle between the machine and the rule maker is a "palette" that is used in making and editing rules, changing the contents of the tape, and changing the current state of the machine. This is explained in the next two sections.

When the machine is running at "Moderate" speed or slower, after each step the applet displays the next applicable rule in the rule maker box -- so you can see why the machine takes the action it does. If there is **no** applicable rule in a give situation, the machine will stop and will display the message "No Rule Defined!" You could then use the rule maker to make the missing rule.

If the machine moves outside the applet as it is computing, the machine along with its tape will jerk back about 1/4 of the width of the applet. (By the way, if you somehow lose the machine off the edge of the applet, clicking the "Run" or "Step" button will make it reappear.)

Using the Mouse

It is possible to work with the xTuringMachine applet using only the mouse (and completely avoiding the keyboard). Here's how.

Before you can edit any item, it must be "hilited." The currently hilited item, if any, is surrounded by a bright blue-green outline. You can always hilite an editable item by clicking on it. Whenever an item is hilted, the palette will display a list of legal values for that item. (The palette is the blue rectangle just below the machine.) You can choose one of these values by clicking on it. You can also type the value you want.

As long as the Turing machine is not running, you can edit the current state of the machine and the contents of the tape. Click on the machine or on one of the tape's cells, then select a value from the palette. When you enter a symbol for the tape, the hilite moves one cell to the right.

You can edit the "Write", "Move", and "New State" columns of the rule table at any time, even while the machine is running. Click on the item you want to change, then select a value from the palette.

The rule maker is somewhat more complicated. The rule maker lets you set up one rule by editing any of the five values for that rule. To set up the rule, click on any of the values in the rule maker and edit it by selecting one of the values in the palette. When the rule is complete, click on the "Make Rule" button to add it to the rules table. (However, if the "In State" and "Reading" items in the rule maker are the same as those for an existing rule, the "Make Rule" button becomes a "Replace" button. When you click it, the rule in the rule maker will replace the rule in the table.) Everytime you make or replace a rule, the rule maker is automatically updated to show the next consecutive rule, since it is at least fairly likely that that is the rule you want to work on next.

You can use the mouse to drag the Turing machine into a new position on its tape or to drag the tape to a new position under the machine. If you want to drag them both together, use the right mouse button or hold down the Control key as you click.

Using the Keyboard

You can use the keyboard to perform any editing task in the xTuringMachine applet. Here's how.

Pressing the tab key will move the hilite among the three major areas: the machine, the rule maker, and the rule table. (If there is no hilited item, pressing the tab key **might** create a hilite in the rule maker; if not, you have to use the mouse.) Within one these three major areas, the up, down, left, and right arrow keys can be used to move the hilite from one item to another. When the hilite is in the tape or in the rule table, the "home" and "end" keys can also be used. Play around to see how they work.

You can always type one of the values displayed in the palette, instead of clicking on it. Note that a blank can be entered either by pressing the space bar or by typing a #. The default symbol ("other" in the "Reading" column or "same" in the "Write" column) is entered by typing a *.

When working in the rule maker, hitting the return key is the same as clicking on the "Make Rule" (or "Replace") button.

The Sample Machines

The applet on this page is set up to load four sample Turing machines. Here are brief descriptions:

- **CopyXYZ.txt**: This machine expects to be started on the left end of a sequence of symbols containing only x's, y's, and z's. It will make a copy of its input, and will halt on the left end of the copy.
- **GatherDollars.txt**: This machine should be started on the left end of a string of symbols. Any of the symbols $, 0, 1, x, y, and z are OK. The machine will move all the $'s to the left end of the string of symbols, leaving all the other symbols in the original order. It halts on the left end of the string.
- **CountInBinary.txt**: Expects to be started on the right end of a sequence of zeros and ones, which is interpreted as a binary number. The machine adds one to its input, and repeats this process forever. If started on a blank tape, it will start counting from one. Run it at "Fastest" speed.
- **BinaryAddition.txt**: The input for this machine should be two binary numbers, separated by a blank. The machine should be started on the right end of the second number. It computes the sum of the two binary numbers. The second number is erased in the process. The first number is replaced by the sum. The machine halts on the right end of the sum.

David Eck (eck@hws.edu), August 1997

The xTurtle Programming Applet

The applet at the bottom of this page -- assuming that you have a Java-enabled browser -- lets you write and run programs written in the "xTurtle" programming language. This applet was written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own. The xTurtle language is designed to be simple enough to learn easily, but complex enough to teach some important programming concepts.

The applet below is set up to load some sample programs. Use the pop-up menu in the upper left corner of the applet, and click on the **Run Program** button to see what it does. Full information is available about the applet and the xTurtle language on xTurtle Info page. A set of tutorial examples is also available.

For a list of other applets and for lab worksheets that use the applets, see the index page.

(Java not available.)

David Eck (eck@hws.edu), June 1997

xTurtle Tutorial Examples

The xTurtle Applet let's you write and execute programs written in a simple programming language, also called xTurtle. The applet at the bottom of this page will try to load eight tutorial examples. To read an example, select it from the pop-up menu at the upper left corner of the applet. Then to run it, click on the "Run Program" button. For full information on the applet and the language, see the xTurtle Info page. The eight tutorial files are also available as text files using the following links: [1], [2], [3], [4], [5], [6], [7], [8].

Warning: This applet seems to stretch or exceed the limits of some browsers.

(Java not available.)

