Final Project: Compilers - II

Prof. Ramakrishna Upadrasta
Department of Computer Science
Indian Institute of Technology, Hyderabad

Tangent

Final Report - Team 4

Team Members:

AMAN PANWAR - CS20BTECH11004
PRANAV K NAYAK - ES20BTECH11035
SHAMBHU KAVIR - CS20BTECH11045
SHREYA KUMAR - ES20BTECH11026
TAHA ADEEL MOHAMMED - CS20BTECH11052
VIKHYATH - CS20BTECH11056

Contents

INTRODUCTION TO TANGENT

1.1 Description

Tangent is a statically typed, object-oriented programming language intended for creating vector graphics. It takes design cues from Python's Matplotlib library and C++'s object-oriented functionality. Using built-in classes on an SVG, Tangent can be used to plot a variety of complicated figures, such as Bézier curves and polygons. Tangent also aims to offer users standard general-purpose programming language functionality such as looping and control flow. Tangent, while being object-oriented, aims to be simple, readable, and portable.

1.2 Motivation behind Tangent

Existing libraries that provide graph-plotting capabilities are hampered by the format that they encode images in, namely, raster graphics. For example, matplotlib produces its images as jpeg files, which, while having many benefits, cannot be relied upon for visual clarity. The very nature of the encoding in the form of bitmap files means the image is static, and zooming in will only expose the pixels whose states were set when the image was first generated.

SVGs take advantage of the nature of vector graphics by placing shapes in a 2D environment by way of a program consisting of a sequence of definitions for shapes and curves. This leads to a smaller file size while still retaining quality. Since the images are the result of a sequence of instructions, they can be re-rendered for every resolution, meaning there is no loss of quality when zooming in. This is especially useful when precision is required in images, like in the case of printing.

1.3 Design Goals

We aim for Tangent to be a user-friendly and particularly, a beginner-friendly language. Users should be able to create vector images with this language in a simple, elegant way. Beginners should be able to rapidly integrate this language into their projects because of the simplicity of the language itself, being very similar in structure to other C-style general-purpose languages.

Tangent's object-oriented facets include inheritance and polymorphism. Programming in Tangent is meant to be intuitive and convenient, allowing users to model real-world graphs and images with a level of control that normal bitmaps simply cannot allow, by harnessing their experience programming in other object-oriented languages towards creating images by making use of the built-in classes (e.g. line, curve, polygon, bezier, etc.) to build their own user-defined classes as per their needs or those of their clients.

LANGUAGE TUTORIAL

1.1 Getting Started

The first step while learning a new programming language is learning how to print “Hello world!”. So let’s learn how to write a “Hello World!” program in Tangent:

```
# A basic “Hello world!” program
driver <: :>
(:
      # Printing statement
      print(“Hello world!”);
:)
```

Now to run this program,

- Save it as file with the extension ‘.tngt’
- Run the ‘make’ command to execute the Makefile and generate the Tangent compiler. Now, compile the ‘.tngt’ source code as follows:

```
./tangent filename.tngt
```

- If your code has no errors, then the tangent compiler will generate an executable which can be run using the command:

```
./filename.out
```

Note: The executable generated will have the same filename as the source code (i.e .tngt file) with ‘.out’ extension

If all goes well, this is what your output should look like:

```
Hello world!
```

Coming to the explanation of this code,
Tangent uses a **driver** as an equivalent to main in any other C-based language (like C, C++, Java, etc). All function execution begins from the driver function. It also uses parentheses with the digraphs (: and :) and curly braces with <: and :>. However they can be used interchangeably.
Here, *print* is an inbuilt function that prints “Hello world!” to the standard output.
By default, the *print* function prints a new line. So if we change our code to look like this:

```
driver <: :>
(:
      print(“Hello”);
      print(“world!”);
:)
```

The output would look like:

```
Hello
world!
```

Notice that “Hello” and “world!” are on different lines, even though we haven’t explicitly used a newline character (“\n”).

In the “Hello world” program you may have noticed some text written after “#” does not look like valid code, this is because these are *comments*. Anything written after a ‘#’ is ignored by the compiler. Tangent supports only single-lined comments. To write multi-line comments, you just need to begin every line with ‘#’. Comments can also be written at the end of a valid Tangent statement. They are usually used to make the program easier to understand.

1.2 Variables and Arithmetic Operations

To learn how to work with variables and arithmetic operations, here is a sample program demonstrating how to calculate the area of a rectangle and a circle.

```
driver <: :>
(:
      var int length := 10, breadth := 20;
      var int radius := 5;
      const float pi := 3.14;

      # Calculating the area of rectangle
      var int area1 := length*breadth;

      # Calculating the area of circle
      var float area2 := pi*radius*radius;

      # Printing the areas
      print("Area of rectangle = " + to_string(area1));
      print("Area of circle = " + to_string(area2));
:)
```

Here, the variables *length, breadth and radius* are initialized when they are created. But we can also initialize them later by replacing those lines of code with:

```
var int length, breadth, radius;
length := 10;
breadth := 20;
radius := 5;
```

pi here is a constant variable. All constant variables must be initialized during declaration and cannot have their values modified throughout the program, otherwise the compiler will throw an error.

Tangent also has an **inbuilt constant PI** which can be used here instead as

```
var float area2 := PI*radius*radius;
```

When we print the final output, we must convert the final result to a string and concatenate it with the rest of the output before we pass this as an argument to the inbuilt print function.
+ operator is overloaded to perform concatenation on strings and perform addition on all numerical data types.

For eg:

print(2 + 3); # output : 5
print(“2” + “3”); # output : 23

Tangent also has a set of inbuilt functions to convert one datatype to another,

- *to_int()*
- *to_float()*
- *to_string()*

1.3 Arrays, Loops and Functions

Now we will write a program to calculate the sum of elements in an array. In this section we will learn how to work with arrays, the different looping constructs and how to work with functions.

```
# Program to calculate the sum of elements in an array
int sum <:int a[], int n:>
(:
      var int sum := 0;
      for(int i := 0; i < n; i++)
      (:
            sum += a[i];
      :)
      send sum;
:)

driver <: :>
(:
      var int a[5];
      a[0] := 1; a[1] := 2; a[2] := 3;
      a[3] := 4; a[4] := 5;

      var int sum := sum(a, 5);
      print("Array sum = " + to_string(sum));
:)
```

var int a[5]; is an array declaration statement.
In tangent, during array declaration, we must specify the number of elements in the array. To initialize the array, we must assign the value of every element separately. Note: Arrays are 0-indexed, so a[0] is the first element of this array and a[4] is the 5th and last element of this array. If we try to access a[5], the compiler will throw an error.

In the declaration of variable *ans*, we have equated it to *sum*(a,5). Here, we have made a call to a function *sum*, which we have defined above driver. It looks very similar to the driver function, but has some minor changes:

(i) The function header looks like *int sum <: int a[ ], int n :>*. 'int' is the return type of the function, the driver function has no return type. *sum* function also has a few arguments passed to it, unlike the *driver* function.

(ii) the last line of the function is the statement ***send sum;***, which is basically a return statement that returns a value back to the caller function. The value returned by this function call is assigned to the variable ans.

The return type of a function can be any valid Tangent data type and there MUST be a send statement that returns a value of that type.
If a function has a void return type, then it need not have a send statement.

We have also used a for loop in the sum function. Every for loop has an equivalent while loop. The for loop in this program can be written in while loop form as:

```
int i := 0;
while(i != n)
(:
        sum += a[i];
        i++;
:)
```

Notice the statement

sum += a[i]

This is equivalent to sum = sum + a[i]

+= is a shorthand for addition assignment operation. Similarly we can also use -=, *=, %= and /=

1.4 Selection Statements

Given below is a program to print the text form of a number from 1 to 5.

```
# Number to Text conversion
driver <: :>
(:
      var int n := 3;
      switch(n)
      (:
            case 1:
                  print(“One”);
                  break;
            case 2:
                  print(“Two”);
                  Break;
            case 3:
                  print(“Three”);
                  Break;
            case 4:
                  print(“Four”);
                  Break;
            case 5:
                  print(“Five”);
                  break;
            default:
                  print(“--”);
      :)
:)
```

Switch statement takes an expression and has cases with constant case labels, i.e a case label cannot be a variable, it must only be a constant. If the expression evaluates to any of the case labels, those specific statements are executed. The statements under the default label are executed if none of the case labels are matched.

This code can be equivalently written using If-else statements,

```
# Number to Text conversion - IfElse version
driver <: :>
(:
      var int n := 3;
      if (n = 1)
      (:
            print(“One”);
      :)
      else if (n = 2)
      (:
            print(“Two”);
      :)
      else if (n = 3)
      (:
            print(“Three”);
      :)
      else if (n = 4)
      (:
            print(“Four”);
      :)
      else if (n = 5)
      (:
            print(“Five”);
      :)
      else
      (:
            print(“---”);
      :)
:)
```

In this version of the program, we use if statements and comparison operators. In tangent, we test equality using the = operator (not ==). Other comparison operators include !=, >=, <=, >, <. There are also logical operators like & (AND) , | (OR) and ! (NOT).
For example,
(i) if you want to print n, only if it a multiple of 3 and 5

```
if(n % 3 = 0 & n % 5 = 0)
(:
      print(n);
:)
```

(ii) if you want to print n only is it divisible by 2 or 5

```
if(n % 2 = 0 | n % 5 = 0)
(:
      print(n);
:)
```

1.5 Object Oriented Programming

One of the key features of Tangent is the easy usage of OOP principles. Now, we will demonstrate how to create classes, inherit from other classes and work with objects of these classes.

```
family Car
(:
	public var string colour;
	private var int seat_count;
	Car<::>
	(:
		colour := “Blue”;
		seat_count := 4;
	:)
	void printCar<::>
	(:
		print(“Colour = “ + colour);
		print(“Seat count = “ + to_string(seat_count));
	:)
:)

driver <: :>
(:
	Car c;
	c>>colour = “Red”; # Changing the colour of the car to red
	c>>printCar();
:)
```

In tangent, classes are called families and are declared using the keyword family. Here, we have created a class called Car which has two member variables - *colour and seat_count* and one member function *void PrintCar()*.

public and **private** are access specifiers which define how these members can be accessed. If there is no access specifier mentioned, then the member is assumed to be public by default.

We have also defined a constructor for the family i,e *Car<::>*. Whenever an object of this family is created, then the constructor is called and initializes all the member variables.

c is an object of family type *Car*. Objects of classes are created just like variables are created for primitive datatypes.
To access the members of the object, we use the operator “>>”.

Tangent also allows one class to inherit from another. A parent class can have multiple derived classes, but a child class can be derived only from a single parent class. The code given below demonstrates how to implement inheritance in Tangent:

```
family BaseClass
(:
	private var int a;
	public var int b, c;
:)

family ChildClass :: public BaseClass
(:
	public var int d;
	ChildClass<: :>
	(:
		d := 1;
		b := 2, c := 3;
	:)
:)

driver <: :>
(:
	ChildClass obj;
	# obj>>a := 3; is an invalid statement
	print(obj>>b);
:)
```

ChildClass publicly inherits BaseClass using “::” operator. Only the public variables of BaseClass are inherited by ChildClass. Therefore, an object of ChildClass cannot access variable a.

