Writing a Simple **Compiler** From Scratch

Barton T. Stander

Preface

Why the need for yet another compiler book?
Numerous compiler books already exist. Most are 700-1000 page monstrosities, including 23 different ways to parse, 17 different approaches to error handling, and 39 tricks for optimizing. These theory books spend page after page talking about aspects of writing a compiler, yet somehow never manage to actually write one.

What does this compiler book offer?
This text is designed to be more practical. It's a fraction of the size of most - meaning a student might bother to read it. It only discusses one way to write a compiler—an easy yet effective way. Most importantly, this approach guides the student in building a complete compiler from scratch, emphasizing software engineering principles along the way.

But isn't all that theory important?
Since very few computer science graduates end up writing compilers for a living, learning the detailed intricacies of the optimal compiler seems less relevant. On the other hand, many computer scientists use a compiler as their primary tool. The programmer who understands his tools is the better programmer, just as the great painter must understand his paints and brushes.

The best way to understand a compiler is to write one. Even if the result is smaller, slower, and inferior to commercial applications, the writing process will empower students with understanding. A one-semester project cannot begin to tackle the creation of a complex compiler, so this text guides the student in building a *Simple Compiler*.

What else is this book good for?
Most computer science classes involve writing several small software projects over the course of a semester. These projects do not accurately reflect the large-scale products a student will work on after graduation. This book provides an opportunity to create a relatively large and meaningful project. Such an accomplishment, (using good software engineering strategies) serves as a blueprint for future, more complex tasks. It also becomes a solid addition to a portfolio.

In addition, the student may be able to use parts of his finished code in other projects. I used code from my compiler to enhance graphing calculator software. The original release interpreted the formulas entered by the users. My new version compiles the formulas on the fly, yielding code that executed many times faster.

Who should use this book?
This text is ideal for a single semester junior or senior-level course in computer science or a related discipline.

What are the prerequisites?
Students should be proficient in C++. The examples in this book were compiled with Microsoft's Visual Studio and Linux g++. But the student should be able to complete most of the *Simple Compiler* project using any compiler and environment. The machine language sections at the end require Linux. Familiarity with software engineering techniques is helpful, though, for many students, this course may provide the first opportunity to exercise those skills on a large project. Knowledge of assembly language is a plus, but not essential. Chapter 8 provides the minimal assembly-language knowledge needed to complete the project. Likewise, Chapter 1 reviews necessary information on languages and grammars, and may be skipped by the familiar student.

Any questions?
I hope this hands-on approach is instructive and enlightening. Problems, questions, corrections, and suggestions should be directed to the author. I will make every effort to respond promptly.

Acknowledgments:
Special thanks to Carol Stander, Marc Wilkinson, Douglas Wardle, and Don Hoefelmann for providing some of the illustrations, and for their editing suggestions. And thanks to my students, who endured countless drafts of this book over the past several semesters.

Sincerely,

Barton T. Stander, Ph.D.
Professor of Computer Science
Utah Tech University
email: bart.stander@utahtech.edu

Contents

Overview

This workbook directs students, step by step, to create a compiler from scratch. It is called a *Simple Compiler* because it is too limited to compete with advanced compilers such as Visual Studio, though it may serve as a starting point for a more advanced compiler. The *Simple Compiler* is a significant software engineering project, which should take the better part of a semester. Keep in mind that there are many approaches to writing a compiler. This will not be the most versatile, fastest, or optimized compiler, but it will be a significant project, accomplished in one semester.

A **Compiler** is a translator, from a **source language** to a **destination language**. The source is usually a high-level programming language such as C++ or Java, and the destination is usually assembly language or machine code. Before one can write a compiler, one needs to comprehend the basics of formal languages, as well as grammars and machines. One also needs to know something about the machine language of Intel-based machines, as that will be the destination language for this project. The source language will be a subset of C++, which the reader should already know. Readers proficient with language theory and/or Intel-machine language may choose to skip those chapters.

A compiler project can be divided into several modules, such as the **State Machine Simulator**, the **Scanner**, the **Parser**, the **Interpreter**, and the **Code Generator**. This work will examine and build each of these in turn.

Since another goal of this book is to promote the practice of good software engineering techniques, many such ideas will be discussed throughout the text. One important principle is to “Test as you go” as contrasted with the rookie programmer’s approach: “write the whole thing, then pray it compiles!” Accordingly, this book will soon begin the creation of the final product, adding a little more as each chapter is completed, keeping it well tested every step along the way.

1: Languages, Grammars, and Machines

This chapter provides a brief introduction to language and computational theory. It presents only what is needed to successfully complete the *Simple Compiler*. Since a compiler translates one language to another, it is necessary to understand how to define a language, how to generate the legal strings of a language using grammars, and how to test a string to see if it belongs to a language using machines.

Languages

The most essential ingredient of a language is its **alphabet**. An alphabet is a set of characters, or symbols. Some examples of alphabets include the digits 0 through 9, the uppercase letters, or the set of all letters and digits, parentheses, and arithmetic operators.

Characters of an alphabet may be sequenced into **strings**. Given the alphabet of digits, example strings include “342,” “00,” and “4301520248920475289056.”

An alphabet can be sequenced into infinitely many strings. A **language** over an alphabet is some subset of all possible strings, also referred to as the **legal** strings.

Languages that compilers use are called **formal languages**. Formal languages differ from **natural languages** such as Italian and Chinese. The legal strings of the English language consist of all correct sentences (subject, verb, complete idea), but the meaning of “correct” is extremely difficult to specify. On the other hand, the structure of a formal language must be precisely definable. The rest of this book deals only with formal languages.

There are several ways to define a formal language. All involve specifying the legal combinations, or strings, of its alphabet. One way to do this is to explicitly list the set of legal strings – if there are only finitely many of them. Or a language may be described. For example, given the alphabet of the digits 0 through 9, the language of all counting numbers could be described as “a non-zero digit followed by zero or more of any of the digits.”

Regular expressions are another way to define a language. Regular expressions are a system for defining patterns using symbols and operators, and are used commonly in Computer Science. The language of counting numbers could be defined using the regular expression: PD^*, where the symbol “P” represents any positive digit (excluding only zero), and “D” represents any digit (including zero). The asterisk operator “*” means “zero or more of these.” So, the language PD^* is read: “a non-zero digit followed by zero or more digits.”

Note that the language of all counting numbers is a proper subset of all strings that could be created with the digit alphabet, as it excludes all strings starting with the character “0.”

Other operators used in regular expressions include “$^{+}$” which means “one or more of these”; “^{n}” which means “exactly n of these”; “|” which means “or”; and parentheses, which are used to group characters. For example, given the alphabet of lowercase letters, the regular expression: $(ab)^2d^+(e|fg)$ defines the language of all strings starting with “ab” exactly twice, followed by “d” one or more times, and ending with “e” or “fg.” Such a language includes the string “ababdddfg” (and many others), but not other strings such as “adeg.” Note that in this regular expression, a, b, d, e, f, and g are actual characters of the alphabet, whereas in the previous example, P and D were symbols representing groups of characters in an alphabet.

Grammars

A **Grammar** is a tool that generates the legal strings of a language. A particular grammar generates the legal strings of a particular language, so we have a one-to-one correspondence between grammars and languages.

Grammars consists of four parts. The first is a set of **terminals**, essentially the alphabet over which the language is defined. The second part is a set of **nonterminals**, which can be thought of as variables, eventually to be transformed into terminals. The third is a **start symbol**, taken from the set of nonterminals. And the fourth part is a set of **production rules**, describing how nonterminals can be replaced with terminals or other nonterminals.

Let us construct a grammar corresponding to the language of all counting numbers. The “or” notation using the “|” symbol allows us to combine several rules on one line.

Grammar: All counting numbers
1. terminals: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
2. nonterminals: S, P, N, D
3. start symbol: S
4. production rules:

   $S \rightarrow P \mid PN$
   $P \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
   $N \rightarrow ND \mid D$
   $D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

If the grammar is correct, then any terminal string it generates is a legal string of the corresponding language, and all legal strings of the language can be generated using the various choices of the grammar’s production rules.

This grammar is tested by first using it to generate various strings, and determine whether they are, indeed, counting numbers. Generating one legal string consists of beginning with the start symbol, then using the production rules to replace it with other nonterminals and terminals, continuing to replace the nonterminals until every remaining symbol is a terminal. It follows that the resulting terminal sequence should be a legal string of the language corresponding to the grammar.

An example of deriving a legal string using the above grammar follows: Starting with the sequence consisting of only the start symbol "S," there are two possible choices for what the symbol "S" can convert into – "P" or "PN". We arbitrarily choose to replace symbol "S" with the symbols "PN", resulting in the sequence "PN." Further, replacing the "P" with "7" results in "7N." Replacing "N" with "ND" yields "7ND." Again, replacing "N" with "ND" results in "7NDD." Replacing "N" with "D" produces "7DDD." For the final three steps, replacing the "D"s with "2," "3," and "4" respectively yields "7234." This sequence consists of all terminals. Game over. All the production steps taken together are called a **derivation**.

Since "7234" is a legal member of all counting numbers, it appears to be working. Following the same procedure, but choosing different production rules wherever choices exist, we may derive many strings of varying lengths, all of which seem to be legal counting numbers.

Let us now test whether the grammar matches the language by proceeding in the opposite direction. That is, can we find a grammar derivation that generates any legal string of the language? Let's try to find a derivation for "255."

Starting with S, we must choose the correct rules at each step of the derivation: S → PN → PND → PDD → 2DD → 25D → 255. Through more experimentation, we convince ourselves that a grammar derivation for all legal counting numbers exists. Furthermore, for non-counting numbers such as "0" or "073," no grammar derivation exists.

Machines

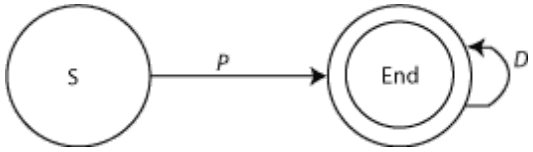
It is sometimes a tedious process to determine whether a particular string is legal using a grammar. Grammars are great for *generating* legal strings but are inadequate for *testing* them. It may be that a particular string is legal, but given all the grammar derivation choices, we simply cannot find a combination that correctly leads to our string, so we mistakenly declare the string illegal. Where a grammar is a "string generator," a **machine** is a "string tester."

The simplest type of machine is called a **finite automata**. A finite automata consists of a set of states and a set of rules for moving between the states. This concept can be represented using a directed graph. Recall that a graph consists of a set of vertices and a set of edges that connect the vertices. The vertices of our graph are the states. One of the states is assigned as the start state, and

one or more are designated end states. The edges of a finite automata graph are labeled with characters of the alphabet, and thus dictate which characters transition the machine from state to state. An edge may be labeled with a particular character of the alphabet, or with several characters separated by commas, or with a symbol which represents a set of characters.

The finite automata machine also maintains a current state, which is initially the start state. It then reads each character of the string to be tested, one at a time. If there is an edge labeled with that character leading from the current state to another, then the current state of the machine switches to that destination state. If there is no edge from the current state with the correct label, the machine rejects that input string. There must not be multiple qualifying edges. If the machine can process through all characters in the input string, and the final state of the machine is one of the end states, then the string is accepted.

The following diagram represents a finite automata machine for testing strings to see if they are in the language of counting numbers. The diagram is simplified by letting "P" represent the digits 1 through 9, and "D" represents the digits 0 through 9. States with double circles are the end states.



Let us test the machine by feeding various strings into it. The string "05" is rejected immediately because the machine has no edge with a label of "0" leaving the start state S. The string "255" is processed first by reading the "2" which switches the current state to End, then reading the "5" which switches the current state to End (itself), then reading the "5" which switches the current state to End again. The input string being exhausted, the machine checks whether the current state is a final state. It is, so the string is accepted.

Example: Even Numbers

In this section, we define the language of even numbers using an English description and a regular expression. Then we create a grammar corresponding to this language and use it to create an arbitrary legal string. We then use the grammar to derive a particular legal string. Finally, we create a finite automata for this language and use it to test whether selected strings represent even numbers.

The language of even numbers is defined over the alphabet of digits. All strings of digits are legal as long as they don't start with "0" and don't end with "1," "3," "5," "7," or "9," except the string "0," which is also legal. Even numbers can also be defined by the regular expression: $E|PE|PD^{+}E$.

A grammar that exactly generates strings belonging to the language of even numbers is as follows:

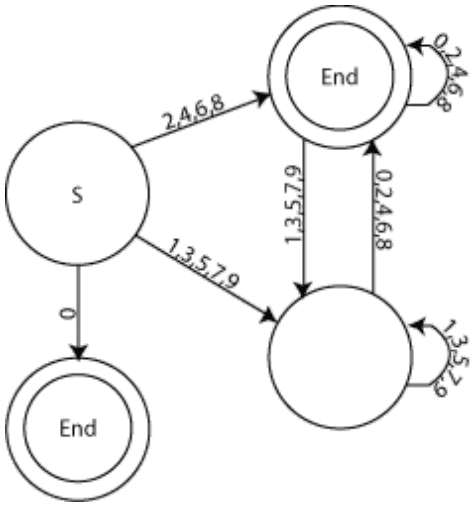
Grammar: Even Numbers
1. terminals: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
2. nonterminals: S, P, N, D, E
3. start symbol: S
4. production rules:

$S \rightarrow E \mid PE \mid PNE$
$E \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8$
$P \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$N \rightarrow ND \mid D$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

This grammar generates an arbitrary even number as follows: Starting with the sequence "S," there are three choices. Replacing "S" with "PNE" results in the sequence "PNE." Replacing the "P" with "7" results in "7NE." Replacing the "N" with "ND," produces "7NDE." Replacing the "N" with "D" results in "7DDE." In the final three steps, we replace the "D"s with "2" and "3," and the "E" with "4," yielding "7234." The resulting sequence consists of all terminals and is a legal string.

We now attempt to find a derivation for "256," which is a legal string.
Starting with "S", we must choose the correct rules at each step of the derivation:
$S \rightarrow PNE \rightarrow PDE \rightarrow 2DE \rightarrow 25E \rightarrow 256$. Observe that it is not possible to derive odd numbers, such as "255."

The following diagram represents a finite automata machine for testing strings to determine whether they are legal even numbers.



Let us test the machine with selected strings. The string "0" moves the current state to an end state, and the string is accepted. The string "05" moves the

current state to an end state, but then cannot process the “5,” so the string is rejected. All strings that do not begin with a “0” fluctuate between the odd and the even states. If the final input character is even, the string is accepted.

More Advanced Machines

Finite automata machines are quite limited in power. Consider the language of all strings starting with a certain number of “a’s” and ending with exactly the same number of “b’s”. This language could also be expressed as “$a^n b^n$,” where “n” is any integer greater than or equal to 1. Finite automata must be built using a finite number of states. We could create a finite automata with 200 states that could successfully count up to 100 “a’s"; then count back that same number of “b’s”, returning to the start state, which is also the end state. But since “n” can be any number (e.g. 101), no finite automata machine can correctly test all strings of that language. A more advanced machine is required.

