Simple Logo ("Slogo") Design

CS32 Assignment 1 (8% of Final Grade)

Important Dates:

Assignment Out:	Thurs. Jan 23, 2003.
Partner Chosen:	Tues. Jan 28, 2003.
Rational Rose Help Session:	Weds. Jan 29, 2003 in CIT165.
Logo Design Due:	Mon. Feb 3, 2003 at NOON! (in handin bin)
Logo Design Workshop:	Tues. Feb 4, 2003 in class.

Welcome!

Congratulations on "choosing" CS32! Now it's time to hit the ground running with an assignment which should reinforce two central principles for this year: teamwork and design. You'll work with one other person, and hand in a single document (with both your names clearly marked). The assignment will be done using the Unified Modeling Language (UML) as taught in class and reinforced during a help-session on Rational Rose this Wednesday, January 29th in CIT165.

You need not inform us of your partner choice before handing in, but if you have not found a partner before the next class period, please contact the TAs immediately. You may not work on this assignment alone. A CS32 conventions document has been handed out in class and is available on the course webpage which details the specific UML conventions you will be held to. Also, it is recommended that you attend the **Rational Rose Help Session, on Wed., Jan. 29th from 7-9pm in CIT 165**. At the help session your TAs will cover the ins and outs of Rational Rose and be avaliable to answer any questions about the Logo assignment.

Goals

- Object Oriented Design
- Introduction to the Unified Modeling Language
- Teamwork!

Your task

For this assignment, you and your partner will design (**not** implement) a simplified version of a programming language called Logo. Your design will not be dependent on a particular language, although a potential implementation language would clearly be object oriented. The handin will consist of a few pages of UML class descriptions, "use cases" and a sequence diagram.

The completed assignments will be the source for a Logo Critique workshop which will take place in class on Feb. 4th. Please be aware that your work may be displayed and analyzed, for both strengths and weaknesses, in front of the entire class. This process will be kept as confidential and anonymous as possible, and to assist us in this you must make your handins neat and clear. You may use Rational Rose or simply pen and paper, but please avoid lined or ruled paper which may not transfer to transparency easily.

Background

Remember that special day in third grade when three Apple II computers appeared in the back of the classroom, 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."

Logo is a computer programming language originally created at the MIT Artificial Intelligence Lab in the 1970s for use by learners. This 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 by controlling the turtle and its pen. The turtle, which has since migrated to the 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 a much simplified version of Logo. Simple Logo should retain the features most commonly used by beginning users.

For more information, check out: http://el.www.media.mit.edu/groups/logo-foundation/ or the logo implementations in /course/cs032/demos/logo. (Please note: Our implementations, both Linux and Solaris versions, may not be perfect nor should be used as a reference!)

Functionality

Simple Logo provides functionality in the following areas:

- User interface: 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.
- Basic turtle graphics: Simple Logo 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.
- 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, via the graphical user interface (GUI.)

Refer to the table of commands in Appendix A and the grammar in Appendix B for a complete explanation of the built in commands.

How to think about the project

Simple Logo can be broken down into two main components: the Parser, responsible for command parsing, interpretation, and environment maintenance, and the Turtle Graphics Library (TGL), in charge of all the graphics and user interaction, as well as the turtle itself. Parsing involves processing commands token by token and verifying syntax, whereas interpretation is the actual execution of code as understood by the parser. It is this execution which will require graphical manipulation and therefore interaction with the TGL.

A good Logo design will appropriately split functionality into compact, easily understand pieces, which could be independently changed or upgraded thanks to well-defined interfaces. Remember, there is no single perfect design, but certain designs (such as one with a single huge class) are clearly inferior to others. Meet with your partner early, and plan the project in stages.

Parsing

The parser will receive a block of input (not necessarily a single line), and be expected to store and evalute it. This should seem very familiar from CS31's Compiler and heavily rely on concepts from object oriented programming which you know well by now. Polymorphism, for example, is likely to be a close ally in helping you transform the Simple Logo grammar (Appendix B) into a set of inheritance relationships.

Graphics

For this project you do not have to learn and understand an actual graphics package, nor base your GUI implementation on any existing graphics libraries. On the next page we have provided a limited API (Application Programmers Interface) specification establishing which classes and what methods you might find useful for writing the graphical sections of the Simple Logo design.

Teamwork

We recommend you work individually on a top-level design before meeting as a group to design the complete solution. The idea is not to discourage working as a team, but rather to promote individual thought and creativity as well. Undoubtedly most of your time will be spent together, and we expect a roughly equal contribution from each student both in thought and authorship.

