Three Letter Acronym (TLA)

(An introduction to programming concepts for children)

Neil Sarkar, Cody Hess, Jide Atoyebi

1. TLA White Paper

Introduction:

TLA is a programming language designed to help children learn the basics of programming: control structures, functions, problem solving, and proper programming convention.

Children are believed to develop abstract reasoning skills by the fourth/fifth grade. Given our current level of technology and the extent to which it has permeated everyday life it is not uncommon to see young kids browsing the web and installing software. Children are capable of learning much faster than adults. It is these abilities we wish to take advantage of as we decrease the code barrier.

"Programmers" (children) will use Three Letter Acronym to give movement instructions to an inchworm so the inchworm may draw a picture with its silk. Imagine a perfectly trained pet inchworm that went exactly where you told it according to commands like 'step' and 'turn'. This incredibly fun concept will motivate students to learn 'until' and 'if' to move their inchworm to greater achievements!

Design:

The program will tell the inchworm how to move within a predetermined environment. The language will implement the following features:

Commands -

The inchworm receives commands that tell it how to move. All programs are built around these commands:

- 'inch' - the inch command makes the inchworm move forward according to the direction it is facing by a predetermined amount of space.
- 'right' and 'left' - each of these commands will change the direction the inchworm is facing by 90 degrees right or left.
- 'sright' and 'sleft' - each of these commands will change the direction the inchworm is facing by 45 degrees right or left.
- 'silk' - this command will turn the inchworm's "silk" on or off - allowing it to draw pictures with its silk or to move across space without drawing - or to change the color of the silk output. It will accept parameters 'on', 'off', 'green', 'red', 'blue', 'white', and other colors.

Control Structures -

A for loop will allow the child to give more detailed instructions to his or her inchworm.

- 'for' - The keyword of a for loop is 'times'. Children can execute a function or command multiple times by typing "command number times". For example, "inch 10 times" would make the inchworm move forward 10 times.

Functions -

- 'function' - Functions will create reusable code named according to "name" that executes according to "code" like "function name start code end". For example, "function square start inch 10 times right inch 10 times right inch 10 times right inch 10 times end" would draw a square whenever "square" was called in the code.
- 'scale' - This will accept a number and a function as an argument, and change the size of the function's drawing according to the number argument. For example, "scale 2 square" would draw a square twice as big as the one defined in the function "square".
- 'flip' - This will accept a function as an argument and draw a mirror image according to the shape of the function, as though the function was reflected opposite the direction the inchworm was originally facing. An example is "flip square".
- 'rotate' - This will accept a function and a number as arguments and draw that function's shape with as many 90 degree clockwise rotations away from what was originally intended as is signified by the number argument.

Syntax:

Tokens are separated by white space. Each keyword has similar syntax as described in its definition, and functions execute code within the keywords "start" and "end". We've given syntax as few limitations as possible so children can understand the concepts without worrying about the language. Still, we've added two elements of syntax for expert programmers - comments and parenthesis.

- Comments - Any time a line includes a // token, the // and everything after it on the line will be ignored by the parser.
- Parenthesis - Parenthesis can group any set of commands to be repeated by a for loop. For example, the square function could be shortened by typing it as "function square start (inch 10 times right) 4 times end".

Example Program:

```
//DRAWS A SMILEY FACE IN TLA
```

```
//drawing face
scale 10 square

//getting to spot to draw left eye
silk off
right 2 times
inch 2 times
left
inch 2 times

//drawing left eye
silk on
scale 2 square

//getting to spot to draw right eye
silk off
inch 4 times

//drawing right eye
silk on
scale 2 square

//getting to spot to draw smile
silk off
inch 3 times
right 2 times
inch 5 times

//drawing smile
rotate 2 smile

//function for square
function square start
silk on
(inch right) 4 times
end

//function for smile
function smile start
silk on
sright
inch 2 times
sleft
inch 5 times
sleft
```

```
inch 2 times
end
```

Conclusion:

Three Letter Acronym should be an exciting language to introduce young people to computer programming. Heck, we're old and we had a great time writing the sample program!

2. TLA Language Tutorial

Introduction:

TLA is a straightforward language that makes use of control structures, functions in an easy to use fashion. The language has been designed to facilitate ease of use to that end we have introduced functions. These functions enable programmers to nest their desired silkworm commands. Variables were omitted to keep pace with the children's in school knowledge. Circumventing the variables detracted from the power of the language so TLA offers scalability; which enables users to effectively draw according to scale rather than a variable size. In conjunction with the nested functions the language offers users children real choices as to what and how they create their respective masterpieces.

Van Gogh's Hello World:

Far from the traditional hello world we've come to know and love, the silkworm performs this task with due diligence. Using a series of nested inch and right and left commands it is possible to output Hello World for all to see.

