Coding and Geometry

Teacher Version

Authors:

Michelle Hills

Fernando Alegre

Juana Moreno

Summer 2015

LA-SiGMA
Louisiana Alliance for Simulation-Guided Materials Applications

LSU

Coding and Geometry:
Table of Contents

Introduction

Overview of Coding

Reference Sheet

Lesson 1:

- Points
- Segments
- Lines

Lesson 2: Intersections

- Line-Line
- Line-Circle
- Circle-Circle

Lesson 3:

- Midpoint
- Distance

Lesson 4: Angles

- Angle Measure
- Angle Bisector

Lesson 5: Lines

- Perpendicular Lines
- Parallel Lines

Lesson 6:

- Perimeter
- Area of Triangles

Lesson 7: Equilateral Triangle

Lesson 8: Transformations

- Translate
- Rotate
- Reflect
- Dilate

End of the Year Project

- Sample rubric accessible at http://rubistar.4teachers.org/index.php?screen=ShowRubric&rubric_id=2547984&

Coding and Geometry

What?

Coding and Geometry is a method of illustrating the use of coding in the context of high school geometry. The students will learn skills to manipulate programs step by step throughout the lessons. From basic skills such as drawing points on a display to more advanced functionality such as calculating the perimeter of an arbitrary triangle, these lessons encompass the goal of incorporating computer science skills into the high school mathematics classroom. The lessons are hands-on, and so they will require student computers and teacher facilitation.

Why?

Computer Science is a growing field. According to the Huffington Post, a study showed that from 2010-2020 there is a projected 19% increase in employment in computer and mathematical occupations compared to 14.3% on average for other growing occupations. Also according to Code.org, "more than 1.4 million computer jobs will be demanded by 2020, yet only 400,000 students will go on to study computer science in college." In talking with several current college students and recent graduates, many stated that the first time they had seen any type of programming language was when they were required to take a programming class in college. Many of them stated that it was flat out scary even in a beginner's course. Many discussions have led to the conclusion that if computer science is implemented into the basics of some high school courses, many more students may feel propelled to pursue a career in computer science or at least not be so overwhelmed by it in college.

The programs that are placed throughout the lessons are designed to be used as supplementary materials to help students have a visual representation of what is taking place in the classroom. Geometry is a visual concept to that allowing the programs to compute random examples for the students will give the students a better conceptual understanding. Also, using the non-random further applications will allow students to better understand how computer/calculator programs are set up. With the technological advances in our society, it is important to help our students take advantage of developing more efficient methods of calculations. Another factor to take into account is differentiated learning. Because the world is so diverse, it is important to cater to the individual learning styles of students. Coding will give the students another outlet to help them further understand the concepts being taught in the classroom.

How?

In order to effectively implement the lessons, go through each lesson carefully to make sure the coding is fully understood before trying to implement in the classroom. Each classroom is different so it will be important to carefully analyze the content and how it relates to your specified scope and sequence. It will not be in the best interest of the student if lessons are skipped as each lesson builds on each other. If it is decided to rearrange/skip lessons, it will be important to carefully dissect what coding knowledge or geometry content will need further explanation. Each lesson is designed for teacher/student interaction. It is important to facilitate each lesson by asking probing questions to help the students be able to draw their own conclusions. Then further direction on the correctness or completion of their conclusion should be given.

Materials Needed:

- Computers for every one to two students (will be best to have one computer per student)
- Teacher computer
- Teacher Version PDF
- Student Handouts (If you cannot make copies for each student, make a class set and have students use loose leaf paper to take notes.)
- Reference Sheet
- Downloadable Haskell Files
    - k12math-student—Files the student will need
    - k12math-teacher—Answers to the Exercises and Further Applications
- Programs installed on each computer being used
    - Windows:
        - Haskell Platform
          (https://www.haskell.org/downloads)
          Choose 64bit unless your computer is over 8 years old.
        - Git (GitBash (terminal) is what you will open to run programs)
          (http://www.git-scm.com/downloads)
        - Notepad++ (Editor)
          (https://notepad-plus-plus.org/download/v6.7.9.2.html)
    - MAC
        - Haskell Platform
          http://www.haskell.org/platform/mac.html
        - Terminal is already installed on Macs.
          To access it open applications then the utilities folder. Open the terminal. Type /usr/bin/ld
          If it ask you to install developer tools, click install.
        - Atom (Editor)
          https://atom.io
          Click download. Then in the Welcome Guide you need to install a package. Click Install a package. Then in the search box type Haskell. Install package language-haskell.

How to access Haskell Files:

Windows:

1. Create a folder in Documents named Haskell
2. Download the files from the website.
3. Open the zip file.
4. Click Extract All Files.
    a. Browse
    b. Save it under Documents/Haskell
5. Extract

Mac:

1. Create a folder in Documents named Haskell
2. Download the files from the website.
3. Drag the files from downloads into the Haskell folder.
4. Then double click to unzip the file.

As you manipulate and create your own files (exercises), you will want to save them in a folder (YourName_Coding) in the Haskell/k12math-student folder on the computer. At the end of the lesson if the computer is not your own, you will want to copy your folder onto your USB drive.

When you come back to manipulate and create files again in another lesson, just copy the folder from your USB to the Haskell/k12math-student folder on the computer. Repeat this process when using a school or someone else's computer.

Getting started with the terminal:

After installing the three programs listed above, open the terminal and perform the following commands:

Teacher & Student

1. Type cd Documents/Haskell/k12math-student
2. Type util/prepare

Teacher only

3. Type cd ..
4. Type cd k12math-teacher
5. Type util/prepare

Running Code:

Windows:

1. Open GitBash
2. cd Documents/Haskell/k12math-__________ (the blank should either be student or teacher)
3. ls
   a. If running a program already installed in k12math-__________, type rungeo prog/_________ /lesson1a.hs (the blank should either be student or teacher)
   b. If running a program you manipulated and saved in your personal folder, type
      rungeo yourname_coding/lesson1a.hs

MAC:

1. Open the terminal (applications/utilities)
2. Open GitBash
3. cd Documents/Haskell/k12math-__________ (the blank should either be student or teacher)
4. ls
   a. If running a program already installed in k12math-__________, type ./rungeo prog/_________ /lesson1a.hs (the blank should either be student or teacher)
   b. If running a program you manipulated and saved in your personal folder, type
      ./rungeo yourname_coding/lesson1a.hs

Coding Overview

These lessons use a programming language called Haskell. This overview tells all you need to know about Haskell in order to follow the lessons. If you want to know more about the language, you may look at the tutorials in http://www.haskell.org/

Haskell programs are written as equations similar to algebraic equations, with some differences:

Item	Algebra	Haskell
Variables	Single letter Latin or Greek alphabet Other subscripts and superscripts are common x = α’	Name can be long Use Latin alphabet Must start with lowercase letter May only have letters, numbers, underscores (_) and primes (‘) my_variable1 = alpha’
Multiplication	No symbol between factors 2xy + 3yz	Needs * between factors 2*x*y + 3*y*z
Exponents	Denoted as a superscript	Uses symbol ^: `x^2`
Functions	Arguments enclosed in ( ) Separated by commas f(x,y) = g(x,y) - 1	Arguments follow name Separated by spaces f x y = g x y - 1

A Haskell program must always have a function named `main`, which is the starting point.

There is no syntactic difference between a function, a constant and a variable. A constant is just a function of zero arguments, and in Haskell variables and constants are just the same concept, as it is not possible to change the value of a variable after its definition.

When you introduce a new variable or function definition in Haskell, you must put the name you want to define on the left hand side of your equation, not on the right hand side. Here are some examples:

Goal	Correct	Incorrect
`Define variable x`	`x = y + z`	`y + z = x`
`Define function f`	`f x = 2*x`	`2*x = f x`
`Define a and b`	`[a,b] = p`	`p = [a,b]`
`Define p`	`p = [a,b]`	`[a,b] = p`

Auxiliary definitions

When you want to define a complicated variable or function, you can introduce auxiliary definitions that will only be valid inside that function. Auxiliary definitions are introduced by the keyword `where` and must be aligned to each other and indented to the right with respect to the main definition. Example:

```
f x = double x + square x

    where double x = x+x

          square x = x*x
```

Note that the keyword `where` is indented to the right, and then the auxiliary definitions are further indented to the right. Proper indentation like this is always necessary. The example above defines a function f that will double and square its argument and then add the two resulting values together. We could have written an equivalent definition of the same function like this:

`f x = 2*x + x^2` or also like this: `f x = (2+x)*x`

Important: Always make sure that auxiliary definitions are perfectly aligned to each other and do not use TAB when you align them. Use only the SPACE bar.

Multiple-case definitions and multiple definitions

Like in Algebra, you can define piecewise functions that behave differently for different values of the argument. Each case is introduced by a vertical bar, and different cases must be aligned to each other. You can use the keyword `otherwise` to catch all cases that you did not list explicitly. For example, here is a definition of the absolute value:

```
abs x | x > 0     = x

      | x < 0     = -x

      | otherwise = 0
```

You can also have several definitions, either single-case or multiple-case, for the same function, as long as you put together all the definitions in consecutive lines of the same file. The definitions will be searched in the order they appear in the file, and the first definition that matches the arguments is chosen for execution. For example, given the following two definitions for the function `f`

```
f 3 = 5

f x = x+1
```

the value of `(f 3)` is 5, and the value of `(f 4)` is also 5. These multiple single-case definitions could have also been done with a single multiple-case definition:

```
f x | x == 3 = 5

    | otherwise = x+1
```

Operators

An operator is just a function with a special syntax. Its name consists of special symbols instead of letters, and it is placed between its arguments instead of before the arguments. Apart from that, operators behave exactly like functions. In fact, you can place an operator before its arguments if you enclose it in parentheses:

`3 + 5` is the same as: `(+) 3 5`

Many operators are already pre-defined, but you can create your own operators too. Important pre-defined operators are:

Arithmetic operators: `+ – * /`

Comparison (relational) operators: `>   <   <=   >=   ==   /=`

List manipulation: `:` `++`

Function manipulation: `$` `.`

Booleans

A Boolean (denoted `Bool` in type annotations) is a special type with only two possible values: `True` and `False`. Booleans are used in multiple-case function definitions to express the different cases. Apart from using the words `True` and `False` literally, you generate Boolean values by using the relational operators. For example, the expression `2 > 3` is exactly the same value as the Boolean `False`. Literal Boolean values must start with an uppercase letter. You can also assign Boolean values to variables and use them in functions:

```
x = (3 == 5)   --  The value of x is False

f x = (x == x) -- The value of (f x) will always be True for any x
```

Lists

In Haskell, you can group several items into a list and use the list as a single item. All items in a list must have the same type, so, for example, you cannot mix integers and strings in the same list. You can deconstruct a list into separate variables, but the number of variables must match the length of the list. Example:

```
list1 = [2,3,5,7]   -- list1 is constructed

[a,b,c,d] = list1   -- list1 is deconstructed into separate variables
```

The previous equations will make a=2, b=3, c=5, d=7 and `list1`=[2,3,5,7]

The list `[]` denotes a list that has no elements.

The cons `(:)` operator lets you add an element to the front of a list:

```
2 : [3,4,5]  -- The result is the list [2,3,4,5]
```

Besides an operator, cons is also a special pattern that can be used to deconstruct lists:

```
x : y = [2,3,4,5]  -- This results in x=2, y=[3,4,5]
```

Two lists can be concatenated with the `(++)` operator. For example, `list1` and `list2` below are exactly the same lists.

```
list1 = [2,3] ++ [4,5]

list2 = [2,3,4,5]
```

The pre-defined function `length` returns the number of elements in a list, so, for example, the expression `length list2` will return 4.

There is no operator for adding elements at the end of a list, but you can still do it by using concatenation: `[3,4,5] ++ [6] -- The result is the list [3,4,5,6]`

Tuples

Tuples are elements of a Cartesian product. Like in algebra, tuples are enclosed in parentheses with its coordinates separated by commas. Tuples are treated as a single entity, so, for example, the function `f(x,y)` does not represent a function of two arguments but a function of a single argument which happens to be a pair. Tuples can be deconstructed into separate variables just like lists. For example, the following lines assign the values `a=17, b=31` and `t=(17,31)`

```
t = (17,31)   -- construct t from 17 and 31

(a,b) = t     -- deconstruct t into a and b
```

To define just a and b, you can also have this single line: `(a,b) = (17,31)`

The main difference between a tuple and a list is that a tuple has no corresponding cons operator or length function, and so you must know its length when you write your program. Another difference is that you can mix different types in a tuple, so, for example, `(2,"hello")` is legal.

Ranges

A range is a special type of list containing consecutive numbers. Examples:

```
[1..5]     -- Same as [1,2,3,4,5]

[2,4..10] -- Same as [2,4,6,8,10]

[0.1,0.2..1] -- Same as [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

[1..]        -- Same as the infinite list of all positive integers
```

List Comprehensions

You can also create lists by picking those elements from another list that satisfy some condition, and then applying some function to the chosen elements. For example, to create the list

```
[(2,3),(3,4),(5,6),(6,7)]
```

you can write the following list comprehension:

```
[(x,x+1) | x <- [2..6], x /= 4 ]
```

You can read that as: the list of pairs (x,x+1) such that x is taken from the list [2..6] and x is not 4.

Strings

In addition to numbers, you can manipulate other objects such as literal character strings, which you can use, for example, to print messages in your program. String literals are enclosed in double quotes, but single character literals are enclosed in single quotes. A string literal is just a list of single character literals. For example, greeting1 and greeting2 below are the same string:

```
greeting1 = “hello”

greeting2 = [‘h’, ’e’, ’l’, ’l’, ’o’]
```

Strings can be concatenated with the `(++)` operator. For example, `text1` and `text2` below are the same exact text:

```
text1 = “Once upon “ ++ “a time”
```

```
text2 = "Once upon a time"
```

Comments

Comments are text included in a program but intended for humans, not for the computer. To make the computer ignore a line with comments, start it with two consecutive dashes: `--`

```
-- This is a comment line and will not be executed
```

Comments are sometimes used as a quick and dirty way to temporarily disable some parts of the code. This is an useful practice while you are writing your code, but you should remove commented out code from your final program. On the other hand, you should have enough comments (text, not code) in your final program to explain anything that would not be immediately obvious to a reader of your code.

Multi-line comments start when the symbol sequence `{-` is encountered and end when the symbol sequence `-}` occurs. If the comment contains the `{-` symbol several times inside, then the comment will not end until each opening symbol is *closed* with a corresponding `-}` symbol.

```
{-

This is a 3-line comment {- this is a nested comment -}

-}
```

You can also have lines with both code and comments:

```
x = {- surprise! -} 2*y+1  -- This line defines x in terms of y
```

The compiler will read the line above as if it only had this code: `x = 2*y+1`

Type Annotations

Type annotations for functions allow us to explain succinctly what type of input arguments a function has and what type of output it returns. The types are separated by arrows, where the output type is the last type listed and all the others correspond to the input types.

For example, the following type annotation denotes that function `f` is a function of two arguments, where the first argument is an integer and the second argument is a point. The function returns a list of points that it creates from those arguments:

```
f :: Integer -> Point -> [Point]
```

Question: How would you create a list of points when given an integer number and a single point as inputs?

Type annotations are not executable code, as they only contain information about what the data is, not about how to compute it. However, unlike comments, the Haskell compiler reads the type annotations and uses the information in them to generate better executable code. Therefore, type annotations are intended for both humans and computers.

Boilerplate

All the programs you will use follow a common pattern and need some particular lines of code at the beginning of each file. These necessary but not very interesting parts of the program are commonly referred to as *boilerplate.* You should just copy the boilerplate into each new file you make and ignore it otherwise, as it is not worth wasting time on it. Our boilerplate code is as follows:

```
import Geometry

import Drawing

main = drawPicture myPicture

myPicture points = <actual program goes here>
```

The actual program will be written as auxiliary definitions inside the function `myPicture`

Eliminating parentheses

You can reduce the number of parentheses in your expressions by replacing the starting parenthesis with a dollar sign. For example, you can write:

```
my_function1 (expr a b c)    or: my_function1 $ expr a b c

f (g (h x y))                or: f $ g $ h x y

f (g (h x) y)                or: f $ g (h x) y
```

The last example shows that it is not always possible to eliminate all parentheses, but using this construction makes expressions more readable in general.

The composition operator `(.)` is also used to reduce the number of parentheses. It is defined as:

```
(f . g) x = f (g x)
```

You can use the composition operator to write expressions such as `f $ g $ h $ x` in an equivalent but more readable form: `f.g.h $ x`

Remember that those expressions execute *backwards,* as they instruct the computer to apply the function `h` first, then `g` and finally `f`, which in algebra is written as: `f(g(h(x)))`

Software Tools for Coding

Compiler and Libraries: The Haskell Platform package contains the Haskell compiler and common system libraries. A compiler is a program that translates the code you write into executable machine instructions. The compiler included in the Haskell Platform is called GHC. A library is a collection of pre-defined functions that makes it easier to write programs without having to reinvent the wheel each time.

Editor: An editor is a program that lets you write your code and save it to a file. It usually has features such as syntax highlighting, which marks keywords and other syntax markers in different colors. You can choose among many different editors which one to use, and in these lessons we chose the following:

In Windows, we use Notepad++

In MACs, we use Atom

Terminal: A terminal is a window where you can give instructions to the computer directly without going through menus or intermediate apps. Terminals are commonly used by programmers, system administrators and other computing professionals because they allow them to interact with the inner parts of the computer that are usually hidden to the end users. Some systems, like MACs, come with a terminal pre-installed. In other systems, such as Windows, you need to install a third party terminal program such as Git Bash.

Understanding and fixing compiler errors

When you make mistakes in your code, the compiler will produce an error message. Sometimes, error messages are long, and they may be confusing to inexperienced programmers. If you follow the advice here, you will be able to understand and fix the errors faster.

1. Do not despair. Be willing to spend the time necessary to fix the error.
2. Read the error message entirely, even if you do not understand any of it. You will on time.
3. Go to the first error shown in your message. There is usually more than one error.
4. Look at the first line in your message. It shows the name of the file where the error was found followed by two numbers: the line and the column in that file.
5. Look at the file, line and column where the error occurred.
6. Check that the indentation is correct. This is the number 1 cause of errors.
7. Check for typos. This is number 2 cause of errors.
8. Check for indentation and typos again. This is number 3 cause of errors.
9. If the message mentions "parse error", then the problem is most likely wrong indentation or typos. Check lines above the error line, as sometimes the error is before that line.
10. If the message mentions "No instance of" or "Couldn't match type", then the problem is that you are using the wrong value in some of the variables mentioned in the error message. Try to understand the message, as it is giving you the information you need to fix the error.
11. The probability of the compiler being wrong is extremely small, so if it says you have an error, look for it until you find it. People make mistakes often. Computers do not.
12. Add type annotations to the functions where you suspect the error is. Type annotations will help the compiler produce error messages that are easier to understand.

There is no magic in computers. Everything happens for a logical reason. Computers are not fickle or unpredictable. All errors in computers are due to some human who was not careful enough when writing the code. The only way to avoid computer errors is to be careful when you write code, and that includes being patient when reading error messages and methodical when fixing errors. There is a reason why this process is popularly known as *debugging*: it is tedious but necessary.

Understanding runtime errors (incomplete definitions)

Haskell programs are safe from most runtime errors, but programs can still crash if you wrote an incomplete definition somewhere in your code. An incomplete definition is a function definition that does not consider all possible values its arguments can have. A multiple-case definition without an `otherwise` clause or definitions that expect the values to always be within a few special cases are typical cases of incomplete functions, as are functions that do not check that divisors are not zero before performing division. Check your code for incomplete definitions if you get a runtime error. In short scripts like ours, it may be OK to let the program crash instead of handling the error more gracefully, but you will still need to understand the cause of the crash.

Function Reference List – Drawing Functions

Drawing functions prepare the graphical elements of a picture to be shown on the screen. Each drawing function creates a single picture, so if you want to show several objects at once you must create a *composite picture* that combines each separate picture into a single object. The `(&)` operator performs this composition. For example, if `drawObject1` and `drawObject2` are two functions that create a picture of an object1 and some other object2 respectively, then to show both objects at once in a single picture you would write: `drawObject1 & drawObject2`

In summary: In order to create a single picture with all your objects, you must put an ampersand `(&)` between each drawing command.

Even though we do not use coordinates explicitly in many of the lessons, the computer always uses coordinates internally. The coordinates are set up so that the main window shows the region between -10 and 10 in both the X and the Y axis. A point is internally stored as a pair of numbers.

```
coordinates :: Picture
```

```
coordinates
```

Draws a coordinate frame consisting of the X axis, the Y axis and guidelines spaced by 1 unit.

```
coordinates’ :: Number -> Picture
```

```
coordinates’ step
```

Draws a coordinate frame like `coordinates`, except that the distance between consecutive guidelines is given by the parameter `step`

```
drawArc :: (Point,Point,Point) -> Picture
```

```
drawArc (a,o,b)
```

Draws a circular arc that starts at point `a`, has its vertex at `o`, and ends when the circle intersects the ray `(o,b)`

```
drawCircle :: (Point,Point) -> Picture
```

```
drawCircle (c,x)
```

Draws a circle with center at `c` and passing through `x`

```
drawLabel :: Point -> String -> Picture
```

```
drawLabel p lbl
```

Shows the text `lbl` at a location close to `p`

```
drawLabels :: [Point] -> [String] -> Picture
```

```
drawLabels pts lbls
```

Takes each text in the list `lbls` and successively shows it at the locations given in the list `pts`. If the two lists are not the same length, the last elements in the longer list will be ignored.

```
drawLine :: (Point,Point) -> Picture
```

```
drawLine (p1,p2)
```

Draws the line passing through `p1` and `p2`, or draws nothing if the two points are too close to each other.

```
drawPoint :: Point -> Picture
```

```
drawPoint p
```

Draws a dot at position `p`

```
drawPointLabel :: Point -> String -> Picture
```

```
drawPointLabel p l
```

Draws a dot at position `p` and shows the text `l` next to it.

```
drawPoints :: [Point] -> Picture
```

```
drawPoints pts
```

Draws each point in the list `pts` using the function `drawPoint` and combines all the drawings in a single picture as if the `(&)` operator was used between them.

```
drawPointsLabels :: [Point] -> [String] -> Picture
```

```
drawPointsLabels pts lbls
```

Draws each label in `lbls` at each point location in `pts` using `drawPointLabel` repeatedly. Look at `drawLabels` and `drawPoints` for further explanation.

```
drawSegment :: (Point,Point) -> Picture
```

```
drawSegment (p1,p2)
```

Draws the segment between `p1` and `p2`. Draws nothing if the two points are the same.

```
drawTriangle :: (Point,Point,Point) -> Picture
drawTriangle (a,b,c)
```

Draws the points `a,b` and `c` and the segments `(a,b),(b,c)` and `(c,a)`. It combines all drawings in a single picture. Look at `drawPoints` and `drawSegment` for more information.

```
message :: String -> Picture
message text
```

Draws `text` in a single line starting at the lower left corner of the main window.

```
messages :: [String] -> Picture
messages lines
```

Draws each element of the list `lines` in a single line, so that the last line starts at the lower left corner of the main window.

```
myPicture :: [Point] -> Picture
myPicture points
```

This function must be defined inside our program to create whatever composite picture we want to show. The argument `points` is an infinite list of random points that the system provides us to use in our drawings. We extract a finite amount of points from this infinite list using the system function `take`. We can then use geometric functions to create interesting geometric objects based on these random points, and then use the previously listed drawing functions to create pictures of those geometric objects. So write this function as if composing a symphony: you must carefully combine all the elements together to create a harmonious result.

```
(&) :: Picture -> Picture -> Picture
composite = pic1 & pic2
```

Combines two pictures `pic1` and `pic2` into a single picture `composite`. Please refer to the beginning of this section for more information on this operator.

Function Reference List – Geometry Functions

Geometry functions are the most interesting functions for us. Using the geometry functions, we can create rich and complex geometric objects and apply geometric transformations to them.

