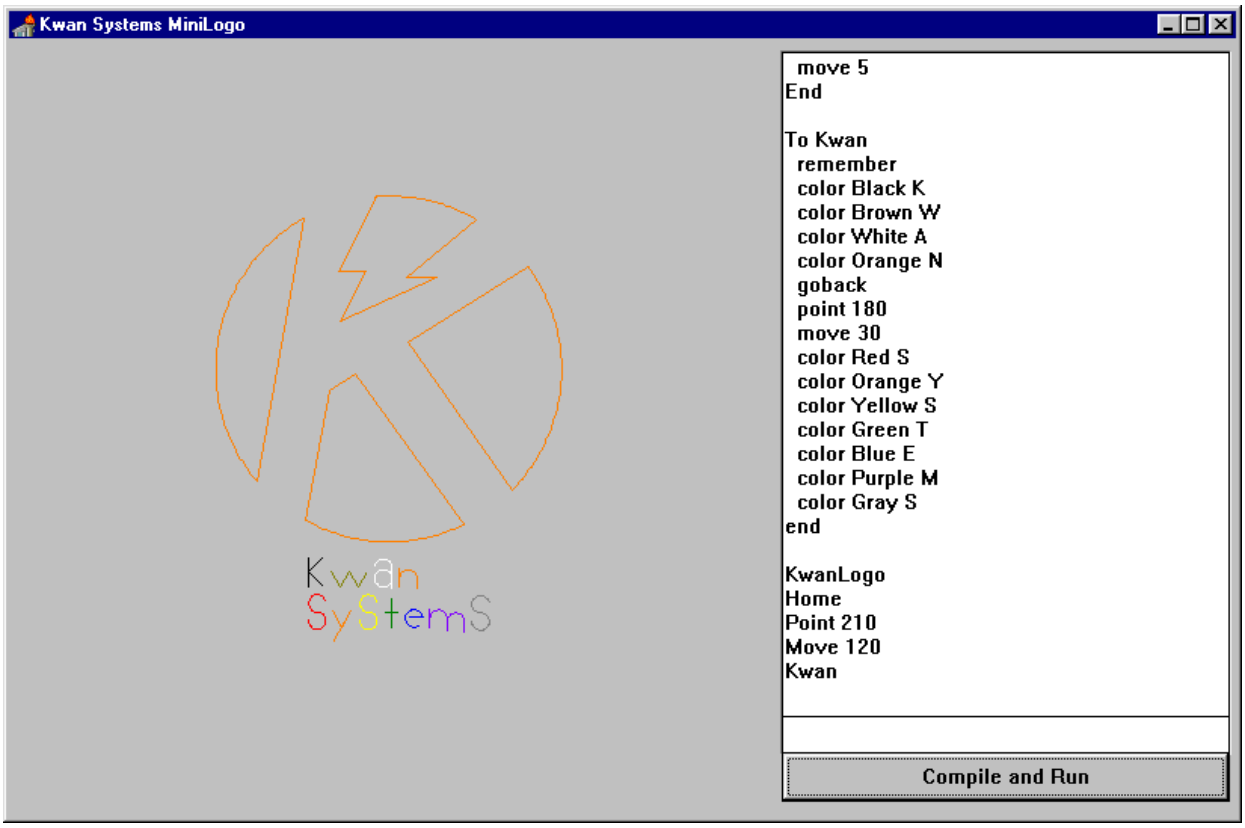
Kwan Systems MiniLOGO, Part 3: Code generator and interpreter
Lab assignment for CST 320
November 3, 1998
Chris Jeppesen

ABSTRACT: Source code, test files, and output of the Code Generator and interpreter for the MiniLOGO Language. Lexical analysis is performed by the same modules written for Lab 1, slightly modified as the requirements of the parser became better defined. Syntactic analysis is performed by the same modules written for Lab 2, with code generation hooks added to the syntactic recognizers.

PROGRAM USAGE:



Screen Capture of Kwan Systems MiniLogo

To Start Logo, run the program KwanLogo.EXE. You will see a large gray space on the right, and on the left two white rectangles. As per specifications, if you wish to enter one line of logo code, you may do so in the lower white rectangle. Pressing return will add the line to the current porgram (the large white rectangle) and then compile and execute the current program, with the results appearing on the left. I prefer to edit directly in the current program box. Any changes to the current program will be reflected in the output. The program generates the text files Screen.PAR (parse trace) Screen.BPC (Binary pcode) and Screen.TPC (Text pcode)

HOW IT WORKS: For this lab, I have written a pcode interpreter and code generator for MiniLOGO. The Structure of the interpreter virtual machine was determined first, as this decides what kind of pcode to make.

