SUMMER TRAINING REPORT

ON

GAME LANGUAGE COMPILER

Submitted in the partial fulfillment of the requirements for the award of degree

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE ENGINEERING

Submitted By: Student

Name: VARUN PAWAR

Roll No.: 10614802718

Semester:7th

Batch :7(C-6)

Maharaja Agrasen Institute of technology, PSP area,
Sector – 22, Rohini, New Delhi – 110085

Guru Gobind Singh Indraprastha University

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this report entitled "Game Language Compiler", in partial fulfilment of the requirement for the award of the degree Bachelor of Technology and submitted in Computer Science & Engineering Department of Maharaja Agrasen Institute of Technology,(affiliated to Guru Gobind Singh Indraprastha University, New Delhi) is an authentic record of my own work.

The work reported in this has not been submitted by me for award of any other degree or diploma of this or any other institute.

VARUN PAWAR

10614802718

CERTIFICATE

PREFACE

This Project Report has been prepared in partial fulfilment of the requirement for the Subject GAME LANGUAGE COMPILER using BISON, FLEX, C++, ARCH LINUX,VAGRANT, VMVARE for the program Bachelor of Technology of branch Computer Science in the academic year 2018-2022.

For preparing the Project Report, we have visited the company named ELITE TECHNO GROUPS(ETG) for the summer training, to avail the internship. The blend of learning and knowledge acquired during our practical studies at the company is presented in this Project Report.

This project report starts with a basic understanding of compiler design followed by various concepts used in compiler design which are used to create compiler and making a game programming language.

VARUN PAWAR

10614802718

COMPANY PROFILE

ELITE TECHNO GROUPS(ETG)

We train the engineers for the industry level knowledge. By giving the live industry level projects to them. The projects given by us is not only enhance the level of skills in child but prepare them for the next industry directly.

We are working with industry specialist who has industry experience of more than 5 years or so. So that our child / engineer can fulfil the industry demand without any training after completion of the college and the degree.

This company is founded by MR MAYANK ARORA who formly is an engineer and had an experience of 8 years in working in different sectors in the industry.

TABLE OF CONTENTS

CHAPTER - 1

INTRODUCTIONS

1.1 Introduction

The industrial training is compulsory for every student of Bachelor of Technology in Information Technology as a condition for the award of the degree. This exercise is also intended to provide exposure and experience to the students about the real situation in the field of Information Technology and as an early preparation for students before entering the working world.

I found a place to undergo industrial training at HCL TECHNOLOGIES for the project of Game Language Compiler . During this training, I was given the opportunity to follow the learning involved technical assistance related. Computer software that we get helps us understand the needs of business and academic ventures and have a clearer understanding and widespread in the intricacies of information technology (programming).

1.2 The objective of industrial training

Industry training is a major component of the extra-curricular learning in engineering. The students are required to pass before the Industrial Training is recommended for award of a degree at any engineering college. The students will be placed in government departments or private companies for 6-8 weeks to expose them to a real work environment and different from the atmosphere at the college.

Often students will be faced with many challenges and problems that have not been confronted. The main purpose of this training was to expose students to the real working environment when the students out of the college later.

Students also have the opportunity to use all the knowledge and theories related to courses taken while at the engineering course of industrial training and provide opportunities for students to use the fruit of a creative mind for the good of the firm and indirectly for their own benefit as well. Thus, there has been this experience, maybe can help students to study in the following semester and a real working environment. Students also can be exposed to ways to communicate well, expanding relationships between partners of the workplace and the people

around, foster teamwork and good relations with industrial workers where this at once can cause a sense responsible to a trust (work) and balance as well as from all aspects.

For example, one kind of algorithm is a classification algorithm. It can put data into different groups. The classification algorithm used to detect handwritten alphabets could also be used to classify emails into spam and not-spam.In addition, following the Industrial Training, students can improve their own weaknesses to improve and think more rationally in the handle which had been given by the employer. In the course of industrial training, students can assess their ability to work from employers. The students themselves can infuse the spirit of productivity to the challenges and obstacles that lie ahead.

