Mini Project

Compiler Frontend and Translation

Name: Akash.S
USN: 1NH16CS002

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

A compiler is a computer program that transforms computer codewritten in one programming language (the source language) into another programming language (the target language). The name *compiler* is primarily used for programs that translate source code from a high-level programming langauge to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

A compiler implements a formal transformation from a high-level source program to a low-level target program. Compiler design can define an end to end solution or tackle a defined subset that interfaces with other compilation tools e.g. preprocessors, assemblers, linkers. Design requirements include rigorously defined interfaces both internally between compiler components and externally between supporting toolsets.

- The *front end* verifies syntax and semantics according to a specific source language. For statically typed languages it performs type checking by collecting type information. If the input program is syntactically incorrect or has a type error, it generates errors and warnings, highlighting them on the source code. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis. The front end transforms the input program into an intermediate representation (IR) for further processing by the middle end. This IR is usually a lower-level representation of the program with respect to the source code.
- The *middle end* performs optimizations on the IR that are independent of the CPU architecture being targeted. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

propagation), relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. Eventually producing the "optimized" IR that is used by the back end.

- The *back end* takes the optimized IR from the middle end. It may perform more analysis, transformations and optimizations that are specific for the target CPU architecture. The back end generates the target-dependent assembly code, performing register allocation in the process. The back end performs instruction

  scheduling, which re-orders instructions to keep parallel execution units busy by

  filling delay slots. Although most algorithms for optimization are NP-hard, heuristic techniques are well-developed and currently implemented in production-quality compilers. Typically the output of a back end is machine code specialized for a particular processor and operating system.

1.2 PROBLEM DEFINITION

To design and implement the *front end* of a compiler that takes as input a program written with a pre-defined (syntax and grammer) and do the lexical analysing, syntax analysing, semantic analysing on the input program and in the end emit c++ code from the gathered data. In short it is a ***source-to-source*** compiler.

1.3 REQUIREMENTS

- Machine that supports Java JRE.
- gcc compiler.

CHAPTER 2

JAVA FEATURES AND OOPS CONCEPT

2.1

Features of Java:

1) Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand.

According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2) Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-

oriented means we organize our software as a combination of different types of objects

that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

3) Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

4) Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside a virtual machine sandbox

5) Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6) Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

2.2

OOPS Concepts:

Object: An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Ex: SyntaxChecker sc = new SyntaxChecker(sq_arr, my_file, quotes_range_indices, id_number, id_char_index);

Class: A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

Ex: SequenceTypeInfo, SyntaxChecker, LexicalAnalyser, SymbolTable

Inheritance: Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

Ex:

```
class StructInfo extends Info {
        String name;
        List<VarDeclInfo> var_infos; }
```

```
class EnumInfo extends Info {
        String name;
        List<String> values;
}
```

Polymorphism: Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Ex:

```
List<Info> infos = new ArrayList<>();
For(Info info: infos) {
        info.validate();
}
```

Encapsulation: Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixture of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

