Simple Logo (“Slogo”)

CS32 Assignment 3

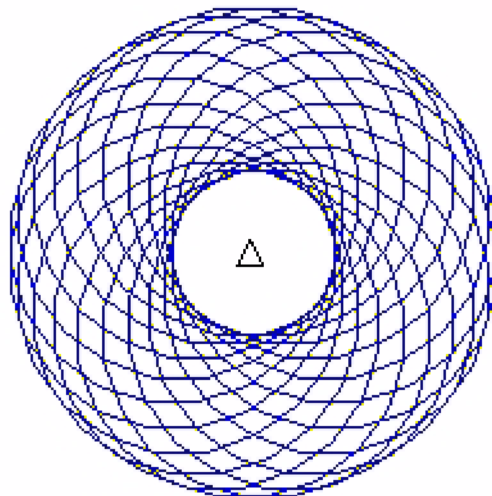
Due Dates:

Assignment Out:	February 26, 1998	
Partner’s Chosen:	Feburary 28, 1998	(11:59pm)
Individual Designs Due:	March 2, 1998	(11:59pm)
Group Design & Interfaces Due:	March 5, 1998	(11:59pm)
Logo Due:	March 12, 1998	(9:00pm)

(The following is a paid advertisement for Slogo by the CS32 TA staff...)

“Say hi to our hosts Don ‘Cash’ and ‘Money’ Lapre!”

One special day in my third grade class, they put three Apple IIe’s in the back of Mrs. Bailey’s classroom, and they were all running Logo. Logo, like BASIC, was intended as an educational language, and, like BASIC, is probably responsible for ruining several million potential programmers with its open encouragement of global variables and absurdly long spaghetti-coded subroutines. You’re older now, and have put aside childish programming techniques; the Good Word of Design has filled you from head to toe. But part of you misses the child-like joy of coding in all capital letters and referring to a triangle as a "turtle." CS 15 made you feel guilty about this side of your personality, but it never completely died.



In this assignment, your two conflicting halves will be brought into glorious union: using the software engineering skills you have carefully cultivated since coming to Brown, you will create an interpreter for the language you once knew and loved so well. Logo truly is the Frosted Mini-Wheats of cs 32 projects: the adult in you must carefully engineer a significant piece of soft-

ware; then the kid in you gets to make saucy pictures in Logo. Unfortunately, you can't play until you're done, so get designing, get hacking, and get done.

"It's based on a simple concept, yet the possibilities are limitless!"

Logo is a computer programming language designed for use by learners (in fact, it is used in CS4 for the first programming assignment). The user-friendly, interpreted language based on LISP was designed with the "low floor, high ceiling" principle in mind; that is, the designers of Logo intended for the language to allow novice programmers to get started quickly with writing programs but also wanted the language to be powerful and extensive for more advanced users.

In the early days, Logo was used to control a simple robot, called a turtle. Users could issue commands such as `FORWARD 50` to make the turtle advance 50 steps, or `RIGHT 90` to make it turn ninety degrees. The turtle robot carried a pen, so users could produce drawings on paper, such as the one on the previous page, by controlling the turtle and its pen. The turtle, which has since moved on screen, has become one of the most familiar and important parts of the Logo language. Children who were using the computer for the first time could relate to "talking to the turtle" and could imagine how the turtle moved by "playing turtle," moving their bodies as the turtle would. The turtle also makes learning basic programming concepts easier and more engaging because it provides immediate feedback.

As a dialect of LISP, Logo is a complex and powerful language. For this project you will design and implement a much simplified version of Logo. Simple Logo should retain the features most commonly used by beginning users so that it can be used to provide an introduction to computer programming.

"Terrific! What do I need to start?"

First of all, you need a partner. Then you need to decide who is going to work on which component of this assignment. Only one member of the group will get credit for doing the GUI part. The person that did not get credit for GUI will need to do the graphics component of the next assignment. That way everyone will have an opportunity to hack some GUI code before the final project.

You need to decide who you will be working with by Saturday, February 28th and sign up on the sheet outside the TA room. Once you do this, we will create a unix group that both members of your group are in so that you can restrict access to your files.

"The Making Logo Quick Package"

Simple Logo will consist of a basic turtle graphics package and support for a set of commands allowing the user to control the turtle and the pen using basic programming constructs such as loops and subroutines. When the Simple Logo user launches the interpreter from the command line, it should bring up a turtle shell and a turtle graphics window. The interpreter will need to receive, parse, and execute commands from the user, reporting any errors it encounters along the way.