David Eck (eck@hws.edu), June 1997

```
{ xTurtle Tutorial Example #1: Basics.

  This file demos some of the built-in
  commands of the xTurtle language.

  First lesson:  This is a comment,
  since it is enclosed between { and }.
  Comments are ignored by the compute.

  Note:  If the scroll bars on this text
  area are not active, it's a bug in
  your browser.  Try resizing the
  window.
}

forward(5)  { Move turtle forward 5 units,
              drawing a line as it goes. }

turn(90)  { Rotate turtle 90 degrees,
            counterclockwise. }

green  { Change drawing color to green.
         Color names include red, green,
         blue, cyan, magenta, yellow,
         black, gray. }

forward(2)  { Move forward 2. }
back(4)     { Move backwards 4. }

{ The net result of the preceding commands
  is to draw a red and green T-shape.
  Red is the default drawing color.
}

PenUp  { When pen is up, turtle doesn't
         draw anything as it moves. }

MoveTo(-5,3)  { Move directly to the point
                with coordinates (-5,3) }

PenDown { Start drawing again. }

rgb(1,0.5,0.5) { Changes drawing color to
                 the color with red, green,
                 blue components given by
                 1, 0.5, 0.5.  This will
                 be sort-of-pink. }

face(0)  { Set turtle's heading to 0 degrees,
           meaning face to the right. }

circle(3)   { Draw a circle of radius 3.
              The circle is drawn to the
              left of the turtle's current
              position. }

PenUp
MoveTo(-3,-7)  { Move again. }
PenDown

Magenta { Change drawing color to magenta. }
```

```
Arc(2,90)  { Draw a 90-degree arc of a
             radius-2 circle. }
forward(3)
Arc(2,90)
forward(3)
Arc(2,90)
forward(3)
Arc(2,90)
forward(3) { A box with rounded corners has
             been drawn. }

PenUp
MoveTo(5,-5)  { Move again. }
PenDown

black  { Draw in black. }
DrawText("Hello")  { Write the message Hello
                     at current cursor position. }

DrawText("World!")  { This lines up under
                      the Hello. }
```

```
{ xTurtle Tutorial Example #2: Variables.

  A variable is a name that can be used
  to hold a value.  Variables must be
  declared before they are used.  A
  variable can be assigned a value with
  an assignment statement.  Variables
  and functions can be used to compute
  values from complex expressions
  such as 3 * sin(x+3).
}

DECLARE x, y  { Allows you to use the
                variables named x and y. }

x := 2  { Assignment statement,
          puts the value 2 into
          into the variable x. }

y := 2*sqrt(x) { Computes sqrt(2), multiplies it
                 by 2, and assigns the resulting
                 value to y.  Sqrt computes
                 the square root.  There are
                 other predefined functions. }

forward(x)  { Variables can be used in commands. }
turn(-135)  { Negative turn rotates the turtle
              in a clockwise direction. }
forward(y)
turn(135)
forward(x)  { These commands have
              drawn a Z-shape. }


DECLARE Rate { Declares another variable.
               They don't have to be declared
               at the start of the program.
               Names can be any number of
               characters. }

Rate := 0.07  { Decimal numbers are OK. }

DECLARE money, interest  { More variables. }
money := 1000
interest := Rate * money

PenUp
MoveTo(-8,8)
PenDown

DrawText("Interest on $#money is #interest.")
  { The value of a variable can be included
    in a string.  Just include the
    character #, followed by the name of
    the variable.  When the string is
    actually printed, the value of the
    variable is shown. }

x := random { Assign a random value in the
              range 0.0 to 1.0 to x.  The
              value will be different every
```

```
              time the program is run.
              Note that x was already declared,
              so I don't have to declare it
              again.  In fact, it would be
              an error to do so. }

y := randomInt(100)  { y is assigned a
                       random integer in
                       the range 1 to 100. }

DrawText("Here's a random number: #x")
DrawText("Here's a random integer: #y")
```

```
{ xTurtle Tutorial Example #3: I/O

  I/O, or Input/Output, refers to
  the exchange of information
  between a program and the person
  using the program.  All the
  turtle graphics commands,
  including DrawText, are examples
  of output (from the computer
  to the user).  This file
  gives examples of several other
  I/O commands in xTurtle.
}

TellUser("Hello World!")
   { This command pops up a box
     displaying the specified
     string to the user.  The
     user must click on a
     displayed OK button before
     the program can continue. }

DECLARE amount

AskUser("What is the amount?", amount)
   { This also pops up a box
     displaying the string.
     There is also an input box
     where the user can type in
     a number.  That number
     is stored as the value of
     the specified variable.
     In this example, the number
     typed by the user is stored
     in the variable amount. }

amount := amount * 1.07
   { Uses the value typed in
     by the user in a computation.
     The computed value is
     then stored as the new
     value of amount. }

TellUser("The amount is now #amount.")
   { Just as in DrawText, strings
     can include variables.  This
     command displays a string
     containing the new value of
     amount. }

DECLARE resp

YesOrNo("Are you happy?", resp)
   { Displays the string, and
     gets a response from the user.
     With this command, the user
     can only answer yes or no.
     If the user says yes, the
     value of the variable is
     set to 1; if the user says
     no, the value is set to 0. }
```

```
TellUser("The recorded answer is #resp")
```

```
{ xTurtle Tutorial Example #4:  Loops

  A loop is used to repeat a sequence
  of statements over and over.  Some
  method must be provided to exit from
  the loop.

  In xTurtle, the beginning of a loop
  is marked with LOOP, and the end
  is marked with END LOOP.
}

LOOP  { start of a loop }
   forward(4)  { draw a line }
   back(4)     { return to center }
   turn(5)     { rotate 5 degrees }
   EXIT IF heading = 0  { maybe exit }
END LOOP { marks end of loop }

  { In the preceding loop, the
     repeated statements draw a
     line radiating out from a
     center point, and then rotate
     the turtle.  Each time
     through the loop, the
     computer asks itself, "Is
     heading = 0".  If the answer
     is yes, then the loop ends
     and the computer goes on to
     the next statement following
     the loop.  If the answer is
     no, the computer continues
     to execute the loop.   (The
     heading is the direction
     that the turtle is facing.) }

PenUp
MoveTo(-2.5,-6)
PenDown

DECLARE count
count := 0
gray

LOOP
   forward(5)
   turn(45)
   count := count + 1
   EXIT IF count = 8
END LOOP
   { In this loop, the computer
     adds 1 to count each time
     it goes through the loop.
     After it does this 8 times,
     the value of count will be 8.
     At that time, the loop will
     end.  The net result is that
     an 8-sided polygon has
     been drawn.  This is an
     example of a counting loop. }
```

```
{ As a final example, the
  following example computes
  the average of 100 randomly
  chosen numbers.  You should
  expect the average to be close
  to 0.5. }

DECLARE total, average
total := 0
count := 0  { This was already declared }

LOOP
   total := total + random
   count := count + 1
   EXIT IF count = 100
END LOOP

average := total / 100

PenUp
MoveTo(-8,8)
PenDown
blue
DrawText("The average was #average.")
```

```
{ xTurtle Tutorial Example #5:  IF

  An IF statement is used to choose
  among several possible courses of
  action.  The IF statement bases
  its decision on the value of one
  or more logical expressions.  A
  logical expression is something
  that can be either true or false,
  such as "x > 0".
}

{ A simple IF..THEN..ELSE makes a
  choice between two alternatives,
  based on whether a condition is
  true or false.  Here is an example: }

DECLARE N, x
N := randomInt(3)  { N is 1, 2, or 3 }
AskUser("Guess a number.", x)

IF x = N THEN
   TellUser("That's right!")
ELSE
   TellUser("Sorry, the number was #N")
END IF
 { The first TellUser statement is
   executed if "x = N" is true.
   The second is executed if
   "x = N" is false.  You can have
   any number of statements between
   THEN and ELSE or between ELSE
   and END IF.  The "END IF" at
   the end is required to mark
   the end of the IF statement.
   The ELSE part of the IF statement
   is optional. }


{ There is another version of the
  IF statement that chooses among
  more than two alternatives.  It
  uses "OR IF" to make additional
  tests.  The conditions in the
  IF and OR IF parts are tested
  in order.  If one is found that
  is true, than the corresponding
  statements are executed.  The
  IF statement then ends, without
  testing the other conditions.
  If an ELSE part is present, it
  is executed in the case where
  all the conditions are false.
  Here is an example that does
  a "random walk." }


LOOP  { begin a loop; statements can
        be nested, so I can put an
        IF statement inside this loop. }
```

```
   N := randomInt(4)  { N is 1, 2, 3, or 4 }

   IF N = 1 THEN
      face(0)   { This is done in case N is 1. }
   OR IF N = 2 THEN
      face(90)  { This is done in case N is 2. }
   OR IF N = 3 THEN
      face(180) { This is done in case N is 3. }
   ELSE
      face(270) { This is done in any other case,
                  which in this example can only
                  happen if N is 4. }
   END IF  { Marks the end of the IF statement }

   forward(0.5) { Moves forward a bit in the
                  direction that has just been
                  chosen at random. }

   EXIT IF (xcoord < -9) OR
              (xcoord > 9) OR
              (ycoord < -9) OR
              (ycoord > 9)
       { The loop ends when the turtle moves
          outside the x and y coordinates in
          the range from -9 to 9 }

END LOOP  { marks the end of the loop }
```

```
{ xTurtle Tutorial Example #6:  Subroutines

  It is possible to define subroutines
  and functions that can be used just
  like built-in subroutines, such as
  forward(x), and built-in functions,
  such as sqrt(x).  A subroutine is
  just a list of statements to be
  executed.  A function is similar,
  except it returns a value to be
  used in further computation. }

SUB Triangle   { Begin definition of
                 a subroutine named
                 "Triangle". }
  forward(3)
  turn(120)
  forward(3)  { Statements to define }
  turn(120)   {    what the subroutine }
  forward(3)  {    does. }
  turn(120)

END SUB  { Marks the end of the subroutine. }

{ Defining this subroutine has not
  actually drawn anything.  When the
  subroutine is called by using its
  name as a statement, the commands
  inside the subroutine are exeuted. }

Triangle  { Call the subroutine to
            draw a triangle. }

SUB Square(length)  { Define a subroutine
                      that has a parameter called
                      "length".  A value for the
                      parameter will be supplied
                      when the subroutine is called. }
  forward(length)
  turn(90)
  forward(length)  { Statements to define }
  turn(90)         {    what the subroutine }
  forward(length)  {    does. }
  turn(90)
  forward(length)
  turn(90)

END SUB  { Marks the end of the subroutine. }

PenUp
MoveTo(-6,-6)
PenDown
blue

Square(5)  { Draw a 5-by-5 square }
PenUp MoveTo(-5,-5) PenDown
Square(3)  { Draw a 3-by-3 square }
PenUp MoveTo(-4,-4) PenDown
Square(1)  { Draw a 1-by-1 square }
```

```
FUNCTION Area(length,width)
    { Begin definition of a function
      named "Area", with two parameters
      named "length" and "width" }

   DECLARE val  { This is a local variable,
                  for use inside this
                  function only. }

   val := length * width  { Compute a value. }

   RETURN val  { Specify the value to be
                 returned by the function. }

END FUNCTION

DECLARE answer
answer := Area(5,7)  { Call the function Area
                       with parameter values
                       5 and 7. }

PenUp MoveTo(-7,7) Pendown
green

DrawText("A 5-by-7 rectangle has area #answer.")
```

```
{ xTurtle Tutorial Example #7:  Recursion

  A recutsive subroutine or function is
  one that calls itself, or that calls
  another routine that calls it back, and
  so on.  You can do recursion in xTurtle.
  This file demos two standard examples:
  the recursive factorial function and
  a recursive tree-drawing subroutine.
}


  { For a positive integer, N,
    factorial N is defined to be
    N * factorial(N-1), as long
    as N > 1.  If N <= 1, then
    the answer is given directly
    as 1.  This definition can
    be expressed easily in a
    function that calls itself
    recursively. }

FUNCTION factorial(N)
   IF N > 1 THEN
      return N * factorial(N-1)
   ELSE
      return 1
   END IF
END FUNCTION

DECLARE N, F
N := 1
PenUp MoveTo(-10,9) PenDown
black
LOOP
   EXIT IF N > 9
   F := factorial(N)
   DrawText("factorial(#N) = #F")
   N := N + 1
END LOOP


  { A "binary tree" might be defined
    as a trunk with two smaller trees
    attached to it.  This is OK, as
    long as we say that the smaller
    trees are simpler than the main
    tree.  In the following subroutine,
    the "level" tells how simple the
    tree is.  When the level gets
    down to zero, only a trunk is
    drawn, with no attached trees. }

SUB Tree(size,level)
   IF level < 1 THEN
      forward(size)
      back(size)
   ELSE
      forward(size/2)
      turn(45)
      Tree(size/2,level-1)
```

```
      turn(-90)
      Tree(size/2,level-1)
      turn(45)
      back(size/2)
   END IF
END SUB

PenUp MoveTo(5,-8) PenDown
green
face(90)

Tree(15,6)  { Draw a "level-6" tree }
```

```
{ xTurtle Tutorial Example #8: multitasking

  Turtles in xTurtle have the cute ability
  to split themselves into a specified
  number of turtles.  Each turtle
  then goes on to execute the rest of
  the program.  (If the split occurs
  inside a subroutine, all the extra
  turtles are gone before the subroutine
  returns.) Turtles are split up in this
  way with the Fork command}


SUB starburst(lineCount)

   { Creates a bunch of lines
     radiating out from a center
     point.  The number of lines
     is given by the parameter.
     The lines have random lengths
     and point in random directions.
     Because of a limitation on the
     Fork command, lineCount can't
     be more than 100. }

   fork(lineCount)  { There are now
                      lineCount turtles }

   face(random*360) { Each turtle faces in
                      a random direction }

   forward(random*5)  { Each turtle goes
                        forward by a random
                        amount. }

   { All the turtles die before the
     subroutine returns. }

END SUB


PenUp MoveTo(5,5) PenDown
Starburst(50)

PenUp MoveTo(-5,-5) PenDown
blue
StarBurst(50)


{ There is a special variable that you
  can use to tell the turtles created in
  a fork statement apart.  The name of
  the variable is ForkNumber.  Each
  of the turtles created by a Fork(N)
  command has a different ForkNumber.
  The values are 1, 2, ..., N.  }

Fork(2)
PenUp
IF ForkNumber = 1 THEN
   MoveTo(-5,5)  { One turtle does this. }
```

```
   green
ELSE
   MoveTo(5,-5)  { The other turtle does this. }
   yellow
END IF
PenDown

Fork(60)  { Both turtles split; there are now
            120 turtles. }

Face(6 * Forknumber)  { Because they have
                        different values for
                        ForkNumber, the turtles
                        face in different directions. }

Forward(4) { Every turtle does this. }
```

