

Rogue Programming Language

Mujeeb Asabi Adelekan

madelekan@mail.stmarytx.edu

CS6395 Comprehensive Project

Fall 2023

Contents

Overview	3
Problem/Issue	3
Background	3
Existing Products.....	5
The Proposed Product	7
Requirements.....	9
Developer Hardware and Software	9
User Hardware and Software.....	9
Functional Requirements.....	10
Design.....	12
Data Structures	12
Identifier Table.....	12
STM Main Memory	25
Software Modules.....	26
Interface Design	26
BNF Grammar for Rogue.....	30
Pointers	35
Enumeration Types	36
Associative Arrays	37
How to Modify	39
Installation	43
Compiling the System	43
Starting the System.....	51
Sample Sessions	55
Troubleshooting.....	67
References	68
Source Code	69
Rogue8Compiler.cpp.....	69
Rogue.h	256
STM.c.....	301

Overview

This document is a manual for the Rogue Programming Language. It will start with an overview of the Rogue product and will outline the requirements and design of Rogue. This will be followed by a step-by-step guide to installing the major modules of the Rogue project, a demonstration of Rogue's functionality through several sample sessions, and a troubleshooting guide for any problems the user may encounter.

This section will start with a discussion on how programming languages greatly aid software development and how a common hurdle of these languages often slows development. This is followed by a brief history of high-level programming languages and their development. Existing solutions to the problem statement are discussed, followed by a detailed description of the Rogue language and its intended users.

After the overview of the product, this section will end with a preview of the remaining content of this manual.

Problem/Issue

The problem that prompted the creation of Rogue is discussed in this subsection.

High-level programming languages enable us to speak to computers by allowing programmers to write instructions using a higher level of abstraction. An important concept that permits abstraction is the ability to break programs into smaller, comprehensive modules (i.e., loops, subprograms, and class declarations). However, most languages use syntax for group modules that may prove confusing for larger programs. For example, it becomes difficult to know which closing curly brace is tied to which opening curly brace in C++ and C#, and it becomes even harder to delimit modules in indentation-based languages like Python. Although most IDEs work to alleviate this issue, it would be helpful if the syntax of a language itself aided the programmer, thus improving the writability of the high-level language. This is the mission of the Rogue programming language.

Background

Here is an in-depth overview on the history of programming languages.

In 1945, German scientist Konrad Zuse conceptualized a way to express computations for his Z4 computer. Named *Plankakul*, Zuse's language may be the first description of a high-level programming language. Features of the language included single-bit, integer, and floating-point datatypes, the ability to declare arrays and define records (like C structs), an iterative "for" statement, a selection statement, and assertions. Zuse used his language to write algorithms for sorting, graph connectivity testing, numerical operations like the square root, syntax analysis, and even chess playing. However, despite the depth of Zuse's language project, *Plankakul* was developed in isolation, was never implemented, and its description was not published until 1972, nearly three decades after its conception.

Although *Plankakul* could not receive recognition in the computer science world at the time, there were many attempts to implement a way to write programs at a higher-level than machine code. These “pseudocodes”, a term which in this era refers to code that is marginally more abstract than machine code, are a stepping stone in the creation of the high-level languages we are familiar with. Examples include Short Code, a pseudocode developed by John Mauchly for the BINAC computer, which represented math expressions as a pair of 6-bit bytes, and the UNIVAC compiling system, a series of compilers spearheaded by Grace Hopper, which allowed for higher-level code to be expanded into machine code, much like macro expansion in assembly.

The introduction of the IBM 704 in the mid-1950s, which included floating-point and index instructions in hardware, prompted the development of the Fortran programming language. The developers of Fortran claimed that the language will combine the efficiency of hard-coding with the ease of interpretive pseudocode. The original 1957 release of Fortran allowed input and output formatting as well as subroutines. The language also included an if-statement and a do-statement to modify the flow of control. Unlike most modern languages, data types in Fortran were implicitly determined by the variable name. Fortran proved popular among IBM 704 programmers, with the language having an adoption rate of around 50% a year after release. Fortran would see many updates, with its second release introducing independent subroutine compilation, removing the need to recompile an entire program, and thus making larger programs more practical.

Fortran’s relationship with the IBM 704 was typical of the era. Most programming languages in the late 1950’s were machine-dependent, which complicated program sharing. In 1958, American and European developers worked on a joint project to create a machine-independent programming language. This language, ALGOL, generalized Fortran’s features to make a more flexible language that the developers hoped would be used to describe algorithms. ALGOL formalized the concept of explicit data typing, introduced block structure, or scoping, allowed subprogram parameters to be passed-by-name or value, allowed recursion, and had stack-dynamic arrays. ALGOL was also the first programming language with formally described syntax, as Backus-Naur form, or BNF, was created to describe the language during its design process.

Up until the late 1960s, the most popular programming languages were purely procedural. That is, the main building blocks of these languages were the statements, and eventually the subroutines. At the turn of the decade, the concept of object-oriented programming began to coalesce. In 1967, the SIMULA 67 language introduced the class construct, and thus the concept of data abstraction, to support simulation applications. Two years later, Alan Kay, whose work was inspired by SIMULA 67, believed that desktop computers will primarily be used by nonprogrammers in the near future. As such, Key argued, those computers will need to offer an interactive, intuitive user interface.

Kay created Dynabook as his idea for a graphical user interface. Dynabook simulates a desk that has sheets of paper, represented by windows, with the top sheet being prioritized, and the user

interacts through keystrokes and the touchscreen. To support and implement Dynabook, Kay developed the SmallTalk programming language. SmallTalk consists entirely of objects, and the languages uses an unconventional, message-based system in which messages are sent to an object so that the object's methods are executed. SmallTalk's support of the object-oriented paradigm and its promotion of graphical user interface design established a link between the former and the latter that persists to this day.

These are only a few examples of how programming languages evolved and the motivations behind their development. Many more languages have been developed for use in artificial intelligence (Lisp), business applications (COBOL), education (Pascal), national defense (Ada), and even music composition!

Existing Products

Here is a discussion of three languages developed to address issues that their designers felt were not being addressed by popular languages of the time.

Lisp

Lisp is the first functional programming language. Designed by John McCarthy and Marvin Minsky, Lisp supports linked list processing, which was a feature needed for early AI applications. In particular, Lisp introduced recursion, conditional expressions, dynamic storage allocation, and implicit deallocation to list processing.

As a functional language, all computations in Lisp are done by applying functions to arguments, removing the need for variables and assignment statements. Loops are unnecessary as well since repetition can be defined with recursion.

Lisp remains a popular language in AI applications. Two popular dialects of Lisp exist today: Scheme, a smaller version of Lisp designed by MIT that is tailored for teaching functional programming, and Common Lisp, a complex amalgamation of various Lisp dialects.

Below is an example Lisp Program in Robert Sebesta's *Concepts of Programming Languages* (11th edition).

```
; Lisp Example function
; The following code defines a Lisp predicate function
; that takes two lists as arguments and returns True
; if the two lists are equal, and NIL (false) otherwise
(DEFUN equal_lists (lis1 lis2)
  (COND
    ((ATOM lis1) (EQ lis1 lis2))
    ((ATOM lis2) NIL)
    ((equal_lists (CAR lis1) (CAR lis2))
     (equal_lists (CDR lis1) (CDR lis2)))
    (T NIL)
  )
)
```

Pascal

Pascal is a programming language designed to teach programming. Designed by Niklaus Wirth, it is based on Wirth's efforts on ALGOL's development team to create an update to ALGOL 60. The update Wirth helped create, later named ALGOL-W, was rejected for being too modest of an improvement. After Wirth left the ALGOL project, he created Pascal as a spiritual successor to ALGOL-W.

Pascal included many features of the nascent programming languages of its time, like user-defined data types, records, and the `case` selection statement. However, Pascal was simple by design to meet its educational purposes. This simplicity prompted the creation of various dialects, like Turbo Pascal and Object Pascal.

Pascal proved successful in its goal, as it was the most popular language used to teach programming from the mid-1970s to the late 1990s.

Below is an example Pascal program from Robert Sebesta's *Concepts of Programming Languages (11th edition)*, slightly altered and written in an online compiler.

```

1 - { Pascal Example Program
2 -   Input: An integer, listlen, where listlen is less than
3 -           100, followed by listlen-integer values
4 -   Output: The number of input values that are greater than
5 -             the average of all input values }
6
7 program pasex (input, output);
8 type intlisttype = array [1..99] of integer;
9 var
10    intlist : intlisttype;
11    listlen, counter, sum, average, result : integer;
12 begin
13   result := 0;
14   sum := 0;
15   readln (listlen);
16   if ( (listlen > 0) and (listlen < 100)) then
17     begin
18   { Read input into an array and compute the sum }
19   for counter := 1 to listlen do
20     begin
21       readln (intlist[counter]);
22       sum := sum + intlist[counter];
23     end;
24   { Compute the average using integer division }
25   average := sum div listlen;
26   { Count the number of input values that are > average }
27   for counter := 1 to listlen do
28     if (intlist[counter] > average) then
29       result := result + 1;
30   { Print the result }
31   writeln ('The number of values > average is: ', result)
32   end { of the then clause of if (listlen > 0 ... )
33   else
34     writeln ('Error-input list length is not legal')
35 end.

```

Swift

Swift is a programming language developed by Apple as a replacement for Objective-C, Apple's previous programming language. Swift, like Pascal, attempts to combine various elements from other programming languages into an easy-to-learn language.

Apple intends for Swift to have a lightweight syntax tailored for modern applications. One instance of this modernity is Swift's ability to support multiple languages and emoji by having its strings use UTF-8 encoding. In addition, Swift includes many safeguards against unsafe code. In Swift, memory is automatically managed, and Swift objects can never be null (`nil` in Swift) unless the object is explicitly declared as an “optional”.

Swift is interoperable with C++ and Objective-C, and the language is open-source, with a development community at <https://developer.apple.com/swift/>.

Below is example Swift code written in an Apple IDE, alongside “playgrounds” that show results as code is written (Image source: <https://www.pcmag.com/news/apples-swift-language-a-really-big-deal.>)

```

func didMoveToView(scene : SKScene,
delegate : SKPhysicsContactDelegate) {
    // ===== Blimp Control =====
    yOffsetForTime = { i in
        return 80 * sin(i / 10.0)
    }

    // ===== Scene Configuration =====
    // Set up balloon lighting and per-pixel collisions.
    balloonConfigurator = { b in
        b.physicsBody.categoryBitMask = CONTACT_CATEGORY
        b.physicsBody.fieldBitMask = WIND_FIELD_CATEGORY
        b.lightingBitMask = BALLOON_LIGHTING_CATEGORY
    }

    // Load images for balloon explosion.
    balloonPop = {...}.map {
        SKTexture(imageNamed: "explode_\$(\$0)")
    }

    // Install turbulent field forces.
    var turbulence = SKFieldNode.noiseFieldWithSmoothness(0.7,
                                                       animationSpeed:0.8)
    turbulence.categoryBitMask = WIND_FIELD_CATEGORY
    turbulence.strength = 0.21
    scene.addChild(turbulence)

    cannonStrength = 210.0

    // ===== Scene Initialization =====
    // Do the rest of the setup and start the scene.
    setupHeroScene(delegate)
    setupPanicScene(delegate)
    setupCannons(scene, delegate)
}

func handleContact(bodyA : SKSpriteNode,
bodyB : SKSpriteNode) {
    if (bodyA == hero) {
        bodyB.runAction(removeBalloonAction)
    } else if (bodyB == hero) {
        bodyA.runAction(removeBalloonAction)
    }
}

```

The Proposed Product

To address the problems of popular programming languages, the author has developed Rogue.

Rogue is a high-level, free-format programming language that prioritizes readability, writability, and reliability. Rogue enables the programmer to see exactly what kind of module is ending. For example, Rogue's main function ends with the keywords “end main”, for-loops end with the keywords “end for”, if-else branches end with the keywords “end if”, and so on. Rogue supports several structural flow-of-control statements, integer and Boolean datatypes, assertions, input statements, arithmetic and logical computations, and the ability to write subprogram modules such as procedures and functions.

The heartbeat of Rogue is the Rogue Compiler, which translates the code into a proprietary assembly language designed to run on STM, a virtual machine (VM) developed by Dr. Hanna that simulates the von Neumann architecture. Using the Rogue Compiler, programmers can write the logic of an application in the Rogue language, run the Rogue source file through the compiler, see logged information about the compilation, including any syntax errors, and then execute the compiled source file as an application process.

Rogue is designed to be used by application programmers, regardless of their level of experience. Rogue supports associative arrays, pointer variables, enumeration types, and four main data types (int, bool, char, and float). These functionalities will make Rogue suitable for traditional software development and will allow the programmer to be flexible in how they can store and operate on information.

The remainder of this manual discusses Rogue's functional and non-functional requirements, the design of Rogue's data structures, software modules, and interfaces, and Rogue's installation process, followed by sample sessions that demonstrate how various Rogue programs are compiled and executed. The manual ends with a troubleshooting section that outlines any problems the programmer may encounter along with possible solutions.

Requirements

This section lists the functional and non-functional requirements of the Rogue Compiler and STM.

Developer Hardware and Software

Here is the hardware and software that was used to develop the project.

The Rogue Compiler was developed in C++ in Visual Studio 2019 on a Windows 10 machine with the following hardware specs:

- Processor: AMD Ryzen 3700U with Radeon Vega Mobile GFX 2.30 GHZ
- Memory (RAM): 16 GB

Rogue source programs, used to test and debug the compiler, was written using Visual Studio Code.

Why the developer chose the hardware and software:

- Windows 10 and Visual Studio 2019 are the operating system and IDE the developer is most familiar with, respectively.
- Preliminary development for the Rogue Compiler was done in C++ using Visual Studio 2019.
- Visual Studio Code allows the developer to edit Rogue source programs and view listing files and assembly code in the same editing environment.

User Hardware and Software

Here is the hardware and software that is required to run this project.

Since the Rogue Compiler and STM are written in C++, the user must have a machine that can run a C++ console application and/or compile a C++ program. The vast majority of modern-day desktops and laptops running the most popular operating systems (i.e., Windows, MacOS, and Linux) are more than capable of running these applications.

To operate the Rogue Compiler and STM, the user must have a basic understanding of the English language and should be familiar with computers to the point where they can use a keyboard (or an equivalent means of input), and can launch, operate, and close applications with ease.

Functional Requirements

Here is a list of the functional requirements that were and were not met in this project.

This project is a continuation of my CS6375 Compilers project, which was extended on for CS6340 Advanced Software Engineering. Rogue, before its development for CS6395 Comprehensive Project, had the following features:

- Flow-of-control statements
 - Loops
 - While/Do-while loops
 - Mid-test do-while loops
 - For loops
 - Selection statements (if—else-if—else)
- Single-line and multi-line comments
- Print and input statements
- Variables and Constants
 - Integer, Boolean, Character, and Floating-point datatypes
- Assignment statements
- Arithmetic and Logical Expressions
 - Arithmetic Operators:
 - + (add), - (subtract), * (multiply), / (divide), % (modulo)
 - ^ ** (exponentiation)
 - abs (absolute value)
 - + - (positive and negative symbols)
 - ++ -- (preincrement and predecrement)
 - Logical Operations
 - or, nor, xor, and, nand, not
 - Comparison Operators
 - <, <=, ==, >, >=, !=
 - <> (greater or less than, i.e., not equal alternative)
- Subprogram Modules
 - Procedures and Functions
- Assertions
- Array declarations
- Pointer variables

Requirements Met

For my comprehensive project, the Rogue programmer can write the following **new** Rogue language features in a code editor:

- Pointer **dereferences**
- Enumerations
- Associative Arrays

The Rogue Compiler can:

- Identify tokens and parse elements (i.e., analyze the syntax) of the Rogue programming language.
- Perform the compilation of Rogue source code into STM assembly code.
- Raise error messages and quit compilation if a syntax error is encountered.
- Create a listing file annotated with compilation information, including errors and their source line.

Requirements Not Met

The following requirements were listed in the proposal but **were not met** in the final project. With each requirement is a hurdle during development that led to it not being met:

- File creation, input, output, and deletion
 - New system service requests for file I/O must be added to STM.
- Data structures that use pointers (stacks, queues, linked lists, trees, and networks)
 - A Rogue equivalent of a record or struct abstract data type must be developed.

Design

This section will discuss the design of the Rogue language. First, the data structures that will be used in the Rogue Compiler and STM Machine are discussed. Next, the compilation process of the Rogue Compiler will be discussed, with a flowchart showing the compilation process and the major software modules of that process. Afterwards, the interfaces of the major modules will be shown with a simple “Hello World”-type program being used for demonstration. Finally, the context-free syntax of the language will be described using a modified version of Backus-Naur Form, and this grammar will display how the functional requirements of Rogue were met. In particular, the new features that were added to Rogue will be highlighted and explained in detail.

Data Structures

This subsection will discuss the data structures in Rogue Compiler and the STM main memory.

Identifier Table

The Rogue Identifier Table is an array of structs (called IDENTIFIERRECORDs) which contain each identifier’s scope, data type, number of dimensions, identifier type, reference, and lexeme. In addition, to add support for pointer dereferencing and enumeration types, the identifier table includes two **new** fields: Contents Type and Enum Type.

- Contents Type: the datatype of the *contents* of a pointer variable (i.e., the datatype of the value that is stored at the address that the variable points to)
- Enum Type: the enumeration type of an enumeration variable, recorded next to that variable’s lexeme in parentheses.

The identifier table is updated during compilation to keep track of which identifiers are already defined in the current scope and how each identifier is referenced in STM assembly language.

The following 12 pages demonstrate how the identifier table updates during the compilation of three example programs: P.rog, E0.rog, and Demo3.rog.

P.rog: demonstrates the Contents Type field in the identifier table

```
$$ -----
$$ Mujeeb Adelekan
$$ P
$$ P.rog
$$ Testing pointer dereferencing
$* -----*$

$*****
$ Main function
$*****
main
$*** Data definitions ***$
int x, y;
ptr p, q;      $$ declares pointers named p

$*** Testing address-of operator *WORKS* ***
p = addr(x);
x = 7;

print(p, "\n");
print(x, "\n");
```

```
print("\n");

$$ *** Testing the dereference operator ***

y = cont(p); $$ y is now equal to 7
print(y, "\n");

print(cont(p), "\n"); $$ should also print 7

x = 299;
print(cont(p), "\n"); $$ should print 299

x = 1234;
print(cont(p), "\n"); $$ should print 1234
print("\n");

$*** Testing pointer to pointer assignment ***$
q = p;      $$ contents type needs to be transferred to q

$$ should print the same address
print(p, "\n");
print(q, "\n");

end main
```

Upon compilation of P.rog, the identifier table will contain the following information at the end of the main module:

Contents of identifier table at end of compilation of PROGRAM module definition							
#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	1	INTEGER		0	PROGRAMMODULE_VARIABLE	SB:0D28	x
2	1	INTEGER		0	PROGRAMMODULE_VARIABLE	SB:0D29	y
3	1	POINTER	INTEGER		PROGRAMMODULE_VARIABLE	SB:0D30	p
4	1	POINTER	INTEGER		PROGRAMMODULE_VARIABLE	SB:0D31	q

E0.rog: demonstrates the Enum Type field in the identifier table

```

season currentSeason;

currentMonth = Sep;
currentSeason = fall;

print(currentMonth, "\n");  $$ should print 8
print(currentSeason, "\n"); $$ should print 3

end main

```

Upon compilation of E0.rog, the identifier table will contain the following information at the end of the main module:

Contents of identifier table at end of compilation of PROGRAM module definition

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	1		0	PROGRAMMODULE_ENUMTYPE			month
2	1		0	PROGRAMMODULE_ENUMTYPE			season
3	1	ENUMTYPE	0	PROGRAMMODULE_VARIABLE	SB:0D28		currentMonth (month)
4	1	ENUMTYPE	0	PROGRAMMODULE_VARIABLE	SB:0D29		currentSeason (season)

```
=====
Demo3.rog: demonstrates how the identifier table updates upon entering and exiting different scopes
=====
```

```
$$-----
$$ Mujeeb Adelekan
$$ Demo #3
$$ Demo3.rog
$$-----
```



```
$**** Global variables ****$
int x, y;

$*****$
```



```
$$ P(in1, out1, io1, ref1):
$$   in1: input-only parameter (passed-by-value)
$$   out1: output-only parameter (passed-by-result)
$$   io1: input and output parameter (passed-by-result/value)
$$   ref1: passed-by-reference
$*****$
```



```
proc P(IN in1 : int, OUT out1 : int, IO io1 : int, REF ref1 : bool)
    $**** Local variable C1 ****$
    perm int C1 = 101;
```

```
$**** out1 = 1 + 101 ****$  
out1 = in1+C1;  
  
$**** io1 = 3 + 1 ****$  
io1 = io1+1;  
  
$**** ref1 = NOT false ****$  
ref1 = NOT ref1;  
end proc  
  
*****$  
$$ Main function  
*****$  
main  
  $$ declare the constant p1  
  perm int p1 = 1;  
  
  $$ declare values p2, p3, and p4  
  int p2, p3;  
  bool p4;  
  
  $$ assign values to p3 and p4  
  p3 = 3;  
  p4 = true;
```

```

$$ call the procedure
call P(p1, p2, p3, p4);

$$ print p1, p2, p3, p4
print("p1 = ", p1, "\n");
print("p2 = ", p2, "\n");
print("p3 = ", p3, "\n");
print("p4 = ", p4, "\n");

end main

```

Upon compilation of `Demo3.rog`, the identifier table will contain the following information at different points of compilation:

Contents of identifier table after compilation of global data definitions

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D0	x
2	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D1	y

Contents of identifier table after compilation of PROCEDURE module header

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D0	x
2	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D1	y
3	0		0	PROCEDURE_SUBPROGRAMMODULE	P		P
4	1	INTEGER	0	IN_PARAMETER		FB:0D0	in1
5	1	INTEGER	0	OUT_PARAMETER		FB:0D2	out1
6	1	INTEGER	0	IO_PARAMETER		FB:0D4	io1
7	1	BOOLEAN	0	REF_PARAMETER		@FB:0D5	ref1

Contents of identifier table after compilation of PROCEDURE local data definitions

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D0	x
2	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D1	y
3	0		0	PROCEDURE_SUBPROGRAMMODULE	P		P
4	1	INTEGER	0	IN_PARAMETER		FB:0D0	in1
5	1	INTEGER	0	OUT_PARAMETER		FB:0D2	out1
6	1	INTEGER	0	IO_PARAMETER		FB:0D4	io1
7	1	BOOLEAN	0	REF_PARAMETER		@FB:0D5	ref1
8	1	INTEGER	0	SUBPROGRAMMODULE_CONSTANT		FB:0D8	C1

Contents of identifier table at end of compilation of PROCEDURE module definition

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D0	x
2	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D1	y
3	0		0	PROCEDURE_SUBPROGRAMMODULE	P		P
4	1	INTEGER	0	IN_PARAMETER		FB:0D0	
5	1	INTEGER	0	OUT_PARAMETER		FB:0D2	
6	1	INTEGER	0	IO_PARAMETER		FB:0D4	
7	1	BOOLEAN	0	REF_PARAMETER		@FB:0D5	

NOTICE: when a subprogram scope has been exited by the compiler, all the local variables and constants in that scope are removed from the table. However, the subprogram's formal parameters remain. This is so references to the subprogram can be compiled. **The identifiers are removed from the table to allow them to be used outside of the subprogram's scope.**

Contents of identifier table at end of compilation of PROGRAM module definition

#	Scope	Data type	Contents type	Dimensions	Type	Reference	Lexeme (Enum Type)
1	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D0	x
2	0	INTEGER	0	GLOBAL_VARIABLE		SB:0D1	y
3	0		0	PROCEDURE_SUBPROGRAMMODULE	P		P
4	1	INTEGER	0	IN_PARAMETER		FB:0D0	
5	1	INTEGER	0	OUT_PARAMETER		FB:0D2	
6	1	INTEGER	0	IO_PARAMETER		FB:0D4	
7	1	BOOLEAN	0	REF_PARAMETER		@FB:0D5	
8	1	INTEGER	0	PROGRAMMODULE_CONSTANT		SB:0D30	p1
9	1	INTEGER	0	PROGRAMMODULE_VARIABLE		SB:0D31	p2
10	1	INTEGER	0	PROGRAMMODULE_VARIABLE		SB:0D32	p3
11	1	BOOLEAN	0	PROGRAMMODULE_VARIABLE		SB:0D33	p4

STM Main Memory

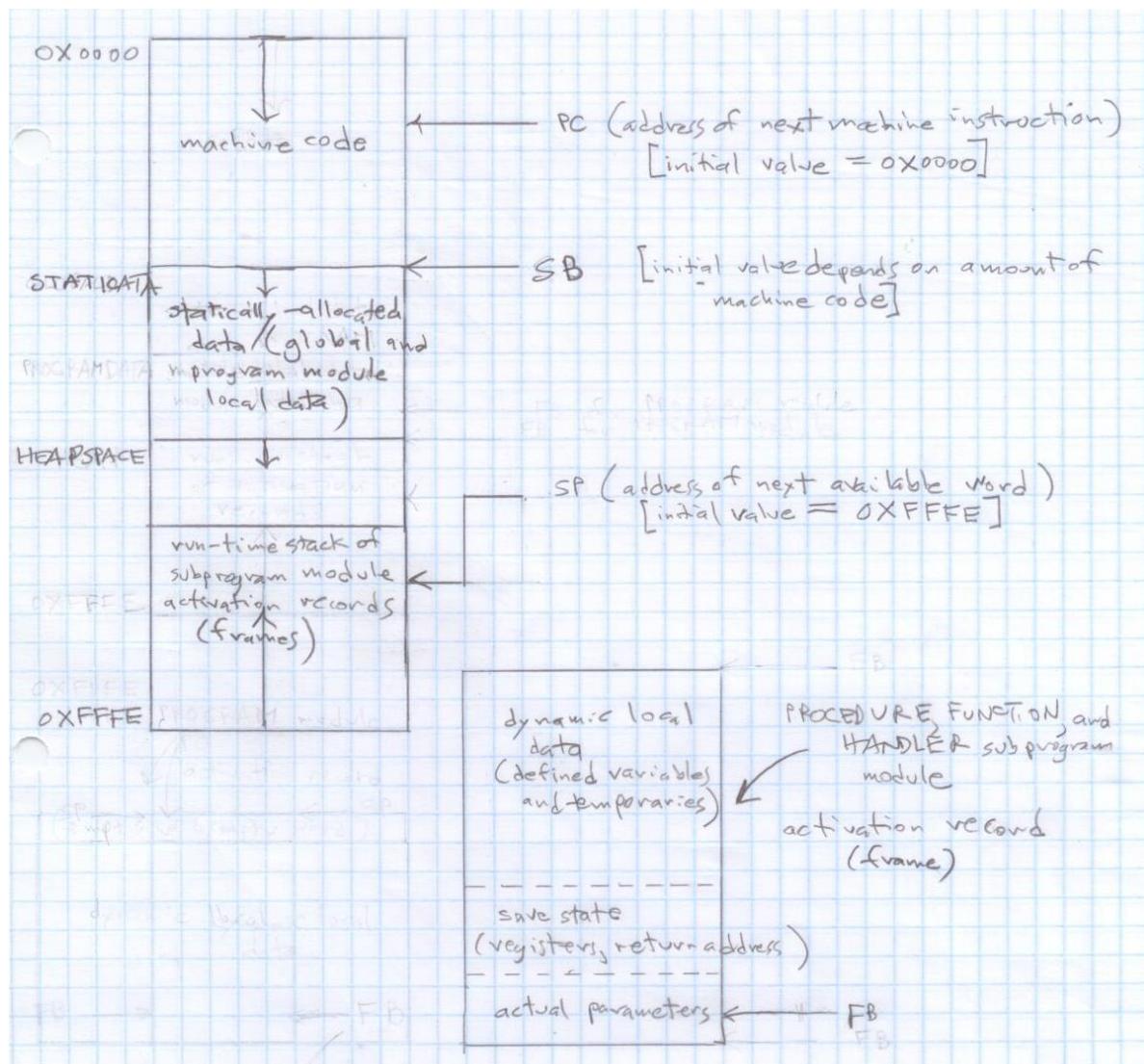
STM uses 16-bit addresses for its main memory (i.e., its address range is [0x0000, 0xFFFF]). Each address stores one byte, and each STM **word** is 2 bytes. STM words are stored in a big-endian manner, that is, the most significant byte in a word is stored first. For example, the word 0x12FE would be stored as:

Address 0x0000	Address 0x0001
12	FE

The runtime stack starts at address 0xFFFFE and grows downward (the addresses get *smaller*).