- **CHAPTER 3**

Requirements and Design

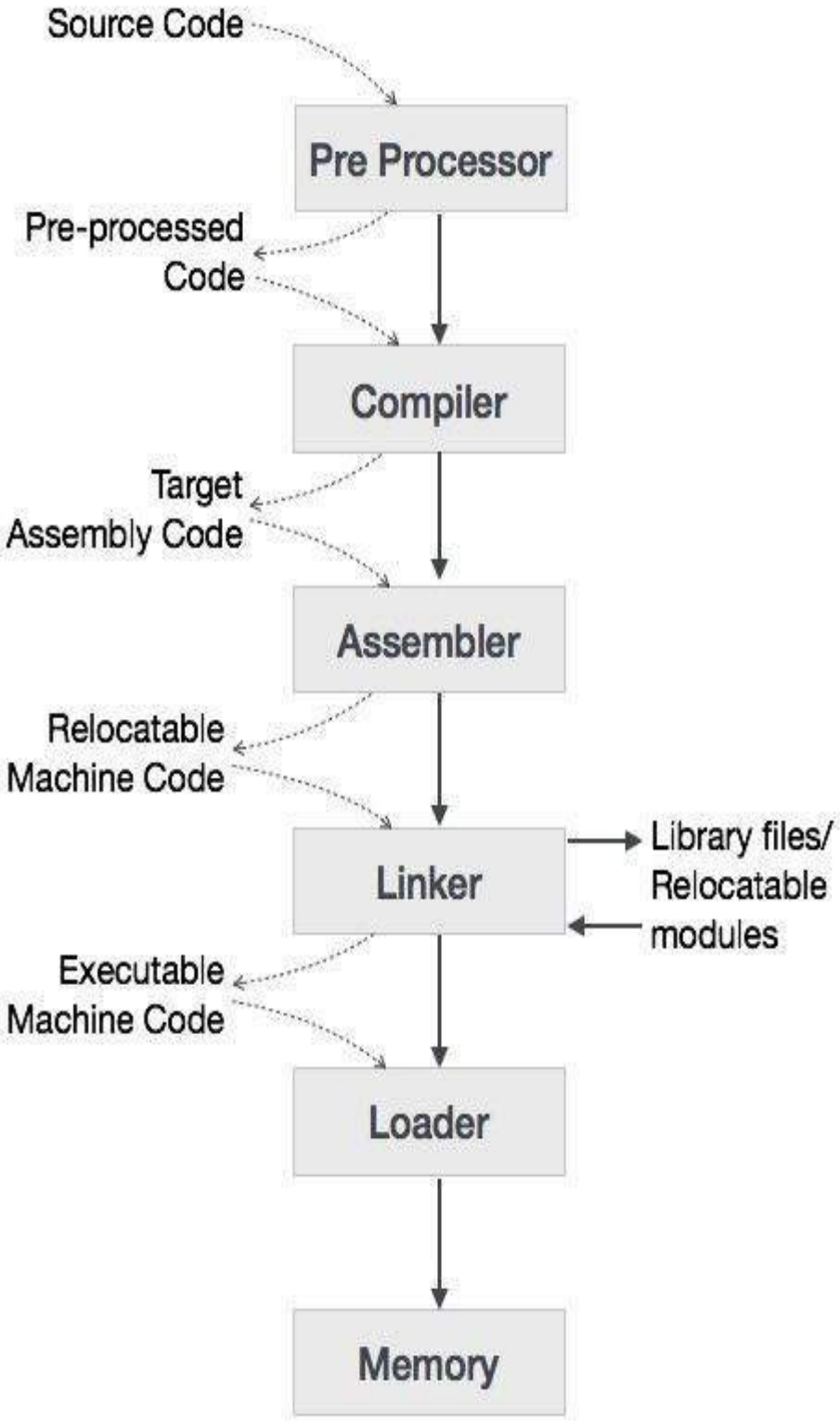
3.1 Hardware Specification

RAM: 128 MB
Disk space: 124 MB for JRE; 2 MB for Java Update
Processor: Minimum Pentium 2 266 MHz processor
Browsers: Internet Explorer 9 and above, Firefox

