COPRO

White Paper

CS3423 : Compilers - 2

Prof. Dr.Ramakrishna Upadrasta

Team

CS20BTECH11064	**Thota Rohan**	Project Manager
CS20BTECH11037	**Patnala Vikas**	Integrator
CS20BTECH11020	**Itreddy RajaSekhar Reddy**	Guru
CS20BTECH11048	**Songa Kotesh Satvik**	Architect
CS20BTECH11058	**Jumala Vamsi Preetham**	Tester
CS20BTECH11007	**Revanth Badavathu**	Architect
CS20BTECH11019	**Gyara Yagnavalkya**	Tester

Contents

1. Introduction to CoPro

1.1 Motivation

The study of conic sections is important not only for mathematics, physics, and astronomy, but also for a variety of engineering applications. The smoothness of conic sections is an important property for applications such as aerodynamics and much more.

Solving conics is not as easy as it seems to be. In case of complex conic sections, you have to write down every calculation to not make a mistake as even a minimal error might be fatal when in case of aerodynamics design, engineering applications, etc. That's where CoPro comes in. Our language solves the problems faster which saves you time and there won't be any chance of error, as it is a program. CoPro also comes in handy when you have to solve multiple problems or same problems repeatedly for a purpose. It's a tiring process when done manually but since it's a program, CoPro will save you a lot of time.

1.2 CoPro

CoPro is a domain specific language which has been designed to help scientists and engineers solve conic section problems. This language solves conic problems faster and easier saving users time and resources.

We have designed this language in such a way that any user with even a minimal amount of knowledge of conic sections can use CoPro to solve conic section problems with the help of in-built libraries, data types, expressions, etc.

1.3 Design Goals

Simple and Familiar: CoPro is explicitly designed to solve conic problems. As mentioned above, this langauge is so simple to use that any user who has the basic minimal knowledge of conic sections can use CoPro to get almost every characteristic of any conic section. Even users with no programming language could easily adapt to the language.

Robust and Efficiency: One could write highly reliable code using CoPro. Since CoPro has been designed to solve conic problems, we would also try to make it highly efficient and keep updating it from time to time depending on the demanding use.

Libraries: Functions provided in the libraries would be extremely helpful for anyone writing in CoPro. A lot of inbuilt functions perform many functionalities required of Conic sections, so that the programmer could use them directly to write bigger functions.

2. Features of CoPro Language

2.1 Data Types

2.1.1 Primitive Data Types

Data Types	Description
int	64-bit signed integer value
double	64-bit signed floating point value
bool	Stores true or false values
string	A collection of alphanumeric characters. A string must be enclosed within double quotes
frac	Rational representation of a number $\frac{p}{q}$ form (q≠0)

Table 2.1: Primitive Data Types

2.1.2 Non-Primitive Data Types

Datatype	Condition	Description
point	$\Delta = 0$, $h = 0$ and $ab > 0$	Euclidian co-ordinate representation of position.
line	$\Delta = 0$, $h^2 = ab$ and $gf = hc$	One dimensional figure with length but no width.
conic	$ax^2 + 2hxy + by^2 + 2gx + 2gy + c = 0$	A super class definition of all conic equations.
line_pair	$\Delta = 0$ and $ab - h^2 \leq 0$	Product of two real linear equations each representing a straight line.
circle	$h = 0$ and $a = c$	Locus of all points equiv-distant from a given point at a distance r.
parabola	$\Delta \neq 0$ and $h^2 = ac$	A curve-distant from a fixed point and fixed line.
ellipse	$\Delta \neq 0$ and $h^2 < ac$	A curve whose distance from two points adds up to a fixed constant.
hyperbola	$\Delta \neq 0$ and $h^2 > ac$	A curve whose absolute difference of distances from two fixed points is always constant.

Table 2.2: Non-Primitive Data Types

2.2 Syntax of CoPro Language

2.2.1 Comments

Only Single line comments are supported in CoPro.
Every character after the '#' symbol on a line are considered as a comment by the compiler.

```
# single line comment
# the below function returns the sum of two int
func : int b, int c -> int
'
    exit b+c
'
```

2.2.2 Whitespace

Whitespace is ignored usually. But there are some cases where we need to seperate the tokens with space or tabs cause otherwise they would become a single token which is useless to us.
Tabs, indentation, space and comments all comes under whitespace.

