<Student Name>

The Negotiator

Senior Project Proposal, Spring 2002

Table of contents should be placed here.

Summary:

This project consists of the design and implementation of a Visual Basic environment aimed at computer science one (CSCI) students. This environment will allow the user to type in C++ code fragments and see immediate output according to the code entered. The language allowed for this project will focus on a limited subset of the Borland C++ 5.0 language. This subset will be composed of those statements most encountered during CSCI, such as assignment statements and cout statements. An interpreter will be designed to produce accurate output in a parallel text box or accurate error messages created according to the rules of C++. Professors could utilize this program as a laboratory exercise or homework assignment to provide students with additional practice outside of class. This added practice would utilize more of the student's time in thinking about the actual programming principles at hand as opposed to the other aspects of writing a complete C++ program to support the code fragment.

Significance:

This project will provide CSCI students with a new learning tool. If a student has never seen code before, nor ever attempted programming prior to this course, this project will simplify the learning process. For the student to produce results from code, they must write an entire C++ program consisting of, but not limited to, library inclusions, a main function, and variable declarations that can be confusing to a student. This project will simplify this process by limiting the student to a subset of the C++ language. In turn, the student can work in code fragments instead of full programs, much like a professor would write on a white board during class. This limitation will focus the student on the principles of the code they are attempting to write. This language subset will include some of the following major statements: loops, assignment statements, cout statements, and cin statements. Thus for instance, a student will not need an entire program to output "Hello" to the screen, but simply the cout statement necessary to write "Hello" to the screen.

The programming side of this project is difficult. An interpreter is needed to drive this project and produce the necessary output and error messages. Many stages of designing this interpreter will be similar to those used in developing a compiler. One of the major hurdles of this project is writing code in Visual Basic as opposed to C++, which is a more powerful language. To add complexity, a variable declaration control will be created, whereby all variables are declared using a point and click method. Variable arrays will also be allowed using this control. On the programming side, nesting of if…else statements and loops will add dimension to what the student can program into the code window and add complexity to the interpreter. Furthermore, providing a clean and easy-to-use interface is also a concern. The focus is not to be flashy and creative, but to provide functionality and feedback to the user. Thus, the user will be able to load and save fragments, print code, and use a step command to see the results of their code line by line.

<u>**Required Tools and Availability:**</u>

This project will be developed on Microsoft Visual Basic 6.0. This software can be accessed both in the DePauw computer labs as well as on my PC.

Demonstration Plans:

The software necessary to demonstrate this program will be available on the moderator computer in room 26. During checkpoints, sample test programs will be run to demonstrate those stages completed between checkpoints. The output window will be used to show the results of the test programs that will demonstrate each stage.

Qualifications:

This project is based on the compilers course which was taken in the fall semester of 2001. To build this interpreter, a basic understanding of Borland C++ code is necessary, particularly syntax for loops. The grammar used in this interpreter will be taken from C++; mainly those components CSCI students will use in their course. Visual Basic was self-taught during winter term through the use of the Step by Step manual written by Michael Halvorson on loan from Dave Berque. Practice programs were provided on a CD-ROM as well.

Project Specifications:

Functional Specification:

This project will perform two basic functions; $^{1)}$ allow the user to type in a code fragment consisting of one or more C++ statements; $^{2)}$ produce output or error messages in the corresponding window of the interface. If there are errors in the code, the program will generate a corresponding message including a dialog box signaling that there was a failure. Additional features to compliment these functions will include several menu items and a “break” button. The menu items will provide the user with the ability to save and load previous code to the code window, clear any code present in the window, create a “new” interface, and the option to print the code window. The “break” button will act as a trap door for the possibility that the user has written an infinite loop. This button will halt the interpreter and return control to the user.

User Interface:

The interface for this project will be very basic. There will be two text windows present when the program loads, the program code window and the output window. Between these two windows are a “run” command button, used to start the interpreter, and a “break” command button (described above). The output window will show text only if the code entered is correct and produces output. If the code is incorrect, a dialog box will appear signaling there is a failure, and an error window will appear at the bottom of the interface with the appropriate error. After the error has been identified, control will be given back to the user in order to correct the code entered. Furthermore, the menu items mentioned earlier will give the user greater control over the program.

Technical Details:

Tokens:

The interpreter will be implemented based on a context free grammar developed from a subset of C++ code. The tokens identified for the purposes of this project are as follows:

```
Keywords: int   char   while   for   do   cout   cin   if   else   endl

Tokens: +   ++   -   --   *   /   %   =   ==   <   <=   >   >=   !=   {

        }   (   )   ;   ,   “   [   ]   ||   &   <<   >>
```

Comments will be designated by “//” followed by any text. Comments will be terminated by the carriage return and thus are limited to a maximum of one line. The multi-line C++ comment designated by “/*” and terminated by “*/” will not be allowed for simplification purposes on the user end.