Hello World code:

```
//drawing HELLO WORLD

//function for hello world

function hw start

//drawing H
silk off
right 2 times
silk on
inch 2 times
```

```
left 2 times
inch
right
inch
right
inch
left 2 times
inch 2 times
silk off

right
inch 2 times
right
silk on

//drawing E
inch 2 times
left
inch
left 2 times
inch
right
inch
right
inch
left 2 times
inch
right
inch
right
inch
silk off

//drawing Ls
(       inch 2 times
        silk on
        right
        inch 2 times
        left
        inch
        silk off
        left
        inch 2 times
        right
) 2 times       //nesting  used to create two L characters

//drawing O
inch 2 times
```

```
silk on
inch
right
inch 2 times
right
inch
right
inch 2 times
right
silk off
inch 4 times

//drawing W
silk on
sright
inch 2 times
left
inch
right
inch
left
inch 2 times
sright
silk off
inch 2 times

//drawing O
inch 2 times
silk on
inch
right
inch 2 times
right
inch
right
inch 2 times
right
silk off
inch 3 times

//drawing R
silk on
inch
right
inch
right
inch
right 2times
```

```
sright
inch
left 2 times
inch
sleft
left
inch
right 2 times
inch 2 times
right
inch
silk off

//drawing L
inch 2 times
silk on
right
inch 2 times
left
inch
silk off
left
inch 2 times
right
inch 2 times

//drawing D
silk on
sright
inch
right
inch
right
sright
inch 2 times

end
```

The Breakdown

The Hello World example provided above illustrates the body of executable commands permissible within the TLA programming language.

Comments have been denoted using the '//' symbols. This will afford children the opportunity to properly chronicle their coding and perhaps develop other keys skill some of us have failed to exercise.

Example: //this is how I comment my code.

Drawing the letters was accomplished by turning the silk on and off. The on position allows you to draw the letter out while the off position enabled us to space the characters.

Example: silk on
    // this turns the silk on
    // the silk color may be changed in a similar way by specifying the color

The actual drawing was performed by having the silkworm inch and turn at the programmers discretion. The inch command moves the worm forward one standard unit while the right & left commands turn in their respective directions. sright and sleft allow users to turn the work at 45° rather than the traditional 90°.

Example: inch
    right
    inch
    left
    // causes the worm to create 2 sides of a square provided silk is on

In addition to being able to direct the worm we are also capable of telling it how many times to perform a certain operation.

Example: inch 2 times
    sright 2 times //equivalent of one 'right'

Nesting is another important feature, which allows programmers to imbed multiple calls to a function or command. Nesting can be used to create multiple objects or to simply reduce redundancy within your code.

Example:
```
(       inch 2 times
        silk on
        right
        inch 2 times
        left
        inch
        silk off
        left
        inch 2 times
        right
) 2 times

// this creates the two consecutive Ls as seen in the Hello World program.
```

The hello world displayed is a function one which may be called over and over by simply referring to it as ‘hw’ it’s designated function name. This function simply prints out drawing hello world.

Having created the function hw it is possible to use terms like scale or rotate to impact the size and orientation of the function.

Example: scale 2 hw // this would increase the output by a factor of 2

Using the rotate command would simply rotate the drawing based on the designated degree number.

3. LRM

1 – Introduction

TLA is a programming language designed to help children learn the basics of programming: control structures, functions, problem solving, and proper programming convention.

These young programmers will be exposed to the TLA language in hopes of providing them with a particular type of outlook. This outlook is intended to initiate and stimulate centers of the brain not often used by individuals so young. Exposing our young programmers to conditionals and other programming concepts should provide them with a measure of familiarity. "Programmers" will use TLA to give movement instructions to an inchworm so the inchworm may navigate a maze or draw a picture with its silk. Using conditionals and commands like ‘step’ and a directional ‘turn’ children will enter a world in which the control the fate and actions of their given inch worm. The combination of learning a fun should facilitate a greater desire to use the language. The language and the some of the mental stimuli provided should provide its’ programmers with a constant source of puzzles and conundrums to solve.

2 – Overview of Lexical Conventions

The TLA language is made up of an assortment of parts including keywords, operators, statements, functions and basic syntax.

2.1 Keywords

The words seen in section 2.1 have been reserved by the TLA programming language for explicit use.

for	print	inch	right
left	sright	sleft	function
scale	flip	rotate	silk

2.3 Statements

The following statements provided added functionality within the TLA language. These statements initiate more complex task not possible by simply using the operators listed above.

2.3.1 The For Statement ‘for’

The ‘for’ statement is a statement that creates a loop one which executes an action for as long as the condition is satisfied. Within the ‘for’ loop the loop time is defined.

Example: sright 8 times // this causes the worm to turn 360°

2.3.2 The Inch Statement ‘inch’

The ‘inch’ statement enables the programmer’s inch-worm to move forward one standard unit.

Example: inch 4 times // causes the worm to move forward 4 units

2.3.3 The Silk Statement ‘silk’

The ‘silk’ statement allows the worm to display it’s path by laying a trail of silk. This trail may be activated or deactivated by the programmer in addition the color may also be determined.

Example: silk on silk off silk red inch // turns the silk on & off then
// changes color and inches

2.3.4 The Right & Left Statements ‘right’ & ‘left’

The ‘right’ & ‘left’ statements allow the worm to perform 90° turns in the direction specified.

Example: (inch right) 4 times // creates a square

2.3.5 The SRight & Sleft Statements ‘sright’ & ‘sleft’

The ‘sright’ & ‘sleft’ statements allow the worm to perform 45° turns in the direction specified.

Example: sright (inch right) 4 times // creates a diamond

2.4 Functions

2.4.1 Function ‘function’

The function ‘function’ will create reusable code based on a user construct. These functions will then perform a predefined set of actions.