The static data record is stored after the machine code (which starts at address 0x0000) and grows upward (the addresses get *larger*).

Below is Dr. Hanna's map of STM's main memory.



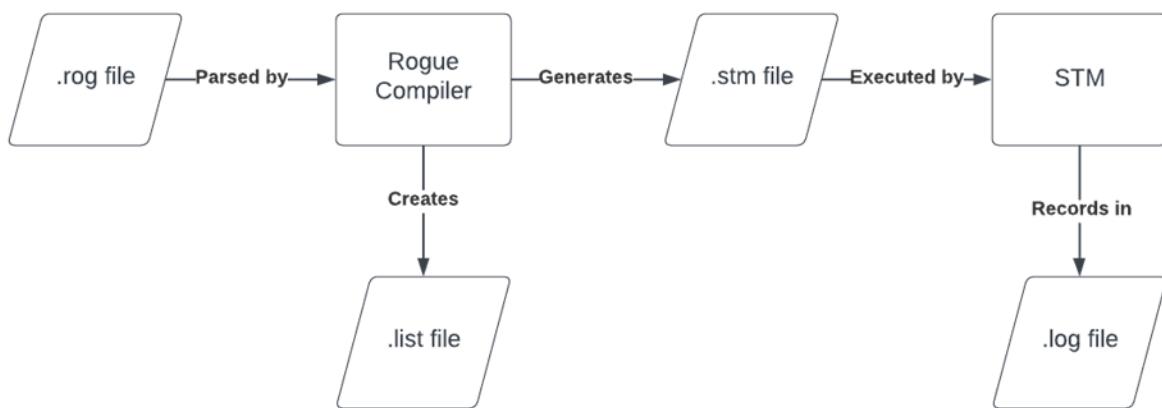
Software Modules

Here is a discussion of the major software modules that make up the Rogue project.

The Rogue compilation process consists of two major modules, both of which are console applications:

- (1) The **Rogue Compiler**, which compiles Rogue source programs and returns compilation information and errors to the programmer. Rogue Compiler is a **single-pass** compiler.
- (2) The **STM**, the target virtual machine developed by Dr. Hanna, simulates a von Neumann machine and can execute files written in its proprietary assembly code.

The following is a flowchart of how the Rogue compilation-to-execution process works:



The Rogue Compiler application asks for a Rogue source file, which must have the extension **.rog**. The compiler reads that source file and, given that there are no syntax errors, translates it into an STM source file, which has the extension **.stm**. The compilation process is recorded in a listing file with extension **.list**.

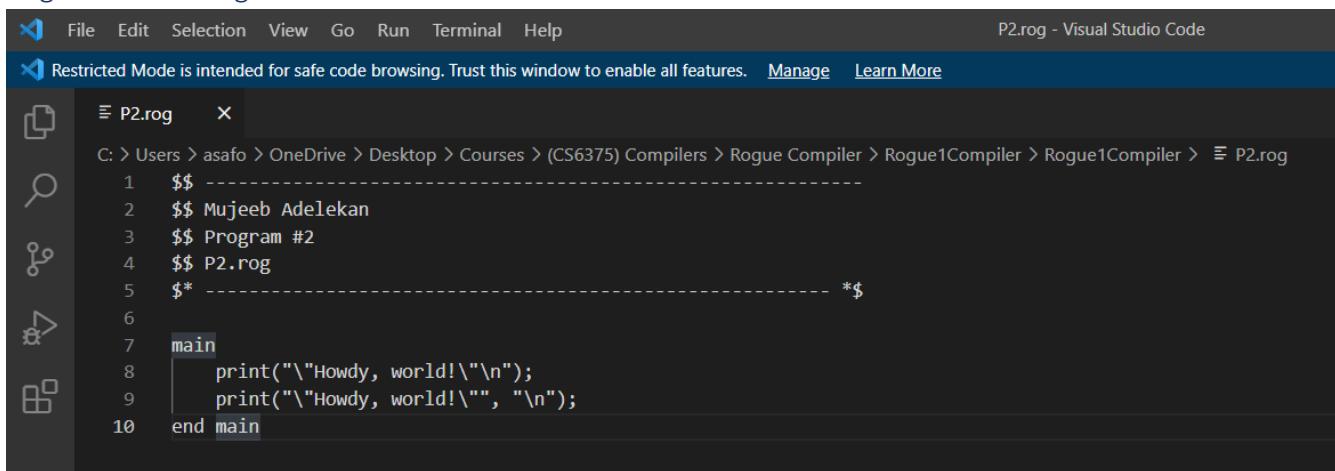
The user must then launch the STM application, which asks for the **.stm** source file. STM then reads that source file and executes the process until its termination. Information about the machine during execution is recorded in a **.log** file.

This fulfills all the Rogue Compiler requirements as well as the requirement of allowing the Rogue programmer to compile Rogue source programs into STM assembly code.

Interface Design

Here is an overview of the four major interface screens in the Rogue compilation process: the Rogue source program, the Rogue Compiler application interface, the STM assembly code, and the STM application interface.

Rogue Source Program



A screenshot of Visual Studio Code showing a file named "P2.rog". The code contains a main function that prints two lines of text. The code is as follows:

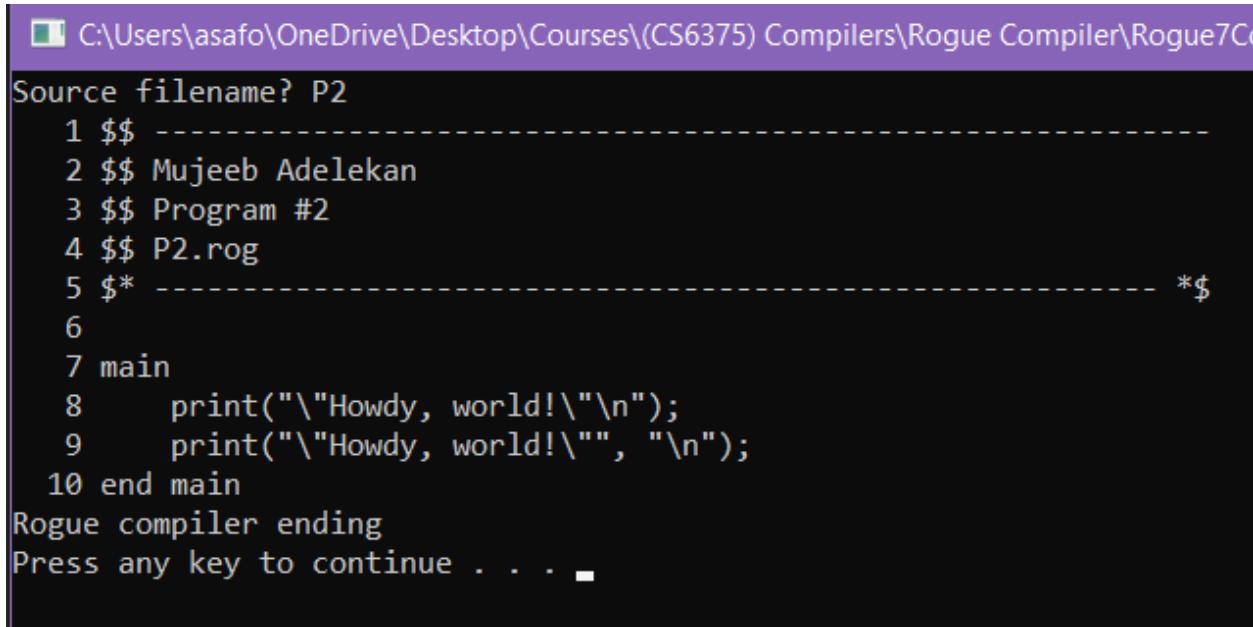
```

1  $$ -----
2  $$ Mujeeb Adelekan
3  $$ Program #2
4  $$ P2.rog
5  $$ *----- *$*
6
7  main
8      print("Howdy, world!\n");
9      print("Howdy, world!\n", "\n");
10 end main

```

A Rogue source program is any text file with the extension **.rog**. The programmer can write a Rogue source program in any text or code editor, such as Visual Studio Code. Above is an example Rogue program that demonstrates the main function, print statement, and comment system of Rogue.

Rogue Compiler



The terminal window shows the following interaction with the Rogue compiler:

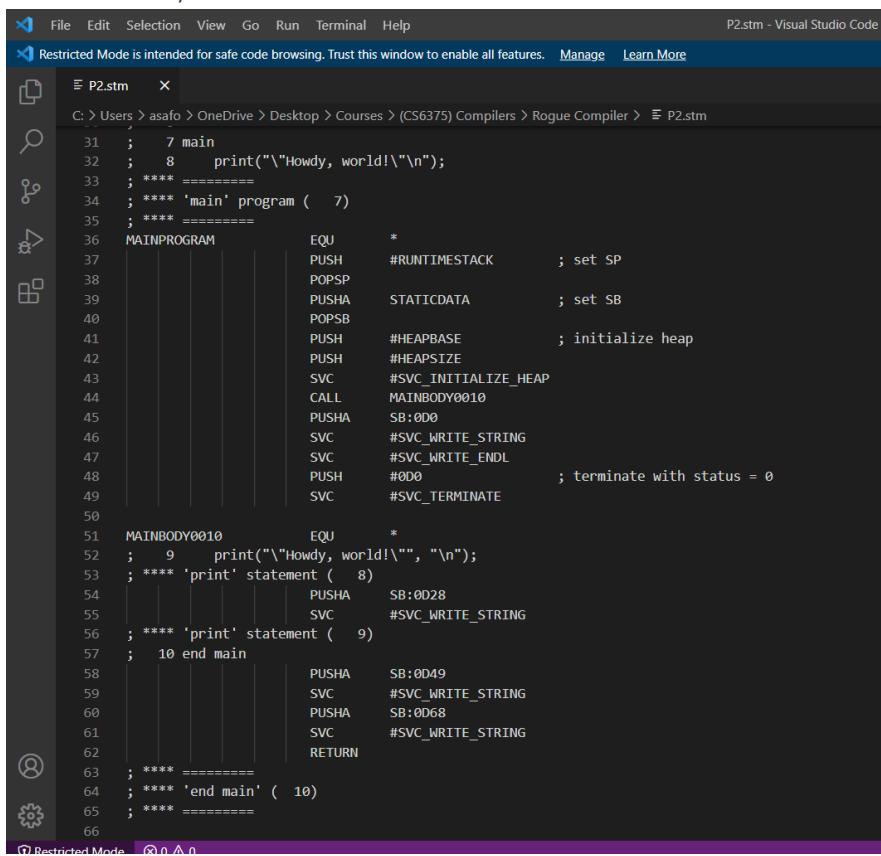
```

Source filename? P2
1 $$ -----
2 $$ Mujeeb Adelekan
3 $$ Program #2
4 $$ P2.rog
5 $$ *----- *$*
6
7 main
8     print("Howdy, world!\n");
9     print("Howdy, world!\n", "\n");
10 end main
Rogue compiler ending
Press any key to continue . . .

```

The compiler asks for the name of the .rog source file. When the user hits Enter (or Return on Mac), if the source file is found, the compiler will print each line of that file. If the source file contains a syntax error, the compiler will stop printing each line of the source file and will raise a compiler error. The compilation information along with any errors will be printed in the listing file created in the same directory of the compiler. If the file compiles correctly, the programmer can see the translated STM assembly code in the same directory as well.

STM assembly code



```

File Edit Selection View Go Run Terminal Help
P2.stm - Visual Studio Code
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More

P2.stm
C:\Users\asafot\OneDrive\Desktop\Courses\CS6375 Compilers\Rogue Compiler> P2.stm

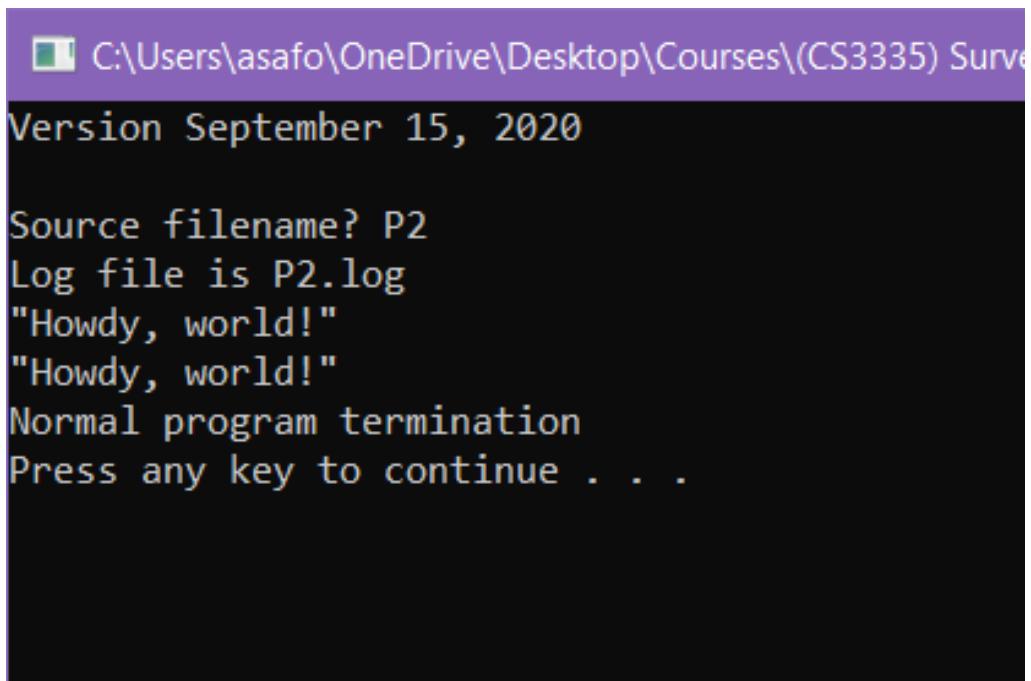
31 ;    7 main
32 ;    8     print("\"Howdy, world!\"\n");
33 ; **** =====
34 ; **** 'main' program ( 7)
35 ; **** =====
36 MAINPROGRAM EQU *
37 PUSH #RUNTIMESTACK ; set SP
38 POPSP
39 PUSH STATICDATA ; set SB
40 POPSB
41 PUSH #HEAPBASE ; initialize heap
42 PUSH #HEAPSIZE
43 SVC #SVC_INITIALIZE_HEAP
44 CALL MAINBODY0010
45 PUSH SB:0D0
46 SVC #SVC_WRITE_STRING
47 SVC #SVC_WRITE_EOL
48 PUSH #0D0 ; terminate with status = 0
49 SVC #SVC_TERMINATE
50
51 MAINBODY0010 EQU *
52 ;    9     print(""\Howdy, world!"", "\n");
53 ; **** 'print' statement ( 8)
54 ;        PUSH SB:0D28
55 ;        SVC #SVC_WRITE_STRING
56 ; **** 'print' statement ( 9)
57 ;    10 end main
58 ;        PUSH SB:0D49
59 ;        SVC #SVC_WRITE_STRING
60 ;        PUSH SB:0D68
61 ;        SVC #SVC_WRITE_STRING
62 RETURN
63 ; **** =====
64 ; **** 'end main' ( 10)
65 ; **** =====
66

```

Restricted Mode 0.0 ▾ 0

The STM assembly code is generated by the Rogue Compiler and can be viewed in any text or code editor. This assembly code is designed to run on the STM target machine.

STM (running the code)



The screenshot shows a terminal window titled 'C:\Users\asafo\OneDrive\Desktop\Courses\CS3335 Survey' with a purple header bar. The window displays the following text:
Version September 15, 2020
Source filename? P2
Log file is P2.log
"Howdy, world!"
"Howdy, world!"
Normal program termination
Press any key to continue . . .

The user is prompted to enter the name of the STM source file to be run. Upon hitting Enter/Return, STM will run the code then terminate. After termination, STM will create a log file that contains information about the machine during execution.

BNF Grammar for Rogue

Here is a listing of the context-free syntax of Rogue, described using Backus-Naur form, or BNF.

Note the BNF conventions used is a modified version developed by Dr. Hanna, which follows the following rules (as written by Dr. Hanna):

- (1) $\langle x \rangle$ means “ x is a non-terminal symbol”
- (2) $::=$ means “is defined as”
- (3) $|$ means “or”
- (4) $\{x\}^*$ means “ x repeated zero or more times”
- (5) $\{x\}_n$ means “ x repeated exactly n times”
- (6) $[x]$ means “ x is optional”
- (7) an underlined metasymbol like $[$ and $]$ means “[and] should be treated as terminal symbols”
- (8) $((x_1 | x_2 | \dots | x_n))$ means “choose exactly 1 from the list of alternates x_1, x_2, \dots, x_n ”
- (9) $||$ introduces a BNF comment that extends to end-of-line.

The full BNF grammar of Rogue is shown on the following four pages. The Rogue compiler enforces this grammar using a recursive-descent parser, where each nonterminal corresponds to a function in which the compiler expects to read the terminal symbols in the order listed in the grammar. If the compiler determines that the syntax is not being followed, a descriptive error will be returned to the programmer and the compilation process will be terminated.

This meets the requirement of allowing the programmer to write the language features of Rogue in a code editor and having the Rogue compiler parse those features. Note that due to the nature of BNF, only the context-free syntax is described. Context-dependent syntax of the new language features will be discussed following the BNF.

The non-terminals highlighted in yellow represent **updated** Rogue grammar, with the specific updates to the definitions highlighted as well. The non-terminals highlighted in cyan represent **new** Rogue grammar.

```

<RogueProgram>      ::= { <enumDeclaration> }*
                           { <dataDefinitions> }*
                           { (( <ProcedureDefinition> | <FunctionDefinition> )) }*
                           <MainProgram> EOPC

<enumDeclaration>   ::= enum : <enumType> { <enumConstant> {, <enumConstant> }* } ;

<enumType>          ::= <identifier>

<enumConstant>     ::= <identifier>

<dataDefinitions>   ::= <variableDefinitions> | <constantDefinitions> | <AssociativeArrayDefinition>

<variableDefinitions> ::= <datatype> <identifier> [ [ <integer> {, <integer> }* ] ]
                           {, [ <datatype> ] <identifier> [ [ <integer> {, <integer> }* ] ] };

<constantDefinitions> ::= perm <datatype> <identifier> = <literal>
                           {, [ <datatype> ] <identifier> = <literal>}* ;

<AssociativeArrayDefinition> ::= assoc <identifier> { <integer> };

<datatype>           ::= ( int | bool | char | float | ptr | <enumType> )

<MainProgram>        ::= main
                           { <enumDeclaration> }*
                           { <dataDefinitions> }*
                           { <statement> }*
                           end main

<ProcedureDefinition> ::= proc <identifier> [ ( <formalParameter> { , <formalParameter> }* ) ]
                           { <enumDeclaration> }*
                           { <dataDefinitions> }*
                           { <statement> }*
                           end proc

```

```

<FunctionDefinition> ::= func <identifier> : <datatype> ( [ <formalParameter> {, <formalParameter>}* ] )
    { <enumDeclaration> }*
    { <dataDefinitions> }*
    { <statement> }*
end func

<formalParameter> ::= [ (( in | out | io | ref )) ] <identifier> : <datatype>

<statement> ::= { <assertion> }*
    (( <PrintStatement>
        | <InputStatement>
        | <AssignmentStatement>
        | <IfStatement>
        | <DoWhileStatement>
        | <ForStatement>
        | <CallStatement>
        | <ReturnStatement>
    ))
    { <assertion> }*

<assertion> ::= assert( <expression> );

<PrintStatement> ::= print_ (( <string> | <expression> )) {, (( <string> | <expression> )) }* _ ;
<InputStatement> ::= input( [ <string> ,] <variable> );
<AssignmentStatement> ::= (( <variableList> | <AssociativeArrayReference> )) = <expression>;
<variableList> ::= <variable> {, <variable> }*

<IfStatement> ::= if ( <expression> )
    { <statement> }*
    { elif ( <expression> )
        { <statement> }* }*
    [ else
        { <statement> }* ]
    end if

```

```

<DoWhileStatement>    ::= [ do
                           { <statement> }*
                           while ( <expression> )
                           { <statement> }*
                           end while

<ForStatement>        ::= for <variable> = <expression> to <expression> [ by <expression> ]
                           { <statement> }*
                           end for

<CallStatement>       ::= call <identifier> [ ( ( <expression> | <variable> ))
                           { , (( <expression> | <variable> )) }* ) ];

<ReturnStatement>     ::= return [ ( <expression> ) ];

<expression>          ::= <conjunction> { (( or | nor | xor )) <conjunction> }*
<conjunction>         ::= <negation> { (( and | nand )) <negation> }*
<negation>            ::= [ not ] <comparison>
<comparison>          ::= <comparator> [ (( < | <= | == | > | >= | (( <> | != )) )) <comparator> ]
<comparator>         ::= <term> { (( + | - )) <term> }*
<term>                ::= <factor> { (( * | / | % )) <factor> }*
<factor>              ::= [ (( abs | + | - )) ] <secondary>
<secondary>           ::= <prefix> [ (( ^ | ** )) <prefix> ]
<prefix>              ::= <primary> || (( ++ | -- )) <variable>
<primary>             ::= <variable> | ( <expression> ) | <literal>
                           | <FunctionReference> | <AssociativeArrayReference> | <enumConstant>

<variable>            ::= <identifier> [ _ <expression> {, <expression> }* _ ]]

<FunctionReference>   ::= <identifier>([ <expression> {, <expression> }* ])

<AssociativeArrayReference> ::= <identifier> _ <expression> _ [as (( int | bool | char | float )) ]

```

```
<identifier>      ::= <letter> { (( <letter> | <digit> | _ )) }*
<literal>        ::= <integer> | <boolean> | <character> | <float> | <string> | addr(<variable>)
                      cont(<variable>)
<integer>        ::= <digit> { <digit> }*
<boolean>        ::= true | false
<character>       ::= '<ASCIICharacter>' || \ and " must be escaped
<float>          ::= <digit> { <digit> }* . <digit> { <digit> }* [e [-] <digit> { <digit> }* ]
<string>          ::= " { <ASCIICharacter> }* " || \ and " must be escaped
<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<comment>         ::= $$ { <ASCIICharacter > } * EOLC           || single-line
                      |$_* { (( <ASCIICharacter> | EOLC )) }* _$           || multi-line

<ASCIICharacter> ::= || Every printable ASCII character in range [ ' ', '~' ]
```

Pointers

Rogue allows programmers to declare, dereference, and assign addresses to pointers.

To declare a pointer, use the keyword `ptr` followed by an identifier. In Rogue, the programmer does not have to declare the datatype of the variable that the pointer points to.

To assign an address to a pointer, use the keyword `addr (<variable>)`, where `<variable>` is the variable that the programmer wants to take the address of.

To dereference a pointer, that is, access the contents of the address, use the keyword `cont (<variable>)`, where `<variable>` is the pointer that is to be dereferenced.

The following is Rogue code that declares two `int` variables, `x` and `y`, and declares two pointers `p` and `q`. `p` is then assigned with the address of `x`, and `x` is assigned with the value 6.

Afterwards, `p` (which points to `x`) is dereferenced. That is, the contents of the address stored in `p` is assigned to `y` (since `p` stores the **address of** `x`, the **value of** `x` is assigned to `y`, so the value of `y` is now 6). Finally, the address stored in `p`, which is the address of `x`, is assigned to the pointer `q`.

```

int x, y;

ptr p, q;           $$ declares pointers named p and q

p = addr(x);

x = 6;

y = cont(p);       $$ y is now equal to 6

q = p;             $$ q now points to the address of x

```

Enumeration Types

Rogue supports enumeration types, which are user-defined data types in which all possible values are listed (enumerated) in the definition. These values are called *enumeration constants* and are implicitly assigned integer values starting at 0.

To declare an enumeration type, use the keyword `enum` followed by a colon `:` and an identifier for the type's name. Then, declare all the enumeration constants of that type in a comma-separated list within curly brackets `{ }`. End the declaration statement with a semicolon `;`.

For example, the following Rogue statement declares an enumeration type `day` with its values being the seven days of the week:

```
enum : day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

An enumeration type can be used as a datatype in the scope it has been declared in. Remember that the values of an enumeration type are the set of constants listed in the definition.

The following Rogue code declares a `day` variable `weekStart` and a 5-element `day` array `weekDays`. The variable `weekStart` is assigned with the `day` value `Mon`, and the elements of `weekDays` are assigned with the `day` values `Mon` through `Fri`.

```
$*** Declare a variable called weekStart ***$
```

```
day weekStart;
```

```
$*** Declare an array called weekDays ***$
```

```
day weekDays[5];
```

```
$*** Assign weekStart with Monday ***$
```

```
weekStart = Mon;
```

```
$*** Assign the elements of weekDays with Mon-Fri ***$
```

```
weekDays[0] = Mon;
```

```
weekDays[1] = Tue;
```

```
weekDays[2] = Wed;
```

```
weekDays[3] = Thu;
```

```
weekDays[4] = Fri;
```

Associative Arrays

Rogue supports associative arrays. In Rogue associative arrays, each element is a **key-value pair** where the key serves as the unique index for its corresponding value. Rogue keys and values can be of type `int`, `bool`, `char`, or `float`.

To declare an associative array, use the keyword `assoc` followed by an identifier with curly braces `{ }` containing an `int` value representing the capacity of the associative array (i.e., the maximum number of key-value pairs the associative array will store). The following Rogue code declares an associative array with a capacity of 6 elements.

