Design and Implementation of an Interpreter Using Software Engineering Concepts

Fan Wu
Department of Computer Science
Tuskegee University
Tuskegee, Alabama, USA

Hira Narang
Department of Computer Science
Tuskegee University
Tuskegee, Alabama, USA

Miguel Cabral
Department of Computer Science
Tuskegee University
Tuskegee, Alabama, USA

***Abstract*—In this paper, an interpreter design and implementation for a small subset of C Language using software engineering concepts are presented. This paper reinforces an argument for the application of software engineering concepts in the area of interpreter design but it also focuses on the relevance of the paper to undergraduate computer science curricula. The design and development of the interpreter is also important to software engineering. Some of its components form the basis for different engineering tools. This paper also demonstrates that some of the standard software engineering concepts such as object-oriented design, design patterns, UML diagrams, etc., can provide a useful track of the evolution of an interpreter, as well as enhancing confidence in its correctness**

Keywords—Interpreter; Software Engineering; Computer Science Curricula

I. INTRODUCTION

In this paper, an interpreter design and implementation for a small subset of C Language using software engineering concepts are presented. This paper summarizes the development process used, detail its application to the programming language to be implemented and present a number of metrics that describe the evolution of the project. Incremental development is used as the software engineering approach because it interleaves the activities of specification, development, and validation. The system was developed as a series of versions (increments) where each version adds functionality to the previous version [1].

The paper will also focus on the relevance of compilers and interpreters to undergraduate computer science curricula, particularly at Tuskegee University. Interpreters and compilers represent two traditional but fundamentally different approaches to implementing programming languages. A correct understanding of the basic mechanisms of each is an indispensable part of the knowledge that every computer science student must acquire [2].

The paper is organized as follows: section II presents the background and related work, and section III describes the design and development process. The conclusions and future work are discussed in section IV.

II. BACKGROUND AND RELATED WORK

A. Background

The main purpose of a compiler or an interpreter is to translate a source program written in a high-level source language to machine language. The language used to write the compiler or interpreter is called implementation language. The difference between a compiler and an interpreter is that a compiler generates object code written in the machine language and the interpreter executes the instructions. A utility program called a linker combines the contents of one or more object files along with any needed runtime library routines into a single object program that the computer can load and execute. An interpreter does not generate an object program. When you feed a source program into an interpreter, it takes over to check and execute the program. Since the interpreter is in control when it is executing the source program, when it encounters an error it can stop and display a message containing the line number of the offending statement and the name of the variable. It can even prompt the user for some corrective action before resuming execution of the program.

The process is divided into 6 functional increments. Before moving to the next increment, the current increment has to be tested and validated. The increments are: 1. the framework, 2. the scanner, 3. the symbol table, 4. parsing and interpreting expressions and assignment statements, 5. parsing and interpreting control statements, 6. parsing and interpreting declarations.

Interpreters are complex programs, and writing them successfully is hard work. To tackle the complexity, a strong software engineering approach can be used. Design patterns, Unified Modeling Languages (UML) diagrams, and other modern object-oriented design practices make the code understandable and manageable [3, 4, 5].

B. Related Work

While the area of interpreter design, as a subset of compiler design is well-established and documented, it is not typically the subject of formalized software engineering concepts.

The application of object-oriented design principles to parsers and compilers has been investigated by Reiss and Davis [6]. Malloy, Power, and Waldon reinforce the argument for the application of software engineering concepts in the area of parser design [7]. Similarly, an incremental approach to compiler design is proposed by Ghuloum [8].

Demaille [9] states that compiler construction is a challenging process that requires material from virtually all computer science courses on the core curriculum. While the idea of compilers is usually furthered and explored in detail later on in an upper level course such as Compiler Construction, Xing [2] argues that the idea of interpreters rarely gets the same "treatment": There is no such a course targeting on interpreter constructions in most undergraduate computer science curricula at universities and colleges [9].