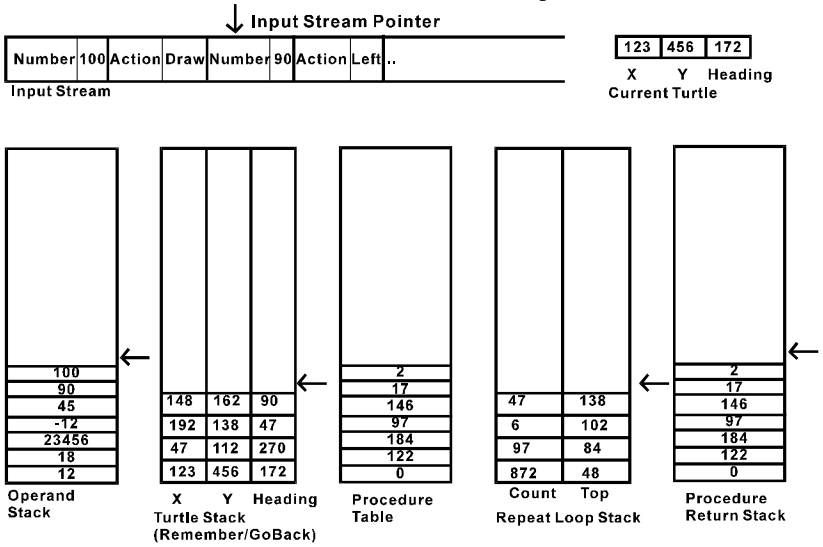


Diagram of Virtual Machine

The Virtual Machine has an input stream, four stacks, and a procedure table. The input stream is a binary file, consisting of a bunch of 3-byte opcodes. The first of each 3 bytes is the opcode type, 0 for number or 1 for action. If it is a number, the next two bytes are a signed integer to be put onto the operand stack. If it is an action, the next two bytes are the action code.

Action Code	Corresponding action
0 (Draw)	Pops the operand stack and moves the turtle that many steps forward, drawing a line in the current color. Negative numbers move the turtle in reverse.
1 (Move)	Pops the operand stack and moves the turtle that many steps forward, not drawing any line. Negative numbers move the turtle in reverse.
2 (Turn)	Pops the operand stack and turns the turtle that many degrees to the right. Negative numbers turn the turtle to the left.
3 (Point)	Pops the operand stack and points the turtle at that heading. 0 is North, 90 is East, 180 is South, 270 is West. Any positive or negative number may be used.
4 (Home)	Moves the turtle to the center of the drawing area and sets his heading to 0 (North).
5 (Remember)	Pushes the current turtle onto the turtle stack.
6 (GoBack)	Pops the turtle stack and makes it the current turtle.
7 (ProcDef)	Pops the operand stack and loads the current input stream pointer into the procedure table at the address pointed to by the operand. Then moves the input stream pointer past the next Return opcode.
8 (Return)	Pops the return stack and loads that into the input stream pointer.
9 (Loop)	Decrements the count on the top of the repeat stack. If Count is not zero, loads the input stream pointer with the Top at the top of the repeat stack. If count is zero, pops the repeat stack
10 (ProcCall)	Pushes the input stream pointer onto the return stack, pops the operand stack, and loads the input stream pointer with the operand.
11 (Rep)	Pops the operand stack, then pushes onto the repeat stack the operand as Count and the input stream pointer as Top.
12 (SetColor)	Pops the operand stack and sets the current color to that entry in the color table.
13 (LastOp)	Last opcode in the pcode stream. Causes the VM to stop.

The code generator is called by the parser. As each sentinential form is found, the parser calls the necessary function in the code generator corresponding to this production. The parser passes the 0, 1, or 2 tokens necessary to generate the opcode. The code generator then converts numbers to strings as necessary and writes the opcode to both the binary pcode file (.BPC) and a text representation of the pcode. (.TPC)

PROGRAMMING ENVIRONMENT: Kwan Systems MiniLogo was written in Borland Delphi 1.0. This language was chosen because it is a fully compiled language, and is compatible with Turbo Pascal 7.0, which is what Labs 1 and 2 were written in.

```
{KwanLogo.Pas ***********************************************************
 Main Program Body
*************************************************************************}
program Kwanlogo;
uses
  Forms,  Logoform in 'LOGOFORM.PAS' {Form1},
  IO in 'IO.PAS',  Lex in 'LEX.PAS',  LogoVM in 'LOGOVM.PAS',
  MiniLogo in 'MINILOGO.PAS',  Parse1 in 'PARSE1.PAS',
  CodeGen in 'CODEGEN.PAS';
{$R *.RES}
begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

{LogoForm.Pas ***********************************************************
 This unit is the form which holds the code window and drawing space
*************************************************************************}
unit LogoForm;

interface

uses
  Classes, Graphics, Controls, Forms, Io,
  MiniLogo, CodeGen, Parse1, StdCtrls, ExtCtrls, LogoVM;

type
  TForm1 = class(TForm)
    Button1: TButton;
    PaintBox1: TPaintBox;
    TXTOneLine: TEdit;
    TXTSource: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  If TXTOneLine.Text<>'' then begin
    TXTSource.Lines.Add(TXTOneLine.Text);
    TXTOneLine.Text:='';
  end;
  OpenSourceTXT(TXTSource);
  OpenLog('screen.par');
  NewPCode('screen');
  PrintLog('Kwan Systems MiniLOGO, Part 3: Code Generator');
  PrintLog('Lab assignment for CST 320');
  PrintLog('October 26, 1998');
  PrintLog('Chris Jeppesen');
  While ParseIt do ;
  CloseSource;
  CloseLog;
  ClosePCode;
  If ValidParse then begin
    PaintBox1.Refresh;
    InitVM(PaintBox1.Canvas);
    Interpret('screen.BPC');
  end;
end;

end.