Push-down automata machines are like finite automata machines, except they also contain an infinite stack. After reading an input character and switching states, the push-down automata can also push or pop characters from its stack. The language “$a^n b^n$” can easily be modeled with a push-down automata as follows: Every “a” gets pushed onto the stack, and every “b” pops an “a” back off the stack. After reading the first “b,” the state switches to one that accepts only more “b’s”. If the input string is successfully exhausted (meaning all input characters resulted in a legal move, and no pops were made on an empty stack), the machine accepts the string if the resulting state is an end state and the stack is empty.

Push-down automata machines are also limited in power. For example, no push-down machine can successfully recognize all strings of the form “$a^n b^n c^n$.” However, if we replace the infinite stack with an infinite tape (which can move left a square, move right a square, write to a square, and read from a square), then such a machine can recognize all strings of the form “$a^n b^n c^n$.” Such a machine is called a Turing machine, invented by the famous British mathematician Alan Turing.

The Church-Turing thesis states that any problem that can be solved by any computing machine can also be solved by a Turing machine. While this hypothesis has not been proven, it is widely accepted as true. Thus, the Turing machine is much more than a language-testing machine in that it can also be used to do any computation currently existing computers can do. In fact, the theoretical Turing machine is more powerful than any real machine, since it has an infinite tape.

Are there problems that cannot be solved by a Turing machine? Yes, there exist a whole class of problems called **undecidables,** which cannot be solved at all. One famous example is called the Halting Problem. The input to the Halting Problem is a Turing machine and a string of input characters. The output of the

problem is supposed to answer the question of whether the given Turing machine will ever halt on the given input string. It has been proven that the Halting Problem cannot be solved by any machine.

Machine Categories

Machines can be categorized by their level of difficulty. There are surprisingly few levels of difficulty—namely, finite automata, push-down automata, and Turing machines. Some textbooks divide Turing machines into two categories, the less powerful being called linear-bounded automata, but we will not make this distinction.

Any language that can be recognized at all can be recognized by a Turing machine. If a Turing machine is not required, a push-down automata can do the job. If a push-down automata isn't required, a finite automata is sufficient. Stated another way, a problem that cannot be solved with a finite automata requires at least a push-down automata; there is no machine greater than a finite automata and less than a push-down automata. And any problem that cannot be solved with a push-down automata necessarily requires a Turing machine, if it can be solved at all.

The fact that there are only three levels of machines is quite remarkable. Similar conclusions are found with respect to languages and grammars—each have exactly three levels. As one might guess, every regular language can be generated by a regular grammar and tested using a finite automata. Every context-free language can be generated with a context-free grammar and tested with a push-down automata. And every unrestricted language can be generated with an unrestricted grammar and tested using a Turing machine. The following table shows this correspondence.

Level	Language	Grammar	Machine
1	Unrestricted	Unrestricted	Turing Machine
2	Context Free	Context Free	Push Down Automata
3	Regular	Regular	Finite Automata

Grammar Categories

The three categories of grammars are differentiated according to the production rules they allow. The simplest class of grammars is called **regular grammars**. The rules of regular grammars must be of a very restricted form. Specifically, the left side of a rule must contain exactly one nonterminal. The right side must contain one terminal, or one terminal followed by one nonterminal. Additionally, there may be a rule from the start symbol going to nothing.

A second class of grammars is called **context-free grammars**. They maintain the restriction that the left side must contain exactly one nonterminal, but the right side may contain any combination of terminals and nonterminals, or nothing.

The final class of grammars is called **unrestricted grammars**. The left side of a rule needs at least one nonterminal, but may contain more nonterminals and terminals. The right side may contain any combination of terminals and nonterminals, or nothing. As with machines, some textbooks divide unrestricted grammars into two categories, the less powerful being called **context sensitive grammars**.

Every level-3 grammar is also a level-2 grammar, and every level-2 grammar is also a level 1.

Consider the grammar rules we used earlier to generate strings of counting numbers:

S → P | PN
P → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
N → ND | D
D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The rules S→P, S→PN, and N→ND all seem to violate the regular grammar restrictions, so one might conclude that the machine, the grammar, and the language of counting numbers are not of the simplest flavor, level 3. However, we have already seen that counting numbers can be tested with a finite automata machine, so there must be an equivalent regular grammar. Just because we can create a grammar beyond the specifications of a regular grammar does not necessarily mean it must be of a higher difficulty level. Remember that the three categories are subsets of each other. We could create a Turing machine to recognize counting numbers, but the problem is still only of the simplest level.

Indeed, the following grammar rules also generate all strings of counting numbers, yet follows the restrictions of regular grammars:

S → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
S → 1N | 2N | 3N | 4N | 5N | 6N | 7N | 8N | 9N
N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
N → 0N | 1N | 2N | 3N | 4N | 5N | 6N | 7N | 8N | 9N

We shall see in the next chapters that a Scanner requires only regular grammars, whereas a Parser requires context-free grammars. We will not use unrestricted grammars beyond this chapter.

Problems

1. For each of these regular languages, give an equivalent grammar. Then draw the corresponding finite automata. For simplicity, you are welcome to use the Context Free type grammar rules, even though they could all be done with only regular grammars.

   A. All strings of lowercase letters that contain at least two “a”s.
   B. All strings of lowercase letters that start with “a” and end with “b.”
   C. All strings of lowercase letters that do not contain an “a.”
   D. All strings of digits that have two adjacent identical digits.
   For example: “83662” is legal because there are two adjacent ‘6’ digits.
   (You may simplify the alphabet to just the digits “1”, “2”, and “3.”)
   E. All strings of digits representing integers between 500 and 799.
   F. All strings of lowercase letters starting with one or more “a”s, followed by one or more “b”s, followed by one or more “c”s, followed by one or more “d”s, then nothing.
   G. All strings of lowercase letters that start with “abc” followed by any sequence of letters containing at least two “d”s, and ending with “efg.”
   H. Labels (all strings of lowercase letters, digits, and underbars that start with a lowercase letter and are optionally followed by any combination of lowercase letters, digits, and underbars).
   I. Non-negative Real Numbers (all strings of digits and periods that have one or more digits followed by a period followed by one or more digits). Leading and trailing zeros are allowed, just as C++ allows them.
   J. Non-negative Extended Real Numbers (Add to the previous problem strings that start with a period followed by one or more digits, and strings that start with one or more digits followed by a period, and strings that are one or more digits without a period)

2. Use your finite automata machines from the previous problem to test whether these strings are legal. If they are, provide a derivation using your grammar.

   A. “bcaad,” “abcda,” “abcde”
   B. “cab,” “ab,” “abcd,” “acdb”
   C. “abd,” bcd”
   D. “11231,” “12322,” “12313,” “12331”
   E. “500,” “5000,” “70,” “356,” “756”
   F. “abcd,” “abcbd,” “abbcccd”
   G. “abcmndxydkefg,” “abcdefgdefg”
   H. “cost_2,” “2cost”
   I. “13.0”, “1.2.3“, “13”, “13.”, “.13”
   J. “13.0”, “1.2.3”, “13”, “13.”, “.13”

3. For each of these context-free languages, create an equivalent context-free grammar.

A. Palindromes (all strings of lowercase letters which read the same forwards and backwards) *“madamimadam” and “eve” were, incidentally, the first words spoken in the Garden of Eden.*

B. All strings starting with one or more “a”s, followed by one or more “b”s, followed by one or more “c”s, followed by one or more “d”s. The number of “a”s and “c”s must be the same.

4. For each grammar, use good English to fully describe the corresponding language. Then classify the underlying language as regular, context free, or unrestricted. Remember that just because a grammar uses Context Free rules doesn’t mean it has to. If a finite automata can test the language, then it is only regular.

A. Grammar: Describe the language? What is the level?

1. terminals: 0, 1
2. nonterminals: S, Z
3. start symbol: S
4. production rules:

$S \rightarrow 1 \mid Z \mid 1S \mid ZS$

$Z \rightarrow 10$

B. Grammar: Describe the language? What is the level?

1. terminals: 0, 1
2. nonterminals: S, Z
3. start symbol: S
4. production rules:

$S \rightarrow 1SZ \mid$ (nothing)

$Z \rightarrow 00$

Extra Mile 1: For the language “$a^n b^n c^n$,” no context-free grammar exists. Find an unrestricted grammar that exactly generates all strings of that language. Assume n >= 1. Show a derivation for “abc,” “aabbcc,” and “aaabbbccc.”

Extra Mile 2: Find a context-free grammar for the language of strings with the same number of 0s and 1s. Show a derivation for “0011,” “1100,” and “101001.”

2: The Scanner

The Scanner module is responsible for converting the source language from a stream of characters into a stream of **tokens**. These tokens are subsequently processed by the Parser module, which is covered in a later chapter.

A token is a group of characters that should be processed as a unit. A token has two parts--the type and the lexeme. For our compiler, the token types might consist of something like the following:

Token types for reserved words:
VOID, MAIN, FOR, IF, WHILE, INT, COUT, etc.
Token types for relational operators:
LESS, LESSEQUAL, GREATER, GREATEREQUAL, EQUAL, NOTEQUAL
Token types for other operators:
INSERTION, ASSIGNMENT, PLUS, MINUS, TIMES, DIVIDE
Token types for symbols:
SEMICOLON, LPAREN, RPAREN, LCURLY, RCURLY
Other token types:
IDENTIFIER, INTEGER, ENDFILE

The token lexemes are the actual characters taken from the input stream. Consider this small sample of C code:

```
void main()
{
	int sum;
	sum = 35 + 400;
	cout << sum;
}
```

A scanner's job is to take this set of characters as input and produce as output the following set of tokens:

Token Type	Lexeme
VOID	void
MAIN	main
LPAREN	(
RPAREN	)
LCURLY	{
INT	int
IDENTIFIER	sum
SEMICOLON	;
IDENTIFIER	sum
ASSIGNMENT	=
INTEGER	35
PLUS	+
INTEGER	400
SEMICOLON	;
COUT	cout
INSERTION	<<

IDENTIFIER	sum
SEMICOLON	;
RCURLY	}
ENDFILE	

Note that we are treating “main” and “cout” as reserved words, whereas a more sophisticated compiler would treat them as an identifier for a function and an object. Note also that we are having the main function return “void” for simplicity, whereas officially the main function should always return an integer.

Recognizing Tokens with a Finite Automata

The previous chapter’s description of a finite automata machine states that if no edge exists for an input character, the machine should reject the string. However, our problem is complicated because we must process one huge input file consisting of many legal strings (tokens), and it is not necessarily clear where one string ends and the next begins. Therefore, we must alter the finite automata as follows: If there is no edge to handle the newest character, then the current state of the machine is what’s important. If it’s an end state, every character up to that point is a legal string, and the machine should reserve the newest character as the first in the next string. The machine resets to the start state. If at any point a character cannot be processed and the current state is not an end state, the whole input file is rejected.

Creating C Code from a Finite Automata

There are well defined, systematic processes for converting finite automata and regular grammars into working C code. The UNIX tool LEX takes as input a regular grammar and produces corresponding C code as output. We’ll write our own scanner and State Machine simulator to handle the job.

Subset Strings

Sometimes a legal string is a subset of another legal string. For example, “<” is a legal string by itself, but can also legally extend to “<=” or “<<”. That is why every input character should be read until a character is reached that cannot be processed.

Reserved Words

Reserved words follow the same language rules as identifiers, so it is convenient to first recognize all reserved words as if they were identifiers. Afterwards, we check the token’s lexeme to see if it should be replaced with a reserved word token type.

Problems

1. Draw a finite automata machine that can tokenize the following input:

```
void main()
{
	int sum;
	sum = 35 + 400;
	cout << sum;
}
```

2. Begin the creation of your Simple Compiler. Create a new project. Add Token.h and Token.cpp. In Token.h, declare an enum called "TokenType" for the token types listed in the chapter. It should look something like this:

```
enum TokenType {
	// Reserved Words:
	VOID_TOKEN, MAIN_TOKEN, INT_TOKEN, COUT_TOKEN,
	// Relational Operators:
	LESS_TOKEN, LESSEQUAL_TOKEN, GREATER_TOKEN, GREATEREQUAL_TOKEN,
	EQUAL_TOKEN, NOTEQUAL_TOKEN,
	// Other Operators:
	INSERTION_TOKEN, ASSIGNMENT_TOKEN, PLUS_TOKEN, MINUS_TOKEN,
	TIMES_TOKEN, DIVIDE_TOKEN,
	// Other Characters:
	SEMICOLON_TOKEN, LPAREN_TOKEN, RPAREN_TOKEN, LCURLY_TOKEN,
	RCURLY_TOKEN,
	// Other Token Types:
	IDENTIFIER_TOKEN, INTEGER_TOKEN,
	BAD_TOKEN, ENDFILE_TOKEN
	};
```

The above enumeration assigns all the TokenTypes to integers. But it is also useful to have names for them, so declare an array of strings like this:

```
// IMPORTANT: The list above and the list below MUST be kept in sync.
const std::string gTokenTypeNames[] = {
	"VOID", "MAIN", "INT", "COUT",
	"LESS", "LESSEQUAL", "GREATER", "GREATEREQUAL", "EQUAL",
	"NOTEQUAL",
	"INSERTION", "ASSIGNMENT", "PLUS", "MINUS", "TIMES", "DIVIDE",
	"SEMICOLON", "LPAREN", "RPAREN", "LCURLY", "RCURLY",
	"IDENTIFIER", "INTEGER",
	"BAD", "ENDFILE"
	};
```

Then you can access the name given the enum using: `gTokenTypeNames[mType]`. Note that you'll need to `#include <string>`.

Next, declare a class called "TokenClass", where each token consists of a TokenType called "mType" and a string called "mLexeme". Note that all classes of your Simple Compiler project should use private, protected, and public appropriately, and provide accessor and constructor methods as needed. We'll start all class variables with the lower case 'm', according to the Hungarian notation standard. Your class declaration might look something like this:

```
class TokenClass
{
private:
	TokenType mType;
	std::string mLexeme;
public:
	TokenClass(TokenType type, const std::string& lexeme);
	TokenType GetTokenType() const { return mType; }
	const std::string& GetTokenTypeName() const
	{
		return gTokenTypeNames[mType];
	}
	std::string GetLexeme() const { return mLexeme; }
	static const std::string& GetTokenTypeName(TokenType type)
	{
		return gTokenTypeNames[type];
	}
};
```

Override the insertion stream operator for TokenClass objects, so it outputs the token type, the token name, and the lexeme. This will be useful for testing and debugging. This prototype should go in the file Token.h, after the TokenClass declaration:

```
std::ostream & operator<<(std::ostream & out, const TokenClass & tc);
```

Put the corresponding implementation code in Token.cpp.