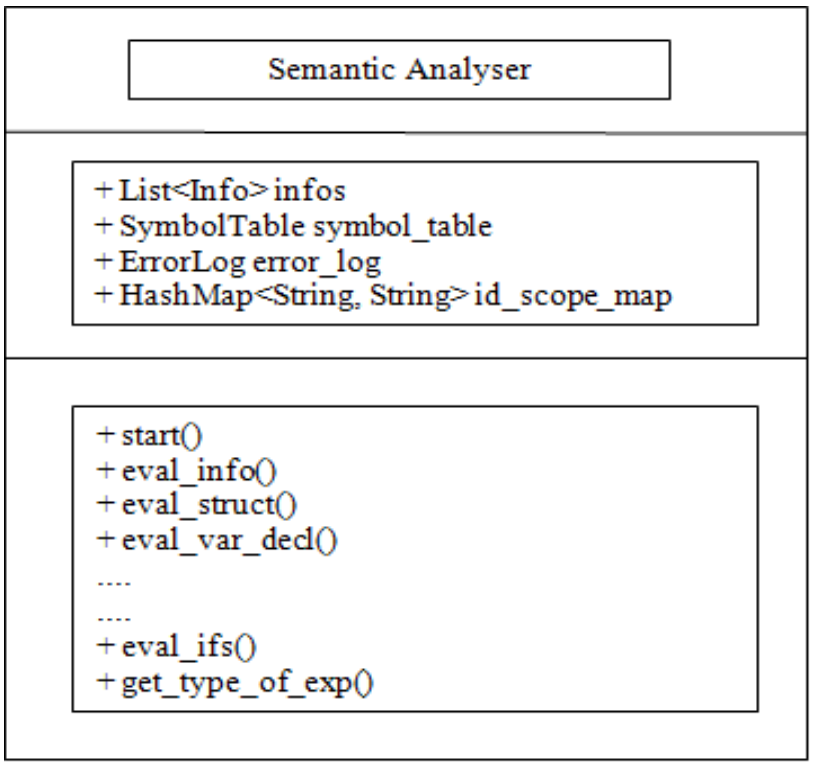
3.2 Software Specification

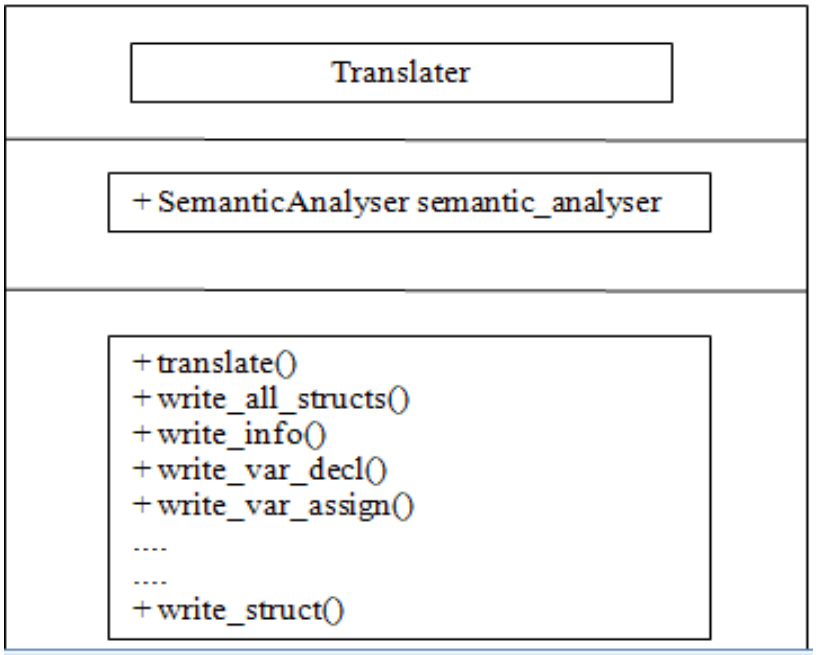
- Purpose
  - Definition
    A **compiler** is a computer program that translates computer code written in one programming language (the source language) into another programming language (the target language). The name *compiler* is primarily used for programs that translate source code from a high-level programming language to a lower-level programming language (e.g., assembly language, object code, or machine code) to create an executable program.
  - Background
    Software for early computers was primarily written in assembly language. It is usually more productive for a programmer to use a high-level language, and programs written in a high-level language can be reused on different kinds of computers. Even so, it took a while for compilers to become established, because they generated code that did not perform as well as hand-written assembler, they were daunting development projects in their own right, and the very limited memory capacity of early computers created many technical problems for practical compiler implementations.