Some of the functions below depend on the **Betweeness relationship**, which is defined as follows:

> Given 2 points A and B, we can divide the plane into 3 regions by finding perpendicular lines to segment AB passing through A and B respectively. A point X is said to be between A and B if it lies in the middle region.
>
> The three regions will called "beyond (A,B)", "between (A,B)" and "beyond (B,A)".

```
beyond :: (Point,Point) -> Point -> Bool

yes_no = beyond (a,b) x
```

Given points `a`, `b` and `x`, the value of Boolean `yes_no` will be `True` whenever point `x` lies in the region `beyond (a,b)` as explained above. Otherwise, the value of `yes_no` will be `False`.

```
circle_circle :: (Point,Point) -> (Point,Point) -> [Point]

intersects = circle_circle (o1,x1) (o2,x2)
```

Given a circle with center `o1` and point `x1` on its circumference and another circle with center `o2` and point `x2` on its circumference, this function creates the list `intersects` consisting of all the points that lie on both circumferences. Note that `intersects` may be an empty list if the circle do not cross, it may contain a single element if the circles are tangent and it will contain two elements otherwise.

```
dilate :: Point -> (Number,Number) -> Point

p' = dilate p (x_dilation,y_dilation)
```

Performs a dilation of point `p` with respect to the origin. The coordinates will be dilated by `x_dilation` horizontally and by `y_dilation` vertically. The point `p'` corresponds to the dilated point.

```
dist :: Point -> Point -> Number

d = dist p q
```

Given two points `p` and `q`, the number `d` will be the Euclidean distance between them.

```
find_apart :: Point -> (Point,Point) -> [Point]
```

```
y = find_apart x pts
```

The point `y` represents the first point in the list of points `pts` that is apart enough from the given point `x` to be clearly distinguishable from it. If no such point exists, this function will fail, making your program crash and produce an error message. Apart enough by default is 0.01 units.

```
line_circle :: (Point,Point) -> (Point,Point) -> [Point]
intersects = line_circle (a,b) (o,x)
```

Given a line passing through points `a` and `b` and a circle with center at point `o` and passing through point `x`, this function creates the list `intersects` consisting of all points that lie on both the line and the circle. Note that this list may contain either 0, 1 or 2 elements.

```
line_line :: (Point,Point) -> (Point,Point) -> [Point]
intersects = line_line (a,b) (c,d)
```

Given a line passing through points `a` and `b` and another line passing through points `c` and `d`, this function creates the list `intersects` consisting of either no point (if the lines are parallel) or the point at the intersection of both lines (if they cross each other.) Note that this function considers that a line is always parallel to itself and returns no point in that case.

```
midpoint :: Point -> Point -> Point
m = midpoint a b
```

The point `m` is the midpoint of the segment joining points `a` and `b`

```
mirror :: (Point,Point) -> Point -> Point
p' = mirror (a,b) p
```

Given a point `p` and a line passing through points `a` and `b`, the point `p'` is the reflection of `p` with respect to line `(a,b)`

```
parallel :: (Point,Point) -> Point -> (Point,Point)
(a',b') = parallel (a,b) c
```

Points `a'` and `b'` lie on a line passing through point `c` that is parallel to line `(a,b)`

```
perpendicular :: (Point,Point) -> (Point,Point)
(c,d) = perpendicular (a,b)
```

Given a line passing through points `a` and `b`, the points `c` and `d` lie on the perpendicular to line to line `(a,b)` that passes through point `b`

```
projection :: (Point,Point) -> Point -> Point
x′ = projection (a,b) x
```

The point `x′` is at the intersection of line `(a,b)` with the perpendicular line passing through `x`

```
quadrilateral :: [Point] -> (Point,Point,Point,Point)
(a′,b′,c′,d′) = quadrilateral [a,b,c,d]
```

The points `a′,b′,c′` and `d′` are a reordering of points `a,b,c` and `d` in such a way that the segments between consecutive points do not cross each other. The resulting figure is a proper quadrilateral.

```
rotate :: Number -> Point -> Point
p′ = rotate angle center p
```

The point `p′` is obtained by rotating point `p` counterclockwise `angle` degrees around `center`

```
translate :: (Number,Number) -> Point -> Point
p′ = translate (vx,vy) p
```

The point `p′` is computed by moving point `p` horizontally by `vx` units and vertically by `vy` units.

Function Reference List – System Functions

System functions are used to manipulate the data stored in the computer. Many system functions are pre-defined in a system library, but it is also possible to define new system functions that combine the existing ones into higher-level actions. Some system functions are just convenience functions to automate common tasks, but other are essential to the operation of the computer, which would not work without them. Thus, many system functions are difficult to understand and look like *magic*, but there is no magic involved. System functions are functions just like any other function, but they operate on the lower level components of a computer, which are usually hidden to the end users.

`main` is a special system function. It is not pre-defined, but it must be defined by the programmer. When the program is executed, the computer looks at the contents of `main` and starts executing the instructions defined there, so `main` is the entry point for any program. In our lessons, however, `main` must always be defined as follows: `main = drawPicture myPicture`

Both `drawPicture` and `myPicture` are functions too. The function `drawPicture` is pre-defined in our library of drawing functions. It uses `myPicture` to create whatever picture we want and converts that picture into screen pixels using several auxiliary system functions. The function `myPicture` is our *de facto* entry point. This function should create a composite picture containing all the objects we want to show on the screen. Look in the Drawing Functions section for more information on this function.

Note: In the functions below, you can replace `[Point]` with any other type of element, and most functions will still work the same.

```
drop :: Integer -> [Point] -> [Point]

new_points = drop n points
```

Creates the list `new_points,` which contains the same elements as the list `points,` except that the first `n` points of `points` are deleted and do not appear in `new_points`

```
find :: (Point -> Bool) -> [Point] -> Maybe Point

case find predicate points of

     Just p -> message $ "Found point " ++ showpoint p

     Nothing -> message "No point found"
```

The first argument of this function is another function that classifies points as acceptable (`True`) or not acceptable (`False`). Functions like that, whose return type is Boolean, are called *predicates.* So, the function `find` takes a `predicate` and a list `points` and searches the list for the first point that satisfies `predicate` (i.e., the first point `p` such that `predicate p` is `True`). The output of this function is `Just p` in case such point is found. Otherwise, the output is `Nothing`.

The example above shows how you can check for each case by enclosing the function within the special `case … of` construct. The syntax for checking each case is similar to multiple-case function definitions, except that instead of using = to separate the case and the action, we use the symbol `->` instead.

In summary, the output of this function may be one of the two patterns `Just p` or `Nothing`. Values of this type generalize Booleans, as `Nothing` is just like `False` and `Just p` is like `True` with some extra information attached. Instead of `Bool`, this type is called `Maybe Point`.

```
length :: [Point] -> Integer
```

```
l = length lst
```

Computes the number of elements in the list `lst`

```
not :: Bool -> Bool
```

```
opposite_condition = not original_condition
```

Negates the `original_condition`, so that if it was `True`, then `opposite_condition` would be `False`. Similarly, if `original_condition` was `False`, then `opposite_condition` would be `True`.

```
show :: Number -> String
```

```
text = show num
```

Converts the number `num` into its textual representation. For example, if `num` was 8.5, then `text` would be the string `"8.5"`

```
shownum :: Number -> String
```

```
text = shownum num
```

Performs the same conversion as `show`, but it rounds up the result so that it shows fewer decimals. For example, `shownum 3.579999996` would be converted into `"3.58"`. Note, however, that due to some low-level interactions, this function is not always able to round up the result properly, but in general it shows numeric results more concisely than `show`

```
take :: Integer -> [Point] -> [Point]
```

```
extracted_points = take n points
```

Creates the list `extracted_points` by taking the first `n` elements of the list `points` and copying them into a new list. Note that the original list `points` is not modified in this process.

```
(++) :: [String] -> [String] -> [String]
```

```
combined = string1 ++ string2
```

Creates a new list `combined` which contains the concatenation of a copy of `string1` with a copy of `string2`. Note that the original strings are not modified. This operator actually works on any list, not just on strings.

```
(.) :: Function -> Function -> Function
```

```
h = f . g
```

The function composition operator takes two functions `f` and `g` and creates a new function `h` so that for any argument `x` we will have: `h x = f (g x)`

Note that in algebra, you would write that equation as: `h(x) = f(g(x))`

Actually, the definition of this operator in the system library consists of just the following line:

```
(f . g) x = f (g x)
```

Reference Sheet

List of functions and coding concepts introduced in each lesson

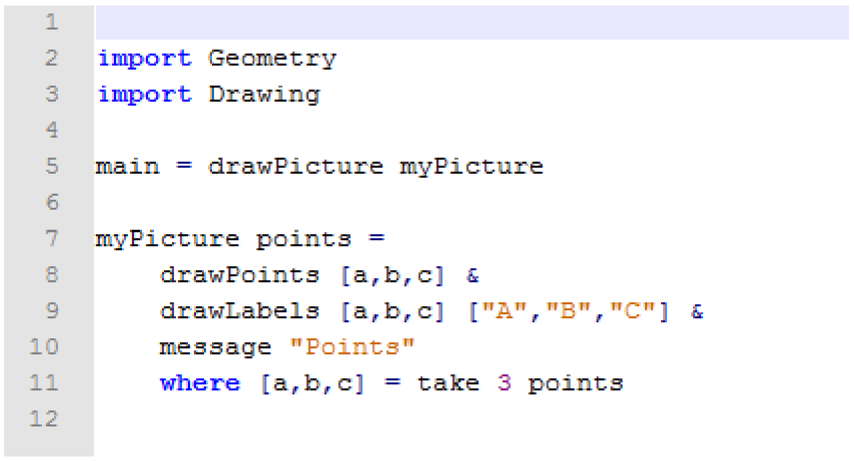
Each coding concept mentioned below is described in a section with the same name in the Coding Overview. Drawing, Geometry and System functions are described under the corresponding section in the Function Reference List, which is also part of the coding overview.

Lesson	Drawing Functions	Geometry Functions	System Functions	Coding Concepts
Lesson 1	drawPoints drawLabels drawSegment drawLine message myPicture &		main take drawPicture	Boilerplate Lists Strings Auxiliary definitions Tuples
Lesson 2	drawCircle	find_apart circle_circle line_circle line_line	drop	Understanding runtime errors (incomplete definitions)
Lesson 3	coordinates messages	dist midpoint	show ++	
Lesson 4	drawArc drawPointLabel drawTriangle	beyond	find not shownum .	Multiple-case definitions
Lesson 5		parallel perpendicular projection		
Lesson 6	coordinates’	quadrilateral		
Lesson 7		mirror		
Lesson 8		dilate rotate translate		

Lesson 1: Points, Lines, Segments

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 1 Part 1: Open the editor and then open lesson1a.hs. Look at the code and write down what you understand in the code. Run the code.



```
1
2   import Geometry
3   import Drawing
4
5   main = drawPicture myPicture
6
7   myPicture points =
8       drawPoints [a,b,c] &
9       drawLabels [a,b,c] ["A","B","C"] &
10      message "Points"
11      where [a,b,c] = take 3 points
12
```

The teacher should refer to the Reference Sheet for the list of concepts and functions used in this lesson and then briefly describe the following characteristics of the code:

Lines 1-7 are boilerplate (Boilerplate section in the Coding Overview)

Lines 8-11 are the actual program

The operator `&` is described at the beginning of the section Function Reference List – Drawing Functions in the Coding Overview.

The program illustrated here consists of a single equation that defines the function `myPicture` as a composite picture which includes 3 graphical elements:

- A picture with points A, B and C
- A picture with their labels
- A message at the bottom that reads “Points”

These 3 partial pictures are combined with the operator `&` to create the final picture.

Line 11 is an auxiliary definition of A,B and C in terms of `points`. The list `points` is an infinite list of random points, from which 3 points are taken. Therefore, `take 3 points` is a list consisting of just the first 3 points from the infinite list. This list is deconstructed into 3 separate variables.

Questions:

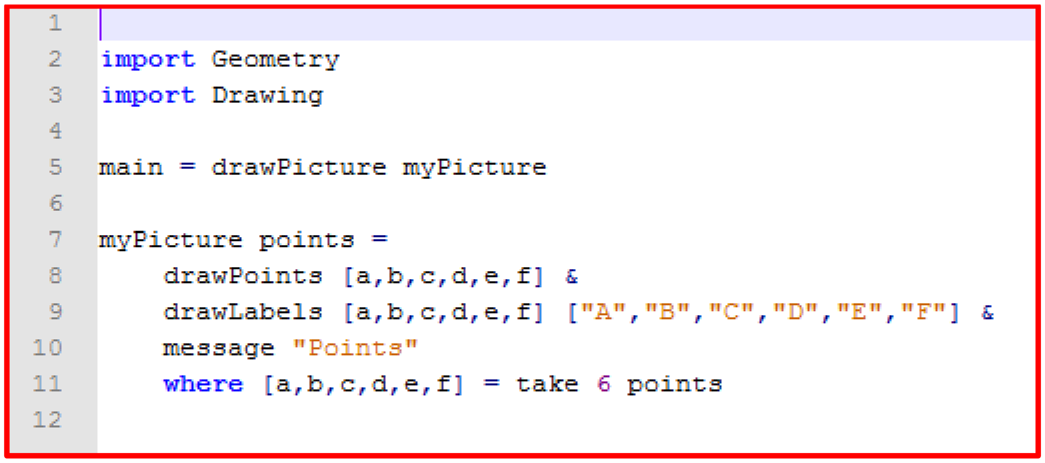
1. What function draws the points?
   drawPoints

2. What function draws the labels?
   drawLabels

Exercise: *Draw and Label three additional points in the program.*

Save the program as yourname_lesson1b.hs

Run the program to check.



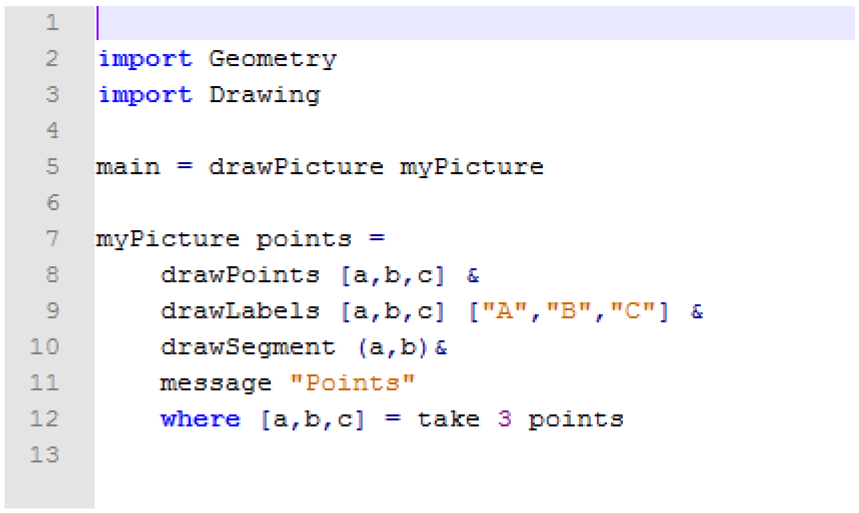
```

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d,e,f] &
    drawLabels [a,b,c,d,e,f] ["A","B","C","D","E","F"] &
    message "Points"
    where [a,b,c,d,e,f] = take 6 points

```

Questions:

How do you think we would draw a segment between points A and B?

Answers may vary: drawSegment (a,b)

Lesson 1 Part 2: Open the editor and then open lesson1c.hs.



```

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawSegment (a,b)&
    message "Points"
    where [a,b,c] = take 3 points

```

Exercise: *Determine what the program will draw.*

Run the program to check.

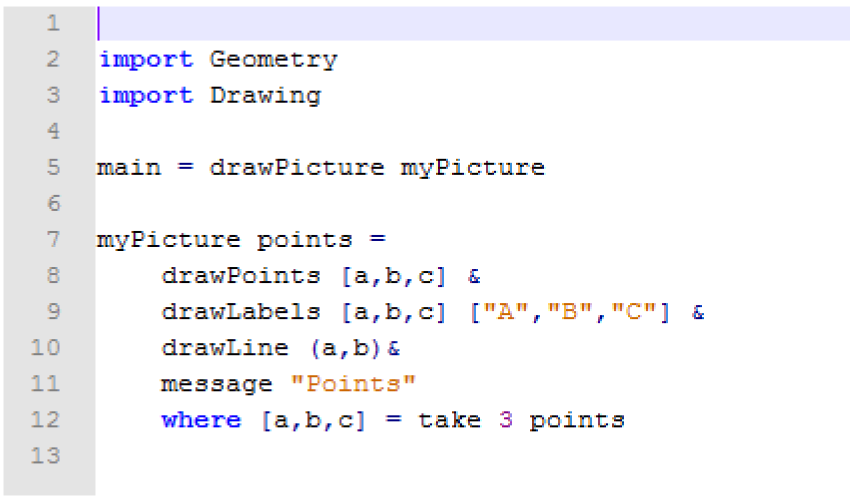
Note that the argument of `drawPoints` is a list of points, but the argument of `drawSegment` is a tuple (pair) of points.

Questions:

What function do you think we would use to draw lines?

Answers may vary: drawLine (a,b)

Lesson 1 Part 3: Open the editor and then open lesson1d.hs.



```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawLine (a,b)&
    message "Points"
    where [a,b,c] = take 3 points

```

Exercise: *Determine what the program will draw.*

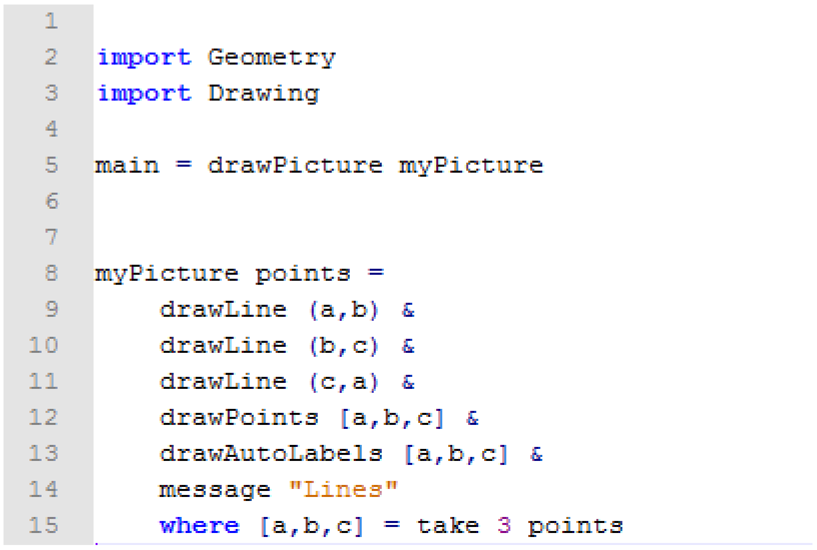
Run the program to check.

Lesson 1 Ending Exercises:

Exercise: *Open lesson1e.hs*

Sketch a drawing of what you think is happening in this program.

Run the program to check your answer.



```haskell

import Geometry
import Drawing

main = drawPicture myPicture


myPicture points =
    drawLine (a,b) &
    drawLine (b,c) &
    drawLine (c,a) &
    drawPoints [a,b,c] &
    drawAutoLabels [a,b,c] &
    message "Lines"
    where [a,b,c] = take 3 points
```

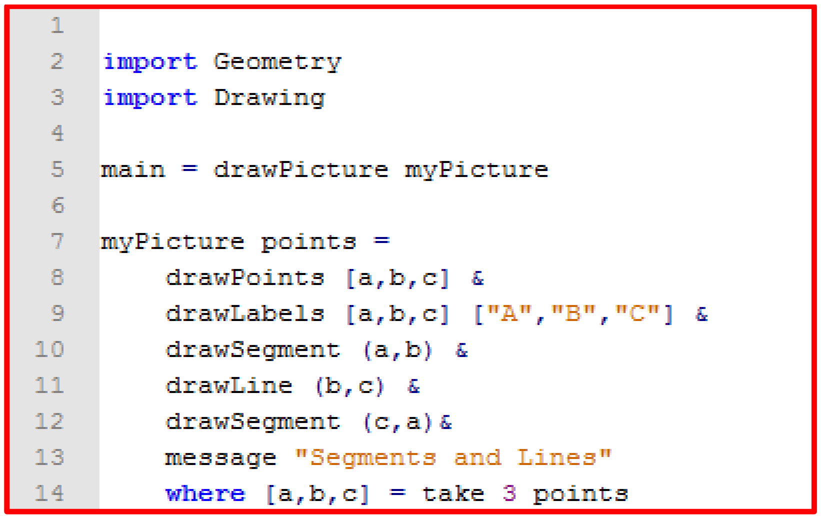
Exercise: *Open lesson1f.hs.*

Complete the program to draw the following objects:

1. *Segment AB*
2. *Line BC*
3. *Segment CA*
4. *Message needs to read "Segments and Lines"*

Save the program as yourname_lesson1f.hs.

Run the Program to check your answer.

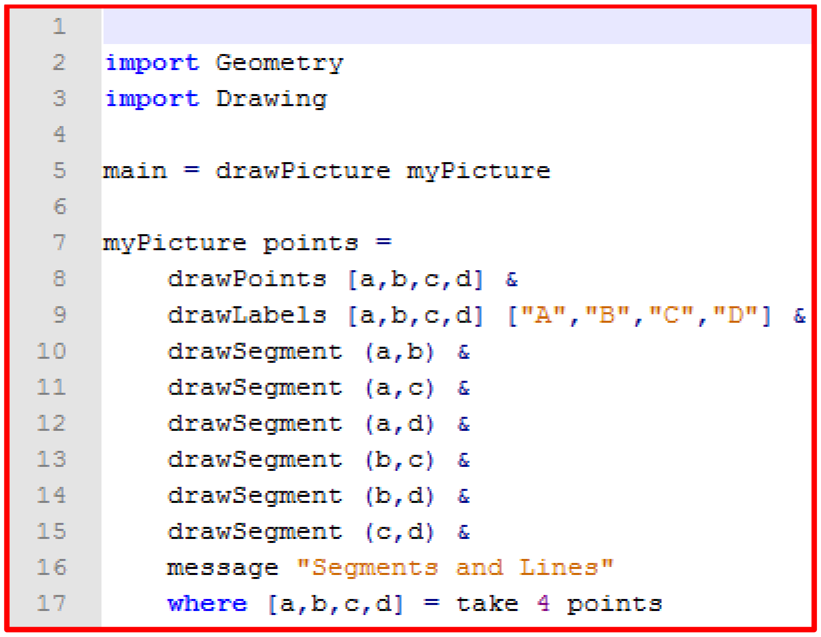


```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawSegment (a,b) &
    drawLine (b,c) &
    drawSegment (c,a)&
    message "Segments and Lines"
    where [a,b,c] = take 3 points
```

Exercise: *Using four random points, how many segments would you need to connect each point with the other three points on the graph?* 6

Manipulate one of the programs above to check your answer.