III. Design and Implmentation

A. Conceptual Design

The conceptual design of a program is a high-level view of its software architecture. The conceptual design includes the primary components of the program, how they're organized, and how they interact with each other. An interpreter is classified as a programming language translator. A translator, as seen at the highest level, consists of a front end and a back end. Both compilers and interpreters can share the same front end, but they'll have a different back end. Fig. 1 shows the conceptual design of the SimpleC interpreter. The front end of a translator reads the source program and performs the initial translation stage. Its primary components are the parser, the scanner, the token, and the source.

The parser controls the translation process in the front end. It continuously asks the scanner for the next token, and it analyzes the sequences of tokens to determine what high-level language elements it is translating. The parser verifies that what it sees is syntactically correct as written in the source program; in other words, the parser detects and flags any syntax errors. The scanner reads the characters of the source program sequentially and constructs tokens, which are the low-level elements of the source language. The scanner scans the source program to break it apart into tokens.

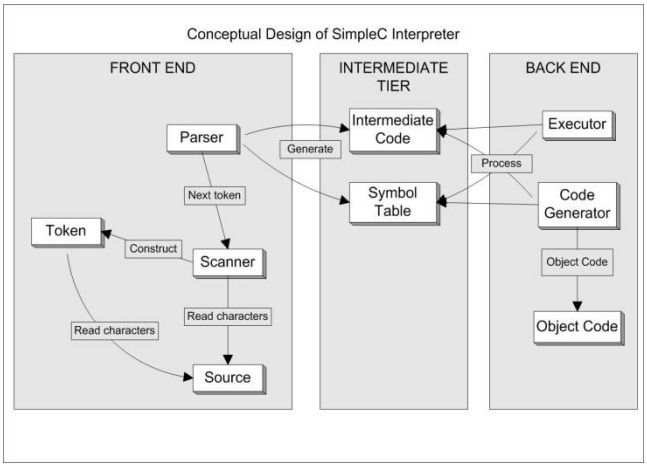


Fig. 1. Conceptual Design of the SimpleC Interpreter/Compiler

1) Syntax and Semantics

The syntax of a programming language is its set of grammar rules that determine whether a statement or an expression is correctly written in that language. The language's semantics give meaning to a statement or an expression. In Simple C, the statement (1):

```
a = b + c;                (1)
```

is a valid assignment statement. The semantics of the language tells that the statement says to add the value of variables 'b' and 'c' and assign the sum's value to the variable 'a'.

A parser performs actions based on both the source language's syntax and semantics. Scanning the source program and extracting tokens are syntactic actions. Looking for '=' token is a syntactic action, entering the identifiers 'a', 'b', and 'c' into the symbol table as variables, or looking them up in the symbol table, are semantic actions because the parser had to understand the meaning of the expression and the assignment to know that it needs to use the symbol table. Syntactic actions occur in the front end, while semantic actions can occur on either the front end or the back end.

B. Basic Interpreter/Compiler Framework

As mentioned previously, the project will be divided into functional increments, using software engineering concepts. In the previous section, the conceptual design of the compiler was briefly explained. In this increment, an initial implementation of a rudimentary interpreter will be presented after conceptual design.

The first part of this increment is to build a flexible framework that supports both compilers and interpreters. The framework will integrate fundamental interpreter and compiler components in the second stage. Finally, end-to-end tests will be run to test the framework and its components.

The goals for this increment are:

- A source language-independent framework that can support both compilers and interpreters
- Initial SimpleC source language-specific components integrated into the front end of the framework
- Initial compiler and interpreter components integrated into the back end of the framework
- Simple end-to-end runs that exercise the components by generating source program listings from the common front end and messages from the compiler or interpreter back end

1) Front End

The front end consists of the language-independent *Parser*, *Scanner*, *Source*, and *Token* classes that represent the framework's components. Consider the class diagram of the frond end in Fig. 2.