1.6 Creating an SVG file

Coming to the main application of our language, writing code to generate SVG files. Programs written in Tangent to serve this purpose are very intuitive and self-explanatory. Tangent has many inbuilt functions that help to write such programs with ease.

Here is a program that generates an SVG image with different points and lines.

```
driver <: :>
(:
      var Image I;
      I.set_dimension(100, 100);
      var Rectangle r;

      # Drawing one rectangle
      r>>set_top_left(20, 20);
      r>>set_dimensions(10, 30);
      I.draw(r);

      # Drawing another rectangle
      r>>set_top_left(60, 60);
      r>>set_dimensions(10, 30);
      I.draw(r);

      # Outputting the final image to an SVG file
      I.output(“image.svg”);
:)
```

Here we create a variable of type Image and set its dimensions to be the dimensions that we want for the final SVG file.

We have created a variable of type Rectangle. Note that both Rectangle and Image are inbuilt data types. Then we have set the dimensions of the rectangle, the location of its top left corner in the final image and added it to the Image using the draw() function.

Finally, we output the image into an SVG file using the output() function.

TANGENT REFERENCE MANUAL

1. **Lexical Convention**
    a. **Comments**
    Comments in the language start with a '#' symbol. Similar to python language the comments are only single-line comments. To make multi-line comments multiple single line comments can be used one after the other.
    b. **Identifiers**
    All identifiers are one or more characters strictly starting from an English character followed by numbers, underscore, or letters.
    Eg: Aman, Pranav_5, etc are valid identifiers
    c. **Keywords**
    The keywords defined in the language are as follows

i.	int	xviii.	send
ii.	float	xix.	const
iii.	string	xx.	me
iv.	bool	xxi.	public
v.	void	xxii.	private
vi.	var	xxiii.	Point
vii.	family	xxiv.	Path
viii.	if	xxv.	Image
ix.	else	xxvi.	Rectangle
x.	for	xxvii.	Circle
xi.	while	xxviii.	Ellipse
xii.	switch	xxix.	Polygon
xiii.	case	xxx.	Curve
xiv.	break	xxxi.	func
xv.	continue	xxxii.	Pi
xvi.	true	xxxiii.	Colour
xvii.	false		

    d. **Punctuations**
        i. Commas (' , ') are used to separate identifiers and constants in sequences during the variable assignment, declaration, and function argument list.
        ii. All statements are terminated by semicolons (' ; ').
        iii. Argument lists are enclosed by the following digraph: '<: :>'.
        iv. The scope is defined by enclosing blocks in the following digraph: '(: :)'.
        v. Inheritance of family is mentioned by the following '::'.
    e. **Constants**

Constants in the language are defined using the ‘const’ keyword. The general way to declare a constant is ‘const <type> <identifier> = <value>;’

i. Int Literals
‘const int num = 3;’
ii. Float Literals
‘const float fnum=8.9;‘
iii. String Literals
iv.
‘const string str = “tangent”;’
v. Bool Literals
‘const bool b = false’

f. Operators

i. Types

1. Arithmetic Operators

The following are the arithmetic operators. They are defined only for float and int types.

a. Addition Operator +
b. Subtraction Operator -
c. Multiplication Operator *
d. Division Operator /
e. Modulus Operator %

2. Relational Operators

a. Equal To =
b. Greater Than >
c. Less Than <
d. Greater Than or Equal To >=
e. Less Than or Equal To <=
f. Not Equal To !=

3. Logical Operators

a. Logical And &
b. Logical Or |
c. Logical Not !

4. Assignment Operator

The assignment operator is defined by the digram ‘:=’
Use: ‘<Identifier> := <value>’

ii. Precedence and Associativity

Precedence follows the template set by languages like C and Java, with the order being as follows with decreasing precedence:

1. Unary Operators
2. Functions
3. Parentheses
4. Multiplication, Division, and Modulo
5. Addition and Subtraction
6. Relational Operators

7. Logical Operators

2. **Types**
   a. **Declaration**

   The syntax for declaration is as follows:

   var <type> <identifier> [:= <value>]$_{opt}$;

   Example:

   var float my_num;
   var int your_num := -12;

   b. **Primitive Types**

   Tangent takes inspiration for its primitive data types from other general purpose programming languages, like C++ and Java. Its primitive data types can handle integers, floating point numbers, boolean values, and strings. Characters in Tangent are treated as strings of unit length.

   The types are:

   i. Integer
   ii. Floating point numbers
   iii. Boolean
   iv. string

   c. **Derived Types**

   i. **Arrays**

   Just like any ordinary language, arrays in Tangent can be used to store multiple data of the same type together. Multi-dimensional arrays can also be created which shall be implemented using a single array of larger size in the background.

   For example:

   'var int[5] arr;' will declare an array of integers of size 5.

   ii. **Families**

   Syntax to define families is:

   Family <family name> :: <access specifiers> <base class>
   (:
   Var <type> <identifier>;
   func <return type> <function_name><: <arg_type> <identifier> :>
   (:
   <body>
   :)
   :)

   iii. **Pre-defined Classes**

   The language defines pre-defined data types which represent various graphical concepts/structures. The types are

   1. A 'Point' object defines a mathematical point on a 2D plane.
   2. A 'Curve' object defines a mathematical 2D curve. 'Rectangle' represents a collection of 4 points that form a rectangle. It is

defined using the attributes length, breadth and centre of the rectangle.

3. ‘Ellipse’ represents a mathematical ellipse. Its attributes are length of major axis, length of minor axis and centre of the ellipse.
4. ‘Circle’ similar to Ellipse, represents a mathematical circle. Its attributes are centre point and radius.
5. ‘Polygon’ is a mathematical regular polygon defined by the number of sides, the size of each size and the centre of the polygon. We can use this to make equilateral triangles, squares and so on.
6. ‘Colour’ is a pre-defined family which lets us define the rgb values for a colour. It helps us set the colour of an image
7. ‘Image’ is the key type to making the .svg images. It acts like a canvas for all the above mentioned types.

3. Syntax Notation

a. Program Structure

The program first defines and functions and classes. The driver function is always defined in the end. This is necessary as the language does not support function primitive and definition being at different places in the program.

b. User-defined Functions

User defined functions are defined as

func <return type> <function_name><: <arg_type> <identifier> :>
(:
    <body>
:)

c. Function Calls

Function calls can be evaluated as the return type of the function.
Ex:

```
A = func(33);
```

d. Driver Function

The driver function the equivalent of the main function in C/C++
Syntax: driver<: :>(: body :)

e. Statements

All statements are terminated by semicolons (‘ ; ‘).

i. Assignment Statements

Assignment is done using the assignment operator (‘:=’). Ex:

A := 56;

ii. Branch Statements

1. If-else construct

The syntax for if-else constructs is as follows

if(<condition expression>)
(:
    <if body>
:)

```
else
(:
        <else body>
:)
```

Else if construct can by made using another if statement inside teh body of if or else clause.

Example:

```
if(a1%2 = 0)
(:
        ans = 1;
:)
else
(:
        ans = 0;
:)
```

2. **Switch Construct**

   The syntax for switch constructs is as follows

```
switch(<expression/varibale>)
(:
        case <expression> :
                <body>
        case <expression> :
                <body>
        default:
        <body>
:)
```

   Example:

```
switch(a)
(:
        case 23:
                X = 3;
                break;
        default:
                T = 4;
:)
```

3. **Break**

   The break construct jumps the execution of the program to the end of if-else or switch or loop constructs.

4. **continue**

   This statement skip to the next interation of the loop

iii. **Loops**

1. **While Loop**

   Syntax:

```
while(<condition>)
```

(:
<body>
:)

Example:

while(i>0)
(:
T = T +i;
i -=1;
:)

2. **For loop**

Syntax:

for(<initiation>; <condition>; <iteration>)
(:
<body>
:)

Example:

for(var int i := 1; i <= 10; i := i + 1)
(:
print(i);
:)

f. **Scope**

Scope in tangent are defined using brackets. Opening bracket is represented by digram '(:' and the closing bracket is represented by ':)'

4. **Error Handling**

By creating strong token definitions, we have attempted to maximise error handling at the lexical stage. Once the compiler encounters an error, the line number and associated error message should be displayed on the command line. The following are examples of compilation errors

1. when a rule evaluates to a false predicate.
2. when some input does not match any expression.
3. when none of the grammar rules provides a specific path to follow.

5. **Object Oriented Programming with Tangent**

We knew from the beginning that the design of our language would be object-oriented, allowing us to provide the user all the resources they need to expand on already-existing data types and create complicated designs that could be implemented using bezier curves to get the desired results. Some of the most fundamental and practical object-oriented programming ideas are listed below, along with how our language handles them:

1. **Abstraction**

An OOP language's objects offer an abstraction that conceals the internal implementation specifics. Similar to the coffee maker in your kitchen, all you need to know to do a certain action is which methods of the object are accessible to call and which input parameters are required while the actual implementation of something might not be necessary to know. So, when it comes to abstraction in

Tangent, there are some functions that can help the user with some functions that might be complex otherwise. For example, if a user would like to rotate the rectangle by a particular angle or say if a user would like to change the centreThe goal of the programming language Tangent is to be user-friendly for beginners. Users may create vector pictures with this programme in a simple, stylish way. Beginners may rapidly come up to speed with the language because of how it is designed.The goal of the programming language Tangent is to be user-friendly for beginners. Users may create vector pictures with this programme in a simple, stylish way. Beginners may rapidly come up to speed with the language because of how it is designed.The goal of the programming language Tangent is to be user-friendly for beginners. Users may create vector pictures with this programme in a simple, stylish way. Beginners may rapidly come up to speed with the language because of how it is designed.The goal of the programming language Tangent is to be user-friendly for beginners. Users may create vector pictures with this programme in a simple, stylish way. Beginners may rapidly come up to speed with the language because of how it is designed. point of a circle around the image, the user could call an existing method to do the same and not worry about how that particular method gets the work done. However, the user may utilize the needed functionality by following the general blueprint that the class contains.

2. **Encapsulation**
   Encapsulation is the term used to describe the packaging of data and information into a single unit. Encapsulation is the binding together of the data and the functions that manipulate them in object-oriented programming. Our language requires this feature to make sure that users don't attempt to modify particular characteristics defined in classes. No function from the class may be accessed directly. To access the method that uses the class member variable, we require an object. Encapsulation refers to the usage of all member variables by the function that we create inside the class.
3. **Inheritance**
   Inheritance allows us to derive from other classes, ‘inheriting’ their member variables and functions. Hence we can reuse code and better represent the relationships between classes. Inheritance is crucial for ‘Tangent’, as the different classes representing graphical objects have many “is-a” relationships, such as squares & rectangles, ellipses & circles, or user defined classes. The syntax for inheritance is similar to C++.
4. **Polymorphism**
   Polymorphism is a core concept in OOPS that allows us to describe situations in which something occurs in several different forms. Mainly, we support function and operator overloading, allowing us to have several functions of the same name but with different parameters. A basic example of this in our language is overloading of Image.draw(), which can take various different types as a parameter, such as points, rectangles, curves, polygon, etc;

TANGENT PROJECT PLAN

The deliverables of each part of the compilers were tied to their submission deadlines.