Example: function diamond start sright (inch right) 4 times end

2.4.2 Scale ‘scale’

The ‘scale’ function will take a number and a function and change the function by a factor of the number taken. This will impact the magnitude of the functions output or drawing.

Example: scale 2 diamond

2.4.3 Flip ‘flip’

The ‘flip’ function will rotate and invert the image so that the original image appears to be reflected.

Example: flip diamond // won’t be noticed given the shape

2.4.4 Rotate ‘rotate’

The ‘rotate’ function will spin the image by 90° clockwise. The number of rotations will be determined by the programmer who inputs that variable number. The default rotation is one 90° revolution.

Example: rotate diamond

2.5 Syntax

2.5.1 Comments ‘//’

The ‘//’ is the syntactical marker for commented code. Comments begin at the first instance of the ‘//’, they end at the end line.

Example: // this is what a comment looks like

2.5.2 Parenthesis ‘(’ & ‘)’

The parenthesis are syntactical markers that enable programmers to group commands. The ‘(‘ begins the grouping and the ‘)’ ends the grouping.
Example: (scale 2 square inch) 4 times

3 - Example Program

```
        //drawing H
silk off
right 2 times
silk on
inch 2 times
left 2 times
inch
right
inch
right
inch
left 2 times
inch 2 times
silk off

right
inch 2 times
right
silk on

//drawing E
inch 2 times
left
inch
left 2 times
inch
right
inch
right
inch
left 2 times
inch
right
inch
right
inch
silk off

//drawing Ls
(       inch 2 times
        silk on
        right
        inch 2 times
        left
```

```
        inch
        silk off
        left
        inch 2 times
        right
) 2 times       //nesting  used to create two L characters

//drawing O
inch 2 times
silk on
inch
right
inch 2 times
right
inch
right
inch 2 times
right
silk off
```

4. TLA Project Plan

Division of Labor

Cody: Front end coding, integration & documentation
Jide: Error testing, tree-walking, documentation
Neil: Back end coding, integration, tree walking & lexer & parser

Development Process

The work was split and assigned using time as the critical and deciding factor in lot assignment. The White Paper, the LRM and the lexer & parser were assigned at the same time. The tree walking followed suit and was assigned to one individual with assistance provided as needed. Error

testing was performed to ensure that the grammatical conventions were maintained. The nature of the front and back end components made it possible to divide them and use the individual documenting as a go between when necessary. The go between furthered the role by providing support in the form of error testing on both ends. Conventions were established and maintained to provide general order and symmetry. Finally the front and back ends were linked with both parties present to smooth out the small kinks that still existed.

Timeline

11/22/04	Language prospects discussed, White paper outlined and completed.
11/23/04	As defined within the White Paper, Lexer and Parser outline.
11/24/04	Lexer and Parser completed, LRM outlined and completed.
11/25/04	Discussions designed reduce scope while improving usefulness.
11/30/04	Back end work begins, Speculative documentation follows.
12/01/04	Front end work begins, Documentation follows.
12/03/04	Error testing for prototype back end demo begins.
12/06/04	Error testing for prototype front end demo begins.
12/17/04	Front end code completed, Documentation follows.
12/18/04	Front end integration begins based on current back end.
12/19/04	Documentation
12/20/04	Back end code completed, Documentation follows.
12/21/04	Integration completed, Documentation
12/22/04	10:50am Power Point Presentation completed for 11am viewing.

Software Development Environment

TLA was created on machines running two different operating systems. Backend and portions of integration were performed in Windows Xp. Front end and the other portion of integration was created on a Linux box running Gentoo. Documentation was performed on a Mac running OS X 10.3.7. TLA was created using Java SDK 1.4.1. The lexer and parser were created using Java based technology ANTLR 2.7.1.

Code Style & Convention for Developers

Within a group setting convention becomes an important feature as it facilitates easier exchanges of code segments. Comments were at the core of what we used to ensure that the coding style remained on par and that conventions were adhered to. Notes were left as headers providing

detailed information for use during the integration process. The code was later removed upon completion of the integration process so as to keep the code relatively neat.

5. TLA Architecture Plan

Elements:

- Lexer
- Parser
- Tree Walker

Lexer:

The lexer has a simple, obvious, yet important job. It checks the user-written code for correct syntax. If the syntax is incorrect, it halts compilation with an error message. If correct, the code is allowed to proceed to the parser.

Parser:

The parser checks user code for grammatical correctness, and sorts (parses) the code into a tree object according to the rules of the language – separate tree nodes for looped blocks, functions, and the like.

Tree Walker:

The tree walker generates the graphic code which displays the programmed drawing. The command tree object generated by the parser is fed to the tree-walker, which steps through each tree node and generates java applet commands accordingly. It keeps global variables which represent the current cursor position, line color, direction, and etc.

The tree walker is responsible for remembering any defined function and recreating the code necessary every time that function is called.

The file InchApp.java (created by the tree walker) is an applet wrapped by pre-written code called InchwormGUI.java. These objects are then compiled by a java compiler and run to display the image the user generated.

6. Error Testing

The error testing for the TLA language happened in two phases for the front end, back end and the unionization. White box testing the easier of the two methods was used to ensure that the know commands all worked properly. Black box testing was done without the use of a test suite. Due to the structure of the language we decided it would be simpler to test the back end using expressions and statements that we knew should be rejected outright.