The xSortLab Applet

SORTING a list of items -- that is, arranging the items into increasing or decreasing order -- is a common operation. It is also the most common example in the "analysis of algorithms," which is the study of computational procedures and of the amount of time and memory that they require. The xSortLab applet knows five different sorting methods. It has a visual sorting mode, where you can watch as sixteen bars are sorted into increasing order. And it has a timed mode, where you can measure the time it takes to sort a large number of items. The applet is easy to use (but you probably won't quite get the point of it unless you already know something about sorting). More information about the applet can be found below.

This applet was originally written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own.

For a list of other applets and for lab worksheets that use the applets, see the index page.

Sorry! Your browser doesn't do Java!

Visual Sort Mode

The xSortLab applet can display three different panels: a panel for "Visual Sort," a panel for "Timed Sort," and a "Log" panel. There is a pop-up menu at the top of the applet that can be used to switch among the three panels. (**Note**: Changing panels while a sort is in progress will abort the sort.)

The applet starts in "Visual Sort" mode, in which 16 bars are sorted step-by-step using one of the sorting algorithms Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, or QuickSort. Below the area where the bars are displayed are two message areas, which display a running commentary when a sort is in progress. The lower message area displays a detailed comment on each step in the sort. The upper area displays more general messages about major phases in the sort. (The lower message area is not used when the sort is being run at "Fast" speed.)

To the right of the bars is a column of controls. The first of these is a pop-up menu, that can be used to select the sorting method. (Again, doing this in the middle of a sort will abort the current sort.) Next comes a checkbox that can be used to determine whether or not the sort is done at "Fast" speed. When this box is **not** checked, you get to see a cute animation of moving bars; also, a longer delay is inserted between steps when you run the sort with the "Go" button. The "Go" and "Step" buttons are used for executing a sort. The "Start Again" button gives you a new, randomized list of 16 bars.

Two basic operations are used in sorting: comparing two items to see which is bigger and copying an item from one place to another. The number of comparisons and the number of copies used in the current sort are displayed below the controls.

Timed Sort Mode

If you switch to the "Timed Sort" panel, you'll see a large message area, with some instructions. This panel is used to obtain statistics about the running times of various running algorithms. The interesting question is how the running time depends on the number of items being sorted. There is a text-input box at the top of the panel where you can specify the size of the array that is to be sorted. You can also specify the number of arrays to be sorted. The point is that a small array takes so little time to sort that the time cannot be measured accurately. So, you should sort a number of arrays, all of the same size. You can measure the total time it takes to sort them all. The time required to sort one array can be obtained by dividing the total time by the number of arrays. You probably want to adjust the number of arrays so that the total time is at least a couple of seconds.

Note: Your computer must have enough memory to store all the numbers you want to sort. (If you are running the applet at all, you probably have enough memory to work with at least one million items.) If you ask for more numbers than you have room for, you **should** just a message telling you that you don't have enough memory. However, most systems that I have tried this on have crashed. This is a bug. You are warned. Stick to a reasonable number of items.

At the bottom of the panel are a pop-up menu that you can use to choose the sort method and a "Go" button that you can click to start the sort. (This changes to "Abort" while a sort is in progress.)

When you begin a sort, the first thing the computer does is to fill up the arrays with random numbers. If there are a lot of numbers, this will take a noticeable amount of time. Then, the computer begins to sort. As the sorting operation proceeds, statistics are displayed about twice per second. The statistics include the number of comparison and copy operations that have been performed, the number of arrays that have been sorted so far, the elapsed time since the computer began sorting, and the approximate compute time that the computer has devoted to sorting. The compute time is not the same as the elapsed time, since the applet is doing other things besides sorting (such as redrawing the screen). The applet tries to use 80% of the time for sorting, and to leave 20% for other tasks. The applet can measure the 20% of its time that it gives away voluntarily, but if other things are going on in your computer, it might lose some other time that it can't measure. This is why the measured compute time is approximate. So, you should **not** try to run a timed sort in the background! Just sit and watch -- or go get a coffee.

The Log

Every time a sort completes successfully, statistics about that sort are written to a log. For a visual sort, the number of copies and the number of comparisons are recorded. For a timed sort, all the statistics that are displayed in the "Timed Sort" panel are written to the log. You can view this log by selecting the "Log" option from the pop-up menu at the top of the applet.

The Log panel has buttons for clearing the log and for saving it to a file. However, it is likely that your browser will not permit you to save the log. Unfortunately, the applet has no provision for printing the log at this time. You might try using copy-and-paste to copy the data from the log to another program from which you can copy or print it.

David Eck (eck@hws.edu), August 1997

The xModels Graphics Applet

The applet at the bottom of this page -- assuming that you have a Java-enabled browser -- let's you write scene descriptions in a simple language. The applet can display still images and animations described in this language. Images are displayed as wireframe models. The point is to learn something about geometric modeling and geometric transformations, which are important in computer graphics. The applet was written by David Eck for use with his introductory computer science textbook The Most Complex Machine. However, it can also be used on its own.

The applet below is set up to load some sample programs. Click on the "RENDER" button to see the image or animation described by a program. Use the pop-up menu at the top of the applet to select among the sample programs. Full information is available about the applet and its scene description language on the xModels Info page. A number of tutorial examples are also available.

(Java not available.)

David Eck (eck@hws.edu), June 1997

xModels Info

Computer-generated graphics images are usually constructed in two stages: modeling followed by rendering. In the modeling stage, a geometric representation of the objects in the scene is constructed. The rendering stage produces the actual images, based on information in the model.

The xModels Applet lets you describe scenes in a simple scene description language. Scenes are rendered as wireframe models, a very minimal kind of rendering which shows just the edges of all the objects in the scene -- even edges that are behind other objects. The main point of the applet is not to produce fancy images; the point is to learn some of the basic ideas of geometric modeling.

This file contains fairly complete information about the xModels Applet and about the scene description language that it uses. Also available on a separate page are some tutorial examples.

I invented xModels to use as an example in The Most Complex Machine, a book that surveys the field of computer science. The xModels applet is based on two similar programs that I wrote for Macintosh computers (one for two-dimensional and one for three-dimensional graphics). These programs are among several that I wrote for use with The Most Complex Machine. All the Macintosh programs are available for downloading. I am in the process of porting all the Macintosh programs to Java.

The xModels Applet

The xModels applet is designed to be easy to use, so the major thing you need to learn about is the scene description language, which is discussed below.

The applet has two modes. In its program mode, it displays a text area where you can type and edit scene descriptions. There is a row of control buttons along the bottom that can be used to render the scene, to load a program from a file, to save the current program in a file, and to clear out all the text from the text area. The applet is in this mode when it first starts up. You can only switch to graphics mode by rendering a legal program.

If you click the "Render" button, the computer will examine the program contained in the text area to see whether it is a legal scene description. If the computer finds an error, it will report it in a box at the top of the text area. (You can make this box go away by clicking on it, if you like.) If there are no errors, the applet will switch to graphics mode and the scene will be displayed.