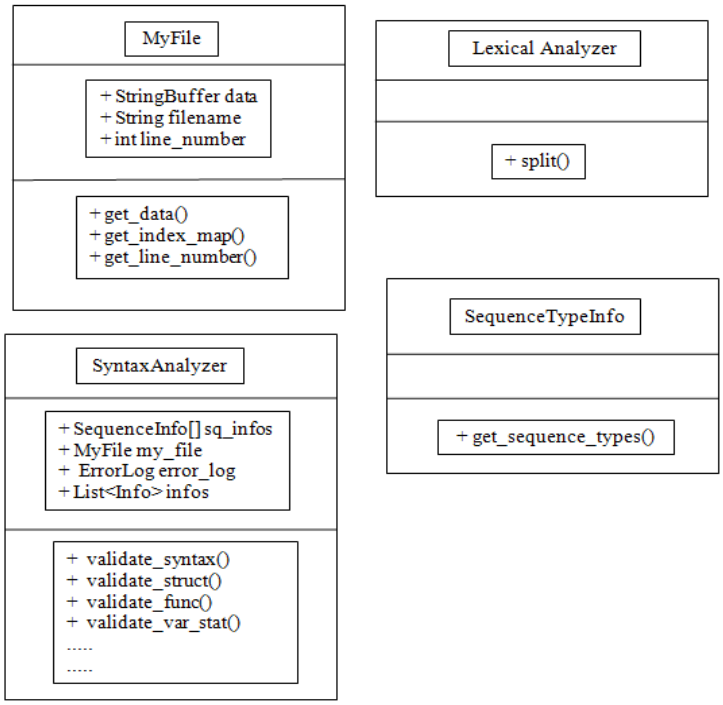
- System overview



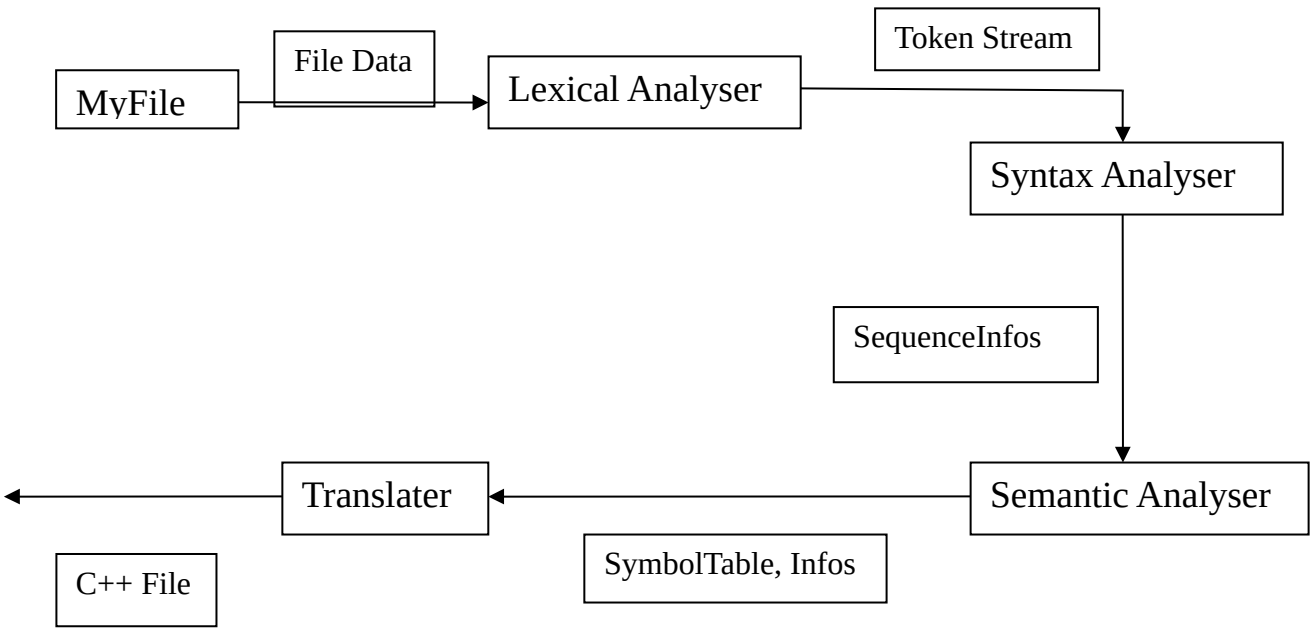
3.3 Class Diagram



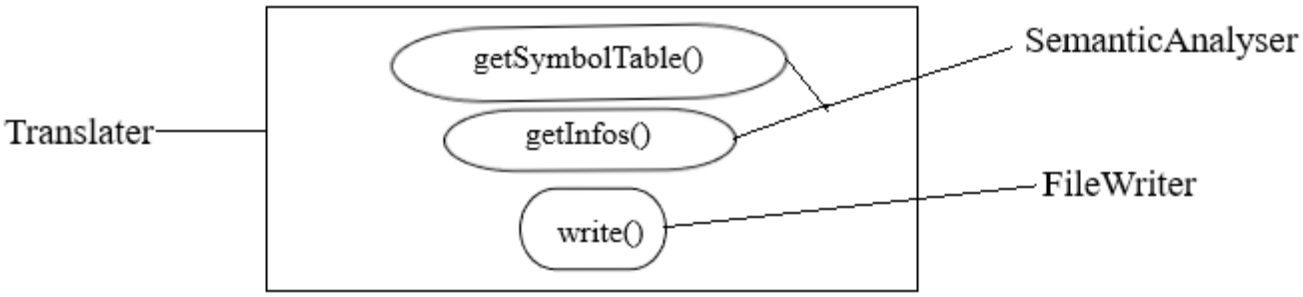


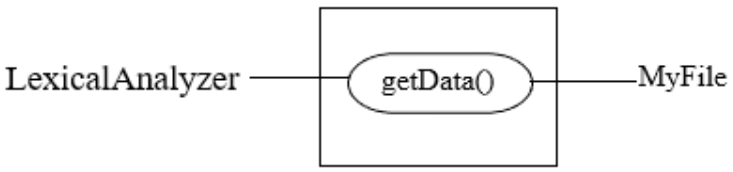


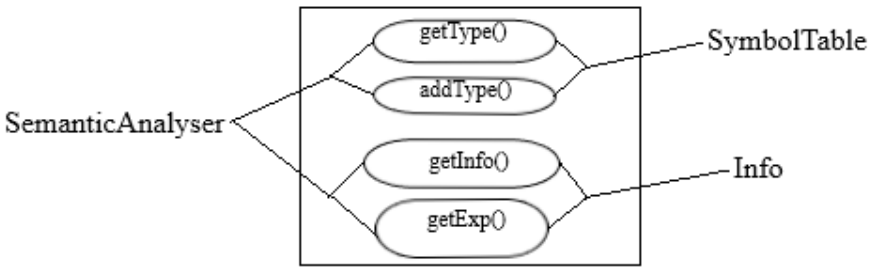
3.4 Data Flow Diagram



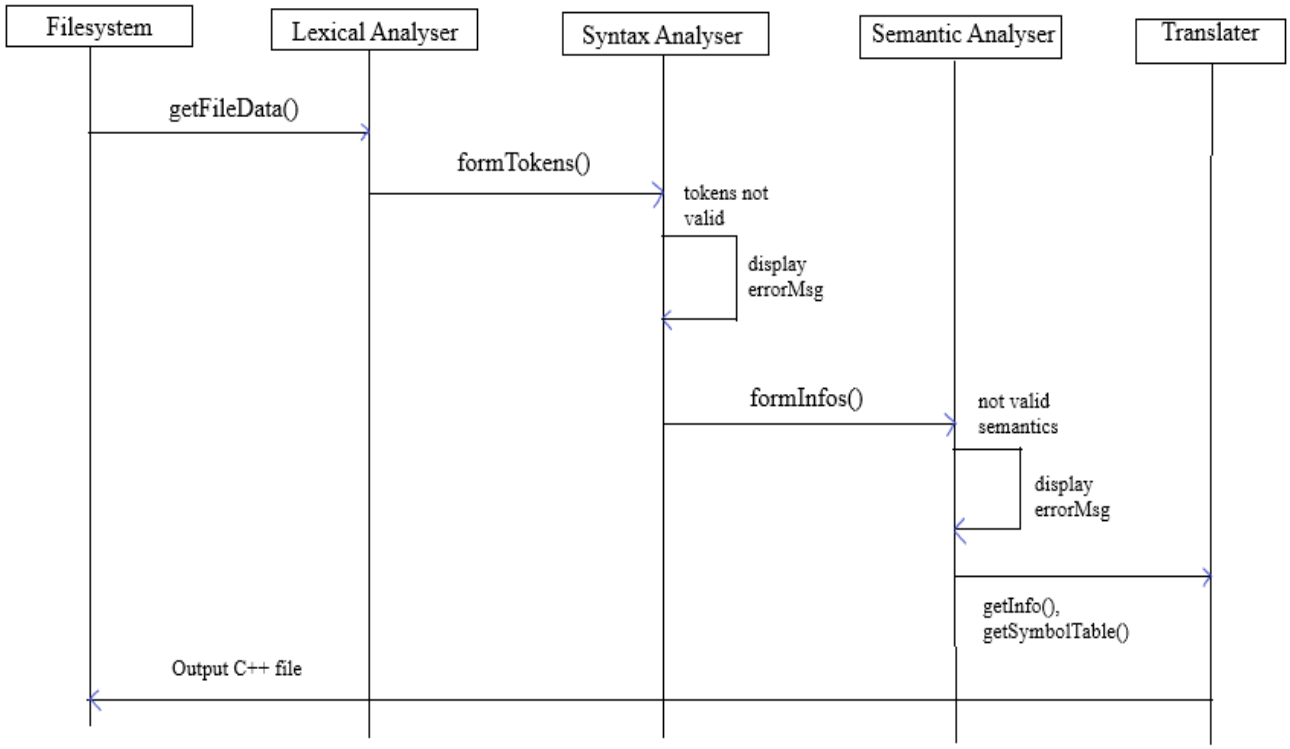
3.5 Use Case Diagram







3.6 Sequence Diagram



● CHAPTER 4

Implementation

A compiler implements a formal transformation from a high-level source program to a low-level target program. Compiler design can define an end to end solution or tackle a defined subset that interfaces with other compilation tools e.g. preprocessors, assemblers, linkers. Design requirements include rigorously defined interfaces both internally between compiler components and externally between supporting toolsets.

This project just implements the front-end of the compiler and emits C++ code at the end of it. The language used to implement the project is Java.

Algorithm:
Step 1) Read the input file.
Step 2) Initialise the keywords and operators.
Step 3) Give the file data to the LexicalAnalyser, which splits the data into a stream of tokens.
Step 4) Get the respective SequenceTypeInfo for all tokens.