Design Workshop

The purpose of the Logo critique in class on Tues. Feb 4th is to emphasize perhaps the most important process in design: redesign. Through examples picked from your handins, you will be exposed to various approaches to the problem, limited only by the creativity of your work. Again, this process is to be as anonymous as possible, since our intent is neither to embarass nor reward specific teams, but rather to provide constructive feedback in a timely manner. However, we can only hope this process will also encourage you to produce work you are proud of.

Requirements

All groups must hand in the following:

Class relationships

One UML diagram of the major classes, in which the interaction between the graphical and parsing portions of the project are clearly described. Make this fit on a single page, but please do not omit any methods which make calls **between** major components.

Two or three extra pages should show more detail regarding inheritance, containment, and use **within** each of the portions. Try to avoid redundancy, if many classes are similar, one or two examples plus a description of classes is enough. For example, if you have many similar language object classes, we would prefer a tree or chart showing inheritance relationships between them and descriptions of common methods than complete UML diagrams for each. The same is true for the GUI, we don't desire to know about every object within each pull-down menu, a few concise examples will suffice.

Sequence Diagram

A single page should contain a sequence diagram for the following use case:

A user types "FD 10" in the GUI input window and clicks submit.

Design Questions

After completing your design, please (briefly!) answer the following questions:

How does your design facilitate the separation of the project into separate portions which could be worked on individually?

What information is necessary for each of the major components to know about the other(s)?

How would one go about testing the pieces independently?

How does your parser represent the commands and how does their evaluation occur?

If you had to support loading and saving files (consisting of a list of previously typed commands), how would you change (or not) your current design?

What would you need to do to your design to support an undo command?

README / Comments

In a README section, you should write any clarifications regarding your design you think are necessary. This is also a good place to point out any interesting or unique elements of your design. If space permits, these comments may be included on your UML diagrams, but please don't sacrifice clarity.

Logo GUI API

The following are a list of classes you may find helpful in designing your Logo GUI. They all begin with 'Q' as they are stripped down versions of their QT equivalents. You may read further documentation on QT if you'd like (http://doc.trolltech.no/ [any version]) and use classes mentioned therein. However, these simplified classes should be enough to design a Simple Logo. Feel free to ignore astericks and amperstands for now, and assume that QStrings behave much like any other strings you are already familiar with.

QWidget (QWidget *parent)

The QWidget class is the base class of all user interface objects. The widget is the atom of the user interface: It receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. A widget that isn't embedded in a parent widget is called a top-level widget. The opposite of top-level widgets are child widgets. Most widgets in Qt are used only as child widgets. Useful public members and methods include:

```
int width ()
int height ()
void setMaximumSize (int maxw, int maxh)
void setFixedSize (int w, int h)
void resize (int w, int h)
void show ()
void hide ()
```

QApplication ()

The QApplication class manages the GUI application's control flow and main settings. It contains the main event loop, where all events from the window system and other sources are processed and dispatched. It also handles application initialization and finalization, and provides session management. Finally, it handles most system-wide and application-wide settings. There is precisely one QApplication object, no matter whether the application has 0, 1, 2 or more windows at the moment. This object initializes the application to the user's desktop settings. It performs event handling, meaning that it receives events from the underlying window system and sends them to the destination widgets. Public methods:

```
void setMainWidget (QWidget *)
```

The main widget is the one which occupies the window space and, if closed, causes the application to exit. (A practical thing to do is set the main widget to a layout object which contains your other widgets.)

```
int exec ()
```

Calling exec() is necessary to start the event loop which handles user interaction.

```
void quit ()
```

QGridLayout (QWidget *parent, int nRows, int nCols, int margin)

The QGridLayout class lays out widgets in a grid. QGridLayout takes the space made available to it (by its parent layout or by the mainWidget()), divides it up into rows and columns, and puts each widget it manages into the correct cell. Each column/row has a minimum width and a stretch factor. The minimum width is the greatest of that set using addColSpacing()/addRowSpacing() and the minimum width of each widget in that column/row. The stretch factor is set using setColStretch()/setRowStretch() and determines how much of the available space the column/row will get over its necessary minimum. Each managed widget or layout is put into a cell of its own using addWidget().

```
void addWidget (QWidget * w, int row, int col)
void setRowStretch (int row, int stretch)
void addRowSpacing (int row, int spacing)
```

QCanvas(QWidget *parent, int h, int v, int tilewidth, int tileheight) [and family]