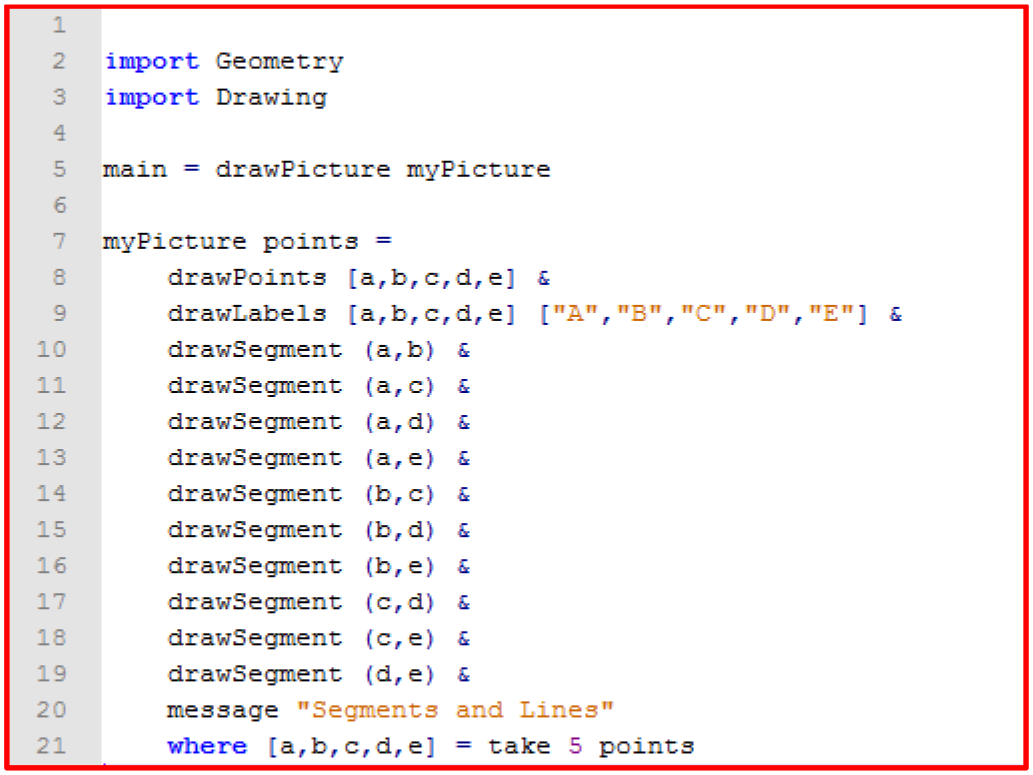
Save the program as yourname_lesson1g.hs



```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d] &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawSegment (a,b) &
    drawSegment (a,c) &
    drawSegment (a,d) &
    drawSegment (b,c) &
    drawSegment (b,d) &
    drawSegment (c,d) &
    message "Segments and Lines"
    where [a,b,c,d] = take 4 points
```

Exercise: *Repeat the above exercise for 5 random points.* *10*

Save the program as yourname_lesson1h.hs

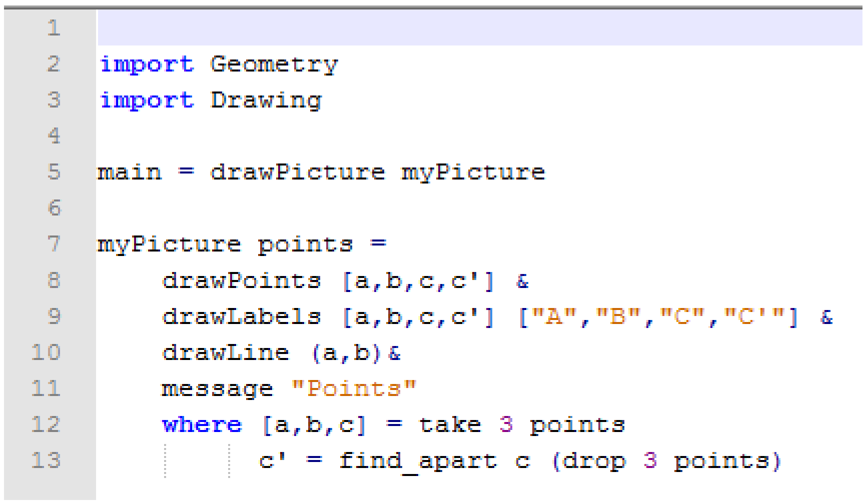


```

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d,e] &
    drawLabels [a,b,c,d,e] ["A","B","C","D","E"] &
    drawSegment (a,b) &
    drawSegment (a,c) &
    drawSegment (a,d) &
    drawSegment (a,e) &
    drawSegment (b,c) &
    drawSegment (b,d) &
    drawSegment (b,e) &
    drawSegment (c,d) &
    drawSegment (c,e) &
    drawSegment (d,e) &
    message "Segments and Lines"
    where [a,b,c,d,e] = take 5 points
```

Lesson 2: Intersections

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 2 Part 1: Open the editor and then open lesson2a.hs Look at the code and write down what you understand in the code. Run the code.



```
 1
 2  import Geometry
 3  import Drawing
 4
 5  main = drawPicture myPicture
 6
 7  myPicture points =
 8      drawPoints [a,b,c,c'] &
 9      drawLabels [a,b,c,c'] ["A","B","C","C'"] &
10      drawLine  (a,b) &
11      message "Points"
12      where [a,b,c] = take 3 points
13            c' = find_apart c (drop 3 points)
```

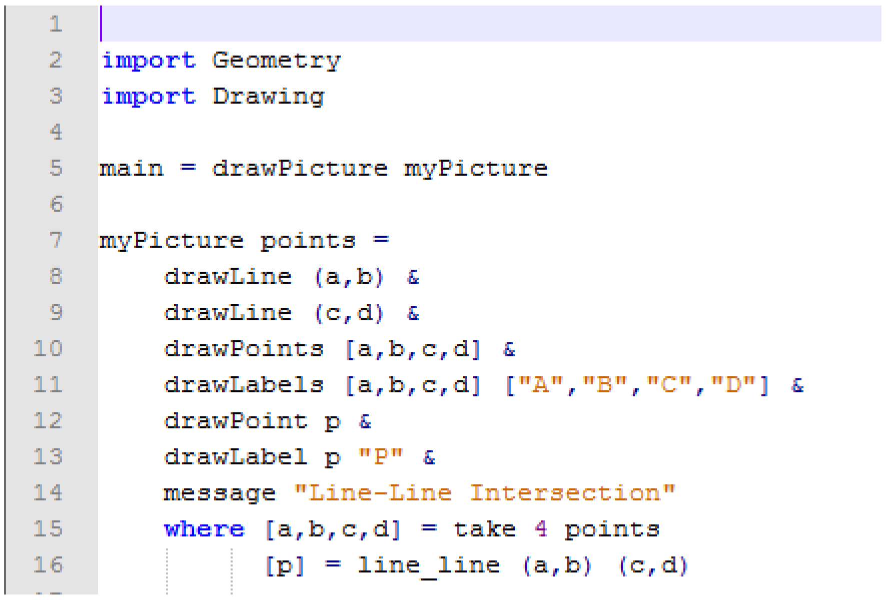
Teachers need to discuss that this program has 2 auxiliary definitions in lines 12 and 13, respectively. Line 12 is similar to previous definitions, but line 13 has new concepts. The function `drop` is kind of the opposite of `take`, as `drop 3 points` is an infinite list with the first 3 points of list `points` removed. In other words, it drops `[a,b,c]` from `points`. The function `find_apart` will scan that list to find the first point in the list that is `apart` from point `c`.

There is a subtle difference between two points being `apart` and two points being *different.* When two points are too close to each other, we may or may not be able to write a proof that establishes whether or not they are the same point. The reason is that the machine can only approximate the points to a finite precision, so we cannot know the exact position of the points. On the other hand, if the points are sufficiently far apart from each other, then we can actually prove that they are not the same point. How `apart` two points must be in order to be distinguishable from each other depends not only on the precision of the machine running our code but also on our tolerance to errors. These lessons use a threshold of 0.01 units to decide apartness.

Questions:

1. What does the function `find_apart` do?
   Gives you a point from a list that is separate from a specified point.
2. What does the `drop 3 points` do?
   Consist of the list of points without the first three points.

Lesson 2 Part 2: Open the editor and the open lesson2b.hs. Look at the code and then run the program in the terminal.



```
 1
 2  import Geometry
 3  import Drawing
 4
 5  main = drawPicture myPicture
 6
 7  myPicture points =
 8      drawLine (a,b) &
 9      drawLine (c,d) &
10      drawPoints [a,b,c,d] &
11      drawLabels [a,b,c,d] ["A","B","C","D"] &
12      drawPoint p &
13      drawLabel p "P" &
14      message "Line-Line Intersection"
15      where [a,b,c,d] = take 4 points
16            [p] = line_line (a,b) (c,d)
```

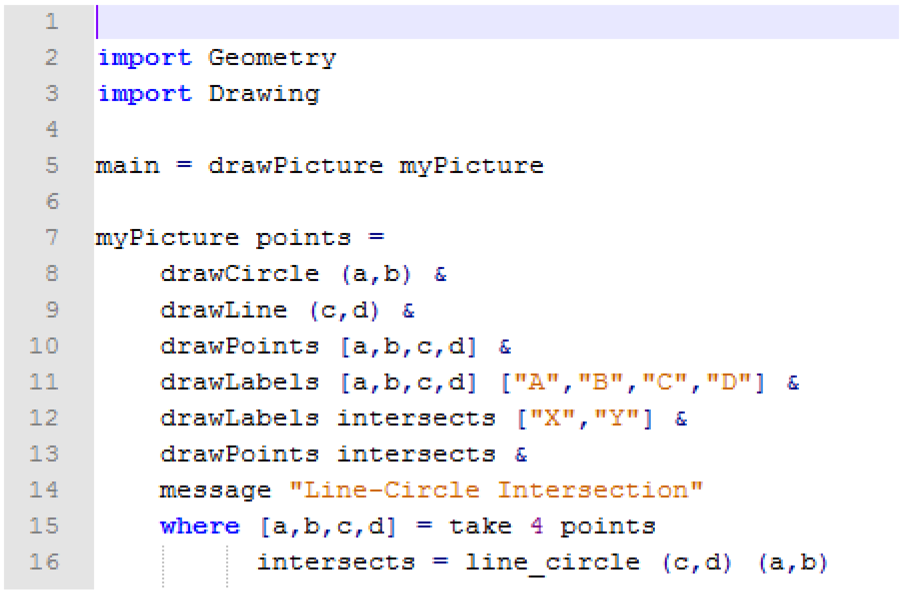
Questions:

1. What does the program do?
   Draws two lines and their intersection point
2. What does p represent?
   The intersection point
3. What does the function `line_line (a,b) (c,d)` used for?
   Finds where the two lines intersect.
4. How many points could the function `line_line (a,b) (c,d)` return?
   One point of intersection
   Zero points of intersection if the lines are parallel.

Line 16 is an incomplete definition. Look at the section named "Understanding runtime errors" in the Coding Overview. If the lines AB and CD are parallel, this program will crash because it (wrongly) expects the two lines to have always one point in common. However, the probability of that case is so low that we do not want to add complexity to the program to handle it.

Nevertheless, it is important to remark the tradeoff we are doing in terms of correctness versus simplicity.

Lesson 2 Part 3: Open lesson2c.hs. Look at the code carefully.



```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawCircle (a,b) &
    drawLine (c,d) &
    drawPoints [a,b,c,d] &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawLabels intersects ["X","Y"] &
    drawPoints intersects &
    message "Line-Circle Intersection"
    where [a,b,c,d] = take 4 points
          intersects = line_circle (c,d) (a,b)
```

Questions:

1. What do you think `intersects = line_circle (c,d) (a,b)` will compute?
   The intersection point(s) of a line and circle.

   Run the program and compare your answers to these questions.

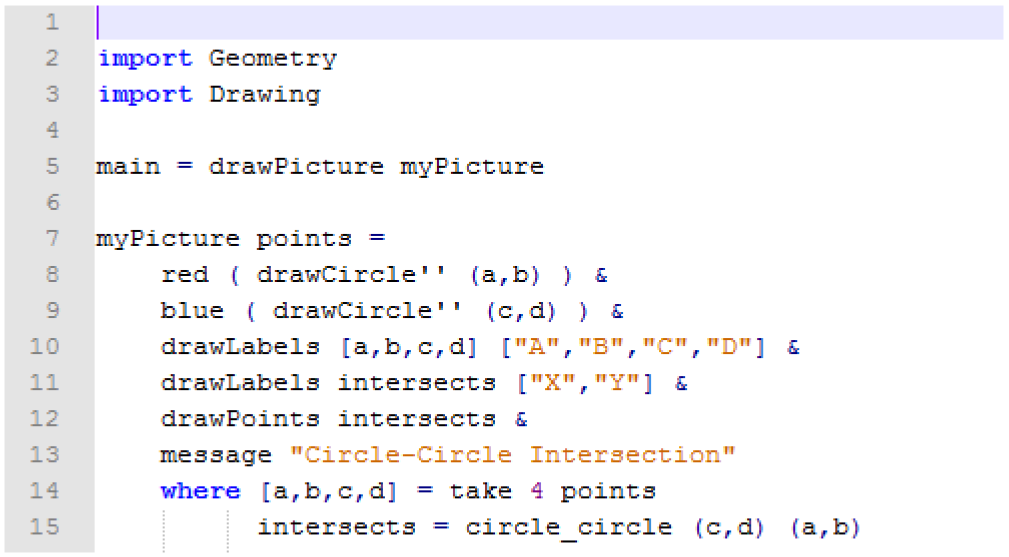
2. What do the letters correspond to in `drawCircle (a,b)`?
   Circle with center at A and B on the circle.
3. What does (c,d) correspond to in `line_circle (c,d) (a,b)`?
   Line CD.

Note to teachers: It may be beneficial to discuss the variable `intersects`. The reason we must assign a variable `intersects` in the code is because there are three different possibilities of how this program can run: 1. Two intersection points, 2. One intersection point, 3. No intersection points. We cannot specify that the intersection will only be one point like we did in demo03lineline.hs.

Lesson 2 Part 4: Open lesson2d.hs.

Exercise: *Determine what the program will draw.*

Run the program to check.



```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    red ( drawCircle'' (a,b) ) &
    blue ( drawCircle'' (c,d) ) &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawLabels intersects ["X","Y"] &
    drawPoints intersects &
    message "Circle-Circle Intersection"
    where [a,b,c,d] = take 4 points
          intersects = circle_circle (c,d) (a,b)
```

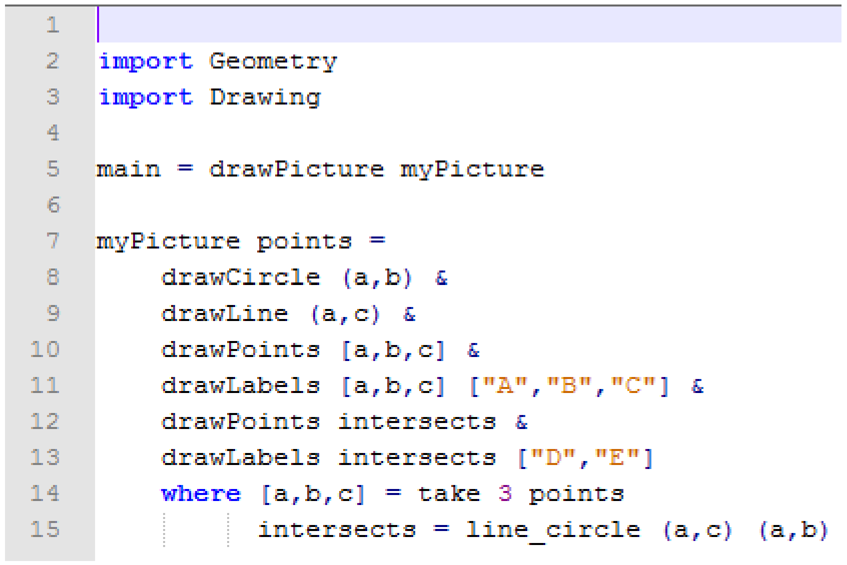
Lesson 2 Ending Exercise:

Exercise: *Open lesson2e.hs. Remove the word* `undefined` *to get started. Create a program to draw the following:*

1. *Circle with center a and point b on the circle.*
2. *Line AC*
3. *Label the intersection points of the circle and line as D and E.*

Save the program as yourname_lesson2e.hs

Run the program to check.



```haskell

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    drawCircle (a,b) &
    drawLine (a,c) &
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints intersects &
    drawLabels intersects ["D","E"]
    where [a,b,c] = take 3 points
          intersects = line_circle (a,c) (a,b)
```

Lesson 2 Further Applications:

Teachers: you can easily extend this lesson by removing the randomness from the program. The program lesson2d.hs has three possible outcomes.

1. *The circles intersect two times.*
2. *The circles intersect one time. (Tangent Circles)*
3. *The circles do not intersect.*

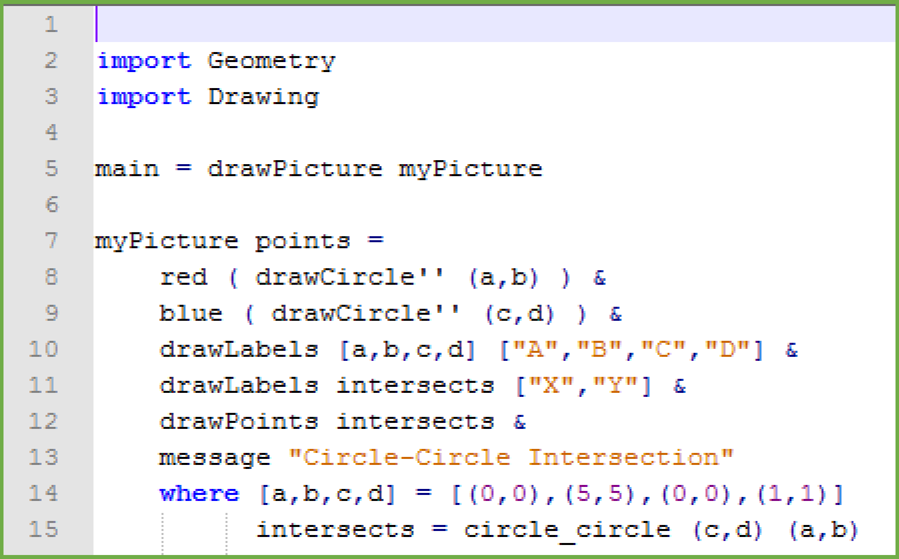
By removing the randomness such as in lesson2dN.hs, you can demonstrate all three of these situations.

You can also have students use this as a tool to discuss whether two circles do not intersect or how many times they intersect. It would be a good tool for checking answers in class or at home.

The only factor you need to change is in the `where` *statement.*

- *Tangent Circles (one intersection):* `[a,b,c,d]=[(-1,0), (-1,-1), (1,0), (1,1)]`
- *Two intersection points:* `[a,b,c,d]=[(2,0), (2,-1), (1,0), (1,1)]`
- *No intersection points:* `[a,b,c,d]=[(0,0), (5,5), (0,0), (1,1)]`

Example:

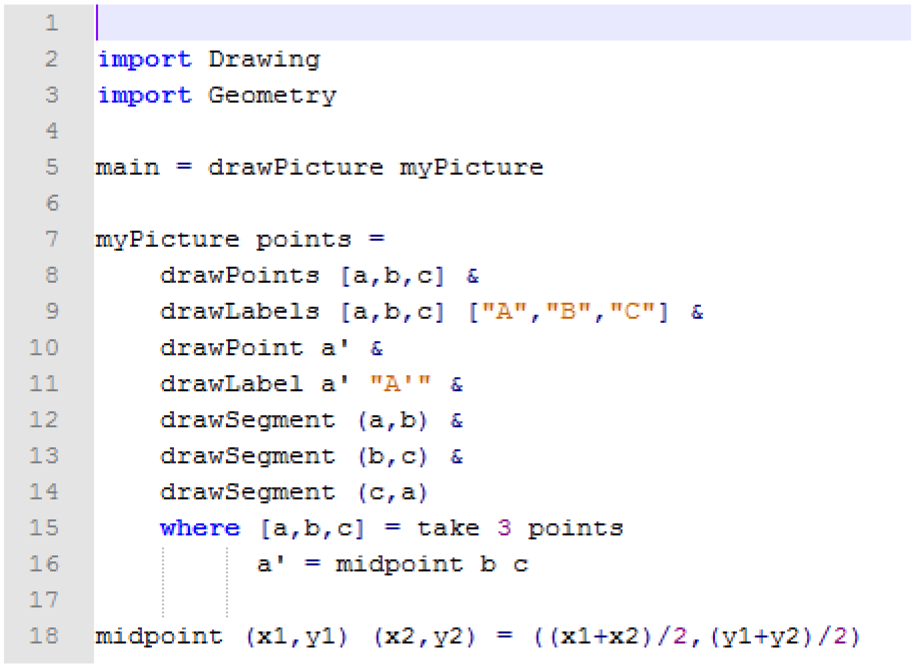


```

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points =
    red ( drawCircle'' (a,b) ) &
    blue ( drawCircle'' (c,d) ) &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawLabels intersects ["X","Y"] &
    drawPoints intersects &
    message "Circle-Circle Intersection"
    where [a,b,c,d] = [(0,0),(5,5),(0,0),(1,1)]
          intersects = circle_circle (c,d) (a,b)
```

Lesson 3: Midpoint & Distance

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 3 Part 1: Open the editor and then open lesson3a.hs. Look at the code and write down what you understand in the code. Run the code.



```
 1  
 2  import Drawing
 3  import Geometry
 4  
 5  main = drawPicture myPicture
 6  
 7  myPicture points =
 8      drawPoints [a,b,c] &
 9      drawLabels [a,b,c] ["A","B","C"] &
10      drawPoint a' &
11      drawLabel a' "A'" &
12      drawSegment (a,b) &
13      drawSegment (b,c) &
14      drawSegment (c,a)
15      where [a,b,c] = take 3 points
16            a' = midpoint b c
17  
18  midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Notes to Teacher:

- This program has two 3 toplevel definitions (main, myPicture and midpoint)
- The function `myPicture` has 2 auxiliary definitions (line 15 and line 16)
- The `midpoint` should be similar to the formula used in class. Discuss it.

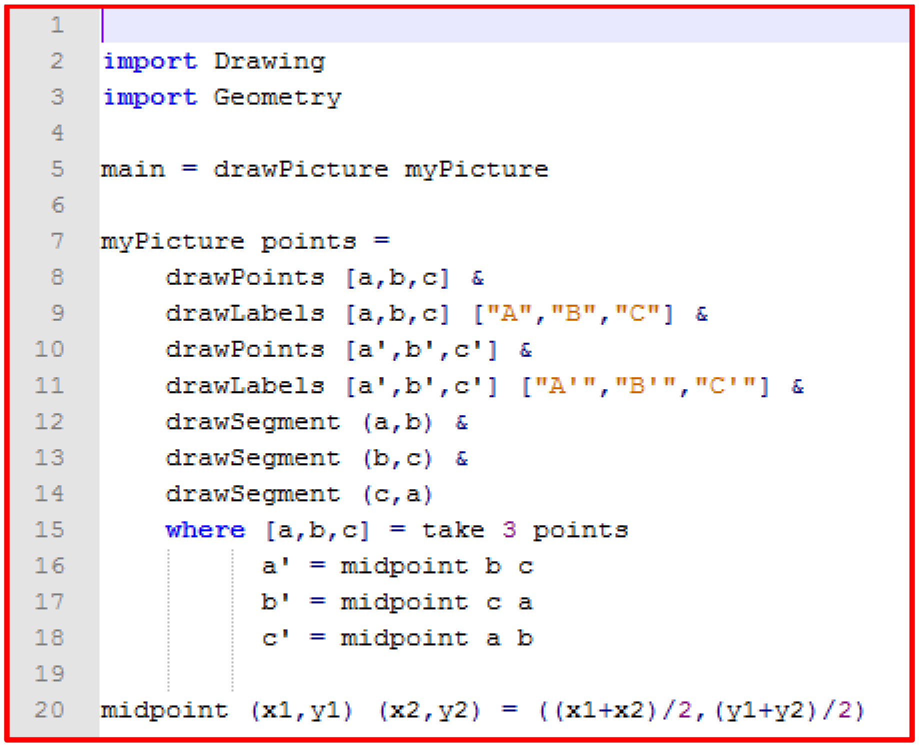
Questions:

1. What does the program do?
   Draws a triangle and the midpoint of side or segment BC.
2. Why did the programmer use `a’` in the language?
   Because the midpoint is across from angle A

Exercise: *Draw and Label the two missing midpoints of the other segments. Label appropriately as discussed in the question above.*

Save your program as yourname_lesson3b.hs

Run the program to check.

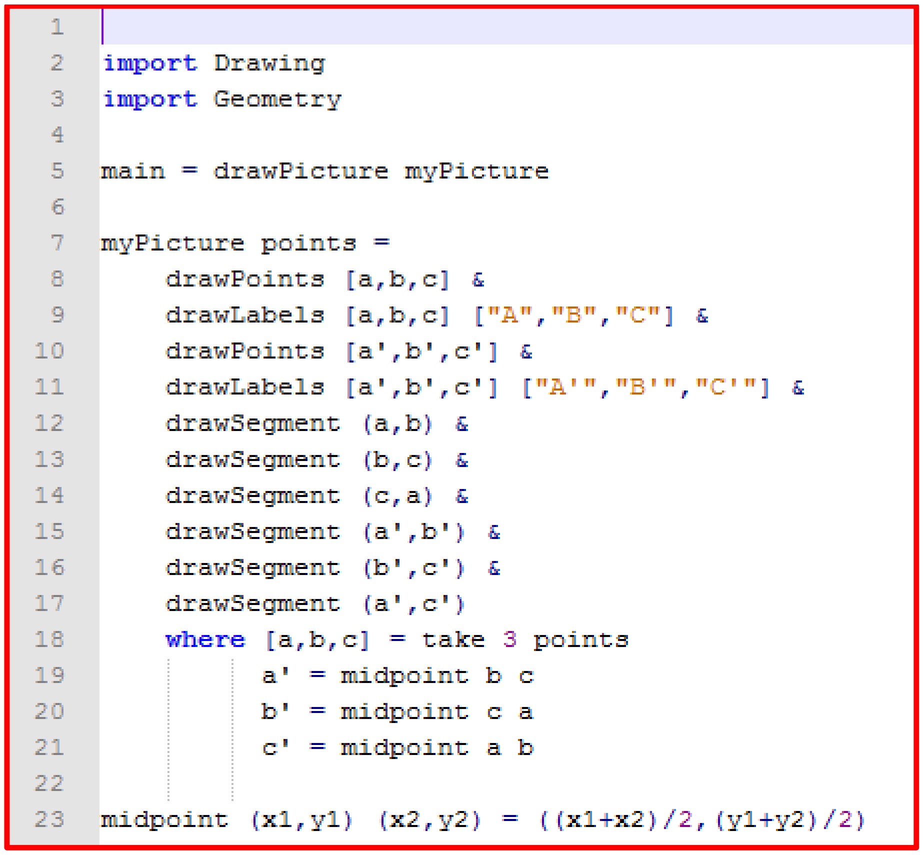


