CS 4098 Project
Report

An Experimental Compiler Design Platform

Interim project report

Submitted by

Roll No	Names of Students
B100168CS	Arun Rajan
B100780CS	Nachiappan V.

Under the guidance of
Dr. Murali Krishnan K

Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India – 673 601

Monsoon Semester 2013

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

Certificate

This is to certify that this is a bonafide record of the project presented by the students whose names are given below during Monsoon 2013 in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Roll No	Names of Students
B100168CS	Arun Rajan
B100780CS	Nachiappan V.

Dr. Murali Krishnan K
(Project Guide)

Jayaraj P B
(Course Coordinator)

Date:

Abstract

Compiler design is a very complex task involving several layers of development. Unless broken down into individual components and implemented stage by stage, it can be a hard nut to crack. Even though this has already been done in theory as well as practice, they are very rarely correlated by students who are in the process of learning to build a compiler. Many students often struggle in the process or fail to comprehend the bigger picture. Often this is due to lack of guidance or lack of proper learning material. To fix this situation, this project is aimed at developing a self-sufficient experimental compiler design platform where students can build their own compiler independent of supervision. In this project, we have used a *back to basics* approach where theory meets practice.

Contents

Chapter 1

Problem Definition

In spite of the availability of plenty of educational resources worldwide to tutor students in the process of building a compiler, most of them lack a systematic approach.

Though many attempts have been made to simplify this process down, these merely serve as a manual to compiler-generation tools such as LEX and YACC. Very few link the usage of the tools to the compiler design process. Also, most of these abstain from exploring the back-end working of these tools. The availability of such resources are limited because they are often localized within the institution or organization of origin.

Chapter 2

Introduction

Building a compiler from scratch can be an intricate and time consuming task. Compiler generator tools such as LEX and YACC have been used for building compilers for over more than three decades now. These utilities have greatly simplified the process since their introduction in 1975 by Lesk and Johnson.

This project aims to develop an online self-sufficient educational platform which can be used to tutor students in writing a compiler. Being instructional in nature, this project gives the learner an insight into the working of LEX, YACC and the usage of these tools to develop a compiler for SIL (Simple Integer language).

Chapter 3

Work Done

3.1 Theoretical Research

3.1.1 Lexical Analysis using LEX

Lexical analysis is the process of breaking up a source program into tokens. A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language [?]. A lexical analyzer scans a given input and produces an output of tokens.

LEX is a tool that translates a set of regular expression specifications into a C implementation of a corresponding finite state machine. This C program when compiled, yields an executable lexical analyzer. Conceptually, LEX constructs a finite state machine to recognize all the regular expression patterns specified in the LEX program file. The lex.yy.c program stores information about the finite state machine in the form of a decision table (transition table). LEX makes it's decision table visible if we compile the LEX program with the `-T` flag. The finite state machine used by LEX is a deterministic finite state automaton (DFA). The lex.yy.c file simulates the DFA.

Also, LEX offers features to execute a single or compound C statement when a pattern match is found in the input stream. Given its ability to scan and identify a given pattern, and the ability to execute a corresponding action, LEX can be used to generate a lexical analyzer.

3.1.2 Pattern matching by LEX

The pattern to be matched in the input stream is specified using regular expression in the LEX program. LEX converts these regular expressions into an expression syntax tree (EST).

This EST is then used by LEX to construct a set of states and the transition table (a two dimensional data structure) of the DFA.

The working of the constructed DFA is simulated using the following algorithm. The information about all the transitions made by the DFA can be obtained from the decision table (generally a two dimensional matrix) through the transition() function.