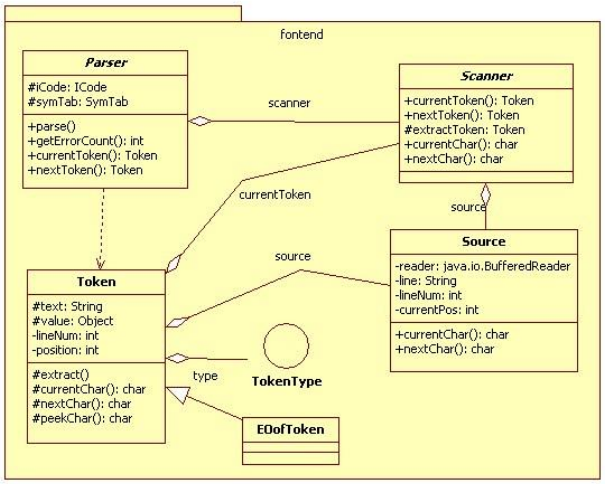


Fig. 2. Front end package

The parser and the scanner are closely related. The parser "owns" a scanner. The parser request tokens from its scanner, and so it has a dependency on tokens. The scanner owns the current token, it owns the source, and it passes the source reference to each token it constructs. Each token then also owns that. During its construction, a token reads characters from the source.

2) Messages

The parser may need to report some status information, such as an error message whenever it finds a syntax error. However, the parser should not worry about where it should send the message or what the recipient does with it. Similarly, whenever the source component reads a new line, it can send a message containing the text of the line and the line number. Keeping the senders of messages loosely coupled to the recipients of the messages minimize their dependencies. In complex applications, loose coupling allows you to develop components independently and in parallel [7].

3) Intermediate Tier

According to the conceptual design, the intermediate code and the symbol table are the interface between the front and back ends. Consider the following UML class diagram.

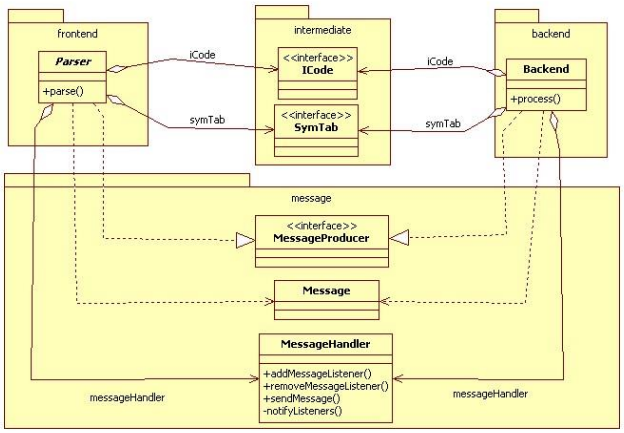


Fig. 3. Front end, intermediate, and backend packages

A framework Parser object in the front end package and a Backend object in the backend package own intermediate code and symbol table object as shown in Fig.3. Both classes, Parser and Backend, have the same relationships to the classes in the message package.

4) Back End

The conceptual design states that the back end will support either a compiler or an interpreter. Just like the Parser and Source classes in package front end, the Back End class in package backend implements the MessageHandler helper class. A compiler would implement the abstract method process to generate object code while an interpreter would implement the same method to execute the program.

5) Test and Validation

Up to this point a framework with components that are language independent has been completed. The backend of the framework can support either a compiler or an interpreter. To test the framework, a SimpleC program is used as the input. The output consists on the input program with line numbers followed by the number of statements, number of syntax errors, total parsing time, statements executed, run-time errors, and total execution time. The error recovery will be added in next increments.

C. Scanning

The second increment of the project consists on implementing a scanner. The scanner is the component in the front end of a compiler or an interpreter that performs the syntactic actions of reading the source program and breaking it apart into tokens. The parser calls the scanner each time it wants the next token from the source program. The goals for this increment are:

- Complete the design and development of the SimpleC scanner.
- The scanner should be able to:
- Extract SimpleC words, numbers, and special symbols from the source program
- Determine whether a word is an identifier or a SimpleC reserved word
- Calculate the value of a number token and determine whether its type is integer or real
- Perform syntax error handling

1) Syntax Error Handling

Every parser must be able to handle syntax errors in the source program. Error handling is a three step process:

- Detection: Detect the presence of a syntax error.
- Flagging: Flag the error by pointing it out or highlighting it, and display a descriptive error message.
- Recovery: Move past the error and resume parsing.