Simple Logo should provide functionality in the following areas:

- Basic turtle graphics: Slogo users should be able to control the turtle by moving it forwards, backwards, changing its heading, and showing/hiding the turtle. Drawing capabilities should be provided by pen manipulation. The user should also be able to make turtle motion and pen queries.

- Some basic logo instructions, as described in Appendix A
- Workspace management: Users should be able to define and use subroutines. Users should also be able to define and use global variables.
- Control structures: loops and conditionals
- Error handling: Simple Logo should gracefully handle any errors that may occur while parsing and interpreting user-given commands. This includes providing appropriate error messages (we'll leave it up to you to define appropriate).
- File I/O: The user should be able to load logo files into the interpreter and should be able to save the current working environment (composed of user-defined subroutines, variables, as well as turtle and pen information) to a file.

Refer to the table of commands at the end of this handout for a full list of the built in commands which you will need to support.

"Don, This sounds great! Where do I start?"

The Simple Logo project can be broken down into two parts, all of which you are responsible for writing:

- Command parsing, interpretation and maintenance of the global environment
- Turtle Graphics Library (TGL), consisting of a display window and methods to move the turtle and draw with the pen.

Parsing involves processing commands token by token and checking for syntactical errors. Interpretation means executing the code as it is understood by the parser. The execution is likely to require graphical manipulation, which is where the TGL comes in. It should utilize X graphics calls and should be simple and well-defined.

Remember that having a good design is key. You should work with your partner to come up with a solid program design. Hammer out interface specifications and know how the different parts interact and what functionality they will need from one another. Before you start writing code, it might be a good idea to write header files with empty code blocks and compile them together. To move the turtle and draw with the pen you will need to have the parser-interpreter-graphics library interaction in place, so you will need to know exactly how all three parts fit together. As has been said a bazillion times, there is no single perfect design. Feel free to be creative. Express yourself. However, keep in mind that functionality is essential, and simplicity is beauty.

Plan this project in stages. It may be helpful to build complete individual components before attempting to integrate them. For the parser/interpreter, be sure that you can handle simple logo commands, such as the math operations and turtle motion commands. You could also write a test driver for the turtle graphics interface. Then you should concentrate on getting Simple Logo to work with input from a terminal before you start to worry about getting file input/output functional. This program is not an all or nothing project: you can and should get smaller components working before you put everything together.

"That's fantastic! It sounds so easy! How does it work and is it legal?"

The Simple Logo language has been restrictively defined to make writing it a reasonable project. In particular, the limited definition should make it easier to do parsing and error checking.

Writing the parser/interpreter

The logo interpreter could consists of a parser, which breaks up the command and arguments into tokens and decides what action to take, and a command interpreter, which executes the parsed command.

The parser will need to read in a line at a time, from either a terminal or a file, and divide the input string into tokens. Given the restricted definition of the language, there should only be one command per line and the command should be in prefix form, with the command name at the beginning of the input string. There can also be lines which assign a variable to have the value of another variable, a number, or a command. For example, `:x = SUM 40 50` would be valid input. The job of the parser is to match the appropriate command handler with the specified command name. In matching a command name with a handler, remember that you have to search not only the built in commands but also the user-defined subroutines. The parser can then pass control to the handler, also giving the handler the list of arguments.

One way to think about the command interpreter is as a set of command handlers which each know how to execute a specific command. Most of the handlers will be very straightforward: the numeric operation handlers will need to make calculations and the turtle graphic commands will need to make calls to the turtle graphics library. The loops, conditionals, and subroutine definitions will be slightly trickier. Remember that the `repeat`, `if`, and `to` commands are the only commands which will have input on more than one line. Subroutines are user-defined commands with their functionality composed of existing logo commands. You will also have to think about how to handle lists of instructions that correspond to each command. Note that you do not have to handle passing parameters to your subroutines, although your design should allow this simple extension.

Variables will also need to be dealt with. You don't need to implement any concept of scope, so you could just have a table of all of the variables that exist and their current values. Scope would be cool though and we're sure you could handle it, but it's not required. When a variable is passed to a command, its value will need to be looked up.

Error checking

As specified above, we will expect some amount of error checking. It should be fairly easy to check for parsing errors, by checking each issued command against the set of legal commands and user-defined procedures which comprise the Simple Logo name space. Another measure of error checking to take is to count the number of argument tokens for each command and to compare the expected number of arguments with the given number of arguments. You should also think about how to identify and handle bad user input values.