- **Week 0**
    - We finalized the idea for the language
    - Flex, bison, and llvm were chosen as the tools we would use to create the compiler
- **Week 1**
    - We discussed the finer details of the language implementation and the feature it will support.
    - We created the syntax for the language and wrote sample programs for the same.
    - We wrote the white pages for the language.
- **Week 2**
    - We completed the initial setup of the repository
    - We slightly modified the language's grammar to remove ambiguities
- **Week 3**
    - We reevaluated the tools we would use to make the project. We considered using ANTLR and CMake but choose to stick with the previously decided tools.
    - We explored how we would implement the inbuilt data structure of our language
- **Week 4**
    - implemented lexer
    - implemented build automation
    - started working on automated testing using github actions
    - started implementing parser
    - We read further on lex/bison from here
- **Week 5**

    - We implemented the parser with bison
    - We updated lexer to work with bison-generated tokens
    - We found a way to automate testing for parser
- **Week 6**
    - We improved the grammar to reduce the sr conflicts and implemented the said changes in the parser
    - We wrote test cases to test the parser. We improved our testing automation and used it for all further testing
    - We started reading up on the next phase which is semantic analysis
- **Week 7**
    - We continued to read up on semantic analysis after which we started to formulate a plan to implement semantic analysis
    - We referred to Kaleidoscope Manual for semantic analysis
- **Week 8**
    - We focused on other academic work and thus did not make any notable progress on the project
- **Week 9**
    - We wrote the classes to be used in the AST
- **Week 10**
    - We made progress on the AST classes and their member functions/variables.
    - We wrote the print and evaluate functions for each class. This is a significant amount of work as the implementation of each function is not repetitive or trivial.
- **Week 11**
    - We implemented the initial AST classes
    - We tied our automated testing with GitHub actions. Now GitHub automatically tests every push we make to it.
    - We updated the lexer to include the symbol table. We also added row and column info to tokens passed from the lexer to the parser
- **Week 12**
    - We started writing actions in bison to generate AST Nodes

    - We updated the bison we use in the project to bison 3.8.3 Using the features of the newer bison version the parser now creates a HTML doc with hyperlinks to allow easy debugging
    - We implemented the functions that are being used in the lexer.l
    - We rewrote parts of the parser to use precedence rules
- **Week 13**
    - We worked on integrating the symbol table with the AST
- **Week 14**
    - We worked on adding the concept of family(the equivalent to classes in C) which was not represented earlier in the AST Classes earlier
- **Week 15**
    - We focused on other academic work and thus did not make any notable progress on the project
- **Week 16**
    - We worked on how to traverse the AST generated by the parser and populate the symbol table
- **Week 17**
    - We progressed on traversing the AST and populating the symbol table
    - We studied how to do the testing for ast, semantic analysis, and code gen
- **Week 18**
    - The parser now created the AST.
    - We implemented a function to traverse ast
    - We rewrote AST nodes functions rewritten to work with some drastic new design decisions
    - We started with code gen. For each class, we wrote functions to do code gen with llvm. This is based on the Kaleidoscope Reference Ch3

Language Evolution:

The start of the project involved a lot of decisions. Questions broadly fell into the arenas of

1. Our Type System
2. Our Syntax
3. Our Architecture
4. What would make the language stand out
5. The Language's Name

Most questions asked were not immediately answerable, since none of us knew what went into the design of a language, nor what the correct questions to ask were. No one knew whether we were going to be able to implement anything resembling object-orientedness, or if we could make our programming paradigm one of functional programming. As such, most decisions were made over the course of the complete project.
The type system was one of the first areas we ended up hashing ideas out in, since everyone understood what the terms we were throwing around meant:

1. **Would our language be statically typed or dynamically typed?**
   We agreed that the language should be statically typed, since we believed that storing type knowledge could be implemented in the front-end, our language being a compiled one and not an interpreted one. At that point, we didn't believe we had the technical knowledge to implement a dynamic type-checker that checks at runtime. This was also one of the reasons we went with a compiled language in the first place.
2. **Would our language be type-safe?**
   To make use of the statically-typed nature of the language, we decided to implement type safety as well. There is no implicit type coercion, and any assignments between types require an explicit cast. This would end up playing an important role later when we decided on what features would make the language stand out.
3. **Which of the primitive types would we let our language use?**

We chose to follow the example of C and implement every basic data type that exists in it, since it laid the template for nearly every general-purpose language to follow it, and almost all of them followed its type system for the primitive types.

Questions revolving around syntax would be the next ones we tackle, since those would primarily have to be implemented during the scanning phase, whose deadline was fast approaching:

1. **Which existing language would we use as a syntax reference?**
   C++ was the language all of us had the most familiarity with, and so we ended up following C++'s template for syntax, with some flavor changes to make the language our own. We borrowed its format for scoping, its concept of there being a single driver function, and its format for function definition and calling.
2. **How would we modify C++'s syntax to give it our own flair?**
   We changed most of the default C++ symbols for scope delimiting, as well as the symbols for surrounding function parameters and those for denoting scope-access.
   We also made our own keywords for our driver function and return statements. In order to have some diversity in our inspirations, the language only has single-line comments, borrowing the idea from Python.

Following these preliminary questions, we believed that we had taken enough decisions in the planning phase to actually begin implementation. Next were questions about what our technology stack would be.

1. **What technology stack would we use to implement our scanner and parser?**
   We had a number of choices. Would we choose ANTLR, in which we could implement both our scanner and parser using the vast reference of ANTLR grammars that exist? Or would we use the default choices of Flex and Bison? We could go with a handwritten lexer and parser, instead of using a generator. We could also use something completely different, like a text-analysis library in OCaml or Haskell.

In the end, our familiarity with C++ and the extensive nature of GNU's documentation was what led to us deciding to use Flex and Bison. We were also told that integrating the various parts together.

2. **How would we distribute the analysis between the scanner and the parser?**
   We decided to follow the standard distribution of letting our scanner detect invalid symbols, and leave the job of handling syntax errors in lexically valid statements to the parser.
3. **What would we use for our backend?**
   LLVM, the library used by C++, C, Julia, and Haskell, among other, very popular languages, seemed the ideal candidate to handle our backend. We believed its bitcode IR would be very straightforward to interface with, and the thought of being able to study such an advanced piece of software engineering could only improve our understanding of the process of compiler construction.

Then came design decisions regarding how we would make our language stand out. What flair could we add to make it different from just another C clone? We arrived at the following features:

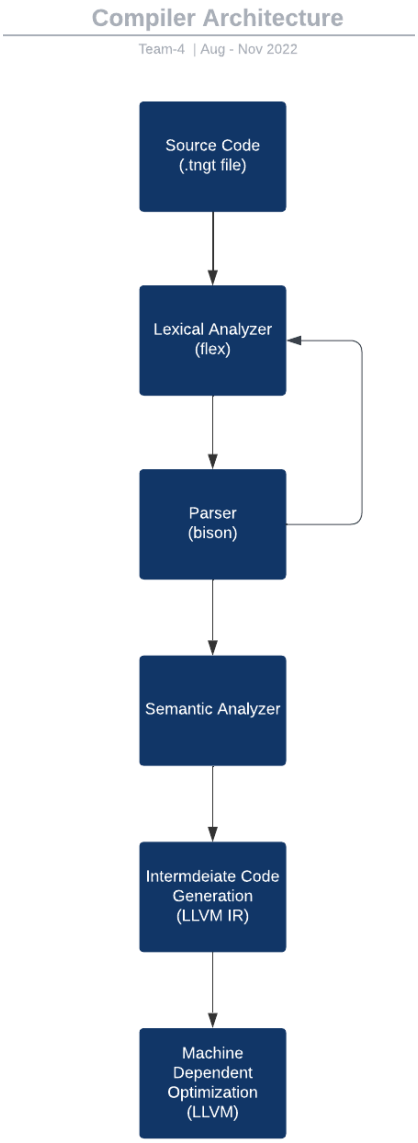
1. **Object Orientedness:** Object oriented programming is the dominant paradigm today. Nearly all production-level software is written in an object oriented language and makes extensive use of object oriented features. If we wanted our language to be usable, we would need it to be able to create and manipulate objects.
2. **Use Case:** This is where we arrived at a use case for the language, the crux of our project. We decided to implement a graphing language, one that can generate SVGs by creating an interface that outputs SVG images from an input specification that would be much simpler to write and edit than the default SVG markup.
3. **Built-in Features:** We decided that we'd rely most heavily on Bezier Curves to create the vector images, since those could be used to simulate any curve that could be parameterized. As such, the language would need a built-in class for said structure.

The language contains not only a class for Bezier Curves, but also ones for simpler structures, like lines and circles. All the classes would come with functions for specifying and modifying their attributes, along with methods to write them to the resultant file.

Finally, we were at the stage where we could decide the name of the language. After much discussion and debate, “Tangent” was born.

As the project progressed, we came to realise that there was a decent chance we had bitten off more than we could chew. We were ambitious, but our ambition resulted in us not having enough code to fully implement our planned test suite on.

Compiler Architecture



Lexer and Parser

In our project, the lexer and parser work hand in hand on our input *.tngt code to analyze it. Firstly, we have our language's keywords and patterns in our lex file, which scans the input code on the parser's request, until it matches a token it can return. Then the parser takes the returned token and tries to either shift or reduce it in its internal stack, to match our language's grammar rules, which are defined in our parser.y file. Our parser is an

Semantic Analysis

While the parser matches the grammar rules using the tokens, we use bison's semantic actions feature to construct the AST and symbol Table for the code.

Our symbol table is implemented as a map from the identifier names, to the symbols. Also, each symbol table can have nested children symbol tables to represent scoping. For example, the global symbol table would hold the names of the functions defined globally, and have a child symbol table for each of these functions, which will contain the local variables of that function. While adding new symbols to the symbol table, we first ensure that the type of the symbol is valid, and that the symbol has not been declared before, hence performing some semantic analysis and storing the semantic value of all the symbols.

Our AST classes have a similar hierarchy as our grammar rules, with a new appropriate node being created and added to our AST when successfully matching a rule. While traversing our AST, while performing operations on expressions, we check the types of both operands and ensure they are operable, else we raise a semantic error. Hence our semantic analysis phase ensures the input is well typed and returns the AST with semantic information to the code generation phase.

Intermediate Code Generation

Intermediate code generation is a phase to translate the source program into machine program. Intermediate code is generated because the compiler cannot generate machine code directly in one pass and thus, it first converts the source code into intermediate code which further performs efficient generation of machine code. The generated intermediate code is machine independent and thus, it can be executed on various platforms. It also simplifies the process of optimizing code efficiently as optimizing machine independent code is simpler and faster when compared to optimizing several machine dependent codes. Because of the machine-independent intermediate code, portability will be enhanced.

Our AST classes have a member function that handles the intermediate code generation for that particular type of expression or statement. The code gen phase takes the Abstract Syntax Tree obtained from the semantic analysis phase and generates intermediate code. Since, we do not have access to various architectures, we decided to generate the intermediate code in LLVM.