```
$***** Define an associative array with 6 (key, value) pairs
*****$  
assoc a_arr{6};
```

Adding an associative array element

To add a key-value pair to an associative array that has been declared in the current scope, write an assignment statement in the following format:

```
arrayName{ <key> } = <value> ;
```

where `<key>` and `<value>` are `int`, `bool`, `char`, or `float` expressions. The following Rogue code adds 6 key-value pairs to the associative array `a_arr`

```
$***** Add the hexadecimal digits A-F with their values in
decimal *****$  
a_arr{'A'} = 10;  
a_arr{'B'} = 11;  
a_arr{'C'} = 12;  
a_arr{'D'} = 13;  
a_arr{'E'} = 14;  
a_arr{'F'} = 15;
```

Modifying the value of a key-value pair

To modify the value of an existing element in an associative array, use the key that corresponds to that value when referencing that element on the left-hand side of an assignment statement. The following Rogue code modifies two values in `a_arr`:

```
$* this will replace the value of key 'B' with a float literal,
3.14 *$
```

```
a_arr{'B'} = 3.14;
```

```
$* this will replace the value of key 'E' with a bool literal,
false *$
```

```
a_arr{'E'} = false;
```

Retrieving the value for a given key

To retrieve the value of an associative array element for a given key, specify the name of the associative array followed by curly braces {} containing the key and use the type-cast operator as to cast the value to its appropriate datatype. The following Rogue code prints the integer value that corresponds to the key `A` in the associative array `a_arr`:

```
$*** Print the decimal value for the hex digit A ***$  
print("The demical value of A is ", a_arr{'A'} as int, "\n");
```

As another example, the following Rogue code assigns an `int` variable `x` with the value corresponding to the key `D` in `a_arr`. Assume that the variable `x` has been declared.

```
$*** Assign an int variable x with a_arr{'D'} (the int value 13)
***$  
  
x = a_arr{'D'} as int;  
print("The demical value of D is ", x, "\n");
```

How to Modify

This section includes instructions on how to modify the Rogue Compiler or STM. Note that this section assumes the user has access to the source files of both Rogue Compiler and STM and has used them to compile the two applications. In addition, the instructions assume the user is on a Windows 10 machine and uses Visual Studio 2019 to modify the source code of the major modules.

Modifying the Rogue Compiler

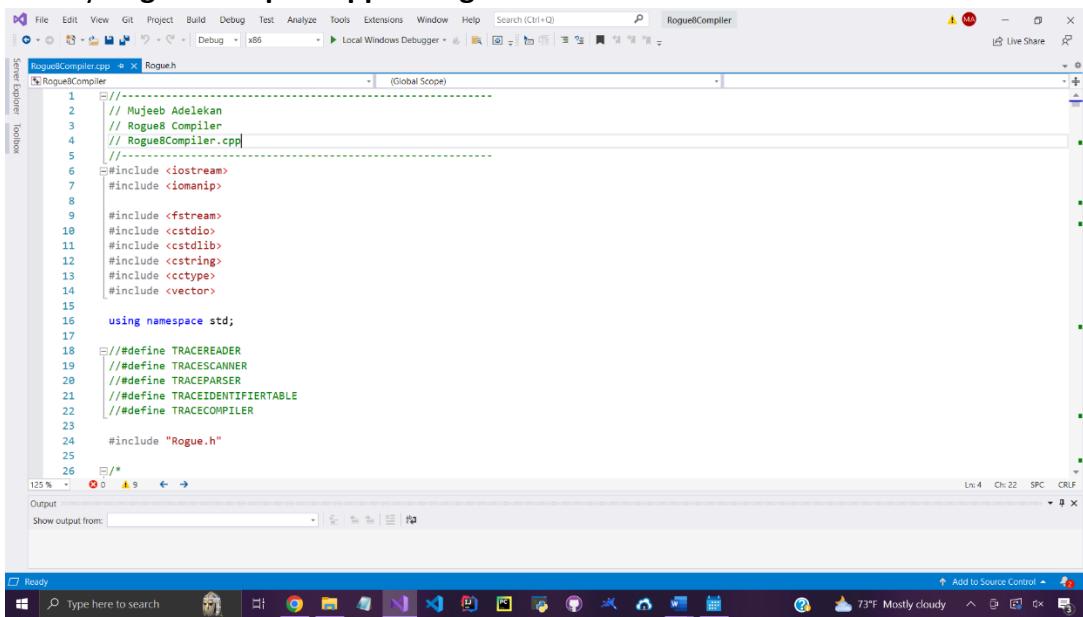
- 1) Open the *Rogue8Compiler* project folder.

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> <i>Rogue8Compiler</i>	⟳	5/11/2023 4:53 AM	File folder	
<input type="checkbox"/> <i>STM_Machine</i>	⟳	5/11/2023 4:47 AM	File folder	

- 2) Click on the *Rogue8Compiler* solution, *Rogue8Compiler.sln*, to open the solution in Visual Studio.

Name	Status	Date modified	Type	Size
<input type="checkbox"/> <i>Debug</i>	✓	5/2/2023 3:33 AM	File folder	
<input type="checkbox"/> <i>Rogue8Compiler</i>	⟳	5/11/2023 4:53 AM	File folder	
<input checked="" type="checkbox"/> <i>Rogue8Compiler.sln</i>	✓	4/25/2023 1:11 AM	Visual Studio Solut...	2 KB

- 3) Modify *Rogue8Compiler.cpp* or *Rogue.h*

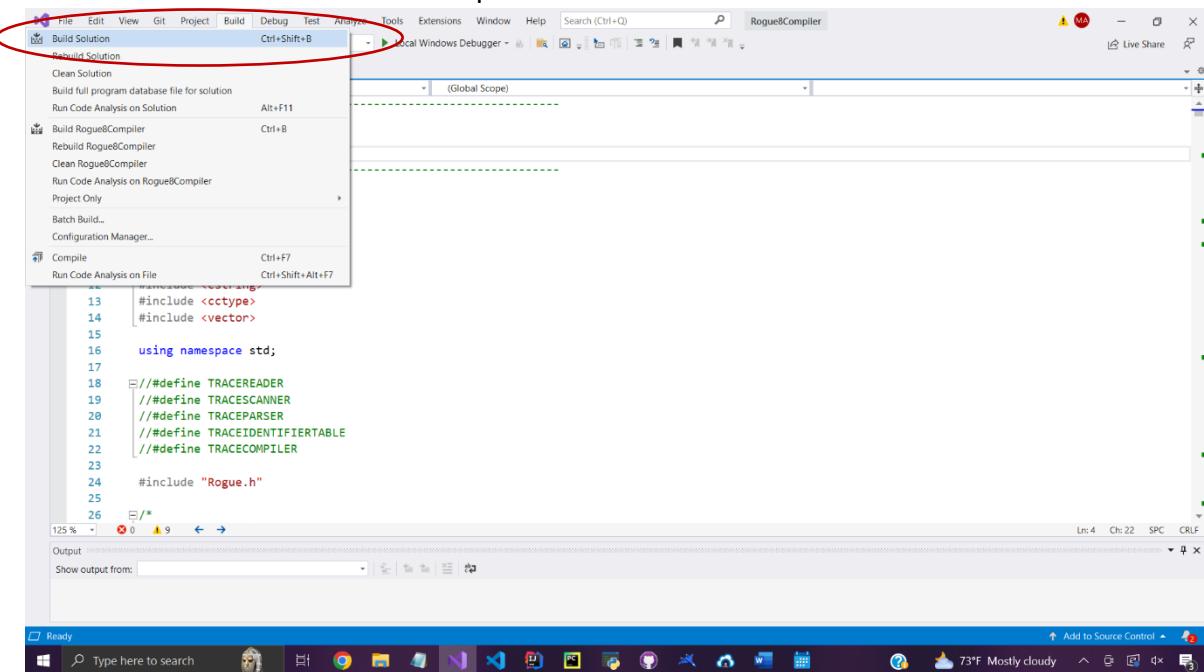


```

1 ///////////////////////////////////////////////////////////////////////////////
2 // Mujeeb Adelekan
3 // Rogue8 Compiler
4 // Rogue8Compiler.cpp
5 ///////////////////////////////////////////////////////////////////////////////
6 #include <iostream>
7 #include <iomanip>
8
9 #include <fstream>
10 #include <cstdio>
11 #include <cstdlib>
12 #include <cstring>
13 #include <cctype>
14 #include <vector>
15
16 using namespace std;
17
18 //define TRACEREADER
19 //define TRACESCANNER
20 //define TRACEPARSER
21 //define TRACEIDENTIFIERTABLE
22 //define TRACECOMPILER
23
24 #include "Rogue.h"
25
26 //*/

```

- 4) Go to *Build > Build Solution* to compile the modified code.



- 5) If there are no errors, open the *Rogue8Compiler* project folder and open the *Debug* folder. The newly compiled version of the Rogue Compiler application will be in the folder.

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Debug	✓	5/2/2023 3:33 AM	File folder	
<input type="checkbox"/> Rogue8Compiler	⟳	5/11/2023 4:53 AM	File folder	
<input type="checkbox"/> Rogue8Compiler.sln	✓	4/25/2023 1:11 AM	Visual Studio Solut...	2 KB

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Rogue8Compiler.exe	✓	5/5/2023 4:11 PM	Application	213 KB
<input type="checkbox"/> Rogue8Compiler.pdb	✓	5/5/2023 4:11 PM	Program Debug D...	5,644 KB

Modifying STM

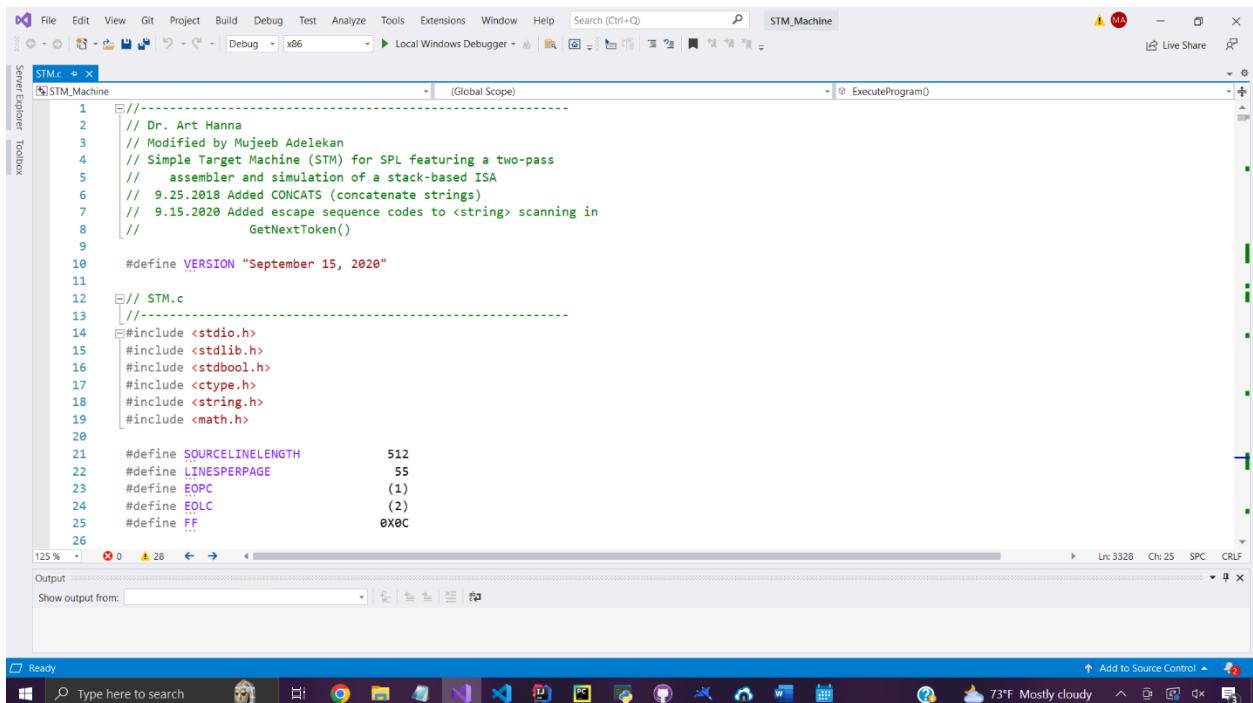
- 1) Open the *STM_Machine* project folder.

Name	Status	Date modified	Type	Size
Rogue8Compiler	⟳	5/11/2023 4:53 AM	File folder	
<input checked="" type="checkbox"/> STM_Machine	✓	5/11/2023 4:47 AM	File folder	

- 2) Click on the **STM_Machine** solution, **STM_Machine.sln**, to open the solution in Visual Studio.

Name	Status	Date modified	Type	Size
Debug	✓	5/11/2023 4:47 AM	File folder	
STM_Machine	✓	5/11/2023 4:47 AM	File folder	
<input checked="" type="checkbox"/> STM_Machine.sln	✓	8/22/2021 4:17 PM	Visual Studio Solut...	2 KB

- 3) Modify **STM.c**



```

1 // Dr. Art Hanna
2 // Modified by Mujeeb Adelekan
3 // Simple Target Machine (STM) for SPL featuring a two-pass
4 // assembler and simulation of a stack-based ISA
5 // 9.25.2018 Added CONCATS (concatenate strings)
6 // 9.15.2020 Added escape sequence codes to <string> scanning in
7 //           GetNextToken()
8 //
9
10 #define VERSION "September 15, 2020"
11
12 // STM.c
13 //-
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <stdbool.h>
17 #include <ctype.h>
18 #include <string.h>
19 #include <math.h>
20
21 #define SOURCELINELENGTH      512
22 #define LINESPERPAGE          55
23 #define EOPC                  (1)
24 #define EOLC                  (2)
25 #define FF                   0EXEC
26

```

The screenshot shows the Visual Studio interface with the STM_Machine solution selected in the Solution Explorer. The STM.c file is open in the main code editor window. The code itself is a C program for a simple target machine (STM). It includes comments about the author (Dr. Art Hanna and Mujeeb Adelekan), the purpose (a two-pass assembler and simulation of a stack-based ISA), and some specific defines like SOURCELINELENGTH and LINESPERPAGE. The code editor shows syntax highlighting for C keywords and comments. The status bar at the bottom indicates the current line (L1 3328), character (Ch: 25), and file type (SPC). The taskbar at the bottom shows various open application icons.

- 4) Go to *Build > Build Solution* to compile the modified code.

The screenshot shows the Microsoft Visual Studio 2020 interface. The menu bar is visible with options like File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help, and a search bar. The solution name 'STM_Machine' is in the title bar. The code editor window contains C-style code with several defines (e.g., SOURCELINELENGTH, LINESPERPAGE, EOPC, EOLC, FF) and includes (e.g., stdio.h, stdlib.h, stdbool.h, ctype.h, string.h, math.h). The status bar at the bottom right shows '2020'. The taskbar at the bottom of the screen also has the Visual Studio icon.

- 5) If there are no errors, open the *STM_Machine* project folder and open the *Debug* folder. The newly compiled version of the STM application will be in the folder.

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Debug	✓	5/11/2023 4:47 AM	File folder	
<input type="checkbox"/> STM_Machine	✓	5/11/2023 4:47 AM	File folder	
<input type="checkbox"/> STM_Machine.sln	✓	8/22/2021 4:17 PM	Visual Studio Solut...	2 KB

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> STM_Machine.exe	✓	5/5/2023 5:15 AM	Application	128 KB
<input type="checkbox"/> STM_Machine.pdb	✓	5/5/2023 5:15 AM	Program Debug D...	1,212 KB

Installation

This section goes over the installation process of the Rogue Compiler and STM. Note that all screenshots are taken on a Windows 10 machine.

Compiling the System

Note that these instructions assume the developer is using Visual Studio 2019 to build the applications.

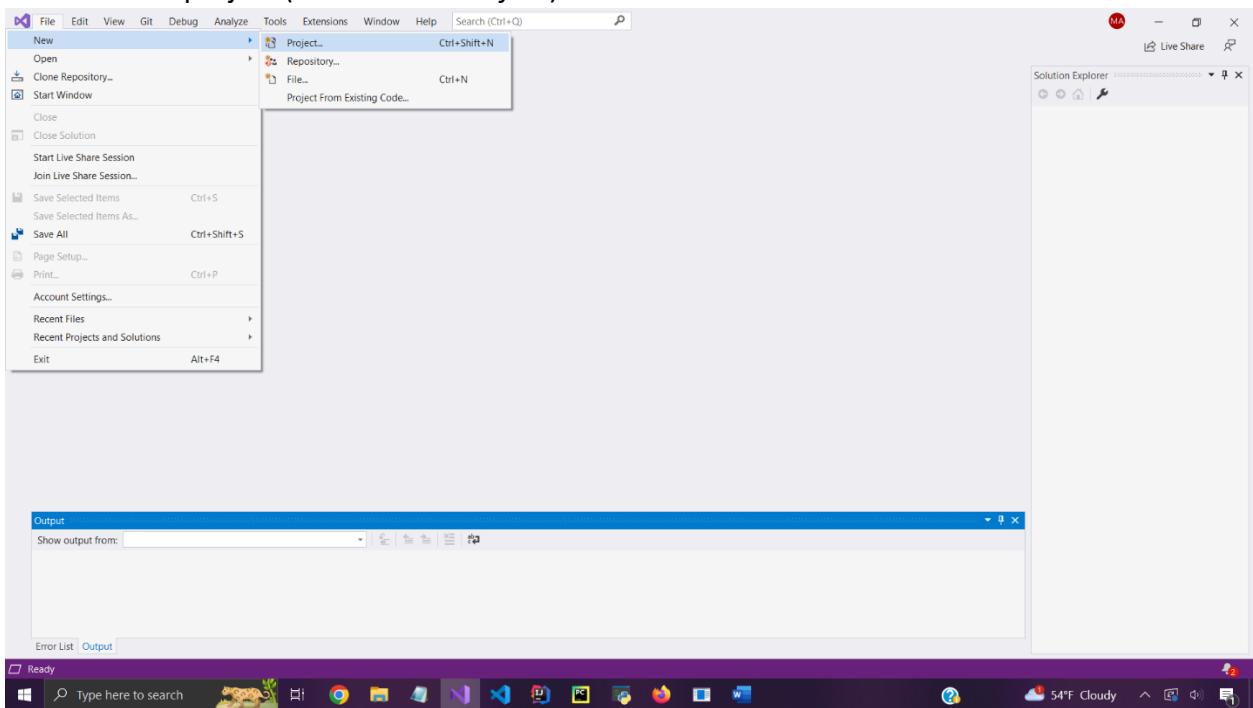
The source files required build the Rogue Compiler and STM are:

- Rogue Compiler
 - **Rogue<#>Compiler.cpp** (<#> refers to the version, the following instructions use version 8)
 - **Rogue.h**
- STM
 - **STM.c**

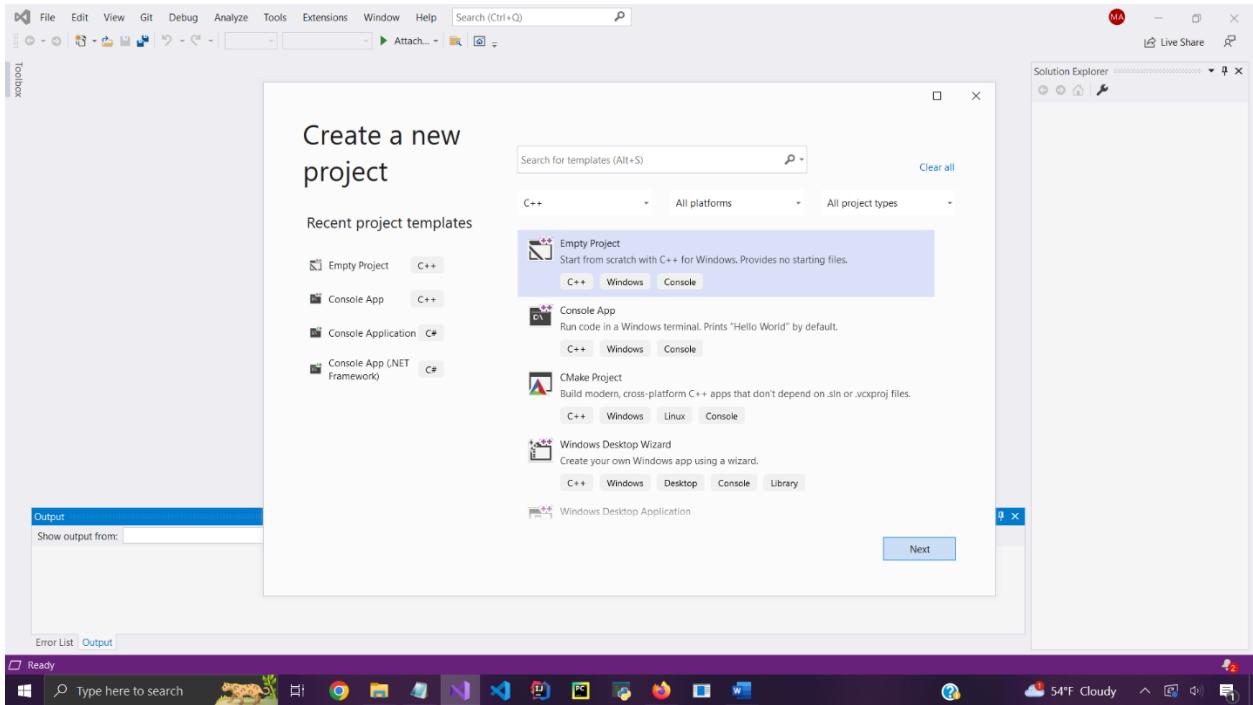
It is recommended that the Rogue Compiler and STM applications are placed in the same folder once they are built.

Building the Rogue Compiler

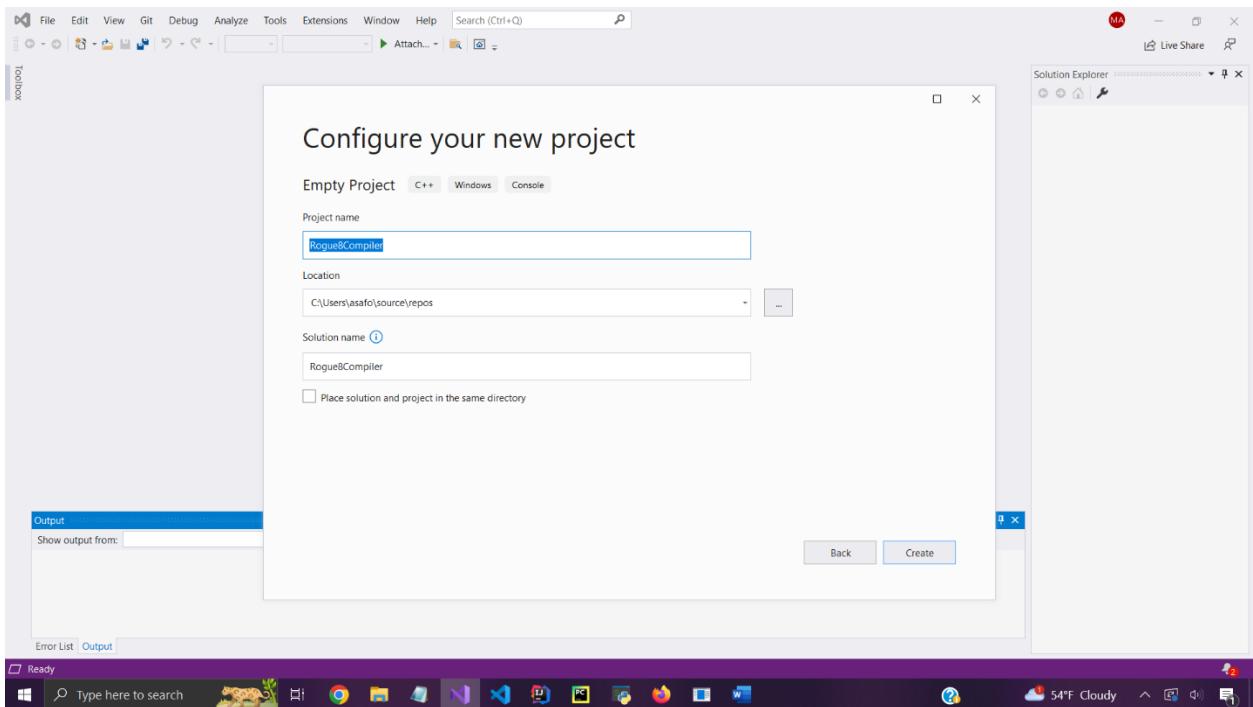
1) Create a new project (*File > New > Project*)



2) Use the “Empty Project” Template

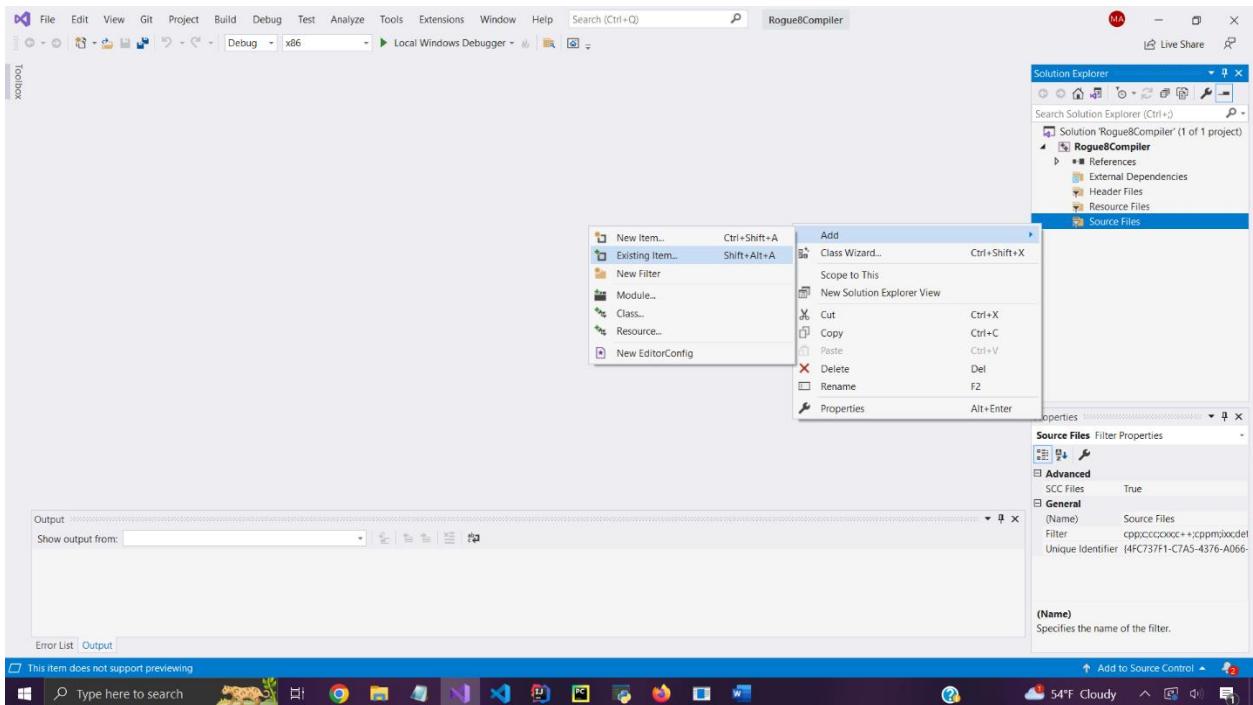
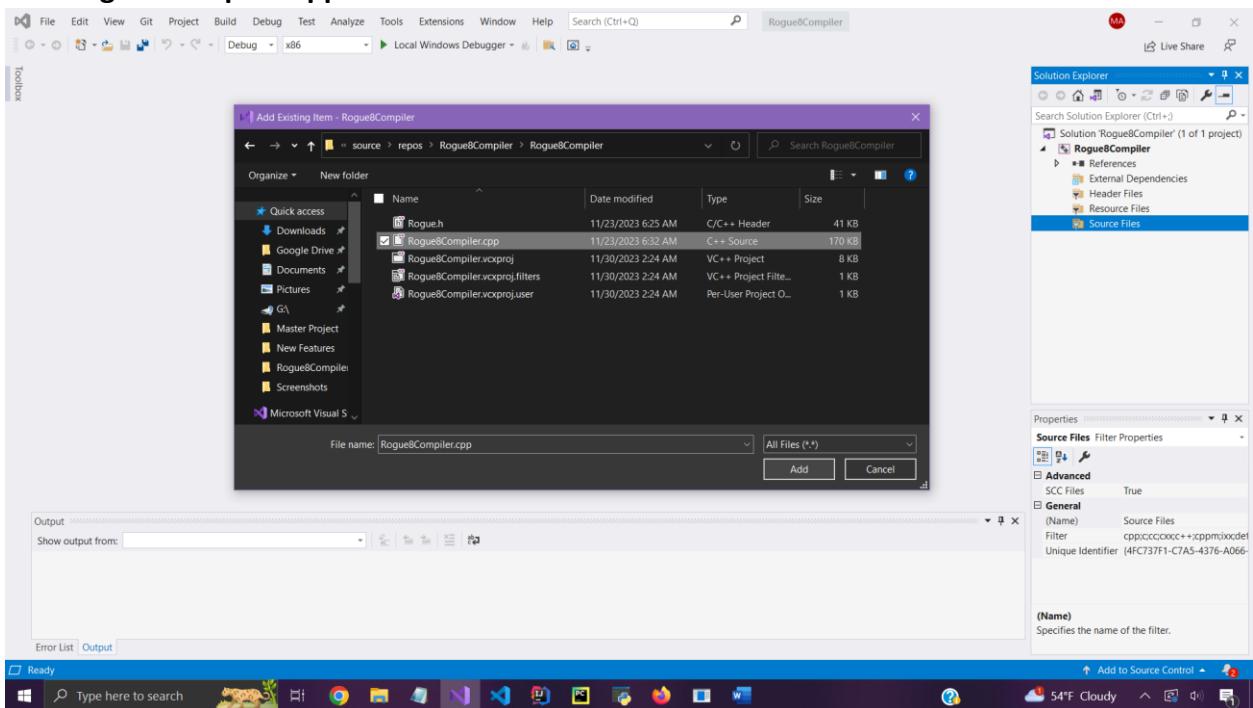


3) Title the project “Rogue8Compiler” (the solution will automatically have the same name)

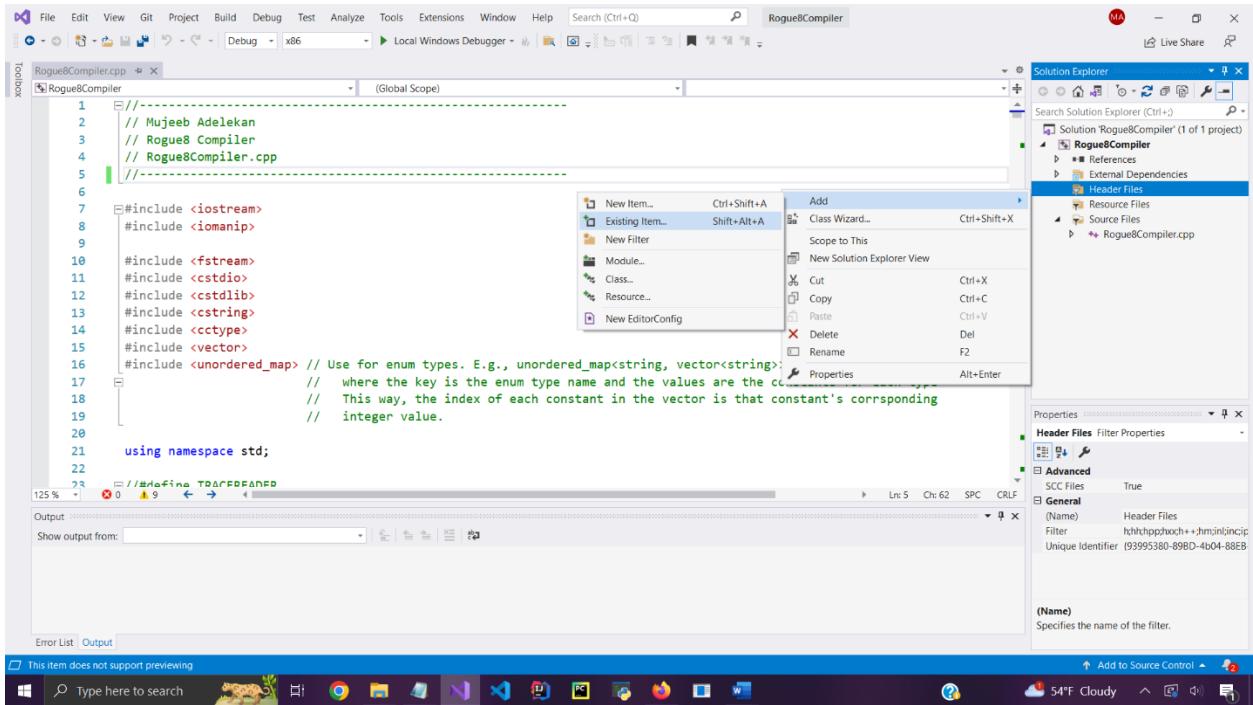


4) In the Solution Explorer, right click Source Files, then go to *Add > Existing Item*

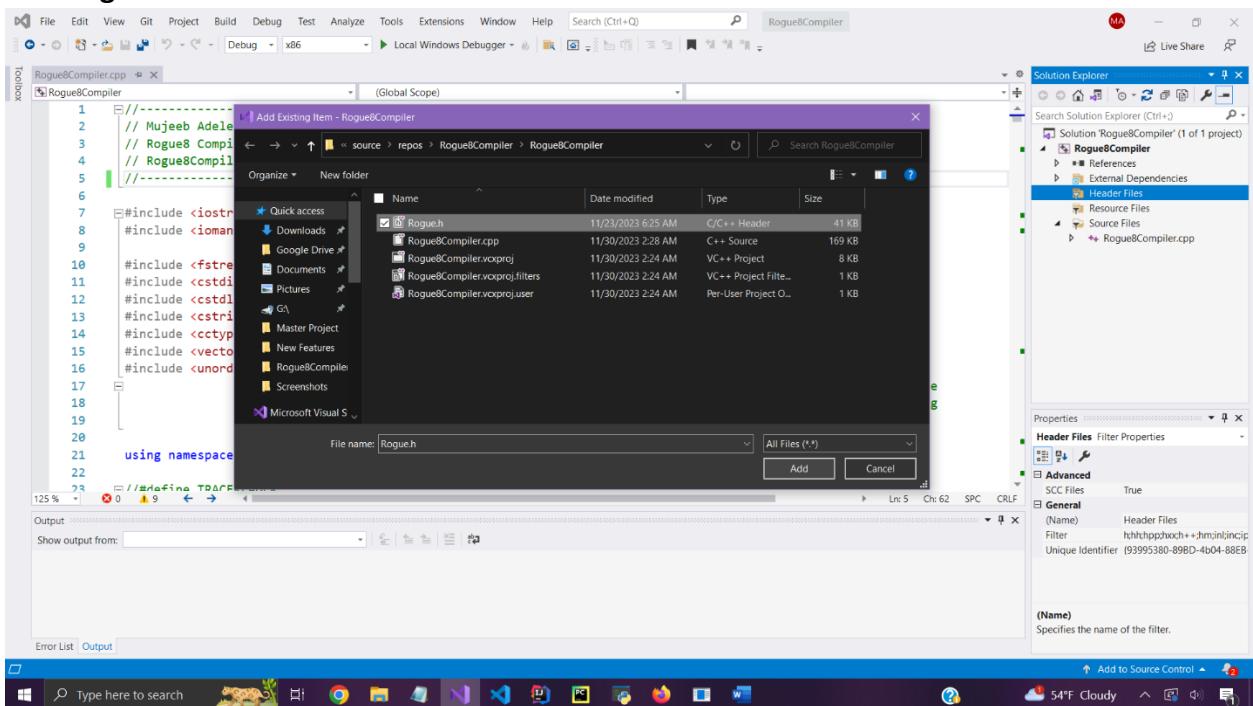
- If the Solution Explorer is not visible, go to *View > Solution Explorer*

5) Add **Rogue8Compiler.cpp**

6) In the Solution Explorer, right click *Header Files*, then go to *Add > Existing Item*

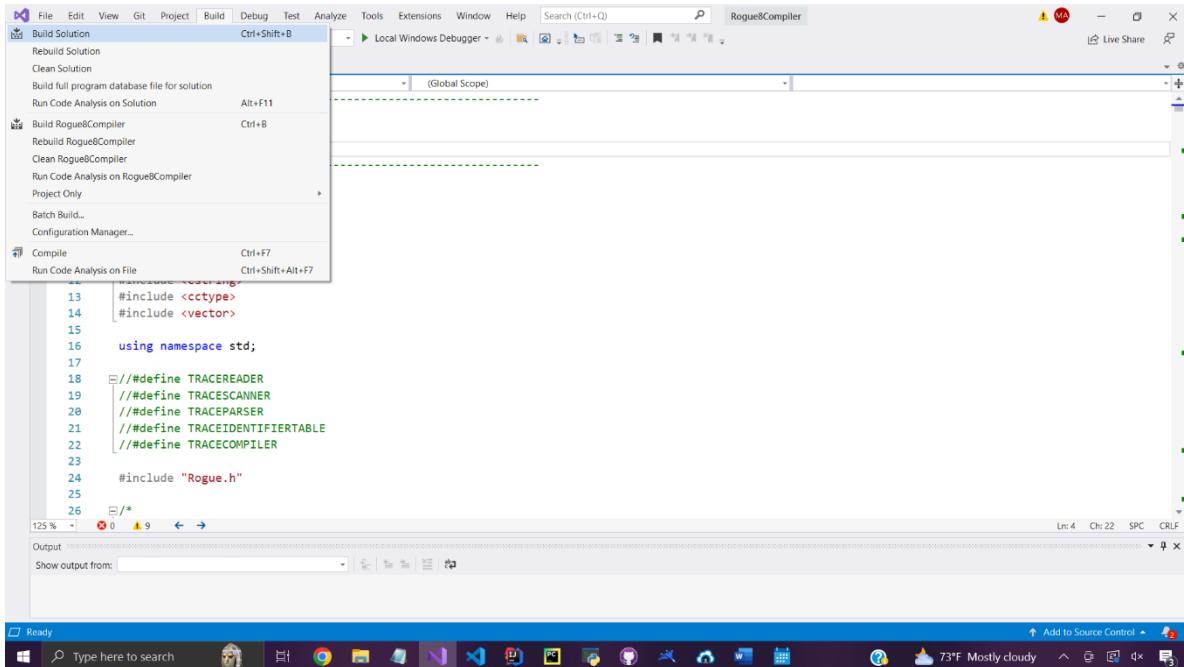


7) Add **Rogue.h**



8) Go to *Build > Build Solution*

- If the compiler warns that some functions are deprecated, go to *Project > Rogue8Compiler Properties > Configuration Properties > C/C++ > Preprocessor* and type `_CRT_SECURE_NO_WARNINGS` in the *Preprocessor Definitions* field.



The screenshot shows the Microsoft Visual Studio interface. The build menu is open, showing options like 'Build Solution', 'Rebuild Solution', and 'Build Rogue8Compiler'. The code editor window displays the following C++ code:

```

13 #include <cctype>
14 #include <vector>
15
16 using namespace std;
17
18 // #define TRACEREADER
19 // #define TRACESCANNER
20 // #define TRACEPARSER
21 // #define TRACEIDENTIFIERTABLE
22 // #define TRACECOMPILER
23
24 #include "Rogue.h"
25
26 /*
```

The status bar at the bottom indicates 'L: 4 Ch: 22 SPC CRLF'.

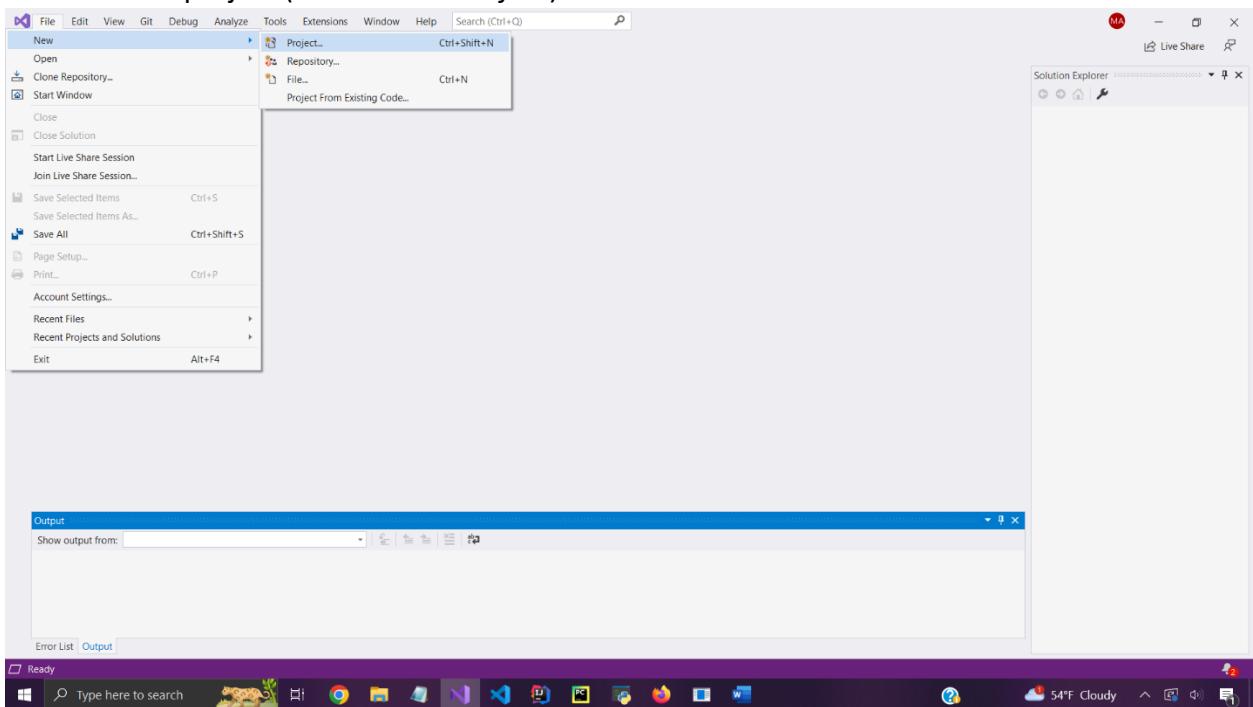
9) If there are no errors, open the *Rogue8Compiler* project folder and open the *Debug* folder. The Rogue Compiler application will be in the folder.

Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Debug	✓	5/2/2023 3:33 AM	File folder	
<input type="checkbox"/> Rogue8Compiler	⟳	5/11/2023 4:53 AM	File folder	
<input type="checkbox"/> Rogue8Compiler.sln	✓	4/25/2023 1:11 AM	Visual Studio Solut...	2 KB

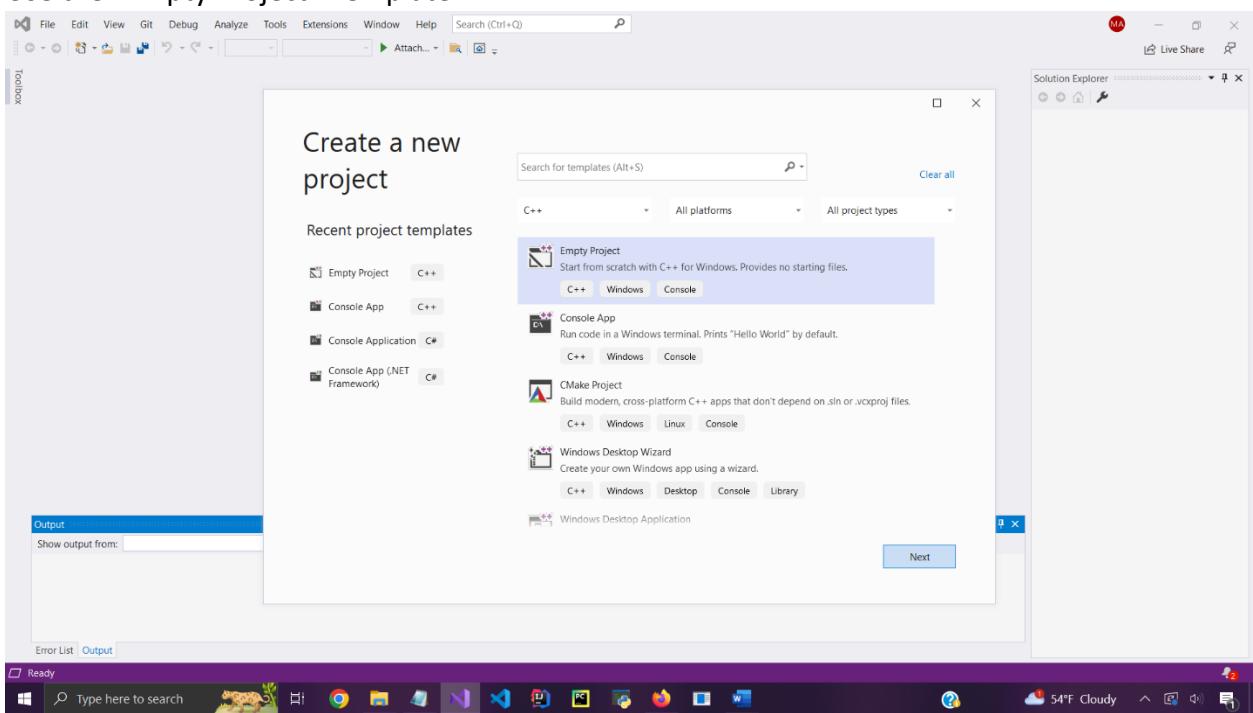
Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Rogue8Compiler.exe	✓	5/5/2023 4:11 PM	Application	213 KB
<input type="checkbox"/> Rogue8Compiler.pdb	✓	5/5/2023 4:11 PM	Program Debug D...	5,644 KB

Building STM

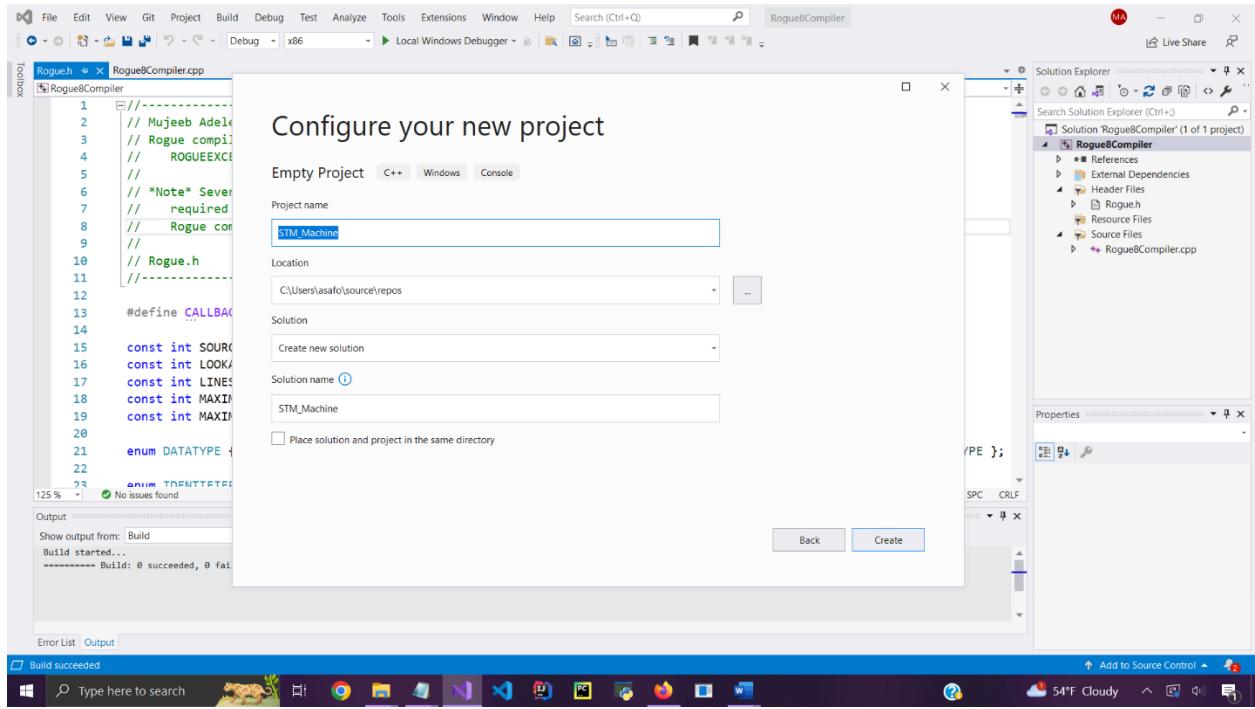
1) Create a new project (*File > New > Project*)



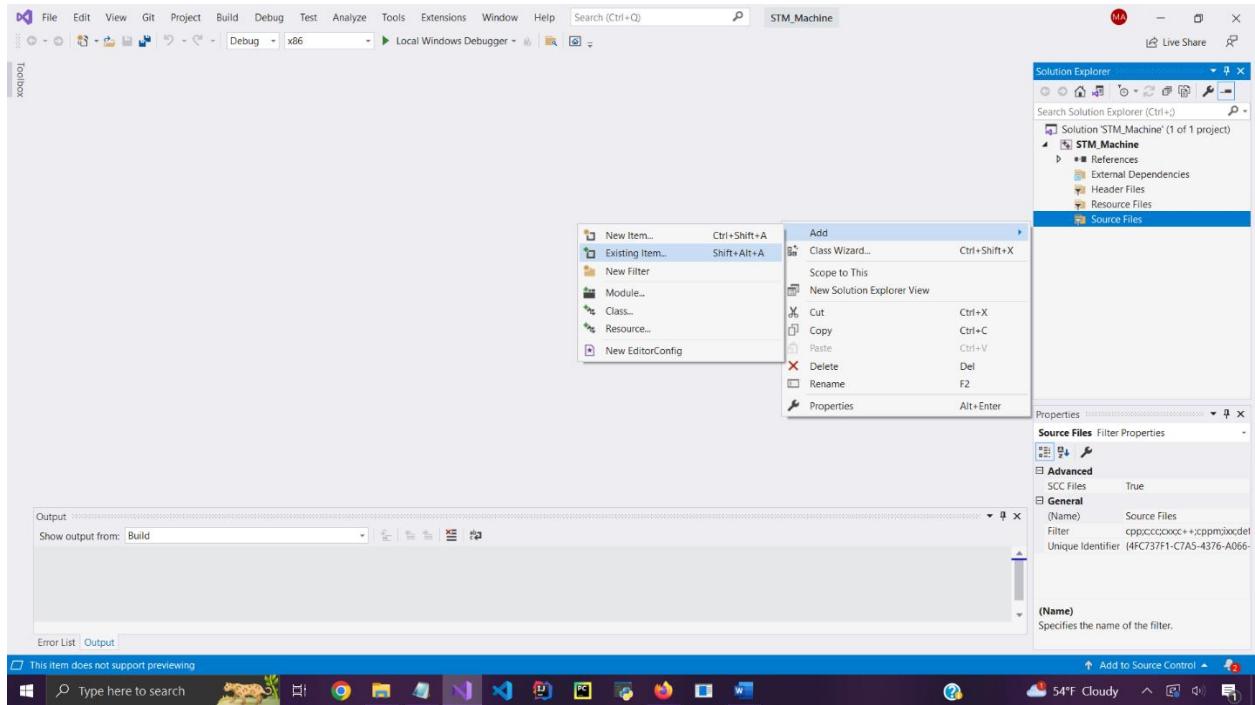
2) Use the “Empty Project” Template



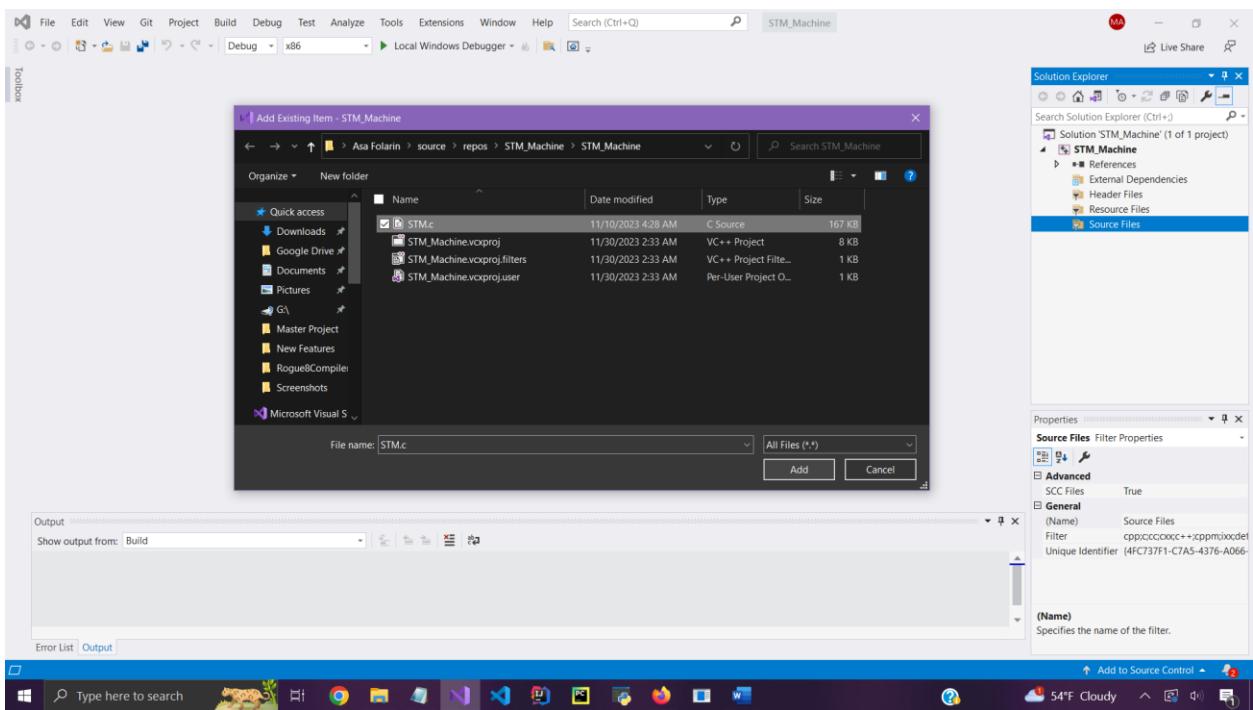
- 3) Title the project “STM_Machine” (the solution will automatically have the same name)



- 4) In the Solution Explorer, right click Source Files, then go to *Add > Existing Item*
- If the Solution Explorer is not visible, go to *View > Solution Explorer*

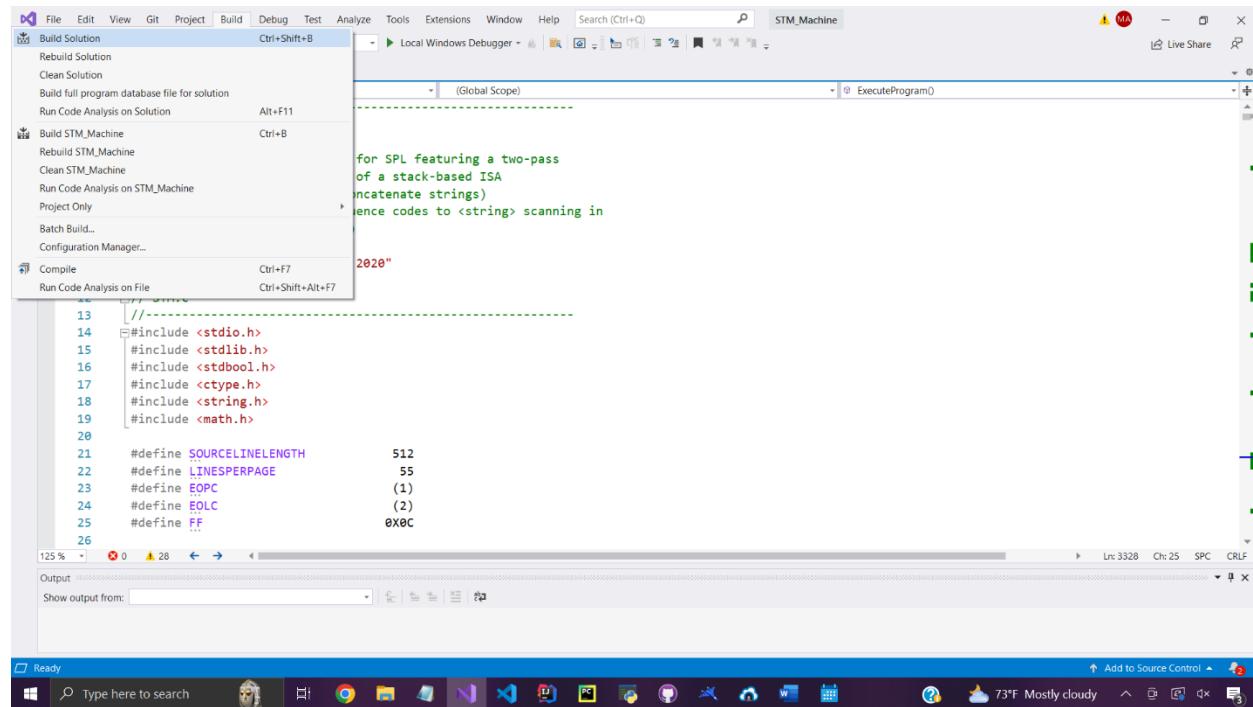


5) Add STM.c

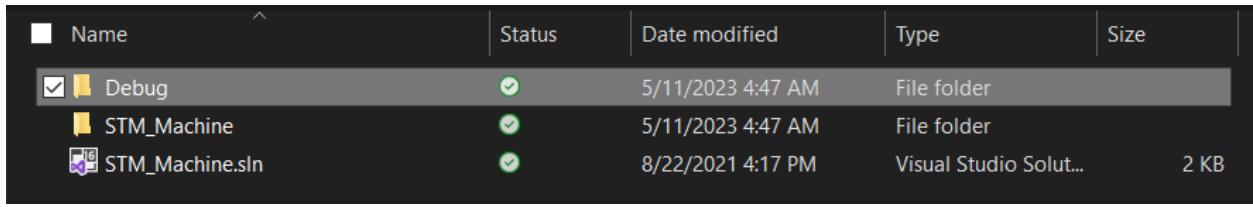


6) Go to Build > Build Solution

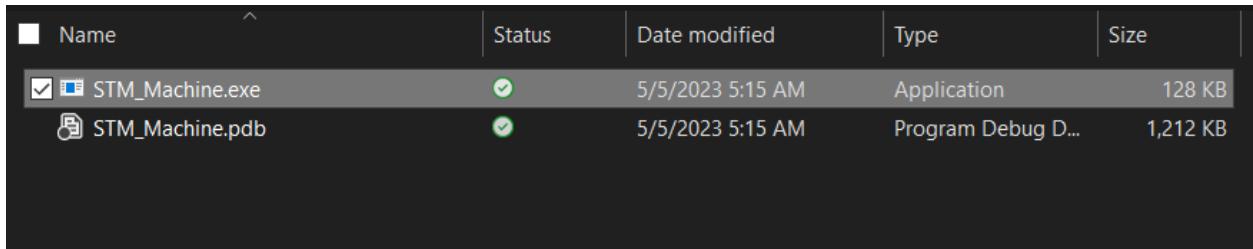
- If the compiler warns that some functions are deprecated, go to *Project > Rogue8Compiler Properties > Configuration Properties > C/C++ > Preprocessor* and type `_CRT_SECURE_NO_WARNINGS` in the *Preprocessor Definitions* field.



- 7) If there are no errors, open the *STM_Machine* project folder and open the *Debug* folder.
 The STM application will be in the folder.



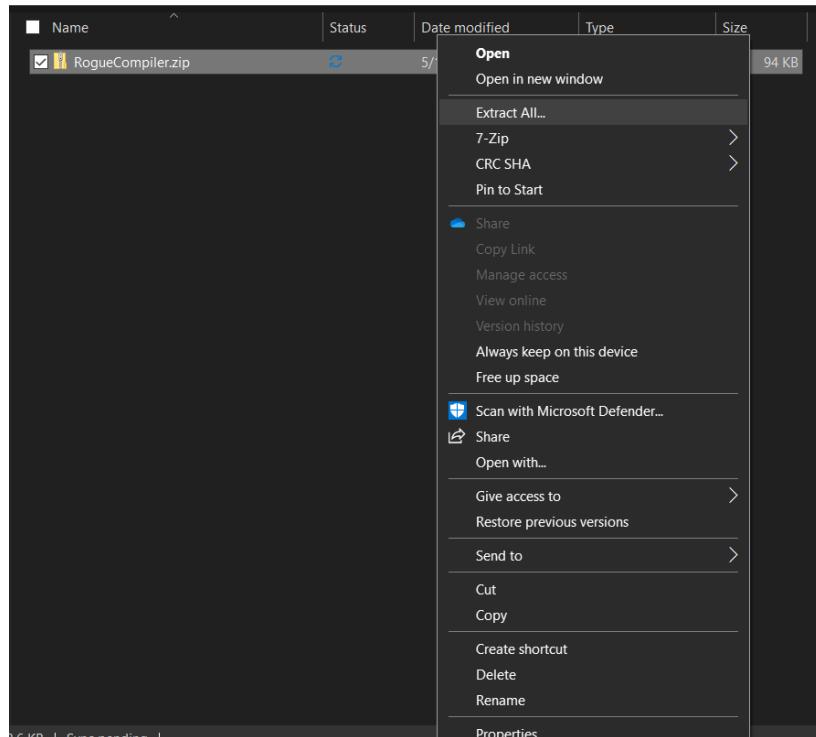
Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> Debug	✓	5/11/2023 4:47 AM	File folder	
<input type="checkbox"/> STM_Machine	✓	5/11/2023 4:47 AM	File folder	
<input type="checkbox"/> STM_Machine.sln	✓	8/22/2021 4:17 PM	Visual Studio Solut...	2 KB



Name	Status	Date modified	Type	Size
<input checked="" type="checkbox"/> STM_Machine.exe	✓	5/5/2023 5:15 AM	Application	128 KB
<input type="checkbox"/> STM_Machine.pdb	✓	5/5/2023 5:15 AM	Program Debug D...	1,212 KB

Starting the System

- 1) **Skip this step if you have compiled the applications using the source files:** Upon receiving the .zip file containing the *Rogue Compiler* and *STM*, go to the file's location on your machine and unzip the file (i.e., extract its contents):
- Make sure the .zip file contains **both** the *Rogue Compiler* and *STM*, named *Rogue<#>Compiler.exe* (<#> refers to the version, the following instructions use version 8) and *STM_Machine.exe*, respectively. **Both applications are needed for Rogue programs to be compiled and executed.**



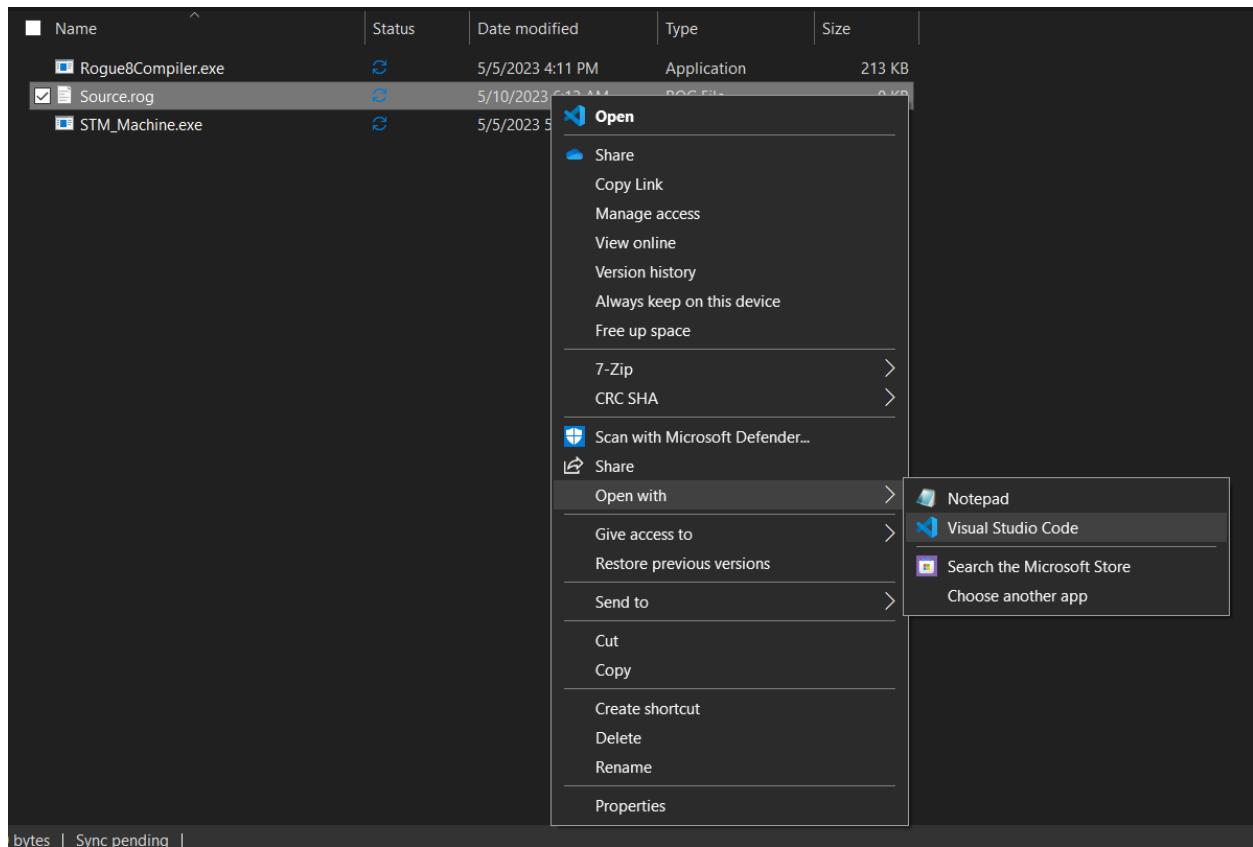
Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	⟳	5/5/2023 4:11 PM	Application	213 KB
STM_Machine.exe	⟳	5/5/2023 5:15 AM	Application	128 KB

2) To open a new Rogue program file, open a .txt file and change the file extension to **.rog**.

Open the file in any text editor or IDE on your machine.

- The developer recommends using Visual Studio Code to open and edit a Rogue program file.
- To edit the file extension of a file on Windows, go to File Explorer > View > File name extensions and have the box be checked.

Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	⟳	5/5/2023 4:11 PM	Application	213 KB
<input checked="" type="checkbox"/> Source.rog	⟳	5/10/2023 6:12 AM	ROG File	0 KB
STM_Machine.exe	⟳	5/5/2023 5:15 AM	Application	128 KB





- 3) To compile a Rogue source file, launch the Rogue Compiler. The application will ask for the name of the Rogue program file. Enter the filename and hit Enter (or Return if on Mac).
- If the source file is found, the compiler will print out each line of the source file until it reaches the end of the file or it encounters a syntax error
 - The compilation information along with any errors will be printed in the listing file created in the same directory of the compiler (extension **.list**).

If the file compiles with no errors, the translated STM assembly code will appear in the compiler's directory (extension **.stm**).

Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	✓	11/23/2023 6:32 AM	Application	335 KB
Source.rog	✓	11/30/2023 1:20 AM	ROG File	1 KB
STM_Machine.exe	✓	11/9/2023 12:21 AM	Application	129 KB