The white box testing for the front end relied on ensuring that the functions were in proper working order. Each command was tested in turn but it was important to ensure that the functions executed properly and that when attempting to execute a given command on a function that it execute with no problems. To that end mock functions were created and tested to make sure that when scale was called upon it that the dimensions were altered accordingly.

Ex1: square // a function square being called

Ex2: scale 2 square // calling scale on a square to alter it's dimensions by a factor of 2

These two examples highlight the core concerns and methods for ensuring the success of TLA's functions. Further testing of TLA was used to ensure that core commands like inch and silk functioned properly.

Ex: silk on
inch
silk off
inch

These are a few examples to illustrate the method in used to confirm the functionality of commands as well the functionality of the functions. White box testing was performed on the gui and the applet to ensure that neither one crashed due to the standard outputs. Testing was also done to make sure the file closed functioned properly and without incident.

The white box testing for the back end portion was performed in a similar by confirming that statements that should be accepted were in fact accepted. First by testing the lexer to ensure that grammatically correct statements were recognized.

Ex: (scale 2 square silk off inch silk on) 4 times // this should increase the dimensions 2x
// and create four copies side by side

It was essential to do thorough testing of nested statements to make sure the nesting was maintained and parsed properly. This examples test the lexer to make certain that the lexical conventions are recognized and not rejected. It then tests to ensure that the code is parsed properly based on the scripting.

The white box testing performed after the unification was designed to make certain that segments of the front and back ends mapped properly. Code was then written up and sent through the lexer and parser upon successful compilation the code was run. This portion of the white box testing involved going back and forth to make certain the ties worked. The methods used were a combination of the two previous cases. The back end would sent some code upon the successful completion of that phase of testing it was then necessary to ensure that the proper output appeared within the gui.

The Black box testing that took place for the back end was done devising statements of varying complexity. These were statements that should not have been accepted. This was done attempting numerous diversions of proper statements and using nonsensical statements.

Ex: scale x square

This is an example of trying to scale a function using a non-integer value.

The black box testing performed on the front end was done primarily using exception capturing as a means of sifting through potential woes. The hit or miss process for locating holes was also used.

Once again unification phase involved using both methods in conjunction to ensure that the front didn't output some invalid statement and the back didn't accept the invalid statement.

7. Lessons Learned

Creating a language from scratch was quite obviously a rather new endeavor for all of us. We each had our own ideas of what we wanted in a language. More importantly we were all comfortable with meshing those ideas together where possible. The potential for conflicts down the road was greatly

improved when upon pitching ideas we discovered that language could be created to suit all our needs.

Through the course of the project we discovered the importance of teamwork and compromise. The latter of the two proved necessary on a number of occasions.

A common concern on the part of all members was the idea of scope. We were quite enamored with our language and it's potential to educate and instruct children in the proper way to conceptualize programming. Compromise was necessary many times along the road to completion as our vision for the perfect TLA language met up with the unfortunate time constraints of the standard Columbia semester. It would become necessary to restrict TLA to areas that would have the most profound impact and ability to achieve our stated objective.

The teamwork aspect came into play when we divvied up the work and attempted to match talent with sections of the project. It was necessary to maximize our time by using the best man for the job. We all wanted to write the lexer and parser but only the best lp guy would get to do it. Each team member was assigned a task and a time period to complete the work. There was a measure of overlap as class loads reached critical points. This brings up the first piece of advice namely make sure that each member of the group is capable of performing at least one other persons job so as to insulate yourself against problems. This measure also allows for increased output should the need arise.

Planning is probably the most essential thing anybody can recommend. Though we were lucky to have found each other not all groups will enjoy still waters. Good planning may help to overcome issues involving teamwork and compromise. Lone wolves should still adhere to timelines even if they seem to flout the group dynamic. The timelines we used and the frequency with which we met went a long way to making sure each of us was on the same page at the end of the week.

Ultimately the daily exchanges of ims and the friendship bracelets we made during our first meeting contributed to the strong bond and brilliant group dynamic.

Finally meet with the advisor they are quite helpful in ironing out all the wrinkles.