Hopefully with the Industrial Training which is able to run to enhance the knowledge of students to enable them to contribute more effectively towards national development in the future and also to get a job that suits them according to the skills they have learned. In conclusion, the objective of the training period is to:

a) To expose students to the real working environment.

b) Let's students see the connection between theoretical learning with practical work.

c) Adopt and comply with safety regulations in the industry.

d) Establish and strengthen confidence in the performance of duties.

e) Instill teamwork and good relationships with other employees.

f) Ability to assess themselves to prepare for the working world after graduation.

g) Raise awareness and increase student interest in the subject selected.

h) Uplifting honest, trustworthy, dedicated and responsible for the tasks assign

1.3 The objective of industrial training report

Some objectives can be defined and made known in this industrial training report are described as follows:

i) Be evidence that the student has been training period by a predetermined time period.

ii) Record all activities during the training period.

iii) A reference in the future.

iv) It proves that students understand and appreciate the work done anything during the Industrial Training.

v) As a reference after the training period in the firm after completed their studies at the Polytechnic.

1.4 The importance of industry training

Industrial Training must be lived by all students in public higher education institutions or Private as a prerequisite for qualifying students receive a Degree in Information Technology is taken. Actually, it is not so qualified, but it is to create awareness about the situation the working environment. With the Industrial Training many useful and valuable experience gained as supply before set foot in the sphere of employment. It also can build confidence with the experience and knowledge available. So quite easy and convenient to carry out the work which will be given later.

Industry training is important because such training can expose students to the real working environment. It also can add and expand technical knowledge and skills of the student, if the student has previously acquired knowledge is limited, but when students attend this training, students can find out more about things, and when something will work. In addition, students can learn about the latest technology or skills in Training Industry.

In addition, this exercise also introduces the students themselves in terms of ability, willingness and attitude to the employer. This exercise can highlight the ability of students to work hard and to work with dedication and show a positive attitude to the employer.

This exercise is also important as it can get rid of inferiority complex while a student at the college, but when students are in training it is likely he will meet with officials of high rank or attend meetings and provide jobs to foreign workers.

Through this exercise, students are able to handle a problem with wise through experience that has been through this before. The value of respect for those around him will arise within the student if the student's Industrial Training heartfelt and sincere. It is hoped that these properties will be sustained in the future.

The conclusion that can be defined on the importance of industrial training are:

a) To build and strengthen the students to be more confident to face any task and tribulations faced in the workplace.

b) Planting teamwork and good relations between workers and employees of an organization.

c) To expose students to the real working environment.

d) To make students do not face any difficulty or clumsy when start working soon.

e) Adopt and comply with safety regulations in the industry.

f) Linking theory to practice and so on.

g) Build confidence.

h) Fostering honesty and responsibility in performing tasks.

i) Provide an official report on completion of training.

Chapter – 2

TECHNICAL CONTENTS

Compiler Design is the structure and set of principles that guide the translation, analysis, and optimization process of a compiler.

A Compiler is computer software that transforms program source code which is written in a high-level language into low-level machine code. It essentially translates the code written in one programming language to another language without changing the logic of the code.

The Compiler also makes the code output efficient and optimized for execution time and memory space. The compiling process has basic translation mechanisms and error detection, it can't compile code if there is an error. The compiler process runs through syntax, lexical, and semantic analysis in the front end and generates optimized code in the back end.

When executing, the compiler first analyzes all the language statements one after the other syntactically and then, if it's successful, builds the output code, making sure that statements that refer to other statements are referred to appropriately, Traditionally, the output code is called Object Code.

Types of Compiler

1. Cross Compiler: This enables the creation of code for a platform other than the one on which the compiler is running. For instance, it runs on a machine 'A' and produces code for another machine 'B'.
2. Source-to-source Compiler: This can be referred to as a transcompiler or transpiler and it is a compiler that translates source code written in one programming language into source code of another programming language.

3. Single Pass Compiler: This directly transforms source code into machine code. For instance, Pascal programming language.
4. Two-Pass Compiler: This goes through the code to be translated twice; on the first pass it checks the syntax of statements and constructs a table of symbols, while on the second pass it actually translates program statements into machine language.
5. Multi-Pass Compiler: This is a type of compiler that processes the source code or abstract syntax tree of a program multiple times before translating it to machine language.

Language Processing Systems Steps

1. Pre-Processor: This produces input for the compiler and also deals with file inclusion, augmentation, macro-processing, language extension, etc. It removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion.

2. Assembler: This translates assembly language code into machine understandable language. Each platform (OS + Hardware) has its own assembler. The output of an assembler is known as an object file which is a combination of machine instruction along with the data required to store these instructions in memory.

3. Interpreter: An interpreter converts high-level language into low-level machine language almost similar to what Compiler does. The major difference between both is that the interpreter reads and transforms code line by line while Compiler reads the entire code at once and outputs the machine code directly. Another difference is, Interpreted programs are usually slower with respect to compiled ones.