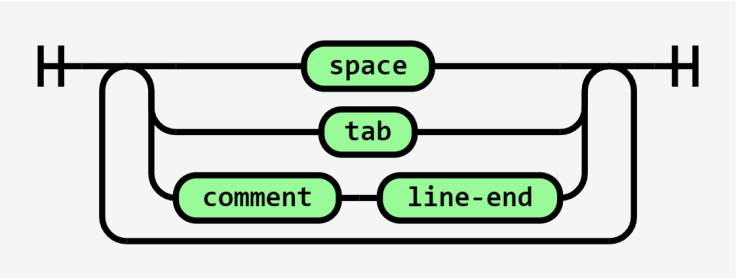

Figure 2.1: Rail-Road Diagram for Whitespace

2.2.3 Keywords

void	int	bool	double	point	string
line	conic	loop	circle	parabola	ellipse
if	frac	true	false	line_pair	hyperbola
else	input	output	break	continue	

2.2.4 Name

Name is a letter optionally followed by one or more letters, digits, or underscores. Name are used for statements, variables, parameters, operators and labels. Name cannot be one of these keywords.

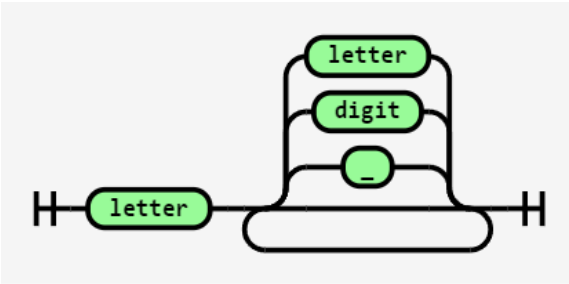

Figure 2.2: Rail-Road Diagram for Names

2.2.5 Numbers

CoPro language supports 2 Number types(int, double). *int* is a 64-bit signed integer value. *double* is 64-bit signed floating point value.

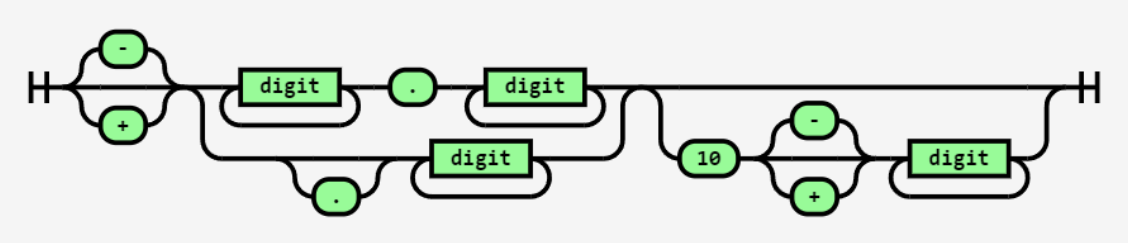

Figure 2.3: Rail-Road Diagram for Numbers

2.3 Literals

Number literals are char-string made of digits 0-9. A number literal with no decimal point is an integer. String literal is a sequence of characters from the source character set(a-z, A-Z) enclosed in double quotation marks (" ").

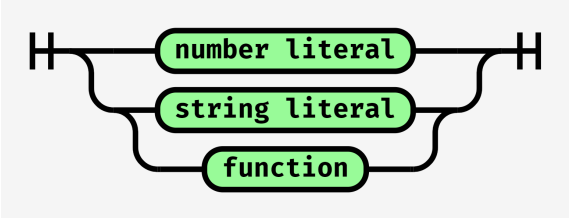

Figure 2.4: Literals

2.3.1 Strings

A string is a sequence of characters. It is wrapped within double quotes(""). It can contain zero or more characters. The \(backslash) is the escape character. If there is a (") in the string, we can use this to consider it as a part of the string. There is no char in CoPro. If you want to represent + a char, just define a string of single character.

