COPRO

Final Report

CS3423 : Compilers - 2

Prof. Dr.Ramakrishna Upadrasta

Team

CS20BTECH11064	**Thota Rohan**	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

1.1 Motivation

CoPro is a domain-specific programming language built to help programmers program complex mathematical problems in the domain of the conic sections, by helping them out with functions related to that domain. A lot of essential ideas of CoPro have been inspired by C. CoPro provides a variety of data types, fundamental data types being void, int, double, string, and bool, and along with them, provides many non-trivial data types such as point, line, conic, line_pair, circle, parabola, ellipse, hyperbola. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Along with such data types and statements, CoPro provides control-flow constructions such as statement grouping, decision-making(if-else), looping with termination tests at the top(while), and loop control statements(break, continue). Functions that the user declares in the code may return any of the primitive data types. Local variables are typically "automatic", or created anew with each invocation. CoPro provides 'input' and 'output' functions that act as READ and WRITE statements; however, there are no built-in file access methods. It also provides multiple inbuilt functions depending on each data type; for eg., it provides inbuilt function 'slope' for data type 'line' and functions 'tangent', 'normal','directrix','eq_' etc. for data types such as 'parabola','ellipse','hyperbola' etc.
CoPro, like any other language, has its fair share of limitations. There are no arrays, structs, or unions which could be used, there are no 'for' loops, etc. Nonetheless, CoPro is very effective for the purpose it was written.

2. Language Tutorial

This document aims to give you an introduction to our language CoPro. This tutorial tries to give any newbie to CoPro, enough information to understand the language, and be able to write some code of his own.

2.1 Getting Started

Hello World program: Here we demonstrate how to write a simple hello world program in our language.

```
main -> int
<<
 output : "Hello World"
>>
```

Output: To run the file, you have to run the following commands in sequence, substituting filename below with your filename.

```
./util_files/project <"filename".cop> SymbolTables_ASTs/"op".txt 2>
    llvmOutputs/"op".ll
llc -o AssemblyFiles/2.s llvmOutputs/2.ll
llvm-as -o llvmObjects/2.bc llvmOutputs/2.ll
g++ -no-pie AssemblyFiles/2.s -o gccObjects/2.out
```

CoPro could primarily be said to consist of variables and functions. Variables are used to store values of operations computed in the code, and functions are a collection of statements that define the operations to be done. Out of all the functions one could declare and the ones already present, 'main' is unique. The execution of your code starts from 'main' function itself. This means that every function must have a 'main' function.
The 'main' function could also call inbuilt functions of our language CoPro. One could also pass arguments to a function following a colon ':' symbol in its call, however it is not necessary for a function to take arguments, 'main' doesn't take any. The statements inside a function are enclosed between '«' and '»'.
In the main function, other functions can be called with their names followed by a list of arguments in parenthesis.

2.1.1 Variables and Arithmetic Expressions

```
# print Fahrenheit-Celsius table
# for fahr = 0,20,.....,300

main -> int
<<
    int fahr, celsius
    int lower, upper, step

    lower = 0
    upper = 300
    step = 20

    fahr = lower
    loop (fahr <= upper)
    <<
        celsius = 5* (fahr-32) /9
        output : fahr
        output : "\n"
        output : celsius
        fahr = fahr + step
    >>
    exit : 0
>>
```

The below example program prints all celsius values for the fahrenheit values 1,20,40,...300. In the program, we see several new ideas such as comments, declarations, variables, arithmetic expressions, loops, and output.
The line 1 and 2 that start with # are recognized as comments and are ignored by the compiler. Comments are used inorder to make the code easily readable.
Every variable used in the program has to be declared before using. A declaration states the data type of variables. It consists of a data type and variable names of that data type.

```
int fahr, celsius
int lower, upper, step
```

The above two lines are variable declarations The int represents the data type of the variables fahr and celsius, lower, upper, step.
The type int means that the variables listed are integers. There are various other data types other than int. The dataype double represents integers with fractional parts.

2.2 The If statement

The if statement is used for executing specific lines of code based on a condition. If the condition in the parentheses is true then the body of the ‘if ‘ is executed. If there is an else block, and the condition is false, then the else block is executed.

