Table of Contents

Introduction and Background 1
1.0 Mini-languages 1
1.1 Description of Core Mini-language 1
1.2 Tools for Building Interpreter 4

Building the Interpreter 5
2.0 Steps to Complete the Project 5
2.1 Writing the Syntax Expressions for Core 5
2.2 Variable Checking With Hash Table in Core 16
2.3 Assignment Statements and Math Operations 22
2.4 If Statements and Compares 28
2.5 While Loop 34
2.6 Input and Output Statements 36

Interpreter Testing 41
3.0 Testing the Program 41
3.1 The Fibonacci Series 41
3.2 The Time Conversion Program 42
3.3 The Variable Not Declared Error 44
3.4 The Variable Already declared Error 44
3.5 The Negative Value Error 45
3.6 The Variable Not Initialized Error 46
3.7 The Syntax Errors 47
3.8 The Size And Overflow Errors 48
3.9 The Never Ending While Loop 49

Creating X-Windows 51
4.0 X-windows 51
4.1 Writing Calls 51
4.2 The Quit Function 52
4.3 The Cut-Paste Function 52
4.4 The Do-Parse Function 53

Summary 55
5.0 Summary and Future Work 51

Appendix

Code

List of Figures

1.1 Context Free Syntax of Mini-Language Core in BNF 3

2.1 Simplest Lex Program 5

2.2 Expanded Lex Program 6

2.3 Variable Definition 6

2.4 Returning Assignment Operators 6

2.5 Defining Math Operators 7

2.6 Complete Core Grammar 7

2.7 Simple Yacc Program to Check Syntax 8

2.8 Defining Syntax Order 9

2.9 Yacc Syntax Expansion 9

2.10 Core With Declaration Statements 10

2.11 Expansion of Statement Variable 10

2.12 Adding Variables to Input and Output Strings 11

2.13 Demonstrate Input and Output Statements 11

2.14 Breakdown of Math Expressions 12

2.15 Expanded If Statement in Yacc 12

2.16 Expansion of Whilelp Variable 13

2.17 Complete Lex and Yacc for Expression Checking 15

2.18 Complete Data File for Syntax Expression Checking 16

2.19 Structure for Hash Table 16

2.20 Create the Hash Table 17

2.21 Function to Return Pointer From Hash Table 17

2.22 Function to Check If Variable in Hash Table 18

2.23 The Dec Subdivision 19

2.24 The Decvar Subdivision 19

2.25 Variable Not Declared, Within Decvar 20

2.26 The Iovar Subdivision 21

2.27 Variable Checking in Assignment Statement 21

2.28 Variable Checking in Operand Subdivision 22

2.29 Various Types of Assignment Statements 22

2.30 Assign New Value Function 23

2.31 Assign Negative Value Function 23

2.32 Using Assign Negative Value Function 24

2.33 Get Value Function 24

2.34 The Math Expansion 25

2.35 Expansion of Factor 25

2.36 Leftbrace exp Rightbrace 26

2.37 Determining First Variable in an Assignment 27

2.38 Datafile For Complex Assignments 28

2.39 True or False Function 29

2.40 Complete comp Subdivision 30

2.41 Example of If Statement 30

2.42 Assigning Statement Type 31

2.43 Incrementing Current Level 31

2.44 Adding The Block Variable 32

2.45 The End If Switch Subdivision 33

2.46 The Else Switch 33

2.47 The Elses Subdivision 34

2.48 Using the F-Tell Function 34

2.49 The While Switch Expansion 35

2.50 The End Switch Expansion 36

2.51 The Iovar Expansion 37

2.52 Use of the Get Value Function 37

2.53 Opening the File for Input 38

2.54 Assigning to HashTable From Input Statement 39

2.55 Completed Iovar Expansion 40

3.1 Fibonacci Series Program 41

3.2 Output From Fibonacci Series 42

3.3 Time Conversion Program 43

3.4 Output Time Conversion Program 44

3.5 Program For Variable Not Declared Error 44

3.6 Output for Figure 3.5 44

3.7 Program for Variable Already Declared Error 45

3.8 Output for Figure 3.7 45

3.9 Program for Negative Value Syntax Error 45

3.10 Output for Figure 3.9 46

3.11 Program for Negative Value Error 46

3.12 Output for Figure 3.11 46

3.13 Program for Variable Not Initialized 47

3.14 Output for Figure 3.13 47

3.15 Syntax Errors 48

3.16 Finding the Maximum Integer Allowed 48

3.17 Size Error 48

3.18 More Size Errors 49

3.19 Overflow Error Output 49

4.1 Main Window of the Interpreter 51

4.2 Function Declarations 52

4.3 Declaration of the Callbacks 52

4.4 Quit Function 52

4.5 Cut and Paste Function 53

4.6 The Do-Parse Function 54

ABSTRACT

This project is the design and implementation of a mini-language interpreter which is implemented with a windows type interface. The mini language is Core. The interpreter will be used to teach students at Texas A&M - Corpus Christi the concepts of programming without limiting the students to a particular language.