The QCanvas class manages a 2D graphic area and all the canvas items the area contains. The QCanvasItem class provides a superclass for graphic objects on a QCanvas. A variety of QCanvasItem subclasses provide immediately usable behaviour. However, QCanvasItem is not intended for direct subclassing. It is much easier to subclass one of its subclasses, e.g. QCanvasPolygonalItem (the commonest base class), QCanvasLine, or QCanvasText. A QCanvasItem subclass is given the QCanvas object in its constructor, and inherits functionality like move(int x, int y), setX(int x), sexY(int y) from QCanvasItem. For example, QCanvasLine:

```
QCanvasLine (QCanvas * canvas)
void setPoints (int xa, int ya, int xb, int yb)
```

(.. would be useful for placing lines as a certain triangular QCanvasPolygonalItem moves)

QPopupMenu (QWidget *parent)

A popup menu widget is a selection menu. It can be either a pull-down menu in a menu bar or a standalone context (popup) menu. Pull-down menus are shown by the menu bar when the user clicks on the respective item. A popup menu consists of a list of menu items, to which you add items with insertItem(). Separators are inserted with insertSeparator(). For submenus, you pass a QPopupMenu in your call to insertItem().

QMenuBar (QWidget *parent)

The QMenuBar class provides a horizontal menu bar. A menu bar consists of a list of pull-down menu items. You add menu items with insertItem(). For example, asuming that menubar is a pointer to a QMenuBar and filemenu is a pointer to a QPopupMenu, the following statement inserts the menu into the menu bar:

```
menubar->insertItem("&File", filemenu);
```

QPushButton (const QString & text, QWidget *parent)

The push button, or command button, is perhaps the most commonly used widget in any graphical user interface. Clicking a button tells the computer to perform some action or answers a question. QPushButtons have a method clicked() which should be filled in by a button class subclassing QPushButton and code the actions to be performed upon clicking.

QMultiLineEdit (QWidget *parent)

The QMultiLineEdit widget is a simple editor for inputting text.

```
int numLines()
QString text ()
```

Returns a copy of the whole text. If the multi-line edit contains no text, a null string is returned.

```
QString textLine (int line)
```

Returns the text at line number line (possibly the empty string), or a null string if line is invalid.

QMessageBox (QWidget *parent)

Creating a QMessageBox class pops up a dialog with a message, an icon, and some buttons of choice (Yes, No, Ok, or Cancel). This enum and method:

```
enum { NoButton = 0, Ok = 1, Cancel = 2, Yes = 3, No = 4 }
int information (QWidget * parent, const QString & caption, const QString
      & text, int button0)
```

should be all that you need to pop up message windows, for example:

```
messBox.information(_widget, "Incorrect command", "Type a real
command!!", "OK");
```

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 / subroutine name.
- A word beginning with a number is a numerical value.
- Words are delimited by spaces, tabs, newlines, or brackets.
- Names of variables and commands are case-insensitive in Simple Logo.
- All variables are global.
- All literal values are integers greater to or equal to 0. (Note: Commands such as Difference can produce a negative value.)
- There is variable assignment. A variable assignment can have one of three forms:

```
MAKE :var_name COMMAND
MAKE :var_name :other_var_name
MAKE :var_name number

For example, the following are valid:
MAKE :foo SUM 12 34
MAKE :heading HEADING
MAKE :heading FD XCOR
MAKE :bar :foo
MAKE :fish 23
```

- Commands will be in prefix form; that is, the command name will precede the arguments.
- **All commands must return a value**. *If no return value is defined, the command should return 0.*
- There are no booleans. All values greater than zero signify TRUE.
- The supported conditional is IF. If the command or variable or literal has a value of 0, the body of the IF should be skipped, otherwise it should be executed.

```
IF <value> [
     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 <value> [
     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. NOTE: You can assume that no one will ever try to redefine a primitive such as FD or IF or MAKE. However, you must properly implement redefinition of subroutines.