4. Relocatable Machine Code: This can be loaded at any point in time and can be run. This enables the movement of a program using its unique address identifier.

5. Linker: It links and merges a variety of object files into a single file to make it executable. The linker searches for defined modules in a program and finds out the memory location where all modules are stored.

6. Loader: It loads the output from the Linker in memory and executes it. It basically loads executable files into memory and runs them

Features Of A Compiler

Correctness: A major feature of a compiler is its correctness, and accuracy to compile the given code input into its exact logic in the output object code due to its being developed using rigorous testing techniques (often called compiler validation) on an existing compiler.

Recognize legal and illegal program constructs: Compilers are designed in such a way that they can identify which part of the program formed from one or more lexical tokens using the appropriate rules of the language is syntactically allowable and which is not.

Good Error reporting/handling: A compiler is designed to know how to parse the error encountered from lack be it a syntactical error, insufficient memory errors, or logic errors are meticulously handled and displayed to the user.

The Speed of the target code: Compilers make sure that the target code is fast because in huge size code its a serious limitation if the code is slow, some compilers do so by translating the bytecode into target code to run in the specific processor using classical compiling methods.

Preserve the correct meaning of the code: A compiler makes sure that the code logic is preserved to the tiniest detail because a single loss in the code logic can change the whole code logic and output the wrong result, so during the design process, the compiler goes through a whole lot of testing to make sure that no code logic is lost during the compiling process.

Code debugging help: Compilers make help the debugging process easier by pointing out the error line to the programmer and telling them the type of error that is encountered so they would know how to start fixing it.

3.1 IMPORTANCE OF MACHINE LEARNING

Improved performance: Using a compiler increases your program performance, by making the program optimized, portable, and easily run on the specific hardware.

Reduced system load: Compilers make your program run faster than interpreted programs because it compiles the program only once, hence reducing system load and response time when next you run the program.

Protection for source code and programs: Compilers protect your program source by discouraging other users from making unauthorized changes to your programs, you as the author can distribute your programs in object code.

Portability of compiled programs: Compiled programs are always portable meaning that you can transfer it from one machine to another without worrying about dependencies as it is all compiled together.

3.2 CHALLENGES IN DEVELOPING COMPILER

- Following are the CHALLENGES while designing COMPILERS...

Many variations:

- Many programming languages (eg, FORTRAN, C++, Java)
- Many programming paradigms (eg, object-oriented, functional, logic)
- Many computer architectures (eg, MIPS, SPARC, Intel, alpha)many operating systems (eg, Linux, Solaris, Windows)

Qualities of a compiler (in order of importance):

- The compiler itself must be bug-freeit must generate correct machine code
- the generated machine code must run fast
- the compiler itself must run fast (compilation time must be proportional to program size)the compiler must be portable (ie, modular, supporting separate compilation)it must print good diagnostics and error messages
- the generated code must work well with existing debuggersmust have consistent and predictable optimization.

Building a compiler requires knowledge of:

- Programming languages (parameter passing, variable scoping, memory allocation, etc)
- Theory (automata, context-free languages, etc)
- algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc)

- computer architecture (assembly programming)software engineering.

Tools used:

- BISON
- FLEX
- REGULAR GRAMMAR
- VAGRANT

BISON

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C or C++ programming in order to use Bison. Java is also supported as an experimental feature.

In order for Bison to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an ‘expression’. One rule for making an expression might be, “An expression can be made of a minus sign and another expression”. Another would be, “An expression can be an integer”.

As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is Backus-Naur Form or “BNF”, which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

There are various important subclasses of context-free grammars. Although it can handle almost all context-free grammars, Bison is optimized for what are called LR(1) grammars. In brief, in these grammars, it must be possible to tell how to parse any portion of an input string with just a single token of lookahead. For historical reasons, Bison by default is limited by the additional restrictions of LALR(1), which is hard to explain simply. See Mysterious Conflicts, for more information on this. You can escape these additional restrictions by requesting IELR(1) or canonical LR(1) parser tables. See LR Table Construction, to learn how.