```
C:\Users\asafo\OneDrive\Desktop\Courses\CS6340 Adv. SW Eng\Rogue>
Source filename? Source
1 $$ -----
2 $$ Mujeeb Adelekan
3 $$ Source
4 $$ Source.rog
5 $$ -----*$$
6
7 ****
8 $$ Main function
9 ****
10 main
11     print("Hello World!\n");
12     print("This is the source file.\n");
13 end main
Rogue compiler ending
Press any key to continue . . .
```

Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	✓	11/23/2023 6:32 AM	Application	335 KB
Source.list	✓	11/30/2023 1:20 AM	LIST File	2 KB
Source.rog	✓	11/30/2023 1:20 AM	ROG File	1 KB
Source.stm	✓	11/30/2023 1:20 AM	STM File	5 KB
STM_Machine.exe	✓	11/9/2023 12:21 AM	Application	129 KB

- 4) To execute an STM assembly code file, launch the STM Machine. The application will ask for the name of the STM assembly file. Enter the filename and hit Enter (or Return if on Mac).
- If the STM file is found, STM will begin executing the file.
 - STM will create a log file that contains information about the machine during execution, including any run-time errors (extension .log).

Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	✓	11/23/2023 6:32 AM	Application	335 KB
Source.list	✓	11/30/2023 1:20 AM	LIST File	2 KB
Source.rog	✓	11/30/2023 1:20 AM	ROG File	1 KB
Source.stm	✓	11/30/2023 1:20 AM	STM File	5 KB
STM_Machine.exe	✓	11/9/2023 12:21 AM	Application	129 KB

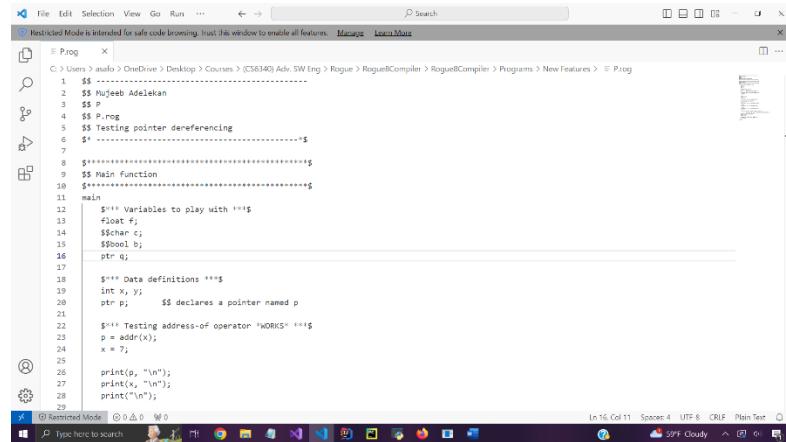
```
C:\Users\asafo\OneDrive\Desktop\Courses\CS6>
Version September 15, 2020

Source filename? Source
Log file is Source.log
Hello World!
This is the source file.
Normal program termination
Press any key to continue . . .
```

Name	Status	Date modified	Type	Size
Rogue8Compiler.exe	⟳	11/23/2023 6:32 AM	Application	335 KB
Source.list	✓	11/30/2023 1:20 AM	LIST File	2 KB
Source.log	⟳	11/30/2023 1:21 AM	LOG File	10 KB
Source.rog	✓	11/30/2023 1:20 AM	ROG File	1 KB
Source.stm	✓	11/30/2023 1:20 AM	STM File	5 KB
STM_Machine.exe	⟳	11/9/2023 12:21 AM	Application	129 KB

Sample Sessions

This section runs through the Rogue compilation process using three sample programs. Each program will be listed in its entirety following all three sessions.

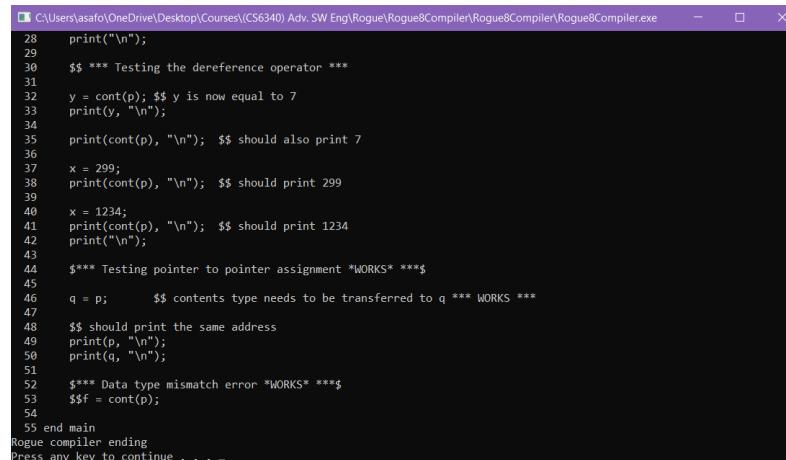


```

File Edit Selection View Go Run ... ⌘ ⌘ Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More ...
Prog x
C:\Users\asafo>adofl>OneDrive\Desktop\Courses\CS6340\Adv.SW Eng\Rogue\RogueCompiler\RogueCompiler\Programs>New Features>P.prog
1 $$ Mujeeb Adelekan
2 $$
3 $>
4 $>#ros
5 $$ Testing pointer dereferencing
6 $# -----
7
8 g*****-----$*****g
9 $$ Main Function
10 g-----$*****g
11 main.
12 |     $*** Variables to play with ***$*
13 |     float f;
14 |     $char c;
15 |     $Bool b;
16 |     $ptr d;
17 |
18 |     $*** Data definitions ***$*
19 |     int x, y;
20 |     $ptr p;      $$ declares a pointer named p
21 |
22 |     $*** Testing address-of operator *WORKS* ***$*
23 |     p = addr(x);
24 |     x = 7;
25 |
26 |     print(p, "\n");
27 |     print(x, "\n");
28 |     print("\n");
29 |
30 |     $*** Testing the dereference operator ***$*
31 |     y = cont(p); $$ y is now equal to 7
32 |     print(y, "\n");
33 |
34 |     print(cont(p), "\n"); $$ should also print 7
35 |
36 |     x = 299;
37 |     print(cont(p), "\n"); $$ should print 299
38 |
39 |     x = 1234;
40 |     print(cont(p), "\n"); $$ should print 1234
41 |     print("\n");
42 |
43 |     $*** Testing pointer to pointer assignment *WORKS* ***$*
44 |
45 |     q = p;        $$ contents type needs to be transferred to q *** WORKS ***
46 |
47 |     $$ should print the same address
48 |     print(p, "\n");
49 |     print(q, "\n");
50 |
51 |     $*** Data type mismatch error *WORKS* ***$*
52 |     $$f = cont(p);
53 |
54 |
55 end main
Rogue compiler ending
Press any key to continue . . .

```

This Rogue program, **P.prog**, demonstrates pointer dereferencing in Rogue.

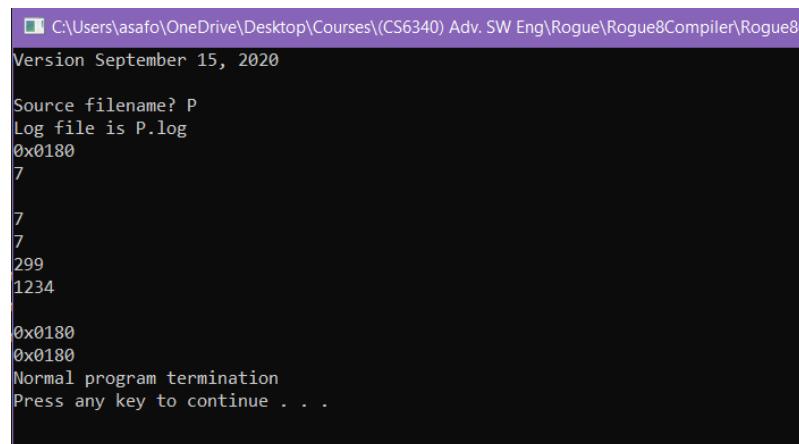


```

C:\Users\asafo>adofl>OneDrive\Desktop\Courses\CS6340\Adv.SW Eng\Rogue\Rogue8Compiler\Rogue8Compiler\Rogue8Compiler.exe
28 print("\n");
29
30 $$ *** Testing the dereference operator ***
31
32 y = cont(p); $$ y is now equal to 7
33 print(y, "\n");
34
35 print(cont(p), "\n"); $$ should also print 7
36
37 x = 299;
38 print(cont(p), "\n"); $$ should print 299
39
40 x = 1234;
41 print(cont(p), "\n"); $$ should print 1234
42 print("\n");
43
44 $*** Testing pointer to pointer assignment *WORKS* ***
45
46 q = p;        $$ contents type needs to be transferred to q *** WORKS ***
47
48 $$ should print the same address
49 print(p, "\n");
50 print(q, "\n");
51
52 $*** Data type mismatch error *WORKS* ***
53 $$f = cont(p);
54
55 end main
Rogue compiler ending
Press any key to continue . . .

```

The user compiles **P.prog** with the **Rogue Compiler**. This generates **P.list** and **P.stm**.



```

C:\Users\asafo>adofl>OneDrive\Desktop\Courses\CS6340\Adv.SW Eng\Rogue\Rogue8Compiler\Rogue8Compiler.exe
Version September 15, 2020

Source filename? P
Log file is P.log
0x0180
7
7
299
1234

0x0180
0x0180
Normal program termination
Press any key to continue . . .

```

The user executes **P.stm** on **STM**.

```

File Edit Selection View Go Run ... Search
Restricted Mode is intended for safe code browsing; trust this window to enable all features. Manage Learn More
A.rog X
C:\Users\asaf0>OneDrive\Desktop>Courses>(CS6340) Adv. SW Eng>Rogue>Rogue8Compiler>Rogue8Compiler>Programs>New Features> A.rog
1 $#
2 ## Asafe Adelekan
3 #
4 $$ A.rog
5 $$ Testing Rogue Associative Arrays
6 $" -----
7
8 $$assoc globalArr{$}; ***WORKS***
9
10
11 prec P1
12     $$ An associative array with a capacity of 4
13     assoc procArr{$};
14
15     $$ Four (key, value) pairs
16     procArr{"C"} = 5 + true;      $$ (5, true)
17     procArr{< 4} = 4 + 7;        $$ (false, 11)
18     procArr{"A"} = 3.5 + 4.7;    $$ ("A", 8.2)
19     procArr{1.2e1} = "#";       $$ (12.0, "#")
20
21     $$ Extra element (run-time error) *** WORKS ***
22     $$procArr{8} = 3;
23
24     $$ Existing key (value should be replaced) *** WORKS ***
25     $$procArr{5} = "8";
26
27     $$ Print the values of the associative array
28     print("procArr{$} ", procArr{$} as bool, "\n");
29     print("procArr{false} = ", procArr{< 4} as int, "\n");
30
31 Remained Mode (A) W

```

This Rogue program, **A.rog**, demonstrates associative arrays in Rogue.

```

C:\Users\asaf0>OneDrive\Desktop>Courses>(CS6340) Adv. SW Eng>Rogue>Rogue8Compiler>Rogue8Compiler>Rogue8Compiler.exe
87     print("The decimal value of F is ", hex_to_dec('F') as int, "\n");
88
89     print("\n");
90
91     $$ Replace the value of key 'B' with a float literal
92     hex_to_dec('B') = 3.14;
93
94     $$ Replace the value of key 'E' with a bool literal
95     hex_to_dec('E') = false;
96
97     $$ Print the replacement values
98     print("The *new* value of B is ", hex_to_dec('B') as float, "\n");
99     print("The *new* value of E is ", hex_to_dec('E') as bool, "\n");
100
101    print("\n");
102
103    $$ Assign an associative array element to a variable
104    x = hex_to_dec('D') as int;
105
106    $$ Print the decimal value of D using x
107    print("The decimal value of D is still ", x, "\n");
108    print("\n");
109
110    $$ Disallow associative array references in addr() (compile-time error) ***WORKS***
111    $$print("addr(x) = ", addr(hex_to_dec), "\n");
112
113 end main
Rogue compiler ending
Press any key to continue . . .

```

The user compiles **A.rog** with the *Rogue Compiler*. This generates **A.list** and **A.stm**.

```

C:\Users\asaf0>OneDrive\Desktop>Courses>(CS6340) Adv. SW Eng>Rogue>Rogue8Compiler>Rogue8Compiler>STM_Machine.exe
Version September 15, 2020
Source filename? A
Log file is A.log
a_arr{5} = T
a_arr{false} = 11
a_arr{"A"} = 8.195
a_arr{1.2e1} = #

procArr{5} = T
procArr{false} = 11
procArr{"A"} = 8.195
procArr{1.2e1} = #

The decimal value of A is 10
The decimal value of B is 11
The decimal value of C is 12
The decimal value of D is 13
The decimal value of E is 14
The decimal value of F is 15

The *new* value of B is 3.139
The *new* value of E is F

The decimal value of D is still 13

Normal program termination
Press any key to continue . . .

```

The user executes **A.stm** on *STM*.

This Rogue program, *E0.rog*, demonstrates enumeration types in Rogue.

```
C:\Users\asafo\OneDrive\Desktop\Courses\CS6340 Adv. SW Eng\Rogue\Rogue8Compiler\Rogue8Compiler\Rogue8Compiler.exe - - 
3 $$ E0
4 $$ E0.rog
5 $$ Testing Enumeration Types
6 $$ -----
7
8 ****
9 $$ Main function
10 ****
11 main
12     enum : month
13     {
14         Jan, Feb, Mar, Apr,
15         May, Jun, Jul, Aug,
16         Sep, Oct, Nov, Dec
17     };
18
19     enum : season {winter, spring, summer, fall};
20
21     month currentMonth;
22     season currentSeason;
23
24     currentMonth = Sep;
25     currentSeason = fall;
26
27     print(currentMonth, "\n");    $$ should print 8
28     print(currentSeason, "\n");  $$ should print 3
29
30 end main
Rogue compiler ending
Press any key to continue . . . -
```

The user compiles **E0.rog** with the *Rogue Compiler*. This generates E0.list and E0.stm.

```
C:\Users\asafo\OneDrive\Desktop\Courses\CS6340 Adv. SW Eng\Rog  
Version September 15, 2020  
  
Source filename? E0  
Log file is E0.log  
8  
3  
Normal program termination  
Press any key to continue . . . ■
```

The user executes E0.stm on STM.