If an instance of one of the SimpleC token subclasses finds a syntax error, it will set its type field to the SimpleCTokenType enumerated value ERROR and its value field to the appropriate SimpleCErrorCode enumerated value.

If the scanner finds syntax errors (such as an invalid character that cannot start a legitimate SimpleC token), it will construct a SimpleCErrorToken.

2) How to Scan for Tokens

The scanner has to read each character at a time, skipping blank spaces. For example consider the following statement:

```
int a = 3;                (2)
```

After scanning the statement, the scanner has extracted the following tokens:

TYPE	TEXT STRING
Word (reserved word)	int
Word (identifier)	a
Special symbol	=
Number (integer)	3
Special symbol	;

The scanner reads and skips white space characters between tokens. When it's done, the current character is nonblank. This nonblank character determines the type of the token the scanner will extract next, and the character becomes the first character of that token. The scanner extracts a token by reading and copying successive source characters up to but not including the first character that cannot be part of the token. Extracting a token consumes all the source characters that constitute the token. Therefore, after extracting a token, the current character is the first character after the last token character. The SimpleC scanner can identify word tokens (identifiers and reserved words), special symbol tokens ('+', "-", etc.), and number tokens (unsigned integers and real numbers).

3) Test and Validation

To test this increment a SimpleC tokenizer utility was written. The tokenizer takes as input a SimpleC source program and outputs a description of a token or an error message in case of a syntax error. Fig. 4 shows the output of the tokenizer for input file simplec_mult.txt.

```
          001 // file-name simplec_mult.txt
          002 .
>>> DOT                line=002, pos = 0, text="."
          003 int mult = 2 * 4;
>>> INT              line=003, pos = 0, text="int"
>>> IDENTIFIER       line=003, pos = 2, text="mult"
>>> ASSIGN           line=002, pos = 3, text="="
>>> INTEGER          line=002, pos = 4, text="2"
>>> STAR             line=002, pos = 5, text="*"
>>> INTEGER          line=002, pos = 6, text="4"
           004 .
>>> DOT              line=004, pos = 0, text="."
```

Fig. 4. Output of SimpleC Tokenizer

D. The Symbol Table

The parser of a compiler or an interpreter builds and maintains a symbol table throughout the translation process as part of semantic analysis. The symbol table stores information about the source program's tokens, mostly the identifiers. As mentioned in previous increments, the symbol table is a key component in the interface between the front and back end.

Goals for this increment:

- A language-independent symbol table
- A simple utility program that parses a SimpleC source program and generates a cross-reference listing of its identifiers

1) The Symbol Table

The approach of this increment is to create the conceptual design of a symbol table, develop interfaces that represent the design, and finally write the classes that implement the interfaces. To verify the correctness of the source code, a cross-reference utility program will be used. It will exercise the symbol table by entering, finding, and updating information.

During the translation process, the interpreter creates and updates entries in the symbol table to contain information about certain tokens in the source program. Each entry has a name, which is the token's text string. The entry also contains information about the identifier. As it translates the source program, the interpreter looks up and updates the information.

The symbol table entry for an identifier will typically include its type, structure, and how it was defined. One of the goals is to keep the symbol table flexible and not limited to SimpleC-specific information. The basic operations a symbol table must support are: 1. Enter new information, 2. Loop up existing information, 3. Update existing information.

2) Conceptual Design of Symbol Table Stack

To parse a language like SimpleC, more than one symbol table might be needed (one table for each function or class, etc.). Because some procedures can be nested, the symbols need to be maintained in a stack. Fig 5 shows the conceptual design of the symbol stack. For this increment, only one table will be used but the concept will be explained for possible future extensions of the language. The symbol table at the top of the stack maintains information for the program, function, block, etc. that the parser is currently working on. As the parser works its way through the SimpleC program and enters and leaves nested functions and blocks, it pushes and pops symbol tables from the stack. The symbol table at the top of the stack is known as the local table.