The second mode is a graphics mode in which the applet displays the rendered scene. In this mode, there is a column of controls along the right edge of the applet. This column contains:

- Frame number label -- shows the currently displayed frame number during an animation. (If you find this distracting, click on it.)

- New Program button -- takes you back to program mode, with a new blank text area.
- Show Program button -- takes you back to program mode, to the program that produced the current scene.
- Go button -- used to restart an animation that has been paused.
- Pause button -- stops an animation, and makes the Go, Next Frame, and Previous Frame buttons available.
- Next Frame button -- goes to the next frame of a paused animation. When the last frame is reached, it goes back to frame zero.
- Previous Frame button -- goes to the previous frame an animation. When frame zero is reached, it goes to the final frame.
- Speed pop-up menu -- selects the desired speed for animation, ranging from one frame per second to 30 frames per second. However, note that the actual rate can be less than the selected rate, depending on the complexity of the scene and on your computer's speed.
- Looping pop-up menu -- Specifies what happens in an animation when the last frame is reached. If this menu is set to "Loop," the animation is repeated starting back at frame zero. If it is set to "Back-And-Forth," the animation is played backwards, then forwards again, and so on. If it is set to "Once Through," the animation is paused when it reaches the final frame.

At the very top of the applet, there is another pop-up menu. This menu is available in both programming mode and graphics mode, and you can use it to switch between modes and among all the programs that the applet knows about. The first item in this pop-up menu is "Graphics". When the applet is in programming mode, selecting "Graphics" from the pop-up menu is exactly the same as clicking on the "RENDER" button. That is, the applet will check the current program for errors and, if no errors are found, will switch to graphics mode and display the rendered scene.

The second item in the pop-up menu is "[New]". Choosing this item will let you write a new program, starting with an empty text area. (This is the same as clicking the New button while in graphics mode.) The new program will have the name "Untitled 1" or "Untitled 2" or....

The remaining items in the menu are names of programs. Selecting one of these names will take you directly to that program. If you do this while the program is in graphics mode, it will switch back to programming mode.

The Scene Description Language

Scenes in xModels are described in terms of three coordinates, x, y, and z. The computer's screen is the xy-plane, with the origin (0,0) at the center of the graphics display area. The positive y-axis extends upwards from this point, and the positive x-axis points to the right. The z-axis points directly out from the screen towards the viewer, so that points in front of the screen have positive z-values, and points behind the screen have negative z-values. This

is a standard coordinate system for three-dimensional computer graphics.

The graphics display area includes the square region with -10 < x <10 and -10 < y <10. Since the display area might not be square, it can actually extend beyond this range in one direction. There is no way to increase or decrease the basic square region that is displayed, so scenes must be sized to fit into this region. (It is easy to scale objects up or down in size to fit.)

The three-dimensional world is projected onto the xy-plane from a point on the positive z-axis. The z-coordinate of this point is called the viewDistance, and its value can be specified as part of a scene description. (This name is somewhat deceptive. Since the display area always shows the same square region, objects on the xy-plane don't look smaller as the viewDistance increases. They just look more squashed in the z-direction.) The viewDistance can be set to "infinity" to give what is called a parallel projection.

Fundamentally, a scene description for xModels is a list of objects that appear in the screen, plus a few special commands. Special commands are used to specify colors, view distance, and animation parameters. There are a few basic named objects, such as circle and cone. There is also a command for defining new named objects. Geometric transformations such as rotate and scale can be applied to objects to specify their size, position, and orientation. You can make complex, hierarchical objects that contain other objects, which can have their own transformations. Animation is done by letting parameters, such as the scaling factor in a scale transformation, vary through a range of values as the animation proceeds from frame to frame. The rest of this file gives the details of the scene description language.

I should note that a scene description can contain comments. A comment begins with a semicolon (;) and continues until the end of the line. For multiline comments, a semicolon is required on each line. A comment doesn't have to start at the beginning of a line.

Except in the case of comments, xModels doesn't pay attention to ends-of-line. They are treated just like spaces. You can lay out your program any way you like on the page.

The xModels language is not case-sensitive: Upper and lower case letters are considered to be equivalent. Names can consist of letters, digits, and the underscore character (_). A name must begin with a letter or with an underscore. Names can be of any length. A word with a predefined meaning, such as square or animate, cannot be reused as the name of a defined object.

Numbers can include decimal points and exponential notation. For example: -17, 3.14, .5, 1.2e5. With just a few exceptions, anyplace where a number can appear in a program, a number range can also appear. Number ranges are used with animation, as described later in this file. Examples of number ranges are 1:10, -1:3:10, and 12::0. (The only places where number ranges cannot be substituted for numbers are in the animate command and for the first parameter of the lathe or extrude command.

As a final preliminary point, I will note that commas can be included in a program to help make it more readable by humans. However, the computer ignores commas. More specifically, it treats them exactly the same as spaces.

Special Commands

The special commands in xModels are animate, viewDistance, background, define, and various commands for specifying the color to be used for drawing.

If the animate command occurs at all in a program, **it must be the first word** in the program (not counting any comments the might precede it). This command, which is used to specify the number of frames in an animation, is defined below.

The viewDistance command specifies the point along the z-axis that is used as the center of projection. An (x,y,z) point is projected onto the xy-plane by drawing a line from the center of projection through the point (x,y,z) and finding the (x,y) point where it intersects the xy-plane. Objects behind the projection point are not displayed. The viewDistance command must be followed by a parameter that specifies the z-coordinate of the projection point. The parameter can be any positive number. If the scene is an animation, the parameter can be a number range. The parameter can also be the word infinity which specifies projection from infinity. A program can contain at most one viewDistance command. If none is specified, a default value of 20 is used. The viewDistance command does not have to come at the beginning of the program. However, it applies to the entire scene in any case.

The background command is used to specified the background color for the scene. This command must be followed by a color specification. The color can be specified by one of the specific color words listed below. It can also be given using the rgb or hsb color command. Since rgb and hsb can use number ranges as parameters, the background color can change from one frame of an animation to the next. The default background color is white. The background command does not have to appear at the beginning of the program.

The define is used to give a name to an object. Once a named object has been defined, it can be used in the same way as any of the built-in objects, including in the definitions of other named objects. The define must be followed by the name of the object, and then by the specification of the object itself. The object is generally a complex object, enclosed between [and], but that is not a requirement. Defining an object does **not** make the object appear in the scene. To do that, you have to include the object name as part of the scene description. The following example defines a "wheel" to consist of a circle and three lines:

```
define wheel [
    circle
    line
    line rotate 60
    line rotate -60
]
```

A color can be specified by one of the following color names: red, green, blue, cyan, magenta, yellow, black, white, gray, lightGray, or darkGray. Color can also be specified by the rgb command or the hsb command. The rgb command lets you specify a color by giving its red, blue, and green components. It requires three parameters to specify the three values. The values must be between 0 and 1, inclusive. For example:

```
rgb  1, 0.5, 0.5    ;  specifies a pinkish color
```

```
rgb  0:1 0 0        ;  specifies a range of colors
                    ;                 from black to red
```

The hsb command is similar, except it specifies a color by giving its hue, brightness, and saturation components. Again, these values must be between 0 and 1. (You can look this up in a graphics textbook if you don't know what it means.)

When a color command is given, it sets the drawing color to be used for all subsequent objects, up until the next color change. Color changes inside complex objects, that is between [and], have no effect past the closing]. The default drawing color is black.

Objects

There are six predefined objects in xModels: line, square, circle, cube, cone, and cylinder. These objects are sized so that each object just fits inside a 1-by-1-by-1 cube, centered at the origin. The line object stretches along the x-axis from (-0.5,0) to (0.5,0). The square object has vertices at (-0.5,-0.5), (0.5,-0.5), (0.5,0.5), and (-0.5,0.5). The circle has center (0,0) and radius 0.5. The cone is oriented to point upwards along the y-axis. The cylinder also has a vertical orientation. To include one of these objects, just list its name in the scene description. Usually, the name will be followed by a transformation that affects the size, position, and orientation of the object.

There are also four commands for creating an object out of a list of points. These commands are polygon, polygon_3d, lathe, and extrude. The polygon command takes a list of parameters that specify a sequence of (x,y) points. The polygon consists of these points joined by lines. Note that there must be an even number of parameters, since there are two parameters per point. For example, the following command creates a triangle:

```
polygon  0,0 4,0 2,2
```

It's legal to have a polygon command with just two points. In that case, it specifies a line. The polygon-3d is similar, except that it takes a list of (x,y,z) points.

The lathe command takes a list of (x,y) points, joins those points with line segments, and then rotates the resulting curve about the y-axis to obtain a three-dimensional object. The original curve is actually copied several times, at different angles of rotation. These copies are then joined with further line segments. The number of copies must be specified as the first parameter to the lathe command. The remaining parameters specify the (x,y) points. For example, the following command makes four rotated copies of the line segment from (0,5) to (3,0) and then connects them with lines to produce a pyramid:

```
lathe 4  0,5  3,0
```

It is legal to have a lathe command with just one point. The result will be a regular n-sided polygon lying in the xz-plane.