"I'm so excited! Where can I find out more?"

To get a feel for how logo works, we would suggest that you play around with the actual program, which is in /cs/bin/logo.To leave logo, enter the command "bye" in the logo turtle shell. There are some sample logo files in /course/cs032/asgn/logo/examples. Simple Logo should be able to read in and execute the procedures contained in the example files. Note that SimpleLogo isn't compatible with Logo, so don't expect the files for one to run in the other. Note that these programs do not test the entire functionality your program should support.

Information about the Logo language and Logo commands can be found, in detail, in the user manual in /cs/lib/logo/usermanual.

“Wow! In just one week I’ve created Logo using the Making Logo Quick Package! I have another week and I want more!”

There is a lot of room for creativity and extras in this project, but as always, do not start implementing any of these until Simple Logo, as specified, is functional. You are welcome to design your program to support any of these features. Some ideas:

- Variable scoping and passing parameters to subroutines (i.e. turning subroutines into procedures)
- An enlarged command set (see the Logo usermanual for ideas)
- Multimedia: audio, animation, more complicated GUI
- More complex control structures.
- List processing functions and lists, stacks, queues, and random access arrays
- A snazzier turtle graphics package

Cashing In

We recommend you individually come up with a top-level design before getting together with your group to work out the complete design. You must meet with a TA on or before Thursday, March 5 to go over your design. Please try to go meet a TA together, rather than seperately. You must give the TA a copy of your detailed, fully annotated design (i.e. commented object diagram) which the TA will keep, and which will count as part of the grade for this assignment. You should also bring a copy of the design for you to make notes on. When you are finished with your program, hand it in electronically by the end of the day Thursday, March 12, 1998. When you hand in your finished program (one handin for each pair), you should also hand in a revised copy of your object diagram which reflects your final design. You should also create a README file which describes your overall design, each file in your project, any known bugs or bells and whistles, and anything else you want us to know. Note that both of the designs you hand in will count as a substantial part of your grade.

You are responsible for printing out source code and the README, labelling the separate files and ***putting both your names and account numbers on them*** before 5pm on the day after your handin. This should give you ample time to print out your code. Please put the printouts in a sensible order, with the README first. Please hand in only the README and the source code, and not any extraneous pages (cover pages, etc.) printout by the CIS printer.

Appendix A

Simple Logo Language Details

This document contains a description of the Simple Logo language, including supported commands, their usage and effects.

Language Structure

- A word beginning with a colon is a variable.
- A word beginning with a letter is a command.
- A word beginning with a number is a numerical value.
- Words are delimited by spaces, tabs, newlines, brackets, or an =.
- Names of variables and commands are case-insensitive in Simple Logo .
- All values are integers.
- Simple Logo, unlike Logo, will expect only one command per line. That is, a single command plus its arguments is to be terminated with the newline character. Thus,

```
fd 50
rt 90
```

  is syntactically valid while

```
fd 50 rt 90
```

  is not.
- There is variable assignment. A variable assignment can have one of three forms:

```
:var_name = COMMAND
:var_name = :other_var_name
:var_name = number
For example, the following are valid:
:foo = SUM 12 34
:bar = :foo
:fish = 23
```

- Commands will be in prefix form; that is, the command name will precede the arguments.
- All commands return a value. If no return value is defined, the command should return 0.
- The variable :TRUE should be defined to be 1, and :FALSE should be 0.
- The supported conditional is IF. If the command or variable has a value of 0, the body of the IF should be skipped, otherwise it should be executed.

```
IF <command or variable> [
      instruction
      instruction
      ...
]
```

- The supported loop is REPEAT. The instructions in the body should be executed the

number of times given by the value or variable.

```
REPEAT <variable or number> [
      instruction
      instruction
      ...
]
```

- The command name `TO` denotes the beginning of a subroutine definition. The next argument should be the name of the subroutine. Then the body of the subroutine is specified.

```
TO mysub [
      instruction
      instruction
      ...
]
```

Table 1: Math Operations

Name	Description
SUM num1 num2	outputs the sum of its inputs
DIFFERENCE num1 num2	outputs the difference of its inputs
MINUS num	outputs the negative of its input
PRODUCT num1 num2	outputs the product of its inputs
QUOTIENT num1 num2	outputs the quotient of its inputs
REMAINDER num1 num2	outputs the remainder on dividing num1 by num2. The result is an integer with the same sign as num2

Table 2: Drawing Operations and Turtle Commands

Command	Description
FORWARD dist FD dist	moves the turtle forward by the amount specified
BACK dist BK dist	moves the turtle backwards by the amount specified
LEFT degrees LT degrees	turns the turtle counterclockwise by the specified angle
RIGHT degrees RT degrees	turns the turtle clockwise by the specified angle
SETXY xcor ycor	moves the turtle to an absolute screen position.