Step 5) Syntactically analyse the SequenceTypes and tokens, if any syntax errors are present then push them into the ErrorLog and quit. Else return back a stream of data structures that contain detailed information about all the statements in the program.
Step 6) Pass these stream of data structures to the SemanticAnalyser, which checks if the statements when put together form a valid meaning. If so, then deduce and assign types to variables and expressions, and store that information in the SymbolTable. Use the symbol table for type checking and error checking the statments. If errors are found then push them into the ErrorLog and quit. Else goto the next phase.
Step 7) Use the SymbolTable and the stream of data structures which contain the statement information and translate every statement to the respective c++ . statement
Step 8) Run the g++ compiler on the output c++ file to get the executable file.

Syntax of the language:

- Defining functions

  func <function-name> (<argument-name> <argument-types> .....) -> <return-type> { .... BODY ... }

  Ex: func main(argc: int, argv: char**) -> int { return 0; }

- Defining structs

  <struct-name>::struct { .... variable-declaration .... }

  Ex: Person::struct { name := “none”; age := 0 }

- Defining variables

  - <variable-name>: <variable-type> = <expression>;
  - <variable-name> := <expression>;
  - <variable-name>: <variable-type>; // declaration

- Conditional statements

  if <expression> { .. body ... }
  while <expression> { .. body .. }

- Include other files

  use <”filename”>

- Using pointers
-
- >> to point to a the address of the variable
  << to deference and get the value at the address

- **CHAPTER 5**

• Output Snapshots

INPUT FILE

```
func main() -> int {
   a := add(1, 2);
   b := 10;

   c := add(>> b, >> a);
   ptr := >> c;

   printf("a: %d\n", a);
   printf("b: %d\n", b);
   printf("c: %d\n", c);
   printf("%d, %d\n", (add(3, 4) + a), (a + b - c + 1.0));

   d := div(add(3, 4) + a, a + b - c + 1.0);

   printf("%f\n", d);

   return 0;
}

func add(a: int*, b: int*) -> int {
   return (<< a) + (<< b);
}

func add(a: int, b: int) -> int {
   return a + b;
}

func div(a: int, b: double) -> double {
   return a / b;
}
```

C++ File

```
int main();
int add(int *a, int *b);
int add(int a, int b);
double div(int a, double b);

int main() {
   int a = add(1,2);
   int b = 10;
   int c = add(&b,&a);
   int *ptr = &c;
   printf("a: %d\n",a);
   printf("b: %d\n",b);
   printf("c: %d\n",c);
   printf("%d, %d\n",(add(3,4)+a),(a+b-c+1.0));
   double d = div(add(3,4)+a,a+b-c+1.0);
   printf("%f\n",d);
   return 0;
}

int add(int *a, int *b) {
   return (*a)+(*b);
}

int add(int a, int b) {
   return a+b;
}

double div(int a, double b) {
   return a/b;
}
```