Parsers for LR(1) grammars are deterministic, meaning roughly that the next grammar rule to apply at any point in the input is uniquely determined by the preceding input and a fixed, finite portion (called a lookahead) of the remaining input. A context-free grammar can be ambiguous, meaning that there are multiple ways to apply the grammar rules to get the same inputs. Even unambiguous grammars can be nondeterministic, meaning that no fixed lookahead always suffices to determine the next grammar rule to apply. With the proper declarations, Bison is also able to parse these more general context-free grammars, using a technique known as GLR parsing (for Generalized LR). Bison’s GLR parsers are able to handle any context-free grammar for which the number of possible parses of any given string is finite.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a symbol. Those which are built by grouping smaller constructs according to grammatical rules are called nonterminal symbols; those which can't be subdivided are called terminal symbols or token kinds. We call a piece of input corresponding to a single terminal symbol a token, and a piece corresponding to a single nonterminal symbol a grouping.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include 'identifier', 'number', 'string', plus one symbol for each keyword, operator or punctuation mark: 'if', 'return', 'const', 'static', 'int', 'char', 'plus-sign', 'open-brace', 'close-brace', 'comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols 'expression', 'statement', 'declaration' and 'function definition'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition; it contains one declaration, and one statement. In the statement, each 'x' is an expression and so is 'x * x'

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the return statement; this would be described with a grammar rule which reads informally as follows:

A 'statement' can be made of a 'return' keyword, an 'expression' and a 'semicolon'.

There would be many other rules for 'statement', one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the start symbol. In a compiler, this means a complete input program. In the C language, the nonterminal symbol ‘sequence of definitions and declarations’ plays this role

For example, ‘1 + 2’ is a valid C expression—a valid part of a C program—but it is not valid as an entire C program. In the context-free grammar of C, this follows from the fact that ‘expression’ is not the start symbol.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar’s start symbol. If we use a grammar for C, the entire input must be a ‘sequence of definitions and declarations’. If not, the parser reports a syntax error.

FLEX

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
Bison produces parser from the input file provided by the user. The function yylex() is automatically generated by the flex when it is provided with a .l file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

Note: The function yylex() is the main flex function that runs the Rule Section and extension (.l) is the extension used to save the programs.

Installing Flex on Ubuntu:

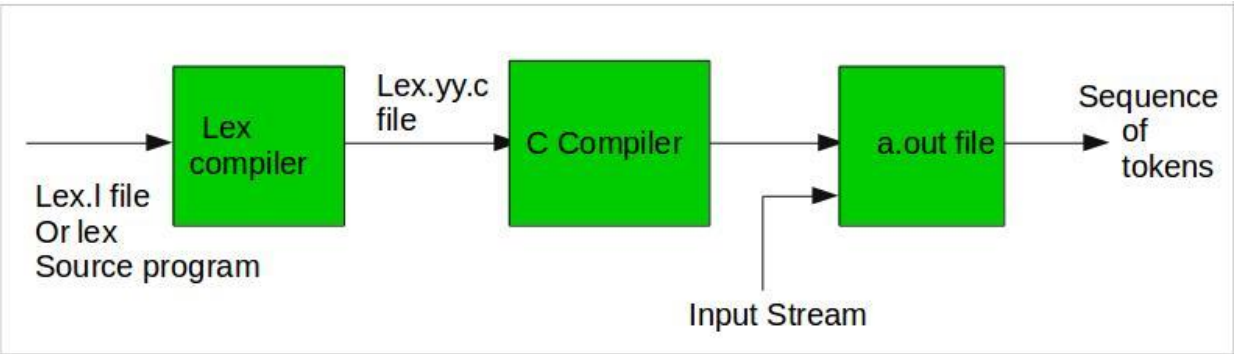
We provide nothing but the best curated videos and practice problems for our students. Check out the C Foundation Course and master the C language from basic to advanced level. Wait no more, start learning today!

sudo apt-get update

sudo apt-get install flex

Note: If Update command is not run on the machine for a while, it's better to run it first so that a newer version is installed as an older version might not work with the other packages installed or may not be present now.

Given image describes how the Flex is used:



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.

Step 2: The C complier compile lex.yy.c file into an executable file called a.out.

Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program Structure:

In the input file, there are 3 sections:

1. Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{ %}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c

Syntax:

```
%{
  // Definitions
%}
```

2. Rules Section: The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in "%% %%".

Syntax:

```
%%
pattern  action
%%
```

3. User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

```
%{
// Definitions
%}
```

%%

Rules

%%

User code section

How to run the program:

To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.

Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with

Step 2: gcc lex.yy.c

Step 3: ./a.out

Step 4: Provide the input to program in case it is required

Note: Press Ctrl+D or use some rule to stop taking inputs from the user. Please see the output images of below programs to clear if in doubt to run the programs.