Datatype	Description
integer	Integer type
long	Long Integer
double	double-precision floating point
bool	Stores True or False values
string	Collection of alphanumeric characters
point	Euclidian co-ordinate representation of position.
line	One dimensional figure with length but no width.
conic	A super class definition of all conic equations.
line_pair	Product of two real linear equations each representing a straight line.
circle	Locus of all points equiv-distant from a given point at a distance r.
parabola	A curve-distant from a fixed point and fixed line.
ellipse	A curve whose distance from two points adds up to a fixed constant.
hyperbola	A curve whose absolute difference of distances from two fixed points is always constant.

2.3 The loop statement

The loop statement is used for executing repeated operations. While the condition in the parentheses is true, the body of the loop (enclosed in curly bases) is executed. The loop ends when the condition of the loop becomes false. The body of the loop can be one or more statements enclosed in curly braces.The condition of the loop can be any expression.

2.4 Functions

A function provides a convenient way to encapsulate some set of computations, which can then be used without worrying about its implementation. This decreases code duplication.
For example, a simple function can be written for adding two integers. Everytime we want to add to integers, we can call the function and not worry about the implementation.
Since we do not have a function for powers, let us see how we can create one ourselves.

```
# Function to find m^n
power : int m, int n -> int
```

```
<<
    int i = 0
    int res = 1
    loop (i<n)
    <<
        res *= m
    >>

    exit : res
>>

main -> int
<<
    int x = 2
    int y = 2
    int z = 3

    # If we want to find y^z then we can call the function power
    int val = power(y, z)

    # Now if we want to find x^(y^z) then
    int res = power(x,val)

    output : res
    exit : 0
>>
```

Here the function 'power' can be called every time we need to compute an exponent. A function declaration has the form :
Function-name : parameter_list -> return-type « Statements »
Function calling has the form: function-name(parameter_list) The returned value from the function can be stored in a variable. A function need not have parameters. If a function has no parameters then, its definition is
Function-name -> return_type

2.5 Arguments - Call by Value

The parameters passed in CoPro are passed "by value". This means that all the values of the parameters passed to a function are stored in temporary variables and executed. The original parameter is not sent into the function. Therefore, if any parameter is changed in the function, the original variable is not changed.
For example in the above program, if the value of n or m is modified in the function, it has no effect on the values of m or n in the main function.
Call by value usually leads to more compact programs with fewer extraneous variables, because can be treated as conveniently initialized local variables in the called routine.

2.6 External Variables and Scope

The variables declared in main are private or local to main, they cannot be directly accessed from other functions.

3. Project Plan

This document elaborates on the project plan devised to complete the compiler for the domain-specific language - CoPro. The planning had begun when the project was announced (roughly post-Aug 15 2022), and after multiple changes, delays, cutbacks, and push-throughs had finally almost achieved what it set out to achieve. The plan was also divided into phases corresponding to deadlines for each phase.

3.1 Whitepaper - Submission deadline: 20 Aug 2022

The plan was to finalize the aim of our language and the roles of our team members by 17th Aug 2022, and then to iron out the features of our language, look at templates of language specification documents and then complete our language specification document by 20th Aug.

3.2 Lexical Analyzer - Submission deadline: 6th Sep 2022

For this phase, we first planned to look at reference material such as ANSI C's lexical analyzer and other examples of how to use flex. Then, using these references, we planned to complete our lexer using flex, and also created slides and videos explaining this phase.

3.3 Parser - Submission deadline: 16th Sep 2022

For this phase, we immediately had to link our lexer to the parser. Then every member had to look at references such as ANSI C's grammar etc. and later discuss together to decide how our grammar would go about. We then worked on our grammar, fixed conflicts and wrote code for a symbol table (no scoping), and built a parser tree(not complete). We also recorded videos and made slides and pushed all of them before the deadline.

3.4 Semantic Analysis - Submission deadline: 16th Oct 2022

We first discussed what we would implement in our semantic checks; once each member was assigned their roles, they had all their holidays to work on their assigned tasks. Once we returned from the semester break, we started integrating each other's work, updated our symbol table to include scoping, and finally completed the semantic analysis phase of our compiler along with slides and videos.

3.5 Code Generation - Submission deadline: 13th Nov 2022

We first had to learn about code generation, and IR's, and tried demo samples of how the IR would look for a particular code. We then had to modify our AST for code generation, and then write code for code generation. We also worked on our inbuilt functions during this time.

Finally, we had completed integrating our language and started writing reports and recording videos to be submitted by 22nd Nov 2022.