Include whatever header files you might need to make it compile, such as <iostream>. At the beginning of Token.h, protect against multiple inclusion with:

```
#pragma once
```

Remember to put similar protection in all your header files that you will create for this project.

Complete Token.cpp by providing an implementation for the TokenClass constructor. After initializing the two class variables, the constructor should change the TokenType from IDENTIFIER_TOKEN, if mLexeme matches one of the reserved words. For example, if mLexeme is “void” then reset mType to VOID_TOKEN. See the list of Reserved Word tokens above and be sure to keep this list up to date as your compiler expands. For now, there are only 4 reserved words you need to check for.

```
	// Check for reserved words:
	if (lexeme == "void")
		mType = VOID_TOKEN;
	else if (lexeme == "main")
		mType = MAIN_TOKEN;
	else if (lexeme == "int")
```

```
            mType = INT_TOKEN;
      else if (lexeme == "cout")
            mType = COUT_TOKEN;
```

Add another file to your project called Main.cpp. Create a main() function to test the code you have written so far. For example, you might test:

```
int main()
{
      TokenType tt = VOID_TOKEN;
      std::string lexeme = "void";
      TokenClass tok1(tt, lexeme);
      std::cout << tok1 << std::endl;

      return 0;
}
```

3. In this step, you will create a class called StateMachineClass that simulates the finite automata machine you drew in problem one. Add StateMachine.h and StateMachine.cpp to your compiler project. In StateMachine.h, enumerate all possible MachineStates your finite automata might be in. Note that as your compiler evolves, this enumeration must be kept in sync with your finite automata picture of problem one. Create another enum for all types of input characters. Note that the enum of input CharacterTypes is smaller than the alphabet, because characters that behave the same (such as letters) should be grouped together into one type.

```
enum MachineState {
      START_STATE, IDENTIFIER_STATE, INTEGER_STATE, CANTMOVE_STATE,
      (etc)
      LAST_STATE
      };

enum CharacterType {
      LETTER_CHAR, DIGIT_CHAR, WHITESPACE_CHAR, PLUS_CHAR, BAD_CHAR,
      (etc)
      LAST_CHAR
      };
```

A programming trick with enumerations is to make the last one be a special value which represents the total count. For example, in the previous code, LAST_STATE is assigned the constant value of how many other MachineStates there are. That constant can be used in C++ code to declare arrays of the correct size, or to loop through arrays the correct number of times. As more states are added to the enum, LAST_STATE automatically adjusts itself to the new count, as long as you keep it at the end of the list. Also note in the code the special state CANT_MOVE, which isn't really a finite automata machine state, but is the value representing that there is no legal state to move to, given the current state and input character.

Next in StateMachine.h, declare the StateMachineClass. It needs two public methods – a constructor and UpdateState. It can be implemented with three private variables. mCurrentState, mLegalMoves, and mCorrespondingTokenTypes.

```
class StateMachineClass
{
public:
	StateMachineClass();
	MachineState UpdateState(char currentCharacter, TokenType &
			previousTokenType);

private:
	MachineState mCurrentState;

	// The matrix of legal moves:
	MachineState mLegalMoves[LAST_STATE][LAST_CHAR];

	// Which end-machine-states correspond to which token types.
	// (non end states correspond to the BAD_TOKEN token type)
	TokenType mCorrespondingTokenTypes[LAST_STATE];
};
```

The constructor should initialize mCurrentState to START_STATE.

Next, the constructor needs to initialize all values of mLegalMoves. These correspond one-to-one with the finite automata you made in problem 1. For example, if your automata has an edge labeled DIGITS from START_STATE to INTEGER_STATE, then you would write this line of code:

```
mLegalMoves[START_STATE][DIGIT_CHAR]= INTEGER_STATE;
```

Since most of the State and Char combinations don't have edges, first initialize the whole two-dimensional array to CANTMOVE_STATE with this code:

```
	// First, initialize all the mLegalMoves to CANTMOVE_STATE
	for(int i=0; i<LAST_STATE; i++)
	{
		for(int j=0; j<LAST_CHAR; j++)
		{
			mLegalMoves[i][j] = CANTMOVE_STATE;
		}
	}
```

Then overwrite with all the edges in your finite automata:

```
	// Then, reset the mLegalMoves that are legitimate
	mLegalMoves[START_STATE][DIGIT_CHAR]= INTEGER_STATE;
	mLegalMoves[INTEGER_STATE][DIGIT_CHAR]= INTEGER_STATE;
	(etc)
```

Finally, the constructor needs to initialize all the values of mCorrespondingTokenTypes. Start by initializing them all to BAD_TOKEN:

```
	// First, initialize all states to correspond to BAD_TOKEN.
	for(int i=0; i<LAST_STATE; i++)
	{
		mCorrespondingTokenTypes[i]=BAD_TOKEN;
	}
```

Then override them with correct values. That is, for each Good ending state, indicate the TokenType that should be created when the finite automata ends (CANT_MOVE) in that state.

```
	// Then, reset end states to correspond to correct token types.
	mCorrespondingTokenTypes[IDENTIFIER_STATE] = IDENTIFIER_TOKEN;
	mCorrespondingTokenTypes[INTEGER_STATE] = INTEGER_TOKEN;
	(etc)
```

UpdateState should change mCurrentState, based on its current value and the input character. Eventually, UpdateState will return CANTMOVE_STATE, which signifies to the calling code that the current token is completed. The reference parameter previousTokenType should be set by UpdateState, so that the calling code knows what kind of TokenType it should make. If the current state of the machine is not a valid end state, then previousTokenType will get set to BAD_TOKEN, so the calling code knows to print an error message and quit.

The UpdateState method is implemented by first converting the input character into a CharacterType (perhaps with a large if-else chain):

```
MachineState StateMachineClass::UpdateState(char currentCharacter,
		TokenType & previousTokenType)
{
	// convert the input character into an input character type
	CharacterType charType = BAD_CHAR;

	if(isdigit(currentCharacter))
		charType = DIGIT_CHAR;
	else if(isalpha(currentCharacter))
		charType = LETTER_CHAR;
	else if(isspace(currentCharacter))
		charType = WHITESPACE_CHAR;
	else if(currentCharacter=='+')
		charType = PLUS_CHAR;
	else if(currentCharacter==EOF)
		charType = ENDFILE_CHAR;
	(etc)
```

Then index the current state and character type into the mLegalMoves array to find the next state, which is the return value. But a little bookkeeping is required. Right before updating mCurrentState, set the previousTokenType reference parameter as follows:

```
	previousTokenType = mCorrespondingTokenTypes[mCurrentState];
	mCurrentState = mLegalMoves[mCurrentState][charType];
```

```
	return mCurrentState;
}
```

That allows the calling code to know what type of a token to make (previousTokenType), when UpdateState returns CANTMOVE_STATE.

4. Add Scanner.h and Scanner.cpp to your compiler project. Declare a class called ScannerClass consisting of an ifstream called "mFin". Declare a constructor method with a string input parameter. The implementation of the constructor should open the file of the input string and attach it to mFin. To avoid future problems, open the file in binary mode, not text mode, as in:

```
mFin.open(inputFileName.c_str(), std::ios::binary);
```

If the input file doesn't open, print an error message and quit:

```
	if (!mFin)
	{
		std::cerr << "Error opening input file " << inputFileName;
		std::exit(1);
	}
```

A corresponding destructor should close mFin.

Add a method called "GetNextToken" that has no input parameters and returns a variable of type TokenClass. The calling code will eventually be the Parser module, but for this chapter it will be the main() function (problem 5 below).

Start GetNextToken by declaring its needed local variables:

```
	StateMachineClass stateMachine;
	std::string lexeme;
	MachineState currentState;
	TokenType previousTokenType;
```

Using a do-while loop, read characters from mFin and pass them to stateMachine.UpdateState, until the return value is CANTMOVE_STATE. You will need to store all read characters in the lexeme string so you can later make a TokenClass object out of them. Also, whenever the currentState gets to START_STATE or ENDFILE_STATE, reset the lexeme, because whitespace characters should not be included in any lexeme.

```
do
{
	char c = mFin.get();
	lexeme += c;
	currentState = stateMachine.UpdateState(c, previousTokenType);
	if (currentState == START_STATE || currentState == ENDFILE_STATE)
		lexeme = "";
} while (currentState != CANTMOVE_STATE);
```

When the do-while loop completes, first check if previousTokenType is BAD_TOKEN, in which case you should print a meaningful error message and quit.

```
if (previousTokenType == BAD_TOKEN)
{
	std::cerr << "Error. BAD_TOKEN from lexeme " << lexeme;
	std::exit(1);
}
```

Note that the last character you read caused the machine to CANT_MOVE. That character is not part of the current lexeme, and needs to be put back into the input stream mFin so that it can properly be read as part of the next token.

```
lexeme.pop_back();
mFin.unget();
```

You are now ready to create a TokenClass from all the letters you just read (lexeme) and from the TokenType associated with the MachineState it was in right before it couldn't move any further (previousTokenType). Then return that TokenClass.

```
TokenClass tc(previousTokenType, lexeme);
return tc;
```

5. Add Main.cpp to your project if you haven't already. For now, it should include the appropriate header files, then create a ScannerClass object called "scanner", initialized with the name of a source file containing code like that of problem 1. It should then call scanner.GetNextToken() over and over, printing each token as it goes, until it gets a token of type ENDFILE_TOKEN. Test your code. Make sure it handles error cases such as BAD_TOKEN and nonexistent input file.

```
ScannerClass scanner("code.txt");
TokenType tt;
do
{
	TokenClass tc = scanner.GetNextToken();
	std::cout << tc << std::endl;
	tt = tc.GetTokenType();
} while (tt != ENDFILE_TOKEN);
```

6. Add Debug.h to your project. Create a #define ShowMessages, which can be set to either 0 or 1. This is used as a switch to control a macro called MSG(X). If ShowMessages is 1, #define MSG(X) to "std::cout << X << std::endl". Otherwise, MSG(X) should map to nothing. All of these #define switches and macros are handled by the C++ preprocessor. To make use of this functionality, annotate all important places in your code with MSG messages. For example, at the beginning of your scanner's constructor, put MSG("Initializing ScannerClass object");. Later, when your code isn't working and things seem hopelessly lost, set ShowMessages to 1, and see exactly the order in which all your code is being

executed. When your code is working, such debug messages can easily be turned back off to avoid screen clutter. Test your code.

7. Enhance your ScannerClass with code to keep track of the current line number. This is useful for testing and debugging. Have a class variable called mLineNumber. The constructor should set this to 1. Every time GetNextToken processes a return, it should increment mLineNumber. Add an accessor method called GetLineNumber. Test your code by having your main loop print line numbers, as well as token types and lexemes.

Warning: If the return character \n caused the CANT_MOVE condition then it will get put back in the input stream and be double counted when it gets read the next time. Be sure to account for this problem.

8. Enhance the finite automata of problem 1 and your code to respect the block comment markers /* and */. Any characters between those markers should be ignored, except for the incrementing of line numbers. Do NOT introduce comments as a new token type. Your finite automata should pass over them and continue from the Start State, just as it does for white space. Test your code.

9. Enhance the finite automata of problem 1 and your code to respect the line comment marker of //. Any characters between that and the end of the line should be ignored. Test your code. Verify that combinations of block comments and line comments work properly.

Hints: You will need to separate WHITESPACE_CHAR into two categories, RETURN_CHAR and WHITESPACE_CHAR. Update your finite automata and your code accordingly. Also, a line comment doesn't have to end with a return. It can also be fulfilled with an ENDFILE_CHAR.

10. Enhance the finite automata of problem 1 and your code to support the rest of the token types listed in problem 2. Test your code. Verify that all token types are correctly identified.

3: The Symbol Table

The variables taken from the source code and stored in our Simple Compiler will each contain a label and a value. The collection of all the variables will be stored in one symbol table. The symbol table will contain methods for declaring variables, setting their values, getting their values, getting their index numbers, and determining their existence – all by the variable label.

Keep in mind that even though we are building the symbol table infrastructure now, and unit testing it, we will not actually be populating the symbol table with variables until a later chapter, when we start Interpreting.

Problems

1. Add Symbol.h and Symbol.cpp to your compiler project. Declare a class called SymbolTableClass which will later be used to store variables, both for interpreting and for compiling. Declare the following methods:

```
bool Exists(const std::string & s);
      // returns true if <s> is already in the symbol table.
void AddEntry(const std::string & s);
      // adds <s> to the symbol table,
      // or quits if it was already there
int GetValue(const std::string & s);
      // returns the current value of variable <s>, when
      // interpreting. Meaningless for Coding and Executing.
      // Prints a message and quits if variable s does not exist.
void SetValue(const std::string & s, int v);
      // sets variable <s> to the given value when interpreting.
      // Meaningless for Coding and Executing.
      // Prints a message and quits if variable s does not exist.
int GetIndex(const std::string & s);
      // returns a unique index of where variable <s> is.
      // returns -1 if variable <s> is not there.
size_t GetCount();
      // returns the current number of variables in the symbol
      // table.
```

2. Each variable consists of a name and a value, so declare a struct called Variable consisting of an mLabel of type string and an mValue of type integer. Declare struct Variable inside your SymbolTableClass so it will be invisible to all code except that containing class. Since a struct has public access by default, the methods of SymbolTableClass can easily modify the data items of Variable objects.

3. In SymbolTableClass, declare a standard template library (stl) vector of Variable types. Remember to always make all class variables private.

4. Implement all the methods of SymbolTableClass. Unit test your code. It is important that this module works before integrating it with your Node and Parser modules.

5. Handle various problems. For example, if GetValue() or SetValue() is given a variable name that does not exist, you should print an error message and quit. similarly handle the problem when AddEntry gets a variable name that already exists. Test your error handling code.

4: The Parse Tree

The output of the Parser module will be a Parse Tree. Before we can write the parser, we must first create the parse tree infrastructure. A parse tree is a tree of nodes, where each node and the relationships between the nodes summarizes information about the original source code. A parse tree is closely tied to a grammar, so we first build a simple grammar that can generate our sample source code (and other similar kinds of source code). We then show a derivation sequence for generating our sample source code using the grammar. We conclude the chapter by creating an equivalent parse tree.

Our first goal is to create a grammar that can generate strings such as our sample C code:

```
void main()
{
	int sum;
	sum = 35 + 400;
	cout << sum;
}
```

A grammar that generates strings belonging to our simplified C language follows. Terminals are listed in all capitals, and nonterminals are listed in angle brackets, with the first letter capitalized. Note that the terminals directly correspond to the token types.