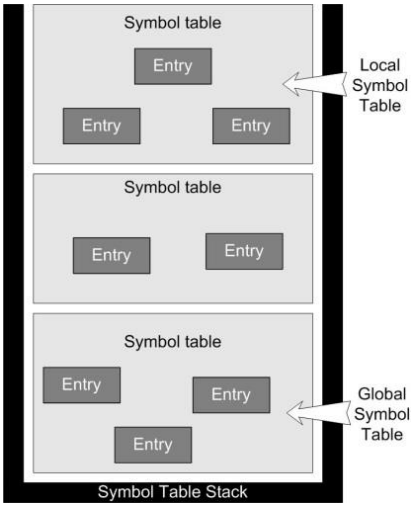


Fig. 5. The conceptual design of the symbol stack.

3) Test and Validation

The last step of the increment is to test the symbol table. This goal can be accomplished by generating a cross-reference of a SimpleC source program. The command line:

```
execute -x example01.txt
```

is used to generate a cross-reference listing of the identifiers found in the file "example01.txt".

```
 ----jGRASP exec: java SimpleC compile -x simplec_assign_ex.txt

001 // file-name: simplec_assign_ex.txt
002 .
003 {
004 five = 5;
005 ten = 10;
006 fifteen = five + ten;
007 }
008 .
009

                  9 source lines.
                  0 syntax errors.
               0.03 seconds total parsing time.

===== CROSS-REFERENCE TABLE =====

Identifier       Line numbers
----------       ------------
fifteen          006
five             004 006
ten              005 006

                  0 instructions generated.
               0.00 seconds total code generation time.

 ----jGRASP: operation complete.
```

Fig. 6. Cross-reference table for simplec_assign_ex.txt

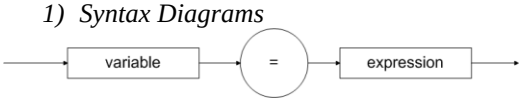
After the source program listing, all of the source program's identifiers are listed alphabetically. Following each identifier name are the source line numbers where the identifier appears as shown on Fig. 6.

E. Expressions and Assignment Statements

In the previous increment, a symbol table was created. The parser builds and maintains the symbol tables on the symbol table stack during the translation process. The parser also performs the semantic actions of building and maintaining intermediate code that represents the source program in the form of parse trees. The back end will then interpret the parse trees in order to execute statements and expressions. The goals for this increment are:

- Parsers in the front end for certain SimpleC constructs: assignment statements, compound statements, and expressions.
- Flexible, language-independent intermediate code generated by the parsers to represent these constructs.
- Language-independent executors in the interpreter back end that will interpret the intermediate code and execute expressions and assignment statements.

1) Syntax Diagrams



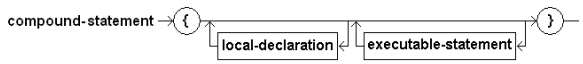


Fig. 7. Assignment statement and compound statement

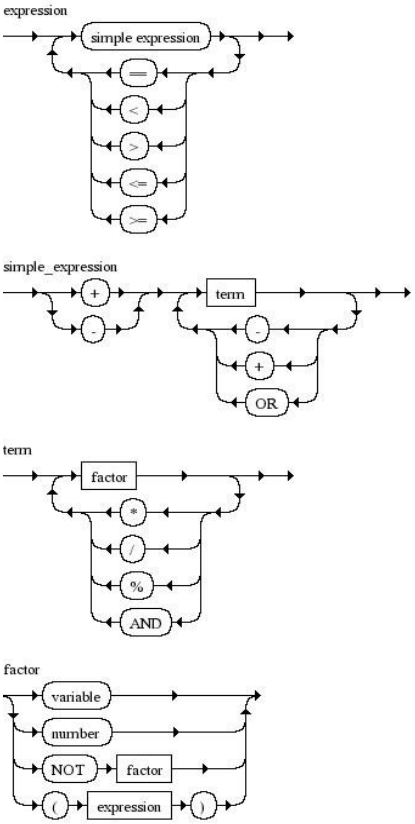


Fig. 8. Syntax diagrams for SimpleC expressions

Figs. 7 and 8 show the syntax diagrams that guide the development of the parsers that will generate the appropriate intermediate code.