```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a)
    where [a,b,c] = take 3 points
          a' = midpoint b c
          b' = midpoint c a
          c' = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Exercise: *Manipulate the program you just created, lesson3b.hs, to draw the 3 segments connecting the midpoints.*

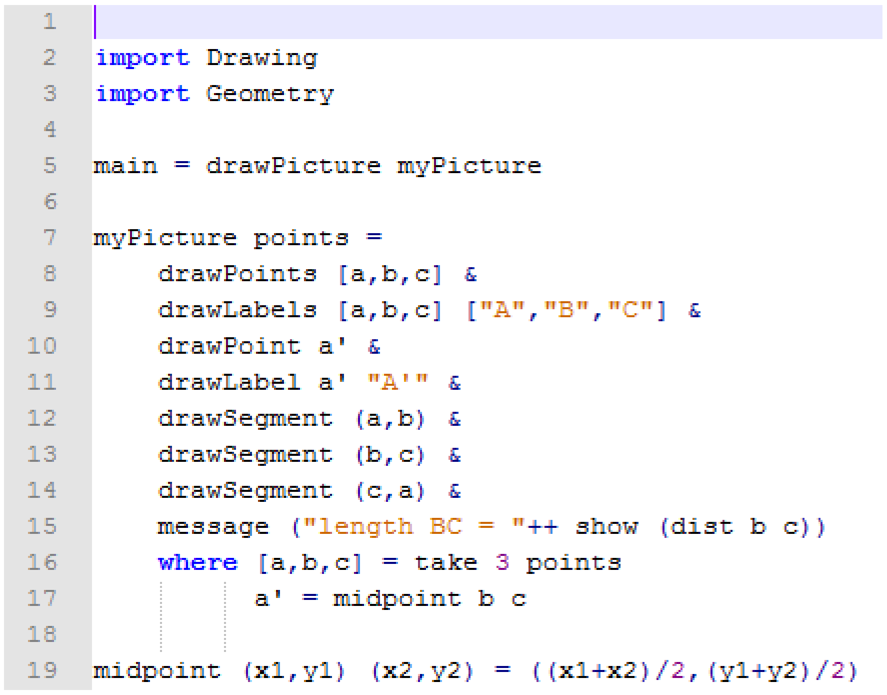
Save your program as yourname_lesson3c.hs

Run the program to check.



```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (a',c')
    where [a,b,c] = take 3 points
          a' = midpoint b c
          b' = midpoint c a
          c' = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Lesson 3 Part 2: Open the editor and then open lesson3d.hs.



```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoint a' &
    drawLabel a' "A'" &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    message ("length BC = "++ show (dist b c))
    where [a,b,c] = take 3 points
          a' = midpoint b c

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Question:

This program should look familiar to lesson3a.hs.

1. What do you think the program does differently than lesson3a.hs?

   Run the program.
   Draws triangle ABC. Identifies the midpoint of side BC as A'. Also displays the distance of side BC.
2. What does the function `dist b c` do?
   Calculates the length or distance of side BC.
3. What does the function `message` do?
   Displays a message to the user.

Note to Teacher: You will need to also discuss all of the factors of the following row:

```
message ("length BC = "++ show (dist b c))
```

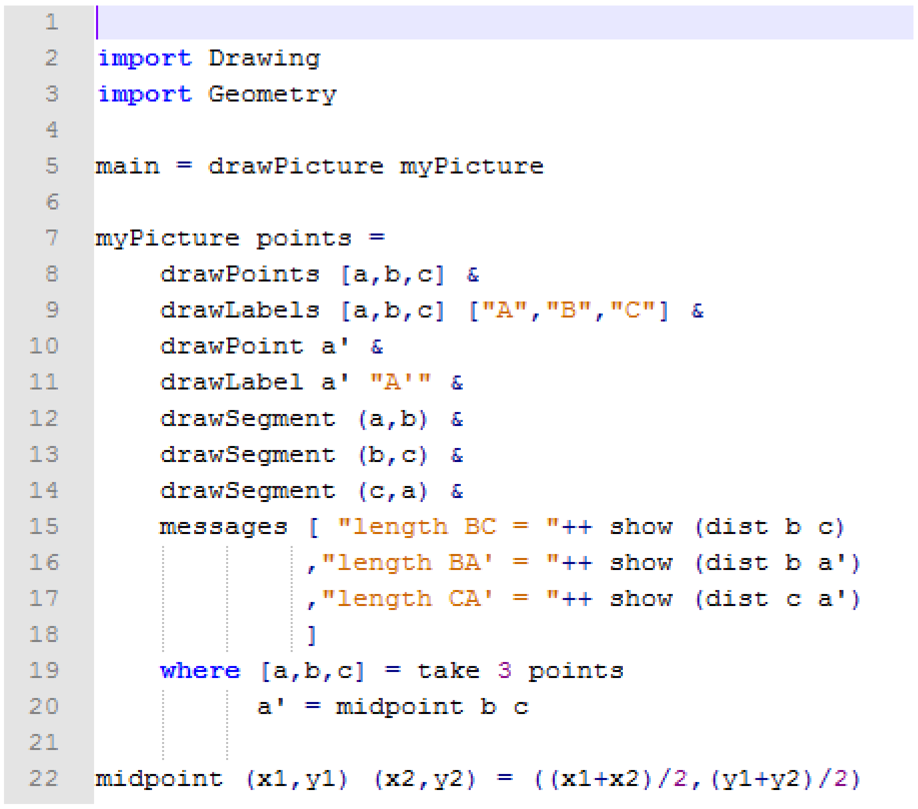
Refer to the Coding Overview.

Exercise: *Open the editor and then open lesson3e.hs.*

1. *What does the program do?*

   Run the program to check.

2. *What does the function* `messages` *do?*



```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoint a' &
    drawLabel a' "A'" &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    messages [ "length BC = "++ show (dist b c)
             ,"length BA' = "++ show (dist b a')
             ,"length CA' = "++ show (dist c a')
             ]
    where [a,b,c] = take 3 points
          a' = midpoint b c

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Note to Teacher: You will need to discuss how the `messages` function is used to show more than one message to the user.

Lesson 3 Ending Exercises:

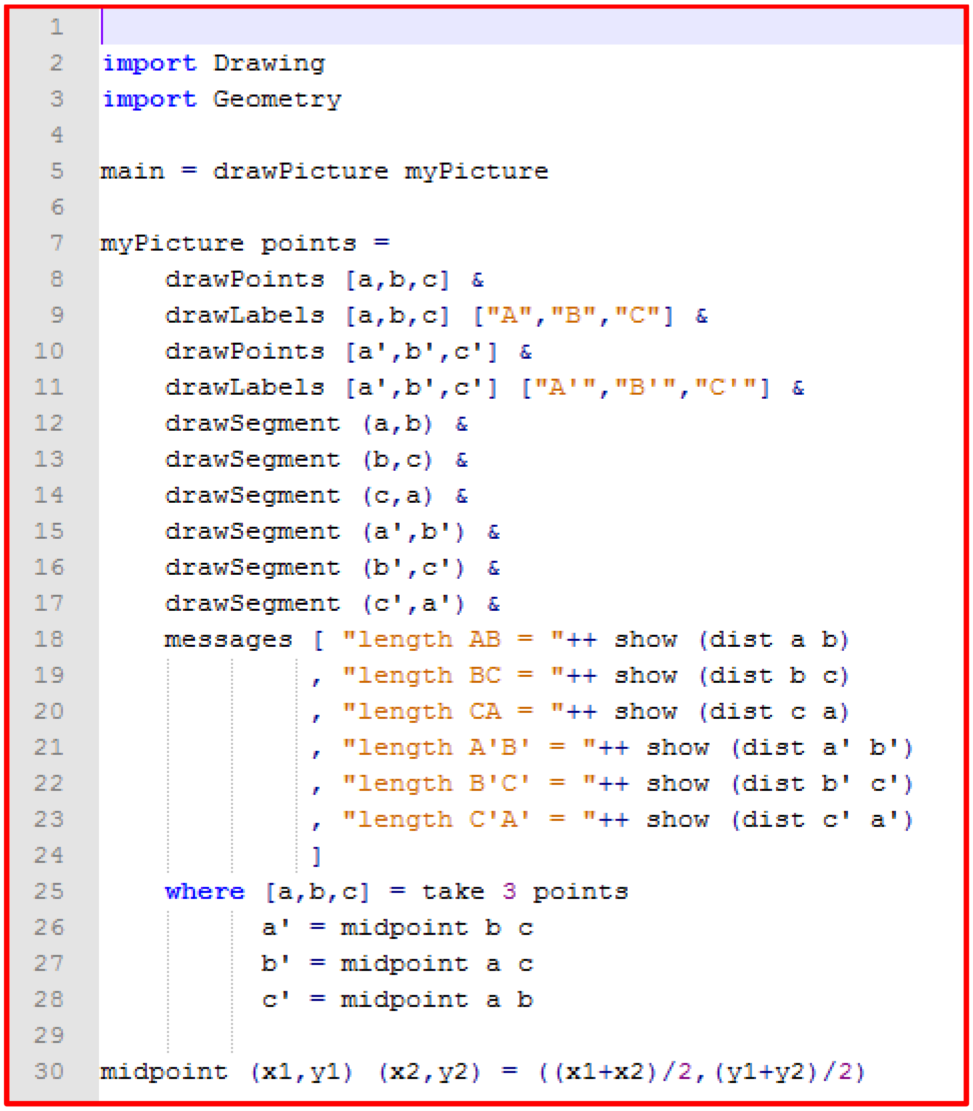
Exercise: *Open lesson3d.hs*

Manipulate the program to do the following:

1. *Draw all three midpoints of the segments.*
2. *Draw all of the segments connecting the midpoints.*
3. *Show the measure the distances of all 3 original sides.*
4. *Show the measure the distances of all 3 segments connecting the midpoints.*

Save the program as yourname_lesson3f.hs

Run the program to check



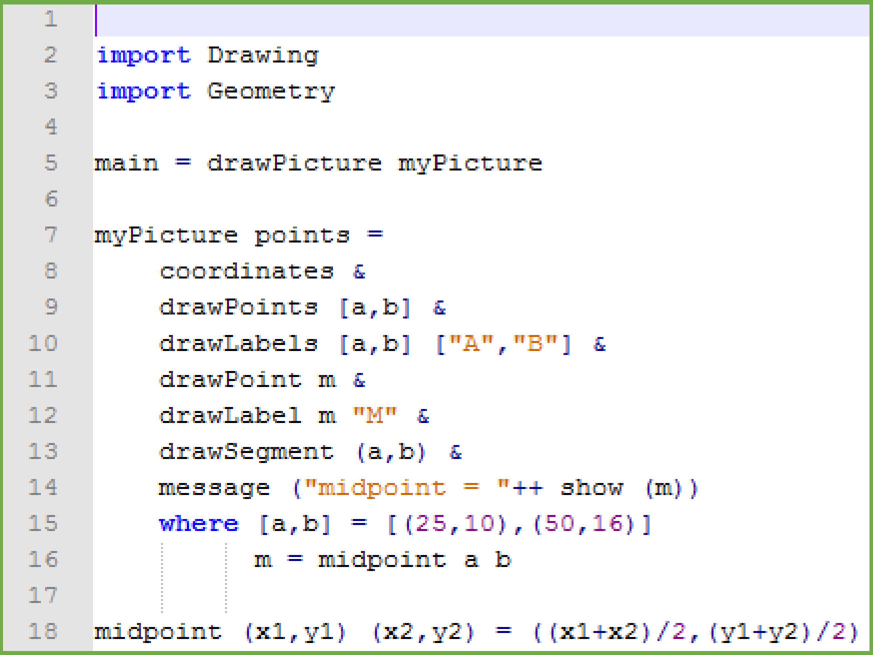
```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',a') &
    messages [ "length AB = "++ show (dist a b)
             , "length BC = "++ show (dist b c)
             , "length CA = "++ show (dist c a)
             , "length A'B' = "++ show (dist a' b')
             , "length B'C' = "++ show (dist b' c')
             , "length C'A' = "++ show (dist c' a')
             ]
    where [a,b,c] = take 3 points
          a' = midpoint b c
          b' = midpoint a c
          c' = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Lesson 3 Further Applications:

Teachers you can easily extend this lesson by removing the randomness from the programs.

Key_lesson3N1

> *This program allows the user to change the points for a and b to actually get an output for the midpoint m.*
> *Students could use this program to check answers for classwork and homework or use it to solve more complex problems in which calculating the midpoint is a minute step.*
>
> *Note to teacher: If the points are out of the domain and range of [-10,10], the points/line may not appear on the graph unless you zoom out. The coordinates of the midpoint will appear whether it is in or out of the domain and range of [-10,10].*



```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b] &
    drawLabels [a,b] ["A","B"] &
    drawPoint m &
    drawLabel m "M" &
    drawSegment (a,b) &
    message ("midpoint = "++ show (m))
    where [a,b] = [(25,10),(50,16)]
          m = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

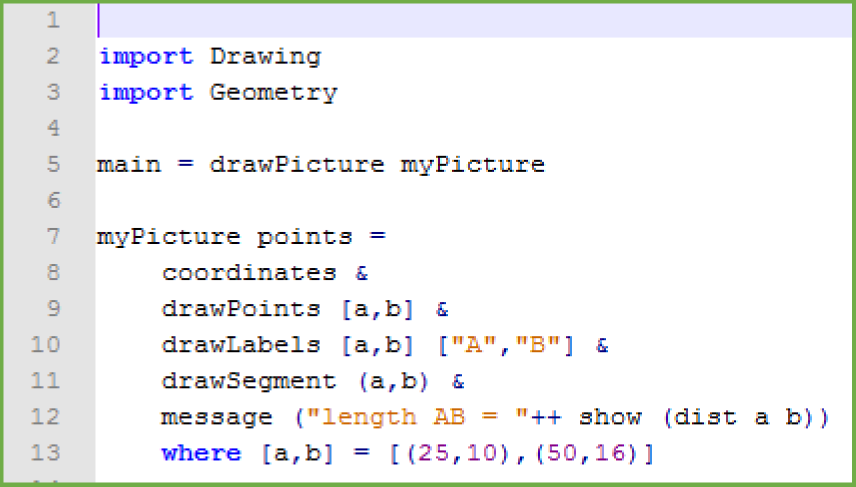
Lesson 3: Further Applications:

Teachers you can easily extend this lesson by removing the randomness from the programs.

Key_lesson3N2

This program allows the user to change the points for a and b to actually get the length of AB.
Students could use this program to check answers for classwork and homework or use it to solve more complex problems in which calculating the distance is a minute step.

Note to teacher: If the points are out of the domain and range of [-10,10], the points/line may not appear on the graph unless you zoom out. The length of the segment will appear whether the points are in or out of the domain and range of [-10,10].



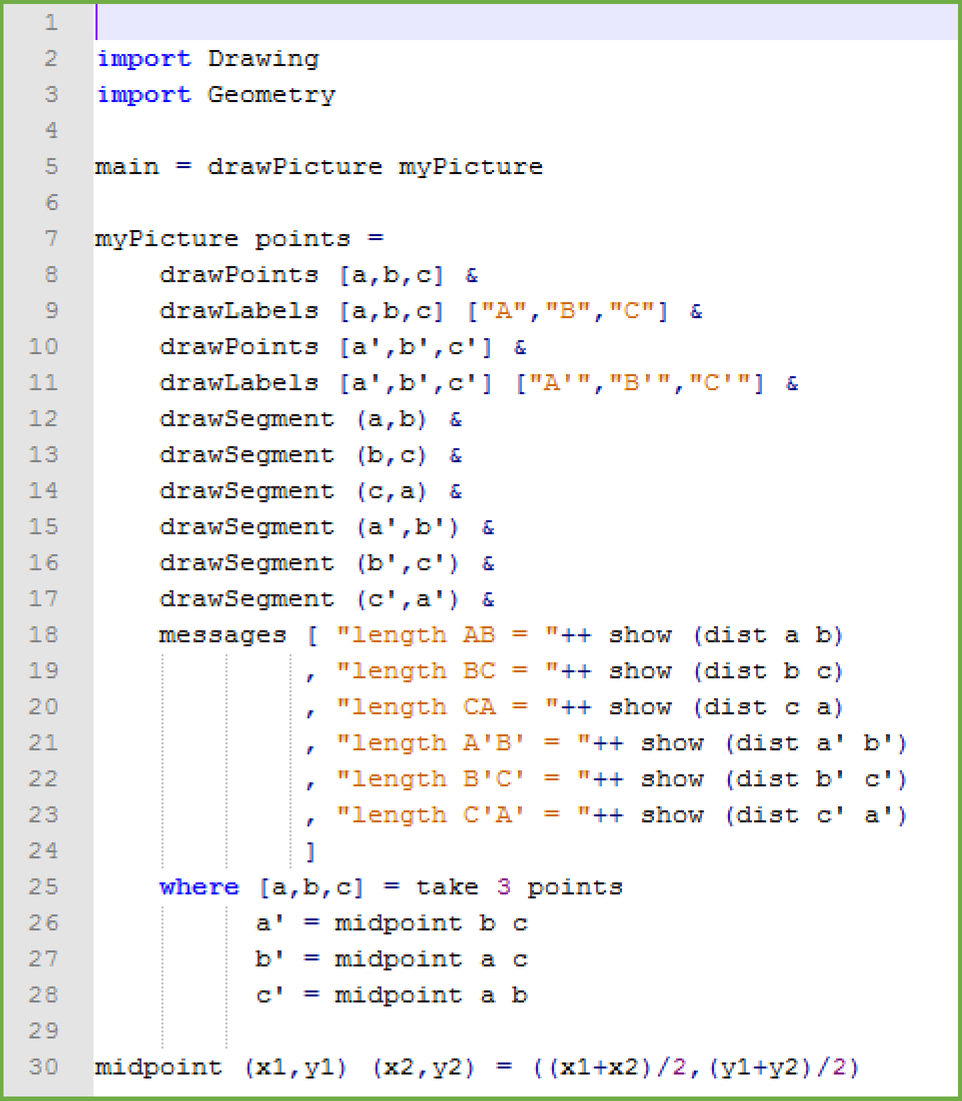
```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b] &
    drawLabels [a,b] ["A","B"] &
    drawSegment (a,b) &
    message ("length AB = "++ show (dist a b))
    where [a,b] = [(25,10),(50,16)]
```

Lesson 3: Further Applications:

Key_lesson3f.hs

The students should have written a program similar to this in the ending exercise. You can show the students how the program created two similar triangles because the side lengths are proportional.

You could also extend this exercise by having students show the distance of all the segments in the four triangles created. This could be used to show that the four triangles are congruent.

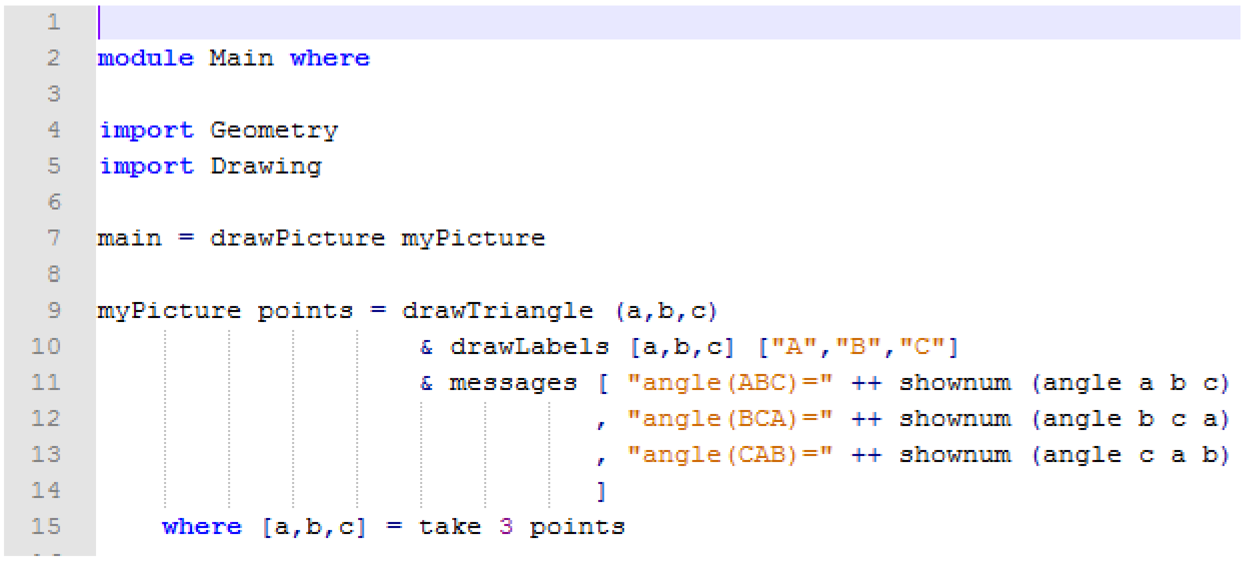


```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',a') &
    messages [ "length AB = "++ show (dist a b)
             , "length BC = "++ show (dist b c)
             , "length CA = "++ show (dist c a)
             , "length A'B' = "++ show (dist a' b')
             , "length B'C' = "++ show (dist b' c')
             , "length C'A' = "++ show (dist c' a')
             ]
    where [a,b,c] = take 3 points
          a' = midpoint b c
          b' = midpoint a c
          c' = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Lesson 4: Angles

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 4: Open the editor and then open lesson4a.hs. Look at the code and write down what you understand in the code. Run the code.



```haskell

module Main where

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points = drawTriangle (a,b,c)
                    & drawLabels [a,b,c] ["A","B","C"]
                    & messages [ "angle(ABC)=" ++ shownum (angle a b c)
                               , "angle(BCA)=" ++ shownum (angle b c a)
                               , "angle(CAB)=" ++ shownum (angle c a b)
                               ]
    where [a,b,c] = take 3 points
```

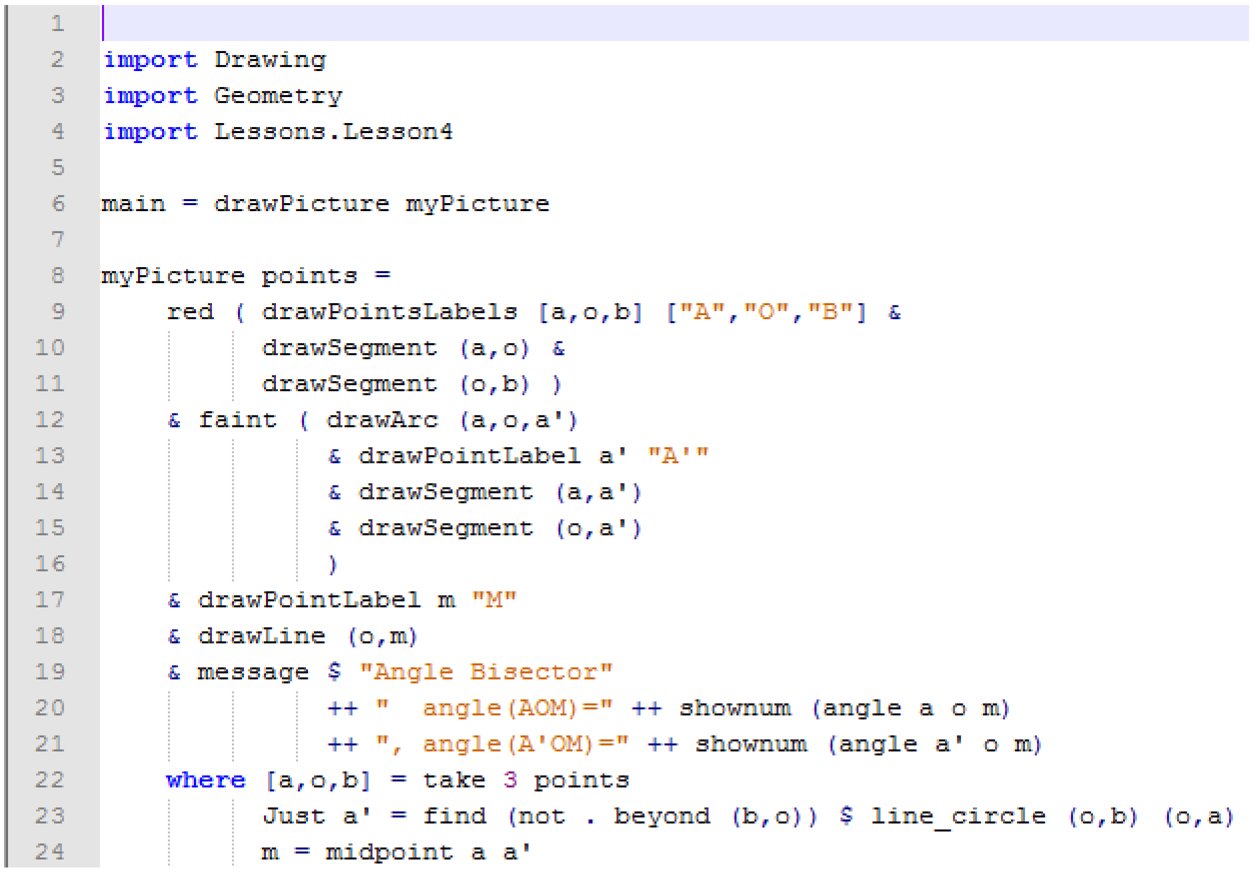
Questions:

1. What does the program do?
   Draws Triangle ABC and displays the angle measure of each angle.
2. What does the b stand for in `shownum (angle a b c)`?
   b is the vertex of the angle being measured.