Grammar: Simplified C

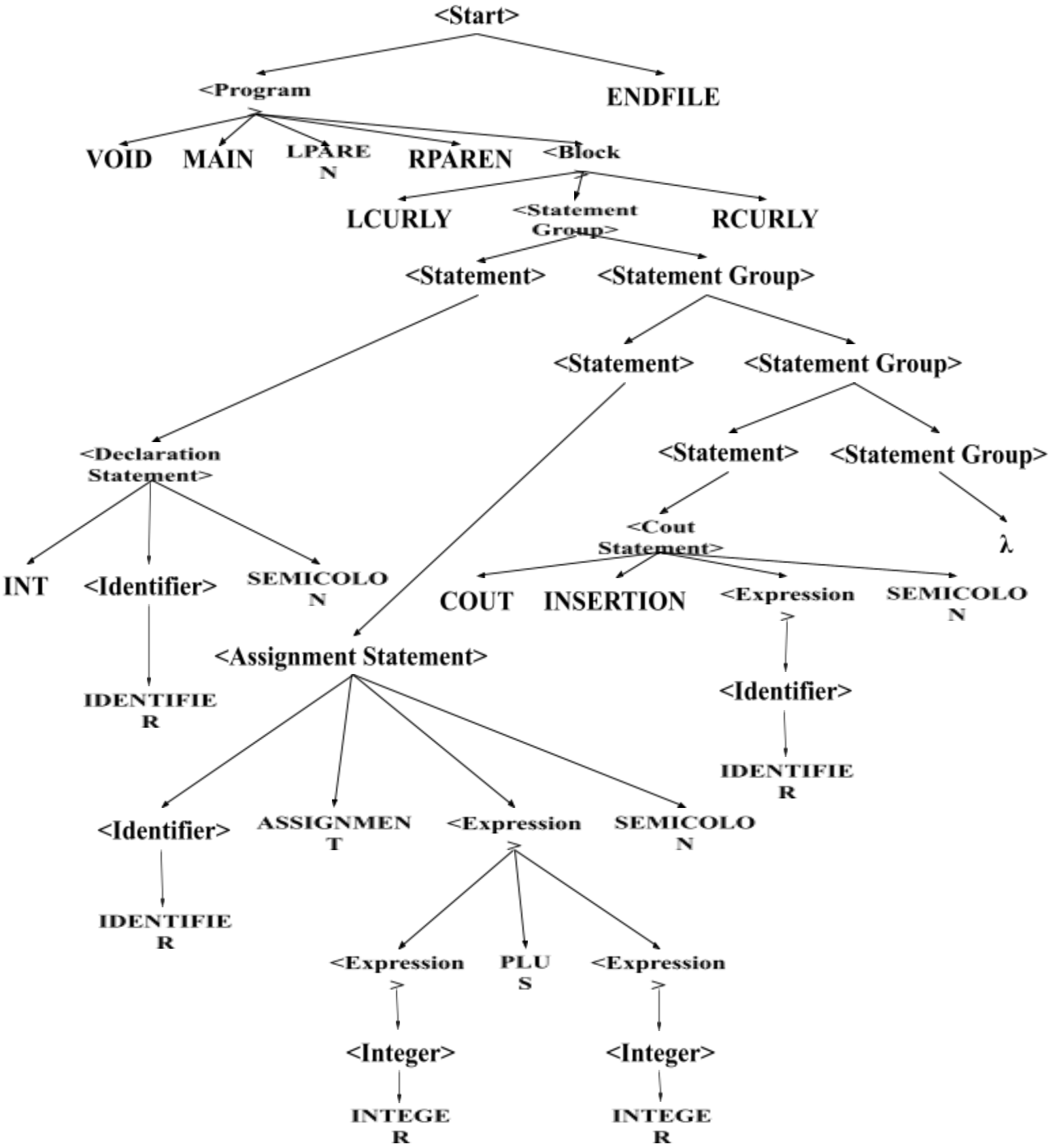
1. terminals: ENDFILE, VOID, MAIN, LPAREN, RPAREN, LCURLY, RCURLY, INT, SEMICOLON, ASSIGNMENT, COUT, INSERTION, PLUS, IDENTIFIER, INTEGER
2. nonterminals: <Start>, <Program>, <Block>, <StatementGroup>, <Statement> <DeclarationStatement>, <AssignmentStatement>, <CoutStatement>, <Expression>, <Identifier>, <Integer>
3. start symbol: <Start>
4. production rules:

<Start>	→ <Program> ENDFILE
<Program>	→ VOID MAIN LPAREN RPAREN <Block>
<Block>	→ LCURLY <StatementGroup> RCURLY
<StatementGroup>	→ <Statement> <StatementGroup>
<StatementGroup>	→ {empty}
<Statement>	→ <DeclarationStatement>
<Statement>	→ <AssignmentStatement>
<Statement>	→ <CoutStatement>
<DeclarationStatement>	→ INT <Identifier> SEMICOLON
<AssignmentStatement>	→ <Identifier> ASSIGNMENT <Expression> SEMICOLON
<CoutStatement>	→ COUT INSERTION <Expression> SEMICOLON
<Expression>	→ <Identifier>
<Expression>	→ <Integer>
<Expression>	→ <Expression> PLUS <Expression>
<Identifier>	→ IDENTIFIER
<Integer>	→ INTEGER

The following shows a derivation sequence for generating our sample of C code:
<Start> → <Program> ENDFILE
→ VOID MAIN LPAREN RPAREN <Block> ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY <StatementGroup> RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY <Statement> <StatementGroup> RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY <Statement> <Statement> <StatementGroup> RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY <Statement> <Statement> <Statement> RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY <DeclarationStatement> <AssignmentStatement> <CoutStatement> RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY INT <Identifier> SEMICOLON <Identifier> ASSIGNMENT <Expression> SEMICOLON COUT INSERTION <Expression> SEMICOLON RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY INT IDENTIFIER SEMICOLON IDENTIFIER ASSIGNMENT <Expression> PLUS <Expression> SEMICOLON COUT INSERTION <Expression> SEMICOLON RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY INT IDENTIFIER SEMICOLON IDENTIFIER ASSIGNMENT <Integer> PLUS <Integer> SEMICOLON COUT INSERTION <Identifier> SEMICOLON RCURLY ENDFILE
→ VOID MAIN LPAREN RPAREN LCURLY INT IDENTIFIER SEMICOLON IDENTIFIER ASSIGNMENT INTEGER PLUS INTEGER SEMICOLON COUT INSERTION IDENTIFIER SEMICOLON RCURLY ENDFILE

This derivation sequence ends with the same sequence of tokens that the scanner assignment from a previous chapter produced as output.

A derivation can alternately be shown using a parse tree. The root node of the tree is the start symbol. The children of the root node are the nonterminals and terminals that the root node transforms into. Likewise, each nonterminal node has children corresponding to the nonterminals and terminals that that nonterminal transforms into. The terminal nodes form the leaves of the tree. The final derivation consists of the terminal nodes, taken from left to right. The following figure illustrates the parse tree for our sample C code:


<Start>
<Program
ENDFILE
VOID
MAIN
LPAREN
RPAREN
<Block
LCURLY
<Statement Group>
RCURLY
<Statement>
<Statement Group>
<Statement>
<Statement Group>
<Declaration Statement>
<Statement>
<Statement Group>
<Cout Statement>
λ
INT
<Identifier>
SEMICOLON
COUT
INSERTION
<Expression>
SEMICOLON
<Assignment Statement>
<Identifier>
IDENTIFIER
IDENTIFIER
<Identifier>
ASSIGNMENT
<Expression>
SEMICOLON
IDENTIFIER
<Expression>
PLUS
<Expression>
<Integer>
<Integer>
INTEGER
INTEGER

Problems

1 Add Node.h and Node.cpp to your compiler project. Declare an abstract base class called Node. For now, it should only have one, empty virtual method – the destructor. The base class of an inheritance hierarchy should always have a virtual destructor so that cleanup takes place properly. As there will be several node classes that depend on each other, it may be helpful to prototype your node classes as follows:

```
// forward declarations:
class Node;
class StartNode;
etc.
```

2. Derive StartNode from Node. A StartNode's only (nonterminal) child is a ProgramNode, so the constructor of StartNode should take a ProgramNode pointer as a parameter and store it in a private variable of type ProgramNode pointer. StartNode's destructor should call delete on that pointer.

WARNING: Put the method implementations for all nodes in Node.cpp. Trying to inline them inside the class declaration causes problems.

3. Derive ProgramNode from Node. Its constructor should take a BlockNode parameter and store it in a private variable. Its destructor should call delete on the BlockNode pointer.

4. Create BlockNode in a similar fashion, with a StatementGroupNode being its only child. (In a later chapter we will change BlockNode to derive from StatementNode so a block can be used anywhere a statement is expected, such as the body of a "for" loop.)

5. Derive StatementGroupNode from Node. It should contain an stl vector of StatementNode pointers. Create an AddStatement method that takes as a parameter a StatementNode pointer and adds it to the vector. The destructor should loop through each StatementNode pointer, calling delete on each one. The vector itself is automatically deleted.

6. Create another abstract base class called StatementNode, which derives from Node. This allows only classes that derive from StatementNode to be stored in the stl vector of StatementNode pointers in a StatementGroupNode.

7. Derive DeclarationStatementNode from StatementNode. A DeclarationStatementNode needs a pointer to the IdentifierNode it declares. Create this class as before.

8. Derive AssignmentStatementNode from StatementNode. An AssignmentStatementNode has two children, an IdentifierNode and an ExpressionNode, so make a constructor that takes pointers to those types and

stores them in two private pointers. As always, the destructor should call delete on both of those pointers.

9. Derive CoutStatementNode from StatementNode. This node needs a pointer to an ExpressionNode.

10. Create another abstract base class called ExpressionNode. This is the first class that does not derive from Node, directly or indirectly. Later we will add the pure virtual function Interpret to Node, so all classes that derive from Node will have to write their own Interpret methods. But expressions are not interpreted, they are evaluated. So give the ExpressionNode class a pure virtual Evaluate method. It takes no parameters and returns an integer. Also make a virtual destructor with an empty implementation.

11. Derive IntegerNode from ExpressionNode. Its constructor should take an integer and store it in a private variable of type integer. The required Evaluate method should simply return that integer. No destructor is needed, since IntegerNodes have no nonterminal children.

12. Derive IdentifierNode from ExpressionNode. An IdentifierNode needs to store its label as a string, and it needs a pointer to the symbol table. Add a method called DeclareVariable(), which adds the IdentifierNode's label to the symbol table. That will later be called by DeclarationStatementNodes interpreting themselves. Add a method called SetValue(int v), which sets the value v into the symbol table by calling something like mSymbolTable→SetValue(mLabel, v). That will be called by AssignmentStatementNodes interpreting themselves. Add a method called GetIndex(), which returns the integer position of where this variable is being stored in the symbol table. That will be needed later when writing machine code. When asked to Evaluate itself, an IdentifierNode should call mSymbolTable→GetValue(mLabel) and return what it returns. Note that IdentifierNode does not need a destructor because it does not own the symbol table it points to.

13. Create another abstract base class called BinaryOperatorNode, which derives from ExpressionNode. Its constructor and data should allow it to store two ExpressionNode pointers, representing the left and right sides of the binary operator. Make the two data members protected instead of private so derived classes can directly access them. This class will need a destructor since it has two nonterminal node children.

14. Derive PlusNode from BinaryOperatorNode. Its constructor takes two ExpressionNode pointers as input, which are forwarded to its parent class with code like this:

```
PlusNode::PlusNode(ExpressionNode * left, ExpressionNode * right)
      : BinaryOperatorNode(left, right)
{
}
```

The required Evaluate method should recursively call Evaluate on its left and right children, returning the sum of their return values. No destructor is needed since the parent class destructor takes care of the left and right child pointers. NOTE: Make sure you do NOT declare mLeft and mRight ExpressionNode pointers as data members of the PlusNode. They are inherited from the parent class BinaryOperatorNode.

15. Following the pattern of PlusNode, create MinusNode, TimesNode, and DivideNode.

16. Create LessNode, LessEqualNode, GreaterNode, GreaterEqualNode, EqualNode, and NotEqualNode. These should all Evaluate to 1 or 0, for true or false respectively.

17. Test your code. For now, try building several nodes by hand, stuffing them into each other to form a very simple parse tree. Then call delete on the StartNode and verify that each node gets deleted exactly once. It is easy to get this wrong, resulting in double delete crashes, or nodes that don't delete at all, causing memory leaks.

5: The Parser – No Output

In previous chapters, we made parse trees by hand to test that our code was handling the creation and deletion of nodes correctly. In this chapter, we'll create a parser that recognizes whether a sequence of tokens is legal or not. In the next chapter, we'll enhance the parser to also output a parse tree, thus automating what we did by hand in previous chapters.

There are many kinds of parsers. We'll use a "Top-Down Recursive Descent" parser because it is quite effective, and yet relatively easy to code into our *Simple Compiler*. A Top-Down Recursive Descent parser is implemented by directly following the rules of some grammar.

For example, suppose we wanted to parse tokens provided by the scanner, to see if they followed the rules of a very simple grammar, such as:

<Start> → VOID MAIN LPAREN RPAREN.

We mirror this grammar rule into our parser class by creating a method called "Start()". The implementation of ParserClass::Start() would look like this:

```
void ParserClass::Start()
{
	Match(VOID_TOKEN);
	Match(MAIN_TOKEN);
	Match(LPAREN_TOKEN);
	Match(RPAREN_TOKEN);
}
```

The ParserClass::Match() method would look something like this:

```
// Verify that the next token in the input file is of the same type
// that the parser expects.
TokenClass ParserClass::Match(TokenType expectedType)
{
	TokenClass currentToken = mScanner->GetNextToken();
	if(currentToken.GetTokenType() != expectedType)
	{
		std::cerr << "Error in ParserClass::Match. " << std::endl;
		std::cerr << "Expected token type " <<
			TokenClass:: GetTokenTypeName(expectedType) <<
			", but got type " << currentToken.GetTokenTypeName() <<
			std::endl;
		exit(1);
	}
	MSG("\tSuccessfully matched Token Type: " <<
		currentToken. GetTokenTypeName() << ". Lexeme: \"" <<
		currentToken.GetLexeme() << "\"");
	return currentToken; // the one we just processed
}
```

Thus, calling the Start() method of the parser class enforces that the correct sequence of tokens are provided by the scanner, or an error is generated.

Suppose we make the grammar slightly more complicated, as follows:

<Start> → <Program> ENDFILE_TOKEN
<Program> → VOID_TOKEN MAIN_TOKEN LPAREN_TOKEN
RPAREN_TOKEN

Every nonterminal of the grammar becomes a method of the parser. For convenience, we give the methods the same name as the nonterminals. The code for ParserClass::Program() is the same as what we wrote for ParserClass::Start() in our previous example. The code for ParserClass::Start() is rewritten as follows:

```
void ParserClass::Start()
{
	Program();
	Match(ENDFILE_TOKEN);
}
```

The ParserClass::Start() code first recursively calls the code for ParserClass::Program(), then insists that the next token is of type ENDFILE_TOKEN, ensuring that there is no garbage in the input file after the required sequence of tokens.