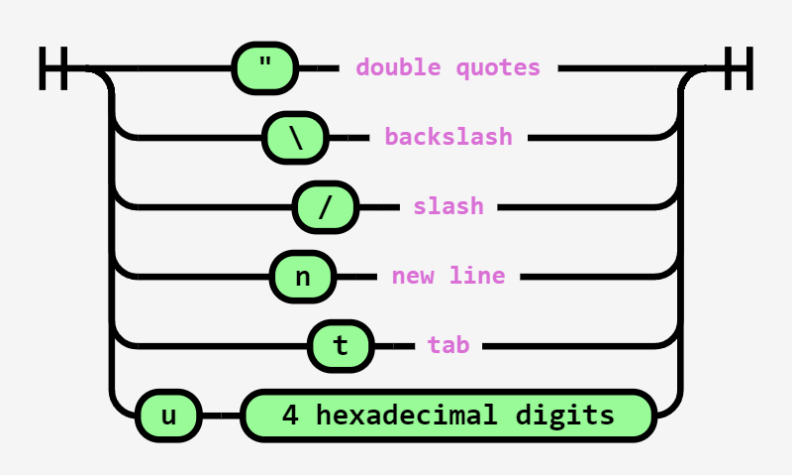

Figure 2.5: Rail-Road Diagram for Escape characters

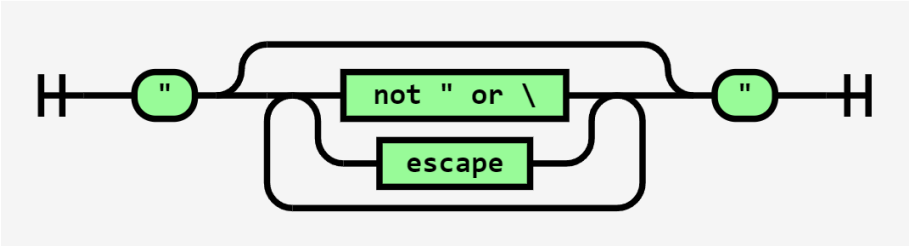

Figure 2.6: Rail-Road Diagram for Strings

2.4 Operators

Operators are one of the features in which has symbols that can be used to perform mathematical, relational, bitwise, conditional, or logical manipulations.

2.4.1 Precedence

The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

Operator	Description	Associative
()	parentheses	left to right
.	member access	left to right
!	unary negation	right to left
^	exponent	left to right
%	modulo	left to right
*	multiplication	left to right
/	division	left to right
+	addition	left to right
-	subtraction	left to right
== , !=	relational operators	left to right
<= , < , >= , >		left to right
&& , | |	logical operators	left to right
= , *= , +=	assignment operators	right to left

Table 2.3: Operators Precedence

2.5 Declarations

As with any other programming language, CoPro also has variables, functions etc. The programmer can utilize any of these only if he has declared them before. We consider two types of declarations:

2.5.1 Variable Declaration

All the variables to be used in the program must be declared by the programmer in a statement until he ends the line.

```
int x = 3, y, z
double e
double p = 3.5
string s
frac fr = 3/5
parabola para: 0, 0, 1, -4, 0, 0
ellipse elp: 3, 0, 5, 0, 0, 15
```

```
add : int b, int c ->int
```

2.5.2 Function Declaration

Functions have to be declared before using them in CoPro. A function doesn't necessarily need parameters. However, it needs to have a return type.

```
func_name : datatype_1 d1, datatype_2 d2 -> datatype_r
```

2.5.3 Initialization

The programmer has an option of initializing his variables for primitive data types(except frac). However, for non-primitive data types (and frac), the programmer has to compulsorily initialize the variables.

Default Initialization: This refers to the case when variables are declared but not initialized. This case holds only for primitive data types(except frac). The string data type is initialized with an empty string, while the other three are initialized to zero.

```
int x                   # x=0
double y                # y=0.0
bool z                  # z=0
string s                # s="abc"
```

Direct Initialization: This refers to the case when non-primitive data types (and frac) variables are declared and initialized. The point variable takes in two parameters to initialize as its x and y co-ordinates, frac takes in two parameters to initialize as its numerator and denominator. All the other non-primitive data types take in six parameters as each of the coefficients of x^2, xy, y^2, x, y, c.

```
point pt: 3, 5
# Creates a point with x and y co-ordinates 3 and 5
frac fr: 1, 3
# Creates a fraction with value 1/3.
parabola para: 0, 0, 1, -4, 0, 0
# Creates a parabola with equation y^2 = 4x
ellipse elp: 3, 0, 5, 0, 0, 15
# Creates an ellipse with equation 3x^2 + 5y^2 + 15 = 0
```

2.6 Statements

Statements tend to be executed in order from top to bottom. The sequence of execution can be altered by the conditional statements (if and else), by the looping statements (loop), by the disruptive statements (break and exit), and by function invocation.
The if statement changes the flow of the program based on the value of the expression. The block is executed if the expression is truthy; otherwise, the optional else branch is taken.

2.6.1 Disruptive Statements

Figure 2.7: Rail-Road Diagram for disruptive statement

These statements indicate the abrupt/complete finishing of particular statement blocks.
Break Statement: This statement is used when the programmer wants to abruptly exit out of *loop*.