Note to Teacher: You will want to briefly discuss the function `drawTriangle (a,b,c)`.

Exercise: *Open lesson4b.hs in the editor. Look at the program and sketch a picture of what the program is doing?*

Run the program to check.



```
 1  
 2  import Drawing
 3  import Geometry
 4  import Lessons.Lesson4
 5  
 6  main = drawPicture myPicture
 7  
 8  myPicture points =
 9      red ( drawPointsLabels [a,o,b] ["A","O","B"] &
10          drawSegment (a,o) &
11          drawSegment (o,b) )
12      & faint ( drawArc (a,o,a')
13              & drawPointLabel a' "A'"
14              & drawSegment (a,a')
15              & drawSegment (o,a')
16              )
17      & drawPointLabel m "M"
18      & drawLine (o,m)
19      & message $ "Angle Bisector"
20              ++ "  angle(AOM)=" ++ shownum (angle a o m)
21              ++ ", angle(A'OM)=" ++ shownum (angle a' o m)
22      where [a,o,b] = take 3 points
23          Just a' = find (not . beyond (b,o)) $ line_circle (o,b) (o,a)
24          m = midpoint a a'
```

Note to Teacher: You will need to discuss line 23.

Review the description of `find` *in the section Function Reference List – System Functions and the description of* `beyond` *in the section Function Reference List – Geometry Functions in the Coding Overview before proceeding. The function* `(not.beyond (b,o))` *is a composite predicate that will return* `True` *for points that lie either* `between (b,o)` *or* `beyond (o,b)`

The definition in line 23 is incomplete, since we do not check for `Nothing`*. However, it is almost impossible in this case that no point be found, and we could actually prove this claim for ideal points. So, in this case, it is OK to do without using the general* `case … of` *construc*tion

Questions:

1. What do you notice about the two angles that are drawn?
   The angle measures are the same.
2. What does the program draw?
   Draws the angle bisector of a given angle.
3. How does the program calculate where `a’` should be?
   Calculating the intersection of line OB and Circle OA that is not beyond line OB.
4. What does the function `drawArc (a,o,a’)` do?
   Draws and Arc centered at O going through A and A’.

Lesson 4 Ending Exercises:

Exercise: *Open yourname_lesson3f.hs*

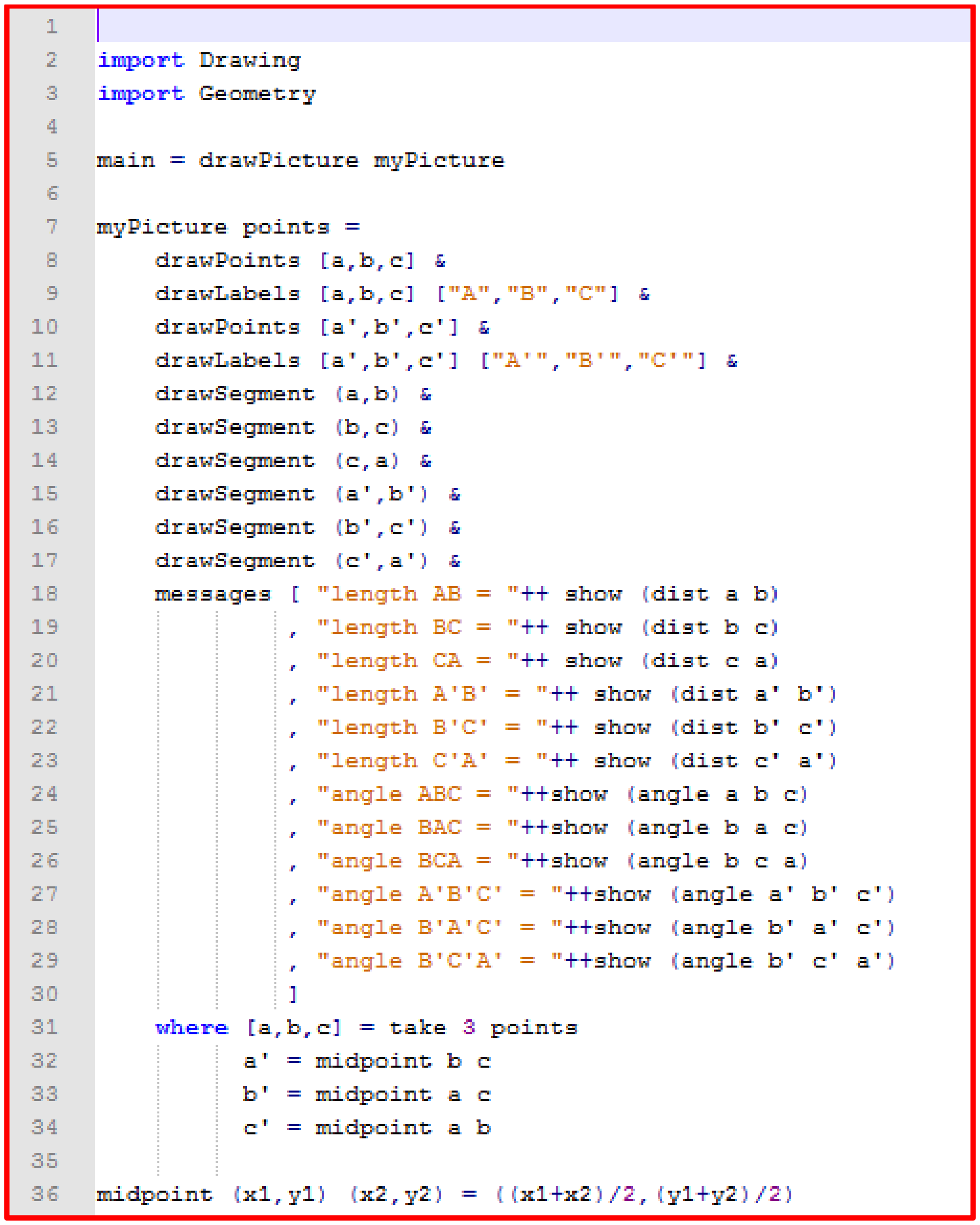
Manipulate the program to show the angle measures of the 3 angles in the large triangle and the three angles in the triangle created by the midpoints.

Save the program as yourname_lesson4c.hs

Run the program to check

Question: *What do you notice about the angle measures in the large triangle compared with the smaller triangle?*

Note to Teacher: Similar Triangles are created.

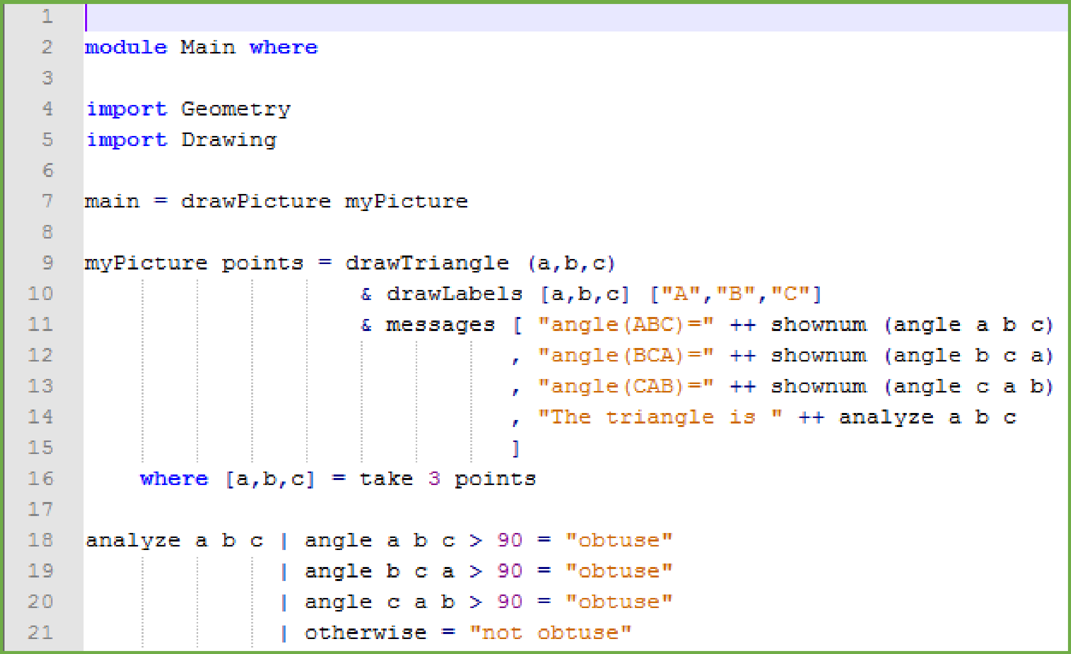


```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',a') &
    messages [ "length AB = "++ show (dist a b)
             , "length BC = "++ show (dist b c)
             , "length CA = "++ show (dist c a)
             , "length A'B' = "++ show (dist a' b')
             , "length B'C' = "++ show (dist b' c')
             , "length C'A' = "++ show (dist c' a')
             , "angle ABC = "++show (angle a b c)
             , "angle BAC = "++show (angle b a c)
             , "angle BCA = "++show (angle b c a)
             , "angle A'B'C' = "++show (angle a' b' c')
             , "angle B'A'C' = "++show (angle b' a' c')
             , "angle B'C'A' = "++show (angle b' c' a')
             ]
    where [a,b,c] = take 3 points
          a' = midpoint b c
          b' = midpoint a c
          c' = midpoint a b

midpoint (x1,y1) (x2,y2) = ((x1+x2)/2,(y1+y2)/2)
```

Lesson 4 Further Applications:

Teachers you can easily extend this lesson through classifying triangles.

lesson4d.hs

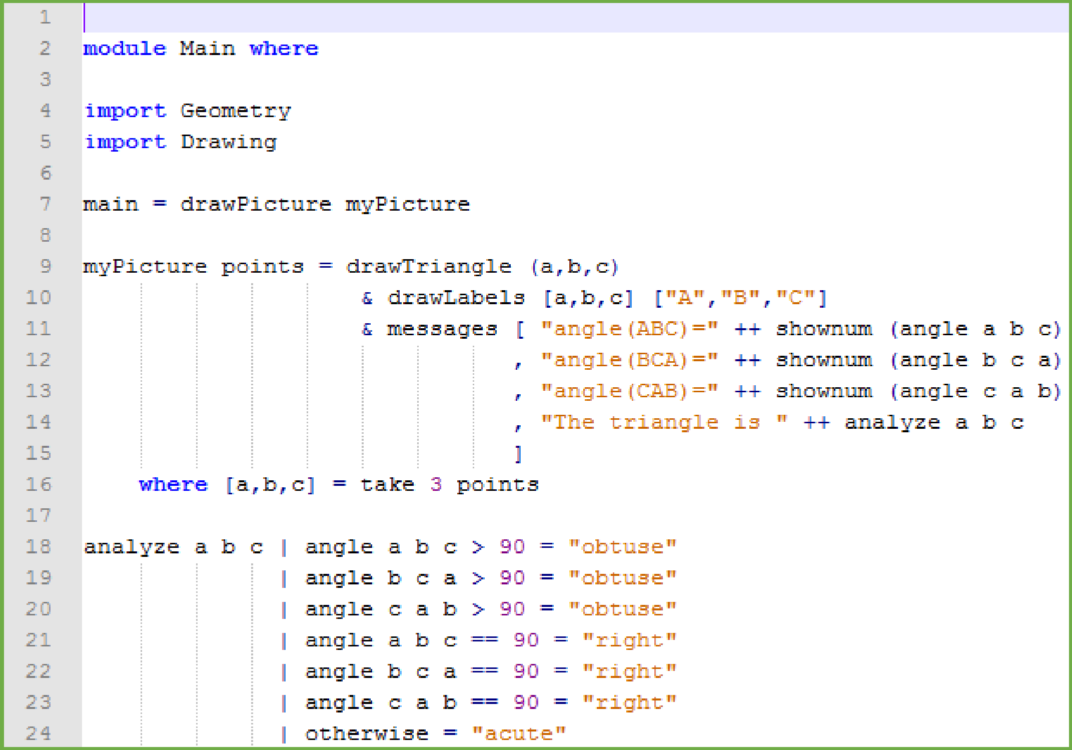


```haskell
module Main where

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points = drawTriangle (a,b,c)
                    & drawLabels [a,b,c] ["A","B","C"]
                    & messages [ "angle(ABC)=" ++ shownum (angle a b c)
                               , "angle(BCA)=" ++ shownum (angle b c a)
                               , "angle(CAB)=" ++ shownum (angle c a b)
                               , "The triangle is " ++ analyze a b c
                               ]
    where [a,b,c] = take 3 points

analyze a b c | angle a b c > 90 = "obtuse"
              | angle b c a > 90 = "obtuse"
              | angle c a b > 90 = "obtuse"
              | otherwise = "not obtuse"
```

The students should have seen a similar program in lesson4a.hs. This program will display a message on whether the triangle is acute, right, or obtuse.

My suggestion is to show the students how to display whether or not it is obtuse. Then have them manipulate the program to show whether it is acute, right, or obtuse as in key_lesson4e.hs.

Example key_lesson4e.hs: There are other methods to writing this program.



```haskell
module Main where

import Geometry
import Drawing

main = drawPicture myPicture

myPicture points = drawTriangle (a,b,c)
                    & drawLabels [a,b,c] ["A","B","C"]
                    & messages [ "angle(ABC)=" ++ shownum (angle a b c)
                               , "angle(BCA)=" ++ shownum (angle b c a)
                               , "angle(CAB)=" ++ shownum (angle c a b)
                               , "The triangle is " ++ analyze a b c
                               ]
    where [a,b,c] = take 3 points

analyze a b c | angle a b c > 90 = "obtuse"
              | angle b c a > 90 = "obtuse"
              | angle c a b > 90 = "obtuse"
              | angle a b c == 90 = "right"
              | angle b c a == 90 = "right"
              | angle c a b == 90 = "right"
              | otherwise = "acute"
```

Lesson 5: Perpendicular & Parallel Lines

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 5 Part 1:

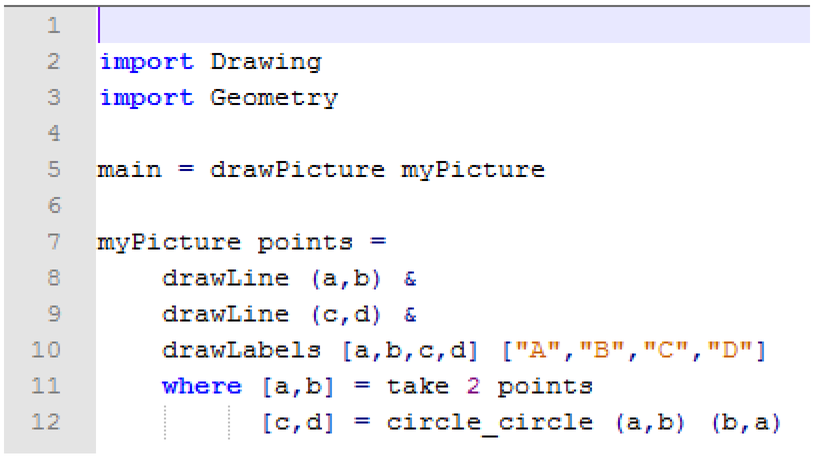
Exercise: *Open lesson5a.hs in the editor. Create a sketch of what the program is drawing.*

Run the program to check your answer.

Exercise: *Manipulate lesson5a.hs to label the intersection point as* `E` *and display the message "Perpendicular Lines."*

Save the program as yourname_lesson5b.hs

Run the program to check your answer.



```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawLine (a,b) &
    drawLine (c,d) &
    drawLabels [a,b,c,d] ["A","B","C","D"]
    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
```

Lesson 5 Part 2:

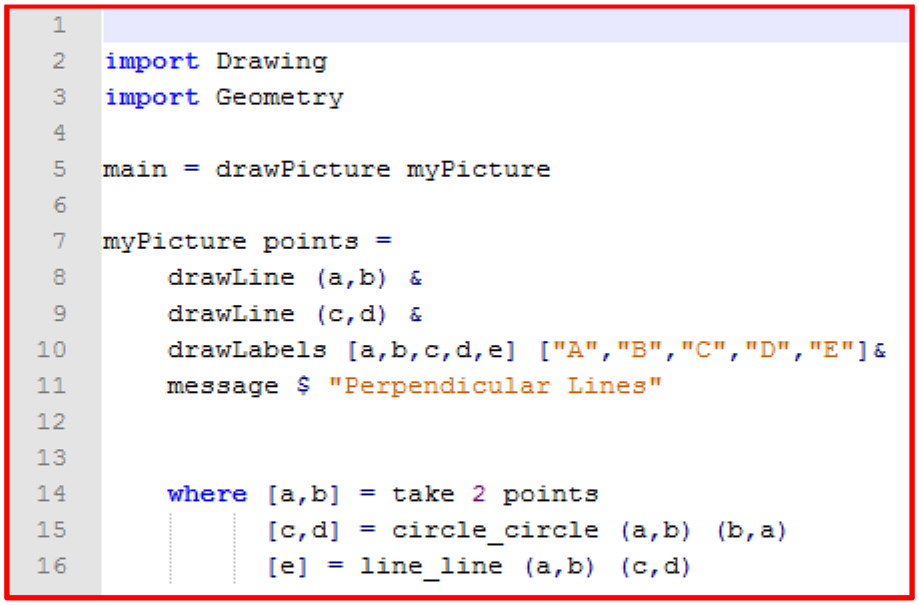
Question:

How can we draw a line parallel to line AB in the previous exercises?
Draw a perpendicular to line CD

Exercise: *Manipulate yourname_lesson5b.hs to draw line FG parallel to line AB and display the message "Parallel Lines."*

Save the program as yourname_lesson5c.hs

Run the program to check your answer.



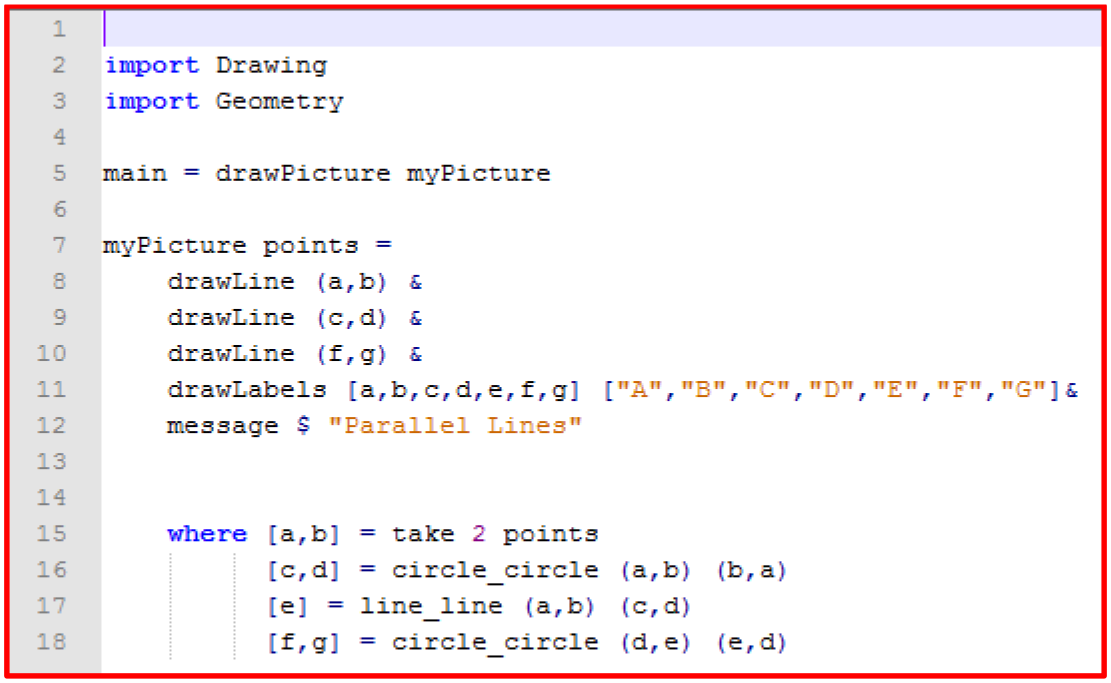
```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawLine (a,b) &
    drawLine (c,d) &
    drawLabels [a,b,c,d,e] ["A","B","C","D","E"]&
    message $ "Perpendicular Lines"


    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
          [e] = line_line (a,b) (c,d)
```

Lesson 5 Ending Exercises:

Exercise: *Manipulate yourname_lesson5c.hs to draw another line HI parallel to line AB and display the message "Parallel Lines." Your display should have three parallel lines and one perpendicular line.*

Save the program as yourname_lesson5d.hs

Run the program to check your answer.