The above theory has been summarized in the following algorithm:

Algorithm 1 DFA Simulation

```
  $currentstate \leftarrow startstate$
  $c \leftarrow getnextchar()$
  while $c \neq EOF$ do
    $currentstate \leftarrow transition(currentstate, c)$
    $c \leftarrow getnextchar()$
    if $currentstate \ \epsilon \ finalstates[\,]$ then
      accept $currentstate$
    else
      reject $currentstate$
    end if
  end while
```

3.1.3 Syntax analysis using YACC

Syntax analysis follows lexical analysis in the compilation process. The syntax of a programming language can be expressed using Context Free Grammars (CFG). Any sentential form of the programming language's grammar is considered a syntactically correct program. The process of checking whether a program can be derived from the programming language's grammar is referred to as *parsing*. YACC (Yet Another Compiler Compiler) was developed in 1970 by Stephen C. Johnson at AT& T Corporation. YACC is tool that translates the given CFG specification in a YACC program to a corresponding Push Down Automaton (PDA) implementation in C language. The generated C program when compiled, yields an executable parser. The source program is fed to the parser to check if it is syntactically correct. Also, YACC offers features to execute a single or compound C statement when the input or a part of the input in the input stream matches the body of a production of the CFG in the YACC program.

3.1.4 Parsing algorithm generated by YACC

On analyzing the y.tab.c file generated by YACC, it was found that it is a simulation of a PDA. The parsing algorithm generated by YACC is a shift-reduce parsing algorithm. In the y.tab.c file, it can be found that the generated shift reduce parser uses LALR parsing algorithm to take decisions regarding shifting and reducing input symbols/grammar symbols to/from the stack[**?**].

Chapter 4

Design

This chapter contains the proposed design which has been followed up till the current state of the project and will be followed (or improvised upon) till the completion of the project.

4.1 Documentation

This project will consists of vast documentation on the usage of LEX and YACC to generate a compiler, and how these tools can be exploited to build a compiler on a systematic basis. Initially, the documentation phase will concentrate more on the mastery of the tools and gradually introduce compiler design concepts and how these can be implemented using these tools.

To differentiate itself from the other existing tutorials, the documentation will follow a very simple explanation approach using plenty of examples and input/output samples. The documentation will be self-sufficient in nature. It will be developed with an assumption that the learner would have basic working proficiency with the C language.

The documentation will focus on the in-depth explanation of the theory used in the back-end working of these tools and eventually link this to compiler design strategies to enhance a better comprehending of the concepts.

4.2 Testing

The document is to be embedded with plenty of code in examples and exercises. Before being used in the documentation, each and every code snippet is to be implemented and tested with the complete program's body.

4.3 Roadmap

The Road map is the key to the achieving the project's objective. Along with the documentation and given code snippets, a learner would be asked to follow the roadmap which is yet to be designed.

4.4 Interpreter

An interpreter for SIL is to be designed and implemented using LEX and YACC. This interpreter would serve as a debugging tool. Students can use the interpreter to verify the output produced by the compiler they build.

4.5 Version Control

Since this project contains various components and evolves through many stages, it needs to be maintained using a version control system. The advantage of using one would be the ease being roll back to any version at any point of time during the development phase of the project. We have chosen Git for this purpose.

4.6 Online platform

To enhance the availability of the project to students, this project will be hosted online at the domain **https://silcnitc.github.io**. The website is being developed with HTML5, CSS3 and JavaScript. Github is a remote server for Git. Under licensed conditions, the project will be released on an open source basis on Github.

4.7 Assembling the framework

Students will use the roadmap to build a compiler for SIL. Towards the code generation phase, they would be instructed to generate code for the SIM architecture[**?**]. Once all the individual components have been completely developed, they will be tested and proof read several times before they will be integrated with each other accordingly on the website.

Chapter 5

Results

With the current work progress, there will be enough content to tutor students in building the initial stages of the compiler. The exact details of results are as follows:

5.1 LEX Documentation

A LEX document for the lexical analysis phase has been designed, compiled and reviewed.

5.2 YACC Documentation

A YACC document for the parsing phase has been designed and compiled. Yet to be reviewed.

5.3 Online Platform

The mainframe of the website has been completed. The individual components are currently under construction.

References

[1] Modern Compiler Implementation in C, by Andrew W. Appel `<urlhere>`

[2] Compilers: Principles,Techniques and Tools, by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman `<urlhere>`

[3] Simple Integer Machine Architecture, Dr. Murali Krishnan K. `http://athena.nitc.ac.in/~kmurali/Compiler/sim.html`