



Khulna University of Engineering & Technology

Report on: Compiler Design (PirateScript)

Course Name: Compiler Design Laboratory

Course Code: CSE3212

Submission Date: 21 November 2023

Submitted By	Submitted To
Sk. Azraf Sami Roll: 1907115 Year: 3rd Term: 2nd Dept. of CSE, KUET, Khulna	Nazia Jahan Khan Chowdhury Assistant Professor Dept. of CSE, KUET, Khulna Dipannita Biswas Lecturer Dept. of CSE, KUET, Khulna



Introduction:

Lexical analysis, also known as scanning or tokenization, is the first phase in the compilation process. Its primary purpose is to analyze the source code and break it down into a sequence of tokens. Tokens are the smallest units of meaning in a programming language, such as keywords, identifiers, literals, and operators. The tool used to perform lexical analysis is often referred to as a lexer or lexical analyzer.

The lexer reads the input source code character by character and groups characters into tokens based on predefined rules. These tokens are then passed to the next phase of the compiler for further processing.

Importance of Lexer in Compiler Design:

Tokenization

Simplifies Parsing

Error Detection and Reporting

Modularity and Separation of Concerns

Efficient Compilation Process

Language Flexibility

Debugging Support

Optimization Opportunities

Integration with Compiler Tools

Bison, often referred to as a parser generator, is a tool used in the field of compiler design to generate parsers for programming languages. It is a part of the GNU Compiler Collection (GCC) and is typically used in conjunction with Lex or Flex (lexical analyzer generator) for the complete parsing process. Bison generates parsers based on a formal grammar specification of the programming language, typically in Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).

Importance of Bison in Compiler Design:

Parsing and Syntax Analysis

Grammar Specification

Automatic Code Generation

Abstract Syntax Tree (AST) Generation

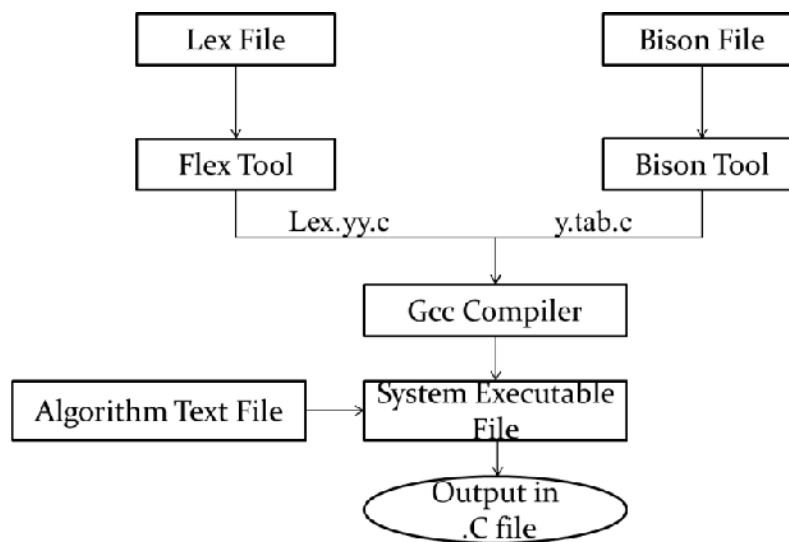
Error Handling

Integration with Lexical Analysis

Modularity and Separation of Concerns

Support for Ambiguous Grammars

Integration with Compiler Tools



Using Command:

bison -d PirateScript.y

flex PirateScript.l

gcc -o PirateScript.out PirateScript.tab.c lex.yy.c -lm # to link math library use lm flag
./PirateScript.out



Documentation:

Introduction to PirateScript: A Programming Language with a Swashbuckling Twist

Ahoy, ye scallywags and code enthusiasts! Welcome to the high seas of programming with PirateScript, a language as adventurous and cunning as a pirate on the hunt for buried treasure. PirateScript be more than just a set of code rules; it's a linguistic journey through the seven seas of syntax and semantics, where every line of code tells a tale of daring escapades and nautical exploits.

Setting Sail with PirateScript: The Basics

PirateScript be designed for both seasoned developers and landlubbers lookin' to embark on a voyage into the world of programming. With its unique blend of traditional programming constructs and the whimsy of the pirate's life, it be a language that be as fun to write as it be to read.

Data Types:

- JackSparrow_i (integer)

The variable JackSparrow_i is an integer used to represent a numeric value associated with the legendary pirate Jack Sparrow.

This variable is intended to store whole number values, such as counts, indices, or any other integer-related information in the context of the program.

- Fury_f (float)

The variable Fury_f is a float used to represent a numeric value associated with the concept of fury or intensity.

This variable is suitable for storing decimal or floating-point values, providing precision for computations involving non-integer numbers.

- Charlotte_c (char)

The variable Charlotte_c is a character variable designed to store a single character, specifically associated with the character "Charlotte."

This variable is intended for holding single character values, such as letters, digits, or special characters.

- DavyJones_d (double)

The variable DavyJones_d is a double used to represent a numeric value associated with the legendary character Davy Jones.

This variable is suitable for storing double-precision floating-point values, providing higher precision compared to the float data type.

Variable Declaration:

pirate_<variable_name>

Example: JackSparrow_i pirate_a

Here " JackSparrow_i" is the data type, by starting "pirate_" that means it is a variable. Here "a" is the variable name.

Variables initialization:

pirate_<variable_name> loot value

Here, "**loot**" is the assign keyword

Example: pirate_a loot 8

pirate_a is the variable which store value 8


Comment

Single line comment start with this emoji 📌

Example: 📌 Black Perl

For multiple line comment:  <comment> 

Example:

 Jack Sparrow

Pirate Captain



Operators

- Arithmetic Operators


"booty" ==> for Plus operation

"mutiny" ==> for Minus operation


"gun" ==> for Multiplication

"gold" ==> for Division

- Unary Operators

"++" ==> Increment Operator

"--" ==> Decrement Operator

"! " ==> Not Operator

- Logical Operators

logical_operator "&&", "||", "!!"

- Relational Operators

"tiny" ==> Use as less than operator

"mighty" ==> Use as greater than operator

Conditional Statement

"Parley" ==> as If

"Aye" ==> as else

"ElseWise" ==> as else if

"Pirate_Code" ==> as CASE

"Code_Book" ==> as SWITCH

Loop:

"Shanty" ==> as For

"Whale" ==> as WHILE

Function:

"Captain" ==> this keyword represents the main function

"crew " ==> this keyword represents user defined function

"gunPowder" ==> work as Pow function of C language

"fact" ==> use to find the factorial of a number

Additional Syntax:

"LeftCannon " ==> use as left first bracket

"RightCannon " ==> use as right first bracket

"yo_ho" ==> use as comma

" " ==> indicates statement

"LeftBow" ==> use as left second bracket

"RightBow" ==> use as right second bracket

Github Repo Link: [Here](#)