```

```
{Codegen.Pas ************************************************
This is the code generator module. It also holds the constants
which define the P-Code actions.
*************************************************************}
Unit CodeGen;

interface

uses MiniLogo;

{Code generation procedures}

{Generates a pcode instruction for a MiniLOGO statement with no operands
  Parameters:
    T1: Token containing name of instruction to be coded}
Procedure GenerateNoOp(T1:TToken);

{Generates an instruction for a statement with one numerical operand
  Parameters:
    T1: Token containing name of instruction to be coded
    T2: Token containing string form of numerical operand}
Procedure GenerateOneOp(T1,T2:TToken);

{Generates an instruction for a setcolor statement
  Parameters:
    T1: Token containing name of color}
Procedure GenerateColor(T1:TToken);

{Generates an instruction for a procedure call
  Parameters:
    T1: Token containing name of procedure to call
  Returns: True if procedure is already defined, false if not}
 Function GenerateProcCall(T1:TToken):Boolean;

{Generates procedure head for procedure, and enters procedure
 name into symbol table
  Parameters:
    T1: Token containing name of procedure}
Procedure GenerateProcHead(T1:TToken);

{Generates instruction for procedure return}
Procedure GenerateProcReturn;

{Generates repeat loop header
  Parameters:
    T1: Token containing string form of repeat count}
Procedure GenerateRepeatHead(T1:TToken);

{Generates repeat loop footer}
Procedure GenerateRepeatLoop;

{Creates a binary pcode file and a text file showing the pcode in
"assembler" format
  Parameters:
    Fn: Filename minus extention for the pcode files. The extentions
        .BPC (binary pcode) and .TPC (Text pcode) are appended}
Procedure NewPCode(Fn:String);
{Closes binary and text pcode files}
procedure ClosePCode;

type TAction=(Draw,Move,Turn,Point,
              Home,Remember,GoBack,
              ProcDef,Return,Loop,
              ProcCall,Rep,
              SetColor,LastOp);

const ActionName:Array[Draw..LastOp] of String[8]=
      ('Draw','Move','Turn','Point',
       'Home','Remember','GoBack',
       'ProcDef','Return','Loop',
       'ProcCall','Rep',
       'SetColor','LastOp');
```

```
type TOpCode=record
       Klass:(Number,Action);
       case integer of
         Number:(Value:Integer);
         Action:(Code:TAction);
       end;


implementation

var OpCode:TOpCode;
    {Symbol table is array of strings.}
    {Array index is symbol number}
    {There can be up to 50 procedures in a program}
    SymbolTable:Array[0..49] of String[32];
const NextSymbol:Integer=0;

var PCodeText:Text;
    PCodeBinary:File of TOpCode;

{Writes OpCode to the binary and text pcode files}
Procedure PutOp;
begin
  If OpCode.Klass=Action then OpCode.Value:=Lo(OpCode.Value);
  Write(PCodeText,FilePos(PCodeBinary):4,': ');
  If OpCode.Klass=Number then begin
    Writeln(PCodeText,OpCode.Value);
  end else begin
    Writeln(PCodeText,ActionName[OpCode.Code]);
  end;
  Write(PCodeBinary,OpCode);
end;

Procedure NewPCode;
begin
  Assign(PCodeText,Fn+'.TPC');Rewrite(PCodeText);
  Assign(PCodeBinary,Fn+'.BPC');Rewrite(PCodeBinary);
  {Reset Code Generator pointers}
  NextSymbol:=0;
end;

{Closes binary and text pcode files}
Procedure ClosePCode;
begin
  OpCode.Klass:=Action;
  OpCode.Code:=LastOp;
  PutOp;
  Close(PCodeText);
  Close(PCodeBinary);
end;

Procedure GenerateNoOp(T1:TToken);
begin
  OpCode.Klass:=Action;
  If T1.Lexeme='HOME' then begin
    OpCode.Code:=Home;
  end else if T1.Lexeme='REMEMBER' then begin
    OpCode.Code:=Remember;
  end else if T1.Lexeme='GOBACK' then begin
    OpCode.Code:=GoBack;
  end;
  PutOp;
end;

Procedure GenerateOneOp(T1,T2:TToken);
  var Code:Integer;
begin
  OpCode.Klass:=Number;
  Val(T2.Lexeme,OpCode.Value,Code);
  {Only turn to the right in the PCode}
  {Reverse value if action is Left}
```

```
  If T1.Lexeme='LEFT' then OpCode.Value:=-Opcode.Value;
  PutOp;
  OpCode.Klass:=Action;