The extrude command is similar to lathe in that it makes several copies of a curve that lies in the xy-plane, and it then joins those copies with lines. However, extrude makes the copies by translating the original curve along the z-axis. Each copy is separated from the next by one unit along the z-axis. The z-values are centered about 0. for example, for extrude 2, the

two z-values are -0.5 and 0.5.

Besides all these basic objects, you can make complex objects. A complex object is a list of items enclosed between a left bracket, [, and a right bracket,]. It can include objects and color specifications. Each object in a complex object can be followed by its own set of transformations, as described below. The objects can include basic objects, named objects created with the define command, and nested complex objects. Because of this ability to nest complex objects inside other complex objects, xModels is said to use hierarchical models.

Transformations

Any object can be followed by a list of one or more transformations that affect the size, position, and orientation of that object. This includes complex objects. Any transformation applied to a complex object is applied to that object as a whole. If an object inside a complex object has its own transformations, they are applied first, followed by the overall transformation of the object as a whole.

A transformation consists of a word specifying the type of transformation, followed by one or more parameters. For example, the command rotate 30 specifies that the object is to be rotated through an angle of 30 degrees about the z-axis. Some transformations take a variable number of parameters. For example, scale 3 will magnify the object by a factor of 3 in all directions, while scale 2,6,0.5 will scale it by factors of 2 in the x-direction, 6 in the y-direction, and 0.5 in the z-direction.

When an object is followed by several transformations, they are applied in the order given. For example in,

```
square  xtranslate 5  rotate 30
```

the square is first translated 5 units in the positive x-direction, and is then rotated by 30 degrees about the origin. Putting the transformations in the opposite order:

```
square  rotate 30  xtranslate 5
```

gives a different result, since the square is first rotated and then translated.

Here is a list of the transformations used in xModels, where A, B, C, D, E, F, and G are numbers (or, in the case of an animation, number ranges):

- scale A B C -- Scales by factors of A in the x direction, B in the y direction, and C in the z direction. Scaling by a fractional amount makes an object smaller. Scaling by a negative amount reflects the object through the corresponding coordinate plane. The scaling is centered at the origin; all other points move away from or towards the origin.
- scale A B -- same as "scale A B B".
- scale A -- same as "scale A A A".
- xscale A -- same as "scale A 1 1"; scales in x-direction only.
- yscale A -- same as "scale 1 A 1"; scales in y-direction only.
- zscale A -- same as "scale 1 1 A"; scales in z-direction only.

- translate A B C -- Moves each point (x,y,z) to (x+A,y+B,z+C). The effect is to move the object A units in the x-direction, B units in the y-direction, and C units in the z-direction.
- translate A B -- same as "translate A B 0".
- translate A -- same as "translate A 0 0".
- xtranslate A -- same as "translate A 0 0"; moves an object A units in the x-direction.
- ytranslate A -- same as "translate 0 A 0"; moves an object A units in the y-direction.
- ztranslate A -- same as "translate 0 0 A"; moves an object A units in the z-direction.
- xrotate A -- Rotates everything though an angle of A degrees about the x-axis. The x-axis is fixed, and everything else pivots around it. The direction to use for positive angles is determined by the "right-hand rule": Point the thumb of your right hand in the direction of the positive axis, and the fingers of your right hand will curl in the direction of a positive angle.
- yrotate A -- Rotates everything though an angle of A degrees about the y-axis.
- zrotate A -- Rotates everything though an angle of A degrees about the z-axis.
- rotate A -- same as "zrotate A". In the xy-plane, this looks like a rotation about the origin, with positive angles representing counterclockwise rotation and negative angles, clockwise rotation.
- rotate A about B C -- Rotate through an angle of A degrees about the line that starts at the point (B,C,0) and extends in the same direction as the positive z-axis. In the xy-plane, this is just rotation about the point (B,C).
- rotate A about line B C D -- Rotate by an angle of A degrees about the line that goes from the origin, (0,0,0), to the point (B,C,D). (The two words "about line" can also be written as a single word "aboutline".) If (B,C,D) = (0,0,0), nothing happens.
- rotate A about line B C D E F G -- Rotate by an angle of A degrees about the line that goes from the point (B,C,D) to the point (E,F,G).
- xSkew A -- This is the transformation which moves (x,y,z) to (x+Ay,y,z). Lines that were perpendicular to the xz-plane are tilted (or "skewed") to the left or right.
- ySkew A -- This is the transformation which moves (x,y,z) to (x,y+Ax,z).
- xyShear A B -- This is the transformation which moves (x,y,z) to (x+Az,y+Bz,z). Lines perpendicular to the xy-plane are skewed. (I have not included a complete set of skew/shear transformations, because I don't expect them to be used much.)

Note that although xModels is a 3-dimensional graphics program, you can restrict yourself to two dimensions if you want. The names and semantics of the transformations were chosen so that all the two-dimensional transformations are available with reasonable names. This explains the otherwise odd rotate A about B C, for example.

It is important to understand what a list of transformation does to an object. All the transformations are applied to the object before it is displayed. So, square scale 2 5 is just a way of specifying a 2-by-5 rectangle, and circle translate 5 is just a way of specifying a circle centered at the point (5,0). You don't actually see the object moving or changing shape. For that, you have to use an animation and specify a range of values for the

transformation. In that case, each frame of the animation gets its own transformation to specify the shape or position of the object in that frame. The object can change from one frame to the next, because a different transformation is used in each frame.

Animation

An animation is just a sequence of frames. Each frame contains a separate image. If the images don't change too much from one frame to the next, the viewer will perceive continuous motion as the frames are played back in rapid succession.

In xModels, a scene description that starts with the command animate N, where N is a positive number, is an animation with N+1 frames. The frames are numbered from 0 to N. (You should think of N as the number of intervals between frames.) Then, to get any kind of motion or change in the animation, you need to make some quantity change from frame to frame. This is done by using number range in place of a number. A number range consists of a starting value, followed by a colon, followed by a final value. In each frame of the animation, the number range represents a different value. For example, in an 11-frame animation, the number range 0:5 represents 0 in frame 0, 0.5 in frame 1, 1 in frame 2,... and 5 in frame 11. Thus, the scene description:

```
animate 10
circle scale 0:5
```

shows a circle that grows from a size of 0 in the first frame to a size of 5 in the last frame. And

```
animate 30
square scale 5 rotate 0:90
```

shows a 5-by-5 square pivoting through a 90-degree turn about the origin. Note that the value 5 in this example is the same in each frame. You don't need to use a number range for each value in an animation -- only for the values that you actually want to change during the animation.

By adding additional parameters to the animate command, you can make "segmented animations.". For example, the command animate 30 50 specifies an animation with two segments. The first segment has 31 frames, and the second segment has 51. The final frame of the first segment is also the first frame of the second segment, so there are 81 frames in all. (Remember that the numbers 30 and 50 actually specify the intervals between frames.) An animation can contain any number of segments. The first frame of the animation, the last frame, and any frame that is on the boundary between two segments are called key frames.

A number range used in a segmented animation must specify a value for each of the key frames. Thus, it must have exactly as many colons as there are segments in the animation. For example, the number range 10:5:7 could be used in a two-segment animation. During the first segment, the value ranges from 10 to 5, and during the second segment, it ranges from 5 to 7. The number range 0:0:10 has the constant value 0 throughout the first segment, and then its value ranges from 0 to 10 during the second segment. Sometimes, you want a quantity that changes at a constant rate during the whole animation, rather than at different rates in different segments. The notation for doing this is to use two or more colons in a row,

with no numbers between. For example, 0::10 represents a quantity that varies evenly from 0 to 10 across both segments of a two-segment animation.

David Eck (eck@hws.edu), June 1997

xModels Tutorial Examples

The xModels Applet displays wireframe models of still images and animations which are specified in a simple scene description language. The applet at the bottom of this page will try to load six tutorial examples. To read an example, select it from the pop-up menu at top of the applet. Then, to see the image it produces, click on the "RENDER" button or select "Graphics" from the pop-up menu. For full information on the applet and the language, see the xModels Info page. The six tutorial files are also directly available as text files using the following links: [1], [2], [3], [4], [5], [6],

---

(Java not available.)