Snapshot 2

INPUT FILE

```
func main() -> int {
   name: char[20];
   copy_str("Hello", name);
   append_to_str(", World.", name);

   printf("%s\n", name);

   str := "Hello, World Java\n";
   print_str(str);

   return 0;
}

func copy_str(from: string, to: char[]) -> void {
   i := 0;
   c := from[0];
   while c != '\0' {
      to[i] = c;
      i = i + 1;
      c = from[i];
   }

   return void;
}

func print_str(str: string) -> void {
   i := 0;
   c := str[i];

   while c != '\0' {
      printf("%c", c);
      i = i + 1;
      c = str[i];
   }

   return void;
```

C++ File

```
#include <iostream>
#include <string>
#include <stdio.h>
using namespace std;

int main();
void copy_str(string from, char to[]);
void print_str(string str);
void append_to_str(string str, char to[]);

int main() {
   char name[20] = {  };
   copy_str("Hello",name);
   append_to_str(", World.",name);
   printf("%s\n",name);
   string str = "Hello, World Java\n";
   print_str(str);
   return 0;
}

void copy_str(string from, char to[]) {
   int i = 0;
   char c = from[0];
   while (c!='\0') {
      to[i] = { c };
      i = { i+1 };
      c = { from[i] };
   }
   return ;
}

void print_str(string str) {
   int i = 0;
   char c = str[i];
   while (c!='\0') {
      printf("%c",c);
      i = { i+1 };
```

Snapshot 1

- **CHAPTER 6**

Conclusion and future scope

As software becomes larger, programming languages become higher-level, and processors continue to fail to be clocked faster, we'll increasingly require compilers to reduce code bloat, eliminate abstraction penalties, and exploit interesting instruction sets. At the same time, compiler execution time must not increase too much and also compilers should never produce the wrong output. This paper examines the problem of making optimizing compilers faster, less buggy, and more capable of generating high-quality output.

Pointer Analyses:

Why is Pointer Analysis Difficult ?
Pointer-alias analysis for C programs is particularly difficult, because C programs can perform arbitrary computations on pointers. In fact, one can read in an integer and assign it to a pointer, which would render this pointer a potential
alias of all other pointer variables in the program. Pointers in Java, known as references, are much simpler. No arithmetic is allowed, and pointers can only point to the beginning of an object.
Pointer-alias analysis must be interprocedural. Without interprocedural analysis, one must assume that any method called can change the contents of all accessible pointer variables, thus rendering any intraprocedural pointer-alias analysis ineffective.
Languages allowing indirect function calls present an additional challenge for pointer-alias analysis. In C, one can call a function indirectly by calling a dereferenced function pointer. We need to know what the function pointer can point to before we can analyze the function called. And clearly, after analyzing the function called, one may discover more functions that the function pointer can point to, and therefore the process needs to be iterated.
While most functions are called directly in C, virtual methods in Java cause many invocations to be indirect. Given an invocation x . m() in a Java program, there may be many classes to which object x might belong and that have a method named m. The more precise our knowledge of the actual type of x, the more precise our call graph is. Ideally, we can determine at compile time the exact class of x and thus know exactly which method m refers to.
Consider the following sequence of Java statements:

```
Object o;
o = new String();
n = o.length();
```

analysis to precisely the method declared for String.
It is possible to apply approximations to reduce the number of targets. For Here o is declared to be an Object. Without analyzing what o refers to, a possible methods called "length" declared for all classes must be considered as possible targets. Knowing that o points to a String will narrow interprocedural example, statically we can determine what are all the types of objects created, and we can limit the analysis to those. But we can be more accurate if we can discover the call graph on the fly, based on the points-to analysis obtained a t the same time. More accurate call graphs lead not only to more precise results but also can reduce greatly the analysis time otherwise needed.
Points-to analysis is complicated. It is not one of those "easy" data flow problems where we only need to simulate the effect of going around a loop of statements once. Rather, as we discover new targets for a pointer, all statements
assigning the contents of that pointer t o another pointer need t o be re-analyzed.