```
=====
P.rog: demonstrates pointer dereferencing in Rogue
=====
```

```
$$ -----
$$ Mujeeb Adelekan
$$ P
$$ P.rog
$$ Testing pointer dereferencing
$* -----*$

*****$
```

```
$$ Main function
*****$
```

```
main
    $*** Data definitions ***$

    int x, y;

    ptr p, q;      $$ declares pointers named p and q

    $*** Testing address-of operator *WORKS* ***$

    p = addr(x);
    x = 7;

    print(p, "\n");
    print(x, "\n");
```

```
print("\n");

$$ *** Testing the dereference operator ***

y = cont(p); $$ y is now equal to 7
print(y, "\n");

print(cont(p), "\n"); $$ should also print 7

x = 299;
print(cont(p), "\n"); $$ should print 299

x = 1234;
print(cont(p), "\n"); $$ should print 1234
print("\n");

$*** Testing pointer to pointer assignment ***$
q = p;      $$ contents type needs to be transferred to q

$$ should print the same address
print(p, "\n");
print(q, "\n");

end main
```

```
=====
A.rog: demonstrates associative arrays in Rogue
=====
```

```
$$ -----
$$ Mujeeb Adelekan
$$ A
$$ A.rog
$$ Testing Rogue Associative Arrays
$* -----*$
```

```
$$assoc globalArr{5}; ***WORKS***
```

```
proc P1
    $$ An associative array with a capacity of 4
    assoc procArr{4};
```

```
    $$ Four (key, value) pairs
    procArr{5} = true;           $$ (5, true)
    procArr{5 < 4} = 4 + 7;     $$ (false, 11)
    procArr{'A'} = 3.5 + 4.7;   $$ ('A', 8.2)
    procArr{1.2e1} = '#';       $$ (12.0, '#')
```

```
    $$ Extra element (run-time error) *** WORKS ***
```

```

$$procArr{8} = 3;

$$ Existing key (value should be replaced) *** WORKS ***
$$procArr{5} = '&';

$$ Print the values of the associative array
print("procArr{5} = ", procArr{5} as bool, "\n");
print("procArr{false} = ", procArr{5 < 4} as int, "\n");
print("procArr{'A'} = ", procArr{'A'} as float, "\n");
print("procArr{1.2e1} = ", procArr{1.2e1} as char, "\n");

print("\n");
end proc

$*****
$ Main function
$*****
main

$$ An associative array with a capacity of 4
assoc a_arr{4};

$$ An associative array that maps hex digits with their decimal values
assoc hex to dec{6};

```

```
## int variable x
int x;

## Four (key, value) pairs
a_arr{5} = true;          ## (5, true)
a_arr{5 < 4} = 4 + 7;    ## (false, 11)
a_arr{'A'} = 3.5 + 4.7;  ## ('A', 8.2)
a_arr{1.2e1} = '#';      ## (12.0, '#')

## Extra element (run-time error) *** WORKS ***
$$a_arr{8} = 3;

## Existing key (value should be replaced) ***WORKS***
$$a_arr{5} = '&';

## Print the values of the associative array
print("a_arr{5} = ", a_arr{5} as bool, "\n");
print("a_arr{false} = ", a_arr{5 < 4} as int, "\n");
print("a_arr{'A'} = ", a_arr{'A'} as float, "\n");
print("a_arr{1.2e1} = ", a_arr{1.2e1} as char, "\n");

print("\n");
```

```
call P1;

$$ Add the hex digits A-F with their decimal values
hex_to_dec{'A'} = 10;
hex_to_dec{'B'} = 11;
hex_to_dec{'C'} = 12;
hex_to_dec{'D'} = 13;
hex_to_dec{'E'} = 14;
hex_to_dec{'F'} = 15;

$$ Print the demical values of A-F
print("The demical value of A is ", hex_to_dec{'A'} as int, "\n");
print("The demical value of B is ", hex_to_dec{'B'} as int, "\n");
print("The demical value of C is ", hex_to_dec{'C'} as int, "\n");
print("The demical value of D is ", hex_to_dec{'D'} as int, "\n");
print("The demical value of E is ", hex_to_dec{'E'} as int, "\n");
print("The demical value of F is ", hex_to_dec{'F'} as int, "\n");

print("\n");

$$ Replace the value of key 'B' with a float literal
hex_to_dec{'B'} = 3.14;
```

```
## Replace the value of key 'E' with a bool literal
hex_to_dec{'E'} = false;

## Print the replacement values
print("The *new* value of B is ", hex_to_dec{'B'} as float, "\n");
print("The *new* value of E is ", hex_to_dec{'E'} as bool, "\n");

print("\n");

## Assign an associative array element to a variable
x = hex_to_dec{'D'} as int;

## Print the decimal value of D using x
print("The demical value of D is still ", x, "\n");
print("\n");

## Disallow associative array references in addr() (compile-time error) ***WORKS***
##print("addr(x) = ", addr(hex_to_dec), "\n");

end main
```

```
=====
E0.rog: demonstrates enumeration types in Rogue
=====
```

```
$$ -----
$$ Mujeeb Adelekan
$$ E0
$$ E0.rog
$$ Testing Enumeration Types
$* -----*$
```

```
$*****$
```

```
$$ Main function
```

```
$*****$
```

```
main
```

```
    enum : month
```

```
{
```

```
    Jan, Feb, Mar, Apr,
```

```
    May, Jun, Jul, Aug,
```

```
    Sep, Oct, Nov, Dec
```

```
};
```

```
enum : season {winter, spring, summer, fall};
```

```
month currentMonth;
```

```
season currentSeason;

currentMonth = Sep;
currentSeason = fall;

print(currentMonth, "\n");  $$ should print 8
print(currentSeason, "\n"); $$ should print 3

end main
```

Troubleshooting

This section highlights a few issues the user may encounter when using the *Rogue Compiler* and *STM Machine*, along with the possible causes and solutions for those issues.

Issue	Possible Causes	Solution
Rogue Compiler gives message: “Unable to open source file”	<ul style="list-style-type: none"> a) A .rog file with the name given does not exist b) A .rog file with the given name is not in the same folder as the <i>Rogue Compiler</i> 	<ul style="list-style-type: none"> a) Make sure that the .rog file exists and that its name is spelled correctly when prompted for the source filename (capitalization does not matter) b) Either move the .rog file into the same folder <u>OR</u> enter the full path of the .rog file.
Rogue Compiler gives message: “Rogue exception: Rogue compiler ending with compiler error!”	The Rogue code contains a syntax error.	Open the .list file generated by the <i>Rogue Compiler</i> and read the detailed error message including the line number. Fix the code in the .rog file accordingly and compile again.
STM gives message: “Error opening source file” followed by the name of the STM file	<ul style="list-style-type: none"> a) An .stm file with the name given does not exist b) A .stm file with the given name is not in the same folder as the <i>STM Machine</i> 	<ul style="list-style-type: none"> a) Make sure that the .stm file exists and that its name is spelled correctly when prompted for the source filename (<u>NOTE:</u> capitalization does not matter) b) Either move the .stm file into the same folder <u>OR</u> enter the full path of the .stm file.
STM gives a “Run time error” followed by a description of the error	The Rogue code contains a logic error (i.e., a “bug”)	Consult the .log file generated by the <i>STM Machine</i> to see which line in the .rog file caused a run-time error. Fix the code in the .rog file accordingly and compile again using the <i>Rogue Compiler</i> .

References

C language. cppreference.com. (2022, September 20). Retrieved February 6, 2023, from
<https://en.cppreference.com/w/c/language>

C++ language. cplusplus.com. (n.d.). Retrieved March 6, 2023, from
<https://cplusplus.com/doc/tutorial/>

Malan, D. J. (2023). *CS50 Introduction to Computer Science*. CS50x 2023.
<https://cs50.harvard.edu/x/2023/>

Microsoft. (2023). *Windows Help and Learning*. Microsoft Support.
https://support.microsoft.com/en-us/windows#ID0EBBD=Windows_10

Pressman R., Maxim B. (2014). Software Engineering: A Practitioner's Approach (8th ed.).
McGraw-Hill Education, New York, United States

Sebesta, R. W. (2016). *Concepts of Programming Languages* (11th ed.). Pearson.

Source Code

This section is a listing of all the source code that supports the Rogue language. **Rogue8Compiler.cpp** and **Rogue.h** are the source files for the *Rogue Compiler*. **STM.c** is the source code for the STM Machine.

Rogue8Compiler.cpp

```
//-----  
// Mujeeb Adelekan  
// Rogue8 Compiler  
// Rogue8Compiler.cpp  
//-----  
  
#include <iostream>  
#include <iomanip>  
  
#include <fstream>  
#include <cstdio>  
#include <cstdlib>  
#include <cstring>  
#include <cctype>  
#include <vector>  
  
#include <unordered_map> // Use for enum types. E.g., unordered_map<string, vector<string>> enumMap  
                      // where the key is the enum type name and the values are the constants for each type  
                      // This way, the index of each constant in the vector is that constant's corresponding  
                      // integer value.  
  
using namespace std;
```

```
/*#define TRACEREADER
/*#define TRACESCANNER
/*#define TRACEPARSER
#define TRACEIDENTIFIERTABLE
#define TRACECOMPILER

/*#define TESTENUMMAP // uncomment to test the enumMap in the compiler

#include "Rogue.h"

/*
=====
Changes to Rogue7 compiler
=====

Added tokens
(pseudo-terminals)
( reserved words) CHAR FLOAT PTR ADDR CONT FILE
ASSOC AS
ENUM
F_PRINT F_INPUT
F_CREATE
F_READ F_WRITE
F_CLEAR
```

```
F_OPEN  
F_CLOSE  
F_DELETE  
( punctuation) L_BRACKET R_BRACKET  
L_BRACE R_BRACE  
( operators)
```

Updated functions

```
ParseDataDefinitions  
ParseAssignmentStatement  
ParsePrimary  
ParseVariable  
ParseStatement
```

Added functions

```
ParseAssociativeArrayReference  
ParseEnumDeclaration  
ParseFilePrintStatement  
ParseFileInputStatement  
ParseFileOperationStatement
```

```
*/
```

```
//-----  
typedef enum  
//-----  
{
```

```
// pseudo-terminals  
  
IDENTIFIER,  
STRING,  
EOPTOKEN,  
UNKTOKEN,  
INTEGER,  
CHARACTER, // added  
FLOATVAL, // added  
  
// reserved words  
  
MAIN,  
ENDMAIN,  
PRINT,  
OR,  
NOR,  
XOR,  
AND,  
NAND,  
NOT,  
TRUE,  
FALSE,  
INT,  
BOOL,  
CHAR, // ADDED  
FLOAT, // ADDED  
PTR, // ADDED  
ADDR, // ADDED
```

```
CONT, // ADDED
FILE_T, // ADDED
ASSOC, // ADDED
AS, // ADDED
ENUM, // ADDED
F_PRINT, // ADDED
F_INPUT, // ADDED
F_CREATE, // ADDED
F_READ, // ADDED
F_WRITE, // ADDED
F_CLEAR, // ADDED
F_OPEN, // ADDED
F_CLOSE, // ADDED
F_DELETE, // ADDED
PERM,
INPUT,
IF,
ELIF,
ELSE,
DO,
WHILE,
ENDIF,
ENDWHILE,
ASSERT,
FOR,
TO,
```

```
BY,  
ENDFOR,  
PROC,  
ENDPROC,  
IN,  
OUT,  
IO,  
REF,  
CALL,  
RETURN,  
FUNC,  
ENDFUNC,  
  
// punctuation  
COMMA,  
SEMICOLON,  
COLON,  
L_PAREN, //left parenthesis  
R_PAREN, //right parenthesis  
L_BRACKET, // left bracket  
R_BRACKET, // right bracket  
L_BRACE,  
R_BRACE,  
  
// operators  
LT,  
LTEQ,  
EQ,
```

```
GT,  
GTEQ,  
NOTEQ, // <> and !=  
PLUS,  
MINUS,  
MULTIPLY,  
DIVIDE,  
MODULUS,  
ABS,  
POWER, // ^ and **  
ASSIGN,  
INC,  
DEC  
} TOKENTYPE;  
  
//-----  
struct TOKENTABLERECORD  
//-----  
{  
    TOKENTYPE type;  
    char description[12+1];  
    bool isReservedWord;  
};  
  
//-----
```

```
const TOKENTABLERECORD TOKENTABLE[] =  
//----------------------------------------------------------------------------  
{  
    // pseudo-terminals  
    { IDENTIFIER , "IDENTIFIER" , false },  
    { STRING      , "STRING"      , false },  
    { EOPTOKEN    , "EOPTOKEN"    , false },  
    { UNKTOKEN    , "UNKTOKEN"    , false },  
    { INTEGER     , "INTEGER"     , false },  
    { CHARACTER   , "CHARACTER"   , false },  
    { FLOATVAL    , "FLOATVAL"    , false },  
  
    // reserved-words  
    { MAIN        , "MAIN"        , true  },  
    { ENDMAIN     , "END MAIN"    , true  },  
    { PRINT        , "PRINT"       , true  },  
    { OR           , "OR"          , true  },  
    { NOR          , "NOR"         , true  },  
    { XOR          , "XOR"         , true  },  
    { AND          , "AND"         , true  },  
    { NAND         , "NAND"        , true  },  
    { NOT          , "NOT"         , true  },  
    { TRUE         , "TRUE"        , true  },  
    { FALSE        , "FALSE"       , true  },  
    { INT           , "INT"         , true  },  
    { BOOL          , "BOOL"        , true  },  
    { CHAR          , "CHAR"        , true  }, // added
```

```
{ FLOAT      , "FLOAT"        ,true  }, // added
{ PTR        , "PTR"          ,true  }, // added
{ ADDR       , "ADDR"         ,true  }, // added
{ CONT       , "CONT"         ,true  }, // added
{ FILE_T     , "FILE"         ,true  }, // added
{ ASSOC      , "ASSOC"        ,true  }, // added
{ AS         , "AS"           ,true  }, // added
{ ENUM       , "ENUM"         ,true  }, // added
{ F_PRINT    , "F_PRINT"      ,true  }, // added
{ F_INPUT    , "F_INPUT"      ,true  }, // added
{ F_CREATE   , "F_CREATE"    ,true  }, // added
{ F_READ     , "F_READ"       ,true  }, // added
{ F_WRITE    , "F_WRITE"      ,true  }, // added
{ F_CLEAR    , "F_CLEAR"      ,true  }, // added
{ F_OPEN     , "F_OPEN"        ,true  }, // added
{ F_CLOSE    , "F_CLOSE"       ,true  }, // added
{ F_DELETE   , "F_DELETE"     ,true  }, // added
{ PERM       , "PERM"         ,true  },
{ INPUT      , "INPUT"        ,true  },
{ IF          , "IF"           ,true  },
{ ELSE       , "ELSE"         ,true  },
{ ELIF       , "ELIF"         ,true  },
{ DO          , "DO"           ,true  },
{ WHILE      , "WHILE"        ,true  },
{ ENDIF      , "END IF"       ,true  },
{ ENDWHILE   , "END WHILE"    ,true  },
```

```
{ ASSERT      , "ASSERT"      ,true  },
{ FOR        , "FOR"        ,true  },
{ TO         , "TO"         ,true  },
{ BY         , "BY"         ,true  },
{ ENDFOR     , "END FOR"    ,true  },
{ PROC       , "PROC"       ,true  },
{ ENDPROC    , "END PROC"   ,true  },
{ IN          , "IN"          ,true  },
{ OUT         , "OUT"         ,true  },
{ IO          , "IO"          ,true  },
{ REF         , "REF"         ,true  },
{ CALL        , "CALL"        ,true  },
{ RETURN     , "RETURN"     ,true  },
{ FUNC        , "FUNC"        ,true  },
{ ENDFUNC    , "END FUNC"   ,true  },

// punctuation

{ COMMA       , "COMMA"       ,false },
{ SEMICOLON   , "SEMICOLON"   ,false },
{ L_PAREN     , "LEFT PAREN"  ,false },
{ R_PAREN     , "RIGHT PAREN" ,false },
{ L_BRACKET   , "LEFT BRACK"  ,false }, // added
{ R_BRACKET   , "RIGHT BRACK" ,false }, // added
{ L_BRACE     , "LEFT BRACE"  ,false }, // added
{ R_BRACE     , "RIGHT BRACE" ,false }, // added

// operators

{ LT          , "LT"          ,false },
```

```

{ LTEQ      , "LTEQ"      ,false },
{ EQ        , "EQ"        ,false },
{ GT        , "GT"        ,false },
{ GTEQ      , "GTEQ"      ,false },
{ NOTEQ     , "NOTEQ"     ,false },
{ PLUS      , "PLUS"      ,false },
{ MINUS     , "MINUS"     ,false },
{ MULTIPLY  , "MULTIPLY" ,false },
{ DIVIDE    , "DIVIDE"    ,false },
{ MODULUS   , "MODULUS"   ,false },
{ ABS        , "ABS"        ,true },
{ POWER     , "POWER"     ,false },
{ ASSIGN    , "ASSIGN"    ,false },
{ INC        , "INC"        ,false },
{ DEC        , "DEC"        ,false }

};

//-----
struct TOKEN
//-----
{
    TOKENTYPE type;
    char lexeme[SOURCELINELENGTH+1];
    int sourceLineNumber;
    int sourceLineIndex;
};

```

```
//-----
// Global variables
//-----

READER<CALLBACKSUSED> reader(SOURCELINELENGTH,LOOKAHEAD);

LISTER lister(LINESPERPAGE);

CODE code;

IDENTIFIERTABLE identifierTable(&lister,MAXIMUMIDENTIFIERS);

/*** ptr_indexes ***/

vector<int> ptr_indexes; // vector of pointer indexes

/*** enumMap ***/

// map of scoped enum types to their constant values

//      key type: IDENTIFIERSCOPE

//      value type: unordered_map< string, vector<string> >

// IDENTIFIERSCOPE and enumType are used as first and second keys,
//      respectively.

unordered_map< IDENTIFIERSCOPE, unordered_map<string, vector<string>> > enumMap;

/*** lValueEnumScope and rValueEnumType ***/

// used for searching the enumMap for an r-value enum constant

IDENTIFIERSCOPE lValueEnumScope;      // stores the current scope of the l-value enum variable
string lValueEnumType = "";          //stores the enum type of the l-value enum variable
```

```
#ifdef TRACEPARSER
int level;
#endif

//-----
void EnterModule(const char module[])
//-----
{
#ifdef TRACEPARSER
    char information[SOURCELINELENGTH+1];

    level++;
    sprintf(information,"    %*s>%s",level*2," ",module);
    lister.ListInformationLine(information);

#endif
}

//-----
void ExitModule(const char module[])
//-----
{
#ifdef TRACEPARSER
    char information[SOURCELINELENGTH+1];

    sprintf(information,"    %*s<%s",level*2," ",module);
    lister.ListInformationLine(information);
}
```

```
level--;

#endif
}

//-----
void ProcessCompilerError(int sourceLineNumber, int sourceLineIndex, const char errorMessage[])
//-----
{

    char information[SOURCELINELENGTH + 1];

    // Use "panic mode" error recovery technique: report error message and terminate compilation!
    sprintf(information, "      At (%4d:%3d) %s", sourceLineNumber, sourceLineIndex, errorMessage);
    lister.ListInformationLine(information);
    lister.ListInformationLine("Rogue compiler ending with compiler error!\n");
    throw(ROGUEEXCEPTION("Rogue compiler ending with compiler error!"));
}

//-----
int main()
//-----
{
    void Callback1(int sourceLineNumber, const char sourceLine[]);
    void Callback2(int sourceLineNumber, const char sourceLine[]);
    void GetNextToken(TOKEN tokens[]);
    void ParseRogueProgram(TOKEN tokens[]);
}
```

```
char sourceFileName[80 + 1];
TOKEN tokens[LOOKAHEAD + 1];

cout << "Source filename? ";
cin >> sourceFileName;

try
{
    lister.OpenFile(sourceFileName);
    code.OpenFile(sourceFileName);

// CODEGENERATION
    code.EmitBeginningCode(sourceFileName);
// ENDCODEGENERATION

    reader.SetLister(&lister);
    reader.AddCallbackFunction(Callback1);
    reader.AddCallbackFunction(Callback2);
    reader.OpenFile(sourceFileName);

// Fill tokens[] for look-ahead
    for (int i = 0; i <= LOOKAHEAD; i++)
        GetNextToken(tokens);

#endif TRACEPARSER
    level = 0;
```

```
#endif

ParseRogueProgram(tokens);

// CODEGENERATION
    code.EmitEndingCode();
// ENDCODEGENERATION

}

catch (ROGUEEXCEPTION rogueException)
{
    cout << "Rogue exception: " << rogueException.GetDescription() << endl;
}

lister.ListInformationLine("***** Rogue compiler ending");

cout << "Rogue compiler ending\n";

system("PAUSE");

return(0);

}

//-----
void ParseRogueProgram(TOKEN tokens[])
//-----
{
    void ParseDataDefinitions(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);
    void GetNextToken(TOKEN tokens[]);
```

```
void ParseEnumDeclaration(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);
void ParseProcedureDefinition(TOKEN tokens[]);
void ParseFunctionDefinition(TOKEN tokens[]);
void ParseMainProgram(TOKEN tokens[]);

EnterModule("RogueProgram");

while ((tokens[0].type == ENUM))
{
    ParseEnumDeclaration(tokens, GLOBALSCOPE);
}

ParseDataDefinitions(tokens, GLOBALSCOPE);

#ifndef TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table after compilation of global data definitions");
#endif

while ((tokens[0].type == PROC) || (tokens[0].type == FUNC) )
{
    switch (tokens[0].type)
    {
        case PROC:
            ParseProcedureDefinition(tokens);
            break;
    }
}
```

```
    case FUNC:
        ParseFunctionDefinition(tokens);
        break;
    }

}

if (tokens[0].type == MAIN)
    ParseMainProgram(tokens);
else
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting MAIN");

if (tokens[0].type != EOPTOKEN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting end-of-program");

ExitModule("RogueProgram");
}

//-----
void ParseDataDefinitions(TOKEN tokens[], IDENTIFIERSCOPE identifierScope) /*** updated ***/
//-----
{
    void GetNextToken(TOKEN tokens[]);
}
```

```

EnterModule("DataDefinitions");

while ( (tokens[0].type == INT) || (tokens[0].type == BOOL) || (tokens[0].type == PERM)
|| (tokens[0].type == CHAR) || (tokens[0].type == FLOAT) || (tokens[0].type == PTR)
|| (tokens[0].type == FILE_T) || (tokens[0].type == ASSOC )
|| (enumMap[GLOBALSCOPE].find(tokens[0].lexeme) != enumMap[GLOBALSCOPE].end())           // check if the token is a global enumType
|| (enumMap[identifierScope].find(tokens[0].lexeme) != enumMap[identifierScope].end())        // check if the token is a local enumType
{
// <constantDefintions>
    if (tokens[0].type == PERM)
    {
        /*
        Set varSeries to true if a datatype is followed by a comma-separated list of constants.
        e.g.,      perm int w = 2, x = 4, bool y = true, z = false;

        The above statement defines two integer constants w == 2 and x == 4
        and two boolean constants y == true and z == false.
        */
        bool varSeries = false;
        DATATYPE datatype;
        do
        {
            char identifier[MAXIMUMLENGTHIDENTIFIER + 1];
            char literal[MAXIMUMLENGTHIDENTIFIER + 1];
            char reference[MAXIMUMLENGTHIDENTIFIER + 1];
            char operand[MAXIMUMLENGTHIDENTIFIER + 1];

```

```
char comment[MAXIMUMLENGTHIDENTIFIER + 1];
bool isInTable;
int index;

GetNextToken(tokens);

// Check if the token is <datatype>
switch (tokens[0].type)
{
    case INT:
        datatype = INTEGERTYPE;
        if (!varSeries)
            varSeries = true;
        GetNextToken(tokens);
        break;
    case BOOL:
        datatype = BOOLEANTYPE;
        if (!varSeries)
            varSeries = true;
        GetNextToken(tokens);
        break;
    case CHAR:
        datatype = CHARACTERTYPE;
        if (!varSeries)
            varSeries = true;
        GetNextToken(tokens);
```

```
        break;

    case FLOAT:
        datatype = FLOATTTYPE;
        if (!varSeries)
            varSeries = true;
        GetNextToken(tokens);
        break;

    case IDENTIFIER:
        if (!varSeries)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting 'int', 'bool', 'char', or
'float');");
        break;

    default:
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting 'int', 'bool', 'char', or 'float'");

    }

// Check if token is <identifier>
if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
strcpy(identifier, tokens[0].lexeme);
GetNextToken(tokens);

// Check if token is the assign operator '='
```

```
if (tokens[0].type != ASSIGN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '='");
GetNextToken(tokens);

// Analyze the data type of the <literal>
if ((datatype == INTEGERTYPE) && (tokens[0].type == INTEGER))
{
    strcpy(literal, "0D");
    strcat(literal, tokens[0].lexeme);
}

else if (((datatype == BOOLEANTYPE) && (tokens[0].type == TRUE))
         || ((datatype == BOOLEANTYPE) && (tokens[0].type == FALSE)))
{
    strcpy(literal, tokens[0].lexeme);
}

else if ((datatype == CHARACTERTYPE) && (tokens[0].type == CHARACTER))
{
    strcpy(literal, tokens[0].lexeme);
}

else if ((datatype == FLOATATTYPER) && (tokens[0].type == FLOATVAL))
{
    strcpy(literal, "0F");
    strcat(literal, tokens[0].lexeme);
}

else
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Data type mismatch");
```

```
GetNextToken(tokens);

index = identifierTable.GetIndex(identifier, isInTable);
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

switch (identifierScope)
{
    case GLOBALSCOPE:
// CODEGENERATION
        code.AddDWToStaticData(literal, identifier, reference);
// ENDCODEGENERATION
        identifierTable.AddToTable(identifier, GLOBAL_CONSTANT, datatype, reference);
        break;
    case PROGRAMMODULESCOPE:
// CODEGENERATION
        code.AddDWToStaticData(literal, identifier, reference);
// ENDCODEGENERATION
        identifierTable.AddToTable(identifier, PROGRAMMODULE_CONSTANT, datatype, reference);
        break;
    case SUBPROGRAMMODULESCOPE:
// CODEGENERATION
        sprintf(reference, "FB:0D%d", code.GetFBOffset());
        strcpy(operand, "#"); strcat(operand, literal);
        sprintf(comment, "initialize constant %s", identifier);
        code.AddInstructionToInitializeFrameData("PUSH", operand, comment);
```

```
        code.AddInstructionToInitializeFrameData("POP", reference);
        code.IncrementFBOffset(1);

// ENDCODEGENERATION

        identifierTable.AddToTable(identifier, SUBPROGRAMMODULE_CONSTANT, datatype, reference);
        break;

    }

} while ( tokens[0].type == COMMA );

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");

GetNextToken(tokens);

}

// <AssociativeArrayDefinition>

else if (tokens[0].type == ASSOC)
{
    char identifier[MAXIMUMLENGTHIDENTIFIER + 1];
    char operand[MAXIMUMLENGTHIDENTIFIER + 1];
    char comment[MAXIMUMLENGTHIDENTIFIER + 1];
    char reference[MAXIMUMLENGTHIDENTIFIER + 1];
    bool isInTable;
    int index;
    int capacity; // the capacity of the array
```

```
GetNextToken(tokens);

// Check if token is <identifier>
if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
strcpy(identifier, tokens[0].lexeme);
GetNextToken(tokens);

index = identifierTable.GetIndex(identifier, isInTable);
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

// Check for a left brace: '{'
if (tokens[0].type != L_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '{'");
GetNextToken(tokens);

// Check if the associative array capacity is an integer literal
if (tokens[0].type != INTEGER)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Associative array capacity must be an integer literal");
}

// Assign the capacity with the integer value
capacity = atoi(tokens[0].lexeme);
```

```
GetNextToken(tokens);

// Check for a right brace: '}'
if (tokens[0].type != R_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '}'");
GetNextToken(tokens);

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");
GetNextToken(tokens);

// CODEGENERATION
int base; // used for the SUBPROGRAMMODULESCOPE

switch (identifierScope)
{
case GLOBALSCOPE:
    // Reserve a word in STATICDATA for the size of the associative array (initial value is 0)
    sprintf(comment, "%s at SB:0%D%d", identifier, code.GetSBOffset());
    code.AddRWToStaticData(1, comment, reference);

    // Add to the identifier table as a one-dimensional associative array
    identifierTable.AddToTable(identifier, GLOBAL_VARIABLE, ASSOCTYPE, reference, 1);

    // Define a word in STATICDATA with the capacity of the associative array
```

```
sprintf(operand, "0D%d", capacity);

code.AddDWToStaticData(operand, "", reference);

// Reserve words for each (key, value) pair in the associative array
code.AddRWToStaticData(2 * capacity, "", reference);

break;

case PROGRAMMODULESCOPE:

    // Reserve a word in STATICDATA for the size of the associative array (initial value is 0)
    sprintf(comment, "%s at SB:0D%d", identifier, code.GetSBOffset());
    code.AddRWToStaticData(1, comment, reference);

    // Add to the identifier table as a one-dimensional associative array
    identifierTable.AddToTable(identifier, PROGRAMMODULE_VARIABLE, ASSOCTYPE, reference, 1);

    // Define a word in STATICDATA with the capacity of the associative array
    sprintf(operand, "0D%d", capacity);
    code.AddDWToStaticData(operand, "", reference);

    // Reserve words for each (key, value) pair in the associative array
    code.AddRWToStaticData(2 * capacity, "", reference);

break;
```

```
case SUBPROGRAMMODULESCOPE:

    code.IncrementFBOffset(1 + 2 * capacity); // not 2+2*capacity because 1 word
    base = code.GetFBOffset();                // is already available because of the
    code.IncrementFBOffset(1);                 // "FBOffset points-to next available word" rule
    sprintf(reference, "FB:0D%d", base);
    identifierTable.AddToTable(identifier, SUBPROGRAMMODULE_VARIABLE, ASSOCTYPE, reference, 1);

    // Add the initial value for the size, 0, to the activation record
    sprintf(reference, "FB:0D%d", base);
    sprintf(operand, "#0D0");
    sprintf(comment, "initialize associative array %s at FB:0D%d", identifier, base);
    code.AddInstructionToInitializeFrameData("PUSH", operand, comment);
    code.AddInstructionToInitializeFrameData("POP", reference);

    // Add the capacity to the activation record
    sprintf(reference, "FB:0D%d", base - 1);
    sprintf(operand, "#0D%d", capacity);
    code.AddInstructionToInitializeFrameData("PUSH", operand);
    code.AddInstructionToInitializeFrameData("POP", reference);

    break;
}

// ENDCODEGENERATION
```

```
}

// <variableDefinitions>

else
{
/*
Set varSeries to true if a datatype is followed by a comma-separated list of variables.
e.g.,      int w, x, bool y, z;

The above statement defines two integer variables w and x
and two boolean variables y and z.