Development Environment

Flex

Flex is used to produce a lexical analyzer from a set of regular expressions and associated actions. The lexical analyzer reads code from an input file (in our instance a .tngt file) and produces a token stream which is later utilized by the parser to build an Abstract Syntax Tree.

Bison

Bison is used to convert a grammar description for a Look Ahead LR (LALR(1)) context-free grammar into a C program which is used to parse that grammar. The parser generated using Bison is a bottom-up parser which tries to reduce the entire input to a single grouping whose symbol is the grammar's start symbol by applying a series of shifts and reductions. The reason we chose Bison is that the parser generated is portable and does not require any specific compiler. It also has several extensions over Yacc.

GNU Make

We created a descriptive makefile as our build system to handle all the commands because we had to run the same set of commands every time we implemented a new feature / fixed a bug / added more functionality and it would have been painful if we had to do it manually every single time. We use Makefile to delete all previous intermediary files, compile and link the new object files before running the executable. We have separate commands to compile files for lexer, parser, semantic analysis, testing and cleaning up if one was to run individual components of the compiler. To keep our working tree clean, our Makefile creates a new directory to store all the generated files which can later be cleaned easily.
Our source files support conditional compilation to generate executables until different phases of the compiling process which can be easily called by our Makefile targets. Similarly, we have rules in our Makefile to build our parser documentation and visualize our grammar including the Deterministic Finite Automata. We also allow invoking the test suites for our lexer, parser, and semantic analysis stages easily from the Makefile.

Some other advantages of Make are:

- It automatically determines the proper order for updating files, in case one non source file depends on another non-source file.
- If we change a few source files and then run Make, it does not need to recompile all of your program. It updates only those non-source files that depend directly or indirectly on the source files that you changed.
- GNU Make has many powerful features for use in makefiles which other Make versions don't have. It can also regenerate, use and then delete intermediate file which need not be saved.

Git

We used Git as our version control system because of the several advantages it has over other version control systems. One major advantage is the branching capability which was heavily utilized while working on multiple features at the same time and making it easy to merge changes. We pushed our work into a remote on GitHub whenever one of us finished working on some part of the project so that the rest of the team could access it.

GitHub

We used GitHub for storing, tracking, and collaborating. GitHub provides so many unique features which were of immense help while keeping track of our goals and making collaboration easier. For instance, we used the project board feature provided by GitHub to keep track of the tasks which were pending at a given time. We also made use of GitHub actions to automate build and testing for every commit we made to the repository. The installation of flex and bison was cached as part of this step so that they wouldn't have to be installed when a commit was pushed. Another feature we used, although rarely, was the issues where we opened some issues we found as we progressed through the project which were fixed later on. Last but not least, we also created PRs (Pull Requests) whenever we were merging branches to review the code and make necessary changes before merging.

Visual Studio Code

We used Visual Studio Code (VS Code) as it is one of the leading text editors which provides a lot of features in the form of extensions such as syntax highlighting for lex and yacc files, auto-complete, integrating with git to visualize the commit graph, merge branches easily while fixing merge conflicts, and many more.

Bash Scripts

We have a template script set up in a bash file which runs a for loop to run test cases for the different phases of our compiler and heuristically check whether the test cases pass or not. This script is called from the Makefile when to run the appropriate tests. The bash script has pretty printing to show the current running test case, their output, and the test cases that fail.

Testing Plan and Testing Suite

Introduction

We made a wide variety of test cases to verify the working of our compiler. We broadly divided the test cases into four categories namely, Lexer, Parser and Semantic Analysis. Each of these categories consisted of two types of test cases, correct and incorrect.

1) Lexer:-

For the lexing phase, we wrote several test cases to match the different kinds of tokens in our code. On running the lexer tests, output files are generated with .token extension, wherein each symbol is matched to the appropriate token along with its location in the code file.
We've constructed our makefile in such a way that we can run all the lexer test cases at once. Once we run the command for running the lexer test cases, both correct and incorrect, they're executed and we can see the results.

```
Running testcases with lexicologically incorrect code for the Lexical Analyzer

Running Testcase (1/2): errortest1.tngt ~> errortest1.token
   Error: errortest1.tngt: Line 8:19 - '$' - Invalid token
   Error: errortest1.tngt: Line 9:13 - '_' - Invalid token
   Error: errortest1.tngt: Line 9:14 - '_' - Invalid token
   Error: errortest1.tngt: Line 9:15 - '_' - Invalid token
   Error: errortest1.tngt: Line 12:31 - '^' - Invalid token
   Error: errortest1.tngt: Line 12:32 - '^' - Invalid token
   Error: errortest1.tngt: Line 13:20 - '`' - Invalid token
   Error: errortest1.tngt: Line 13:21 - '`' - Invalid token
   Error: errortest1.tngt: Line 14:4 - '$' - Invalid token
   Error: errortest1.tngt: Line 14:5 - '$' - Invalid token
   Error: errortest1.tngt: Line 14:6 - '$' - Invalid token
   Error: errortest1.tngt: Line 14:8 - '~' - Invalid token
   Error: errortest1.tngt: Line 17:15 - '"' - Invalid token
        Testcase Passed, as the invalid tokens are rejected by the lexer

Running Testcase (2/2): errortest2.tngt ~> errortest2.token
   Error: errortest2.tngt: Line 5:25 - '_' - Invalid token
   Error: errortest2.tngt: Line 6:19 - '.' - Invalid token
   Error: errortest2.tngt: Line 7:20 - '$' - Invalid token
   Error: errortest2.tngt: Line 10:9 - '~' - Invalid token
   Error: errortest2.tngt: Line 10:10 - '~' - Invalid token
   Error: errortest2.tngt: Line 11:4 - '`' - Invalid token
   Error: errortest2.tngt: Line 11:5 - '`' - Invalid token
   Error: errortest2.tngt: Line 12:4 - '$' - Invalid token
   Error: errortest2.tngt: Line 12:5 - '$' - Invalid token
   Error: errortest2.tngt: Line 15:19 - '$' - Invalid token
   Error: errortest2.tngt: Line 15:20 - '$' - Invalid token
   Error: errortest2.tngt: Line 15:26 - '$' - Invalid token
   Error: errortest2.tngt: Line 15:27 - '$' - Invalid token
        Testcase Passed, as the invalid tokens are rejected by the lexer

Success: All invalid tokens were rejected by the Lexical Analyzer.  (Outputs can be viewed in tests/Lexer_Tests/Incorrect_codes/output)
```

Running the incorrect test cases

```
Running testcases with lexicologically correct code for the Lexical Analyzer

Running Testcase (1/8): comments.tngt ~> comments.token
        Testcase Passed

Running Testcase (2/8): dataTypes.tngt ~> dataTypes.token
        Testcase Passed

Running Testcase (3/8): family-and-functions.tngt ~> family-and-functions.token
        Testcase Passed

Running Testcase (4/8): ifElse.tngt ~> ifElse.token
        Testcase Passed

Running Testcase (5/8): illegal-code.tngt ~> illegal-code.token
        Testcase Passed

Running Testcase (6/8): literals.tngt ~> literals.token
        Testcase Passed

Running Testcase (7/8): loops-jumps.tngt ~> loops-jumps.token
        Testcase Passed

Running Testcase (8/8): operators.tngt ~> operators.token
        Testcase Passed

Success: All valid Lexical Analyzer testcases passed!  (Outputs can be viewed in tests/Lexer_Tests/Correct_codes/output)
```

Running the correct test cases

Examples:-

Test Case-1

Code:-

```
1  # lexer-test to check working comments      You, 17 hours ago • updated yytable and added testcases …
2  #sample program
3
4
5  driver<: :>
6  (:
7      #The comments will be ignored
8      #Comment-1
9      #comment-2
10 :)
```

Output:-

```
1  comments.tngt: Line 5:0 driver       was matched to the token DRIVER              (= 293)
2  comments.tngt: Line 5:6 <:           was matched to the token '('                 (= 40)
3  comments.tngt: Line 5:9 :>           was matched to the token ')'                 (= 41)
4  comments.tngt: Line 6:0 (:           was matched to the token '{'                 (= 123)
5  comments.tngt: Line 10:15   :)           was matched to the token '}'                  (= 125)
6
7
```

Observation:-

We can see that the lines of code which are comments have been ignored by our lexer.

Test Case-2

Code:-

```
#lexer-test to check matching of datatype keywords

driver<: :>
(:
    #primitive datatypes
    var int i;      You, 17 hours ago • updated yytable and added testcases …
    var bool b;
    var float f;
    var string s;
    const var int param1, param2;

    #non-primitive datatypes
    var Point x;
    var Path line;
    var Circle circ;
    var Image jpg;
    var Rectangle rect;
    var Ellipse elli;
    var Polygon pol;
    var Colour col;
    var Curve cur;

:)
```

Output:-

```
dataTypes.tngt: Line 3:0    driver      was matched to the token DRIVER             (= 293)
dataTypes.tngt: Line 3:6    <:          was matched to the token '('                (= 40)
dataTypes.tngt: Line 3:9    :>          was matched to the token ')'                (= 41)
dataTypes.tngt: Line 4:0    (:          was matched to the token '{'                (= 123)
dataTypes.tngt: Line 6:9    var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 6:13   int         was matched to the token INT                (= 262)
dataTypes.tngt: Line 6:17   i           was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 6:18   ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 7:4    var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 7:8    bool        was matched to the token BOOL               (= 260)
dataTypes.tngt: Line 7:13   b           was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 7:14   ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 8:4    var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 8:8    float       was matched to the token FLOAT              (= 261)
dataTypes.tngt: Line 8:14   f           was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 8:15   ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 9:4    var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 9:8    string      was matched to the token STRING             (= 263)
dataTypes.tngt: Line 9:15   s           was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 9:16   ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 10:4   const       was matched to the token CONST              (= 258)
dataTypes.tngt: Line 10:10  var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 10:14  int         was matched to the token INT                (= 262)
dataTypes.tngt: Line 10:18  param1      was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 10:24  ,           was matched to the token ','                (= 44)
dataTypes.tngt: Line 10:26  param2      was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 10:32  ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 13:9   var         was matched to the token VAR                (= 259)
dataTypes.tngt: Line 13:13  Point       was matched to the token POINT              (= 265)
dataTypes.tngt: Line 13:19  x           was matched to the token IDENTIFIER         (= 294)
dataTypes.tngt: Line 13:20  ;           was matched to the token ';'                (= 59)
dataTypes.tngt: Line 14:4   var         was matched to the token VAR                (= 259)
```

```
dataTypes.tngt: Line 14:8   Path        was matched to the token PATH              (= 266)
dataTypes.tngt: Line 14:13  line        was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 14:17  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 15:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 15:8   Circle      was matched to the token CIRCLE            (= 269)
dataTypes.tngt: Line 15:15  circ        was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 15:19  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 16:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 16:8   Image       was matched to the token IMAGE             (= 267)
dataTypes.tngt: Line 16:14  jpg         was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 16:17  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 17:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 17:8   Rectangle   was matched to the token RECTANGLE         (= 268)
dataTypes.tngt: Line 17:18  rect        was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 17:22  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 18:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 18:8   Ellipse     was matched to the token ELLIPSE           (= 270)
dataTypes.tngt: Line 18:16  elli        was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 18:20  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 19:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 19:8   Polygon     was matched to the token POLYGON           (= 271)
dataTypes.tngt: Line 19:16  pol         was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 19:19  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 20:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 20:8   Colour      was matched to the token COLOUR            (= 274)
dataTypes.tngt: Line 20:15  col         was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 20:18  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 21:4   var         was matched to the token VAR               (= 259)
dataTypes.tngt: Line 21:8   Curve       was matched to the token CURVE             (= 272)
dataTypes.tngt: Line 21:14  cur         was matched to the token IDENTIFIER        (= 294)
dataTypes.tngt: Line 21:17  ;           was matched to the token ';'               (= 59)
dataTypes.tngt: Line 23:0   :)          was matched to the token '}'               (= 125)
```

Observation:-

We can see that all the identifiers and datatypes, primitive and non-primitive, have been matched to the appropriate tokens.