2) Intermediate Code

A data tree structure represents the SimpleC intermediate code. Therefore, the intermediate code takes the form of a parse tree. A parse tree consists of sub-trees that represent SimpleC constructs, such as statements and expressions. Each tree node has a node type and a set of attributes. Each node other than the root node has a single parent node. The industry-standard XML can represent the tree structures in text form.

3) Executing Expressions and Assignment Statements

Expressions and statements are executed in the back end of the interpreter.

The intermediate code that represents the parse trees was implemented using language-independent classes in the back end. Consider the UML diagram if Fig. 9 for the statement executor classes in the back end package.

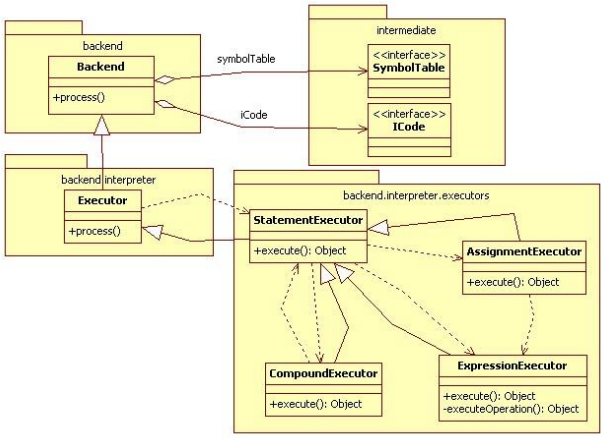


Fig. 9. Executor subclasses in the back end

4) Test and Validation

To test and validate this increment, a simple interpreter was written. The simple interpreter takes as input a SimpleC source program. Only assignment statements, compound statements, and expressions are recognized by the interpreter. Fig. 10 shows the output of file simplec_assign.txt.

```
 ----jGRASP exec: java SimpleC execute simplec_assign_ex.txt

001 // file-name: simplec_assign_ex.txt
002 .
003 {
004 a = 10;
005 b = a + 2;
006 c = 2 * (b - 2);
007 }
008 .
009

                  9 source lines.
                  0 syntax errors.
              0.03 seconds total parsing time.

 ------- OUTPUT -------

>>> LINE 004: a = 10
>>> LINE 005: b = 12
>>> LINE 006: c = 20

                  3 statements executed.
                  0 runtime errors.
              0.00 seconds total execution time.

 ----jGRASP: operation complete.
```

Fig. 10. Output of simplec_assign.txt

F. Control Statements

The next increment focuses on parsing and interpreting control statements. The goals for this increment are:

- Parsers in the front end for SimpleC control statements if, while, and for.
- Flexible, language-independent intermediate code generated by the parsers to represent these constructs.
- Reliable error recovery to ensure that the parsers can continue to work despite syntax errors in the source program.

The syntax diagrams shown in Fig. 11 were used to guide the development of the parsers.

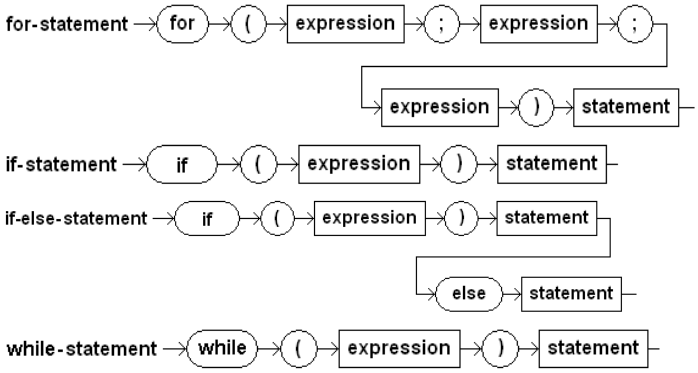


Fig. 11. Syntax diagrams for SimpleC control statements

1) Error Recovery

When the parser encounters an error, the three possible options for error recovery are: 1. Terminate the program after encountering a syntax error, 2. Attempt to parse the rest of the source program, 3. Skip tokens after the erroneous one until it finds a token it recognizes and safely resume syntax checking.