Table 2: Drawing Operations and Turtle Commands

Command	Description
SETX xcor	moves the turtle horizontally to a new absolute horizontal coordinate
SETY ycor	moves the turtle vertically to a new absolute vertical coordinate.
HOME	moves the turtle to the center of the screen (0 0)
XCOR	outputs the turtle's X coordinate
YCOR	outputs the turtle's Y coordinate
HEADING	outputs the turtle's heading in degrees
TOWARDS xcor ycor	outputs a heading the turtle should be facing to point from its current position to the given position
SHOWTURTLE ST	makes the turtle visible
HIDETURTLE HT	makes the turtle invisible
CLEAN	clears the drawing area (the turtles statistics do not reset)
CLEARSCREEN CS	erases the drawing area and sends the turtle to the home position (Like CLEAN and HOME)
PENDOWN PD	sets the pen's position to DOWN
PENUP PU	sets the pen's position to UP
PENDOWNP PENDOWN?	outputs 1 (:TRUE) if the pen is down, 0 (:FALSE) if it's up.

Table 3: File Commands

Command	Description
SAVE filename	Saves the definitions of all procedures and variables in the named file.
LOAD filename	Loads the definitions from the named file.
BYE	Exits LOGO

Table 4: Control Structures and Procedures

Command	Description
REPEAT numOrVar [instructionlist]	runs instructionlist numOrVar times
IF varOrCommand [instructionlist]	if varOrCommand is not 0, run instructionlist
TO subr_name [instructionlist]	defines a new subroutine (command) named subr_name. When invoked, the subroutine will execute the body of instructions included in the definition.

Table 5: Boolean Operations

Command	Description
LESS? num1 num2	outputs 1(:TRUE) if its first input is strictly less than its second, or 0 otherwise (:FALSE)
GREATER? num1 num2	outputs 1 if its first input is strictly greater than its second, or 0 otherwise
EQUAL? thing1 thing2	outputs 1 if the two inputs are equal, 0 otherwise
NOTEQUAL? thing1 thing2	outputs 1 if the two inputs are not equal, 0 otherwise

If all of these tables seem confusing and you'd just rather see the grammar for slogo, here's all you need. If this doesn't make sense to you, review your notes from 31 or 51. If you're still stuck, come see a TA. It's worth knowing.

```
! grammar for simple logo

! a program is a list of instructions
<prog>        ::= <ilist>

! an ilist is a list of instructions and function declarations
<ilist>       ::=
              ::= <instr> <ilist>
              ::= <fdecl> <ilist>

! an instruction might be an assignment; function call; a conditional; or a loop
<instr>       ::= <asgn>
              ::= <fcall>
              ::= <if>
              ::= <repeat>
```

```
! an assignment always looks the same: a variable gets an expression
<asgn>          ::= <var> = <expr>

! variables always look the same; colon followed by an identifier
<var>           ::= :identifier

! expressions can look like anything; they bottom out with vars and numerical constants
<expr>          ::= number
                ::= <var>
                ::= SUM <expr> <expr>
                ::= DIFFERENCE <expr> <expr>
                ::= MINUS <expr>
                ::= PRODUCT <expr> <expr>
                ::= QUOTIENT <expr> <expr>
                ::= REMAINDER <expr> <expr>

! fcalls are either subroutine calls or built-ins; we don't bother putting
! in all the built-ins here for brevity
<fcall>         ::= identifier
                ::= ...                ! Imagine all the built-ins from table 2 and 3 here

! blocks of code delimited by [ ] cannot contain function declarations, unlike
! the top-level instruction list, so we introduce strict_ilist to be a list
! only of instructions
<strict_ilist>  ::=
                ::= <instr> <strict_ilist>

! loops
<repeat>        ::= REPEAT <expr> [ <strict_ilist> ]

! conditionals
<if>            ::= IF <cond> [ <strict_ilist> ]

! function declarations
<fdecl>         ::= TO identifier [ <strict_ilist> ]

! conditions; for tests in IF statements
<cond>          ::= <condop> <expr1> <expr2>

<condop>        ::= LESS?
                ::= GREATER?
                ::= EQUAL?
                ::= NOTEQUAL?
```