Test Case-3

Code:-

```
#lexer-test for if-else stmt

drive<: :>
(:
    var int a := 2;
    #if-else
    if(a > 10)
    (:
        a /= 10;
    :)
    else
    (:
        a/= 4;
    :)


:)
```

Output:-

```
ifElse.tngt: Line 3:0   drive        was matched to the token IDENTIFIER          (= 294)
ifElse.tngt: Line 3:5   <:           was matched to the token '('                 (= 40)
ifElse.tngt: Line 3:8   :>           was matched to the token ')'                 (= 41)
ifElse.tngt: Line 4:0   (:           was matched to the token '{'                 (= 123)
ifElse.tngt: Line 5:4   var          was matched to the token VAR                 (= 259)
ifElse.tngt: Line 5:8   int          was matched to the token INT                 (= 262)
ifElse.tngt: Line 5:12  a            was matched to the token IDENTIFIER          (= 294)
ifElse.tngt: Line 5:14  :=           was matched to the token ASSIGN              (= 295)
ifElse.tngt: Line 5:17  2            was matched to the token INTEGER_LITERAL     (= 275)
ifElse.tngt: Line 5:18  ;            was matched to the token ';'                 (= 59)
ifElse.tngt: Line 7:9   if           was matched to the token IF                  (= 279)
ifElse.tngt: Line 7:11  (            was matched to the token '('                 (= 40)
ifElse.tngt: Line 7:12  a            was matched to the token IDENTIFIER          (= 294)
ifElse.tngt: Line 7:14  >            was matched to the token GR_THAN             (= 308)
ifElse.tngt: Line 7:16  10           was matched to the token INTEGER_LITERAL     (= 275)
ifElse.tngt: Line 7:18  )            was matched to the token ')'                 (= 41)
ifElse.tngt: Line 8:4   (:           was matched to the token '{'                 (= 123)
ifElse.tngt: Line 9:8   a            was matched to the token IDENTIFIER          (= 294)
ifElse.tngt: Line 9:10  /=           was matched to the token DIV_ASSIGN          (= 299)
ifElse.tngt: Line 9:13  10           was matched to the token INTEGER_LITERAL     (= 275)
ifElse.tngt: Line 9:15  ;            was matched to the token ';'                 (= 59)
ifElse.tngt: Line 10:4  :)           was matched to the token '}'                 (= 125)
ifElse.tngt: Line 11:4  else         was matched to the token ELSE                (= 280)
ifElse.tngt: Line 12:4  (:           was matched to the token '{'                 (= 123)
ifElse.tngt: Line 13:8  a            was matched to the token IDENTIFIER          (= 294)
ifElse.tngt: Line 13:9  /=           was matched to the token DIV_ASSIGN          (= 299)
ifElse.tngt: Line 13:12 4            was matched to the token INTEGER_LITERAL     (= 275)
ifElse.tngt: Line 13:13 ;            was matched to the token ';'                 (= 59)
ifElse.tngt: Line 14:4  :)           was matched to the token '}'                 (= 125)
ifElse.tngt: Line 17:0  :)           was matched to the token '}'                 (= 125)
```

Observation:-

We can see that the if else keywords have been matched to the appropriate tokens as well as the identifiers and literals.

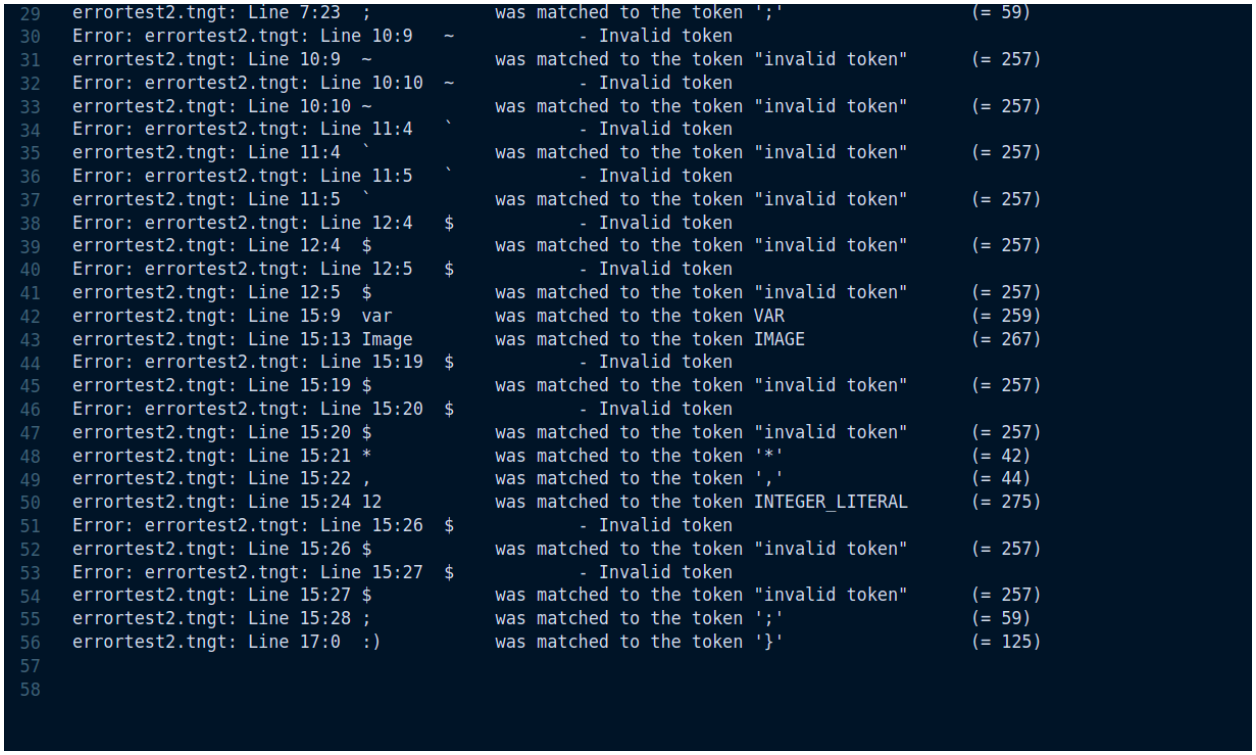
Test Case-4

Code:-

```
# This sample code consists of several unmatched sequences of characters
# The lexical analyzer will throw an error.

driver <: :>
(:
    #incorrect identifier names
    var int 3abc;
    var string abdh$;
    var bool ___;

    # appearance of illegal characters
    print("Hello World"); ^^
    var int a := 5; ``
    $$$ ~

    # open strings
    print("hello);

:)
```

Output:-

```
errortest2.tngt: Line 2:1   driiver      was matched to the token IDENTIFIER          (= 294)
errortest2.tngt: Line 2:9   <:           was matched to the token '('                 (= 40)
errortest2.tngt: Line 2:12  :>           was matched to the token ')'                 (= 41)
errortest2.tngt: Line 3:0   (:           was matched to the token '{'                 (= 123)
errortest2.tngt: Line 5:9   var          was matched to the token VAR                 (= 259)
errortest2.tngt: Line 5:13  int          was matched to the token INT                 (= 262)
errortest2.tngt: Line 5:17  a            was matched to the token IDENTIFIER          (= 294)
errortest2.tngt: Line 5:19  :=           was matched to the token ASSIGN              (= 295)
errortest2.tngt: Line 5:22  123          was matched to the token INTEGER_LITERAL     (= 275)
Error: errortest2.tngt: Line 5:25   _              - Invalid token
errortest2.tngt: Line 5:25  _            was matched to the token "invalid token"     (= 257)
errortest2.tngt: Line 5:26  4            was matched to the token INTEGER_LITERAL     (= 275)
errortest2.tngt: Line 5:27  ;            was matched to the token ';'                 (= 59)
errortest2.tngt: Line 6:4   var          was matched to the token VAR                 (= 259)
errortest2.tngt: Line 6:8   float        was matched to the token FLOAT               (= 261)
errortest2.tngt: Line 6:14  b            was matched to the token IDENTIFIER          (= 294)
errortest2.tngt: Line 6:16  :=           was matched to the token ASSIGN              (= 295)
Error: errortest2.tngt: Line 6:19   .            - Invalid token
errortest2.tngt: Line 6:19  .            was matched to the token "invalid token"     (= 257)
errortest2.tngt: Line 6:20  ;            was matched to the token ';'                 (= 59)
errortest2.tngt: Line 7:4   var          was matched to the token VAR                 (= 259)
errortest2.tngt: Line 7:8   int          was matched to the token INT                 (= 262)
errortest2.tngt: Line 7:12  c            was matched to the token IDENTIFIER          (= 294)
errortest2.tngt: Line 7:14  :=           was matched to the token ASSIGN              (= 295)
errortest2.tngt: Line 7:17  789          was matched to the token INTEGER_LITERAL     (= 275)
Error: errortest2.tngt: Line 7:20   $            - Invalid token
errortest2.tngt: Line 7:20  $            was matched to the token "invalid token"     (= 257)
errortest2.tngt: Line 7:21  89           was matched to the token INTEGER_LITERAL     (= 275)
```



Observations:-

We can see that due to the incorrect use of Identifier name format, or literal(in the case of string), many of the symbols are matched to the wrong tokens or are not matched to any token.

2) Parser

In the parsing phase, we ran multiple test cases to ensure that our parser catches different kinds of syntax errors such as incorrect closing and opening of braces, using the wrong symbol or ending a statement without the correct statement terminator, incorrectly defining loop statements and wrongful use of keywords.