Let's further enhance the grammar by altering the production rule for a <Program> and adding a few more rules:

<Program> → VOID_TOKEN MAIN_TOKEN LPAREN_TOKEN
RPAREN_TOKEN <Block>
<Block> → LCURLY_TOKEN <StatementGroup> RCURLY_TOKEN

As one would guess, the corresponding code for the ParserClass::Block() method insists on matching an LCURLY_TOKEN, then recursively calls the ParserClass::StatementGroup() method, then insists on matching an RCURLY_TOKEN, then returns.

Up to this point it has been easy because there has been only one choice for each nonterminal in the grammar. However, a StatementGroup can consist of any number of Statements, zero or more. The ParserClass::StatementGroup() method should therefore call the Statement() method over and over until it returns false. The ParserClass::Statement() method in turn needs to "peek" at the next token that will be coming from the scanner to decide what to do next. We are currently supporting only three kinds of statements – DeclarationStatements, AssignmentStatements, and CoutStatements. They begin with INT_TOKEN, IDENTIFIER_TOKEN, and COUT_TOKEN respectively. If the peeked token is one of those three, call the corresponding method and return true. Otherwise return false. As we enhance our compiler to support more kinds of statements, the Statement() method needs to be updated accordingly.

It is important that the tokens peeked at are not consumed. We'll put them back into the input file stream, as shown in problem 1.

Limitations of Recursive Descent Parsers

We were able to decide which kind of Statement to choose by peeking only one token ahead. Not all grammars can be discerned by looking ahead only one token. The subset of Context Free grammars that have this property are called LL1 grammars, and we will restrict our activities accordingly. This is a limitation of Recursive Descent Parsing that other kinds of parsers do not necessarily suffer. However, most grammars that are not LL1 can be re-written to be so, and the grammars that cannot be made LL1 are not necessary for our purposes anyway.

Expressions

We will expand the production rules for <Expression> to include a much richer language than we had in the Parse Tree chapter. We'll support add, subtract, multiply, and divide, and the six relational operators. We'll want the operators to operate in the correct order. That is, multiply and divide should happen first, from left to right, followed by add and subtract, from left to right, followed by the relational operators. We will also support parenthesis, nested to any degree.

An example expression is `Rate*3+(5-3+x)/10 <= x*(3+x*(2+x))`

Following is our expanded grammar for handling Expressions:

```
// <Expression> -> <Relational>

// <Relational> -> <PlusMinus> <RelationalTail>
// <RelationalTail> -> LESS_TOKEN <PlusMinus>
// <RelationalTail> -> LESSEQUAL_TOKEN <PlusMinus>
// <RelationalTail> -> GREATER_TOKEN <PlusMinus>
// <RelationalTail> -> GREATEREQUAL_TOKEN <PlusMinus>
// <RelationalTail> -> EQUAL_TOKEN <PlusMinus>
// <RelationalTail> -> NOTEQUAL_TOKEN <PlusMinus>
// <RelationalTail> -> {empty}

// <PlusMinus> -> <TimesDivide> <PlusMinusTail>
// <PlusMinusTail> -> PLUS_TOKEN <TimesDivide> <PlusMinusTail>
// <PlusMinusTail> -> MINUS_TOKEN <TimesDivide> <PlusMinusTail>
// <PlusMinusTail> -> {empty}

// <TimesDivide> -> <Factor> <TimesDivideTail>
// <TimesDivideTail> -> TIMES_TOKEN <Factor> <TimesDivideTail>
// <TimesDivideTail> -> DIVIDE_TOKEN <Factor> <TimesDivideTail>
// <TimesDivideTail> -> {empty}

// <Factor> → <Identifier>
// <Factor> → <Integer>
// <Factor> → LPAREN_TOKEN <Expression> RPAREN_TOKEN

// <Identifier> → IDENTIFIER_TOKEN
// <Integer> → INTEGER_TOKEN
```

These grammar rules can be easily converted into parser methods. For example, the rule that starts with <Relational> and the seven rules that start with <RelationalTail> are all combined into one method called ParserClass::Relational(). It uses Peek the same way that ParserClass::Statement() does to decide which of the 7 rules to choose. You could code it like this:

```
void ParserClass::Relational()
{
	PlusMinus();

	// Handle the optional tail:
	TokenType tt = mScanner->PeekNextToken().GetTokenType();
	if(tt == LESS_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	else if(tt == LESSEQUAL_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	else if(tt == GREATER_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	else if(tt == GREATEREQUAL_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	else if(tt == EQUAL_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	else if(tt == NOTEQUAL_TOKEN)
	{
		Match(tt);
		PlusMinus();
	}
	return;
}
```

Parser::PlusMinus() is handled in a similar fashion, except a while loop is needed to support these kinds of rules:

```
<PlusMinusTail> -> PLUS_TOKEN <TimesDivide> <PlusMinusTail>
```

The while loop allows multiple PLUS_TOKENs and MINUS_TOKENs to be handled. With Relationals, we only allow one or none, so we did not use a while loop.

```
void ParserClass::PlusMinus()
{
```

```
        TimesDivide();
        while(true)
        {
                TokenType tt = mScanner->PeekNextToken().GetTokenType();
                if(tt == PLUS_TOKEN)
                {
                        Match(tt);
                        TimesDivide();
                }
                else if(tt == MINUS_TOKEN)
                {
                        Match(tt);
                        TimesDivide();
                }
                else
                {
                        return;
                }
        }
}
```

Code your TimesDivide() method following exactly the same pattern as PlusMinus().

For the Factor() method, Peek ahead for an IDENTIFIER_TOKEN, an INTEGER_TOKEN, or an LPAREN_TOKEN. Match and Call accordingly. Don't use a while loop. If Factor doesn't see one of those three Token Types, print an appropriate error message and quit.

For now, ParserClass::Identifier and ParserClass::Integer should just Match their corresponding Token Types. Remember to include in your code the Match method provided earlier in this chapter.

Note that our expression grammar rules, and our corresponding methods, started with relational operators first, then add and subtract, then multiply and divide. This forces the higher priority operators to the bottom of the parse tree. This is correct, since parse trees will be evaluated recursively in postfix order – that is, from the leaf nodes towards the root. If we were going to introduce an even higher priority operator such as exponentiation, we'd need to introduce that layer at the bottom of our grammar, between TimesDivide and Factor.

Problems

1. Enhance your ScannerClass by adding a method called PeekNextToken(). It should behave just like GetNextToken(), except that it should not consume the token. This is easily accomplished by having PeekNextToken() first record the "get" position of mFin and the current line number, then call GetNextToken, and finally reset the "get" position of mFin and the current line number to where they were. Use the mFin methods tellg() and seekg(). Note that if the last character of the input file is the right curly brace, then peeking ahead will trigger the End Of File condition. In that case, mFin.seekg(filePos) will not work, until after the file has been cleared, so your PeekNextToken method will be more robust if you add code to do that, such as:

```
. . .
TokenClass tc = GetNextToken();
if(!mFin) // if we triggered EOF, then seekg doesn't work,
      mFin.clear();// unless we first clear()
. . .
```

2. Create a parser with no output. Add Parser.h and Parser.cpp to your compiler project. Create a new class called ParserClass. Its constructor should take as parameters pointers to a ScannerClass object and a SymbolTableClass object, which should in turn be stored in private variables. Only one public method besides the constructor is needed – Start(). When called, Start() should mimic some set of grammar rules (as provided below) using recursive descent parsing. If an error is found, your code should print an appropriate message and then quit. If Start() finishes, then the caller can conclude that the sequence of tokens provided by the scanner followed the rules of the grammar. Use the grammar given below, as well as the grammar rules provided within this chapter.

```
// <Start> → <Program> ENDFILE
// <Program> → VOID MAIN LPAREN RPAREN <Block>
// <Block> → LCURLY <StatementGroup> RCURLY
// <StatementGroup> → {empty}
// <StatementGroup> → <Statement> <StatementGroup>
// <Statement> → <DeclarationStatement>
// <Statement> → <AssignmentStatement>
// <Statement> → <CoutStatement>
// <Statement> → <Block>
// <DeclarationStatement> → INT <Identifier> SEMICOLON
// <AssignmentStatement> → <Identifier>ASSIGNMENT<Expression> SEMICOLON
// <CoutStatement> → COUT INSERTION <Expression> SEMICOLON
```

IMPORTANT: Note that we added a rule allowing a Statement to create a Block, so a block of code with curly braces can be inserted anywhere a single statement can be. Rewrite your BlockNode class so it derives from StatementNode instead of Node, and add Block as a forth kind of statement type in ParserClass::Statement(). Remember to thoroughly test your code, including token sequences that should not work.

6: The Parser – With Output

The parser we created in the last chapter checks any sequence of tokens for legality, according to whatever grammar we built the parser to match. If a sequence of tokens is not legal, the parser will at some point print an error message and quit. If the Start() method of the parser successfully returns then the sequence of tokens is legal. Other than “legal” or “not legal”, the parser has no output.

In this chapter we’ll enhance the parser so it also builds a parse tree, if the input sequence is legal. The trick is to enhance each of the parser methods to create and return a node equivalent to their respective types.

For example, the Start() method, which used to look like this:

```
void ParserClass::Start()
{
	Program();
	Match(ENDFILE_TOKEN);
}
```

will be enhanced to return a StartNode pointer, as follows:

```
StartNode * ParserClass::Start()
{
	ProgramNode *pn = Program();
	Match(ENDFILE_TOKEN);
	StartNode * sn = new StartNode(pn);
	return sn;
}
```

Hence, instead of just verifying there is a Program followed by an ENDFILE_TOKEN, we also create a StartNode and return it. The StartNode constructor needs a ProgramNode pointer, which we expect the Program() method to return as its output. Note that the StartNode is returned to the main function and becomes the root of the parse tree.

In turn, the Program() method ends by creating and returning a ProgramNode pointer, which must be initialized with a BlockNode pointer. The BlockNode pointer is obtained by storing the return value of the Block() method. The Block() method is coded in a similar fashion.

The StatementGroup() method is a little different. It begins by immediately creating a StatementGroupNode. In the while loop from last chapter, calls to Statement() returned true or false. Now they should return pointers to Statements, or NULL. As long as StatementNode pointers are being returned, use the AddStatement() method of StatementGroupNode to add these StatementNodes to your StatementNodeGroup.

The Statement() method should peek and then call specific Statement methods, as in the last chapter. But it should catch their return values and return what they return, or return NULL.

Enhance the DeclarationStatement(), AssignmentStatement(), and CoutStatement() methods of ParserClass so they return DeclarationStatementNodes, AssignmentStatementNodes, and CoutStatementNodes. These nodes are easily constructed using the return values of their sub-method calls.

The various Expression methods get somewhat tricky. Every one of them needs to return a pointer to a base class ExpressionNode, except Parser::Identifier() should return an IdentifierNode pointer and Parser::Integer() should return an IntegerNode pointer. Most of them need to construct a specific child node of ExpressionNode, and some need the option to do that in a layered way inside a while loop. Let's enhance the PlusMinus() method given in the last chapter. Before it looked like this:

```
void ParserClass::PlusMinus()
{
	TimesDivide();
	while(true)
	{
		TokenType tt = mScanner->PeekNextToken().GetTokenType();
		if(tt == PLUS_TOKEN)
		{
			Match(tt);
			TimesDivide();
		}
		else if(tt == MINUS_TOKEN)
		{
			Match(tt);
			TimesDivide();
		}
		else
		{
			return;
		}
	}
}
```

We need it to look something more like this:

```
ExpressionNode * ParserClass::PlusMinus()
{
	ExpressionNode * current = TimesDivide();
	while(true)
	{
		TokenType tt = mScanner->PeekNextToken().GetTokenType();
		if(tt == PLUS_TOKEN)
		{
			Match(tt);
			current = new PlusNode(current, TimesDivide());
		}
```

```
                else if(tt == MINUS_TOKEN)
                {
                        Match(tt);
                        current = new MinusNode(current, TimesDivide());
                }
                else
                {
                        Return current;
                }
        }
}
```

Enhance ParserClass::TimesDivide() the same way. ParserClass::Relational() is nearly the same, except it doesn't need the while loop, since we don't allow more than one relational operator in an expression.

It is important to remember to keep everything properly layered. That is, ParserClass::Expression() method should call the lowest priority operator, which for now is Relational. The Relational level should call the next lowest priority operators, which is the PlusMinus level. It calls the highest priority operators, which is the TimesDivide level. The highest level finishes by calling the Factor level:

Expression -> Relational -> PlusMinus -> TimesDivide -> Factor.

Later, when we add more operators, we must layer them in the correct positions. Since the logical OR operator has the lowest priority, and the logical AND operator the second lowest, and the Exponent operator the highest, the layering at that point will look like this:

Expression -> Or -> And -> Relational -> PlusMinus -> TimesDivide -> Exponent -> Factor.

To enhance the ParserClass::Factor() method, catch whatever pointer is returned by the sub call to Identifier(), Integer(), or Expression() method, and return that pointer to the calling code. To Enhance ParserClass::Identifier(), catch the TokenClass returned by the Match() method. Use its lexeme to make and return an IdentifierNode. Do likewise for ParserClass::Integer(), except you'll need to convert the lexeme to an integer using a function such as "atoi()". Note that atoi() expects an array of characters, not a string type, so the string must be converted to a character array using the c_str() method of the string class.

Problems

1. Enhance the parser so it outputs a correct parse tree. Test your code.

7: The Interpreter

Given a correct parse tree, interpreting is easy. Recall from the previous chapter that the return value of `ParserClass::Start()` is a `StartNode*`, which is also the root of the resulting parse tree. We simply need to teach all the different kinds of nodes how to interpret themselves, and start it going with code like this:

```
StartNode * root = parser.Start();
root→Interpret();
```

As all the nodes that derive from Node need to be able to Interpret themselves, add the pure virtual method Interpret() to the base class Node.

The implementation for StartNode should simply forward the call to its ProgramNode child, as follows:

```
void StartNode::Interpret()
{
	mProgramNode→Interpret();
}
```

Likewise, a ProgramNode interprets itself by telling its BlockNode to interpret itself. The BlockNode implementation is similar.

A StatementGroupNode interprets itself by looping through all its child statements, telling each one in turn to interpret itself.

A DeclarationStatementNode interprets itself by calling the DeclareVariable method of its IdentifierNode child.

An AssignmentStatementNode determines the value to assign by calling the Evaluate method of its expression child. It then sets that value into its Identifier child by calling the SetValue method of IdentifierNode.