  If T1.Lexeme='DRAW' then begin
    OpCode.Code:=Draw;
  end else if T1.Lexeme='MOVE' then begin
    OpCode.Code:=Move;
  end else if T1.Lexeme='LEFT' then begin
    OpCode.Code:=Turn;
  end else if T1.Lexeme='RIGHT' then begin
    OpCode.Code:=Turn;
  end else if T1.Lexeme='POINT' then begin
    OpCode.Code:=Point;
  end;
  PutOp;
end;

Procedure GenerateColor(T1:TToken);
begin
  OpCode.Klass:=Number;
  {Use resistor code for color <--> number}
  If T1.Lexeme='BLACK' then begin
    OpCode.Value:=0;
  end else if T1.Lexeme='BROWN' then begin
    OpCode.Value:=1;
  end else if T1.Lexeme='RED' then begin
    OpCode.Value:=2;
  end else if T1.Lexeme='ORANGE' then begin
    OpCode.Value:=3;
  end else if T1.Lexeme='YELLOW' then begin
    OpCode.Value:=4;
  end else if T1.Lexeme='GREEN' then begin
    OpCode.Value:=5;
  end else if T1.Lexeme='BLUE' then begin
    OpCode.Value:=6;
  end else if T1.Lexeme='PURPLE' then begin
    OpCode.Value:=7;
  end else if T1.Lexeme='GRAY' then begin
    OpCode.Value:=8;
  end else if T1.Lexeme='WHITE' then begin
    OpCode.Value:=9;
  end;
  PutOp;
  OpCode.Klass:=Action;
  OpCode.Code:=SetColor;
  PutOp
end;

Procedure GenerateProcHead(T1:TToken);
begin
  SymbolTable[NextSymbol]:=T1.Lexeme;
  OpCode.Klass:=Number;
  OpCode.Value:=NextSymbol;
  PutOp;
  OpCode.Klass:=Action;
  OpCode.Code:=ProcDef;
  PutOp;
  NextSymbol:=NextSymbol+1;
end;

function GenerateProcCall(T1:TToken):Boolean;
  {Returns false if T1.Lexeme isn't in symbol table}
  var I:Integer;
begin
  GenerateProcCall:=False;
  OpCode.Klass:=Number;
  For I:=0 to NextSymbol-1 do begin
    If SymbolTable[I]=T1.Lexeme then begin
      GenerateProcCall:=True;
      OpCode.Value:=I;
      PutOp;
      OpCode.Klass:=Action;
```

```
      OpCode.Code:=ProcCall;
      PutOp;
      Exit;
    end;
  end;
end;

Procedure GenerateProcReturn;
begin
  OpCode.Klass:=Action;
  OpCode.Code:=Return;
  PutOp;
end;

Procedure GenerateRepeatHead(T1:TToken);
  var Code:Integer;
begin
  OpCode.Klass:=Number;
  Val(T1.Lexeme,OpCode.Value,Code);
  PutOp;
  OpCode.Klass:=Action;
  OpCode.Code:=Rep;
  PutOp;
end;

Procedure GenerateRepeatLoop;
begin
  OpCode.Klass:=Action;
  OpCode.Code:=Loop;
  PutOp;
end;

end.
```

```
{LogoVM.Pas ****************************************************
PCode virtual machine
************************************************************}
Unit LogoVM;

interface

uses Graphics,CodeGen;

{Initializes VM pointers, and sets the drawing area on which
 to interpret the binary pcode
  Parameters:
    LCanvas: Drawing area for turtle}
procedure InitVM(LCanvas:TCanvas);
{Interprets named binary pcode file}
Procedure Interpret(Fn:String);

implementation

const clOrange=$000080FF;
const clPurple=$00FF0080;
const ColorList:Array[0..9] of TColor=(clBlack,clOlive,clRed,
                                       clOrange,clYellow,
                                       clGreen,clBlue,clPurple,
                                       clGray,clWhite);

type TTurtleState=Record
       X,Y,Heading:Real; {Turtle State. X is Left to right, Y is top to bottom}
                         {Heading is 0 degrees (North),  90 degrees (East),
                                     180 degrees (South), 270 degrees (West).}
     end;

{Structure for the loop stack}
type TLoopStruc=Record
       Count:Integer;
       Top:Integer;
     end;

{Holds the numbers for an RPN Interpreter}
var Turtle:TTurtleState;
    OpStack:Array[0..50] of Integer; {It will probably never get this deep.}
    OpTop:Integer;

{Holds the return addresses for procedure calls}
    RetStack:Array[0..50] of Integer;
    RetTop:Integer;

{Holds the entry point of the procedures}
    Proc:Array[0..50] of LongInt;

{Holds the loop points}
    LoopStack:Array[0..50] of TLoopStruc;
    LoopTop:Integer;

{Holds the remember/goback states}
    TurtleStack:Array[0..50] of TTurtleState;
    TurtleTop:Integer;

    Canvas:TCanvas; {Target to draw upon}

{Converts degrees to radians}