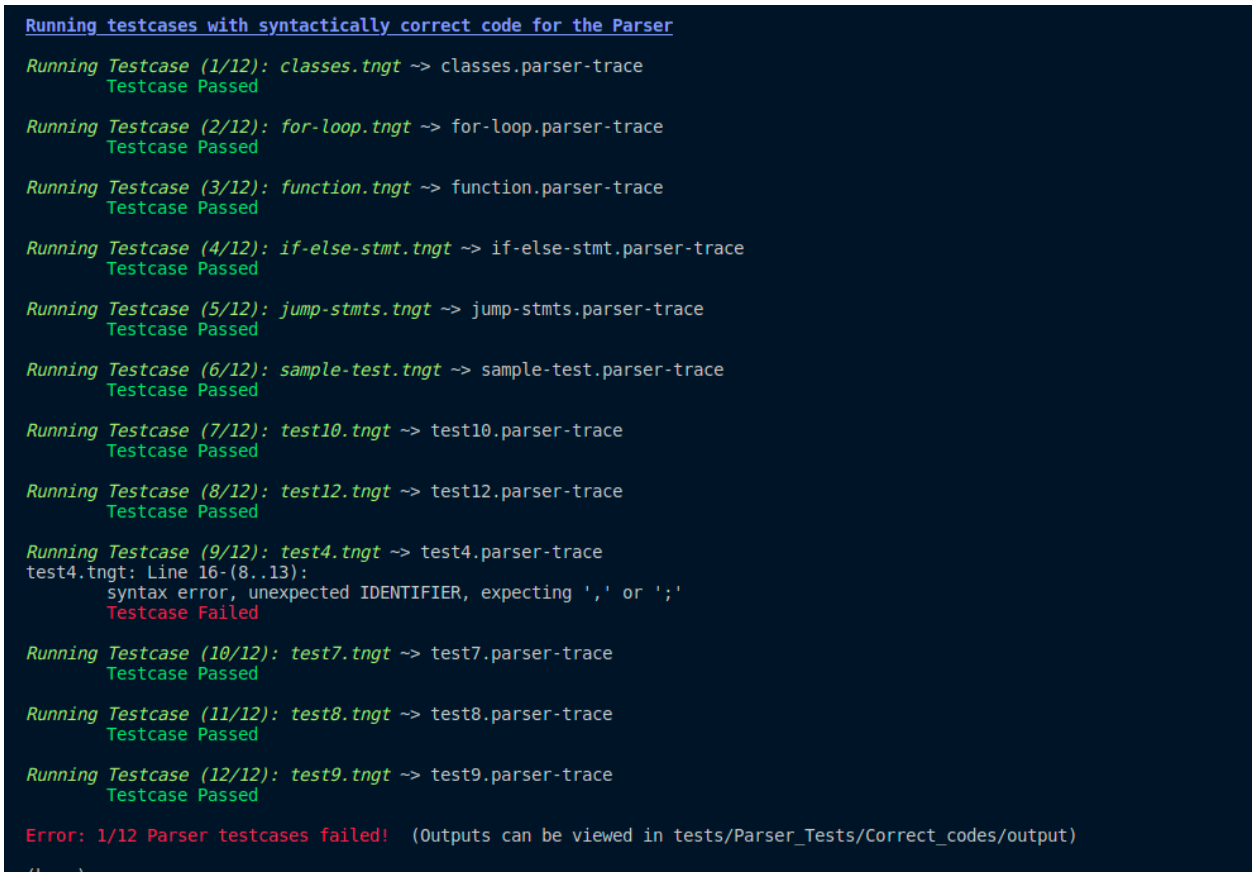
We have designed our testing plan in such a way that if a code which is supposed to be correct has errors, then our parser immediately catches them and that test case is deemed to have failed. As for the incorrect codes, all the syntactic errors are listed as can be observed below.

All the parser test cases are run with the help of a single command and the output files are generated and can be viewed in the appropriate output directory.

```
Running testcases with syntactically incorrect code for the Parser

Running Testcase (1/14): errortest10.tngt ~> errortest10.parser-trace
errortest10.tngt: Line 8-(4..6):
        syntax error, unexpected IF, expecting '{'
errortest10.tngt: Line 10-(4..8):
        syntax error, unexpected ELSE, expecting ')'
errortest10.tngt: Line 12-(4..6):
        syntax error, unexpected '}', expecting ')'
        Testcase Passed. All syntax errors detected

Running Testcase (2/14): errortest11.tngt ~> errortest11.parser-trace
errortest11.tngt: Line 3-(29..30):
        syntax error, unexpected ';', expecting ',' or ')'
        Testcase Passed. All syntax errors detected

Running Testcase (3/14): errortest12.tngt ~> errortest12.parser-trace
errortest12.tngt: Line 6-(5..7):
        syntax error, unexpected ASSIGN, expecting IDENTIFIER
        Testcase Passed. All syntax errors detected

Running Testcase (4/14): errortest13.tngt ~> errortest13.parser-trace
errortest13.tngt: Line 9-(18..19):
        syntax error, unexpected IDENTIFIER, expecting ';'
errortest13.tngt: Line 13-(21..22):
        syntax error, unexpected IDENTIFIER, expecting ';'
        Testcase Passed. All syntax errors detected

Running Testcase (5/14): errortest14.tngt ~> errortest14.parser-trace
errortest14.tngt: Line 5-(5..7):
        syntax error, unexpected ASSIGN, expecting IDENTIFIER
        Testcase Passed. All syntax errors detected

Running Testcase (6/14): errortest1.tngt ~> errortest1.parser-trace
errortest1.tngt: Line 3-(25..27):
        syntax error, unexpected ')', expecting IDENTIFIER
        Testcase Passed. All syntax errors detected

Running Testcase (7/14): errortest2.tngt ~> errortest2.parser-trace
errortest2.tngt: Line 6-(11..18):
        syntax error, unexpected IDENTIFIER, expecting ',' or ';'
errortest2.tngt: Line 14-(24..25):
        syntax error, unexpected IDENTIFIER, expecting ',' or ')'
        Testcase Passed. All syntax errors detected
```

```
Running Testcase (9/14): errortest4.tngt ~> errortest4.parser-trace
errortest4.tngt: Line 6-(12..13):
        syntax error, unexpected end of file
        Testcase Passed. All syntax errors detected

Running Testcase (10/14): errortest5.tngt ~> errortest5.parser-trace
errortest5.tngt: Line 5-(12..17):
        syntax error, unexpected FLOAT, expecting IDENTIFIER
errortest5.tngt: Line 6-(14..20):
        syntax error, unexpected STRING, expecting IDENTIFIER
errortest5.tngt: Line 7-(10..15):
        syntax error, unexpected FLOAT
errortest5.tngt: Line 8-(11..17):
        syntax error, unexpected STRING
        Testcase Passed. All syntax errors detected

Running Testcase (11/14): errortest6.tngt ~> errortest6.parser-trace
errortest6.tngt: Line 4-(0..1):
        syntax error, unexpected '(', expecting '{'
        Testcase Passed. All syntax errors detected

Running Testcase (12/14): errortest7.tngt ~> errortest7.parser-trace
errortest7.tngt: Line 6-(16..17):
        syntax error, unexpected ';', expecting ')'
errortest7.tngt: Line 6-(28..29):
        syntax error, unexpected ')', expecting ',' or ';'
        Testcase Passed. All syntax errors detected

Running Testcase (13/14): errortest8.tngt ~> errortest8.parser-trace
errortest8.tngt: Line 5-(17..20):
        syntax error, unexpected VAR
errortest8.tngt: Line 5-(43..44):
        syntax error, unexpected ')', expecting ',' or ';'
        Testcase Passed. All syntax errors detected

Running Testcase (14/14): errortest9.tngt ~> errortest9.parser-trace
errortest9.tngt: Line 3-(20..23):
        syntax error, unexpected FOR
        Testcase Passed. All syntax errors detected

Success: All syntax errors were caught by the parser!  (Outputs can be viewed in tests/Parser_Tests/Incorrect_codes/output)
```

Running incorrect test cases



Running correct test cases

Examples:-

Test Case-1

Code:

```
#using jump statements

driver<: :>
(:
    var int n := 100;
    for(var int i := 1; i <= n; i := i + 1)
    (:
        if(i%10 = 0)
        (:
            continue;
        :)

        if(i%11 = 0)
        (:
        print(i);
        break;
        :)
    :)

:)
```

Snippet of parser trace:-

```
Entering state 160
Stack now 0 19 32 40 51 66 100 160 150 205 224 234 239 242 244 66 100 160 145 200 219 229 66 100 160
Reading a token
Next token is token BREAK (16.8-12: )
Shifting token BREAK (16.8-12: )
Entering state 151
Stack now 0 19 32 40 51 66 100 160 150 205 224 234 239 242 244 66 100 160 145 200 219 229 66 100 160 151
Reading a token
Next token is token ';' (16.13: )
Shifting token ';' (16.13: )
Entering state 206
Stack now 0 19 32 40 51 66 100 160 150 205 224 234 239 242 244 66 100 160 145 200 219 229 66 100 160 151 206
Reducing stack by rule 133 (line 461):
   $1 = token BREAK (16.8-12: )
   $2 = token ';' (16.13: )
-> $$ = nterm jump_statement (16.8-13: )
Entering state 165
Stack now 0 19 32 40 51 66 100 160 150 205 224 234 239 242 244 66 100 160 145 200 219 229 66 100 160 165
Reducing stack by rule 111 (line 414):
   $1 = nterm jump_statement (16.8-13: )
-> $$ = nterm statement (16.8-13: )
Entering state 212
Stack now 0 19 32 40 51 66 100 160 150 205 224 234 239 242 244 66 100 160 145 200 219 229 66 100 160 212
```

Result:

```
Running Testcase (5/12): jump-stmts.tngt -> jump-stmts.parser-trace
        Testcase Passed
```

We can see that each of the tokens are processed one after the other and held in a stack.As there are no errors, the test case passes.

Test Case-2

Code:

```
#using if-else Statements

driver<: :>
(:
    var int ab := 8;
    if(ab%2 = 0)
    (:
        print("is even");
    :)
    else
    (:
        print("is odd");
    :)
:)

```

Snippet of parser trace:

```
Entering state 236
Stack now 0 19 32 40 51 66 100 160 145 200 219 229 236
Reading a token
Next token is token ELSE (10.4-7: )
Shifting token ELSE (10.4-7: )
Entering state 240
Stack now 0 19 32 40 51 66 100 160 145 200 219 229 236 240
Reading a token
Next token is token '{' (11.4-5: )
Shifting token '{' (11.4-5: )
Entering state 66
Stack now 0 19 32 40 51 66 100 160 145 200 219 229 236 240 66
Reading a token
Next token is token IDENTIFIER (12.8-12: )
Reducing stack by rule 114 (line 420):
-> $$ = nterm $@11 (11.6: )
Entering state 100
Stack now 0 19 32 40 51 66 100 160 145 200 219 229 236 240 66 100
Next token is token IDENTIFIER (12.8-12: )
Shifting token IDENTIFIER (12.8-12: )
Entering state 154
Stack now 0 19 32 40 51 66 100 160 145 200 219 229 236 240 66 100 154
Reading a token
Next token is token '(' (12.13: )
Reducing stack by rule 72 (line 363):
   $1 = token IDENTIFIER (12.8-12: )
-> $$ = nterm variable (12.8-12: )
```

Result:

```
Running Testcase (4/12): if-else-stmt.tngt ~> if-else-stmt.parser-trace
        Testcase Passed
```

As there are no errors, the test case passes.