8. Appendix

TreeWalker.g -

header {

```
   import java.awt.Color;
   import java.util.*;
   import java.io.*;
   import antlr.*;
   import antlr.collections.AST;
   import java.util.Random;
}


class TLAWalker extends TreeParser;
options {
   importVocab = TLA;
   buildAST = true;
}

{
   //function lookup table stuff
   Hashtable takenFunctions = new java.util.Hashtable();
   Random random = new Random();

   //code generator. inFunction is so that the code generator knows which file to write to
(InchApp.java or functions.tmp)
   FileGenerator fg = new FileGenerator();
   boolean inFunction = false;
}

//walk is the only rule in this tree walker. it handles all possible nodes
walk
   :  #(root:STARTCOMMANDS
         {
            fg.initialize();

            //this is a common procedure here. it says process all the commands that are
children of this node
            AST currentCommand = root.getFirstChild();
            do {
               walk(currentCommand);
            } while( (currentCommand = currentCommand.getNextSibling()) != null );

            //this actually does a lot of stuff. look at FileGenerator.java
            fg.close();
         }
      )
   |  #(dFunc:FUNCCOMMANDS
         {
            AST currentCommand = dFunc.getFirstChild();
            String f = currentCommand.getText();
```

```
            //FileGenerator creates a function if one with the same name doesn't exist
already
            if( !takenFunctions.containsKey(f) ) {
               takenFunctions.put( f, new Integer(random.nextInt()) );
               currentCommand = currentCommand.getFirstChild();
               fg.defineFunction(f);

               inFunction = true;

               //see. same loop again
               do {
                  walk(currentCommand);
               }
               while( (currentCommand = currentCommand.getNextSibling()) != null );
               inFunction = false;
               fg.endFunction();
            }

            else
               System.err.println("The function " + f + " has already been defined");
         }
      )
   |  #(eFunc:FUNCNAME
         //EXECUTES A FUNCTION

         {
            String f = eFunc.getText();
            if( !takenFunctions.containsKey(f) ) {
               System.out.println("Function " + f + " has not been defined");            }

            //this is a little verbose because the rotate stuff has 4 possible cases, but it
basically just evaluates whether or not to run a built in function from this node, and what
the values of the parameters are, and passes this information to FileGenerator.java
            else {
               int scale = 1;
               int rotate = 0;
               if( eFunc.getFirstChild() != null ) {
                  eFunc = eFunc.getFirstChild();
                  do {

                     if( eFunc.getText().equalsIgnoreCase("scale") )
                        scale = -1;
                     else if( eFunc.getText().equalsIgnoreCase("sright") )
                        rotate = -1;
                     else if( eFunc.getText().equalsIgnoreCase("right") )
                        rotate = - 2;
                     else if( eFunc.getText().equalsIgnoreCase("left") )
                        rotate = -6;
```

```
                else if( eFunc.getText().equalsIgnoreCase("sleft") )
                   rotate = -7;

                else {
                   if( scale == -1 )
                      scale = Integer.parseInt(eFunc.getText());
                   else if( rotate == -1 )
                      rotate *= Integer.parseInt(eFunc.getText());
                   else
                      System.out.println("Syntax error in function " + f);
                }
             } while( (eFunc = eFunc.getNextSibling()) != null );
             rotate *= -1;
          }
          fg.runFunction(f, scale, rotate);
       }
    }
  )
| #(times:INT

    //this matches a repeated command stream, e.g.
    //(inch right 2 times) 3 times
    {
       AST currentCommand;

       for( int i = 0; i < Integer.parseInt(times.getText()); i++ ) {
          currentCommand = times.getFirstChild();
          do {
             if( currentCommand.getText().equals("Function") )
                System.err.println("Error: can't declare a function within...");
             else
                walk(currentCommand);
          } while( (currentCommand = currentCommand.getNextSibling()) != null );
       }
    }
  )
| #(silk:"silk"
    {
       //passes the first child node of silk (the silk parameter) to the code generator
       fg.changeSilk(silk.getFirstChild().getText(), inFunction);
    }
  )
| #(inch:"inch"
    {
       //all the complex inch stuff is in the code generator
       int i = 1;
       if( inch.getFirstChild() != null )
          i = Integer.parseInt(inch.getFirstChild().getText());
```

```
            fg.inch(i, inFunction);
        }
    )
  // the next 4 rules are basically the same rule, for the 4 cases of turn
  |  #(right:"right"
        {
            int i = 2;
            if( right.getFirstChild() != null )
                i *= Integer.parseInt(right.getFirstChild().getText());
            fg.turn(i, inFunction);
        }
    )
  |  #(left:"left"
        {
            int i = 6;
            if( left.getFirstChild() != null )
                i *= Integer.parseInt(left.getFirstChild().getText());
            fg.turn(i, inFunction);
        }
    )
  |  #(sright:"sright"
        {
            int i = 1;
            if( sright.getFirstChild() != null )
                i *= Integer.parseInt(sright.getFirstChild().getText());
            fg.turn(i, inFunction);
        }
    )
  |  #(sleft:"sleft"
        {
            int i = 7;
            if( sleft.getFirstChild() != null )
                i *= Integer.parseInt(sleft.getFirstChild().getText());
            fg.turn(i, inFunction);
        }
    )
  ;
```

TLA.g –

```
////////////////////////////////////////////////////////////////////////////

header {
  /* */
}
```

```
/* PARSER */

class TLAParser extends Parser;

options {
   buildAST=true;
   k = 2;
   exportVocab = TLA;
}


//tree walker needs to be able to distinguish btwn. regular commands and commands in
functions for the purposes of code generation
tokens {
   STARTCOMMANDS;
   FUNCCOMMANDS;
}

//programs are "start" (not included in tree), a series of commands, and "end" (not
included in tree). In tree walker, creates a top level node called Program
program
   : "start"! (command)+ "end"! EOF!
      { #program = #([STARTCOMMANDS, "Program"], #program); }
   ;


/* commands */

command
   :  LPAREN! (command)+ RPAREN! INT^ "times"!
   |  silkCmd
   |  inchCmd
   |  turnCmd
   |  defineFunction
   |  execFunction
   ;

silkCmd
   :  "silk"^ ("on" | "off" | "white" | "blue" | "purple" | "yellow"
   | "orange" | "red" | "pink" | "green");

inchCmd
   :  "inch"^ (INT "times"!)?//(ifStmt)?
   ;

turnCmd
   :  ("right"^ | "left"^ | "sright"^ | "sleft"^) (INT "times"!)?
   ;
```

```
defineFunction
   :  "function"! FUNCNAME^ "start"! (command)+ "end"!
      {#defineFunction = #([FUNCCOMMANDS,"Function"], #defineFunction);}
   ;

execFunction
   :  FUNCNAME^ ("scale" INT)?
      (("rotate"! ("right"|"left"|"sright"|"sleft") (INT "times"!)?))?
   ;


////////////////////////////////////////////////////////////////////////

/* LEXER */


class TLALexer extends Lexer;

options {
   k=3; //for newline stuff
   charVocabulary='\u0000'..'\u007F';
   testLiterals = false;
   caseSensitive=false;
   exportVocab = TLA;
}


LPAREN  :  '(' ;
RPAREN  :  ')' ;


FUNCNAME options {testLiterals=true; paraphrase="name of a valid function";}
   :  LETTER (LETTER | DIGIT)* ;



protected DIGIT: '0'..'9';
protected LETTER: ('a'..'z' | '_');
INT: (DIGIT)+;



COMMENT
   :  "/*"
      ( options {greedy=false;} :
         ~('\n'|'\r')
      | ('\n'|'\r') {newline();}
```

```
        )*
        "*/"
        { $setType(Token.SKIP); }
    |   "//" (~('\n'|'\r'))*
        { $setType(Token.SKIP); }
    ;

WS
    :   (' '
        |'\t'
        | ( "\r\n"
          | '\n'
          | '\r'
          )
          {newline(); }
        )
        { $setType(Token.SKIP); }
    ;
```