1.0 Mini-languages

A mini-language is a programming language with limited functionality, syntax, and semantics. It may be a subset of a general language or a small language in its own right. Mini-languages are useful in teaching students to develop skills involving the design of programming languages. Mini-languages contain basic control structures found in all procedural programming languages. A mini-language allows students to study the workings of control structures without having to deal with the nuances of a fully functional language. Mini-languages allow the programmer to study programming languages with severe constraints on language flexibility.

Creating an interpreter for the mini-language Core will allow programmers to fully implement and test the limits of this language. The interpreted mini-language allows teaching students how to recognize syntax vs semantic errors when programming. The X-Windows System interface will provide ease of use, allowing the programmer to concentrate on the mechanisms of the language rather than the intricacies of the interpreter. The implementation of this interpreter will also allow programmers to study the utilization of Unix tools to create an interpreter and possibly find better ways of writing interpreters.

1.1 Description of Core Mini-language

Core is a Pascal-like mini-language with limited functionality developed by Marcotty and Ledgard. Variables can be only positive integers. Variable names must be capital letters with an optional underscore anywhere after the first letter. There are two control structures - an "if then else" selector and a "while" loop. Core uses a "not equals" symbol (≠) for comparison that is not implemented on an ASCII keyboard, therefore the C convention of "not equals" (!=) is used.

Core has an assignment operator (:=) and a compare operator (=) that are be handled in the interpreter. Comparison operations for the language also include greater/less than (>/<). Core contains the mathematical operations multiplication (*), addition (+), and subtraction (-).

Core has six semantic and syntax error conditions that can lead to an early termination of the program. The undefined value error occurs when the variable has not been declared but is assigned an integer value in the program. The overflow error occurs when values applied to a math operator exceed the capacity of the machine to handle the result. The negative value error occurs during subtraction when the result of the operation is not positive, as Core can only handle positive values. A negative value entry will be a syntax error. Insufficient data error occurs when the variable has been declared but never assigned a value. The size error occurs when the size of the integer being used exceeds the capacity of the machine on which it is running. The illegal character error occurs when the syntax rules are violated such as a digit within a variable name or a lower case letter. The context free syntax in Backus-Naur Form (BNF) is illustrated in Figure 1.1.

<program>	::=	program
		  <declaration-sequence>
		begin
		  <statement-sequence>
		end ;

<declaration-sequence>	::= |	<declaration> <declaration><declaration-sequence>
<statement-sequence>	::= |	<statement> <statement><statement-sequence>
<declaration>	::=	<identifier-list> : integer ;
<identifier-list>	::= |	<identifier> <identifier> , <identifier-list>
<statement>	::= | | | |	<assignment-statement> <if-statement> <loop-statement> <input-statement> <output-statement>
<assignment-statement>	::=	<identifier> := <expression> ;

<if-statement>	::=
	
	|
		if <comparison> then
		  <statement-sequence>
		end if ;
		if <comparison> then
		  <statement-sequence>
		else
		  <statement-sequence>
		end if ;

<loop-statement>	::=	while <comparison> loop <statement-sequence> end loop ;
<input-statement>	::=	input <identifier-list> ;
<output-statement>	::=	output <identifier-list> ;
<comparison>	::=	(<operand><comparison-operator><operand>)
<expression>	::= | |	<factor> <expression> + <factor> <expression> - <factor>
<factor>	::= |	<operand> <factor> * <operand>
<operand>	::= | |	<integer> <identifier> (<expression>)
<comparison-operator>	::=	< | = | != | >
<identifier>	::= | |	<letter> <identifier><letter> <identifier> _ <letter>
<integer>	::=	<digit> | <integer><digit>
<letter>	::= | |	A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 1.1, Context Free Syntax of Mini-Language Core in BNF

1.2 Tools for Building Interpreter

The project also involves the use of the lex and yacc Unix tools to create the interpreter. Lex is used solely to perform lexical analysis and pass tokens to yacc. Yacc has the grammar rules for the language to be interpreted and repeatedly calls lex so that the syntax can be checked. The main program is included in yacc to allow for error checking. The X-Windows System is integrated into the main program in yacc to produce the windows interface.

2.0 Steps to Complete the Project

1. Syntax expressions.
2. Variable checking with hash table.
3. Assignment statements and math operations.
4. If statements and compares.
5. While loop.
6. Input and output statements.
7. Test.

2.1 Writing the Syntax expressions for Core

Syntax expressions for a mini-language enforce the checking for basic errors which can be anything from a keyboard error to a wording error as simple as not ending the program with the word **end**. The yacc generated parser takes the tokens from lex and fits them into the expression. If there is a token that is not found in the expression, a syntax error is given by yacc and the program is halted. Syntax rules for Core also require that reserved words be in lowercase while all variables are in uppercase.

Passing reserved words and math and comparison operators from lex to yacc was the first task. The simplest lex specification is shown in Figure 2.1.

```
%%
program             return PROGRAM;
begin               return BEGINS;
end                 return END;
";"                 return SEMICOLON;
%%
```

Figure 2.1. Simplest Lex Program.

To run the file in Figure 2.1, save the file with a ".l" extension. The filename used to save the lex is spec1.l Then use the unix command **lex spec1.l**. This creates the file lex.yy.c. This file is then included in the yacc program.

For the Core program the lex file returns upper case versions of the reserved words to distinguish between tokens and variables in the yacc program, shown in Figure 2.2.