Test Case-3

Code:

```
#incorrect/missing semicolon

void calculateAns<: var int a; :>
(:
    a = a * 10;
    print(a)
:) ;

driver<: :>
(:
    var int a := 34
    calculateAns(a)
:)
```

Snippet of parser trace:

```
Entering state 167
Stack now 0 25 35 41 53 70 103 167
Reducing stack by rule 49 (line 269):
   $1 = token VAR (3.20-22: )
   $2 = nterm type (3.24-26: )
   $3 = token IDENTIFIER (3.28: )
-> $$ = nterm arg (3.20-28: )
Entering state 73
Stack now 0 25 35 41 53 73
Reducing stack by rule 46 (line 259):
   $1 = nterm arg (3.20-28: )
-> $$ = nterm args_list (3.20-28: )
Entering state 72
Stack now 0 25 35 41 53 72
Reading a token
Next token is token ';' (3.29: )
missing-semicolon.tngt: Line 3-(29..30):
   syntax error, unexpected ';', expecting ',' or ')'
Error: popping nterm args_list (3.20-28: )
Stack now 0 25 35 41 53
Error: popping nterm $@5 (3.19: )
Stack now 0 25 35 41
Error: popping token '(' (3.17-18: )
Stack now 0 25 35
Error: popping token IDENTIFIER (3.5-16: )
```

Result:

```
Running Testcase (12/12): missing-semicolon.tngt ~> missing-semicolon.parser-trace
missing-semicolon.tngt: Line 3-(29..30):
        syntax error, unexpected ';', expecting ',' or ')'
```

In the above code, the semicolon is missing for few of the statements and so the error is caught by the parser as we can see above.

Test Case-4

Code:

```
1   #reserverd keywords used as variable names
2
3   driver<: :>
4   (:
5       var int float := 12;
6       var float string := 24.5;
7       print(float);
8       print (string);
9   :)
```

Snippet of parser trace:

```
41  Entering state 5
42  Stack now 0 19 32 40 51 66 100 2 5
43  Reducing stack by rule 11 (line 166):
44     $1 = token INT (5.8-10: )
45  -> $$ = nterm type (5.8-10: )
46  Entering state 30
47  Stack now 0 19 32 40 51 66 100 2 30
48  Reducing stack by rule 31 (line 192):
49  -> $$ = nterm $@2 (5.11: )
50  Entering state 37
51  Stack now 0 19 32 40 51 66 100 2 30 37
52  Reading a token
53  Next token is token FLOAT (5.12-16: )
54  incorrect-identifier-name.tngt: Line 5-(12..17):
55     syntax error, unexpected FLOAT, expecting IDENTIFIER
```

Result:

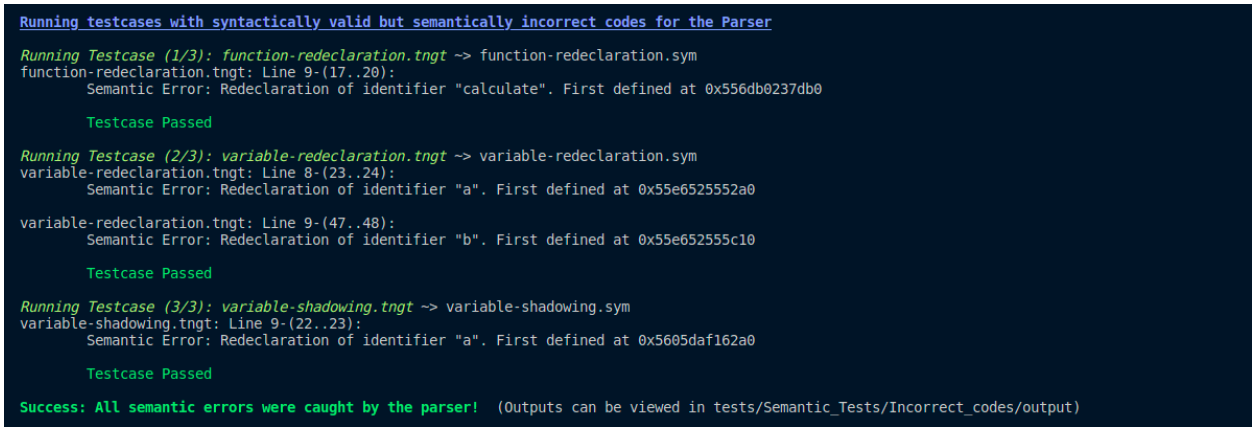
```
Running Testcase (11/12): incorrect-identifier-name.tngt ~> incorrect-identifier-name.parser-trace
incorrect-identifier-name.tngt: Line 5-(12..17):
        syntax error, unexpected FLOAT, expecting IDENTIFIER
incorrect-identifier-name.tngt: Line 6-(14..20):
        syntax error, unexpected STRING, expecting IDENTIFIER
incorrect-identifier-name.tngt: Line 7-(10..15):
        syntax error, unexpected FLOAT
incorrect-identifier-name.tngt: Line 8-(11..17):
        syntax error, unexpected STRING
```

In the above code, keywords(datatypes) have been used incorrectly as identifier names, and so the parser catches these errors.

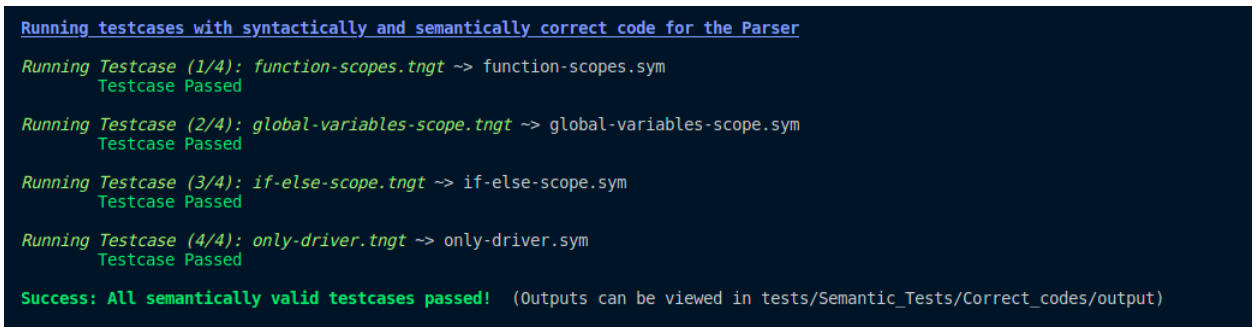
3) Semantic Analysis

We use our symbol table to do some basic semantic checks.

Our global symbol table holds the names of functions/variables defined globally, and there are child symbol tables for each of these functions, which contain their respective local variables. While adding new symbols to the table, we assert that the type info of the symbol is valid and that the symbol has not been declared before, hence performing semantic analysis.



Running semantically incorrect codes



Running semantically correct codes

Examples:

Test Case-1

Code:

```
#sample program to check for semantics and working of symbol table

driver<: :>
(:
    print("Hello World");
    var int a := 4;
    var int b := 10;
    var int sum := a+b;
    print(sum);
:)
```

Symbol Table:

```
(global)
-------------------------------------------------------------------------------------------
|   Name          |    Typename      |    Kind                    |  Location              |
-------------------------------------------------------------------------------------------
|   Circle        |    Circle        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Color         |    Color         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Curve         |    Curve         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Ellipse       |    Ellipse       |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Image         |    Image         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Path          |    Path          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Point         |    Point         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Polygon       |    Polygon       |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Rectangle     |    Rectangle     |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   bool          |    bool          |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   draw          |    void          |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   driver        |    void->()      |    Function                 |  line 3-(0..6)            |
-------------------------------------------------------------------------------------------
|   float         |    float         |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   int           |    int           |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   print         |    void          |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   string        |    string        |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   void          |    void          |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------
```

```
--->(driver)
    |
    |   --->(driver::0)
    |       -------------------------------------------------------------------------------------------
    |       |   Name          |    Typename      |    Kind                    |  Location              |
    |       -------------------------------------------------------------------------------------------
    |       |   a             |    int           |    Primitive variable      |  line 6-(12..13)       |
    |       -------------------------------------------------------------------------------------------
    |       |   b             |    int           |    Primitive variable      |  line 7-(12..13)       |
    |       -------------------------------------------------------------------------------------------
    |       |   sum           |    int           |    Primitive variable      |  line 8-(12..15)       |
    |       -------------------------------------------------------------------------------------------
```

Observations:

We can see the generated symbol table, with the global symbol table and the child symbol tables for each of the global functions.

Test Case-2

Code:

```
#semantic and symbol table test for global variables

var int val := 10;
var bool check := false;
var string word := "Testing";

driver<: :>
(:
    var int a := 6+val;
    var int b := 4-val;
    var int c := (a+b)*val;
    print(c);
    print(word);
:)
```

Symbol Table:

```
(global)
-------------------------------------------------------------------------------------------------
|   Name          |    Typename       |    Kind                    |  Location                |
-------------------------------------------------------------------------------------------------
|   Circle        |    Circle         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Color         |    Color          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Curve         |    Curve          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Ellipse       |    Ellipse        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Image         |    Image          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Path          |    Path           |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Point         |    Point          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Polygon       |    Polygon        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Rectangle     |    Rectangle      |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   bool          |    bool           |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   check         |    bool           |    Primitive variable       |  line 4-(9..14)          |
-------------------------------------------------------------------------------------------------
|   draw          |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   driver        |    void->()       |    Function                 |  line 7-(0..6)           |
-------------------------------------------------------------------------------------------------
|   float         |    float          |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   int           |    int            |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   print         |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   string        |    string         |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   val           |    int            |    Primitive variable       |  line 3-(8..11)          |
-------------------------------------------------------------------------------------------------
|   void          |    void           |    Inbuilt primitive typename|  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   word          |    string         |    Primitive variable       |  line 5-(11..15)         |
-------------------------------------------------------------------------------------------------
```

```
--->(driver)

    --->(driver::0)
        -------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                    |  Location                |
        -------------------------------------------------------------------------------------------
        |   a             |    int            |    Primitive variable      |  line 9-(12..13)         |
        -------------------------------------------------------------------------------------------
        |   b             |    int            |    Primitive variable      |  line 10-(12..13)        |
        -------------------------------------------------------------------------------------------
        |   c             |    int            |    Primitive variable      |  line 11-(12..13)        |
        -------------------------------------------------------------------------------------------
```

Observations:

We can note that the global symbols have been added to the global symbol table.