function Radians(Degrees:Real):Real;
begin
  Radians:=Degrees/180*Pi;
end;

{Moves the turtle <Steps> pixels without drawing a line}
procedure DoMove(Steps:Real);
begin
  Turtle.X:=Turtle.X+Steps*Sin(Radians(Turtle.Heading));
  Turtle.Y:=Turtle.Y-Steps*Cos(Radians(Turtle.Heading));
  Canvas.MoveTo(Round(Turtle.X),Round(Turtle.Y));
```

```
end;

{Moves the turtle <Steps> pixels and draws a line}
procedure DoDraw(Steps:Real);
begin
  Turtle.X:=Turtle.X+Steps*Sin(Radians(Turtle.Heading));
  Turtle.Y:=Turtle.Y-Steps*Cos(Radians(Turtle.Heading));
  Canvas.LineTo(Round(Turtle.X),Round(Turtle.Y));
end;

{Turns the turtle <Theta> degrees to the right}
Procedure DoTurn(Theta:Real);
begin
  Turtle.Heading:=Turtle.Heading+Theta;
  while Turtle.Heading<0 do Turtle.Heading:=Turtle.Heading+360;
  while Turtle.Heading>360 do Turtle.Heading:=Turtle.Heading-360;
end;

{Centers the turtle in the canvas}
Procedure DoHome;
begin
  Turtle.X:=250;
  Turtle.Y:=250;
  Turtle.Heading:=0;
  Canvas.MoveTo(Round(Turtle.X),Round(Turtle.Y));
end;

procedure InitVM;
  var I:Integer;
begin
  Canvas:=LCanvas;
  for I:=0 to 50 do begin
    OpStack[I]:=-1;
    Proc[I]:=-1;
    LoopStack[I].Count:=-1;
    LoopStack[I].Top:=-1;
    RetStack[I]:=-1;
  end;
end;

{Pushes <Op> onto the operand stack}
Procedure PushOp(Op:Integer);
begin
  OpStack[OpTop]:=Op;
  Inc(OpTop);
end;

{Pops the top value from the operand stack}
function PopOp:Integer;
begin
  Dec(OpTop);
  PopOp:=OpStack[OpTop];
  OpStack[OpTop]:=-1;
end;

{Pushes the current turtle onto the Remember/Goback turtle stack}
procedure PushTurtle;
begin
  TurtleStack[TurtleTop]:=Turtle;
  Inc(TurtleTop);
end;

{Pops the top turtle from the turtle stack and makes
 it the current turtle}
procedure PopTurtle;
begin
  Dec(TurtleTop);
  Turtle:=TurtleStack[TurtleTop];
  Canvas.MoveTo(Round(Turtle.X),Round(Turtle.Y));
end;

{Pushes the current instruction address onto the return stack}
```

```
procedure PushRet(P:LongInt);
begin
  RetStack[RetTop]:=P;
  Inc(RetTop);
end;

{Pops the top return pointer from the return stack}
function PopRet:LongInt;
begin
  Dec(RetTop);
  PopRet:=RetStack[RetTop];
end;

{Pushes <Count,P(ointer)> onto the stack for a Repeat loop}
procedure PushLoop(Count:Integer;P:LongInt);
begin
  LoopStack[LoopTop].Count:=Count;
  LoopStack[LoopTop].Top:=P;
  Inc(LoopTop);
end;

{Pops the top loop track structure from the repeat stack}
procedure PopLoop;
begin
  Dec(LoopTop);
end;

{Decrements Count on the top structure of the repeat stack
  Returns true to continue this loop, or false if the loop ends}
Function Iterate:Boolean;
begin
  Dec(LoopStack[LoopTop-1].Count);
  Iterate:=(LoopStack[LoopTop-1].Count>0);
end;

{Loads a procedure entry address into the procedure table}
Procedure LoadProc(N:Integer;Entry:LongInt);
begin
  Proc[N]:=Entry;
end;

procedure Interpret;
  var PCodeFile:File of TOpCode;
      OpCode:TOpCode;
      Done:Boolean;
begin
  DoHome;
  Assign(PCodeFile,Fn);Reset(PCodeFile);
  Done:=False;
  While not Done do begin
    Read(PCodeFile,OpCode);
    If OpCode.Klass=Number then begin
      PushOp(OpCode.Value);
    end else begin
      Case TAction(OpCode.Value) of
        Move:DoMove(PopOp);
        Draw:DoDraw(PopOp);
        Turn:DoTurn(PopOp);
        Point:Turtle.Heading:=PopOp;
        Home:DoHome;
        Remember:PushTurtle;
        GoBack:PopTurtle;
        ProcDef:begin
          LoadProc(PopOp,FilePos(PCodeFile));
          {Read past procedure}
          While Not ((Opcode.Klass=Action) and (OpCode.Code=Return)) do
             Read(PCodeFile,OpCode);
        end;
        ProcCall:begin
          PushRet(FilePos(PCodeFile));
          Seek(PCodeFile,Proc[PopOp]);
        end;
```

```
        Return:Seek(PCodeFile,PopRet);
        Rep:PushLoop(PopOp,FilePos(PCodeFile));
        Loop:If Iterate then Seek(PCodeFile,LoopStack[LoopTop-1].Top) Else PopLoop;
        SetColor:Canvas.Pen.Color:=ColorList[PopOp];
        LastOp:Done:=True;
      end;
    end;
  end;
  Close(PCodeFile);
end;

end.