The first two options are undesirable. To implement the third option, a parser must "synchronize" itself frequently at tokens it expects. Whenever there is a syntax error, the parser must find the next token in the source program where it can reliably resume syntax checking [7].

2) Interpreting Control Statements

The interpreting capabilities of the program increase after each increment. It is time to add new executor classes for SimpleC control statements. The control statement executor classes can be appreciated in Fig. 12.

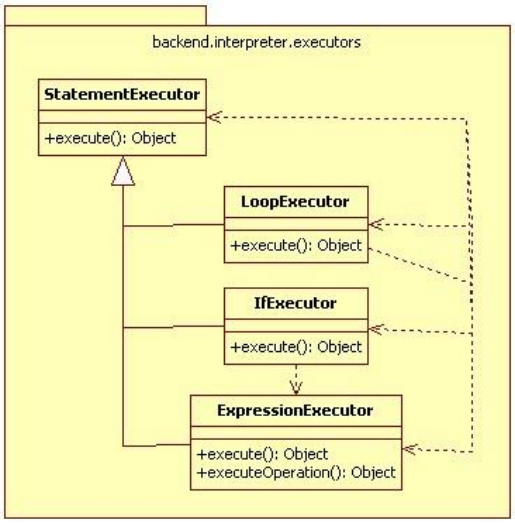


Fig. 12. Control statement executors in the backend

3) Test and Validation

To test this increment, a syntax checker utility was written to identify syntax errors. Also, the simple interpreter described in the previous increment was expanded. The interpreter takes as input a SimpleC source program. At this point, the program

identifies conditional statements and loop statements. Fig. 13 shows a sample output of a SimpleC if-statement.

```
 001 // file-name simplec_if.txt
002 .
003 a = 2;
004 b = 3;
005 if(b > a)
006 {
007  a = a + b;
008 }
009 .
        8 source lines.
        0 syntax errors.
        0.02 seconds total parsing time.

     -------- OUTPUT --------
       >>> LINE 003: a = 2
       >>> LINE 004: b = 3
       >>> LINE 007: a = 5
        4 statements executed.
        0 runtime errors.
        0.01 total execution time.
```

Fig. 13. Execution of a SimpleC if statement

G. Parsing Declarations

Parsing declarations expands the work in The Symbol Table increment because all the information from the declarations has to be entered in the symbol table. The goals for this increment are:

- Parsers in the front end for SimpleC type definitions and type specifications.
- Additions to the symbol table to contain type information.

1) SimpleC Declarations

There are three basic types of variables in SimpleC; they are: char, int, and float. The syntax diagram is shown in fig. 14.

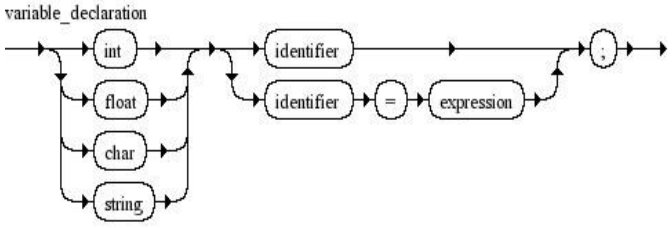


Fig. 14. Syntax diagram for variable declarations

2) Types and the Symbol Table

The type specification parser developed in this increment enters type information into the symbol table. The first step is to design language-independent interfaces that treat a type specification simply as a collection of attributes.