4. Language Evolution Report

Conic Problems (CoPro), as the name suggests, was started with the aim to build a compiler for a domain specific language that could help in solving tough mathematical problems in the area of conics. We first started with working on comments in our language

4.1 Commenting

We have used '#' to comment a line in the code. The whole line gets commented if it starts with '#' After which, we planned to work on declarations,

4.2 Declarations

The format in which we decided to declare is similar to C, "type-specifier identifier : initialization". Type-specifier can be specifically named as point/ line/ pairs of lines/ parabola/ ellipse/, or overall it can be named as a conic. Coming to the identifier we can use any combination of alphabets or "_" or even numbers, other than the main keywords. An identifier should also not start with a number. After the identifier is written we can assign the values using colon(":"). For initializations all the other variables are declared as in C-Language (int, double, string). We have no use of char or char* so we have removed that. Conics are initialized by their equation coefficients (6 values separated by commas) a, b, 2h, 2g, 2f, c from the general form of conic (

$$ax^2 + by^2 + 2hxy + 2gx + 2fy + c$$

). Now we started to work on function definitions.

4.3 Functions

The format of function definition is
Function_name: arguments -> return_type compound_statement Function_name can be anything as an identifier and followed by arguments separated by commas and the compound_statement. Initially we planned to begin and end the compound statement with single quotes(') and we have changed it to "«" and "»". Also many inbuilt functions are written to be easier for the user like finding the tangents, normals at a point on the curve, intersection of lines. These can be accessed directly without the usage of any headers. Now we went for printing and scanning statements.

4.4 OUT/IN Statements

Input can be taken by just writing “input : identifier”. To take multiple inputs we have to write this input followed by an identifier multiple times in the new line. Output is also similar, it too requires each item to be printed to be written on multiple lines. After this, we started doing conditional statements

4.5 Conditional Statements

while/for : We have used only one that is loop. The syntax is Loop (condition) : compound_statement if/elseif/else : Syntax is similar to loop If (condition): compound_statement elseif (condition): compound_statement else: compound_statement Next to this we decided to remove the end of line. At first we had many conflicts with writing grammar and then at last we worked and fixed it. This is the manner in which our language has evolved over time due to various reasons.

5. System Architecture

The components of compiler are:

5.1 Lexer

Lexer converts a sequence of characters into a sequence of tokens.errors like invalid characters,incomplete multi line comments are handled with white spaces getting ignored.

5.2 Parser

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

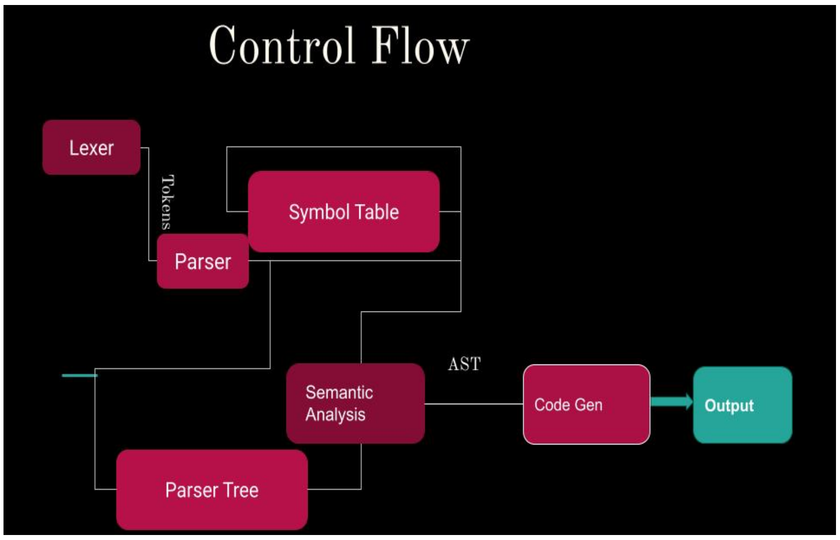
5.3 Semantic Analysis

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar.
Function of semantic analysis are:

1. Type checking : Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language.

2. Label checking : Ensures that data types are used in a way consistent with their definition.Every program must contain labels references.

3. Flow control check :
   Keeps a check that control structures are used in a proper manner.It occurs during compile time and run time.(example: no break statement outside a loop).



5.4 Intermediate Code Generation

Intermediate code can translate the source program into the machine program. Intermediate code is generated because the compiler can't generate machine code directly in one pass. Therefore, first, it converts the source program into intermediate code, which performs efficient generation of machine code further.

6. Development Environment

This document aims to give you an idea of our development environment, i.e how we used various tools for our project.