For simplicity, we shall focus mainly on Java. We shall start with flow-insensitive and context-insensitive analysis, assuming for now that no methods are called in the program. Then, we describe how we can discover the call graph on the fly as the points-to results are computed. Finally, we describe one way of handling context sensitivity.

A Model for Pointers and References
Let us suppose that our language has the following ways to represent and manipulate references:
1. Certain program variables are of type "pointer to T" or "reference to T," where T is a type. These variables are either static or live on the run-time stack. We call them simply variables.
2. There is a heap of objects. All variables point to heap objects, not to other variables. These objects will be referred to as heap objects.
3. A heap object can have fields, and the value of a field can be a reference to a heap object (but not t o a variable).
Java is modeled well by this structure, alnd we shall use Java syntax in examples.
Note that C is modeled less well, since pointer variables can point to other pointer variables in C, and in principle, any C value can be coerced into a pointer.
Since we are performing an insensitive analysis, we only need to assert that a given variable v can point to a given heap object h; we do not have to address
the issue of where in the program v can point to h, or in what contexts v can point to h. Note, however, that variables can be named by their full name. In Java, this name can incorporate the module, class, method, and block within a method, as well as the variable name itself. Thus, we can distinguish many variables that have the same identifier.

Heap objects do not have names. Approximation often is used to name the objects, because an unbounded number of objects may be created dynamically. One convention is to refer to objects by the statement a t which they are created.
As a statement can be executed many times and create a new object each time,
an assertion like "v can point to h" really means "v can point t o one or more of the objects created at the statement labeled h."
The goal of the analysis is to determine what each variable and each field of each heap object can point to. We refer to this as a points-to analysis; two pointers are aliased if their points-to sets intersect. We describe here an inclusion-based analysis; that is, a statement such as v = w causes variable v t o point to all the objects w points to, but not vice versa. While this approach may seem obvious, there are other alternatives t o how we define points-to analysis.For example, we can define an equivalence-based analysis such that a statement like v = w would turn variables v and w into one equivalence class, pointing.

Flow Insensitivity
We start by showing a very simple example to illustrate the effect of ignoring control flow in points-to analysis.
1) h:
2) i :
3) j :
4
5 1
6)
a=newObject();
b = new O b j e c t ( ) ;
c = new O b j e c t ( ) ;
a = b;
b = c;
c = a;
If you follow the statements (4) through (6), you discover that after line (4) a points only to i. After line (5), b points only to j , and after line (6), c points only to i. The above analysis is flow sensitive because we follow the control flow and compute what each variable can point to after each statement. In other words, in addition to considering what points-to information each statement "generates ," we also account for what points-to information each statement "kills." For instance, the statement b = c ; kills the previous fact "b points to j" and generates the new relationship "b points to what c points to.?'
A flow-insensitive analysis ignores the control flow, which essentially assumes that every statement in the program can be executed in any order. It computes only one global points-to map indicating what each variable can possibly point to at any point of the program execution. If a variable can point to two different objects after two different statements in a program, we simply record that it can point to both objects. In other words, in flow-insensitive analysis, an assignment does not "kill7' any points-to relations but can only "generate7' more points-to relations. To compute the flow-insensitive results, we repeatedly

add the points- to effects of each statement on the points-to relationships until no new relations are found.
There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use,* with general-purpose programming languages distinguished from domain-specific programming languages. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming.More refined paradigms include procedural programming,object-oriented programming,functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. Anassembly languageis not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose,system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these).Some general purpose languages were designed largely with educational goals.

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being deliberately esotericor not.

Measuring programming language popularity

- Counting the number of times the language name is mentioned in web searches, such as is done by Google Trends
- Counting the number of job advertisements that mention the language
- The number of books sold that teach or describe the language
- Estimates of the number of existing lines of code written in the language– which may underestimate languages not often found in public searches
- Counts of language references (i.e., to the name of the language) found using a web search engine
- Counting the number of projects in that language on SourceForge and GitHub
- Counting the number of postings in Usenet newsgroups about the language
- Comparing the number of commits or changed source lines for open source projects on Open Hub
- The number of courses sold by programming bootcamps
- The number of students enrolled in programming classes around the world
- The number of videos on each language on YouTube
- The number of postings on Reddit or Stack Exchange about a language