A CoutStatementNode interprets itself by calling Evaluate on its expression, then printing to the monitor the value it returns. For now, also print a space to separate one from the next.

Problems

1. Implement the Node Interpreting code as explained in this chapter. Note that your Simple Compiler should already have the Evaluate() method for all nodes deriving from ExpressionNode. Give all the code a good testing.

2. Enhance your code to support the “if” and “while” statements, as they work in the C language. Make sure they Interpret correctly.

3. Enhance your code to support the “&&” and “||” operators, as they work in C. As with the PlusMinus and TimesDivide levels, you should allow multiple AND and OR operators. Enforce the following priorities for order of operations, from highest to lowest:

* /

+ -

< <= > >= == !=

&&

||

Follow the same pattern as before to get the priorities and left to right order within a priority. That is, enhance the grammar itself by adding two more layers to the existing three layers, and then write Parser methods to correspond to the grammar rules.

8: Machine Language Basics

The result of compiling a source code file is usually an executable file. However, we will not produce an executable file because they require several bytes of header information, which is beyond the scope of this book. Instead, our Simple Compiler will create a huge array and fill it with machine codes. After compiling into that array, our Simple Compiler will then jump into that array as if it were a function, execute its contents, then return from that array. Thus, our Compiler is really a "Compiler and Executer".

Machine language is inherently non-portable, so we'll have to limit our platforms. We'll make it work on Linux/Intel machines and Windows WSL (Windows Subsystem for Linux) environments, using the g++ compiler.

It might be helpful to get a reference book on Intel machine language, though the machine codes we'll need for our Simple Compiler will be explained briefly in this book.

We'll start with an array of unsigned chars (bytes) to hold the machine codes, and make it behave like a function. We'll stuff it with a few specific codes, then show how to CALL into it and how to RETURN again.

A standard function always starts with the same 3 machine codes which are, in hex notation, 0x55, 0x8B, and 0xEC. These set up the stack frame for the function (in the EBP register). After those three initial bytes, we can add more machine codes to do whatever it is we want our array-function (program) to do. Finally, the function must end with these two bytes: 0X5d and 0XC3. The first pops the contents of the system stack back into the EBP register to restore it to what it was before, and the second is a RETURN instruction, which causes the instruction pointer register to jump back to wherever the code was executing before it jumped to our array.

After our Simple Compiler is done compiling machine codes into the array, it must jump into that array as if it were a standard function. Suppose the name of the `unsigned char` array is mCode. The following C++ code tricks a compiler into thinking an array of unsigned chars is a function:

```
void * ptr = mCode;
void (*f)(void);
f = (void (*)(void)) ptr ;
f();
```

It first changes the type of mCode from unsigned char array to void pointer. Then it creates a variable called f of type "pointer to a function that takes no parameters and returns no parameters." Then it sets variable f to the same address as the mCode array. The final line calls our array, just as if it were a function. The g++ compiler that you are using to compile your Simple Compiler handles the f() call by pushing the current instruction pointer register onto the

stack, then setting the instruction pointer to the address contained in variable f. Your machine code then takes over, until it ends with the RETURN instruction, which as we have already indicated, pops the previous value of the instruction pointer back into the instruction pointer register, causing execution to continue with whatever code is after the f() call.

Note that modern machines prevent transferring execution from the code segment to the data segment, as that is how some viruses work. To get around this “protection” we indicate that mCode is something we intentionally want to execute, with this code:

```
mprotect((void *)((uintptr_t)mCode & ~(sysconf(_SC_PAGE_SIZE)-1)),
      MAX_INSTRUCTIONS, PROT_READ|PROT_WRITE|PROT_EXEC);
```

Before proceeding further, you might want to try this much of it, and make sure it works on your Linux platform. Try compiling with g++ and executing the following code.

IMPORTANT NOTE: If you are copy/pasting from here into a vi editor, you’ll want to enter “paste” mode using
:set paste
You can return to the previous mode with
:set unpaste

```
#include <iostream>
#include <unistd.h>
#include <sys/mman.h>

int main()
{
      // Beginning and end of every function:
      unsigned char mCode[] = {0x55, 0x8B, 0xEC,
            0X5d, 0XC3};

      // This allows mCode data to be called as if it were a function:
      int MAX_INSTRUCTIONS = 5;
      mprotect((void *)((uintptr_t)mCode& ~(sysconf(_SC_PAGE_SIZE)-1)),
            MAX_INSTRUCTIONS, PROT_READ|PROT_WRITE|PROT_EXEC);

      std::cout << "About to Execute the machine code...\n";
      void * ptr = mCode;
      void (*f)(void);
      f = (void (*)(void)) ptr ;
      f(); // call the array as if it were a function
      std::cout << "There and back again!\n\n";
      return 0;
}
```

When you run it, the output should be:

```
      About to Execute the machine code...
      There and back again!
```

If not, that needs to be fixed before you can proceed.

In the end-of-chapter problems, we'll create a class called "Instructions" for organizing the machine codes. We'll do this in its own project – separate from our Simple Compiler project. We'll create methods of class Instructions that will make it easy to convert our parse tree into the correct machine codes.

In the next chapter, we'll add the pure virtual method Code() to the base class Node, and then implement Code() for each derived node class so that they properly call the methods developed in this chapter. At that point, we can continue to Interpret() the parse tree, or Code() it and then execute the result.

Problems

1. We are going to refactor the main() function from this chapter into several files and methods that will be better suited for expansion. Create a new project, separate from your Simple Compiler project. You will need to create the files Instructions.h, Instructions.cpp, and InstructionsTester.cpp. In Instructions.h, declare the InstructionsClass as follows:

```
const int MAX_INSTRUCTIONS = 5000;
class InstructionsClass
{
public:
	InstructionsClass();
	void Encode(unsigned char c);
	void Finish();
	void Execute();
private:
	unsigned char mCode[MAX_INSTRUCTIONS];
	int mCurrent; // where we are in mCode
	int mStartOfMain;
};
```

Then in Instructions.cpp implement the methods as follows:

```
#include <iostream>
#include <unistd.h>
#include <sys/mman.h>
#include "Instructions.h"

// Some assembly like definitions for our machine code:
const unsigned char PUSH_EBP = 0x55;
const unsigned char MOV_EBP_ESP1 = 0x8B;
const unsigned char MOV_EBP_ESP2 = 0xEC;

const unsigned char POP_EBP = 0x5D;
const unsigned char NEAR_RET = 0xC3;

// Put one instruction at a time into mCode:
void InstructionsClass::Encode(unsigned char c)
{
	if(mCurrent + sizeof(unsigned char) < MAX_INSTRUCTIONS)
	{
		mCode[mCurrent] = c;
		mCurrent += sizeof(unsigned char);
	}
	else
	{
		std::cerr << "Error.  Used up all " << MAX_INSTRUCTIONS
			<< " instructions." << std::endl;
		exit(1);
	}
}

InstructionsClass::InstructionsClass()
{
	mprotect((void *)((uintptr_t)mCode& ~(sysconf(_SC_PAGE_SIZE)-1)),
		MAX_INSTRUCTIONS, PROT_READ|PROT_WRITE|PROT_EXEC);
```

```
	mCurrent = 0;
	mStartOfMain = 0;

	// All functions start this way. So does our main function.
	Encode(PUSH_EBP);
	Encode(MOV_EBP_ESP1);
	Encode(MOV_EBP_ESP2);
}

void InstructionsClass::Finish()
{
	// All functions end this way:
	Encode(POP_EBP);
	Encode(NEAR_RET);

	std::cout << "Finished creating " << mCurrent <<
		" bytes of machine code" << std::endl;
}

void InstructionsClass::Execute()
{
	// Jump into the main function of what we just coded,
	//    found at mCode[mStartOfMain]
	std::cout << "About to Execute the machine code..." << std::endl;
	void * ptr = &(mCode[mStartOfMain]);
	void (*f)(void);
	f = (void (*)(void)) ptr ;
	f();
	// Did everything work?
	std::cout << "\nThere and back again!" << std::endl;
}
```

Finally, in InstructionsTester.cpp, write a main() function that does this:

```
InstructionsClass code;
code.Finish();
code.Execute();
```

Make sure it compiles and executes without crashing.

2.a Declare and implement a second, overloaded helper function called Encode that takes an integer as the input parameter. The input integer should be copied into the next 4 bytes of mCode. The following fancy casting should help. Also, be sure to add code to check for buffer overflow, like the previous Encode does.

```
void InstructionsClass::Encode(int x)
{
	*((int*)(&( mCode [mCurrent]))) = x;
	mCurrent += sizeof(int);
}
```

2.b Do likewise for an overloaded version of Encode that takes a long long. This is needed for 64 bit architecture addresses. Variables of type long long are 8 bytes in length.

2.c You'll need one more overload of Encode that takes a pointer of 4 or 8 bytes, depending on the build platform. This will in turn call one of the above versions.

```
void InstructionsClass::Encode(void * p)
{
	int pointerSize = sizeof(p);

	if (pointerSize==4)
	{
		Encode((int)(long long)p);
	}
	else if(sizeof(p)==8)
	{
		Encode((long long)p);
	}
}
```

3. If one function uses certain registers and then calls another function that clobbers those registers, bad things happen. The convention is that the registers EBP, ESP, EBX, ESI, and EDI should be saved (pushed) and restored (popped) by the called function, if it changes them. The remaining registers should be saved and restored by the calling function, if it needs their values kept intact. Since we will be changing all these registers' values, add these lines to the very end of the InstructionsClass constructor:

```
	// Make sure we save and restore all 5 Callee-Save registers.
	// That is, EBP, ESP, EBX, ESI, and EDI
	Encode(PUSH_EBX);
	Encode(PUSH_ESI);
	Encode(PUSH_EDI);
```

And add these lines to the very beginning of the Finish method:

```
	// Retore Callee-Saved registers:
	Encode(POP_EDI);
	Encode(POP_ESI);
	Encode(POP_EBX);
```

Add these constants at the top of Instructions.cpp to make it work:

```
const unsigned char PUSH_EBX = 0x53;
const unsigned char PUSH_ESI = 0x56;
const unsigned char PUSH_EDI = 0x57;
const unsigned char POP_EDI = 0x5F;
const unsigned char POP_ESI = 0x5E;
const unsigned char POP_EBX = 0x5B;
```

Test your project to make sure it still doesn't crash.

4. In the next chapter, we will compile code to evaluate expression nodes. Our protocol will always be to push the resulting integer on the stack. We will support that infrastructure in this chapter. For starters, write a method called PushValue which takes as input an integer parameter named "value", and fills mCode with machine codes that will push the given integer value onto the stack. The following implementation works:

```
	Encode(IMMEDIATE_TO_EAX);
```

```
	Encode(value);
	Encode(PUSH_EAX);
```

The first byte tells the CPU to Move an integer into the EAX register, and the next 4 bytes are the immediate value to copy. The second machine language instruction requires only one byte and no parameters. It pushes the value in the EAX register onto the stack.

You will need these constants:

```
const unsigned char IMMEDIATE_TO_EAX = 0xB8;
	//followed by 4 byte address.
const unsigned char PUSH_EAX = 0x50;
```

Make sure your code still builds. Don't test it yet because your stack won't be balanced until you Pop that integer back off the stack.

5. Writing machine code that prints one integer is perhaps the most difficult part of this entire Simple Compiler project. To help simplify, we'll break the job into 4 methods, PopAndWrite, PrintIntegerLinux64, PrintMinusLinux64, and PrintSpaceLinux64. PopAndWrite will take whatever integer was just pushed to the stack (by PushValue, for example), pop it back off the stack, and move it to a special class variable named mPrintInteger. Then PopAndWrite will CALL the method PrintIntegerLinux64, which will in turn print whatever integer is stored in mPrintInteger. PrintMinusLinux64 and PrintSpaceLinux64 are pieces of PrintIntegerLinux64 but separated out for clarity.

Since PrintIntegerLinux64 is long, we want it to behave like a function so we can code it once and then call it many times. We'll write the code once from the constructor of InstructionsClass and then call it perhaps many times from PopAndWrite, as already mentioned. As you might guess from the function names, this code only works on Linux 64-bit architectures. All the code is given here:

Enhance the InstructionsClass::InstructionsClass constructor to first write the code for the PrintIntegerLinux64 function to the mCode array. Afterwards we'll write the code for the main function entry point:

```
InstructionsClass::InstructionsClass()
{
	// This command allows mCode data to be called as a function.
	mprotect((void *)((uintptr_t)mCode& ~(sysconf(_SC_PAGE_SIZE)-1)),
		MAX_INSTRUCTIONS, PROT_READ|PROT_WRITE|PROT_EXEC);

	// Initialize all class variables:
	mCurrent = 0;
	mStartOfMain = 0;
	mPrintInteger = 0;
	mTempInteger = 0;
	mMinusString = '-';
	mSpaceString = ' ';
```

```
	// Record where the PrintIntegerLinux64 function starts:
	mStartOfPrint = mCurrent;

	// Code one function PrintIntegerLinux64 that prints an integer.
	// It will be called later many times.
	PrintIntegerLinux64(); // Write all the codes into mCode.

	// Now record where the main function will start:
	mStartOfMain = mCurrent;

	// All functions start this way. So does the main function.
	Encode(PUSH_EBP);
	Encode(MOV_EBP_ESP1);
	Encode(MOV_EBP_ESP2);
	// Make sure we save and restore all 5 Callee-Save registers.
	// That is, EBP, ESP, EBX, ESI, and EDI,
	Encode(PUSH_EBX);
	Encode(PUSH_ESI);
	Encode(PUSH_EDI);
}

// Modified from
https://baptiste-wicht.com/posts/2011/11/print-strings-integers-intel-ass
embly.html
// This is made into a function, so it can be called per cout instead of
rewritten every time.
// The integer to print should be previously stored in mPrintInteger.
void InstructionsClass::PrintIntegerLinux64()
{
	// All functions start this way. So does PrintInteger.
	Encode(PUSH_EBP);
	Encode(MOV_EBP_ESP1);
	Encode(MOV_EBP_ESP2);
	// Make sure we save and restore all 5 Callee-Save registers.
	// That is, EBP, ESP, EBX, ESI, and EDI,
	Encode(PUSH_EBX);
	Encode(PUSH_ESI);
	Encode(PUSH_EDI);

	// Get the integer to print from mPrintInteger:
	Encode(MEM_TO_EAX);
	Encode(&mPrintInteger);

	// Check if the number to print is negative:
	Encode(CMP_EAX1);
	Encode(CMP_EAX2);
	Encode((unsigned char)0);

	// Jump if the integer is Not negative
	Encode(JGE);
	unsigned char DistanceToJump = 0; // fill in later
	int fillInAddress = mCurrent;
	Encode(DistanceToJump); // fill in this distance later
	unsigned char * jumpFrom = GetAddress();

	// Negate negative integers:
	Encode(NEG_EAX1);
```

```
Encode(NEG_EAX2);

// Print the minus sign, first saving register EAX.
Encode(PUSH_EAX);
WriteMinusLinux64();
Encode(POP_EAX);

// Fill in how far to jump from before:
unsigned char * beginningOfPrintPositiveInteger = GetAddress();
mCode[fillInAddress]= (unsigned char)
      (beginningOfPrintPositiveInteger -jumpFrom);

// Jump to here!
// Beginning of Write Positive Integer:

// ECX is counter for how many decimal bytes are in the integer
Encode(IMMEDIATE_TO_ECX);
Encode((int)0);

// Beginning of loop1.
// It puts all the ascii characters of the integer on the stack.
      unsigned char *divide_loop= GetAddress();

      // increment the counter
      Encode(ADD_ECX1);
      Encode(ADD_ECX2);
      Encode((unsigned char)1);

      // Clear out high bytes before division
      Encode(CDQ);    // Necessary to clear the D register.

      // edx = eax % 10
      Encode(IMMEDIATE_TO_EBX);
      Encode((int)10); // Divide by 10
      Encode(DIV_EAX_EBX1); // divide eax by ebx
      Encode(DIV_EAX_EBX2); // remainder result in edx

      // convert decimal remainder to ascii by adding 48.
      Encode(ADD_EDX1);
      Encode(ADD_EDX2);
      Encode((unsigned char)48);

      // ascii remainder onto stack
      Encode(PUSH_EDX);

      // we stop loop1 when quotient (eax) is zero
      Encode(CMP_EAX1);
      Encode(CMP_EAX2);
      Encode((unsigned char)0);

      // repeat loop1 while quotient != 0. Encode the jump.
      unsigned char *end_divide_loop = GetAddress() + 2;
      Encode(JNE);
      Encode((unsigned char)(divide_loop-end_divide_loop));
// End of loop1.

// Beginning of loop2.
```

```
// It pops and prints all ascii characters.
      unsigned char *print_loop = GetAddress();

      // Code to print a character using syscall:
      {
      Encode(POP_EAX);
      Encode(EAX_TO_MEM);
      Encode(&mPrintInteger);

      Encode(BIT64);
      Encode(IMMEDIATE_TO_ESI); // move into ESI
      Encode(&mPrintInteger); // address of bytes to print

      Encode(IMMEDIATE_TO_EAX);
      Encode((int)1); // 1 for print syscall

      Encode(IMMEDIATE_TO_EDI);
      Encode((int)1); // 1 for stdout port

      Encode(IMMEDIATE_TO_EDX);
      Encode((int)1); // length of bytes to print

      // Engage 64 bit syscall with special codes 0x0F, 0x05
      // The EAX register indicates which syscall (print)
      // The EDI tells which port to print to (stdout)
      // The ESI contains the address to print
      // The EDX says how many bytes to print
      Encode(PUSH_ECX); // First save important registers
      Encode((unsigned char)SYS_CALL1);
      Encode((unsigned char)SYS_CALL2);
      Encode(POP_ECX); // Restore important register
      }

      // Decrement the count of characters left to print
      Encode(SUB_ECX1);
      Encode(SUB_ECX2);
      Encode((unsigned char)1);

      // Are there no characters left to print?
      Encode(CMP_ECX1);
      Encode(CMP_ECX2);
      Encode((unsigned char)0);

      // Repeat loop if there are more characters to print
      unsigned char *end_print_loop = GetAddress() + 2;
      Encode(JNE);
      Encode((unsigned char)(print_loop-end_print_loop));
// End of loop2.

// Separate multiple printed items with a space.
WriteSpaceLinux64();

// Restore Callee-Saved registers:
Encode(POP_EDI);
Encode(POP_ESI);
Encode(POP_EBX);
// All functions end this way:
```

```
	Encode(POP_EBP);
	Encode(NEAR_RET);
}

// 64 bit print syscall Taken from:
//
https://stackoverflow.com/questions/22503944/using-interrupt-0x80-on-64-b
it-linux
void InstructionsClass::WriteMinusLinux64()
{
	Encode(IMMEDIATE_TO_EAX);
	Encode((int)1); // 1 for print syscall

	Encode(IMMEDIATE_TO_EDI);
	Encode((int)1); // 1 for stdout port

	Encode(BIT64);
	Encode(IMMEDIATE_TO_ESI);
	Encode(&mMinusString); // address of bytes to print

	Encode(IMMEDIATE_TO_EDX);
	Encode((int)1); // length of bytes to print

	// 64 bit syscall:
	Encode((unsigned char)SYS_CALL1);
	Encode((unsigned char)SYS_CALL2);
}

void InstructionsClass::WriteSpaceLinux64()
{
	Encode(IMMEDIATE_TO_EAX);
	Encode((int)1); // 1 for print syscall

	Encode(IMMEDIATE_TO_EDI);
	Encode((int)1); // 1 for stdout port

	Encode(BIT64);
	Encode(IMMEDIATE_TO_ESI);
	Encode(&mSpaceString); // address of bytes to print

	Encode(IMMEDIATE_TO_EDX);
	Encode((int)1); // length of bytes to print

	// 64 bit syscall:
	Encode((unsigned char)SYS_CALL1);
	Encode((unsigned char)SYS_CALL2);
}

void InstructionsClass::PopAndWrite()
{
	// Move the integer to be printed from stack to mPrintInteger:
	Encode(POP_EAX);
	Encode(EAX_TO_MEM);
	Encode(&mPrintInteger);

	// Call previously coded function that prints mPrintInteger
	Call( (void*) &(mCode[mStartOfPrint]));
```

```
            // &(mCode[mStartOfPrint])is where PrintIntegerLinux64 is.
}

void InstructionsClass::Call(void * function_address)
{
      unsigned char * a1 = (unsigned char*)function_address;
      unsigned char * a2 = (unsigned char*)(&mCode[mCurrent+5]);
      int offset = (int)(a1 - a2);
      Encode(CALL);
      Encode(offset);
}

unsigned char * InstructionsClass::GetAddress()
{
      return &(mCode[mCurrent]);
}
```

You will need these additional codes defined:

```
const unsigned char SYS_CALL1 = 0x0F;
const unsigned char SYS_CALL2 = 0x05;
const unsigned char CALL = 0xE8;
      // function Call within segment. Add 4 byte offset
const unsigned char POP_EAX = 0x58;
const unsigned char EAX_TO_MEM = 0xA3;
      // Add 4 (or 8) byte address value in reverse order
const unsigned char BIT64 = 0x48;
      // BIT64 (0x48) makes IMMEDIATE_TO_EAX take 8 byte address,
      //          or IMMEDIATE_TO_RAX
const unsigned char IMMEDIATE_TO_EDX = 0xBA;
const unsigned char IMMEDIATE_TO_ESI = 0xBE;
const unsigned char IMMEDIATE_TO_EDI = 0xBF;
const unsigned char CMP_EAX1 = 0x83; // followed by 1 byte
const unsigned char CMP_EAX2 = 0xF8;
const unsigned char MEM_TO_EAX = 0xA1;
      // Add 4 (or 8) byte address value in reverse order
const unsigned char JNE = 0x75;
const unsigned char JGE = 0x7D;
const unsigned char NEG_EAX1 = 0xF7;
const unsigned char NEG_EAX2 = 0xD8;
const unsigned char IMMEDIATE_TO_ECX = 0xB9;
const unsigned char IMMEDIATE_TO_EBX = 0xBB;
const unsigned char ADD_ECX1 = 0x83; // followed by 1 byte to add.
const unsigned char ADD_ECX2 = 0xC1;
const unsigned char SUB_ECX1 = 0x83; // followed by 1 byte to add.
const unsigned char SUB_ECX2 = 0xE9;
const unsigned char DIV_EAX_EBX1 = 0xF7;
const unsigned char DIV_EAX_EBX2 = 0xFB;
      // Registers D and A divided by B. (clear D with CDQ first!)
      // .. Quotient->A and Remainder->D.
const unsigned char CDQ = 0x99;
const unsigned char ADD_EDX1 = 0x83; // followed by 1 byte
const unsigned char ADD_EDX2 = 0xC2;
const unsigned char POP_ECX = 0x59;
const unsigned char POP_EDX = 0x5A;
const unsigned char PUSH_ECX = 0x51;
const unsigned char PUSH_EDX = 0x52;
```

```
const unsigned char CMP_ECX1 = 0x83; // followed by 1 byte
const unsigned char CMP_ECX2 = 0xF9;
```

And this complete set of class variables:

```
	unsigned char mCode[MAX_INSTRUCTIONS];
	int mCurrent; // where we are in mCode
	int mPrintInteger;
		// Location to store an integer about to be printed.
	int mTempInteger;
		// Location to store one char of integer to be printed.

	int mStartOfPrint;
		// The value of mCurrent where PrintIntegerLinux 64 starts.
		// Jump to this offset of mCode to print.
	int mStartOfMain;
		// the value of mCurrent after coding the Print functions.
		// Jump to this offset of mCode to start program execution.
	char mMinusString; // Holds '-'
	char mSpaceString; // Holds ' '
```

Test all this code by adding code.PushValue and code.PopAndWrite to InstructionsTester.cpp:

```
int main()
{
	InstructionsClass code;

	code.PushValue(-500);
	code.PopAndWrite();
	code.PushValue(1000);
	code.PopAndWrite();

	code.Finish();
	code.Execute();
	return 0;
}
```

Stop and test your code. Make sure it compiles clean, and prints this when you run it:

```
Finished creating 208 bytes of machine code
About to Execute the machine code...
-500 1000
There and back again!
```

6. The previous Symbol Table worked great for Interpreting, but for Coding and Executing we'll need a chunk of RAM memory in which to store all our integer variables. In Instructions.h, declare this array, and declare MAX_DATA to be about 5000 big:

```
int mData[MAX_DATA];
```

We need a helper method that will tell us the address of any integer in mData. It should look similar to the following, but should also include code to make sure index is less than MAX_DATA.

```
int * InstructionsClass::GetMem(int index)
{
	return  &(mData[index]);
}
```

7. Just as PushValue (written previously) when executed, put the value of an immediate integer on the stack, we'll need a PushVariable to copy the contents of a symbol table entry from mData to the stack.

PushVariable takes one integer parameter - the index into mData specifying which integer should be pushed onto the stack. We use a machine code to move RAM Memory to the EAX register, which we already declared as MEM_TO_EAX. The four bytes following are the RAM address of which integer to move, which you can find by calling the helper method GetMem. Finally, you'll need to push the contents of the EAX register onto the stack, using PUSH_EAX. In all, PushVariable should create six bytes of machine code. Try to write it yourself.

8. PopAndStore should remove an integer from the stack and copy it into your symbol table, mData. It takes one parameter, the integer index into mData. First POP_EAX to move the integer from the top of the stack to the EAX register. Then EAX_TO_MEM to move it from the EAX register to the correct RAM address, as found by calling GetMem. Be sure to put the RAM address After the EAX_TO_MEM code. You need to write this one! Then test with code like this:

```
int main()
{
	InstructionsClass code;

	code.PushValue(500); // 500 to stack
	code.PopAndStore(1); // to any slot number of mData
	code.PushVariable(1); // from mData back onto stack
	code.PopAndWrite();

	code.Finish();
	code.Execute();
	return 0;
}
```

9. We need to support evaluating mathematical expressions in machine code. Again, our protocol when evaluating anything is to leave its integer value on the stack. For example, suppose we want to evaluate "100 / x". We would first use PushValue to get the 100 onto the stack. Then PushVariable to get the current value of x onto the stack (explained more later). Then we'll call PopPopDivPush. It assumes its two operands are already on the stack. It needs to pop them both off, divide them, and then push that integer result back onto the stack. The code is provided here:

```
void InstructionsClass::PopPopDivPush()
{
	Encode(POP_EBX);
	Encode(POP_EAX);
	Encode(CDQ); // Necessary to clear the EDX before the divide.
	Encode(DIV_EAX_EBX1); // Divide EAX by EBX. Result in EAX.
```

```
        Encode(DIV_EAX_EBX2);
        Encode(PUSH_EAX);
}
```

Test with code like this:

```
code.PushValue(100);
code.PushValue(8);
code.PopPopDivPush();
code.PopAndWrite(); // should print 12
```

10. Following the same pattern, write PopPopAddPush, PopPopSubPush, and PopPopMulPush. Note that they do not need the CDQ code as divide does, so their total machine code bytes will be only 5 instead of 6.

You'll need these codes:

```
const unsigned char ADD_EAX_EBX1 = 0x03;
const unsigned char ADD_EAX_EBX2 = 0xC3;
const unsigned char SUB_EAX_EBX1 = 0x2B;
const unsigned char SUB_EAX_EBX2 = 0xC3;
const unsigned char MUL_EAX_EBX1 = 0xF7;
const unsigned char MUL_EAX_EBX2 = 0xEB;
```

Write code to test all of them.