David Eck (eck@hws.edu), June 1997

```
; xModels Tutorial 1:  Basic Ideas

; A program written in the xModels scene description
; language is basically a list of the objects in the
; scene, with "transformations" that say how the objects
; are sized, oriented, and placed.  There are a few
; other things that can occur in programs.  This
; program contains examples of the basic objects
; and some of the transformations that can be used
; in a program.  It also shows how to use color.

; A semicolon, like the one at the left is the beginning
; of a "comment" which is ignored by the computer.  A
; comment ends at the end of the line.


square  ; A square is one type of basic object.
        ; Putting its name in the scene description
        ; adds a square to the image.  A basic square
        ; is a rather small square at the center of
        ; the image.

square scale 3 ; This is ANOTHER square in the same
        ; image.  The transformation "scale 3" placed
        ; after an object causes that object to be
        ; magnified by a factor of 3.  Every point in
        ; the object is moved away from the origin to
        ; three times its original distance.  Since
        ; the basic square was centered at the origin,
        ; a scaled square is also centered at the
        ; origin.

red ; The command "red" tells the computer to draw the
    ; following objects, up to the next color change
    ; in red.  Other color commands include green,
    ; blue, cyan, magenta, yellow, black, white, and gray.
    ; (It is also possible to specify "RGB" colors
    ; and "HSB" colors.  See the full documentation for
    ; details.)

circle translate 6 6 ; A "circle" is a small circle
    ; at the center of the screen, but the "translate"
    ; transformation is used here to move it over 6
    ; units and up 6 units from its original position.
    ; This will show in the image as a small, red
    ; circle centered at the point (6,6).

circle scale 5 translate -6 6  ; Transformations can
    ; be combined.  The circle is FIRST scaled by
    ; a factor of 5, and then the resulting object
    ; is moved 6 units to the left and 6 units up.
    ; The result is a circle of diameter 5 centered
    ; at the point (-6,6).  The drawing color is
    ; still red for this circle,

blue  ; Color changes to blue for following objects

square scale 5    ; Start with a square,
   rotate 30      ; rotate it 30 degrees about the origin,
   translate 6 -6 ; then move it 6 units over and 6 down.
      ; (Note that you don't have to list all the
```

```
      ;  transformations on one line.)

square scale 6 2  ; A scale command can also have two or
  translate -5 -5 ; three parameters.  "scale 2 6" magnifies
                  ; by a factor of 6 horizontally and
                  ; 2 vertically.  Scaling a square in
                  ; this way gives a rectangle.

green

cube scale 4 translate 5 0  ; A 1-by-1-by-1 cube is
    ; magnified by a factor of 4 and translated
    ; 5 units to the right,

cone scale 4 translate -5 0 ; The cone shows up
    ; 5 units to the left of its default position
    ; at the origin.

magenta ; A bright purple-ish color

cylinder scale 4 translate 0 5 ; The cylinder
    ; shows up 5 units above the origin

line  scale 6 translate 0 -7 ; Finally, a humble line.
    ; The basic line is one unit long, extending along
    ; the x-axis from (-0.5,0) to (0.5,0).
    ; Here, it is scaled to a length of 6 and moved
    ; down 7 units.

; You can render this scene by clicking on the "RENDER"
; button.  You should be able to find all the objects
; listed above in the picture.  The colors will help
; you identify them.  The 3-dimensional objects will
; look sort-of distorted because they are being projected
; onto the screen from the point (0,0,20).  The near side
; of the cube, for example, looks bigger than the far
; side (as it should, really).
```

```
; xModels Tutorial 2:  Animation

; Moving images are more interesting than still images.
; An animated image is displayed on the computer screen
; by showing a sequence of "frames" in rapid succession,
; with small changes from one frame to the next.  In
; xModels, a number range such as 1:5 is used to specify
; a value that changes from frame to frame in the animation.
; For 1:5, the value is 1 in the first frame and is 5 in
; the last frame.  Between the first and last frame, the
; value changes by the same amount in each frame.  This
; file defines an animation for xModels.


animate 60  ; An animated scene MUST begin with the
            ; word animate, followed by a specification of
            ; the number of frames in the animation.  Here,
            ; there will be 61 frames, numbered from 0 to 60.

background yellow ; The "background" command is used to
            ; specify a background color for the scene.
            ; (Used here just for fun.)

square scale 1:5 rotate 0:90 ; Specifies a different
            ; square in each scene.  The first square has
            ; size 1 and is not rotated at all.  The
            ; second square is a little bigger and is
            ; rotated a bit (by 1.5 degrees, to be exact).
            ; The last square will be 5 units large and
            ; rotated through an angle of 90 degrees.

circle scale 1:5 5:1 translate 6 6 ; This starts out
       ; as a 5-by-1 ellipse and ends up as a 1-by-5
       ; ellipse.  The translate command moves it
       ; to the upper right corner of the display area.

blue

cube scale 2 translate -6:6 -7  ; The cube moves
      ; along the bottom of the screen, from the
      ; point (-6,7) to (6,7).

line scale 4      ; A four-units long line
   rotate 0:180   ; rotate from 0 to 180 degrees.
   translate 0 7  ; The rotating line is moved 7 units
                  ; upwards.  (Note: this line is blue.)

red

line scale 4       ; A four-units long line
   translate 0 7   ; is moved seven units upwards
   rotate 0:180    ; and from there rotates through
                   ; 180 degrees about the origin.  (Note
                   ; that the order in which the
                   ; transformations are applied makes a
                   ; big difference.  (This line is red.)

; (When you render this animation, note how the image of
;  the cube changes as it moves.)
```

```
; xModels Tutorial 3: Three-dimensional Scenes.

; The two previous tutorials included three-dimensional
; objects, but the transformations that were applied
; only affected two dimensions.  Transformations can,
; in fact affect all three dimensions.  This file
; discusses three-dimensional transformations and
; has more information about transformations.

animate 60  ; This will be a 61-frame animation

circle scale 8      ; The "translate" command can take three
   translate 0 0 6  ; parameters, giving changes in x,y and z.
   yrotate 0:360    ; A translation of 6 units in the z
                    ; direction moves the circle forward
                    ; six units towards the viewer.  From that
                    ; position, it is rotated 360 degrees
                    ; around the y-axis.  This sends it
                    ; towards the right and away from the
                    ; viewer, then all the way around the
                    ; axis and back to its original position.

red

square xscale 5     ; "xscale 3" is short for "scale 3 1 1".
       ytranslate 8 ; This means "translate 0 8 0".  Similarly,
       xrotate 0:360; there are commands: yscale, zscale,
                    ; xtranslate, ztranslate, xrotate, and
                    ; zrotate.  zrotate is actually the same
                    ; as the rotate command, since it just
                    ; rotates everything around the z-axis,
                    ; which points out of the screen towards the
                    ; user.  This rectangle starts at the top
                    ; of the screen and rotates around the
                    ; x-axis.

blue

cube scale 2 translate -12:12 2:-2 -6:6
   ; A small blue cube moves from (-12,2,-6) to (12,-2,6).


green

cylinder scale 5 rotate 0:360 about line 1 1 0
   ; The cylinder rotates about the line that extends
   ; from (0,0,0) to (1,1,0).  The "rotate about line"
   ; command lets you rotate objects about lines other
   ; than the coordinate axes.
```

```
; xModels Tutorial 4:  Complex Objects

; The real power of xModels (what there is of it) comes from
; fact that it is a "hierarchical" modeling language.  You
; can combine several objects into one complex object,
; which can then be treated as a unit.  Transformations
; applied to a complex object apply to the object as a
; whole.  A complex object can even include other complex
; objects.  It is also possible to define a name to
; represent a complex object.  Then the name can be
; used in the scene, just as if it were one of the
; predefined objects such as "square" and "cone".

animate 60  ; This will be a 60-frame animation

[       ; A complex object begins with a "[" and ends with "]".

   square scale 4  ; Transformations can be applied to the
                   ; objects inside a complex object.

   [   ; Complex objects can be nested inside other objects.
      circle translate 5
      circle translate -5
   ] rotate 45:-45  ; The rotation command applies to the
                    ; entire complex object, consisting of
                    ; two circles.

   red ; A complex object can contain color commands
       ; in addition to objects

   line scale 8 rotate 90 xtranslate -4:4

] scale 0.5:1.2  ; The entire complex object grows from
                 ; half of its basic size to 1.2 times
                 ; that size.

; Note: At this point, the drawing color is black.
;       The color change INSIDE the object has no
;       effect outside.  (On the other hand, a color
;       change made BEFORE the object does "leak into"
;       a complex object.


define wheel [  ; Begin the defintino of an object named "wheel".
   circle
   line              ; A wheel contains a circle and three
   line rotate 60    ; lines, which act as spokes.
   line rotate -60
]   ; The end of the definition

; "Wheel" has been defined, but no wheel has been put into
; the scene.  Now we add two wheels with different
; colors, sizes, and rotation speeds to the scene:

wheel scale 2 rotate 0:360 xtranslate -8  ; This one is black.

green

wheel scale 4 rotate 0:-180 xtranslate 8
```

```
; A defined object can be used inside the definition
; of another object.


define wagon [
   red
   square scale 4 2
   blue
   wheel scale 2 rotate 0:-720 translate -2 -1
   wheel scale 2 rotate 0:-720 translate 2 -1
]

wagon ytranslate -8        ; A wagon across the bottom of
      xtranslate -10:10    ;    the screen
```

```
; xModels Tutorial 6:  Polygons, Lathing and Extrusion

; xModels has a "polygon" for creating polygon objects.
; The polygon command takes a list of (x,y) points and
; connects them with line segments.  An example is given below.
; Modeling real objects (like cars or faces) in 3D requires
; that they be approximated with large numbers of polygons,
; perhaps thousands of polygons per object.  You won't want
; to do anything so complicated with xModels-3D.  But it
; does have two ways of producing fairly complicated, but also
; fairly regular, objects.  The methods are called lathing
; and extrusion.  The idea is to take a figure consisting
; of a connected sequence of line segments in the xy-plane.
; Some specified number of copies of this figure are made and
; then more line segments are added to join the copies.


animate 30


; In lathing, the copies are made by rotating the original
; figure about the y-axis.  The command for doing lathing
; is "lathe":

lathe 8  0,5 1,1 3,0 1,-1 0,-5  ; The first parameter is the number of
                                ; copies; then comes the list of points.
     yrotate 0:45  ; You can apply transformations
                   ; to lathed objects.

; Compare the figure produced by the above to the polygon
; made with the same points:

polygon 0,5 1,1 3,0 1,-1 0,-5
      xtranslate 6  ; Move it over so you can see it.

; The extra vertical line from (0,-5) back to (0,5), which
; is added to close the polygon, is not used in the lathing
; operation.  The remaining sides form the figure that is
; rotated about the y-axis by the lathe command.

; Here are more examples:

red

lathe 12 1,2 2,-2  ; Lathing a single line, to make a "lamp shade".
    xrotate 0:360  ; Tumble it about the x-axis
    xtranslate -7

blue

lathe 4  2 1:-1  3 0  4 0 ; You can use number ranges
     translate -4 7       ;        in the point list!

; Extrusion is not quite so interesting as lathing.  In
; extrusion, the copies of the original figure are made
; by translating the original in the z-direction, instead
; of by rotating it.  The copies are spaced one unit apart,
; although you can change that, of course, by scaling
; in the z direction.  The extruded figure extends
; equally far behind the xy-plane as it does in front
; of it.  The command for doing extrusion is "extrude".
```

```
; Here, for example, is a 3D "E":

cyan

extrude 2                           ; The number of copies.
        0,0 0,5 3,5 3,4 1,4 1,3     ; The list of points.
        2,3 2,2 1,2 1,1 3,1 3,0 0,0
   translate -7.5,-9.5,0

; And here is an example that rotates so you can see
; it better:

magenta

extrude 5 -2,-2 0,2 2,-2 ; Extrude 5 copies of an inverted "V"
    yrotate -30:30
    translate 3 -6.5
```

```
; Tutorial 6: Segments

; An animation can have "segments."  For example, the command
; "animate 30 50" creates an animation with a segment that
; contains 31 frames, followed by a segment that contains
; 51 frames.  The segments are "spliced" together because
; the final frame of the first segment is the same as
; the first frame of the second segment.  This file defines
; an animation with four segments.

animate 90 90 90 90

define flap [
   square scale 5 3
          xtranslate 5
 ]

define paddles [
   hsb 0 1 1  ; Gives a color by hue, saturation, brightness.
   flap yrotate 0:0:0:0:360
   hsb 0.125 1 1
   flap yrotate 0:45:45:45:360      ; Eight flaps rotate
   hsb 0.25 1 1                     ; into position during
   flap yrotate 0:90:90:90:360      ; the first segment of
   hsb 0.375 1 1                    ; the animation.  After
   flap yrotate 0:135:135:135:360   ; that, the y-rotation of
   hsb 0.5 1 1                      ; each flap remains
   flap yrotate 0:180:180:180:360   ; constant for the next
   hsb 0.625 1 1                    ; two segments.  Then
   flap yrotate 0:225:225:225:360   ; all the flaps rotate back
   hsb 0.75 1 1                     ; to their original
   flap yrotate 0:270:270:270:360   ; positions.
   hsb 0.875 1 1
   flap yrotate 0:305:305:305:360
]


background black

paddles
   yrotate 0:0:360:720:720
   xrotate 15:15:15:375:375
; The entire paddles object rotates around the y-axis
; during each of the second and third segments.  In the
; third segment, the object is ALSO rotating about the
; x-axis.  (There is an extra, constant 15 degree rotation
; about the x-axis to make the object easiest to view.
```