Test Case-3

Code:

```
#semantic and symbol table test for function scope

void add<: var int a, var int b :>
(:
    var int ans := a+b;
    print(ans);
:)

void multiply<: var int a, var int b :>
(:
    var int ans := a*b;
    print(ans);
:)

void divide<: var float c, var float d :>
(:
    var float ans := c/d;
    print(ans);
:)

bool checkOddEve<: var int n :>
(:
    var bool ans;
    if(n % 2 = 0)
    (:
        ans = true;
    :)
    else
    (:
        ans = false;
    :)
    send ans;
:)

driver<: :>
(:
    var int a := 6, b := 7;
    var float c := 3.4, d := 12.35;
    var bool result := checkOddEve(a);
    add(a,b);
    multiply(a,b);
    divide(c,d);
    print(result);
:)
```

Symbol Table:

```
(global)
-------------------------------------------------------------------------------------------------
|   Name          |    Typename       |    Kind                   |  Location                 |
-------------------------------------------------------------------------------------------------
|   Circle        |    Circle         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Color         |    Color          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Curve         |    Curve          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Ellipse       |    Ellipse        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Image         |    Image          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Path          |    Path           |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Point         |    Point          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Polygon       |    Polygon        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Rectangle     |    Rectangle      |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   add           |    void->(int,int)|    Function                 |  line 3-(0..4)            |
-------------------------------------------------------------------------------------------------
|   bool          |    bool           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   checkOddEve   |    bool->(int)    |    Function                 |  line 21-(0..4)           |
-------------------------------------------------------------------------------------------------
|   divide        |    void->(float,float)|    Function               |  line 15-(0..4)           |
-------------------------------------------------------------------------------------------------
|   draw          |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   driver        |    void->()       |    Function                 |  line 35-(0..6)           |
-------------------------------------------------------------------------------------------------
|   float         |    float          |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   int           |    int            |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   multiply      |    void->(int,int)|    Function                 |  line 9-(0..4)            |
-------------------------------------------------------------------------------------------------
|   print         |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   string        |    string         |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   void          |    void           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
```

```
--->(add)
    ----------------------------------------------------------------------------------------------
    |   Name          |    Typename       |    Kind                   |  Location               |
    ----------------------------------------------------------------------------------------------
    |   a             |    int            |    Primitive variable     |  line 3-(19..20)        |
    ----------------------------------------------------------------------------------------------
    |   b             |    int            |    Primitive variable     |  line 3-(30..31)        |
    ----------------------------------------------------------------------------------------------

    --->(add::0)
        ----------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                   |  Location               |
        ----------------------------------------------------------------------------------------------
        |   ans           |    int            |    Primitive variable     |  line 5-(12..15)        |
        ----------------------------------------------------------------------------------------------

--->(checkOddEve)
    ----------------------------------------------------------------------------------------------
    |   Name          |    Typename       |    Kind                   |  Location               |
    ----------------------------------------------------------------------------------------------
    |   n             |    int            |    Primitive variable     |  line 21-(27..28)       |
    ----------------------------------------------------------------------------------------------

    --->(checkOddEve::0)
        ----------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                   |  Location               |
        ----------------------------------------------------------------------------------------------
        |   ans           |    bool           |    Primitive variable     |  line 23-(13..16)       |
        ----------------------------------------------------------------------------------------------

        --->(checkOddEve::0::0)

--->(divide)
    ----------------------------------------------------------------------------------------------
    |   Name          |    Typename       |    Kind                   |  Location               |
    ----------------------------------------------------------------------------------------------
    |   c             |    float          |    Primitive variable     |  line 15-(24..25)       |
    ----------------------------------------------------------------------------------------------
    |   d             |    float          |    Primitive variable     |  line 15-(37..38)       |
    ----------------------------------------------------------------------------------------------

    --->(divide::0)
        ----------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                   |  Location               |
        ----------------------------------------------------------------------------------------------
        |   ans           |    float          |    Primitive variable     |  line 17-(14..17)       |
        ----------------------------------------------------------------------------------------------
```

```
--->(driver)

    --->(driver::0)
        ----------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                   |  Location               |
        ----------------------------------------------------------------------------------------------
        |   a             |    int            |    Primitive variable     |  line 37-(12..13)       |
        ----------------------------------------------------------------------------------------------
        |   b             |    int            |    Primitive variable     |  line 37-(20..21)       |
        ----------------------------------------------------------------------------------------------
        |   c             |    float          |    Primitive variable     |  line 38-(14..15)       |
        ----------------------------------------------------------------------------------------------
        |   d             |    float          |    Primitive variable     |  line 38-(24..25)       |
        ----------------------------------------------------------------------------------------------
        |   result        |    bool           |    Primitive variable     |  line 39-(13..19)       |
        ----------------------------------------------------------------------------------------------

--->(multiply)
    ----------------------------------------------------------------------------------------------
    |   Name          |    Typename       |    Kind                   |  Location               |
    ----------------------------------------------------------------------------------------------
    |   a             |    int            |    Primitive variable     |  line 9-(24..25)        |
    ----------------------------------------------------------------------------------------------
    |   b             |    int            |    Primitive variable     |  line 9-(35..36)        |
    ----------------------------------------------------------------------------------------------

    --->(multiply::0)
        ----------------------------------------------------------------------------------------------
        |   Name          |    Typename       |    Kind                   |  Location               |
        ----------------------------------------------------------------------------------------------
        |   ans           |    int            |    Primitive variable     |  line 11-(12..15)       |
        ----------------------------------------------------------------------------------------------
```

Observations:

We can observe the child symbol tables for all the functions created in the global space.

Test Case-4

Code:

```
#semantic-test for wrongful redeclaration of function Name

void calculate<: var int a, var int b:>
(:
    var int addn := a+b;
    print(addn);
:)

void calculate<: var int c, var int d:>
(:
    var int mult := c*d;
    print(mult);
:)

driver<: :>
(:
    var int a := 2;
    var int b := 3;
    calculate(a,b);
:)
```

Symbol Table:

```
function-redeclaration.tngt: Line 9-(17..20):
    Semantic Error: Redeclaration of identifier "calculate". First defined at 0x560e82066db0

      | Kind                    |  Location                |
-------------------------------------------------------------------------------------------
|   Circle         |    Circle         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Color          |    Color          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Curve          |    Curve          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Ellipse        |    Ellipse        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Image          |    Image          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Path           |    Path           |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Point          |    Point          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Polygon        |    Polygon        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   Rectangle      |    Rectangle      |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   bool           |    bool           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------
|   calculate      |    void->(int,int)|    Function                 |  line 9-(0..4)           |
-------------------------------------------------------------------------------------------
|   draw           |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   driver         |    void->()       |    Function                 |  line 15-(0..6)          |
-------------------------------------------------------------------------------------------
|   float          |    float          |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------
|   int            |    int            |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------
|   print          |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------
|   string         |    string         |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------
|   void           |    void           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------
```

```
--->(calculate)
    ---------------------------------------------------------------------------------------------
    |   Name           |    Typename       |    Kind                    |  Location              |
    ---------------------------------------------------------------------------------------------
    |   c              |    int            |    Primitive variable      |  line 9-(25..26)       |
    ---------------------------------------------------------------------------------------------
    |   d              |    int            |    Primitive variable      |  line 9-(36..37)       |
    ---------------------------------------------------------------------------------------------

    --->(calculate::0)
        ---------------------------------------------------------------------------------------------
        |   Name           |    Typename       |    Kind                    |  Location              |
        ---------------------------------------------------------------------------------------------
        |   mult           |    int            |    Primitive variable      |  line 11-(12..16)      |
        ---------------------------------------------------------------------------------------------

--->(driver)

    --->(driver::0)
        ---------------------------------------------------------------------------------------------
        |   Name           |    Typename       |    Kind                    |  Location              |
        ---------------------------------------------------------------------------------------------
        |   a              |    int            |    Primitive variable      |  line 17-(12..13)      |
        ---------------------------------------------------------------------------------------------
        |   b              |    int            |    Primitive variable      |  line 18-(12..13)      |
        ---------------------------------------------------------------------------------------------
```

Observations:

We can note that since the function has already been defined once, a semantic error is thrown.

Test Case-5

Code:

```
#semantic-test for variable redeclaration


driver<: :>
(:
    var int a := 5;
    var int b := 10;
    var float a := 3.45;    #illegally declared again
    var int b := 3;         #illegally declared again
:)
```

Symbol Table:

```
variable-redeclaration.tngt: Line 8-(23..24):
    Semantic Error: Redeclaration of identifier "a". First defined at 0x559fcf6d72a0

variable-redeclaration.tngt: Line 9-(47..48):
    Semantic Error: Redeclaration of identifier "b". First defined at 0x559fcf6d7c10

-------------------------------------
|   Circle         |    Circle         |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Color          |    Color          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Curve          |    Curve          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Ellipse        |    Ellipse        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Image          |    Image          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Path           |    Path           |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Point          |    Point          |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Polygon        |    Polygon        |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   Rectangle      |    Rectangle      |    Inbuilt family typename  |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   bool           |    bool           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   draw           |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   driver         |    void->()       |    Function                 |  line 4-(0..6)            |
-------------------------------------------------------------------------------------------------
|   float          |    float          |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   int            |    int            |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   print          |    void           |    Inbuilt function         |  Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   string         |    string         |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------
|   void           |    void           |    Inbuilt primitive typename| Language defined symbol  |
-------------------------------------------------------------------------------------------------

--->(driver)

    --->(driver::0)
        ---------------------------------------------------------------------------------------------
        |   Name           |    Typename       |    Kind                   |  Location               |
        ---------------------------------------------------------------------------------------------
        |   a              |    float          |    Primitive variable     |  line 8-(14..15)        |
        ---------------------------------------------------------------------------------------------
        |   b              |    int            |    Primitive variable     |  line 9-(41..42)        |
        ---------------------------------------------------------------------------------------------
```

Observations:

We can note that since the variables have been already defined, we cannot redefine them and hence a semantic error is thrown.

4) Code Generation

We could not complete the code generation part, and so we could not do the testing for it. The approach remains similar to the above parts, first we generate some test input and generate the expected output. Then we compare the said output to the output the compiler generates. Since we know that LLVM is well tested we can trust the LLVM output of any given LLVM IR.

Therefore, to test the code gen we will take some input and generate the corresponding LLVM IR and compare it with the output of the tangent compiler(if it generated the IR).

To deal with the errors we come across, one approach we could follow is random testing. Although there are no set of compiler options which completely eliminates all the errors, running a large number of test cases would certainly help in reducing the faced errors to a significant extent.

CONCLUSION

- In this project, we got insight into the process of designing a compiler. Although we successfully implemented only the first few phases, we definitely learnt a lot.
- We learnt how to write regular expressions to represent various constructs in a language
- We learnt how to write BNG grammar for a language and how to implement the said grammar. We also learnt how to recognise and fix SR and RR conflicts
- We learnt the inner working and implementation of symbol table. We also learnt how to implement scopes through symbol table.
- We learnt how to represent a program as an AST and the proprocess required to convert code to an AST.
- We got some insight into planning what syntax a language should have. The DOs and DONTs of designing a programming language is more of an art than science.
- The learnt how to piece together the above components and have them work together
- We learnt how to test the above components to make sure the components work as expected individually and as a whole.
- Towards the ending of the project timeline we ran out of time to finish implementing code gen but we still gained experience in doing so. We learnt how to concert our AST to LLVM IR using the LLVM classes and libraries.

APPENDIX

Language Specifications:

- The Cool Reference Manual
- The Java Language: A White Paper

C lex file: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html
Sample Project: https://github.com/kaushiksk/mini-c-compiler
Flex manpage: http://dinosaur.compilertools.net/flex/manpage.html

---

Links shared by the TA (Anilava Bhaiya)

- **Sample Project: Modern Compiler in C**
- **Youtube Playlist for semantic and Parsing** **(watch 34 to 44)**
- **Sample Project for Syntax Tree**

ASTs->

Building AST: **https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/**
https://github.com/lsegal/my_toy_compiler

Lex/Flex Book: https://web.iitd.ac.in/~sumeet/flex__bison.pdf
Constructs Doc: **Constructs**
Building AST (useful) :
http://web.eecs.utk.edu/~bvanderz/teaching/cs461Sp11/notes/parse_tree/
Kaleidoscope

Code Gen ->

LLVM Tutorial #9: CodeGen for Functions