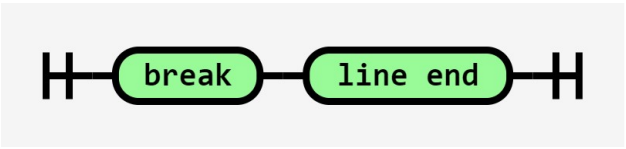

Figure 2.8: Rail-Road Diagram for break statement

It is most commonly used inside if/else statements.

```
.
.
break
# exits a loop .
.
```

Exit Statement: This statement is used when the programmer wants to exit out of a function. Exit is followed by an expression which has an output data-type same as the return type of the function.

Figure 2.9: Rail-Road Diagram for exit statement

```
.
.
exit a+b
# exits a loop .
.
```

2.6.2 If statements

The If/else conditions are used to impact the control flow of the execution. If is followed by an expression which generally has an output of 0/1. There is a block of statements following the expression which are implemented only if the expression evaluates to true.

```
int a = 3
if(a == 3)
'
    a = 4
'
if(a == 5)
'
    a = 6
'
else
'
    a = 7
'
# value of a : 7
```

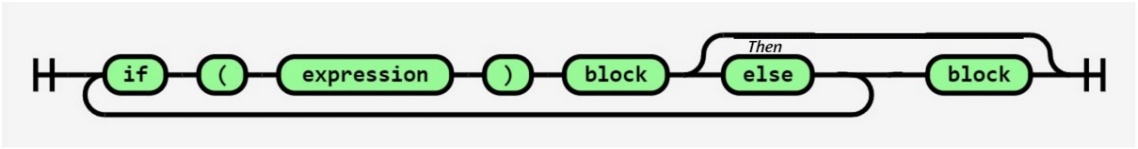

Figure 2.10: Rail-Road Diagram for if statement

2.6.3 Loop Statements

Loop statement consists of a condition followed by a statement block. The statements are repeatedly evaluated as long as the condition remains True before each repetition. Note that assigning over the looping variable will not change the original variable in the scope.
Continue statement: The continue statement skips the current iteration of the loop and continues with the next iteration. The continue statement is almost always used with the if...else statement.

```
int a = 3, b = 17
loop(a < 10 && b > 15)
'
    a += 1
    b = b-1
    if(b != 16)
    '
        continue
    '
'
```

Figure 2.11: Rail-Road Diagram for loop statement

2.6.4 Block Statement

This refers to a block of statements wrapped in single quotes which can be used inside functions, if statement blocks and loop statement blocks.

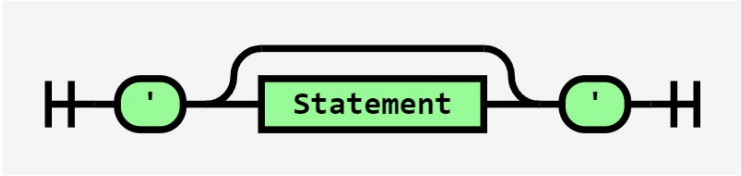

Figure 2.12: Rail-Road Diagram for block statements

2.7 Expressions

2.7.1 Expression

An expression is a combination of numbers , variables , functions. It is similar to a phrase in a language.It is different from an equation which is a combination of two expressions.

```
int a = 3 + 5
int a = 9, b = 6, c = a * b
a + b + c = 8
```

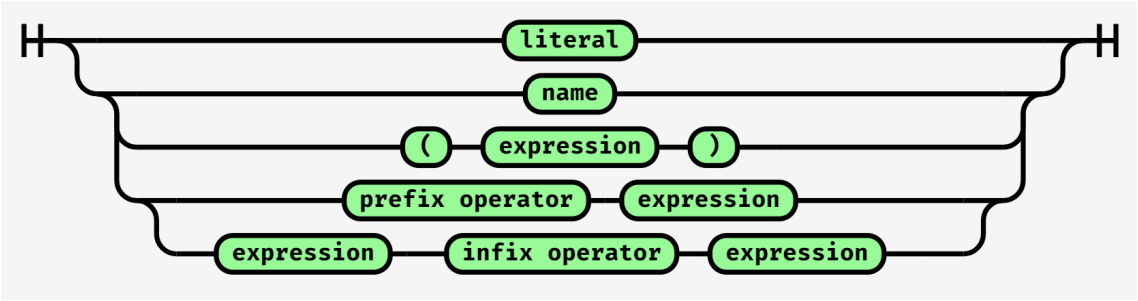

Figure 2.13: Expression

2.7.2 Prefix Operator

A prefix operator is used generally to increment, negate or its data type declaration.