*/
bool varSeries = false;
DATATYPE datatype;
bool firstLoop = true;

do
{
    char identifier[MAXIMUMLENGTHIDENTIFIER + 1];
    char operand[MAXIMUMLENGTHIDENTIFIER + 1];
    char comment[MAXIMUMLENGTHIDENTIFIER + 1];
    char reference[MAXIMUMLENGTHIDENTIFIER + 1];
    bool isInTable;
    int index;
    int dimensions; // number of dimensions in array (0 is scalar variable)
```

```
vector<int> capacities; // vector of the n capacities in an n-dimensional array
int totalCapacity; // the total capacity of the array

char enumType[MAXIMUMLENGTHIDENTIFIER + 1]; // the enumType for enumerated variables

if (firstLoop)
{
    firstLoop = false;
}
else
{
    GetNextToken(tokens);
}

// Check if the token is <datatype>
switch (tokens[0].type)
{
case INT:
    datatype = INTEGERTYPE;
    if (!varSeries)
        varSeries = true;
    GetNextToken(tokens);
    break;

case BOOL:
    datatype = BOOLEANTYPE;
    if (!varSeries)
```

```
    varSeries = true;

    GetNextToken(tokens);

    break;

case CHAR:

    datatype = CHARACTERTYPE;

    if (!varSeries)

        varSeries = true;

    GetNextToken(tokens);

    break;

case FLOAT:

    datatype = FLOATTTYPE;

    if (!varSeries)

        varSeries = true;

    GetNextToken(tokens);

    break;

case PTR:

    datatype = POINTERTYPE;

    if (!varSeries)

        varSeries = true;

    GetNextToken(tokens);

    break;

case FILE_T:

    datatype = FILETYPE;

    if (!varSeries)

        varSeries = true;

    GetNextToken(tokens);
```

```
        break;

    case IDENTIFIER:

        // Get the index of the identifier to check its datatype.
        index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);

        if (!varSeries)
        {

            // If there isn't a variable series AND <identifier> is NOT an <enumType>:
            if (identifierTable.GetType(index) != GLOBAL_ENUMTYPE &&
                identifierTable.GetType(index) != PROGRAMMODULE_ENUMTYPE &&
                identifierTable.GetType(index) != SUBPROGRAMMODULE_ENUMTYPE)
            {
                ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                    "Expecting 'int', 'bool', 'char', 'float', 'ptr', 'file', or an enum type");
            }

            // If there isn't a variable series BUT <identifier> IS an <enumType>:
            datatype = ENUMTYPE;
            strcpy(enumType, tokens[0].lexeme);
            varSeries = true;
            GetNextToken(tokens);

        }

    }
```

```
    else
    {
        // If there IS a variable series AND <identifier> IS an <enumType>:
        if (identifierTable.GetType(index) == GLOBAL_ENUMTYPE ||
            identifierTable.GetType(index) == PROGRAMMODULE_ENUMTYPE ||
            identifierTable.GetType(index) == SUBPROGRAMMODULE_ENUMTYPE)
        {

            datatype = ENUMTYPE;
            strcpy(enumType, tokens[0].lexeme);
            GetNextToken(tokens);
        }

    }

    break;

default:
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting 'int', 'bool', 'char', 'float', 'ptr', 'file', or an enum type");
}

// Check if token is <identifier>
if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
strcpy(identifier, tokens[0].lexeme);
```

```
GetNextToken(tokens);

index = identifierTable.GetIndex(identifier, isInTable);
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

// Set dimensions to 0 by default
dimensions = 0;

// Check for a left bracket: '['
if (tokens[0].type == L_BRACKET)
{
    // Clear the vector of n capacities
    capacities.clear();

    // Initialize the total capacity to 1
    totalCapacity = 1;

    // Parse the comma separated list of capacities
    do
    {
        // Get the next token
        GetNextToken(tokens);

        // Check if the array capacity is an integer literal
    }
}
```

```
if (tokens[0].type != INTEGER)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Array capacity must be an integer literal");
}

// Push the capacity into array
capacities.push_back(atoi(tokens[0].lexeme));

// Increment dimensions
dimensions++;

// update the total capacity of the array
totalCapacity *= capacities.back();

// Get next token
GetNextToken(tokens);

} while (tokens[0].type == COMMA);

// Check for a right bracket: ']'
if (tokens[0].type != R_BRACKET)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ']'");
}
```

```
}

GetNextToken(tokens);

}

// If the variable is a scalar
if (dimensions == 0)
{
    switch (identifierScope)
    {
        case GLOBALSCOPE:
            // CODEGENERATION
            code.AddRWToStaticData(1, identifier, reference);
            // ENDCODEGENERATION

            identifierTable.AddToTable(identifier, GLOBAL_VARIABLE, datatype, reference);
            break;

        case PROGRAMMODULESCOPE:
            // CODEGENERATION
            code.AddRWToStaticData(1, identifier, reference);
            // ENDCODEGENERATION

            identifierTable.AddToTable(identifier, PROGRAMMODULE_VARIABLE, datatype, reference);
            break;

        case SUBPROGRAMMODULESCOPE:
```

```
// CODEGENERATION
sprintf(reference, "FB:0D%d", code.GetFBOffset());
code.IncrementFBOffset(1);
// ENDCODEGENERATION

identifierTable.AddToTable(identifier, SUBPROGRAMMODULE_VARIABLE, datatype, reference);
break;
}

if (datatype == ENUMTYPE)
{
    // save the enumerated type for the variable
    index = identifierTable.GetIndex(identifier, isInTable);
    identifierTable.SetEnumType(index, enumType);
}

// If the variable is an array
else
{
    // CODEGENERATION
    int base;

    switch (identifierScope)
    {
        case GLOBALSCOPE:
```

```
    sprintf(operand, "0D%d", dimensions);

    sprintf(comment, "%s at SB:0D%d", identifier, code.GetSBOffset());

    code.AddDWToStaticData(operand, comment, reference);

    identifierTable.AddToTable(identifier, GLOBAL_VARIABLE, datatype, reference, dimensions);

    for (int i = 1; i <= dimensions; i++)

    {

        sprintf(operand, "0D0");

        code.AddDWToStaticData(operand, "", reference);

        sprintf(operand, "0D%d", capacities[i - 1] - 1);

        code.AddDWToStaticData(operand, "", reference);

    }

    code.AddRWToStaticData(totalCapacity, "", reference);

    break;

case PROGRAMMODULESCOPE:

    sprintf(operand, "0D%d", dimensions);

    sprintf(comment, "%s at SB:0D%d", identifier, code.GetSBOffset());

    code.AddDWToStaticData(operand, comment, reference);

    identifierTable.AddToTable(identifier, PROGRAMMODULE_VARIABLE, datatype, reference, dimensions);

    for (int i = 1; i <= dimensions; i++)

    {

        sprintf(operand, "0D0");

        code.AddDWToStaticData(operand, "", reference);

        sprintf(operand, "0D%d", capacities[i - 1] - 1);

        code.AddDWToStaticData(operand, "", reference);

    }

    code.AddRWToStaticData(totalCapacity, "", reference);
```

```
        break;

    case SUBPROGRAMMODULESCOPE:

        code.IncrementFBOffset(2 * dimensions + totalCapacity);      // not 1+2*dimensions+totalCapacity because 1 word
        base = code.GetFBOffset();                                     // is already available because of
        code.IncrementFBOffset(1);                                    // "FBOffset points-to next available word" rule
        sprintf(reference, "FB:0D%d", base);
        identifierTable.AddToTable(identifier, SUBPROGRAMMODULE_VARIABLE, datatype, reference, dimensions);

        sprintf(reference, "FB:0D%d", base);
        sprintf(operand, "#0D%d", dimensions);
        sprintf(comment, "initialize array %s at FB:0D%d", identifier, base);
        code.AddInstructionToInitializeFrameData("PUSH", operand, comment);
        code.AddInstructionToInitializeFrameData("POP", reference);
        for (int i = 1; i <= dimensions; i++)
        {
            sprintf(operand, "#0D0");
            code.AddInstructionToInitializeFrameData("PUSH", operand);
            sprintf(reference, "FB:0D%d", base - (2 * (i - 1) + 1));
            code.AddInstructionToInitializeFrameData("POP", reference);

            sprintf(operand, "#0D%d", capacities[i - 1] - 1);
            code.AddInstructionToInitializeFrameData("PUSH", operand);
            sprintf(reference, "FB:0D%d", base - (2 * (i - 1) + 2));
            code.AddInstructionToInitializeFrameData("POP", reference);
        }
        break;
```

```
    }

    if (datatype == ENUMTYPE)
    {
        // save the enumerated type for the variable
        index = identifierTable.GetIndex(identifier, isInTable);
        identifierTable.SetEnumType(index, enumType);
    }

}

// ENDCODEGENERATION

} while (tokens[0].type == COMMA);

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");

GetNextToken(tokens);

}

}

ExitModule("DataDefinitions");

}

//-----
```

```
void ParseEnumDeclaration(TOKEN tokens[], IDENTIFIERSCOPE identifierScope)  /***added***/  
//-----  
{  
    void GetNextToken(TOKEN stokens[]);  
  
    char enumType[MAXIMUMLENGTHIDENTIFIER + 1];  
    char enumConstant[MAXIMUMLENGTHIDENTIFIER + 1];  
    int index;  
    bool isInTable;  
  
    EnterModule("EnumDeclaration");  
  
    // Get the next token  
    GetNextToken(tokens);  
  
    // Check for colon  
    if (tokens[0].type != COLON)  
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ':'");  
    GetNextToken(tokens);  
  
    // Check for <enumType> ::= <identifier>  
    if (tokens[0].type != IDENTIFIER)  
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");  
  
    strcpy(enumType, tokens[0].lexeme);  
    index = identifierTable.GetIndex(enumType, isInTable);
```

```
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

// Add to the identifier table as an enumeration type
switch (identifierScope)
{
case GLOBALSCOPE:
    identifierTable.AddToTable(enumType, GLOBAL_ENUMTYPE, NOTYPE, "\0");
    break;
case PROGRAMMODULESCOPE:
    identifierTable.AddToTable(enumType, PROGRAMMODULE_ENUMTYPE, NOTYPE, "\0");
    break;
case SUBPROGRAMMODULESCOPE:
    identifierTable.AddToTable(enumType, SUBPROGRAMMODULE_ENUMTYPE, NOTYPE, "\0");
    break;
}

// Get the next token
GetNextToken(tokens);

// Check for left brace: '{'
if (tokens[0].type != L_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '{'");

// Scan all the enum constants
```

```
do
{
    // Get the next token
    GetNextToken(tokens);

    // Check for <identifier>
    if (tokens[0].type != IDENTIFIER)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");

    strcpy(enumConstant, tokens[0].lexeme);

    // Ensure that all values for enumType are unique
    for (string value : enumMap[identifierScope][enumType])
    {
        if (value == enumConstant)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Duplicate enumeration constant");
    }

    // Store the enumConstant as a value for enumType
    enumMap[identifierScope][enumType].push_back(enumConstant);

    GetNextToken(tokens);

} while (tokens[0].type == COMMA);

// Check for right brace: '}'
```

```
if (tokens[0].type != R_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '}');");
GetNextToken(tokens);

// Check for semicolon: ';'
if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'\"");
GetNextToken(tokens);

// *** TESTING that enumMap works *** (it works!)
#ifndef TESTENUMMAP
for (auto entry : enumMap)
{
    // Prints:
    // '0' if GLOBALSCOPE
    // '1' if PROGRAMMODULESCOPE
    // '2' if SUBPROGRAMMODULESCOPE
    cout << entry.first << " | ";

    for (auto val : entry.second)
    {
        cout << val.first << " : ";

        for (string constant : val.second)
        {
            cout << constant << ", ";
        }
    }
}
```

```
    }

    cout << "|| ";

}

cout << endl;

}

#endif

ExitModule("EnumDeclaration");

}

//-----
void ParseProcedureDefinition(TOKEN tokens[])
//-----
{

    void ParseFormalParameter(TOKEN tokens[], IDENTIFIERTYPE & identifierType, int& n);
    void ParseEnumDeclaration(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);
    void ParseDataDefinitions(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);
    void ParseStatement(TOKEN tokens[]);
    void GetNextToken(TOKEN tokens[]);
```

```
bool isInTable;
char line[SOURCELINELENGTH + 1];
int index;
char reference[SOURCELINELENGTH + 1];

// n = # formal parameters, m = # words of "save-register" space and locally-defined variables/constants
int n, m;
char label[SOURCELINELENGTH + 1], operand[SOURCELINELENGTH + 1], comment[SOURCELINELENGTH + 1];

EnterModule("ProcedureDefinition");

GetNextToken(tokens);

if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");

index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

identifierTable.AddToTable(tokens[0].lexeme, PROCEDURE_SUBPROGRAMMODULE, NOTYPE, tokens[0].lexeme);

// CODEGENERATION
code.EnterModuleBody(PROCEDURE_SUBPROGRAMMODULE, index);
code.ResetFrameData();
```

```
code.EmitUnformattedLine("; **** ======");
sprintf(line, "; **** PROCEDURE module (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
code.EmitUnformattedLine("; **** ======");
code.EmitFormattedLine(tokens[0].lexeme, "EQU", "*");

// ENDCODEGENERATION

identifierTable.EnterNestedStaticScope();

GetNextToken(tokens);
n = 0;
if (tokens[0].type == L_PAREN)
{
    do
    {
        IDENTIFIERTYPE identifierType;

        GetNextToken(tokens);
        ParseFormalParameter(tokens, identifierType, n);
    } while (tokens[0].type == COMMA);

    if (tokens[0].type != R_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')''");
    GetNextToken(tokens);
}
```

```
#ifdef TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table after compilation of PROCEDURE module header");
#endif

// CODEGENERATION
    code.IncrementFBOffset(2); // makes room in frame for caller's saved FB register and the CALL return address
// ENDCODEGENERATION

    while (tokens[0].type == ENUM)
    {
        ParseEnumDeclaration(tokens, SUBPROGRAMMODULESCOPE);
    }

    ParseDataDefinitions(tokens, SUBPROGRAMMODULESCOPE);

#endif TRACECOMPILER

    identifierTable.DisplayTableContents("Contents of identifier table after compilation of PROCEDURE local data definitions");
#endif

// CODEGENERATION
    m = code.GetFBOffset() - (n + 2);
    code.EmitFormattedLine("", "PUSHSP", "", "set PROCEDURE module FB = SP-on-entry + 2(n+2)");
    sprintf(operand, "#0D%d", 2 * (n + 2));
    sprintf(comment, "n = %d", n);
    code.EmitFormattedLine("", "PUSH", operand, comment);
    code.EmitFormattedLine("", "ADDI");
```

```
code.EmitFormattedLine("", "POPFB");

code.EmitFormattedLine("", "PUSHSP", "", "PROCEDURE module SP = SP-on-entry + 2m");
sprintf(operand, "#0D%d", 2 * m);
sprintf(comment, "m = %d", m);
code.EmitFormattedLine("", "PUSH", operand, comment);

code.EmitFormattedLine("", "SUBI");

code.EmitFormattedLine("", "POPSP");

code.EmitUnformattedLine("; statements to initialize frame data (if necessary)");

code.EmitFrameData();

sprintf(label, "MODULEBODY%04d", code.LabelSuffix());

code.EmitFormattedLine("", "CALL", label);

code.EmitFormattedLine("", "PUSHFB", "", "restore caller's SP-on-entry = FB - 2(n+2)");
sprintf(operand, "#0D%d", 2 * (n + 2));
code.EmitFormattedLine("", "PUSH", operand);

code.EmitFormattedLine("", "SUBI");

code.EmitFormattedLine("", "POPSP");

code.EmitFormattedLine("", "RETURN", "", "return to caller");

code.EmitUnformattedLine("");

code.EmitFormattedLine(label, "EQU", "*");

code.EmitUnformattedLine("; statements in body of PROCEDURE module (may include RETURN)");

// ENDCODEGENERATION

while (tokens[0].type != ENDPROC)
    ParseStatement(tokens);

// CODEGENERATION
```

```
code.EmitFormattedLine("", "RETURN");

code.EmitUnformattedLine("");
code.EmitUnformattedLine("; **** ======");
sprintf(line, "; **** END (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
code.EmitUnformattedLine("; **** ======");
code.ExitModuleBody();

// ENDCODEGENERATION

identifierTable.ExitNestedStaticScope();

// Clear all subprogram-scoped enumeration types from enumMap
enumMap.erase(SUBPROGRAMMODULESCOPE);

#ifndef TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table at end of compilation of PROCEDURE module definition");
#endif

GetNextToken(tokens);

ExitModule("ProcedureDefinition");
}

//-----
void ParseFunctionDefinition(TOKEN tokens[])
//-----
```

```
{  
  
void ParseFormalParameter(TOKEN tokens[], IDENTIFIERTYPE & identifierType, int& n);  
void ParseEnumDeclaration(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);  
void ParseDataDefinitions(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);  
void ParseStatement(TOKEN tokens[]);  
void GetNextToken(TOKEN tokens[]);  
  
bool isInTable;  
DATATYPE datatype;  
char identifier[SOURCELINELENGTH + 1];  
char line[SOURCELINELENGTH + 1];  
int index;  
char reference[SOURCELINELENGTH + 1];  
  
// n = # formal parameters, m = # words of return-value, "save-register" space, and locally-defined variables/constants  
int n, m;  
char label[SOURCELINELENGTH + 1], operand[SOURCELINELENGTH + 1], comment[SOURCELINELENGTH + 1];  
  
EnterModule("FunctionDefinition");  
  
GetNextToken(tokens);  
  
if (tokens[0].type != IDENTIFIER)  
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");  
  
strcpy(identifier, tokens[0].lexeme);
```

```
index = identifierTable.GetIndex(identifier, isInTable);

if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

GetNextToken(tokens);

if (tokens[0].type != COLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ':'");

GetNextToken(tokens);

switch (tokens[0].type)
{
case INT:
    datatype = INTEGERTYPE;
    break;
case BOOL:
    datatype = BOOLEANTYPE;
    break;
case CHAR:
    datatype = CHARACTERTYPE;
    break;
case FLOAT:
    datatype = FLOATTYPE;
    break;
default:
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting int, bool, char, or float");
}
```

```
GetNextToken(tokens);

identifierTable.AddToTable(identifier, FUNCTION_SUBPROGRAMMODULE, datatype, identifier);
index = identifierTable.GetIndex(identifier, isInTable);

// CODEGENERATION
code.EnterModuleBody(FUNCTION_SUBPROGRAMMODULE, index);
code.ResetFrameData();

// Reserve frame-space for FUNCTION return value
code.IncrementFBOffset(1);

code.EmitUnformattedLine("; **** ======");
sprintf(line, "; **** FUNCTION module (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
code.EmitUnformattedLine("; **** ======");
code.EmitFormattedLine(identifier, "EQU", "*");

// ENDCODEGENERATION

identifierTable.EnterNestedStaticScope();

n = 0;
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");

// Use token look-ahead to make parsing decision
```

```
if (tokens[1].type != R_PAREN)
{
    do
    {
        IDENTIFIERTYPE identifierType;

        GetNextToken(tokens);
        ParseFormalParameter(tokens, identifierType, n);

        // STATICSEMANTICS
        if (identifierType != IN_PARAMETER)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Function parameter must be IN");
        // ENDSTATICSEMANTICS

    } while (tokens[0].type == COMMA);
}

else
{
    GetNextToken(tokens);
    if (tokens[0].type != R_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");
    GetNextToken(tokens);

#ifdef TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table after compilation of FUNCTION module header");
#endif
}
```

```
// CODEGENERATION
    code.IncrementFBOffset(2); // makes room in frame for caller's saved FB register and the CALL return address
// ENDCODEGENERATION

    while (tokens[0].type == ENUM)
    {
        ParseEnumDeclaration(tokens, SUBPROGRAMMODULESCOPE);
    }

    ParseDataDefinitions(tokens, SUBPROGRAMMODULESCOPE);

#endif TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table after compilation of FUNCTION local data definitions");
#endif

// CODEGENERATION
    m = code.GetFBOffset() - (n + 3);
    code.EmitFormattedLine("", "PUSHSP", "", "set FUNCTION module FB = SP-on-entry + 2(n+3)");
    sprintf(operand, "#0D%d", 2 * (n + 3));
    sprintf(comment, "n = %d", n);
    code.EmitFormattedLine("", "PUSH", operand, comment);
    code.EmitFormattedLine("", "ADDI");
    code.EmitFormattedLine("", "POPFB");
    code.EmitFormattedLine("", "PUSHSP", "", "FUNCTION module SP = SP-on-entry + 2m");
    sprintf(operand, "#0D%d", 2 * m);
    sprintf(comment, "m = %d", m);
```

```
code.EmitFormattedLine("", "PUSH", operand, comment);
code.EmitFormattedLine("", "SUBI");
code.EmitFormattedLine("", "POPSP");
code.EmitUnformattedLine("; statements to initialize frame data (if necessary)");
code.EmitFrameData();
sprintf(label, "MODULEBODY%04d", code.LabelSuffix());
code.EmitFormattedLine("", "CALL", label);
code.EmitFormattedLine("", "PUSHFB", "", "restore caller's SP-on-entry = FB - 2(n+3)");
sprintf(operand, "#0D%d", 2 * (n + 3));
code.EmitFormattedLine("", "PUSH", operand);
code.EmitFormattedLine("", "SUBI");
code.EmitFormattedLine("", "POPSP");
code.EmitFormattedLine("", "RETURN", "", "return to caller");
code.EmitUnformattedLine("");
code.EmitFormattedLine(label, "EQU", "*");
code.EmitUnformattedLine("; statements in body of FUNCTION module (*MUST* execute RETURN"));
// ENDCODEGENERATION

while (tokens[0].type != ENDFUNC)
    ParseStatement(tokens);

// CODEGENERATION
sprintf(operand, "#0D%d", tokens[0].sourceLineNumber);
code.EmitFormattedLine("", "PUSH", operand);
code.EmitFormattedLine("", "PUSH", "#0D3");
code.EmitFormattedLine("", "JMP", "HANDLERUNTIMEERROR");
```

```
    code.EmitUnformattedLine("; **** ======");
    sprintf(line, "; **** END (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    code.EmitUnformattedLine("; **** ======");
    code.ExitModuleBody();

// ENDCODEGENERATION

    identifierTable.ExitNestedStaticScope();

// Clear all subprogram-scoped enumeration types from enumMap
enumMap.erase(SUBPROGRAMMODULESCOPE);

#ifndef TRACECOMPILER
    identifierTable.DisplayTableContents("Contents of identifier table at end of compilation of FUNCTION module definition");
#endif

    GetNextToken(tokens);

    ExitModule("FunctionDefinition");
}

//-----
void ParseFormalParameter(TOKEN tokens[], IDENTIFIERTYPE& identifierType, int& n)
//-----
{
    void GetNextToken(TOKEN tokens[]);
```

```
char identifier[MAXIMUMLENGTHIDENTIFIER + 1], reference[MAXIMUMLENGTHIDENTIFIER + 1];
bool isInTable;
int index;
DATATYPE datatype;

EnterModule("FormalParameter");

// CODEGENERATION
switch (tokens[0].type)
{
case IN:
    identifierType = IN_PARAMETER;
    sprintf(reference, "FB:0D%d", code.GetFBOffset());
    code.IncrementFBOffset(1);
    n += 1;
    GetNextToken(tokens);
    break;
case OUT:
    identifierType = OUT_PARAMETER;
    code.IncrementFBOffset(1);
    sprintf(reference, "FB:0D%d", code.GetFBOffset());
    code.IncrementFBOffset(1);
    n += 2;
    GetNextToken(tokens);
    break;
}
```