```
TO <subroutine_name> [
     instruction
     instruction
     ...
]
```

Table 1: Math Operations

Name	Description
SUM num1 num2	returns the sum of its inputs
DIFFERENCE num1 num2	returns the difference of its inputs
PRODUCT num1 num2	returns the product of its inputs
QUOTIENT num1 num2	returns the quotient of its inputs
REMAINDER num1 num2	returns 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.
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)
SETPENCOLOR r g b	changes the pen color to the specified rgb color (0 to 255)

Table 2: Drawing Operations and Turtle Commands

Command	Description
XCOR	returns the turtle's X coordinate
YCOR	returns the turtle's Y coordinate
HEADING	returns the turtle's heading in degrees
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

Table 3: Control Structures and Procedures

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

Table 4: Boolean Operations

Command	Description
LESS? num1 num2	returns 1(:TRUE) if its first input is strictly less than its second, or 0 otherwise (:FALSE)
GREATER? num1 num2	returns 1 if its first input is strictly greater than its second, or 0 otherwise
EQUAL? thing1 thing2	returns 1 if the two inputs are equal, 0 otherwise

Table 4: Boolean Operations

Command	Description
NOTEQUAL? thing1 thing2	returns 1 if the two inputs are not equal, 0 otherwise
PENDOWN?	returns 1 (:TRUE) if the pen is down, 0 (:FALSE) if it’s up.

Appendix B

Slogo Grammar

This is the grammar which describes the Simple Logo language. It describes the same rules as are contained in Appendix A, only more formally. Those students who have taken CS 31 should be reminded of the Compiler assignment and the Blaise grammar.

Table 5: Slogo Terminals

num	An integer >= 0
string	A series of ASCII characters, excluding ':'

Note: A string is defined as starting with a letter, and containing only letters, numbers, or underscores.

Table 6: Slogo Grammer Rules

<Input>	::==	<List_of_Instructions> <List_of_Subroutines> <List_of_Instructions>
<List_of_Subroutines>	::==	
<List_of_Subroutines>	::==	<Subroutine_Declration> <List_of_Subroutines>
<Subroutine_Declaration>	::==	TO <Subroutine_Name> [<List_of_Instructions>]
<List_of_Instructions>	::==	
<List_of_Instructions>	::==	<Instruction><List_of_Instructions>
<Instruction>	::==	<No_Arg_Command>
<Instruction>	::==	<One_Arg_Command><Value>
<Instruction>	::==	<Two_Arg_Command><Value><Value>
<Instruction>	::==	<Three_Arg_Command><Value><Value> <Value>
<Instruction>	::==	IF <Value> **[** <List_of_Instructions>**]**
<Instruction>	::==	REPEAT <Value> **[** <List_of_Instruction> **]**
<Instruction>	::==	MAKE <Variable_Name><Value>

<Variable_Name>	::==	**:** string
<Subroutine_Name>	::==	string

<Value>	::==	<Instruction>
<Value>	::==	num
<Value>	::==	<Variable_Name>

<No_Arg_Command>	::==	HOME
<No_Arg_Command>	::==	HEADING
<No_Arg_Command>	::==	CS
<No_Arg_Command>	::==	CLEARSCREEN
<No_Arg_Command>	::==	PU
<No_Arg_Command>	::==	PENUP
<No_Arg_Command>	::==	PD
<No_Arg_Command>	::==	PENDOWN
<No_Arg_Command>	::==	PENDOWN?
<No_Arg_Command>	::==	BYE
<No_Arg_Command>	::==	XCOR
<No_Arg_Command>	::==	YCOR
<No_Arg_Command>	::==	HIDETURTLE
<No_Arg_Command>	::==	HT
<No_Arg_Command>	::==	SHOWTURLE
<No_Arg_Command>	::==	ST
<No_Arg_Command>	::==	CLEAN
<No_Arg_Command>	::==	<Subroutine_Name>

<One_Arg_Command>	::==	FORWARD
<One_Arg_Command>	::==	FD
<One_Arg_Command>	::==	BACK
<One_Arg_Command>	::==	BK
<One_Arg_Command>	::==	LEFT
<One_Arg_Command>	::==	LT
<One_Arg_Command>	::==	RIGHT
<One_Arg_Command>	::==	RT
<One_Arg_Command>	::==	SETX
<One_Arg_Command>	::==	SETY
<One_Arg_Command>	::==	SAVE
<One_Arg_Command>	::==	LOAD
<One_Arg_Command>	::==	MINUS

<Two_Arg_Command>	::==	NOTEQUAL?
<Two_Arg_Command>	::==	EQUAL?
<Two_Arg_Command>	::==	GREATER?
<Two_Arg_Command>	::==	LESS?
<Two_Arg_Command>	::==	SUM
<Two_Arg_Command>	::==	DIFFERENCE
<Two_Arg_Command>	::==	PRODUCT
<Two_Arg_Command>	::==	QUOTIENT
<Two_Arg_Command>	::==	REMAINDER
<Two_Arg_Command>	::==	SETXY

<Three_Arg_Command>	::==	SETPENCOLOR