//

FileGenerator.java –

/* Neil Sarkar 12/22/2004

This gets invoked by the tree walker. The basic way it works is that it determines from tree walker input a) what it should be writing and b) whether the command(s) should be in a function or in the main body of the code.
If in the main body, it writes directly to the init() method of InchApp.java. If in a function, it writes to a separate file called functions.tmp. At the end of the tree walker input, it copies the functions in functions.tmp to InchApp.java, and generates two necessary functions for any tla program--inch(distance) and pause()

*/

```
import java.io.*;

public class FileGenerator {

    File app;
    FileWriter applt;

    File functions;
    FileWriter func;

    public FileGenerator() {
            try {
                app = new File("InchApp.java");
```

```
            applt = new FileWriter( "InchApp.java", false);

            functions = new File("functions.tmp");
            func = new FileWriter( "functions.tmp", false);
        }
        catch(IOException e) { System.err.println(e); }
    }

    public void initialize() {
        try {
            applt.write("import java.applet.*;\n");
            applt.write("import java.awt.*;\n\n");
            applt.write("public class InchApp extends Applet { \n\n");
            applt.write("    private int direction = 2;\n");
            applt.write("    private int x = 100; private int y = 205;\n");
            applt.write("    private boolean silk = true;\n");
            applt.write("    private final int INCH_DISTANCE = 15;\n");
            applt.write("    public void init() {\n");
            applt.write("        this.setBackground( Color.black );\n");
            applt.write("        Graphics g = getGraphics();\n");
            applt.write("        g.setColor( Color.white );\n");
        }
        catch( IOException e ) { System.err.println("Error in initialising: " + e); }
    }

    public void defineFunction( String funcName ) {
        try {
            func.write("    private void " + funcName + "(Graphics g, int x, int y, int scale,
int rotate) {\n");
            func.write("        direction += rotate;\n");
        }
        catch( IOException e ) {System.err.println("Error in defining " + funcName + ": "
+ e); }
    }

    public void endFunction() {
        try {
            func.write("    }\n\n");
        }
        catch( IOException e ) { System.err.println("Error ending function: " + e); }
    }

    public void runFunction( String funcName, int scale, int rotate ) {
        try {
            applt.write("        " + funcName + "(g, x, y, " + scale + ", " + rotate + ");\n" );
        }
        catch( IOException e ) { System.err.println("Error running function " + funcName
+ ": " + e ); }
```

```
    }


    public void turn(int degree, boolean inFunction) {
        try {
           if( inFunction )
                func.write("        direction += " + degree + ";\n");
           else {
                applt.write("       direction += " + degree + ";\n");
           }
        }
        catch( IOException e ) { System.err.println("Error turning: " + e); }
    }

    public void inch(int distance, boolean inFunction) {
        try {
           if( inFunction )
                func.write("        inch(g, scale * " + distance + ");\n");
           else
                applt.write("       inch(g, " + distance + ");\n");
        }
        catch( IOException e ) { System.err.println("Error in inch " + distance + ": " + e);
}
    }

    public void changeSilk( String color, boolean inFunction ) {
        String currentColor = "";
        boolean colorChange = false;
        try {
           if( color.equalsIgnoreCase("on") && inFunction)
                func.write("        silk = true;\n");
           else if( color.equalsIgnoreCase("on") && !inFunction)
                applt.write("       silk = true;\n");
           else if( color.equalsIgnoreCase("off") && inFunction)
                func.write("        silk = false;\n");
           else if( color.equalsIgnoreCase("off") && !inFunction)
                applt.write("       silk = false;\n");
        }
        catch( IOException e ) { System.err.println("Error in silk " + color + ": " + e); }

        if( color.equalsIgnoreCase("purple") ) {
           currentColor = "new Color(160,32,240)";
           colorChange = true;
        }
        else if( !color.equalsIgnoreCase("off") && !color.equalsIgnoreCase("on") ) {
           currentColor = "Color." + color;
           colorChange = true;
        }
```

```
        if( colorChange ) {
           try {
                 if( inFunction ) {
                    func.write("       g.setColor( " + currentColor + " );\n");
                    func.write("       silk = true;\n");
                 }
                 else {
                    applt.write("       g.setColor( " + currentColor + " );\n");
                    applt.write("       silk = true;\n");
                 }
           }
           catch( IOException e ) { System.err.println("Error in setting color " + color + ":
" + e); }
        }
   }

   public void close() {
        try {
           applt.write("\n    }\n\n");

           func.close();

           BufferedReader br = new BufferedReader( new FileReader("functions.tmp") );
           String line;
           while( (line = br.readLine()) != null ) {
                 applt.write(line);
                 applt.write("\n");
           }

           applt.write("    private void inch(Graphics g, int distance) {\n        int x0 = x; int
y0 = y;\n        distance *= INCH_DISTANCE;        int halfDistance =
(int)(distance/2);\nswitch( direction % 8 ) {\ncase 0: y -= distance; break;\ncase 1: x +=
halfDistance; y -= halfDistance; break;\ncase 2: x += distance; break;\ncase 3: x +=
halfDistance; y += halfDistance; break;\ncase 4: y += distance; break;\ncase 5: x -=
halfDistance; y += halfDistance; break;\ncase 6: x -= distance; break;\ncase 7: x -=
halfDistance; y -= halfDistance; break;\n}\n        if( x > 600 ) x = 600; else if( x < 0 ) x =
0;\n        else if( y > 600 ) y = 600; else if( y < 0 ) y = 0;\n");
        applt.write("        if( silk ) g.drawLine(x0, y0, x, y); ;\n");
           applt.write("        pause();\n    }\n\n");

           applt.write("    private void pause() { pause(660); }\n    private void pause( int
duration ) {\n        try{ Thread.currentThread().sleep(duration); } \n        catch(
InterruptedException e ) { e.printStackTrace(); }\n    }\n}");
           applt.close();
        }
        catch( IOException e ) { System.err.println("Error in closing files: " + e); }
```

```
    }

}
```