README file for the tmcm-java-web-site archive March 2000

The tmcm-java directory in this archive contains Web pages and applets that are also available on line at:

    http://math.hws.edu/eck/TMCM/java/

The Web pages include some lab worksheets that were written to be used with my introductory computer science textbook, The Most Complex Machine. They can also be used on their own. There are also some information/tutorial pages for each applet. The main Web page is index.html, and it includes links to all the other pages.

If you would like to use any of the material in this archive for commercial purposes, please contact me for permission. Contact information is given below.

The applets can be freely used for any non-commercial purpose.

The lab worksheets can be used for any private, non-commercial purpose, but I ask that they not be used as an official part of a course unless my textbook is adopted for that course. (However, I will consider making exceptions to this.)

You are welcome to post the entire, unmodified contents of this archive on your own Web server.

You are also welcome to create non-commercial Web pages that use the applets. If you do this, you might want to create your own sample input files for the applets. To do that, you will probably want to download the stand-alone application version of the applets. That version is in an archive named tmcm-java-apps which is available through a link at the address: http://math.hws.edu/TMCM/newjava/DownloadingAndInfo.html

USING THE APPLETS ON YOUR OWN WEB PAGES:

To use one of the applets on a Web page, that page must have access to the compiled Java program for that applet. You can find these programs in the directory named "classes" inside the tmcm-java directory. The classes are in ".zip" files. There is one zip file for each applet: DataReps.zip, xComputer.zip, xLogicCircuits.zip, and so on. To use one of the applets on a Web page, you can copy the corresponding zip file into the same directory as the HTML source file for your Web page. The HTML source file will have an <applet> tag to load the applet. This applet tag must refer to the zip file and to the applet class. For example, the applet class for the xLogicCircuits applet is "tmcm.xLogicCircuitsApplet.class". An <applet> tag for using this applet has the form:

```
<p align=center>
<applet archive="xLogicCircuits.zip"
        code="tmcm.xLogicCircuitsApplet.class"
        height=380 width=500>
</applet>
```

```
    </p>

The classes for the other applets are named similarly:
tmcm.DataRepsApplet.class, tmcm.xComputerApplet.class, and
so on.  In addition to these applets, which appear right on
the web page, there are "launcher" versions of the applets.
In the launcher version, only a button appears on the Web
page.  When the user clicks the button, the applet is opened
in a separate window.  The names for the launcher versions
are tmcm.DataRepsLauncher.class, tmcm.xLogicCircuitsLauncher.class
and so on.  For example, to use the launcher version of
xLogicCircuits, you could use the <applet> tag

    <p align=center>
    <applet archive="xLogicCircuits.zip"
            code="tmcm/xLogicCircuitsLauncher.class"
            width=180 height=30>
    </applet>
    </p>

Some of the applets can load sample input files.  Such files
can be created using the "Save" button of one of the applets.
However, this button will generally not be functional when you
are running the applet in a Web browser.  If you want to use
the "Save" button, get the application version of the applets,
mentioned above.  (Or, if you have the Java Development Kit,
try running the applet with the appletviewer command.)

To be used by an applet, a sample input file should be in the
same directory as the HTML source file for the Web page that
contains the applet.  The names of the sample input files must
be specified as "params" in the <applet> tag.  For example, the
xLogicCircuits applet can read one sample file.  The file
is specified in a param named "LOAD".  For example, if
you want xLogicCircuits to load a sample file named
"SampleCircuits.txt", use the applet tag:

    <p align=center>
    <applet archive="xLogicCircuits.zip"
            code="tmcm.xLogicCircuitsApplet.class"
            height=380 width=500>
        <param name = "LOAD" value = "SampleCircuits.txt">
    </applet>
    </p>

The param name, "LOAD", must be given in uppercase letters, as
shown.  Param names are case-sensitive.  You can also use an
input file with the launcher version of the applet:

    <p align=center>
    <applet archive="xLogicCircuits.zip"
            code="tmcm.xLogicCircuitsLauncher.class"
            width=180 height=30>
       <param name="LOAD" value="SampleCircuits.txt">
    </applet>
    </p>

The xComputer, xTuringMachine, xTurtle, and xModels applets can
load several sample files.  The files must be specified using
the param names "URL", "URL1", "URL2", and so on.  You have to
be careful to use the right names, without any omissions.
```

(If there is no URL2, for example, the applet won't even check for URL3.) For example, to use four sample input files with the launcher version of the xComputer applet, you could use the following tag on your web page.

```
    <applet archive="xComputer.zip"
            code="tmcm.xComputerLauncher.class"
            width=150 height=30>
       <param name="URL" value="SimpleCounter.txt">
       <param name="URL1" value="MultiplyByAdding.txt">
       <param name="URL2" value="ThreeNPlusOne.txt">
       <param name="URL3" value="ListSum.txt">
    </applet>
```

The preceding tags assume that the zip files and sample input files are in the same directory with the HTML source file of the Web page. It's possible to put them in other directories. If the zip file is not in the directory with the HTML file, then you must specify a codebase in the applet tag. The codebase in an <applet> tag is the directory that contains the compiled Java code for the applet. It is specified relative to the directory that contains the Web page. For example:

```
    <p align=center>
    <applet codebase="../classes/" archive="xLogicCircuits.zip"
            code="tmcm.xLogicCircuitsLauncher.class"
            width=180 height=30>
       <param name="LOAD" value="SampleCircuit.txt">
    </applet>
    </p>
```

Here, the codebase directory is found by going up to the directory that contains the Web page (specified as "../") and then looking in that directory for a directory named "classes/". (I've found that the "/" at the end of the directory name is necessary, at least for some browsers.)