6.1 Ubuntu

We used Ubuntu 20.04 as our operating system because it comes pre-installed with all the tools required for software development and makes installing new tools easy.

6.2 VSCode

We used VSCode as our IDE which has various useful features for software development. It combines common developer tools into a single GUI. We used extensions for flex, bison which help in formatting the code properly. We also used the LiveShare feature in VSCode which allows all the team members to work on the same file at the same time and see the changes made by the other in real time.
We integrated git with VSCode. This allowed us to clone, push and pull from the repository from VSCode.

6.3 Git

We used git for source code management. It is a version control system which handles the project efficiently. It enables multiple developers to work together. It also tracks the changes made by each developer in the source code.

6.4 MAKE

We used MAKE to compile a program from source code. The make utility comes in handy as we need not run all the commands each time we make a change instead run make which takes care of all the commands for compiling. The makefile defines the set of tasks to be executed.

6.5 FLEX (Fast Lexical Analyzer Generator)

We used flex for creating the tokens from the source code. It reads an input stream specifying the lexical analyser and writes source code which implements the lexical analyzer in C.

6.6 Bison

We used Bison to generate our parser which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar in the .y file.

7. TestPlan

This document aims to give you an idea of our development environment, i.e how we used various tools for our project.

7.1 Ubuntu

We used Ubuntu 20.04 as our operating system because it comes pre-installed with all the tools required for software development and makes installing new tools easy.

7.2 Data Types

We have done the error testing by using mixed declarations and incorrect assigning If any error found, it points and specifies the type of error at that place.

7.3 Comments

We have done the error testing by commenting parts of the code and then checking for error.

7.4 Whitespace

We have done the error testing by varying spaces between the declarations.

7.5 Keywords

We have done the error testing by declaring some keywords as variables or functions.

7.6 Name

We have done the error testing by using keywords, digit, or any other symbol initialisation as the name.

7.7 Numbers

We have done the error testing by assigning mismatched types and numbers beyond bounds.

7.8 Literals

We have done the error testing of number literals at numbers and of string literals by removing some quotes or multi assigning them.

7.9 Variable Declaration

We have done the error testing by declaring them in nested loops and declaring same variables with different types.

7.10 Function Declaration

We have done the error testing by declaring same function names with different input and output types and doing recursion.

7.11 Initialization

We have done the error testing by non-default initialization of non-primitive data types and mutual assigning of both data types.

7.12 Disruptive Statements

We have done the error testing by exiting from void functions and not exiting from normal functions and multi break statements.

7.13 If Statements

We have done the error testing by checking nested ifs and different type checking.

7.14 Loop Statements

We have done the error testing by checking nested loops, iterations, break and continue statements.

7.15 Block Statements

We have done the error testing by using different blocks all at a time.

7.16 Expressions

We have done the error testing by assigning the wrong type of expressions and type mismatched expressions.

7.17 Prefix Operators

We have done the error testing by using different types of operators at same time.

7.18 Infix Operators

We have done the error testing by using different types of operators at same time.

8. Appendix - Reference Manual

8.1 Introdution

This Appendix A aims to give the overview of the whole language and how they work.

8.2 Tokens

There are six classes of tokens: identifiers, keywords, constants, operators, and other separators.
Blanks, horizontal and vertical tabs, newlines and comments as described below (collectively, ”white space”) are ignored except as they separate tokens.
Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

8.3 Comments

The character # indicates as a comment. Comments do not nest, and they do not occur within a string or character literals.

8.4 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter, the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length, and for internal identifiers, at least the first 31 characters are significant some implementations may take more characters significant.

8.5 Keyword

The following identifiers are reserved for the use as keywords, and may not be used otherwise

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

8.6 Numbers

8.6.1 Integer

An integer consisting of a sequence of digits. It is represented as **int**. Int is a 32-bit signed integer.

8.6.2 Double

A Double consists of an integer part, a decimal part. The integer and fraction parts both consist of a sequence of digits. It is represented as a **double**. Double is a 64-bit signed floating point value.

8.6.3 Long

An integer consisting of a sequence of digits. It is represented as **int**. Long is a 128-bit signed integer.

8.7 Literals

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

8.7.1 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 characters.