Run.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.debug.misc.ASTFrame;
import antlr.collections.AST;

public class Run {

    public static void main(String[] args) throws Exception {

        if( args.length < 1 )
            System.out.println("@ Please specify file to parse");

        else {
            try {
            BufferedReader file = new BufferedReader(new FileReader(args[0]));
            TLALexer lexer = new TLALexer(file);
            TLAParser parser = new TLAParser(lexer);

            parser.program();
            CommonAST tree = (CommonAST)parser.getAST();
            //System.out.println("TREE:\n" + tree.toStringList());

            //ASTFrame frame = new ASTFrame("TLA Parser for " + args[0], tree);
            //frame.setVisible(true);
            TLAWalker walker = new TLAWalker();
            walker.walk(parser.getAST());
            }

            catch( Exception er ) {
                System.err.println("error: " + er);
            }
        }
    }
}
```

ViewTree.java –

```
import java.io.*;
```

```
import antlr.CommonAST;
import antlr.debug.misc.ASTFrame;
import antlr.collections.AST;

public class ViewTree {

   public static void main(String[] args) throws Exception {
         if( args.length < 1 )
            System.out.println("@ Please specify file to parse");

         else {
            try {
            BufferedReader file = new BufferedReader(new FileReader(args[0]));
            TLALexer lexer = new TLALexer(file);
            TLAParser parser = new TLAParser(lexer);

            parser.program();
            CommonAST tree = (CommonAST)parser.getAST();
            System.out.println("TREE:\n" + tree.toStringList());

            ASTFrame frame = new ASTFrame("TLA Parser for " + args[0], tree);
            frame.setVisible(true);
            }
```

```
            catch( Exception er ) {
                System.err.println("error: " + er);
            }
        }
    }
}
```

InchGUI.java –

```
/* Cody Hess 12/2004 */
import java.applet.*;
import java.awt.*;
// Wrap the Applet in a GUI so it can run by itself

public class InchGUI extends InchApp {

    public static void main(String args[]) {

        Applet applet = new InchGUI();

        Frame frame = new AppletFrame("Three Letter Acronym", applet, 600, 600);

    }
```

```
}


class AppletFrame extends Frame {

    public AppletFrame(String title, Applet applet, int width, int height) {

        // create the Frame with the specified title

        super(title);



        // Add a menubar, with a File menu, with a Quit button.

        MenuBar menubar = new MenuBar();

        Menu file = new Menu("File", true);

        menubar.add(file);

        file.add("Quit");
```

```
    this.setMenuBar(menubar);



    // Add the applet to the window.  Set the window size.  Pop it up.

    this.add("Center", applet);

    this.resize(width, height);

    this.show();



    // Start the applet.

    applet.init();

    applet.start();

}

// Handle the Quit menu button.
```

```
    public boolean action(Event e, Object arg)

    {

        if (e.target instanceof MenuItem) {

            String label = (String) arg;

            if (label.equals("Quit")) System.exit(0);

        }

        return false;