3) Parsing SimpleC Declarations

In previous increments it is assumed that identifiers were variables. For this increment, an identifier's symbol table entry must indicate how it was defined. Fig. 15 shows the classes for parsing declarations.

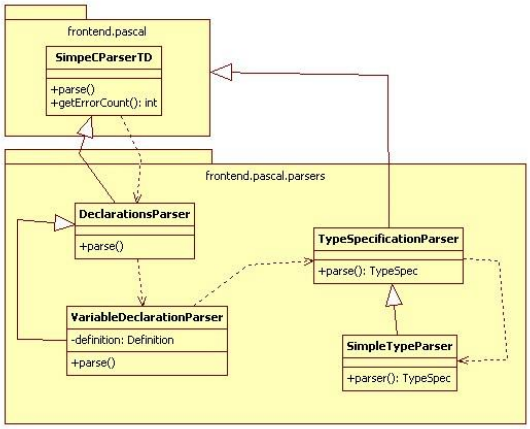


Fig. 15. The classes for parsing declarations

4) Test and Validation

To test and validate the code written for this increment, a SimpleC cross-reference utility similar to the one implemented in The Symbol Table increment, is implemented. The output of the cross-reference utility includes the line number where an identifier is found and how it is defined, as shown in Fig. 16.

```
          ==== CROSS-REFERENCE TABLE ====
        *** PROGRAM simplec_assign.txt ***
  Identifier     Line numbers    Type specification
-----------------------------------------------------
    a             002 005         Defined as: integer
    b             003 006         Defined as: integer
    c             004 007          Defined as: real
                        …
```

Fig. 16. Sample output of cross-reference table

IV. Conclusions and Future Work

A. Conclusions

In this paper, the design of an interpreter for the SimpleC programming language in the context of a software engineering project has been presented. The paper also has demonstrated that some of the standard software engineering concepts such as object-oriented design, design patterns, UML diagrams, etc., can provide a useful track of the evolution of an interpreter, as well as enhancing confidence in its correctness. A similar project could be introduced at Tuskegee University to meet some requirements not satisfied by shorter projects. Some requirements include, but are not limited to, writing a complete project using challenging algorithms and data structures, use of different development tools, object-oriented design, and team management which is an important issue to consider given that only team work in software engineering and database courses.

B. Future Work

Future work will focus on creating an interactive source-level debugger for the SimpleC language that enables the use of command lines to interact with the interpreter as well as an Integrated Development Environment (IDE) with a graphical user interface (GUI). If time is not a constraint, the interpreter will be extended to a SimpleC compiler that generates object

code for the Java Virtual Machine (JVM). The compiled programs will then be able to run on multiple platforms.

REFERENCES

[1] Sommerville, I. Software Engineering. Addisson Wesley, 9th edition, 2010

[2] C. Xing. "How Interpreters Work: An Overlooked Topic in Undergraduate Computer Science Education," Proc. In CCSC Southern Eastern Conference, JCSC Vol. 25,Iissue 2. December 2009

[3] R. Mak. Writing Compilers and Interpreters: A Modern Software Engineering Approach Using Java. Wiley, 3rd edition, 2009

[4] H. Deitel, Java: How to Program (early projects), 10th edition, Prentice Hall Inc. , 2014.

[5] R. Sebesta, Concepts of Programming Languages, 10th Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2012.

[6] S. Reiss, and T. Davis. "Experiences Writing Object-Oriented Compiler Front Ends". Tech. Rep., Brown University, January 1995.

[7] B. Malloy., J. Power, and J. Waldron. "Applying Software Engineering Techniques to Parser Design: The Development of a C# Parser," in Proc. of SAICSIT 2002, pp. 74–81

[8] A. Ghuloum, "An Incremental Approach to Compiler Construction," in Proc. of the 2006 Scheme and Functional Programming Workshop, 2006, pp. 28.

[9] A. Demaille. "Making Compiler Construction Projects Relevant to Core Curriculums," In proceeding of: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005, Caparica, Portugal, June 27-29, 2005.