8.8 Operators

Operators are one of the features which has symbols that can be used to perform mathematical, relational, bitwise, conditional, or logical manipulations.
The multiplicative operators *, /, and % group left-to-right. The binary * operator denotes multiplication. The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second.

```
multiplicative_expression:
    multiplicative_expression '*' cast_expression
    multiplicative_expression '/' cast_expression
    multiplicative_expression '%' cast_expression
```

The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands.

```
multiplicative_expression:
    additive_expression '+' multiplicative_expression
    additive_expression '-' multiplicative_expression
```

The operators < (less), > (greater), <= (less or equal) and >= (greater or equal) all yield 0 if the specified relation is false and 1 if it is true.

```
relational_expression:
    shift_expression
    relational_expression '<' shift_expression
    relational_expression '>' shift_expression
    relational_expression '<=' shift_expression
    relational_expression '>=' shift_expression
```

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.

```
equality_expression:
    relational-expression
    equality_expression '==' relational_expression
    equality_expression '!=' relational_expression
```

The && operator returns 1 if both its operands compare unequal to zero, 0 otherwise.

```
logical_and_expression:
    inclusive_or_expression
    logical_and_expression '&&' inclusive_or_expression
```

The || operator returns 1 if either of its operands compare unequal to zero, and 0 otherwise.

```
logical_or_expression:
    logical_and_expression
    logical_or_expression || logical_and_expression
```

8.9 Declarations

We have different types of declarations one for the normal C type of declaration which are

```
declaration:
    type_specifier init_declarator_list EOL
    type_specifier mulendoflines init_declarator_list EOL
```

8.9.1 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.
Here we can declare variables just like "int a = 10, n = 15, c = 13" or "circle c1 : 1, 0, 1, 0, 0, 4"

8.10 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 true; otherwise, the optional else branch is taken.

8.10.1 Disruptive Statements

```
jump_statement :
    CONTINUE EOL
    BREAK EOL
    EXIT exit
```

Break statements is used when the programmer wants to abruptly exit out of loop.
Exit statements 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.

8.10.2 If Else 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.

```
selection_statement:
    IF '(' expression ')' EOL compound_statement EOL ELSE EOL
   compound_statement
    IF '(' expression ')' EOL compound_statement EOL
```

8.10.3 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.

```
iteration_statement:
    LOOP '(' expression ')' EOL compound_statement
```

8.10.4 Block Statements

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

```
compound_statement:
    FUN_ST FUN_EN
    FUN_ST temp_fun statement_list FUN_EN
```

8.11 Expressions

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.

```
expression:
    assignment_expression
    expression ',' assignment_expression
```

8.12 Variables and Declaration

We can have any name of inbuilt variables except that we shouldn't have to use the keywords We have to type of inbuilt variables

```
POINT IDENTIFIER ':' initializer ',' initializer EOL
conic_specifier IDENTIFIER ':' initializer ',' initializer ',' initializer
    ',' initializer ',' initializer ',' initializer EOL
conic_specifier IDENTIFIER '=' IDENTIFIER '.' IDENTIFIER '(' ')'
conic_specifier IDENTIFIER '=' IDENTIFIER '.' IDENTIFIER '(' IDENTIFIER ')'
```

8.13 InBuilt Functions

Function	Return Type	Arguments	Description
conic_name	string		returns name of the conic
eq_	string		returns the 6 tuple equation
slope	double		returns the slope of the line
delta	double		returns the delta value
eccen	double		returns the eccentricity of conic
area	double		returns the area of the circle or ellipse
latus_rectum	double		return the latus rectum for conic
tangent	line	point	return the tangent at particular point
normal	line	point	return the normal at particular point
p_axis	line		return the principal axis of conic
directrix	line		return the directrix of conics
vertex	point		returns the vertex of parabola, circle, hyperbola
focii	point		returns the focii of the parabola, ellipse
on_curve	point	bool	returns whether point is in curve or not
aux_circle	circle		returns aux_circle of the ellipse
dir_circle	circle		returns dir_circle of the hyperbola

Table 8.1: Inbuilt Functions

8.14 Calling Inbuilt Functions

We have particular type of syntax of calling inbuilt functions for functions **delta, slope, eccen, latus_rectum, area** the function calling should be like

```
function_call:
    IDENTIFIER '.' IDENTIFIER '(' ')'
```

for the function tangent, normal, p_axis, directrix, vertex, focii the function calling should have to be like

```
function_call:
    conic_specifier IDENTIFIER '=' IDENTIFIER '.' IDENTIFIER '(' ')'
    conic_specifier IDENTIFIER '=' IDENTIFIER '.' IDENTIFIER '(' IDENTIFIER
     ')'
```