```

```
{Io.Pas    ***********************************************************************
 Controls I/O of source and parse trace files
***********************************************************************************}
Unit IO;

interface

uses MiniLogo,StdCtrls,Classes;

{Opens a source code text file
  Parameters:
    Fn: full filename of source file to open}
procedure OpenSourceFile(Fn:String);

{Initializes reading of source code from a TMemo control
  Parameters:
    TXT: TMemo Control to read source from}
procedure OpenSourceTXT(TXT:TMemo);

{Creates a parse trace log file
  Parameters:
    Fn: full filename of log file to open}
procedure OpenLog(Fn:String);

{Gets one character from the source. Works with either
 file or TMemo input
  Returns: Next character to be read from the source
           Chr(13) if at end of a source line
           Chr(26) if at end of source code}
 function Get:Char;

{Puts a character back into the input buffer
  Parameters:
    C: Character to put back into the buffer}
procedure Push(C:Char);

{Closes source code file, or frees memory used in reading
 source from a TMemo}
Procedure CloseSource;

{Closes the parse log file}
Procedure CloseLog;

{Prints a string to the parse log file
  Parameters:
    S: String to print in file}
procedure PrintLog(S:String);

{Prints "Error: ", then a string to the parse log file
  Parameters:
    S: String to print in file}
procedure Error(S:String);

{Prints a text dump of a token to the log file
  Parameters:
    T: Token to print to the file}
procedure PrintToken(T:TToken);

implementation

var SourceFrom:(FromFile,FromTXT);
    LinePtr:Integer;
    Source,Log:Text;
    SourceTXT:TStringList;
    Buffer:String;

{Source Reading Functions}
procedure OpenSourceFile;
begin
  SourceFrom:=FromFile;
  Assign(Source,Fn);
  Reset(Source);
```

```
end;

procedure OpenSourceTXT;
begin
  SourceFrom:=FromTXT;
  SourceTXT:=TStringList.Create;
  SourceTXT.AddStrings(TXT.Lines);
  LinePtr:=0;
end;

procedure CloseSource;
begin
  If SourceFrom=FromFile then begin
    Close(Source);
  end else begin
    LinePtr:=0;
    SourceTXT.Free;
  end;
end;

function Get;
begin
  If Buffer='' then begin
    If SourceFrom=FromFile then begin
      {Read next source line out of file}
      while Buffer='' do begin
        If Eof(Source) then begin
          Get:=Chr(26);
          Exit;
        end else begin
          Readln(Source,Buffer);
          Get:=Chr(13);
        end;
      end;
    end else begin
      while Buffer='' do begin
        If LinePtr>=SourceTXT.Count then begin
          Get:=Chr(26);
          Exit;
        end else begin
          Buffer:=SourceTXT.Strings[LinePtr];
          Inc(LinePtr);
          Get:=Chr(13);
        end;
      end;
    end;
  end else begin
    Get:=UpCase(Buffer[1]);
    Buffer:=Copy(Buffer,2,length(Buffer)-1);
  end;
end;

procedure Push;
begin
  Buffer:=C+Buffer;
end;

{Output Producing Functions}

Procedure OpenLog(Fn:String);
begin
  Assign(Log,Fn);Rewrite(Log);
end;

Procedure CloseLog;
begin
  Close(Log);
end;

Procedure PrintLog(S:String);
begin
  Writeln(Log,S);
```

```
end;

Procedure Error(S:String);
begin
  Writeln(Log,'Error: ',S);
end;

procedure PrintToken(T:TToken);
begin
  If T.Klass=Null then Exit;
  PrintLog('Token Lexeme: '+T.Lexeme);
  PrintLog('       Class: '+TTokenKlassName[T.Klass]);
  PrintLog(' ');
end;

end.
```

```
{Lex.Pas    ***************************************************************
  Lexical Analyzer module
*****************************************************************************}

Unit Lex;

interface

uses Io,MiniLogo;

{DFA to get the next token from the source
  Return parameters:
    Token: New token from source
  Returns:
    True if there are more tokens to get, false if not}
function GetNextToken(var Token:TToken):Boolean;

{Checks if an ID token is actually a keyword
  Parameters:
    Token: Token to check if keyword. Token is modified if
           it is a keyword}
procedure CheckKeyword(var Token:TToken);

implementation

function GetNextToken;
  var State,L:Byte;
      OldState:Byte;
      C:Char;
  label LeaveLoop;
begin
  Token.Lexeme:='';
  Token.Klass:=Null;
  GetNextToken:=True;
  State:=1;
  While State<>0 do begin
    C:=Get;
    L:=Classify(C);
    If L<>7 then Token.Lexeme:=Token.Lexeme+C;
    OldState:=State;
    State:=LexDFATable[State,L];
  end;
LeaveLoop:
  State:=OldState;
  Token.Klass:=LexTokenKlass[State];
  If LexPushTable[State] then begin
    Push(C);
    If L<>7 then Token.Lexeme:=Copy(Token.Lexeme,1,Length(Token.Lexeme)-1);
  end;
  CheckKeyword(Token);
end;

procedure CheckKeyword;
  var I:Integer;
begin
  For I:=1 to NumKeywords do begin
    If Token.Lexeme=KWList[I].Lexeme then begin
      Token:=KWList[I];
      Exit;
    end;
  end;
end;

end.