    }

}
```

InchApp.java –

```
import java.applet.*;
import java.awt.*;

public class InchApp extends Applet {
```

```
private int direction = 2;
private int x = 100; private int y = 205;
private boolean silk = true;
private final int INCH_DISTANCE = 15;
public void init() {
    this.setBackground( Color.black );
    Graphics g = getGraphics();
    g.setColor( Color.white );
    g.setColor( Color.red );
    silk = true;
    H(g, x, y, 2, 0);
    space(g, x, y, 1, 0);
    g.setColor( Color.white );
    silk = true;
    E(g, x, y, 1, 0);
    space(g, x, y, 1, 0);
    g.setColor( Color.blue );
    silk = true;
    L(g, x, y, 1, 0);
    space(g, x, y, 1, 0);
    g.setColor( Color.blue );
    silk = true;
    L(g, x, y, 1, 0);
```

```
space(g, x, y, 1, 0);
g.setColor( Color.red );
silk = true;
O(g, x, y, 1, 0);
space(g, x, y, 1, 0);
space(g, x, y, 1, 0);
space(g, x, y, 1, 0);
space(g, x, y, 1, 0);
g.setColor( new Color(160,32,240) );
silk = true;
W(g, x, y, 2, 0);
space(g, x, y, 1, 0);
g.setColor( Color.yellow );
silk = true;
O(g, x, y, 1, 0);
space(g, x, y, 1, 0);
g.setColor( Color.green );
silk = true;
R(g, x, y, 1, 0);
space(g, x, y, 1, 0);
g.setColor( Color.yellow );
silk = true;
L(g, x, y, 1, 0);
space(g, x, y, 1, 0);
```

```
    g.setColor( new Color(160,32,240) );
    silk = true;
    D(g, x, y, 1, 0);
    space(g, x, y, 1, 0);

}

private void space(Graphics g, int x, int y, int scale, int rotate) {
     direction += rotate;
     silk = false;
     inch(g, scale * 1);
     silk = true;
}

private void H(Graphics g, int x, int y, int scale, int rotate) {
     direction += rotate;
     silk = true;
     direction += 6;
     inch(g, scale * 2);
     direction += 4;
     inch(g, scale * 1);
     direction += 6;
     inch(g, scale * 1);
     direction += 6;
```

```
        inch(g, scale * 1);
        direction += 4;
        inch(g, scale * 2);
        direction += 6;
    }

    private void E(Graphics g, int x, int y, int scale, int rotate) {
        direction += rotate;
        inch(g, scale * 1);
        direction += 12;
        inch(g, scale * 1);
        direction += 2;
        inch(g, scale * 2);
        direction += 2;
        inch(g, scale * 1);
        direction += 2;
        silk = false;
        inch(g, scale * 1);
        direction += 2;
        silk = true;
        inch(g, scale * 1);
        direction += 6;
        inch(g, scale * 1);
        direction += 6;
```

```
    inch(g, scale * 1);
}

private void L(Graphics g, int x, int y, int scale, int rotate) {
    direction += rotate;
    direction += 6;
    inch(g, scale * 2);
    direction += 4;
    inch(g, scale * 2);
    direction += 6;
    inch(g, scale * 1);
}

private void O(Graphics g, int x, int y, int scale, int rotate) {
    direction += rotate;
    inch(g, scale * 1);
    direction += 6;
    inch(g, scale * 2);
    direction += 6;
    inch(g, scale * 1);
    direction += 6;
    inch(g, scale * 2);
    direction += 6;
    inch(g, scale * 1);
```

```
}

private void W(Graphics g, int x, int y, int scale, int rotate) {
    direction += rotate;
    direction += 6;
    inch(g, scale * 2);
    direction += 4;
    inch(g, scale * 2);
    direction += 21;
    inch(g, scale * 2);
    direction += 2;
    inch(g, scale * 2);
    direction += 21;
    inch(g, scale * 2);
    direction += 4;
    inch(g, scale * 2);
    direction += 6;
}

private void R(Graphics g, int x, int y, int scale, int rotate) {
    direction += rotate;
    direction += 6;
    inch(g, scale * 2);
    direction += 2;
```

```
    inch(g, scale * 1);
    direction += 2;
    inch(g, scale * 1);
    direction += 2;
    inch(g, scale * 1);
    direction += 21;
    inch(g, scale * 2);
    direction += 7;
}

private void D(Graphics g, int x, int y, int scale, int rotate) {
    direction += rotate;
    direction += 6;
    inch(g, scale * 2);
    direction += 3;
    inch(g, scale * 2);
    direction += 2;
    inch(g, scale * 2);
    direction += 6;
    direction += 7;
}

private void inch(Graphics g, int distance) {
   int x0 = x; int y0 = y;
```

```
        distance *= INCH_DISTANCE;        int halfDistance = (int)(distance/2);
switch( direction % 8 ) {
case 0: y -= distance; break;
case 1: x += halfDistance; y -= halfDistance; break;
case 2: x += distance; break;
case 3: x += halfDistance; y += halfDistance; break;
case 4: y += distance; break;
case 5: x -= halfDistance; y += halfDistance; break;
case 6: x -= distance; break;
case 7: x -= halfDistance; y -= halfDistance; break;
}
        if( x > 600 ) x = 600; else if( x < 0 ) x = 0;
        else if( y > 600 ) y = 600; else if( y < 0 ) y = 0;
        if( silk ) g.drawLine(x0, y0, x, y); ;
        pause();
    }

    private void pause() { pause(660); }
    private void pause( int duration ) {
        try{ Thread.currentThread().sleep(duration); }
        catch( InterruptedException e ) { e.printStackTrace(); }
    }
}
```