11. We'll next handle the 6 relational operators. Since these are almost identical, we'll factor out the common code, given here:

```
void InstructionsClass::PopPopComparePush(unsigned char
            relational_operator)
{
        Encode(POP_EBX);
        Encode(POP_EAX);
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2); // The FLAG register is now set.
        Encode(IMMEDIATE_TO_EAX); // load A register with 1
        Encode(1); // assume the result of compare is 1, or TRUE.
        Encode(relational_operator);
            // Depending on the FLAG register and this
            // particular relational_operator,
            // possibly skip around setting A register
            // to zero, or FALSE, leaving it at TRUE.
        Encode((unsigned char)5);
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        Encode(PUSH_EAX); // push 1 or 0
}
```

Here is how you would call that common code for the Less operator:

```
void InstructionsClass::PopPopLessPush()
{
        PopPopComparePush(JL);
}
```

You'll need these constants:

```
const unsigned char CMP_EAX_EBX1 = 0x3B; // compares A and B registers.
const unsigned char CMP_EAX_EBX2 = 0xC3; // followed by 1 byte value
const unsigned char JL = 0x7C;
```

Test with code like this:

```
code.PushValue(3);
code.PushValue(4);
code.PopPopLessPush();
code.PopAndWrite(); // should print 1

code.PushValue(4);
code.PushValue(4);
code.PopPopLessPush();
code.PopAndWrite(); // should print 0

code.PushValue(5);
code.PushValue(4);
code.PopPopLessPush();
code.PopAndWrite(); // should print 0
```

Write the other 5 relational operators in a similar way, using these constants. Remember to test them all.

```
const unsigned char JLE = 0x7E;
const unsigned char JG = 0x7F;
//const unsigned char JGE = 0x7D; // already declared
const unsigned char JE = 0x74;
//const unsigned char JNE = 0x75; // already declared
```

12. The two logical operators are harder. Their code is given here, but be sure to test them just the same:

```
void InstructionsClass::PopPopAndPush()
{
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        Encode(POP_EBX); // load B register with stack item 2
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2);
        Encode(POP_EBX); // load B register with stack item 1
        Encode(JE); // if stack item 2 is zero, jump to FALSE code
        Encode((unsigned char)11);
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2);
        Encode(JE); // if stack item 1 is zero, jump to FALSE code
        Encode((unsigned char)7);
        // TRUE code:
        Encode(IMMEDIATE_TO_EAX); // load A register with 1
        Encode(1);
        Encode(JUMP_ALWAYS); // Jump around FALSE code
        Encode((unsigned char)5);
        // FALSE code:
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        // Save A to the stack
        Encode(PUSH_EAX); // push 1 or 0
}
```

```
void InstructionsClass::PopPopOrPush()
{
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        Encode(POP_EBX); // load B register with stack item 2
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2);
        Encode(POP_EBX); // load B register with stack item 1
        Encode(JNE); // if stack item 2 is not zero, jump to TRUE code
        Encode((unsigned char)11);
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2);
        Encode(JNE); // if stack item 1 is not zero, jump to TRUE code
        Encode((unsigned char)7);
        // FALSE code:
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        Encode(JUMP_ALWAYS); // Jump around TRUE code
        Encode((unsigned char)5);
        // TRUE code:
        Encode(IMMEDIATE_TO_EAX); // load A register with 1
        Encode(1);
        // Save A to the stack
        Encode(PUSH_EAX); // push 1 or 0
}
```

Add these constants:

```
const unsigned char JUMP_ALWAYS = 0xEB; // followed by 1 byte value
```

13. If and While statements need to jump across their body if their expression evaluates to zero. To help us prepare for necessary expression testing and jumping, add these final methods:

```
unsigned char * InstructionsClass::SkipIfZeroStack()
{
        Encode(POP_EBX);
        Encode(IMMEDIATE_TO_EAX); // load A register with 0
        Encode(0);
        Encode(CMP_EAX_EBX1);
        Encode(CMP_EAX_EBX2);
        Encode(JE_FAR1); // If stack had zero, do a jump
        Encode(JE_FAR2);
        unsigned char * addressToFillInLater = GetAddress();
        Encode(0); // the exact number of bytes to skip gets set later,
                   // when we know it!  Call SetOffset() to do that.
        return addressToFillInLater;
}

unsigned char *  InstructionsClass::Jump()
{
        Encode(JUMP_ALWAYS_FAR);
        unsigned char * addressToFillInLater = GetAddress();
        Encode(0); // the exact number of bytes to jump gets set later,
                   // when we know it!  Call SetOffset() to do that.
        return addressToFillInLater;
}
```

```
void InstructionsClass::SetOffset(unsigned char *
                                  codeAddress, int offset)
{
        *((int*)codeAddress) = offset;
}
```

Using these constants:

```
const unsigned char JE_FAR1 = 0x0f; // 4 byte jump
const unsigned char JE_FAR2 = 0x84; // 4 byte jump
const unsigned char JUMP_ALWAYS_FAR = 0xE9; // 4 byte jump
```

14. For help with debugging, or to just see what is going on, you can also add the method PrintAllMachineCodes(), and call it after code.Finish(), if that sort of thing excites you.

```
void InstructionsClass::PrintAllMachineCodes()
{
      for (int i=0; i<mCurrent; i++)
      {
      printf("HEX: %2x  Decimal: %3i\n", (int)mCode[i], (int)mCode[i]);
      }
}
```

15. Let's take a minute to make sure we've got our public and private sections correct. In Instructions.h, in the InstructionsClass declaration, ALL variables should be private. These helper methods should also be private:

```
      void Encode(unsigned char c);
      void Encode(int x);
      void Encode(long long x);
      void Encode(void * p);
      int * GetMem(int index);
      void PrintIntegerLinux64();
      void WriteMinusLinux64();
      void WriteSpaceLinux64();
      void Call(void * function_address);
      void PopPopComparePush(unsigned char relational_operator);
```

These methods are used in the next chapter, and should be public:

```
      InstructionsClass();
      void Finish();
      void Execute();
      void PushValue(int value);
      void PopAndWrite();
      unsigned char * GetAddress();
      void PushVariable(int index);
      void PopAndStore(int index);
      void PopPopDivPush();
      void PopPopAddPush();
      void PopPopSubPush();
      void PopPopMulPush();
```

```
void PopPopLessPush();
void PopPopLessEqualPush();
void PopPopGreaterPush();
void PopPopGreaterEqualPush();
void PopPopEqualPush();
void PopPopNotEqualPush();

void PopPopAndPush();
void PopPopOrPush();

unsigned char * SkipIfZeroStack();
unsigned char * Jump();
void SetOffset(unsigned char * codeAddress, int offset);
void PrintAllMachineCodes();
```

16. Add more tests as you see fit. When testing, a good rule of thumb is that every method and function in your code base should be called (or “exercised”) at least once. A better rule is that every branch of every if, else, while, etc. should also be exercised at least once. That may be excessive for now. Do what you reasonably can.

As you write tests in InstructionsTester.cpp, be careful to ensure that the contents of the stack must return to exactly empty. Otherwise, your program will crash.

9: The Code Generator

In this chapter, we integrate Instructions.h and Instructions.cpp from the last chapter into our Simple Compiler project.

Recall that to make our Interpreter work, every class that inherited from the base class Node had to be able to Interpret itself, and every class that inherited from ExpressionNode had to be able to Evaluate itself.

To make our Coder work, every class that inherits from Node must be able to Code itself, and every class that inherits from ExpressionNode must be able to CodeEvaluate itself, meaning write code that when executed leaves its integer value on the stack.

To start the process of interpreting, we called the Interpret method of the StartNode. To start the process of Coding, we'll call the Code method of the StartNode. But when that is done, nothing will yet be executed. Coding will just build an array of machine codes inside the InstructionClass that must then subsequently be executed by calling the Execute method of the InstructionClass.

At the end of this chapter, you should be able to Interpret, Code and Execute, or both.

Problems

1. Get your Simple Compiler project from past chapters to build in your Linux environment. Make sure you can Interpret from Linux. Add Instructions.h and Instructions.cpp from the last chapter into your Simple Complier Linux project, and make sure it still builds.

2. Add the pure virtual method Code() to your Node class, and the pure virtual method CodeEvaluate to your ExpressionNode class. All classes that derive from these (and get instantiated) will need to provide their own version, as the following problems will help explain. The pure virtual method of Node should look like this:

```
virtual void Code(InstructionsClass &machineCode)=0;
```

Similarly, to the ExpressionNode class declaration, add:

```
virtual void CodeEvaluate(InstructionsClass &machineCode)=0;
```

3. The implementation for StartNode should simply pass the call to its child node:

```
void StartNode::Code(InstructionsClass &machineCode)
{
	mProgramNode->Code(machineCode);
}
```

Do similarly for ProgramNode and BlockNode. StatementGroupNode should pass the call to all its StatementNode children.

4. To Code a DeclarationStatementNode, simply call DeclareVariable on its mIdentifierNode, same as the Interpret method does. However, when coding and executing, the SymbolTable won't be used in the same way as when interpreting. We only use the SymbolTable to keep track of variable declarations and to have it give us a unique (small) integer for every unique variable. The actual storage for the variables will be in mData, which is declared in Instructions.h.

5. To code an AssignmentStatementNode, have its mExpressionNode CodeEvaluate itself with:

```
mExpressionNode->CodeEvaluate(machineCode);
```

That will write code into the mCode array that when later executed will evaluate that expression (no matter how complicated or recursive) and leave its integer value on the stack. We then want to write code to move it from the stack to the appropriate variable slot (index) in mData. This returns the position of where we'll store that variable in mData:

```
int index = mIdentifierNode->GetIndex();
```

And this writes code that when later executed moves that integer from the stack to the appropriate position in mData:

```
machineCode.PopAndStore(index);
```

6. To code a CoutStatementNode, have its mExpressionNode CodeEvaluate itself, then call `machineCode.PopAndWrite()`

7. Coding an IfStatementNode is more challenging. First, have its mExpressionNode CodeEvaluate itself. That will leave a zero (false) or non-zero (true) integer on the stack. If non-zero we'll want to execute the mStatementNode, but if zero, skip it. Since we don't know now (at compile time) what mExpressionNode will evaluate to (at run time), we'll need to always code the mStatementNode, but write code before it that might jump over and skip it, depending on that value that gets put on the stack (at run time). These two lines should do it:

```
machineCode.SkipIfZeroStack(); // Maybe jump over mStatementNode
mStatementNode->Code(machineCode);
```

But how far do we skip? However many bytes mStatementNode->Code() takes, but we don't know how far that is when we code SkipIfZeroStack, because we haven't yet had a chance to write mStatementNode->Code() and measure it. So SkipIfZeroStack initially codes to skip zero bytes, but the correct number of bytes needs to be filled in later, after you've carefully measured it. How do you measure it, and how do you fill it back in? Here is the complete solution:

```
void IfStatementNode::Code(InstructionsClass &machineCode)
{
	mExpressionNode->CodeEvaluate(machineCode);
	unsigned char * InsertAddress = machineCode.SkipIfZeroStack();
	unsigned char * address1 = machineCode.GetAddress();
	mStatementNode->Code(machineCode);
	unsigned char * address2 = machineCode.GetAddress();
	machineCode.SetOffset(InsertAddress, (int)(address2-address1));
}
```

Notice that SkipIfZeroStack returns the address of where to later fill in how many bytes to jump. Then we measure the current coding address before and after coding mStatement. The difference is how many bytes need to be skipped if the expression evaluates to zero. The last line then sets that offset into the InsertAddress so SkipIfZeroStack will jump the right number of bytes if the stack integer is zero.

8. Coding a WhileStatementNode is even harder than coding an IfStatementNode. This table should help you work it out:

	GetAddress 1
Code the While's test expression	
Code SkipIfZeroStack	Save the InsertAddressToSkip
	GetAddress 2
Code the While's statement	
machineCode.Jump()	Save the InsertAddressToJump
	Get Address 3

Now you can set InsertAddressToSkip to Address3 – Address2.

Also set InsertAddressToJump to Address1 – Address3.

9. To CodeEvaluate an IdentifierNode, use the PushVariable method:

```
machineCode.PushVariable(this->GetIndex());
```

10. To CodeEvaluate an IntegerNode, use the PushValue method.

11. A PlusNode is supposed to add the results of its two children. So first, it needs to write code that when executed leaves those two integers on the stack. The PlusNode should then Pop those two integers, add them, then Push the result. Code to do that is as follows:

```
void PlusNode::CodeEvaluate(InstructionsClass &machineCode)
{
	mLeft->CodeEvaluate(machineCode);
	mRight->CodeEvaluate(machineCode);
	machineCode.PopPopAddPush();
}
```

12. Following the pattern of PlusNode, do likewise for MinusNode, TimesNode, DivideNode, the six relational nodes, AndNode, and OrNode.

13. To get all this to work, have your main function call CodeAndExecute, which does all this:

```
void CodeAndExecute(string inputFile)
{
	// Create scanner, symbol table, and parser objects.
	ScannerClass scanner(inputFile);
	SymbolTableClass symbolTable;
	ParserClass parser(&scanner, &symbolTable);

	// Do the parsing, which results in a parse tree.
	StartNode * root = parser.Start();

	// Create the machine code instructions from the parse tree
	InstructionsClass machineCode;
	root->Code(machineCode);
	machineCode.Finish();
	machineCode.PrintAllMachineCodes();

	// Execute the machine code instructions previously created
	machineCode.Execute();

	// cleanup recursively
	delete root;
}
```

14. For testing, start small. I would run this input file:

```
void main()
{
}
```

followed by this one:

```
void main()
```

```
{
	cout << 1000;
}
```

Then:

```
void main()
{
	int sum;
}
```

followed by:

```
void main()
{
	int sum;
	sum = 35;
}
```

followed by:

```
void main()
{
	int sum;
	sum = 35 + 400;
}
```

followed by:

```
void main()
{
	int sum;
	sum = 35 + 400;
	cout << sum;
}
```

You get the idea. Continue incremental testing until all functionality has been exercised. With incremental testing, if something doesn't work along the way, it's the thing you just added. Look closely at it and fix it. If you can't figure it out, ask the instructor or another student.

10: Enhancement Ideas

At this point, you should have a functioning Interpreter as well as a Compiler/Executor. But it only works on a very small subset of the C/C++ language. In this chapter's problems, we'll add various enhancements to make it just a little bit more like the C/C++ language. It will still be a "Simple Compiler," but you could continue to add features to make yours more fully C/C++ compatible, or to make it something completely different according to your own specifications.

Problems

1. Support chained cout statements when interpreting and compiling, including the special symbol "endl." That is, be able to support statements like:

```
cout << 10 << x+2 << endl << endl << x*3+1 << endl ;
```

Update the CoutStatementNode to have a std::vector of ExpressionNode pointers, instead of just one. Use the NULL pointer to indicate the presence of an endl. During ParserClass::CoutStatement, every “<<” token must be followed by an expression or an ENDL token. Insist that the COUT token is immediately followed by at least one set of “<<” with corresponding expression or ENDL. After that, peek ahead to check for “<<” or “;”. As long as there are more “<<” tokens, keep reading in a do-while loop.

Note that you will need to do a second peek after each “<<” to check if what’s next is an ENDL. If so, add NULL to the CoutNode’s vector of points. If not, Call Expression and add its return pointer to CoutNode’s vector of pointers. This is a lot of tricky code. Here is a regular expression for how cout works:

$$\texttt{COUT\_TOKEN (INSERTION\_TOKEN (expression | ENDL\_TOKEN))}^{1+}\ \texttt{SEMICOLON\_TOKEN}$$

In other words, a COUT_TOKEN must be followed “one or more times” by an INSERTION_TOKEN, which in turn must be followed by an expression or an ENDL_TOKEN. After “one or more of those” sets, it must end with a SEMICOLON_TOKEN.

To machine code this, you will need to write an InstructionsClass method called WriteEndlLinux64, following the same pattern as the existing method WriteSpaceLinux64.

2. Support += and -= style assignments. Base these off the AssignmentStatementNode, not PlusNode or MinusNode. Since they all start with an IDENTIFIER_TOKEN, ParserClass::Statement will initially call AssignmentStatement for all three. After calling Identifier(), peek ahead to check for “=”, “+=”, or “-=” tokens. Then if/else branch to make an AssignmentStatementNode, a PlusEqualsStatementNode, or a MinusEqualsStatementNode.

To “Code” a PlusEqualsStatementNode, have it push its own identifier’s index on the stack, then push its expression on the stack, then PopPopAddPush, then PopAndStore the stack back into its own identifier’s index.

3. (Optional) Support C++ style scoping. That is, every block of code should introduce a new scope. When exiting the block, all variables declared within that block should be removed from the symbol table.