C++ use

GNU flex and bison when C++ output is desired, which has been possible for some time. The flex utility is a *scanner generator* while bison is a *parser generator*; each utility outputs compilable source code from a user-defined input file containing configuration options, syntax specific to the utility, and C++ code fragments. Simply put, the (generated) parser program is meant to call the (generated) scanner repeatedly; the scanner translates its input into a sequence of *tokens*, each with an optional *semantic value*. The parser checks this token

stream for valid *grammatically correct* sequences, which causes it to perform *actions* using the semantic values. Invalid sequences trigger a “syntax error” within the parser.

Despite the fact that it is possible to create C++ scanner and parser classes with flex and bison, much online advice seems to suggest merely creating a C scanner and/or C parser, and then compiling this output with a C++ compiler in order to be able to use (unions with fields of) std::string* etc. for semantic types. However this first article describes the use of flex in its C generation mode with bison generating C++. (If you really want to know how to interface a C++ bison class with a C++ flex class straightaway, then skip to part 3, which also has links to downloads of the resources used. However the reason I cover this later is because it involves creation of a supplementary custom-written header file, and is a less “clean” method in my view.)

VAGRANT

Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases production parity, and makes the "works on my machine" excuse a relic of the past.

If you are already familiar with the basics of Vagrant, the documentation provides a better reference build for all available features and internals.

Why Vagrant?

Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize the productivity and flexibility of you and your team.

To achieve its magic, Vagrant stands on the shoulders of giants. Machines are provisioned on top of VirtualBox, VMware, AWS, or any other provider. Then, industry-standard provisioning tools such as shell scripts, Chef, or Puppet, can automatically install and configure software on the virtual machine.

For Developers

If you are a **developer**, Vagrant will isolate dependencies and their configuration within a single disposable, consistent environment, without sacrificing any of the tools you are used to working with (editors, browsers, debuggers, etc.). Once you or someone else creates a single Vagrantfile, you just need to vagrant up and everything is installed and configured for you to work. Other members of your team create their development environments from the same configuration, so whether you are working on Linux, Mac OS X, or Windows, all your team members are running code in the same environment, against the same dependencies, all configured the same way. Say goodbye to "works on my machine" bugs.

Chapter 3

WORKING LANGUAGE

In this we finally developed a game programming language that can be used to make the game and can be very use and easy to make the simple games.

In future we are going to develop the more advanced version of this language that can be used in the entire world just like python, c, c++.

This language is just the beginning of the work that can be done by our team in future.

CODE -> That run on our language.

```
//initialize the window variables
int window_width = 600;
int window_height = 300;
string window_title = "Block shooter gpl";

// initialize the variables to be used
int paddle_increment = 20;
int ball_x_increment = 4;
int ball_y_increment = 2;
int paddle_width = 80;
int paddle_height = 10;
int ball_size = 20;
int score = 0;

// TextBox to show current score
textbox score_text(text="Current Score: " + score ,x = window_width / 2 - 60 , y = 10 , red = 0 , blue = 0  , green = 0,visible = 1 );
```

```
	// TextBox to show game end message
	textbox game_over (text="  Oops !!!  ==>  Final Score ->  " + score + "  <==  Tap q to quit the game. " ,x = 0 , y = window_height/2 , red = 0 , blue = 0  , green = 0,visible = 0 );

	// Initialize Ball animation
	forward animation ball_animate(circle cur_ball);

	//Intialize the rectangle that act as paddle
	rectangle paddle(x = 0, y = 30 , w = paddle_width, h = paddle_height,visible = 1);

	//Initialize the block
	rectangle block(x = random(window_width - 20), y = random(window_height - 40), w = 20, h = 20,red = 0, blue = 0  , green = 0,visible = 1);

	//Initialize the block
	circle ball(x = window_width/2, y = window_height/2, radius = ball_size/2, animation_block = ball_animate);

	//Ball animation
	animation ball_animate(circle cur_ball)
	{
		// To check ball remains in the window
		if (cur_ball.x < 0 || cur_ball.x > window_width - ball_size)
			ball_x_increment = -ball_x_increment;
		if (cur_ball.y > window_height - ball_size)
			ball_y_increment = -ball_y_increment;
```

```
		//If ball touches the paddle then the the ball get deflected
		if (cur_ball touches paddle)
			ball_y_increment = -ball_y_increment;

		//If ball do not touch the paddle the game is over
		if (cur_ball.y < 0)
		{
			game_over.visible = 1;
	//This makes the game over text box visible when game is over
			game_over.text = "  Oops !!!  ==>  Final Score ->  " + score + "  <==  Tap q
to quit the game. ";
			paddle.visible = 0;
		//This makes the paddle invisible when game is over
			block.visible = 0;
		//This makes the block invisible when game is over
			score_text.visible = 0;
	//This makes the score invisible when game is over
		}

		//If ball touches the block then the score is increased
		if (cur_ball touches block)
		{
			score+=1;
		// This is to increment the current score
			score_text.text = "Current Score: " + score;
		// This is to show the current score
			block.x = random (window_width - block.w);
		// This is to get the new block x-position
```