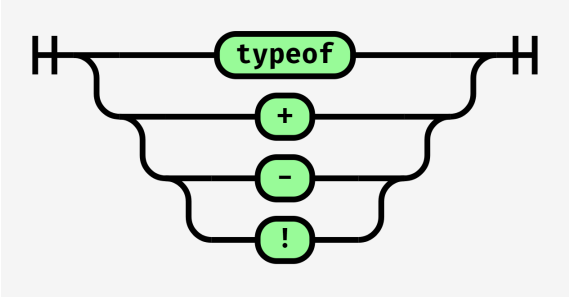

Figure 2.14: Prefix Operator

2.7.3 Infix Operator

The infix operator is used to implement all types of mathematical operations.

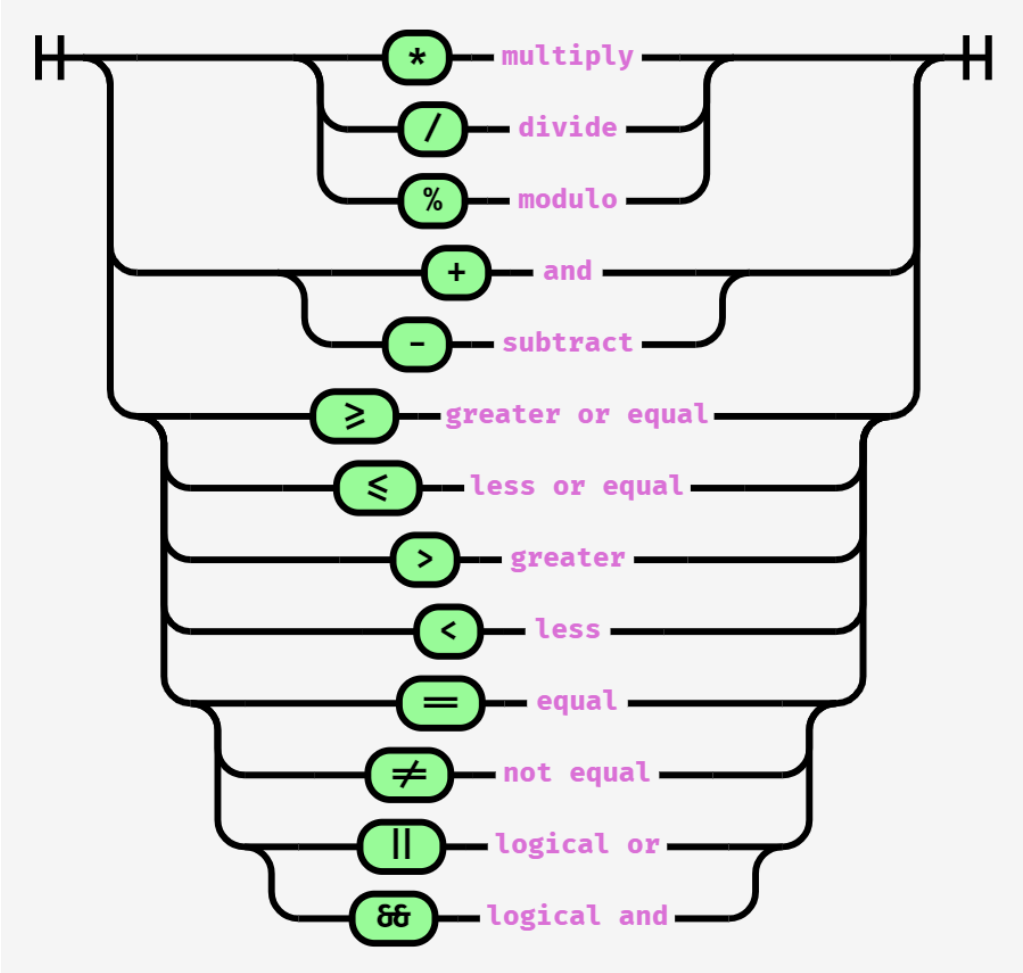

Figure 2.15: Infix Operator

3. Standard Libraries

We aim to implement all of the conic functions mentioned below that would make programmers life easy.