It's a good idea to have just one copy of a .zip file, even if you are going to use the applet on several Web pages. Then, if the user visits several of those pages, the Web browser will only have to download one zip file. That's why I put all the zip files in one "classes" directory on my own site.

You can also have sample input files in a different directory from the Web page. Just include the directory name in the name of the input file. For example, "samples/SampleCircuits.txt" or "../models/FirstModel.txt". The name should be given relative to the directory that contains the Web page.

David Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY   14456   USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/

README file for the tmcm-java-apps archive March 2000

This archive contains Java applets and sample input files for the applets. The applets are meant to help teach some of the basic concepts of computer science. They were written for use with my textbook, The Most Complex Machine, but they can also be used independently of the book. For more information, see: http://math.hws.edu/TMCM/java/ (If you want to understand the applets, you will have to look at the material at this address.)

The material in this archive can be freely redistributed and used for non-commercial purposes.

Several Java "applications" are included here that will let you run the applets without a Web browser. One big advantage of doing this is that you will be able to save and load files. For example, you could create sample files to use with the applets on your own Web pages. There are three applications. One simply makes the applets available, without loading any sample files. Another loads the applets with the sample data files used in the labs worksheets from http://math.hws.edu/TMCM/java/. The third application loads the applets with the sample files and tutorial examples from the applet information pages at the same address.

(The idea for the latter two applications is that you could read the Web material in a Web browser at the same time that you run the applets using the applications. This will let you use files with the applets. When an applet is run in a Web browser, it probably won't be able to save and load files. If you've downloaded the labs and tutorials for use on your own computer, you can view them in a Web browser, but in that case, the applets in the Web browser will be even more limited. They probably won't even be able to load the sample files that are used in the labs and tutorials!)

All the sample files are included in this archive. They have names that end with ".txt", and they are really just plain text files that you can read with a text editor, if you want. I know it's messy to have all these files in one directory, but I couldn't find a way to get the applets to work reliably with the sample files, unless the files are in the same directory with the applets.

The applets themselves are contained in the file tmcm.jar. This is a "Java archive file". In addition to the applets, this file contains the three applications described above. The names of these applications are "tmcm.Apps", "tmcm.Labs", and "tmcm.Tutorials". (The funny names arise because the applications are in the Java "package" named tmcm.)

When you run one of these applications, a window will pop up. The window contains several buttons. Click on a button to launch one of the applets. For the applications tmcm.Labs and tmcm.Tutorials, the applets will load the appropriate sample files. For tmcm.Apps, the applets do not load any files. You can move tmcm.jar to another location and still use it to run tmcm.Apps. However, if you want to run tmcm.Labs or tmcm.Tutorials, the tmcm.jar file must be in the same directory as the sample files.

To run the applications, you need the tmcm.jar file plus some additional Java software. Here are specific instructions for various platforms:

For Windows:

If you have Microsoft Internet Explorer with Java support, then you should be able to use the "jview" command in a DOS Window. To run the applications in tmcm.jar, open a DOS window and change to the tmcm-java-apps directory. This is the directory that you got when you unzipped this archive. (Tip: Open a directory window for this directory. Then select the "Run" command from the Start menu and enter "command" in the dialog box that pops up. This will open a DOS window, already set to use tmcm-java-apps as its working directory.) You have to tell jview to put tmcm.jar on its "classpath". This is done by adding the option "-cp tmcm.jar" to the command. So, the commands for running the three applications are:

```
jview  -cp tmcm.jar  tmcm.Apps
jview  -cp tmcm.jar  tmcm.Labs
jview  -cp tmcm.jar  tmcm.Tutorials
```

Alternatively, if you have the JDK (Java Development Kit) installed on your computer, you can use the JDK's "java" command to run the applications. This is similar to the jview command, but unfortunately, you also have to specify the location of the standard system classes. These are contained in a file named "classes.zip" in one of the JDK directories. You need the full path name of this file, such as C:\jdk1.1.8\lib\classes.zip. (This is the correct name if you using version 1.1.8 of the jdk and did the default installation. For other versions of the jdk, only the numbers should be different.) Once you've found this file, the commands for running the applications are:

```
java  -classpath tmcm.jar;C:\jdk1.1.8\lib\classes.zip  tmcm.Apps
java  -classpath tmcm.jar;C:\jdk1.1.8\lib\classes.zip  tmcm.Labs
java  -classpath tmcm.jar;C:\jdk1.1.8\lib\classes.zip  tmcm.Tutorials
```

You can download the JDK for Windows from Sun Microsystem's Web site at: http://java.sun.com/products/jdk/1.1/

For Macintosh:

You need to have the MRJ (Macintosh Runtime for Java) installed on your Macintosh in order to run the applications in tmcm.jar. This might have been installed with your original system. If not, it can be downloaded from: http://www.apple.com/java/ (I think you will need version 2.1.4 or higher, but I haven't tried it with earlier versions.)

The Macintosh version of the tmcm-java-apps archive includes three double-clickable applications. Double-click the program named "Run TMCM Applets" to run tmcm.Apps. You can move this program to another location, as long as you include a copy of the tmcm.jar file in the same directory. The other two programs need both tmcm.jar and all the sample files. Double-click the program named "Run With Lab Examples" to run tmcm.Labs, and double-click the program named "Run With Tutorial Examples" to run tmcm.Tutorials.

For Liunx and UNIX:

If you have the JDK (Java Development Kit) installed on your computer, you can use the JDK's "java" command to run the applications. The JDK is included in most Linux distributions, although it might not have been installed by default. JDK for Solaris can be downloaded from the Sun

```
 Website,   http://java.sun.com/products/jdk/1.1/
 For other versions of UNIX... you're on your own.
    To use the java command with tmcm.jar, you will need to know
 the location of the standard system classes.  These are contained in
 a file named "classes.zip" in the "lib" subdirectory of the JDK
 installation.  You need the full path name of this file.  On my
 SuSE Linux system, this is /usr/lib/java/lib/classes.zip.  If you
 have trouble finding it on your system, try using the command
 "type java" to find out the full path name of the java command.
 The java command is in the "bin" subdirectory of the JDK installation.
 Once you've found the classes.zip file, you can use the following
 commands to run the applications.  Substitute the appropriate
 pathname for my "/usr/lib/java/lib/classes.zip":

    java  -classpath tmcm.jar:/usr/lib/java/lib/classes.zip  tmcm.Apps
    java  -classpath tmcm.jar:/usr/lib/java/lib/classes.zip  tmcm.Labs
    java  -classpath tmcm.jar:/usr/lib/java/lib/classes.zip  tmcm.Tutorials

 (If you plan to use these commands often, I would suggest making a
 shell script or an alias.)

-----------------------------------------------
```

David Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY   14456   USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/

README file for the tmcm-java-source archive March, 2000

This archive contains the source code for a set of Java applets. The applets were written for use with a textbook, The Most Complex Machine, by David Eck. They are meant to help teach some basic concepts of computer science. For more information about the applets and for a set of lab worksheets that use the applets, see: http://math.hws.edu/TMCM/java/

*** NOTE *** This source code was not originally written with the intent of making it public. There are very few comments in the code. Some parts of the code are a bit strange because they were translated from Pascal programs. So, take the code for what it's worth...

The source code is contained in the directory named "tmcm". All the classes for the applets belong to the package named "tmcm" and to its sub-packages. The sub-packages correspond to subdirectories in the tmcm directory.

The applets were written in Java 1.0, so they use many deprecated methods whose use is discouraged in Java 1.1 and later. When the source code is compiled, you might get warning messages about deprecated methods. There will also be MANY warning messages to the effect that a class should not be used outside the file where it is defined, unless the name of the class agrees with the name of the file where it is defined. (The development system that I used when I wrote the applets didn't enforce this rule.) This could cause a problem if you try to compile the files one at a time, but it's OK as long you compile all the files in a directory at the same time (or compile them in the right order). The warning messages are not errors and will not stop the files from being compiled.

The code can be compiled with the "javac" command from the JDK (Java Development Kit). To use this command with classes that are defined in the tmcm package or one of its sub-packages, you must be in the directory that contains the tmcm directory, and you have to specify the full path to the file or files you want to compile. For example, to compile all the files in the package tmcm.xSortLab, you would say

```
     javac tmcm/xSortLab/*.java      in Linux or UNIX
or
     javac tmcm\xSortLab\*.java      in a DOS command window.
```

For convenience, I have included script files that will compile all the classes in the tmcm file. For Linux/UNIX, there is a shell script named "compile.sh". For DOS/Windows, there is a DOS batch file named "compile.bat". These scripts will also build a .jar archive containing all the compiled class files. You can get more information by reading the script files themselves.

(Note for Macintosh users: Theoretically, it should be possible to compile the files using Apple's SDK for Java, but I have not been able to make it work. The applets were originally written

with CodeWarrior for Macintosh.)

David Eck
Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, NY 14456 USA
Email: eck@hws.edu
WWW: http://math.hws.edu/eck/