```

```
{MiniLogo.Pas ***************************************************************
Holds language definition tables for the MiniLOGO language
*******************************************************************************}
Unit MiniLogo;

interface

Const MaxTokenLen=32;

Type TTokenKlass=(Null,Error,Done,Id,Num,LBracket,RBracket,
                  Color,ColorID,Rept,ToProc,OneOp,NoOp,EndProc);

Const TTokenKlassName:Array[Null..EndProc] of string[28]=
                 ('No  Token','Error','Done','Identifier','Number','Left  Bracket','Right
Bracket',
                  'Color Keyword','Color Identifier',
                  'Repeat Keyword','Procedure Definition Keyword',
                  'One Operand Keyword','No Operand Keyword','End Procedure');

type TToken=record
       Lexeme:String[MaxTokenLen];
       Klass:TTokenKlass;
     end;

const LexDFATable:Array[1..8,1..8] of byte=(
        {l d + - [ ] 0 eof}
  {1}   (2,3,4,5,6,7,1,8),
  {2}   (2,2,0,0,0,0,0,0),
  {3}   (0,3,0,0,0,0,0,0),
  {4}   (0,3,0,0,0,0,0,0),
  {5}   (0,3,0,0,0,0,0,0),
  {6}   (0,0,0,0,0,0,0,0),
  {7}   (0,0,0,0,0,0,0,0),
  {8}   (0,0,0,0,0,0,0,0));

      LexPushTable:Array[1..8] of Boolean=
         (false,true,true,false,false,True,True,false);

      LexTokenKlass:Array[1..8] of TTokenKlass=
         (Error,Id,Num,Error,Error,LBracket,RBracket,Done);

      NumKeywords=23;

      KWList:Array[1..NumKeywords] of TToken=(
        (Lexeme:'DRAW';Klass:OneOp),
        (Lexeme:'MOVE';Klass:OneOp),
        (Lexeme:'LEFT';Klass:OneOp),
        (Lexeme:'RIGHT';Klass:OneOp),
        (Lexeme:'POINT';Klass:OneOp),
        (Lexeme:'COLOR';Klass:Color),
        (Lexeme:'HOME';Klass:NoOp),
        (Lexeme:'REMEMBER';Klass:NoOp),
        (Lexeme:'GOBACK';Klass:NoOp),
        (Lexeme:'REPEAT';Klass:Rept),
        (Lexeme:'TO';Klass:ToProc),
        (Lexeme:'END';Klass:EndProc),
        (Lexeme:'BLACK';Klass:ColorId),
        (Lexeme:'BROWN';Klass:ColorId),
        (Lexeme:'RED';Klass:ColorId),
        (Lexeme:'ORANGE';Klass:ColorID),
        (Lexeme:'YELLOW';Klass:ColorID),
        (Lexeme:'GREEN';Klass:ColorID),
        (Lexeme:'BLUE';Klass:ColorID),
        (Lexeme:'PURPLE';Klass:ColorID),
        (Lexeme:'GRAY';Klass:ColorID),
        (Lexeme:'WHITE';Klass:ColorID),
        (Lexeme:'';Klass:Null));

{Determines which column of the DFA table to use for this character
  Parameters:
    C: Character to classify
  Returns:
```

```
    DFA column to use for this character}
function Classify(C:Char):Byte;

implementation

function Classify;
begin
  C:=UpCase(C);
  If Pos(C,'ABCDEFGHIJKLMNOPQRSTUVWXYZ_')>0 then begin
    Classify:=1; {Letter}
    Exit;
  end;
  If Pos(C,'0123456789')>0 then begin
    Classify:=2; {Digit}
    Exit;
  end;
  If C='+' then begin
    Classify:=3; {Plus}
    Exit;
  end;
  If C='-' then begin
    Classify:=4; {Minus}
    Exit;
  end;
  If C='[' then begin
    Classify:=5; {LBracket}
    Exit;
  end;
  If C=']' then begin
    Classify:=6; {RBracket}
    Exit;
  end;
  If C=#26 then begin
    Classify:=8; {Eof}
    Exit;
  end;
  Classify:=7; {Not anything else}
end;

end.
```

```
{Parse1.Pas ******************************************************
Recursive Descent Parser for the following grammar
   <Prog>       -> Done | <Statement> <Prog>
   <Statement>  -> NoOp | Id | OneOp Num | Color ColorID |
                   To Id <ProcBody> | Repeat Num [ <ReptBody>
   <ProcBody>   -> End | <Statement> <ProcBody>
   <ReptBody>   -> ] | <Statement> <ReptBody>
*****************************************************************}

unit Parse1;

interface

{Set during the course of a parse.
  True if most recent parse is successful
  False if not
}
     var ValidParse:Boolean;

{Parses the input stream.