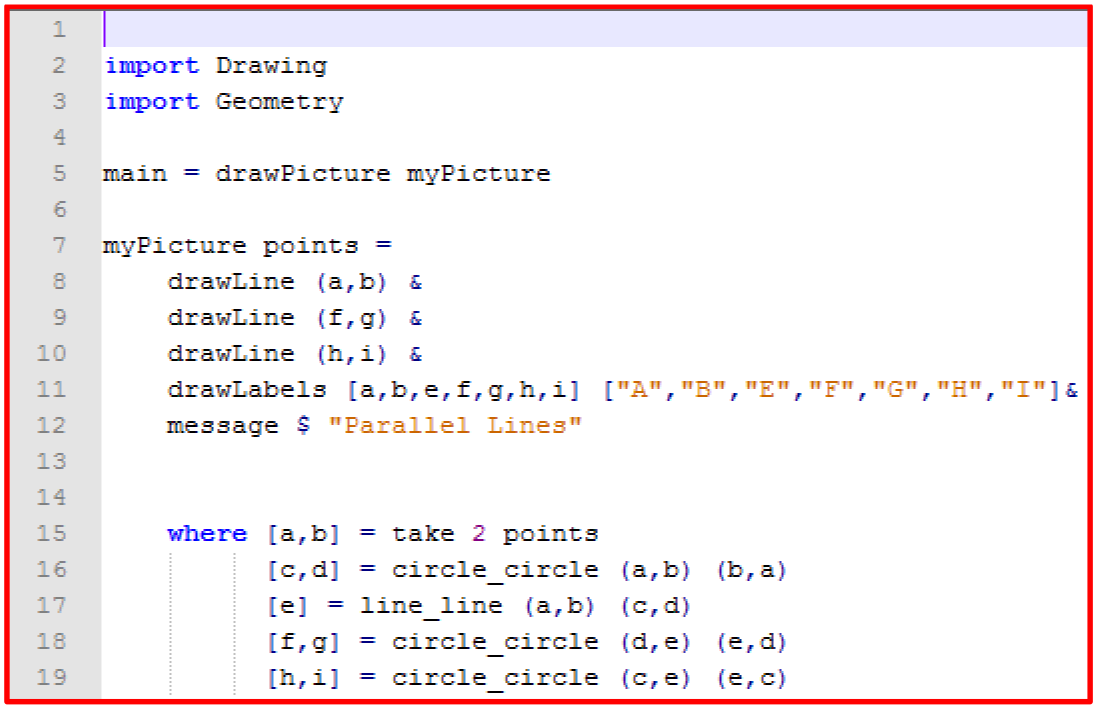


```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawLine (a,b) &
    drawLine (c,d) &
    drawLine (f,g) &
    drawLabels [a,b,c,d,e,f,g] ["A","B","C","D","E","F","G"]&
    message $ "Parallel Lines"


    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
          [e] = line_line (a,b) (c,d)
          [f,g] = circle_circle (d,e) (e,d)
```

Exercise: *Manipulate yourname_lesson5d.hs to remove the perpendicular line and labels of the perpendicular line from the display.*

Save the program as yourname_lesson5e.hs

Run the program to check your answer.



```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawLine (a,b) &
    drawLine (f,g) &
    drawLine (h,i) &
    drawLabels [a,b,e,f,g,h,i] ["A","B","E","F","G","H","I"]&
    message $ "Parallel Lines"


    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
          [e] = line_line (a,b) (c,d)
          [f,g] = circle_circle (d,e) (e,d)
          [h,i] = circle_circle (c,e) (e,c)
```

Lesson 6: Perimeter & Area

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 6:

Questions:

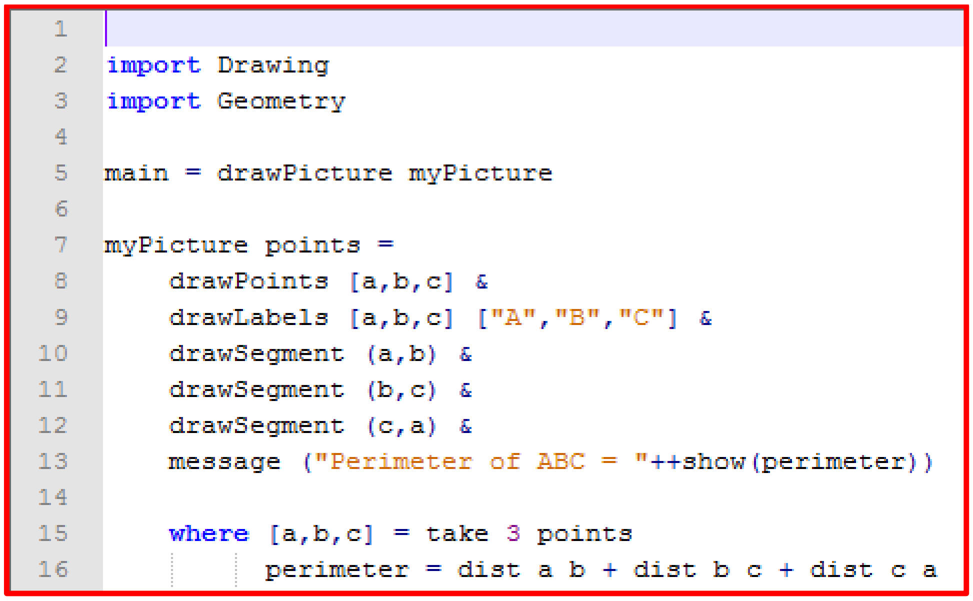
What function(s) could we use to help us create a program to calculate the perimeter of a figure?

dist

Exercise: *Using lesson1f.hs in the editor, create a program to draw a triangle using three points. The program should also show the lengths of the three sides and show the perimeter of the triangle.*

Save the program as yourname_lesson6a.hs

Run the program to check.



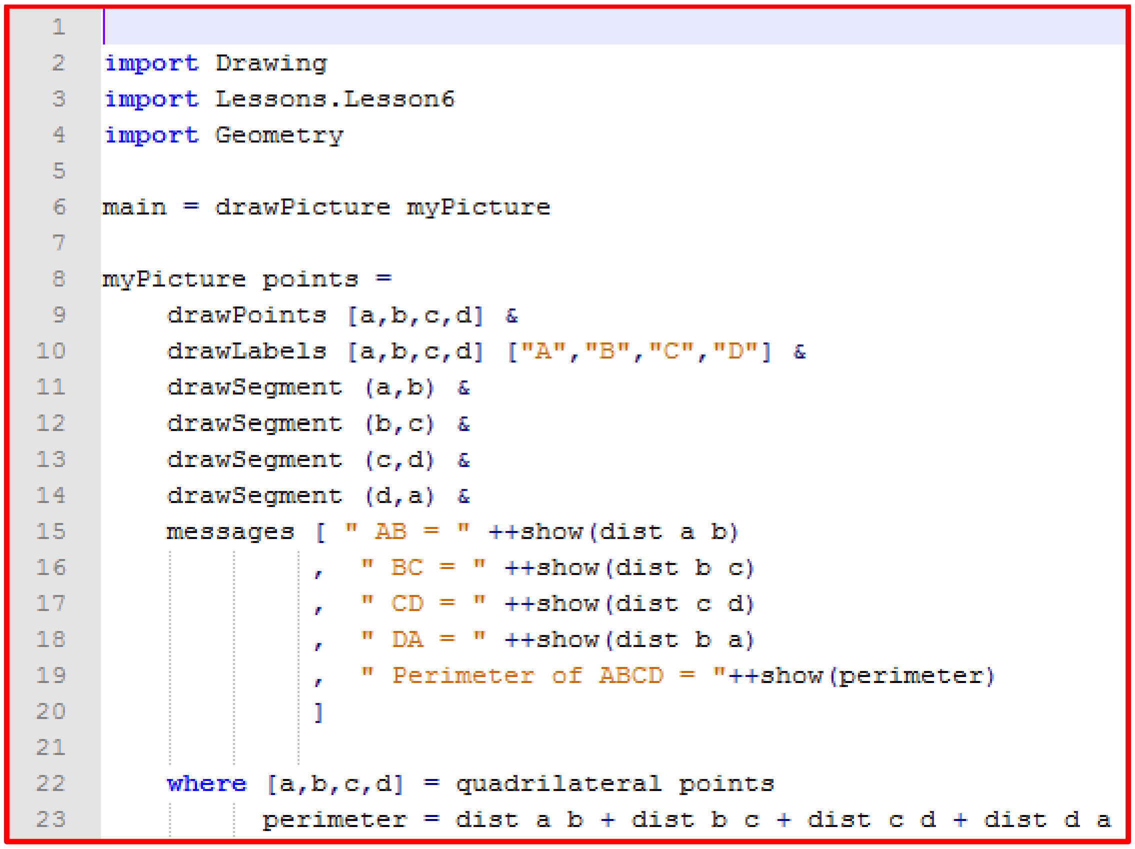
```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    message ("Perimeter of ABC = "++show(perimeter))

    where [a,b,c] = take 3 points
          perimeter = dist a b + dist b c + dist c a
```

***Exercise:** Manipulate the above program for a four sided figure and only display the perimeter as a message.*

Note: Instead of using `take 4 points`*, use the function* `quadrilateral points`.

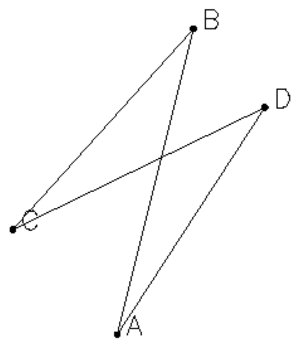
Save the program as yourname_lesson6b.hs

Run the program to check.



```haskell

import Drawing
import Lessons.Lesson6
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d] &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,d) &
    drawSegment (d,a) &
    messages [ " AB = " ++show(dist a b)
             , " BC = " ++show(dist b c)
             , " CD = " ++show(dist c d)
             , " DA = " ++show(dist b a)
             , " Perimeter of ABCD = "++show(perimeter)
             ]

    where [a,b,c,d] = quadrilateral points
          perimeter = dist a b + dist b c + dist c d + dist d a
```

Note to Teacher: We have to use a quadrilateral function. If we were to take four random points like we did in the program with the triangle, we may not always have a quadrilateral as seen below. The quadrilateral function allows four random points to be chosen and following certain criteria it will rename the points, if necessary, to form a quadrilateral. In the case below if the quadrilateral function would have been used, it would have switched the points B and C.



It is at your discretion whether to discuss this with your students.

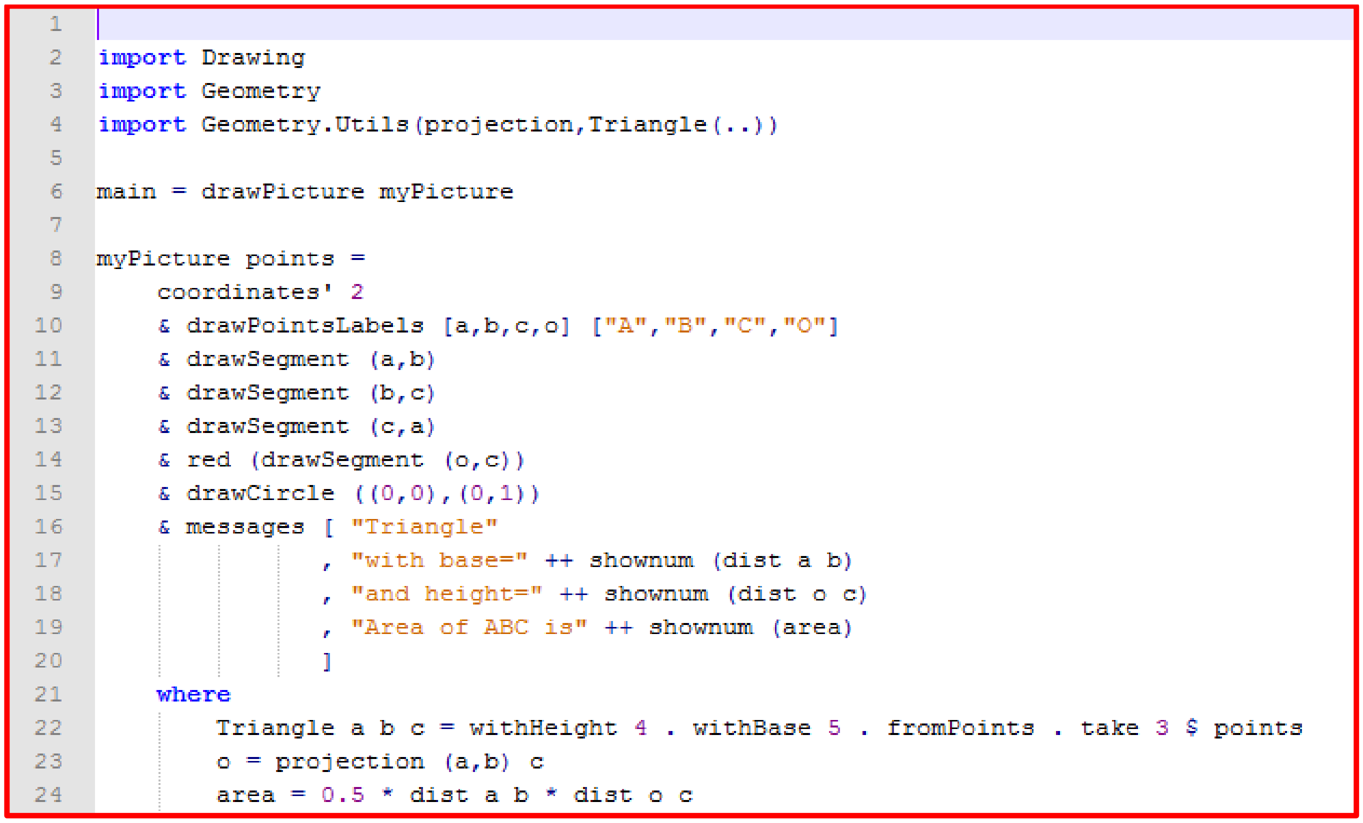
Lesson 6 Ending Exercises:

Note to Teacher: You are going to want to discuss that the triangle is drawn with a specified height and width. The students should only focus on calculating the area with the lengths of the sides, not the actual numerical value that is displayed on line 22.

Exercise: *Open lesson6c.hs to calculate the area of a triangle. Show the area as a message.*

Save the program as yourname_lesson6c.hs

Run the program to check.



```
 1
 2  import Drawing
 3  import Geometry
 4  import Geometry.Utils(projection,Triangle(..))
 5
 6  main = drawPicture myPicture
 7
 8  myPicture points =
 9      coordinates' 2
10      & drawPointsLabels [a,b,c,o] ["A","B","C","O"]
11      & drawSegment (a,b)
12      & drawSegment (b,c)
13      & drawSegment (c,a)
14      & red (drawSegment (o,c))
15      & drawCircle ((0,0),(0,1))
16      & messages [ "Triangle"
17                 , "with base=" ++ shownum (dist a b)
18                 , "and height=" ++ shownum (dist o c)
19                 , "Area of ABC is" ++ shownum (area)
20                 ]
21      where
22          Triangle a b c = withHeight 4 . withBase 5 . fromPoints . take 3 $ points
23          o = projection (a,b) c
24          area = 0.5 * dist a b * dist o c
```

Lesson 6 Further Applications:

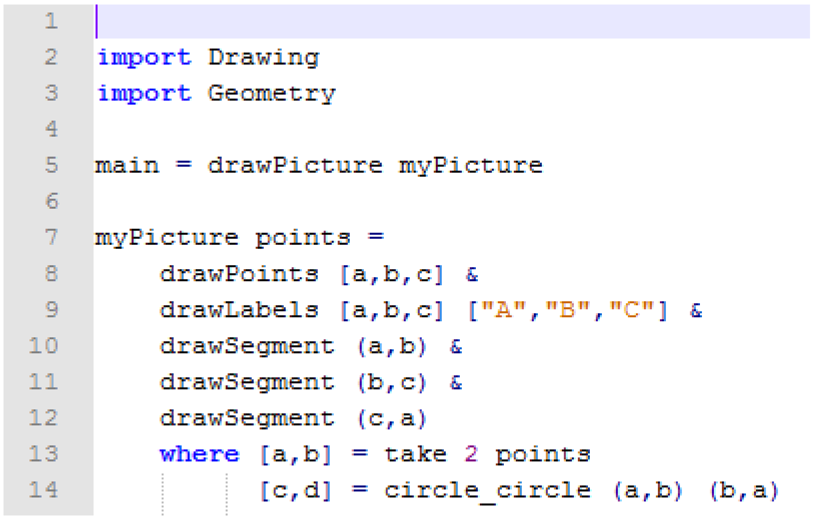
Teachers you can easily extend this lesson through letting the students use program they just manipulated in lesson6d.hs to calculate the area of triangles with different a different height and base.

This could also be used as a tool for students to calculate area in a different method.

Lesson 7: Equilateral Triangle

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 7: Open the editor and open lesson7a.hs. Look at the code carefully.



```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a)
    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
```

Exercise: *Determine what the program will draw.*

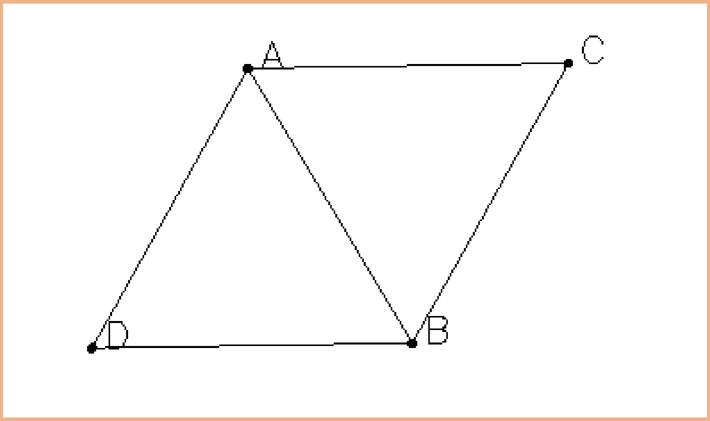
Run the program to check.

Questions:

1. What shape did the program create?
   Equilateral Triangle
2. How is point c computed in the program?
   C is computed by the intersection of circle A with B on the circle and circle B with A on the circle.
3. Why is the list `[c,d]` assigned to `circle_circle (a,b) (b,a)` ?
   The circles will intersect in two places.
4. Why is `d` not shown when you run the program?
   We choose to only show c. It is also not listed in the `drawPoints` or `drawLabels` functions.

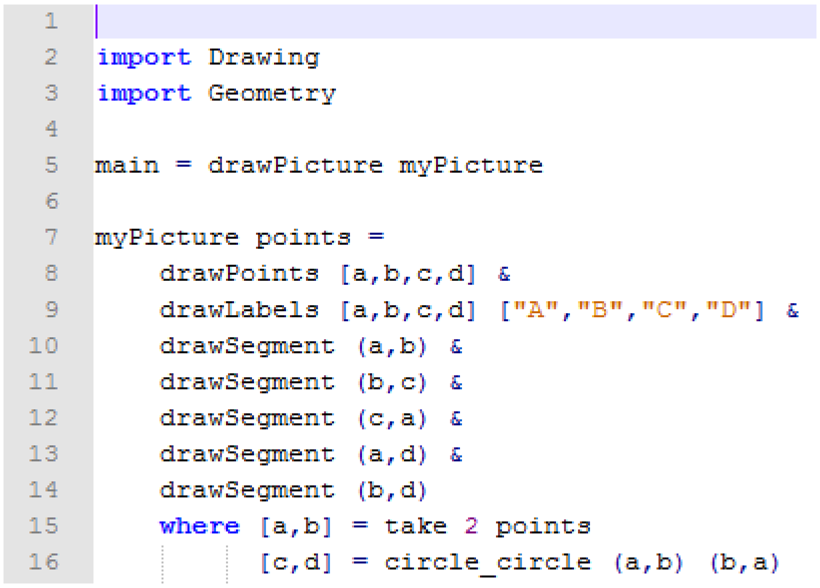
Lesson 7 Ending Exercises

Exercise: *Manipulate the program lesson7a.hs to draw a picture with the characteristics of the one below:*



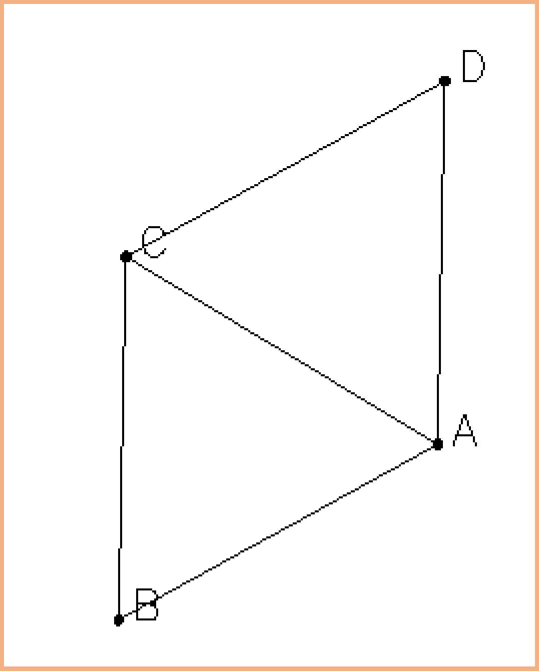
Save the program as yourname_lesson7b.hs

Run the program to check.



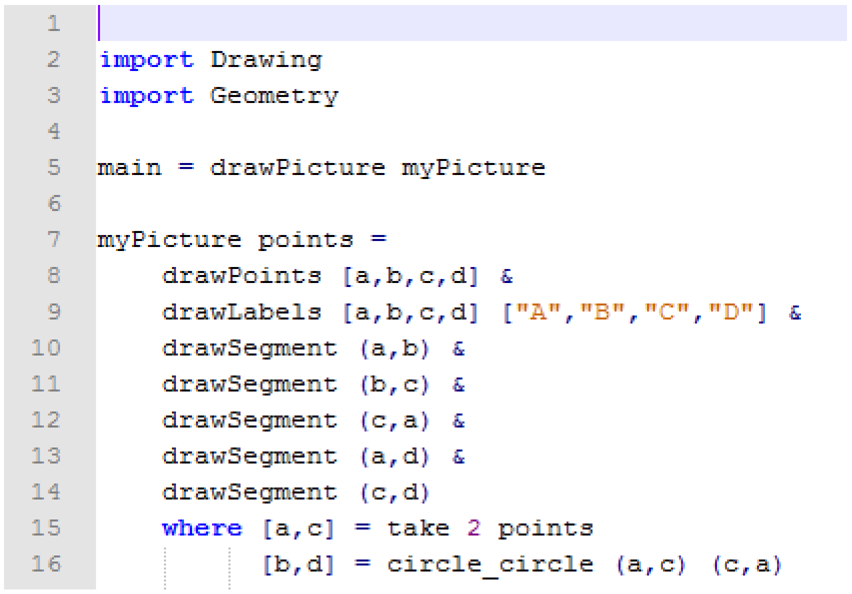
```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d] &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a,d) &
    drawSegment (b,d)
    where [a,b] = take 2 points
          [c,d] = circle_circle (a,b) (b,a)
```

Exercise: *Manipulate the program lesson7b.hs to draw a picture with the characteristics of the one below:*



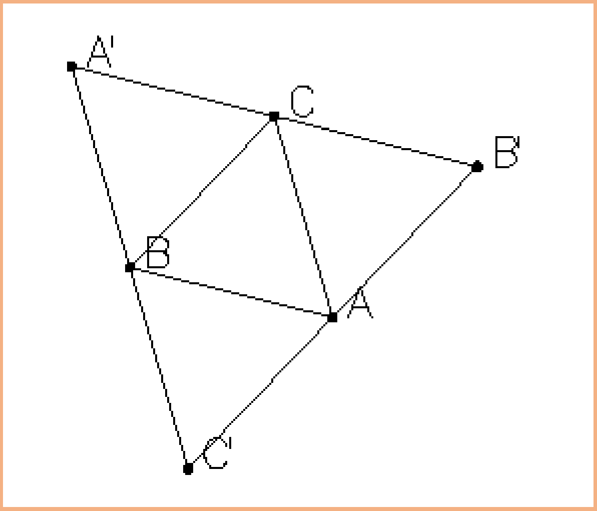
Save the program as yourname_lesson7c.hs

Run the program to check.



```

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,d] &
    drawLabels [a,b,c,d] ["A","B","C","D"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a,d) &
    drawSegment (c,d)
    where [a,c] = take 2 points
          [b,d] = circle_circle (a,c) (c,a)
```

Exercise: *Manipulate the program lesson7a.hs to draw a picture with the characteristics of the one below:*



Save the program as yourname_lesson7d.hs

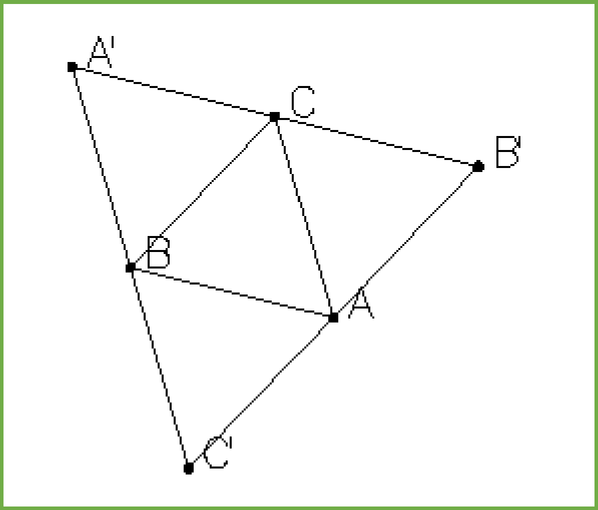
Run the program to check.

