PUNJAB ENGINEERING COLLEGE

(Deemed To Be University)

MINOR PROJECT

CSN 321

De-Cipher

Project Mentor:
Dr. Sanjeev Sofat

Group Id: 16

Submitted By:
Srishti Arora (18103011)
Jasmeen Bansal (18103016)
Bhavya Gulati (18103024)
Saksham Basandrai (18103048)

Background

The main purpose of a compiler or an interpreter is to translate a source program written in a high-level source language to machine language. The language used to write the compiler or interpreter is called implementation language. The difference between a compiler and an interpreter is that a compiler generates object code written in the machine language and the interpreter executes the instructions.

A utility program called a linker combines the contents of one or more object files along with any needed runtime library routines into a single object program that the computer can load and execute. An interpreter does not generate an object program. When you feed a source program into an interpreter, it takes over to check and execute the program. Since the interpreter is in control when it is executing the source program, when it encounters an error it can stop and display a message containing the line number of the offending statement and the name of the variable. It can even prompt the user for some corrective action before resuming execution of the program.

Motivation

"I hear and I forget. I see and I remember. I do and I understand."
-Confucius

These famous lines said by Confucius, holds true in all scenarios. For the best way to fully test the understanding of a concept is by building or creating it.

If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then we can build an interpreter that solves the problem by interpreting these sentences.

So, following the same philosophy, we as Computer Science students, use programming languages every day, and thus, we attempt to build our very own **Interpreter** from scratch, so as to gain a better insight on how it works and functions.

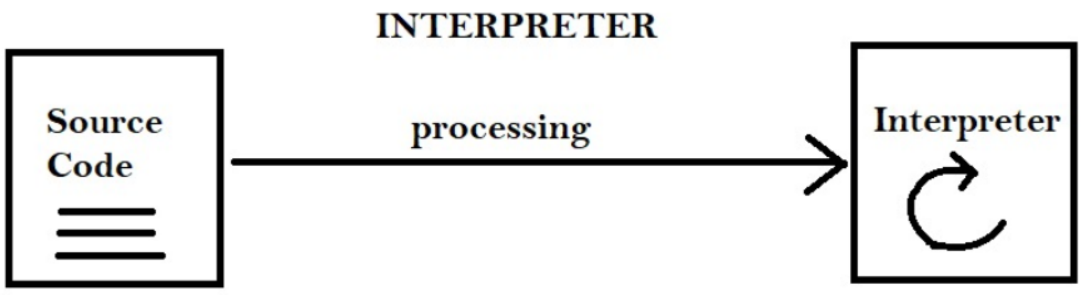
To write an interpreter we need to have a lot of technical skills that needs to be used together. Writing an interpreter will help us improve those skills and become better software developers. Also, the skills we will learn will be useful in writing any software, not just interpreters.

Thus, we thought that knowing how computers actually understand these languages would be a great insight in the core of Computer Science.

Description

We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans. This is called **source code**. However, a computer does not understand high-level language. It only understands the program written in 0's and 1's in binary, called the **machine code**.

To convert source code into machine code, we use either a compiler or an interpreter. To avoid the hassle of converting to machine code, an interpreter directly executes the source program without translating it into machine language first.



When we feed a source program into an interpreter, it takes over to check and execute the program. Since the interpreter is in control when it is executing the source program, when it encounters an error it can stop and display a message containing the line number of the offending statement and the name of the variable. It can even prompt the user for some corrective action before resuming execution of the program.

The process of building an interpreter is divided into 6 functional increments. Before moving to the next increment, the current increment has to be tested and validated. The increments are:

1. The framework,
2. The scanner
3. The symbol table
4. Parsing and interpreting expressions and assignment statements,
5. Parsing and interpreting control statements,
6. Parsing and interpreting declarations.

We are going to create a simple interpreter for a large subset of a **Procedural Language having similar construct as Pascal Language** and a source-level debugger which will be **implemented in C++.**

As far as the applications of this project are concerned, we want to create our very own programming language or domain specific language. If we create one, we will also need to create an interpreter for it.

Recently, there has been a resurgence of interest in new programming languages and we can see a new programming language pop up almost every day: Elixir, Go, Rust just to name a few. Thus, the skills gained in this process will eventually lead us towards our goal of building our very own programming language.

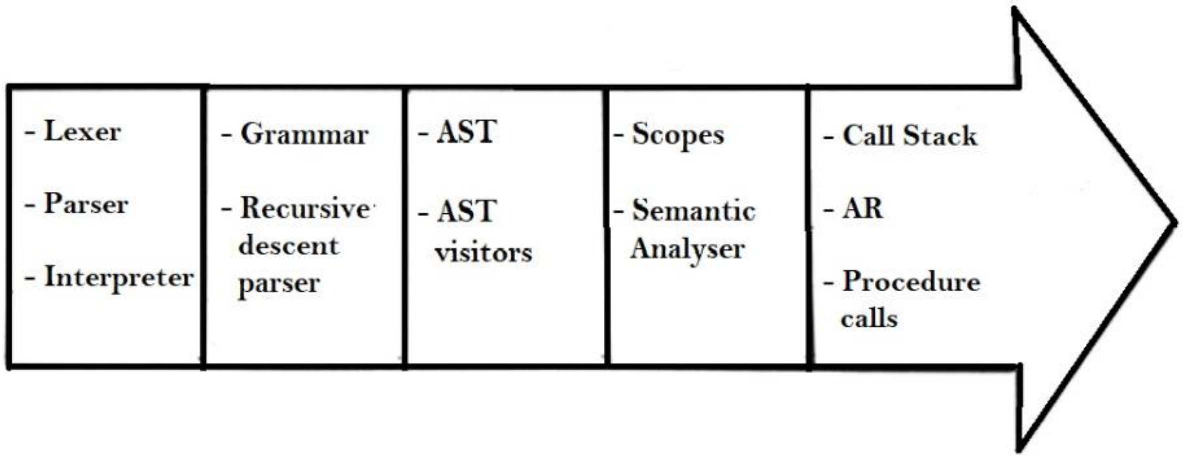
Interpreters are complex programs, and writing them successfully is hard work. To tackle the complexity, a strong software engineering approach can be used. **Design patterns**, **Unified Modelling Languages (UML) diagrams**, and other modern object-oriented design practices make the code understandable and manageable.

Features

We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans.

During the course of this project, we wish to implement the following **features:**

- Evaluating complex arithmetic expressions
- Checking associativity and precedence
- Assignment and Unary operator's evaluation
- Evaluating compound statements
- Semantic Analysis
- Procedure and function calls
- Solving nested procedures and functions



Outcome

The planned outcome is a fully functional procedural language interpreter performing complex arithmetic operations, procedural calls, nested calls, and semantic analysis.

Conclusion

We plan to build an interpreter for a procedural language in the context of a software engineering project. We feed the source program into the interpreter which does the processing and executes the source program, thus giving us the desired output.

So, we believe that building an interpreter from scratch would be a good idea, as building something from the very basic brings out the true beauty as well as the power of a piece of software.