Design and Implementation of Python-JavaScript mini compiler

The project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfilment of the requirements to award the degree of

**Bachelor of Technology/Master of Technology**

In

**Computer Science and Engineering**

**School of Engineering and Sciences**

Submitted by

B.V.S LAXMAN -AP21110010014

B.V.N. Hareesh -AP21110010839

A.SAI KEERTHI -AP21110010024

P. DHEERAJ -AP21110010022

S. REETHU BHARGAVI -AP21110010045

P. HARSHA -AP21110010050

K. LAHARI -AP21110011172

S. udaya sruthi -AP21110011171



Under the guidance of

Dr. Shuvendu Rana

**CERTIFICATE**

Date: 24/11/2023

This is to certify that the work present in this project entitled “Design and Implement a Mini Compiler for Python and JavaScript Programming Language” has been carried out under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University AP for the award of Bachelor of Technology/Master of Technology in the **School of Engineering and Sciences.**

**Supervisor**

Dr. Shuvendu Rana

**ABSTRACT:**

This project introduces a Lexical Analyzer designed to parse Python and JavaScript source code, identifying and classifying tokens within a unified framework. The analyzer, implemented using Flex, is geared towards facilitating the development of mini-compilers or interpreters that seamlessly handle both languages within a single codebase.

The lexical rules outlined in the code capture a range of language constructs, including keywords, identifiers, numeric values, operators, brackets, and colons. Keywords specific to Python and JavaScript are recognized, and each token's type and value are printed as part of the analysis.

The abstracted lexical analysis lays the groundwork for subsequent stages in the compiler or interpreter design process. As the code traverses the input source code, it lays the foundation for syntactic and semantic analysis, paving the way for a comprehensive language processing system. The open-ended nature of this implementation encourages further refinement and adaptation to accommodate evolving language specifications and project-specific needs.

**Introduction:**

As the landscape of modern software development evolves, the demand for flexible tools that seamlessly accommodate the integration of multiple programming languages within a single codebase has become increasingly apparent. This project addresses this demand by introducing a Lexical Analyzer capable of parsing both Python and JavaScript source code.

* **Python and JavaScript Integration Challenges:**

Python and JavaScript are pivotal languages in web development. Syntactic and lexical differences between Python and JavaScript create challenges for unified development.

* **Significance of Lexical Analysis:**

Lexical analysis is a critical phase in language processing. Identifies and classifies essential language constructs, such as keywords, identifiers, and operators.

* **Flex-Based Implementation:**

The provided implementation is based on Flex, a lexical analyzer generator. Offers a foundation for subsequent stages of language processing.

* **Open and Extensible Codebase:**

The provided code is open-ended and extensible. Encourages innovation in language processing tools to address diverse contemporary development needs.

* **Towards Comprehensive Language Processing:**

By delving into lexical analysis intricacies, the project paves the way for more comprehensive language processing systems. Addresses the evolving landscape of software development and the need for adaptable language tools.

**Tools Used:**

Lexical Analysis:

* Flex (lex) for generating lexical analysers.

**Lex Code:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

%}

%%

and|as|break|class|continue|def|del|elif|else|except|False|finally|for|from|global|if|import|in|is|lambda|None|nonlocal|not|or|pass|raise|return|True|try|while|with|yield|int|input|flat|complex|str|list|tuple|range|bool { printf("keyword: %s\n ", yytext); }

break|case|catch|class|const|continue|debugger|default|delete|do|else|export|extends|finally|for|function|if|import|in|instanceof|let|new|return|super|switch|this|throw|try|typeof|var|void|while|with|yield { printf("keyword: %s\n ",yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("identifier: %s\n", yytext);

}

[0-9]+ {

printf("val ");

}

[+\-\*/%=.>]|new {

printf("operator: %s\n",yytext);

}

[(){}] {

printf("bracket: %s\n",yytext);

}

[:] {

printf(": %s\n",yytext);

}

[\t \n] ;

. {

// Ignore other characters

}

%%

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <input\_file.py/js>\n", argv[0]);

return 1;

}

FILE \*file = fopen(argv[1], "r");

if (!file) {

perror("Error opening file");

return 1;

}

yyin = file;

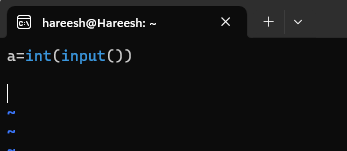
yylex();

fclose(file);

return 0;

}

**INPUT:**

****

**OUTPUT:**

A screenshot of a computer program

Description automatically generated

**Lex code: -**

%{

#include "y.tab.h"

%}

%%

[a-z] return VAR;

[0-9]+ return NUMBER;

[\t] ;

\n return 0;

. return yytext[0];

%%

**Yacc code:-**

%{

#include <stdio.h>

int yylex();

void yyerror(char \*s);

%}

%token NUMBER VAR

%%

S: E

;

E: E '+' T

| T

;

T: T'\*'F

| F

;

F: '('E')'

| NUMBER

| VAR

;

%%

int main(){

yyparse();

}

int yywrap(){

return 1;

}

void yyerror(char \*s){

printf("Error %s",s);

}

**OUTPUT: -**

**A screen shot of a computer

Description automatically generated**

**Conclusion:**

The Compiler Design Project has completed Phase 3, achieving milestones in language design, and grammar creation. The language now supports Javascript, python, and a rich feature set. The team has acquired proficiency in utilizing lexical and syntax analysis tools such as Flex.