Note to Teacher: The `find_apart` *function is used to help students find a point that is not already part of the triangle.*

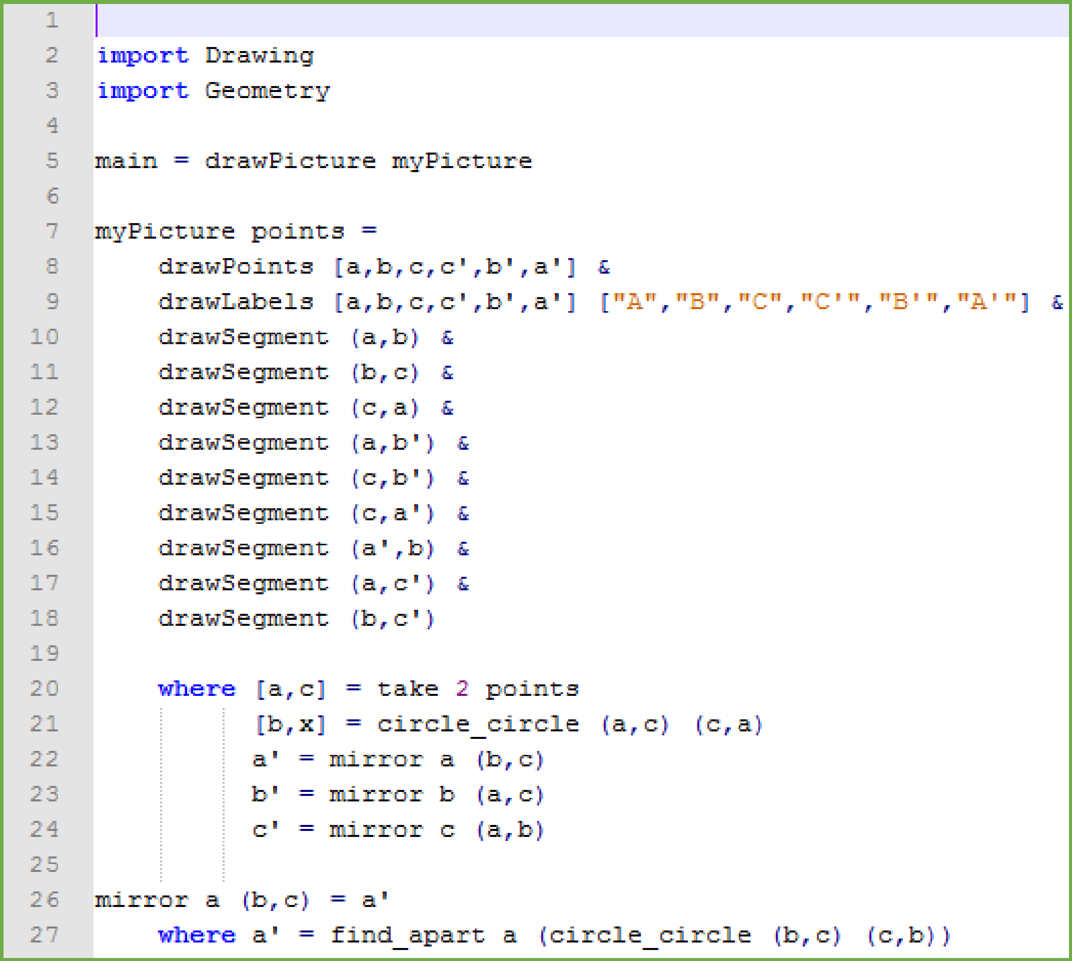
```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =

    drawPoints [a,b,c,c',b',a'] &
    drawLabels [a,b,c,c',b',a'] ["A","B","C","C'","B'","A'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a,b') &
    drawSegment (c,b') &
    drawSegment (c,a') &
    drawSegment (a',b) &
    drawSegment (a,c') &
    drawSegment (b,c')

    where [a,c] = take 2 points
          [b,x] = circle_circle (a,c) (c,a)
          b' = find_apart b [b,x]
          [m,n] = circle_circle (c,b) (b,c)
          a' = find_apart a [m,n]
          [e,d] = circle_circle (a,b) (b,a)
          c' = find_apart c [e,d]
```

Lesson 7 Further Applications:

You can create a mirror function within our program key_lesson7d.hs since the program is repetitive. (key_lesson7e.hs)



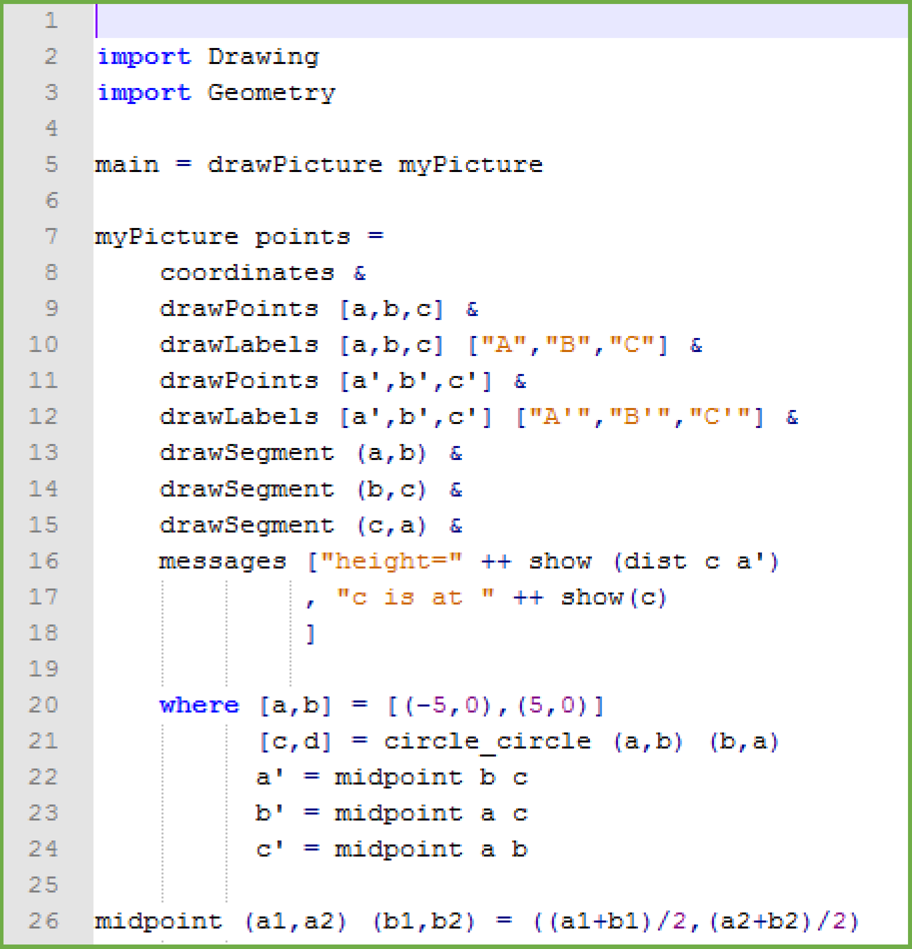
Run the program to check.



```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    drawPoints [a,b,c,c',b',a'] &
    drawLabels [a,b,c,c',b',a'] ["A","B","C","C'","B'","A'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a,b') &
    drawSegment (c,b') &
    drawSegment (c,a') &
    drawSegment (a',b) &
    drawSegment (a,c') &
    drawSegment (b,c')

    where [a,c] = take 2 points
          [b,x] = circle_circle (a,c) (c,a)
          a' = mirror a (b,c)
          b' = mirror b (a,c)
          c' = mirror c (a,b)

mirror a (b,c) = a'
    where a' = find_apart a (circle_circle (b,c) (c,b))
```

Lesson 7: Further Applications:

You can also consider choosing points a and b specifically so that we can find point c or the height. Example shown in key_lesson7N.hs

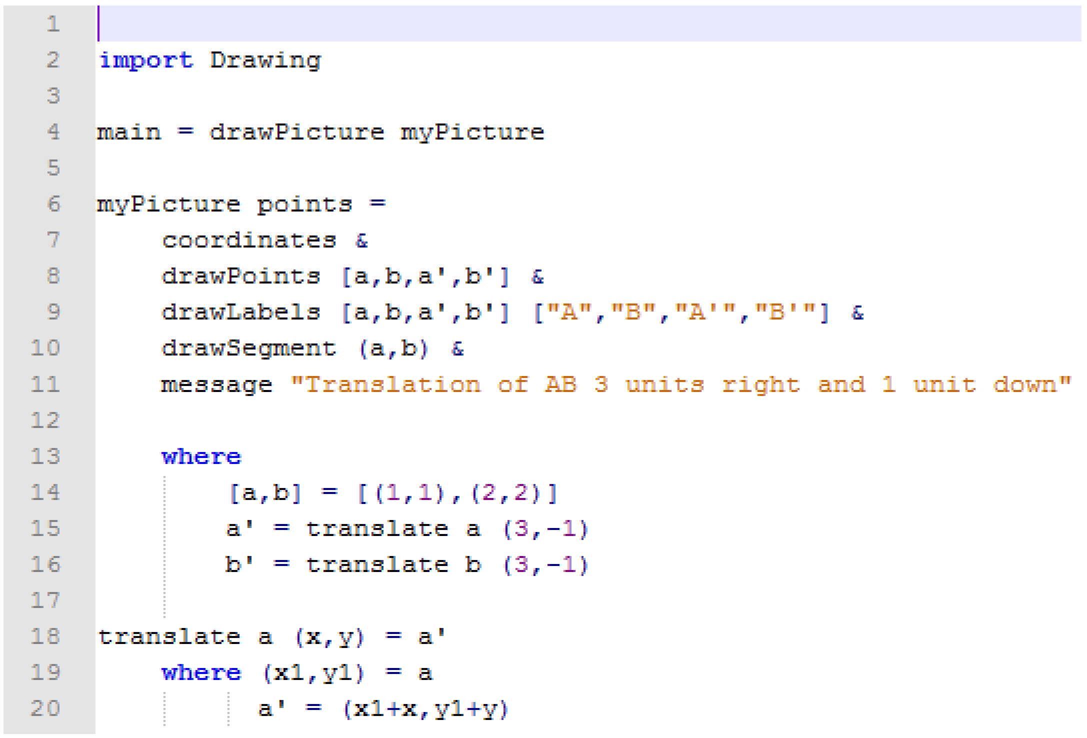


```haskell

import Drawing
import Geometry

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,c] &
    drawLabels [a,b,c] ["A","B","C"] &
    drawPoints [a',b',c'] &
    drawLabels [a',b',c'] ["A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    messages ["height=" ++ show (dist c a')
            , "c is at " ++ show(c)
            ]

    where [a,b] = [(-5,0),(5,0)]
          [c,d] = circle_circle (a,b) (b,a)
          a' = midpoint b c
          b' = midpoint a c
          c' = midpoint a b

midpoint (a1,a2) (b1,b2) = ((a1+b1)/2,(a2+b2)/2)
```

Lesson 8: Transformations

*All exercises should be opened in the editor and ran in the terminal as stated in the introduction.

Lesson 8 Part 1: Open the editor and then open lesson8a.hs. Look at the code and write down what you understand in the code. Run the code.



```haskell

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,a',b'] &
    drawLabels [a,b,a',b'] ["A","B","A'","B'"] &
    drawSegment (a,b) &
    message "Translation of AB 3 units right and 1 unit down"

    where
        [a,b] = [(1,1),(2,2)]
        a' = translate a (3,-1)
        b' = translate b (3,-1)

translate a (x,y) = a'
    where (x1,y1) = a
          a' = (x1+x,y1+y)
```

The teacher should then briefly describe the following characteristics of the code (refer to reference sheet):

`translate`

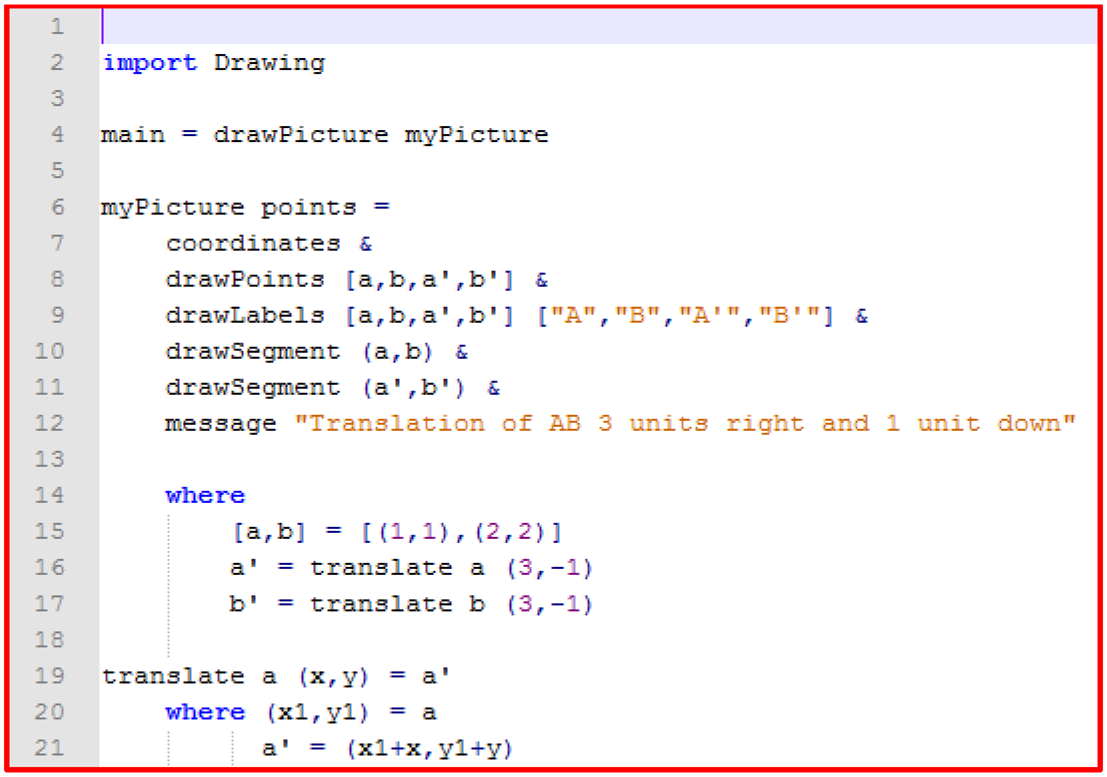
Questions:

1. What function makes the points move?
   translate
2. Why in the translate function is `a’` written as `(x1+x,y1+y)`?
   When you translate points, you add/subtract from the original point.

Exercise: *Manipulate the program lesson8a.hs to draw segment between the translated points.*

Save the program as yourname_lesson8b.hs

Run the program to check.

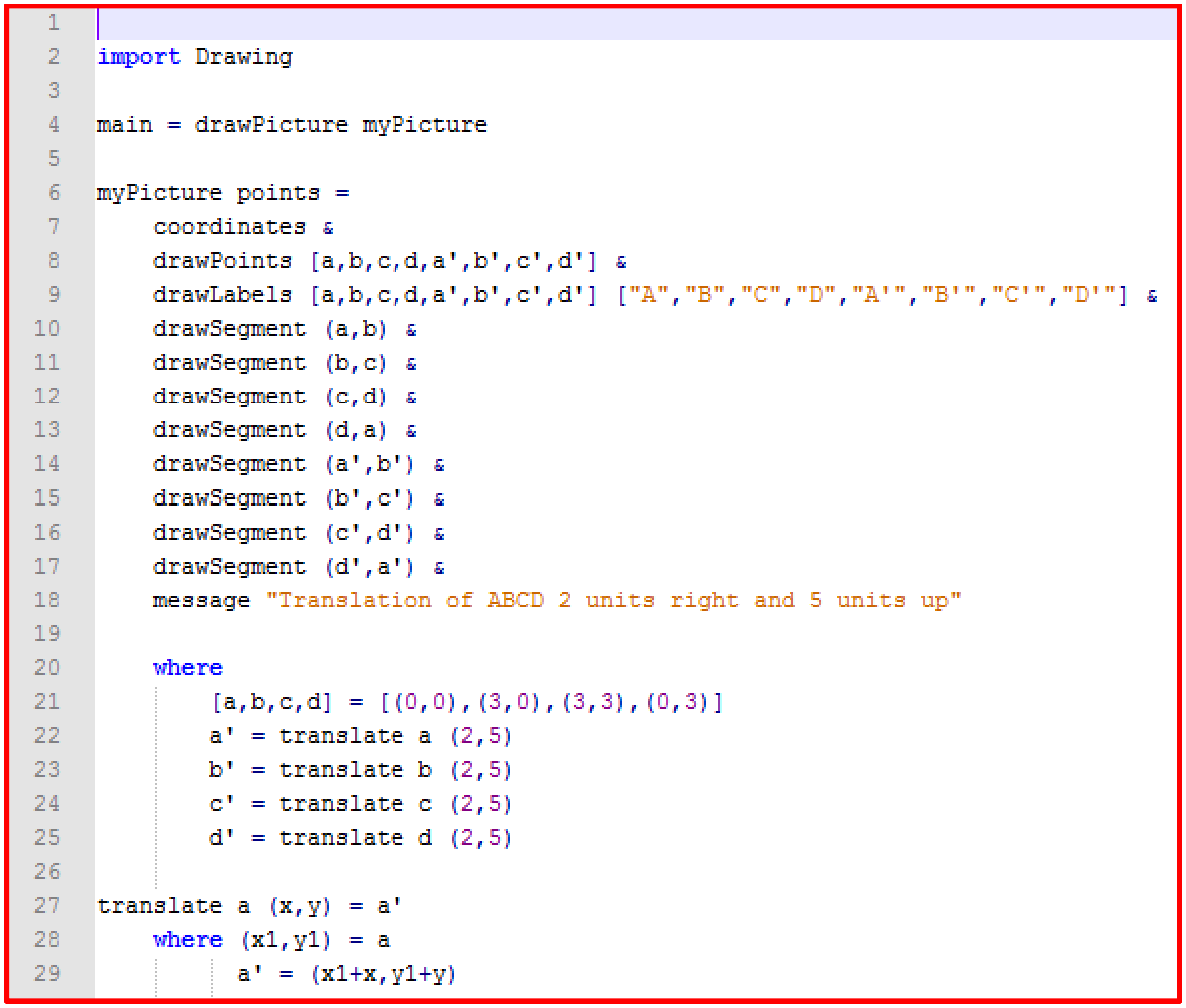


```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,a',b'] &
    drawLabels [a,b,a',b'] ["A","B","A'","B'"] &
    drawSegment (a,b) &
    drawSegment (a',b') &
    message "Translation of AB 3 units right and 1 unit down"

    where
        [a,b] = [(1,1),(2,2)]
        a' = translate a (3,-1)
        b' = translate b (3,-1)

translate a (x,y) = a'
    where (x1,y1) = a
          a' = (x1+x,y1+y)
```

Exercise: *Manipulate the program lesson8b.hs to translate a quadrilateral 2 units right and 5 units up. The quadrilateral should be made of points (0,0), (3,0), (3,3),(0,3).*

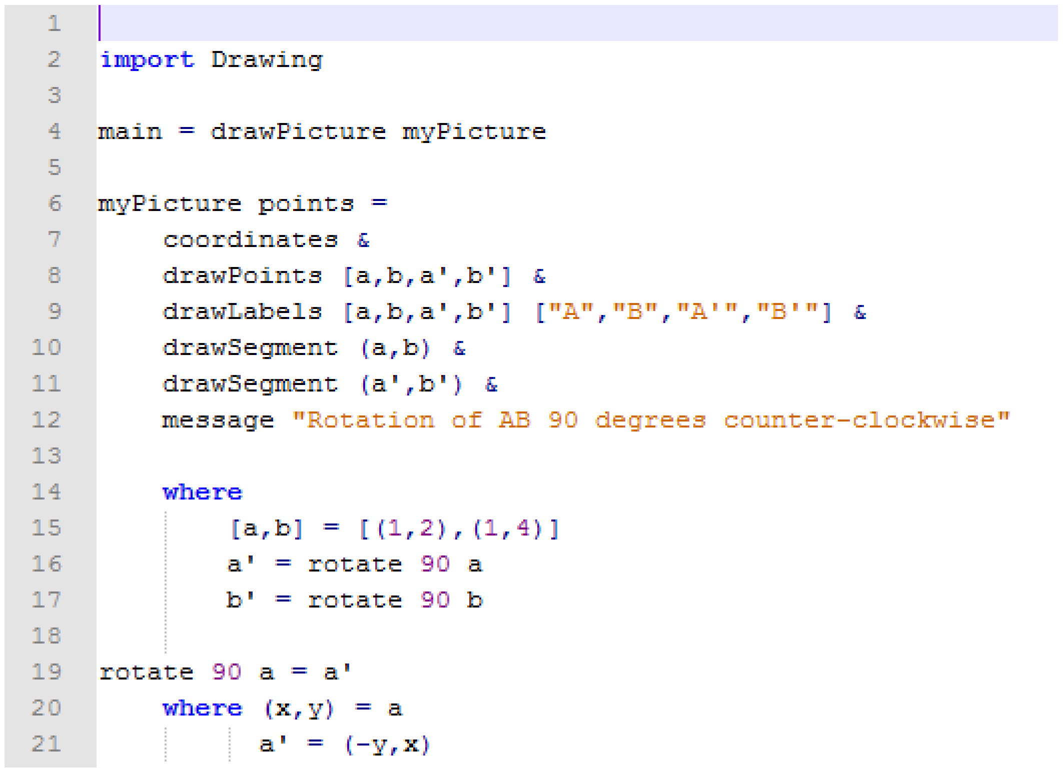
Save the program as yourname_lesson8c.hs

Run the program to check.



```haskell

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,c,d,a',b',c',d'] &
    drawLabels [a,b,c,d,a',b',c',d'] ["A","B","C","D","A'","B'","C'","D'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,d) &
    drawSegment (d,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',d') &
    drawSegment (d',a') &
    message "Translation of ABCD 2 units right and 5 units up"

    where
        [a,b,c,d] = [(0,0),(3,0),(3,3),(0,3)]
        a' = translate a (2,5)
        b' = translate b (2,5)
        c' = translate c (2,5)
        d' = translate d (2,5)

translate a (x,y) = a'
    where (x1,y1) = a
          a' = (x1+x,y1+y)
```

Lesson 8 Part 2: Open the editor and then open lesson8d.hs



```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,a',b'] &
    drawLabels [a,b,a',b'] ["A","B","A'","B'"] &
    drawSegment (a,b) &
    drawSegment (a',b') &
    message "Rotation of AB 90 degrees counter-clockwise"

    where
        [a,b] = [(1,2),(1,4)]
        a' = rotate 90 a
        b' = rotate 90 b

rotate 90 a = a'
    where (x,y) = a
          a' = (-y,x)
```

Exercise: *Determine what the program will draw.*

Run the program to check.