Rules:

1. A variable identifier will consist of a letter followed by any combination of letters and digits. Identifiers are limited to a maximum of 20 characters and are case sensitive.
2. Integer constants are limited to 7 digits.
3. String constants are limited to 60 characters in length.
4. The structure for a valid program is as follows:

```
{ code fragment
      •
      •
      •
  code fragment }
```

   a. Any variable declarations will be made using a separate variable control. The user will be able to select from a type of variable (int, char, array), name the variable, set the size of the variable (for arrays), add and delete variables, and clear all the variables using this control.
   b. The minimum requirements for a valid program consist of a single statement enclosed by braces.
5. All expressions must contain a space between each token. For instance, 5 + 5 > 9 is allowed. However, 5+5>9 is not allowed. This is to allow negative integers.

Implementation:

The interpreter will operate using two basic parsers developed from parsers written in the compiler course. The first will be the expression parser, based on the operator precedence algorithm. The second will be the recursive descent parser, based on the context free grammar. These two parsers will work in conjunction with a symbol table implemented to handle any variable declarations. Finally, the interpreter will run output generation whereby any output to the output window will be written.

Timeline:

Checkpoint #1

Lexical Scanner

Note: For all test programs of this stage, the program will only search for valid tokens; syntax is ignored.

- I will run the program on a set of valid tokens. Each token will be written to the output window along with its length, class, and subclass.
- I will run the program on a set of tokens containing two invalid tokens. I will run the program three times, demonstrating correct error messages identifying the incorrect tokens. After each run I will edit out one of the invalid tokens, until on the final run, it is again a valid program.
- I will run the program on a set of tokens containing an identifier that consists of exactly 20 characters and an identifier that exceeds 20 characters. This will demonstrate the program's ability to identify the limited specifications as well as correctly identify the error.
- In a similar fashion, for integers and strings I will run the program on a set of tokens containing a token that meets their maximum length and a token that exceeds their maximum length. In each case, the correct error will be identified.

Expression Parser

Note: For all test programs of this stage, a successful program will consist of a valid mathematical expression.

- I will run the program on a valid expression containing only mathematical operations; i.e. add, subtract, multiply, divide, and mod. The reduction of the expression will be written to the screen.
- I will run the program on a valid expression containing only relational operators. The reduction of the expression will be written to the screen.
- I will run the program on a valid expression containing a mixture of mathematical operations and relational operators. The reduction of the expression will be written to the screen.
- I will run the program on two invalid expressions, one containing two integers without an operator, and a second containing two operators without an operand. In each case, the proper error message will be displayed.
- I will run the program on an expression containing non-mathematical symbols. The illegal symbol will be identified in the proper error message.

Miscellaneous

- I will demonstrate several menu items. I will demonstrate the ability of the open command to open a text file for interpretation, the close command to close the current file, the save as command to save the current text in the code window to a file, and the exit command to quit the program.

Checkpoint #2

Recursive Descent Parser

Note: For this stage, no output will be produced in the output window. A valid program will only display a "Success" dialog box.

- ♦ I will run the program on a simple valid code fragment. This will include assignment statements, cout statements, and cin statements.
- ♦ I will run the program on a complicated valid code fragment. This will include if…else statements and a loop.
- ♦ I will run the program on a complex valid code fragment. This will include nested if…else statements and nested loops.
- ♦ I will run the program on several invalid code fragments demonstrating the program's ability to detect syntactical errors. There will be a minimum of three test cases detecting errors on such things as a program with no fragment, illegal statements, missing statement elements, and missing braces.

Miscellaneous

- ♦ I will demonstrate the new menu command that will create a "new" interface, which basically means reestablishing the program to its state when first initiated. I will also demonstrate the print command, which will print any text in the code window.

Checkpoint #3

Symbol Table

Note: For this stage, all code fragments will be syntactically valid.

- ♦ I will run the program on code containing no variables. A message will be written to the output window signaling that there were no variables declared.
- ♦ I will run the program on code containing variables. The variables and there type will be written to the output window.
- ♦ I will run the program on code accessing undeclared variables. An error message identifying the undeclared variable will be displayed.
- ♦ I will run the program on code containing duplicate variables. An error message identifying the duplicate variable will be displayed.

Preliminary Output

Note: For this stage, the result of any expression will be output to the screen regardless if it is the intention of the programmer.

- ♦ I will run the program on code containing expressions consisting only of mathematical operators.
- ♦ I will run the program on code containing expressions consisting only of relational operators.
- ♦ I will run the program on code containing expressions with a mixture of mathematical operators and relational operators.

Miscellaneous

- ♦ I will demonstrate the "break" button by running the program on an infinite loop and using the button to stop interpretation and return control to the user.

Checkpoint #4

Complete Output

Note: For this stage, all output to the screen will be as a result of cout statements.

- ♦ I will run the program on simple code. This will include assignment statements, cout statements, and cin statements. Variables of type array will not be allowed.
- ♦ I will run the program on complicated code. This will include if…else statements and loops. All variables will be allowed.
- ♦ I will run the program on complex code. This will include nesting of if…else statements and loops.
- ♦ Final note: It is planned to have the project done at this point. I have chosen to do this in order to have the final two weeks in order to do any final debugging if necessary, or, in the best case scenario, have the opportunity to get user feedback on the program. This feedback could be used immediately in implementation or as final thoughts on the future of such a program.

Bibliography

Halvorson, Michael. Microsoft Visual Basic 6.0 Professional Step by Step. Microsoft Press: Redmont, 1998.

Parsons, Thomas W. Introduction to Compiler Construction. Computer Science Press: New York, 1997.