  Returns: True if there is more to parse, False if done}
function ParseIt:Boolean;

implementation

uses Lex,MiniLogo,Io,CodeGen;

var Token:TToken;

{Returns true, consumes tokens, and generates
 code if the token stream starts with a <Program>}
function Prog:Boolean;Forward;

{Returns true, consumes tokens, and generates
 code if the token stream starts with a <Statement>}
function Statement:Boolean;Forward;

{Returns true, consumes tokens, and generates
 code if the token stream starts with a <ProcBody>}
function ProcBody:Boolean;Forward;

{Returns true, consumes tokens, and generates
 code if the token stream starts with a <ReptBody>}
function ReptBody:Boolean;Forward;

{Prints the current token, then gets the next token}
Procedure ConsumeToken;
begin
  PrintToken(Token);
  If Not GetNextToken(Token) then begin
    PrintLog('End of Program');
  end;
end;

function Prog:Boolean;
   {<Prog>       -> Done | <Statement> <Prog>}
begin
  If Token.Klass=Done then begin
    PrintLog('   <Prog>       -> Done');
    Prog:=True;
  end else If Statement then begin
    if Prog then begin
      PrintLog('   <Prog>       -> <Statement> <Prog>');
      Prog:=True;
    end else begin
      Error('Program Expected');
      Prog:=False;
    end;
  end else begin
    Error('Statement Expected');
    Prog:=False;
  end;
```

```
end;

function Statement;
  {   <Statement>  -> NoOp | OneOp Num | Color ColorID | Id
                   To Id <ProcBody> | Repeat Num [ <ReptBody>  }
  var CodeTokens:Array[1..3] of TToken;
begin
  If Token.Klass=NoOp then begin
    Statement:=True;
    CodeTokens[1]:=Token;
    ConsumeToken;
    PrintLog('   <Statement>  -> NoOp');
    GenerateNoOp(CodeTokens[1]);
  end else If Token.Klass=Id then begin
    Statement:=True;
    CodeTokens[1]:=Token;
    ConsumeToken;
    PrintLog('   <Statement>  -> Id');
    If Not GenerateProcCall(CodeTokens[1]) then begin
      Statement:=False;
      Error('Undefined Procedure');
    end;
  end else If Token.Klass=OneOp then begin
    CodeTokens[1]:=Token;
    ConsumeToken;
    If Token.Klass=Num then begin
      Statement:=True;
      CodeTokens[2]:=Token;
      ConsumeToken;
      PrintLog('   <Statement>  -> OneOp Num ');
      GenerateOneOp(CodeTokens[1],CodeTokens[2]);
    end else begin
      Statement:=False;
      Error('Number Expected');
    end;
  end else if Token.Klass=Color then begin
    ConsumeToken;
    If Token.Klass=ColorID then begin
      Statement:=True;
      CodeTokens[1]:=Token;
      ConsumeToken;
      PrintLog('   <Statement>  -> Color ColorID');
      GenerateColor(CodeTokens[1]);
    end else begin
      Statement:=False;
      Error('Color ID Expected');
    end;
  end else if Token.Klass=ToProc then begin
    ConsumeToken;
    If Token.Klass<>Id then begin
      Statement:=False;
      Error('Identifier Expected');
      Exit;
    end;
    CodeTokens[1]:=Token;
    ConsumeToken;
    GenerateProcHead(CodeTokens[1]);
    If Not ProcBody then begin
      Statement:=False;
      Error('Procedure Body Expected');
      Exit;
    end;
    Statement:=True;
    PrintLog('   <Statement>  -> To Id <ProcBody>');
  end else if Token.Klass=Rept then begin
    ConsumeToken;
    If Token.Klass<>Num then begin
      Statement:=False;
      Error('Number Expected');
      Exit;
    end;
    CodeTokens[1]:=Token;
```

```
    ConsumeToken;
    If Token.Klass<>LBracket then begin
      Statement:=False;
      Error('Left Bracket Expected');
      exit;
    end;
    GenerateRepeatHead(CodeTokens[1]);
    ConsumeToken;
    If Not ReptBody then begin
      Statement:=False;
      Error('Repeat Body Expected');
      Exit;
    end;
    Statement:=True;
    PrintLog('   <Statement>  -> Repeat Num [ <ReptBody>');
  End Else begin
    Statement:=False;
    Error('Incomplete Statement');
  end;
end;

function ProcBody;
  {   <ProcBody>   -> End | <Statement> <ProcBody>}
begin
  If Token.Klass=EndProc then begin
    ConsumeToken;
    ProcBody:=True;
    GenerateProcReturn;
    PrintLog('   <ProcBody>   -> End');
  End Else if Statement then begin
    If ProcBody then begin
      PrintLog('   <ProcBody>   -> <Statement> <ProcBody>');
      ProcBody:=True;
    End Else Begin
      Error('Procedure Body Expected');
      ProcBody:=False;
    end;
  end else begin
    Error('Statement Expected');
    ProcBody:=False;
  end;
end;

function ReptBody;
  {   <ReptBody>   -> ] | <Statement> <ReptBody>}
begin
  If Token.Klass=RBracket then begin
    ConsumeToken;
    ReptBody:=True;
    GenerateRepeatLoop;
    PrintLog('   <ReptBody>   -> ]');
  End Else if Statement then begin
    If ReptBody then begin
      PrintLog('   <ReptBody>   -> <Statement> <ReptBody>');
      ReptBody:=True;
    End Else Begin
      Error('Repeat Body Expected');
      ReptBody:=False;
    end;
  end else begin
    Error('Statement Expected');
    ReptBody:=False;
  end;
end;

function ParseIt;
begin
  ConsumeToken;
  ValidParse:=Prog;
  ParseIt:=Token.Klass<>Done;
end;
```

```
end.
```