The teacher should then briefly describe the following characteristics of the code (refer to reference sheet):

`Rotate`

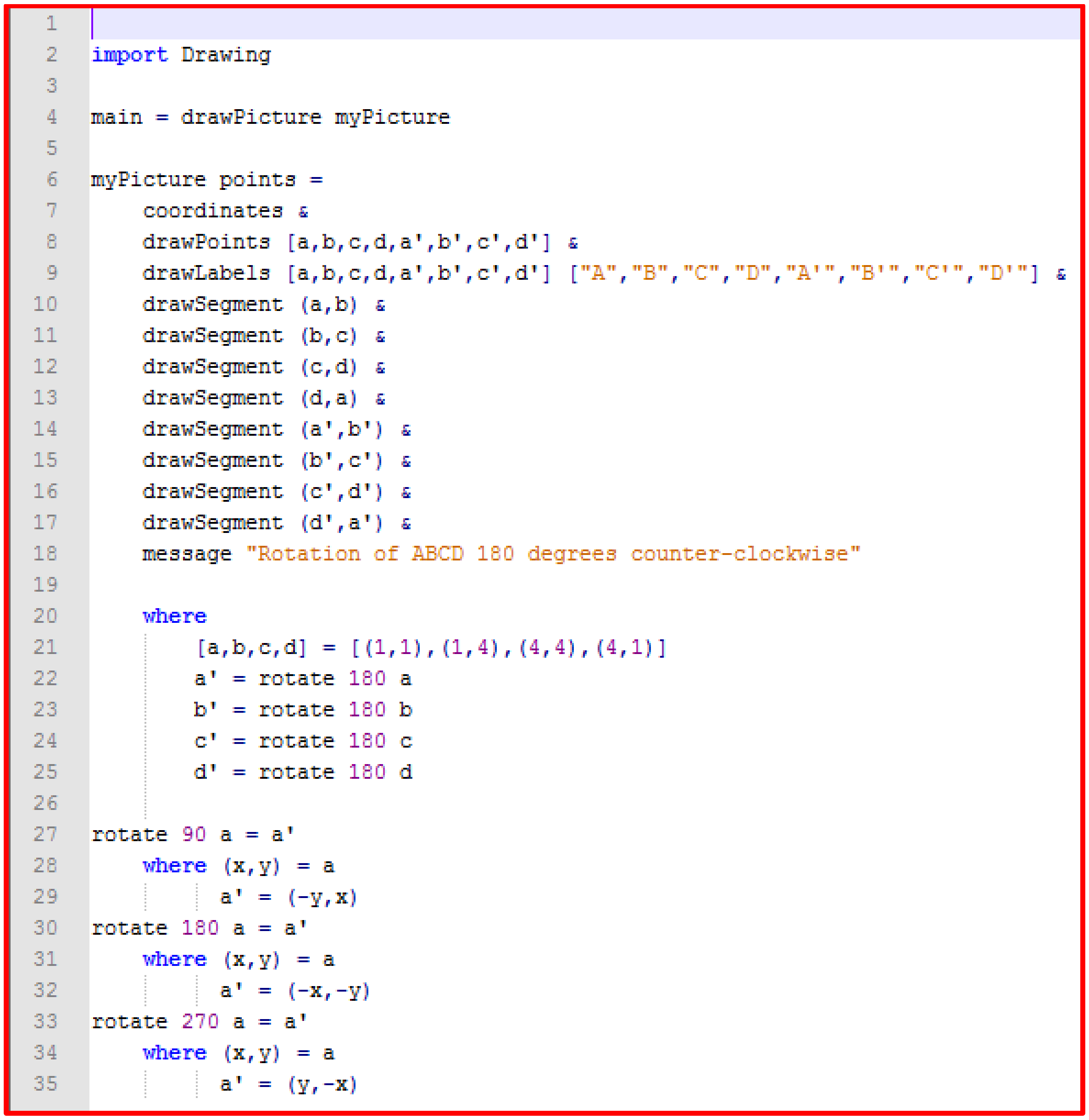
Question:

1. Why is `a'  =  (-y,x)`?
   When you rotate 90 degrees counter-clockwise, the x and y switch and the new x value becomes opposite.
2. What would happen if you rotate (9,8) 180 degrees counter-clockwise?
   The x and y would both become opposite. (-9,-8)

Exercise: *Manipulate the program lesson8d.hs that will rotate a quadrilateral 90, 180, or 270 degrees. The program should have all three functions at the bottom. Write this specific program to rotate the quadrilateral (1,1), (1,4), (4,4), (4,1) 180 degrees counter-clockwise.*

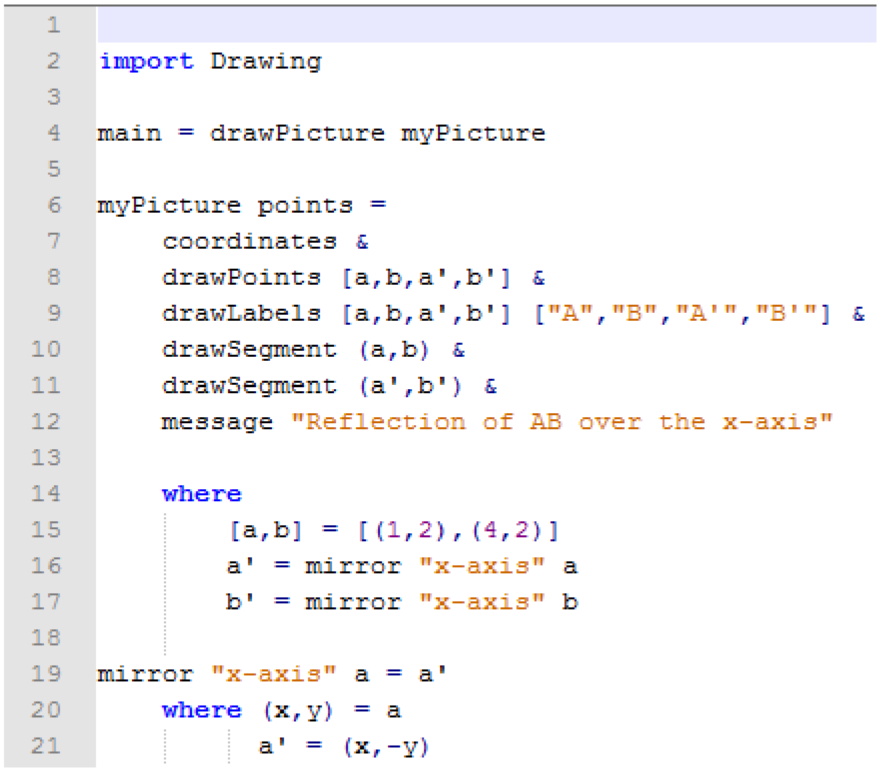
Save the program as yourname_lesson8e.hs

Run the program to check.



```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,c,d,a',b',c',d'] &
    drawLabels [a,b,c,d,a',b',c',d'] ["A","B","C","D","A'","B'","C'","D'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,d) &
    drawSegment (d,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',d') &
    drawSegment (d',a') &
    message "Rotation of ABCD 180 degrees counter-clockwise"

    where
        [a,b,c,d] = [(1,1),(1,4),(4,4),(4,1)]
        a' = rotate 180 a
        b' = rotate 180 b
        c' = rotate 180 c
        d' = rotate 180 d

rotate 90 a = a'
    where (x,y) = a
          a' = (-y,x)
rotate 180 a = a'
    where (x,y) = a
          a' = (-x,-y)
rotate 270 a = a'
    where (x,y) = a
          a' = (y,-x)
```

Lesson 8 Part 3: Open the editor and then open lesson8f.hs.



```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,a',b'] &
    drawLabels [a,b,a',b'] ["A","B","A'","B'"] &
    drawSegment (a,b) &
    drawSegment (a',b') &
    message "Reflection of AB over the x-axis"

    where
        [a,b] = [(1,2),(4,2)]
        a' = mirror "x-axis" a
        b' = mirror "x-axis" b

mirror "x-axis" a = a'
    where (x,y) = a
            a' = (x,-y)
```

Questions:

1. Why is the function mirrors used as a transformation of reflection?
   A reflection is a mirrored image.
2. Why is `a' = (x, -y)`?
   A reflection over the x-axis only changes the y value of the original point.

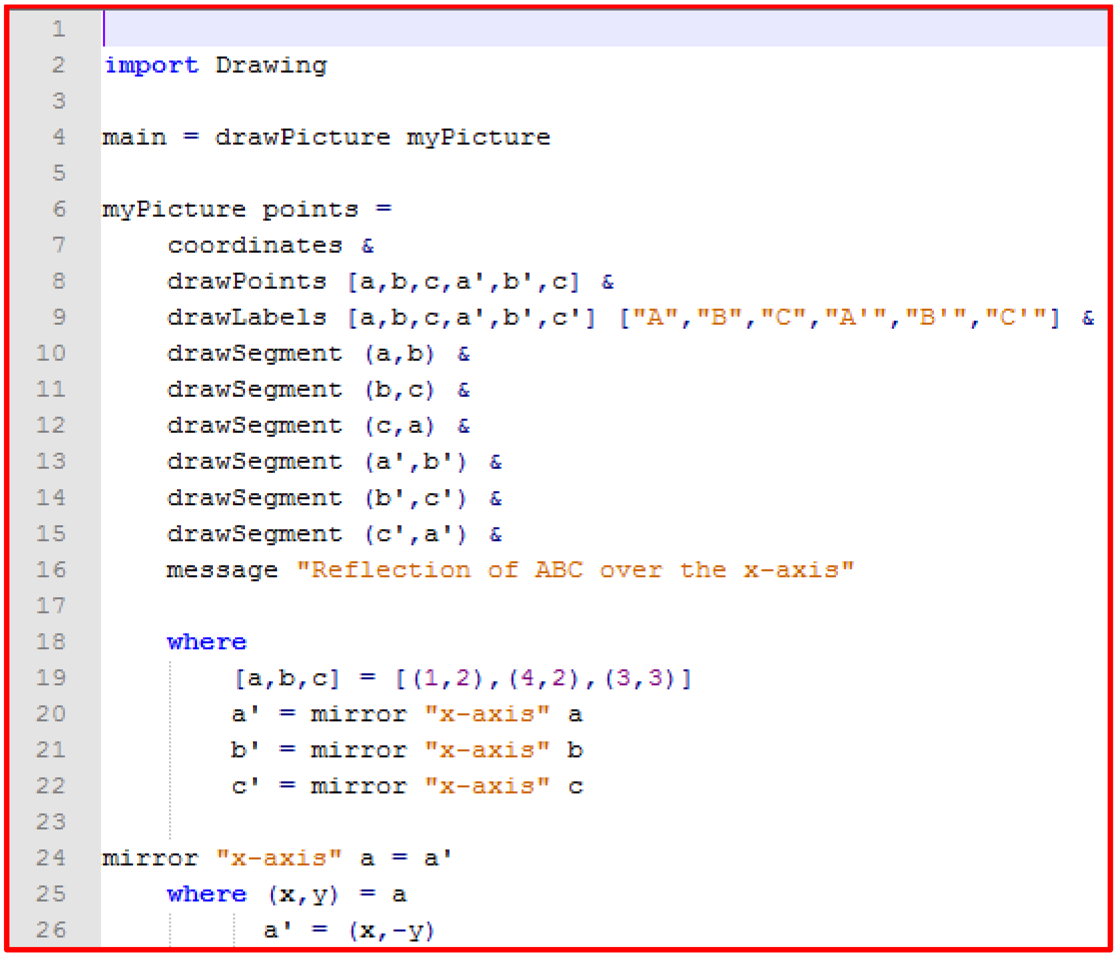
The teacher should then briefly describe the following characteristics of the code (refer to reference sheet):

`mirror`

Exercise: *Manipulate the program lesson8f.hs to reflect triangle (1,2),(4,2),(3,3) over the x-axis.*

Save the program as yourname_lesson8g.hs

Run the program to check your answer.

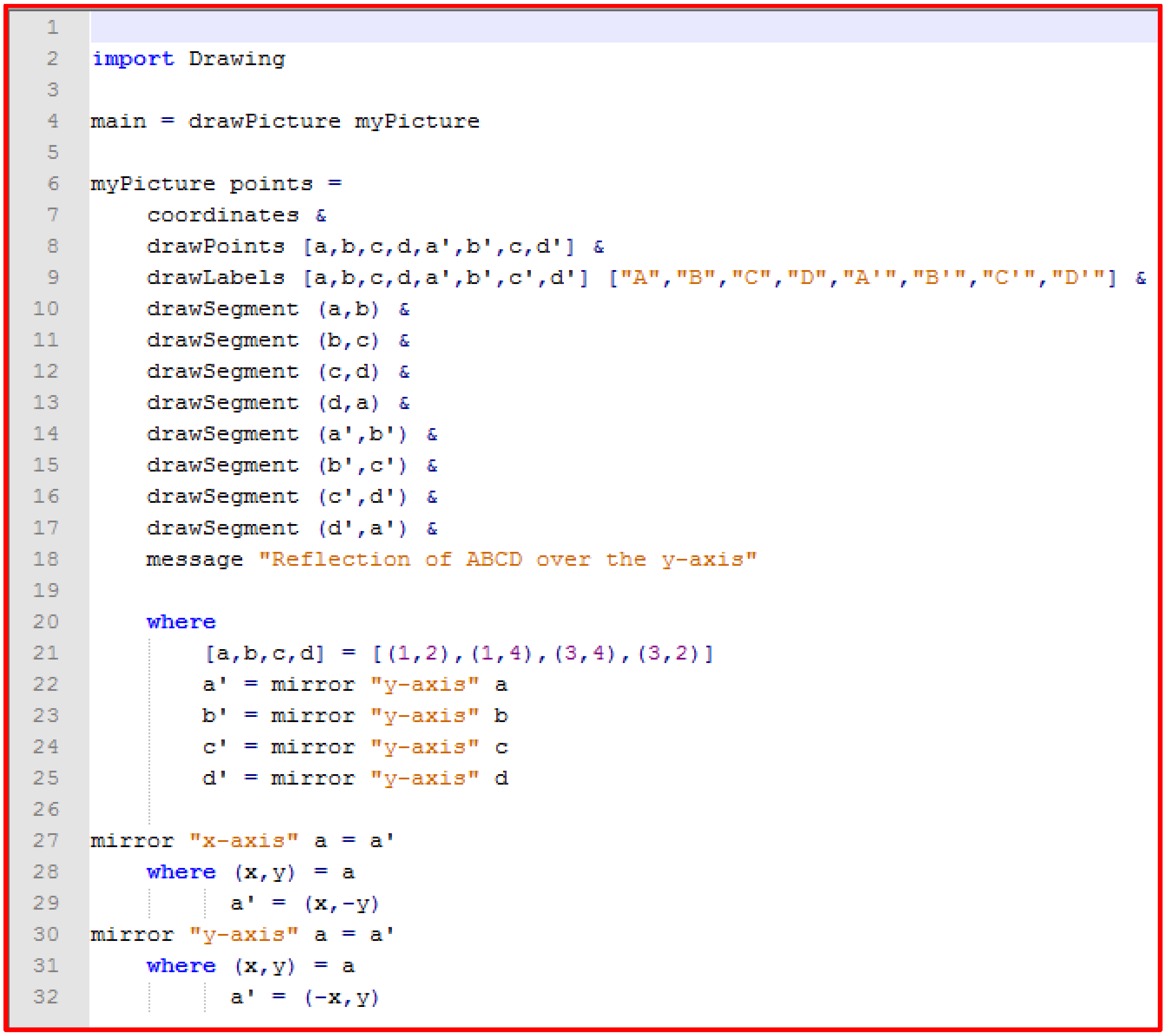


```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,c,a',b',c] &
    drawLabels [a,b,c,a',b',c'] ["A","B","C","A'","B'","C'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',a') &
    message "Reflection of ABC over the x-axis"

    where
        [a,b,c] = [(1,2),(4,2),(3,3)]
        a' = mirror "x-axis" a
        b' = mirror "x-axis" b
        c' = mirror "x-axis" c

mirror "x-axis" a = a'
    where (x,y) = a
          a' = (x,-y)
```

Exercise: *Manipulate the program lesson8g.hs that could reflect a quadrilateral over the x or y axis. Both mirrors functions should be listed at the bottom of the program. For this specific program reflect the quadrilateral (1,2), (1,4), (3,4), (3,2) over the y-axis.*

Save the program as yourname_lesson8h.hs

Run the program to check your answer.



```haskell

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawPoints [a,b,c,d,a',b',c,d'] &
    drawLabels [a,b,c,d,a',b',c',d'] ["A","B","C","D","A'","B'","C'","D'"] &
    drawSegment (a,b) &
    drawSegment (b,c) &
    drawSegment (c,d) &
    drawSegment (d,a) &
    drawSegment (a',b') &
    drawSegment (b',c') &
    drawSegment (c',d') &
    drawSegment (d',a') &
    message "Reflection of ABCD over the y-axis"

    where
        [a,b,c,d] = [(1,2),(1,4),(3,4),(3,2)]
        a' = mirror "y-axis" a
        b' = mirror "y-axis" b
        c' = mirror "y-axis" c
        d' = mirror "y-axis" d

mirror "x-axis" a = a'
    where (x,y) = a
          a' = (x,-y)
mirror "y-axis" a = a'
    where (x,y) = a
          a' = (-x,y)
```

Lesson 8 Ending Exercises:

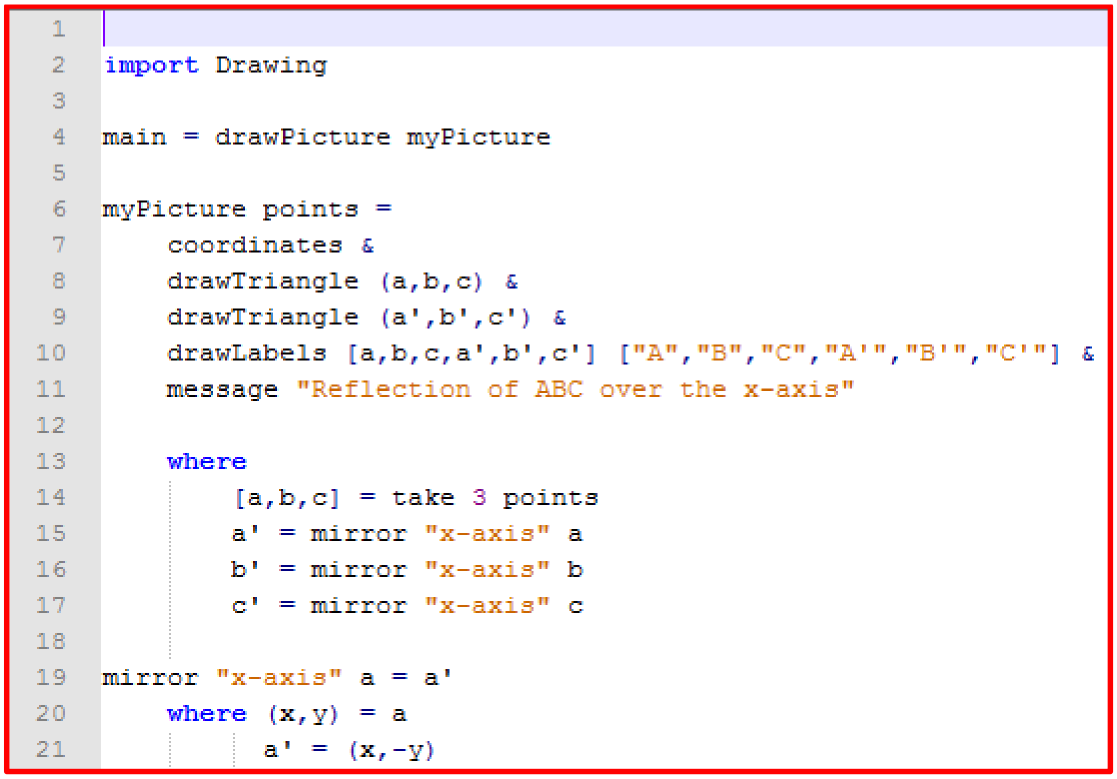
Exercise:

Question: How can we draw a random triangle in a program?

Manipulate yourname_lesson8g.hs to reflect a random triangle over the x-axis.

Save the program as yourname_lesson8i.hs.

Run the Program to check your answer.



```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawTriangle (a,b,c) &
    drawTriangle (a',b',c') &
    drawLabels [a,b,c,a',b',c'] ["A","B","C","A'","B'","C'"] &
    message "Reflection of ABC over the x-axis"

    where
        [a,b,c] = take 3 points
        a' = mirror "x-axis" a
        b' = mirror "x-axis" b
        c' = mirror "x-axis" c

mirror "x-axis" a = a'
    where (x,y) = a
          a' = (x,-y)
```

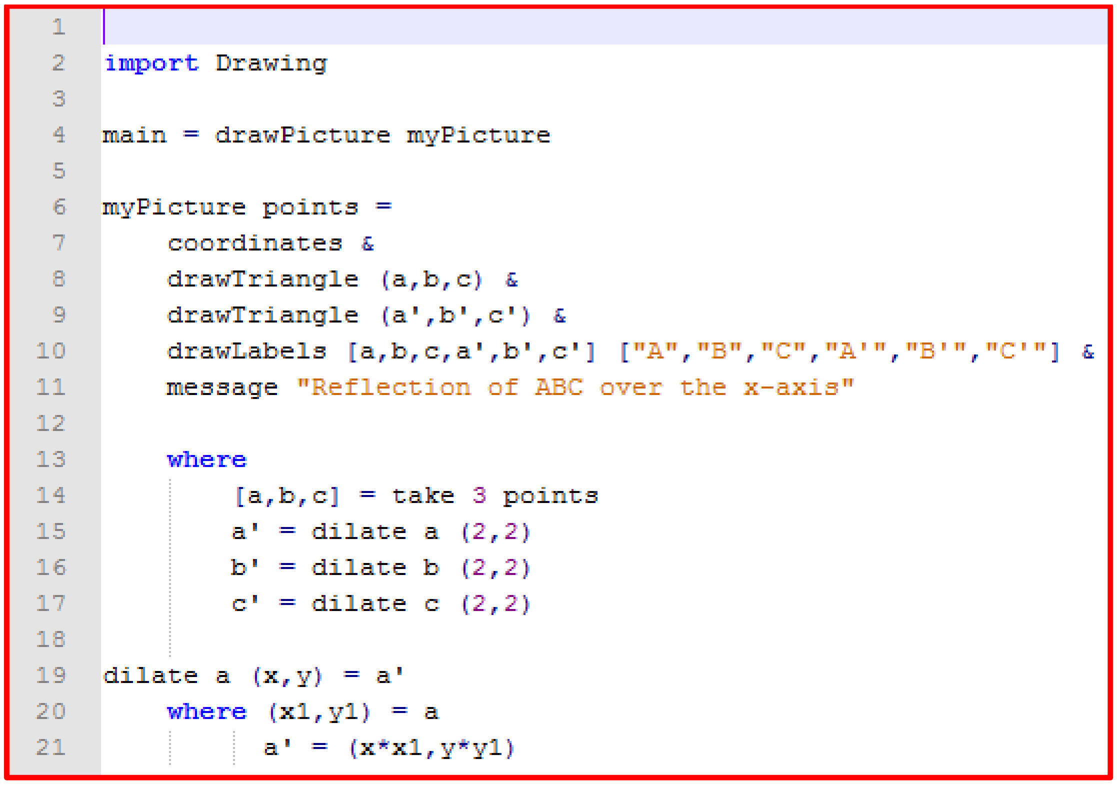
Exercise:

Question: How do we perform a dilation on a figure?

Manipulate yourname_lesson8g.hs to dilate the triangle by a scale factor of 2 rather than reflect the triangle.

Save the program as yourname_lesson8j.hs

Run the program to check your answer.



```

import Drawing

main = drawPicture myPicture

myPicture points =
    coordinates &
    drawTriangle (a,b,c) &
    drawTriangle (a',b',c') &
    drawLabels [a,b,c,a',b',c'] ["A","B","C","A'","B'","C'"] &
    message "Reflection of ABC over the x-axis"

    where
        [a,b,c] = take 3 points
        a' = dilate a (2,2)
        b' = dilate b (2,2)
        c' = dilate c (2,2)

dilate a (x,y) = a'
    where (x1,y1) = a
          a' = (x*x1,y*y1)
```

Name: ______________________________ Date: __________ Class: __________

Coding Project

Create a program using 10 functions with at least one function from each lesson we have covered. Refer to your handouts, notes, programs, and reference sheet for help. You should open file project.hs in Notepad++ to help you get started. Before you begin, remove the word `undefined` from the program.

You will be graded based on required elements, content-accuracy, organization, and creativity/originality. Consult the rubric for more information.

Save your project as yourname_project.hs

Name: ______________________________ Date: __________ Class: __________

Coding Project

Create a program using 10 functions with at least one function from each lesson we have covered. Refer to your handouts, notes, programs, and reference sheet for help. You should open file project.hs in Notepad++ to help you get started. Before you begin, remove the word `undefined` from the program.

You will be graded based on required elements, content-accuracy, organization, and creativity/originality. Consult the rubric for more information.

Save your project as yourname_project.hs