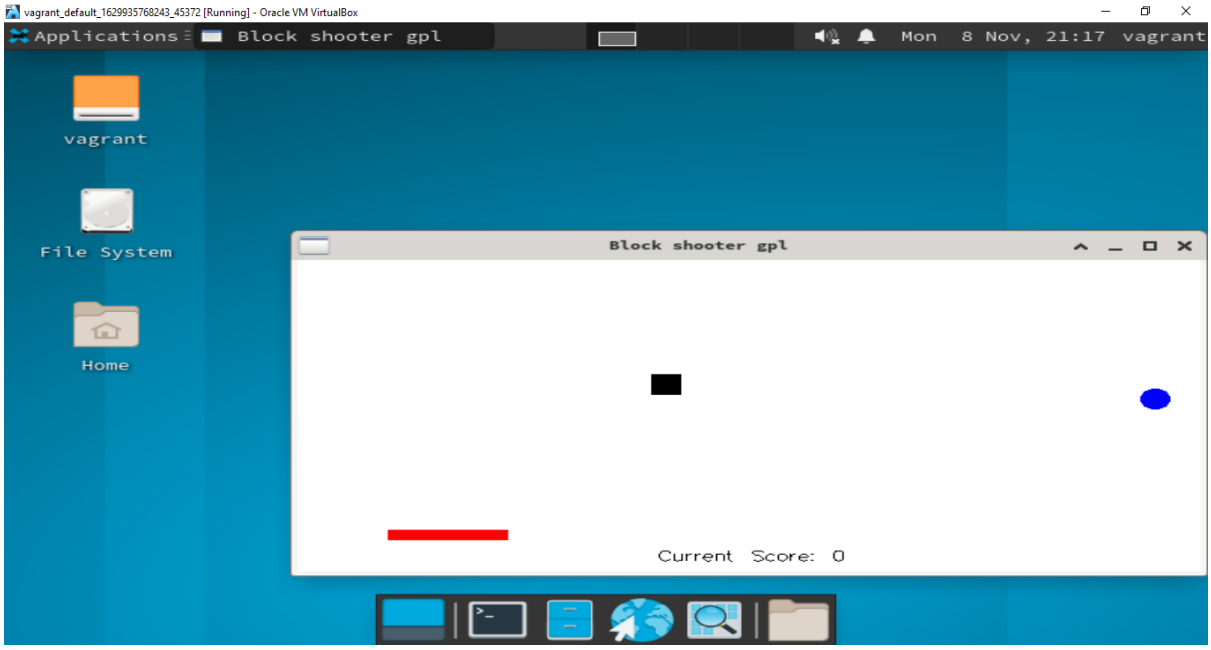
```
                block.y = random (window_height - block.h);
        // This is to get the new block y-position
        }

        //If block is formed below the paddle
        if (block.y < paddle.y + paddle.h)
        {
                block.x = random(window_width - block.w);
                block.y = random (window_height - block.h);
        }
        cur_ball.x += ball_x_increment;
        cur_ball.y += ball_y_increment;
}

//Event handler on pressing right arrow
on rightarrow
{
        if (paddle.x < window_width - paddle.w )
                paddle.x += paddle_increment;
}

//Event handler on pressing left arrow
on leftarrow
{
        if (paddle.x > 0)
                paddle.x -= paddle_increment;
}
```

OUTPUT - >


vagrant_default_1629935768243_45372 [Running] - Oracle VM VirtualBox
Applications
Block shooter gpl
Mon 8 Nov, 21:17 vagrant
vagrant
File System
Home
Block shooter gpl
Current Score: 0

Chapter 4

Conclusion

In this project we develop the whole programming language which we can used to make the game. So when we developed it completely we can see we can able to make the game on it just like minecraft , or flappy bird very easily , which can be made other compliers too but our compiler is just like English and you have to make the logic and then write down the code. Which makes the game development easier and faster.