3.1 Basic functions

- **eq** : Returns a string in the form of general equation $ax^2+by^2+2hxy+2gx+2fy+c$.
- **a** : Returns the value of a in general form.
- **b** : Returns the value of b in general form.
- **h** : Returns the value of h in general form.
- **g** : Returns the value of g in general form.
- **f** : Returns the value of f in general. form
- **c** : Returns the value of c in general. form
- Δ : Returns the value $abc+2fgh-af^2-bg^2-ch^2$ which is used to classify the type of conic.
- **on_curve** : It takes a point as a parameter and returns true when the point lies on the conic and false when the point is not on the conic.

3.2 Point

- **x** : Returns the x co-ordinate of the point.
- **y** : Returns the y co-ordinate of the point.

3.3 Line

- **slope** : Returns the slope of the line.

3.4 Pair of Straight Lines

- **acute_ang_bisecs** : Returns the equations of acute angle bisectors for the pair of lines.
- **obtuse_ang_bisecs** : Returns the equations of obtuse angle bisectors for the pair of lines.
- **intersec_pt** : Returns the intersection point of the pair of lines.
- **angle_** : Returns the angle between the pair of lines at the intersection point.
- **slopes** : Returns the slopes of both the pair of lines.

3.5 Common for the below Conics

- **centre** : Returns the centre of the conic.
- **tangent** : It takes the point as a parameter and returns tangent of that point on the conic.
- **normal** : It takes the point as a parameter and returns normal of that point on the conic.

3.6 Circle

- **radius** : Returns the radius of the circle.
- **circumf** : Returns the circumference of the circle.
- **area**: Returns the area enclosed by the circle.

3.7 Common for the below Conics

- **focii** : Returns the focii of the conic.
- **direct** : Returns the directrices of the conic.
- **eccen** : Returns the eccentricity of the conic.
- **p_axis** : Returns the prinicpal axis of the conic.
- **vertices** : Returns the vertices of the conic.

3.8 Parabola

- **focal_chord** : It takes a point as parameter and returns the chord passing through the point and the focus of the parabola.
- **double_ordinate** : It takes a point as parameter and returns the length of chord passing through the point which is parallel to the directrix.
- **latus_rectum** : Returns the latus rectum of the parabola.

3.9 Ellipse

- **len_maj_axis** : Returns the length of major axis of the ellipse.
- **len_min_axis** : Returns the length of minor axis of the ellipse.
- **latus_recta** : Returns the equations of the latus rectus of the ellipse.
- **latus_rectum** : Returns the latus rectum of the ellipse.
- **focal_radii** : Returns the focal radii of the ellipse.
- **dir_circle** : Returns the equation of director circle of the ellipse.
- **aux_circle** : Returns the equation of auxiliary circle of the ellipse.
- **circumf** : Returns the circumference of the ellipse.
- **area**: Returns the area enclosed by the ellipse

3.10 Hyperbola

- **trans_axis** : Returns the equation of the transverse axis of the hyperbola.
- **conj_axis** : Returns the equation of the conjugate axis of the hyperbola.
- **foc_radii** : Returns the focal radii of the hyperbola.
- **dir_circle** : Returns the equation of director circle of the hyperbola.
- **aux_circle** : Returns the equation of auxiliary circle of the hyperbola.
- **asymptotes** : Returns the equation of asymptotes of the hyperbola.

4. Example Codes

4.1 Example - 1

Find the area between auxiliary circle and director circle of the given ellipse $\frac{x^2}{25} + \frac{y^2}{16} = 1$

```
@copro.h

main -> int
'
    ellipse eqn:16, 0, 25, 0, 0, -400
    double aux_area = area(aux_circle((eqn)))
    double dir_area = area(aux_circle((eqn)))

    double ans = dir_area - aux_area
    output: "Area betwene director circle and director circle is", ans
    exit: 0
'
```

4.2 Example - 2

Find the intersection point of the tangent at (2,2) to the parabola $y^2 = 4x - 4$ and the line $x + y = 8$

```
@copro.h

intersection_point_fun:line l,line m -> point
'
    point temp
    if(l.slope()==m.slope())
    '
        output:"The Tangent at given point is parallel to given line"
    '
    else
    '
        temp:(f(l)*c(m) - f(m)*c(l))/(g(l)*f(m) - g(m)*f(l)),
             (g(m)*c(l) - g(l)*c(m))/(g(l)*f(m) - g(m)*f(l))
    '
    exit:temp
'

main -> int
'
    parabola par:0,1,0,-4,0,4
    point p:2,2
    line given_line:0,0,0,1,1,-8
    point i_point
    i_point = intersection_point_fun(tangent(par,p),given_line)
    output:"The intersection point is",i_point.x,i_point.y
    #The intersection point is 4, 4
    exit:0
'
```