```
case IO:
    identifierType = IO_PARAMETER;
    code.IncrementFBOffset(1);
    sprintf(reference, "FB:0D%d", code.GetFBOffset());
    code.IncrementFBOffset(1);
    n += 2;
    GetNextToken(tokens);
    break;

case REF:
    identifierType = REF_PARAMETER;
    sprintf(reference, "@FB:0D%d", code.GetFBOffset());
    code.IncrementFBOffset(1);
    n += 1;
    GetNextToken(tokens);
    break;

default:
    identifierType = IN_PARAMETER;
    sprintf(reference, "FB:0D%d", code.GetFBOffset());
    code.IncrementFBOffset(1);
    n += 1;
    break;
}

// ENDCODEGENERATION

if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
```

```
strcpy(identifier, tokens[0].lexeme);

GetNextToken(tokens);

if (tokens[0].type != COLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ':'");

GetNextToken(tokens);

switch (tokens[0].type)
{
    case INT:
        datatype = INTEGERTYPE;
        break;
    case BOOL:
        datatype = BOOLEANTYPE;
        break;
    case CHAR:
        datatype = CHARACTERTYPE;
        break;
    case FLOAT:
        datatype = FLOATTYPE;
        break;
    default:
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting int, bool, char, or float");
}
GetNextToken(tokens);
```

```
index = identifierTable.GetIndex(identifier, isInTable);
if (isInTable && identifierTable.IsInCurrentScope(index))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Multiply-defined identifier");

identifierTable.AddToTable(identifier, identifierType, datatype, reference);

ExitModule("FormalParameter");
}

//-----
void ParseMainProgram(TOKEN tokens[])
//-----
{
    void ParseDataDefinitions(TOKEN tokens[], IDENTIFIERSCOPE identifierScope);
    void GetNextToken(TOKEN tokens[]);
    void ParseStatement(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
    char label[SOURCELINELENGTH + 1];
    char reference[SOURCELINELENGTH + 1];

    EnterModule("MainProgram");

// CODEGENERATION
    code.EmitUnformattedLine("; **** ======");
    sprintf(line, "; **** 'main' program (%4d)", tokens[0].sourceLineNumber);
```

```
code.EmitUnformattedLine(line);

code.EmitUnformattedLine("; **** ======");
code.EmitFormattedLine("MAINPROGRAM", "EQU", "*");

code.EmitFormattedLine("", "PUSH", "#RUNTIMESTACK", "set SP");
code.EmitFormattedLine("", "POPSP");
code.EmitFormattedLine("", "PUSHA", "STATICDATA", "set SB");
code.EmitFormattedLine("", "POPSB");
code.EmitFormattedLine("", "PUSH", "#HEAPBASE", "initialize heap");
code.EmitFormattedLine("", "PUSH", "#HEAPSIZE");
code.EmitFormattedLine("", "SVC", "#SVC_INITIALIZE_HEAP");
sprintf(label, "MAINBODY%04d", code.LabelSuffix());
code.EmitFormattedLine("", "CALL", label);
code.AddDSToStaticData("Normal program termination", "", reference);
code.EmitFormattedLine("", "PUSHA", reference);
code.EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");
code.EmitFormattedLine("", "SVC", "#SVC_WRITE_ENDL");
code.EmitFormattedLine("", "PUSH", "#0D0", "terminate with status = 0");
code.EmitFormattedLine("", "SVC", "#SVC_TERMINATE");
code.EmitUnformattedLine("");
code.EmitFormattedLine(label, "EQU", "*");

// ENDCODEGENERATION

GetNextToken(tokens);

identifierTable.EnterNestedStaticScope();
```

```
while (tokens[0].type == ENUM)
    ParseEnumDeclaration(tokens, PROGRAMMODULESCOPE);

ParseDataDefinitions(tokens, PROGRAMMODULESCOPE);

while (tokens[0].type != ENDMAIN)
    ParseStatement(tokens);

// CODEGENERATION
code.EmitFormattedLine("", "RETURN");
code.EmitUnformattedLine("; **** ======");
sprintf(line, "; **** 'end main' (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
code.EmitUnformattedLine("; **** ======");
// ENDCODEGENERATION

#ifndef TRACECOMPILER
identifierTable.DisplayTableContents("Contents of identifier table at end of compilation of PROGRAM module definition");
#endif

identifierTable.ExitNestedStaticScope();
GetNextToken(tokens);

ExitModule("MainProgram");
}
```

```
//-----
void ParseStatement(TOKEN tokens[])
//-----
{
    void ParseAssertion(TOKEN tokens[]);
    void ParsePrintStatement(TOKEN tokens[]);
    void ParseInputStatement(TOKEN tokens[]);
    void ParseAssignmentStatement(TOKEN tokens[]);
    void ParseIfStatement(TOKEN tokens[]);
    void ParseDoWhileStatement(TOKEN tokens[]);
    void ParseForStatement(TOKEN tokens[]);
    void ParseCallStatement(TOKEN tokens[]);
    void ParseReturnStatement(TOKEN tokens[]);
    void GetNextToken(TOKEN tokens[]);

    void ParseFilePrintStatement(TOKEN tokens[]);
    void ParseFileInputStatement(TOKEN tokens[]);

    EnterModule("Statement");

    while ( tokens[0].type == ASSERT )
        ParseAssertion(tokens);
    switch ( tokens[0].type )
    {
```

```
case PRINT:  
    ParsePrintStatement(tokens);  
    break;  
  
case INPUT:  
    ParseInputStatement(tokens);  
    break;  
  
case IDENTIFIER:  
    ParseAssignmentStatement(tokens);  
    break;  
  
case IF:  
    ParseIfStatement(tokens);  
    break;  
  
case DO:  
  
case WHILE:  
    ParseDoWhileStatement(tokens);  
    break;  
  
case FOR:  
    ParseForStatement(tokens);  
    break;  
  
case CALL:  
    ParseCallStatement(tokens);  
    break;  
  
case RETURN:  
    ParseReturnStatement(tokens);  
    break;  
  
case F_PRINT:
```

```
ParseFilePrintStatement(tokens);
break;

case F_INPUT:
ParseFileInputStatement(tokens);
break;

default:
ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,
                     "Expecting beginning-of-statement");
break;
}

while ( tokens[0].type == ASSERT )
ParseAssertion(tokens);

ExitModule("Statement");
}

//-----
void ParseAssertion(TOKEN tokens[])
//-----
{
void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
void GetNextToken(TOKEN tokens[]);

char line[SOURCELINELENGTH + 1];
```

```
DATATYPE datatype;

EnterModule("Assertion");

sprintf(line, "; **** %4d: assertion", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);

GetNextToken(tokens);

if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '(' ");
GetNextToken(tokens);

ParseExpression(tokens, datatype);

// STATICSEMANTICS
if (datatype != BOOLEANTYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean expression");
// ENDSTATICSEMANTICS

// CODEGENERATION
/*
SETT
JMPT      E?????
PUSH      #0D(sourceLineNumber)
PUSH      #0D1
```

```

JMP      HANDLERUNTIMEERROR

E???? EQU      *
DISCARD #0D1

*/
char Elabel[SOURCELINELENGTH + 1], operand[SOURCELINELENGTH + 1];

code.EmitFormattedLine("", "SETT");
sprintf(Elabel, "E%04d", code.LabelSuffix());
code.EmitFormattedLine("", "JMPT", Elabel);
sprintf(operand, "#0D%d", tokens[0].sourceLineNumber);
code.EmitFormattedLine("", "PUSH", operand);
code.EmitFormattedLine("", "PUSH", "#0D1");
code.EmitFormattedLine("", "JMP", "HANDLERUNTIMEERROR");
code.EmitFormattedLine(Elabel, "EQU", "*");
code.EmitFormattedLine("", "DISCARD", "#0D1");

// ENDCODEGENERATION

if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");
GetNextToken(tokens);

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");
GetNextToken(tokens);

ExitModule("Assertion");

```

```
}
```

```
-----
```

```
void ParsePrintStatement(TOKEN tokens[])
-----
```

```
{
```

```
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
    DATATYPE datatype;

    EnterModule("PrintStatement");

    GetNextToken(tokens);

    // CODEGENERATION
    sprintf(line, "; **** 'print' statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    // ENDCODEGENERATION

    //Check if the next token is a left parentheses
    if (tokens[0].type != L_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Expecting '(';
```

```
do
{
    GetNextToken(tokens);

    if (tokens[0].type == STRING) {
        // CODEGENERATION
        char reference[SOURCELINELENGTH + 1];

        code.AddDSToStaticData(tokens[0].lexeme, "", reference);
        code.EmitFormattedLine("", "PUSHA", reference);
        code.EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");
        // ENDCODEGENERATION

        GetNextToken(tokens);
    }
    else {
        ParseExpression(tokens, datatype);

        // CODEGENERATION
        switch (datatype)
        {
            case INTEGERTYPE:
            case ENUMTYPE:
                code.EmitFormattedLine("", "SVC", "#SVC_WRITE_INTEGER");
                break;
        }
    }
}
```

```
        case BOOLEANTYPE:
            code.EmitFormattedLine("", "SVC", "#SVC_WRITE_BOOLEAN");
            break;
        case CHARACTERTYPE:
            code.EmitFormattedLine("", "SVC", "#SVC_WRITE_CHARACTER");
            break;
        case FLOATTYPE:
            code.EmitFormattedLine("", "SVC", "#SVC_WRITE_FLOAT");
            break;
        case POINTERTYPE:
            code.EmitFormattedLine("", "SVC", "#SVC_WRITE_HEXADECIMAL");
            break;
    }
    // ENDCODEGENERATION

}

} while (tokens[0].type == COMMA);

//Check if the next token is a right parentheses
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ')''");

GetNextToken(tokens);

//Check if the next token is a semicolon
```

```
if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ';'");

GetNextToken(tokens);

ExitModule("PrintStatement");
}

//-----
void ParseInputStatement(TOKEN tokens[])
//-----
{
    void ParseVariable(TOKEN tokens[],bool asLValue,DATATYPE &datatype);
    void GetNextToken(TOKEN tokens[]);

    char reference[SOURCELINELENGTH+1];
    char line[SOURCELINELENGTH+1];
    DATATYPE datatype;

    EnterModule("InputStatement");

    sprintf(line," *** 'input' statement (%4d)",tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);

    GetNextToken(tokens);
```

```
//Check if the next token is a left parentheses
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '(');
GetNextToken(tokens);

if ( tokens[0].type == STRING )
{
    // CODEGENERATION
    code.AddDSToStaticData(tokens[0].lexeme,"",reference);
    code.EmitFormattedLine("", "PUSHA", reference);
    code.EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");
    // ENDCODEGENERATION

    GetNextToken(tokens);

    //Check if the next token is a comma
    if (tokens[0].type != COMMA)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Expecting ','");
    GetNextToken(tokens);
}

ParseVariable(tokens,true,datatype);
```

```

// CODEGENERATION

switch ( datatype )
{
    case INTEGERTYPE:
        code.EmitFormattedLine("", "SVC", "#SVC_READ_INTEGER");
        break;
    case BOOLEANTYPE:
        code.EmitFormattedLine("", "SVC", "#SVC_READ_BOOLEAN");
        break;
    case CHARACTERTYPE:
        code.EmitFormattedLine("", "SVC", "#SVC_READ_CHARACTER");
        break;
    case FLOATTYPET:
        code.EmitFormattedLine("", "SVC", "#SVC_READ_FLOAT");
        break;
}
code.EmitFormattedLine("", "POP", "@SP:0D1");
code.EmitFormattedLine("", "DISCARD", "#0D1");

// ENDCODEGENERATION

//Check if the next token is a right parentheses
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                         "Expecting ')''");
GetNextToken(tokens);

```

```
//Check if the next token is a semicolon
if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ';'");

GetNextToken(tokens);

ExitModule("InputStatement");
}

//-----
void ParseAssignmentStatement(TOKEN tokens[]) /*** updated ***/
//-----
{

    void ParseVariable(TOKEN tokens[],bool asLValue,DATATYPE &datatype);
    void ParseExpression(TOKEN tokens[],DATATYPE &datatype);
    void ParseAssociativeArrayReference(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    void ParseFileOperationStatement(TOKEN tokens[]);

char line[SOURCELINELENGTH+1];
```

```
EnterModule("AssignmentStatement");

sprintf(line,"; **** assignment statement (%4d)",tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);

int index;
bool isInTable;
DATATYPE datatype;

// STATICSEMANTICS
index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Undefined identifier");

datatype = identifierTable.GetDatatype(index);

// <FileOperationStatement>
if (datatype == FILETYPE)
{
    ParseVariable(tokens, true, datatype);
    ParseFileOperationStatement(tokens);
}

// <AssociativeArrayReference> = <expression>;
```

```
else if (datatype == ASSOCTYPE)
{
    DATATYPE valueDatatype;

    ParseAssociativeArrayReference(tokens, true, datatype);

    if (tokens[0].type != ASSIGN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '='");
    GetNextToken(tokens);

    ParseExpression(tokens, valueDatatype);

    // Check that the value is a scalar datatype
    if ((valueDatatype != INTEGERTYPE) && (valueDatatype != FLOATTYPE) &&
        (valueDatatype != CHARACTERTYPE) && (valueDatatype != BOOLEANTYPE))
    {
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Value must be of a scalar datatype (int, bool, char, or float)");
    }

    //CODEGENERATION
    /** Add the (key, value) pair to the associative array and increment the size ***/
    code.EmitFormattedLine("", "SWAP");
    code.EmitFormattedLine("", "SETAAE", identifierTable.GetReference(index));
    //ENDCODEGENERATION
}
```

```
// <variableList> = <expression>;
else
{
    DATATYPE datatypeLHS,datatypeRHS;
    int n;
    ParseVariable(tokens, true, datatypeLHS);
    n = 1;

    while (tokens[0].type == COMMA)
    {
        DATATYPE datatypeNEXT;
        int indexNEXT;

        GetNextToken(tokens);

        indexNEXT = identifierTable.GetIndex(tokens[0].lexeme, isInTable);

        ParseVariable(tokens, true, datatypeNEXT);
        n++;

        if (datatypeNEXT != datatypeLHS)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Mixed-mode variables not allowed");

        if (datatypeNEXT == ENUMTYPE)
    {
```

```
    if (identifierTable.GetEnumType(indexNEXT) != identifierTable.GetEnumType(index))
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Mixed-mode variables not allowed");
}

}

if (tokens[0].type != ASSIGN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '='");
GetNextToken(tokens);

// If the datatype of the LHS is an enumeration type
if (datatype == ENUMTYPE)
{
    // Save the scope and enum type of the l-value(s)
    switch (identifierTable.GetType(index))
    {
        case GLOBAL_VARIABLE:
            lValueEnumScope = GLOBALSCOPE;
            break;
        case PROGRAMMODULE_VARIABLE:
            lValueEnumScope = PROGRAMMODULESCOPE;
            break;
        case SUBPROGRAMMODULE_VARIABLE:
            lValueEnumScope = SUBPROGRAMMODULESCOPE;
            break;
    }
}
```

```
lValueEnumType = identifierTable.GetEnumType(index);

// and if the token about to be parsed is an identifier

if (tokens[0].type == IDENTIFIER)
{
    // Check if the identifier is an enum type

    int indexRHS = identifierTable.GetIndex(tokens[0].lexeme, isInTable);

    // If so, see if its enum type matches that of the LHS

    if (identifierTable.GetDatatype(indexRHS) == ENUMTYPE &&
        identifierTable.GetEnumType(index) != identifierTable.GetEnumType(indexRHS))
    {
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Data type mismatch");
    }
}

ParseExpression(tokens, datatypeRHS);

if (datatypeLHS != datatypeRHS)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Data type mismatch");

// CODEGENERATION

for (int i = 1; i <= n; i++)
{
```

```
        code.EmitFormattedLine("", "MAKEDUP");

        code.EmitFormattedLine("", "POP", "@SP:0D2");
        code.EmitFormattedLine("", "SWAP");
        code.EmitFormattedLine("", "DISCARD", "#0D1");

    }

    code.EmitFormattedLine("", "DISCARD", "#0D1");
    // ENDCODEGENERATION

}

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");

GetNextToken(tokens);

// clear the vector of pointer indexes
ptr_indexes.clear();

// reset the lValueEnumType to ""
lValueEnumType = "";

ExitModule("AssignmentStatement");

}

//-----
void ParseIfStatement(TOKEN tokens[])
//-----
```

```
{  
  
void ParseExpression(TOKEN tokens[], DATATYPE & datatype);  
void ParseStatement(TOKEN tokens[]);  
void GetNextToken(TOKEN tokens[]);  
  
char line[SOURCELINELENGTH + 1];  
char Ilabel[SOURCELINELENGTH + 1], Elabel[SOURCELINELENGTH + 1];  
DATATYPE datatype;  
  
EnterModule("IfStatement");  
  
sprintf(line, "; **** 'if' statement (%4d)", tokens[0].sourceLineNumber);  
code.EmitUnformattedLine(line);  
  
GetNextToken(tokens);  
  
if (tokens[0].type != L_PAREN)  
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");  
GetNextToken(tokens);  
ParseExpression(tokens, datatype);  
if (tokens[0].type != R_PAREN)  
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");  
GetNextToken(tokens);  
  
if (datatype != BOOLEANTYPE)  
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean expression");
```

```
// CODEGENERATION
/*
Plan for the generalized IF statement with n ELIFs and 1 ELSE (*Note* n
can be 0 and the ELSE may be missing and the plan still "works.")

...expression...           ; boolean expression on top-of-stack

SETT
DISCARD #0D1
JMPNT I??1

...statements...
JMP E???
I??1 EQU *           ; 1st ELIF clause

...expression...
SETT
DISCARD #0D1
JMPNT I??2

...statements...
JMP E???
.

.

I??n EQU *           ; nth ELIF clause

...expression...
SETT
DISCARD #0D1
JMPNT I???
```

```

...statements...

JMP      E????  

I???? EQU      *           ; ELSE clause  

...statements...
E???? EQU      *  

*/  

sprintf(Elabel, "E%04d", code.LabelSuffix());  

code.EmitFormattedLine("", "SETT");  

code.EmitFormattedLine("", "DISCARD", "#0D1");  

sprintf(Ilabel, "I%04d", code.LabelSuffix());  

code.EmitFormattedLine("", "JMPNT", Ilabel);  

// ENDCODEGENERATION

while ( (tokens[0].type != ELIF) &&  

       (tokens[0].type != ELSE) &&  

       (tokens[0].type != ENDIF))  

ParseStatement(tokens);

// CODEGENERATION
code.EmitFormattedLine("", "JMP", Elabel);
code.EmitFormattedLine(Ilabel, "EQU", "*");
// ENDCODEGENERATION

while (tokens[0].type == ELIF)
{
  GetNextToken(tokens);
}

```

```
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");
GetNextToken(tokens);
ParseExpression(tokens, datatype);
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");
GetNextToken(tokens);

if (datatype != BOOLEANTYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean expression");

// CODEGENERATION
code.EmitFormattedLine("", "SETT");
code.EmitFormattedLine("", "DISCARD", "#0D1");
sprintf(Ilabel, "I%04d", code.LabelSuffix());
code.EmitFormattedLine("", "JMPNT", Ilabel);
// ENDCODEGENERATION

while ((tokens[0].type != ELIF) &&
       (tokens[0].type != ELSE) &&
       (tokens[0].type != ENDIF))
    ParseStatement(tokens);

// CODEGENERATION
code.EmitFormattedLine("", "JMP", Elabel);
code.EmitFormattedLine(Ilabel, "EQU", "*");
```

```
// ENDCODEGENERATION
}

if (tokens[0].type == ELSE)
{
    GetNextToken(tokens);
    while (tokens[0].type != ENDIF)
        ParseStatement(tokens);

}

GetNextToken(tokens);

// CODEGENERATION
code.EmitFormattedLine(Elabel, "EQU", "*");

// ENDCODEGENERATION

ExitModule("IfStatement");
}

//-----
void ParseDowhileStatement(TOKEN tokens[])
//-----
{
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void ParseStatement(TOKEN tokens[]);
    void GetNextToken(TOKEN tokens[]);
```

```
char line[SOURCELINELENGTH + 1];
char Dlabel[SOURCELINELENGTH + 1], Elabel[SOURCELINELENGTH + 1];
DATATYPE datatype;

EnterModule("DoWhileStatement");

sprintf(line, "; **** 'do/while' statement (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);

// CODEGENERATION
/*
D???? EQU      *
...statements...
...expression...
SETT
DISCARD #0D1
JMPNT   E???
...statements...
JMP      D?????
E???? EQU      *
*/
sprintf(Dlabel, "D%04d", code.LabelSuffix());
sprintf(Elabel, "E%04d", code.LabelSuffix());
code.EmitFormattedLine(Dlabel, "EQU", "*");
// ENDCODEGENERATION
```

```
if (tokens[0].type == DO) {
    GetNextToken(tokens);

    while (tokens[0].type != WHILE)
        ParseStatement(tokens);

    GetNextToken(tokens);

    if (tokens[0].type != L_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");
    GetNextToken(tokens);
    ParseExpression(tokens, datatype);
    if (tokens[0].type != R_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");
    GetNextToken(tokens);

    if (datatype != BOOLEANTYPE)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean expression");

// CODEGENERATION
code.EmitFormattedLine("", "SETT");
code.EmitFormattedLine("", "DISCARD", "#0D1");
code.EmitFormattedLine("", "JMPNT", Elabel);
// ENDCODEGENERATION
```

```
    while (tokens[0].type != ENDWHILE)
        ParseStatement(tokens);

    GetNextToken(tokens);

// CODEGENERATION
    code.EmitFormattedLine("", "JMP", Dlabel);
    code.EmitFormattedLine(Elabel, "EQU", "*");
// ENDCODEGENERATION

    ExitModule("DoWhileStatement");
}

//-----
void ParseForStatement(TOKEN tokens[])
//-----
{
    void ParseVariable(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void ParseStatement(TOKEN tokens[]);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
    char Dlabel[SOURCELINELENGTH + 1], Llabel[SOURCELINELENGTH + 1],
        Clabel[SOURCELINELENGTH + 1], Elabel[SOURCELINELENGTH + 1];
```

```
char operand[SOURCELINELENGTH + 1];
DATATYPE datatype;

EnterModule("ForStatement");

sprintf(line, "; **** 'for' statement (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);

GetNextToken(tokens);

ParseVariable(tokens, true, datatype);

if (datatype != INTEGERTYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer variable");

/*
; v := e1
....v...          ; &v = run-time stack (bottom to top)
....e1...         ; &v,e1
POP      @SP:0D1   ; &v
....e2...         ; &v,e2
....e3...         ; &v,e2,e3
SETNZPI
; if ( e3 = 0 ) then
JMPNZ    D?????
PUSH    #0D(current line number)
```

```

PUSH      #0D2
JMP      HANDLERUNTIMEERROR
D????  SETNZPI
; else if ( e3 > 0 ) then
JMPN      L?????
SWAP          ; &v,e3,e2
MAKEDUP        ; &v,e3,e2,e2
PUSH      @SP:0D3      ; &v,e3,e2,e2,v
SWAP          ; &v,e3,e2,v,e2
;   if ( v <= e2 ) continue else end
CMPI          ; &v,e3,e2 (set LEG)
JMPLE     C?????
JMP      E?????
; else ( e3 < 0 )
L????? SWAP          ; &v,e3,e2
MAKEDUP        ; &v,e3,e2,e2
PUSH      @SP:0D3      ; &v,e3,e2,e2,v
SWAP          ; &v,e3,e2,v,e2
;   if ( v >= e2 ) continue else end
CMPI          ; &v,e3,e2 (set LEG)
JMPGE     C?????
JMP      E?????
; endif
C????? EQU      *
...statements...
SWAP          ; &v,e2,e3

```

```

MAKEDUP           ; &v,e2,e3,e3
; v := e3+v

PUSH    @SP:0D3      ; &v,e2,e3,e3,v
ADDI          ; &v,e2,e3,(e3+v)
POP     @SP:0D3      ; &v,e2,e3
JMP      D?????
E????? DISCARD #0D3      ; now run-time stack is empty
*/
}

if (tokens[0].type != ASSIGN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '='");
GetNextToken(tokens);

ParseExpression(tokens, datatype);
if (datatype != INTEGERTYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer data type");

// CODEGENERATION
code.EmitFormattedLine("", "POP", "@SP:0D1");
// ENDCODEGENERATION

if (tokens[0].type != T0)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting T0");
GetNextToken(tokens);

ParseExpression(tokens, datatype);

```

```
if (datatype != INTEGERTYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer data type");

if (tokens[0].type == BY)
{
    GetNextToken(tokens);

    ParseExpression(tokens, datatype);
    if (datatype != INTEGERTYPE)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer data type");
}

else
{

// CODEGENERATION
    code.EmitFormattedLine("", "PUSH", "#0D1");
// ENDCODEGENERATION

}

// CODEGENERATION
    sprintf(Dlabel, "D%04d", code.LabelSuffix());
    sprintf(Llabel, "L%04d", code.LabelSuffix());
    sprintf(Clabel, "C%04d", code.LabelSuffix());
    sprintf(Elabel, "E%04d", code.LabelSuffix());
```

```
code.EmitFormattedLine("", "SETNZPI");

code.EmitFormattedLine("", "JMPNZ", Dlabel);
sprintf(operand, "#0D%d", tokens[0].sourceLineNumber);

code.EmitFormattedLine("", "PUSH", operand);
code.EmitFormattedLine("", "PUSH", "#0D2");
code.EmitFormattedLine("", "JMP", "HANDLERUNTIMEERROR");

code.EmitFormattedLine(Dlabel, "SETNZPI");
code.EmitFormattedLine("", "JMPN", Llabel);
code.EmitFormattedLine("", "SWAP");
code.EmitFormattedLine("", "MAKEDUP");
code.EmitFormattedLine("", "PUSH", "@SP:0D3");
code.EmitFormattedLine("", "SWAP");
code.EmitFormattedLine("", "CMPI");
code.EmitFormattedLine("", "JMPLE", Clabel);
code.EmitFormattedLine("", "JMP", Elabel);
code.EmitFormattedLine(Llabel, "SWAP");
code.EmitFormattedLine("", "MAKEDUP");
code.EmitFormattedLine("", "PUSH", "@SP:0D3");
code.EmitFormattedLine("", "SWAP");
code.EmitFormattedLine("", "CMPI");
code.EmitFormattedLine("", "JMPGE", Clabel);
code.EmitFormattedLine("", "JMP", Elabel);
code.EmitFormattedLine(Clabel, "EQU", "*");

// ENDCODEGENERATION
```

```
while (tokens[0].type != ENDFOR)
    ParseStatement(tokens);

GetNextToken(tokens);

// CODEGENERATION
code.EmitFormattedLine("", "SWAP");
code.EmitFormattedLine("", "MAKEDUP");
code.EmitFormattedLine("", "PUSH", "@SP:0D3");
code.EmitFormattedLine("", "ADDI");
code.EmitFormattedLine("", "POP", "@SP:0D3");
code.EmitFormattedLine("", "JMP", Dlabel);
code.EmitFormattedLine(Elabel, "DISCARD", "#0D3");
// ENDCODEGENERATION

ExitModule("ForStatement");
}

//-----
void ParseCallStatement(TOKEN tokens[])
//-----
{
    void ParseVariable(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);
```

```
char line[SOURCELINELENGTH + 1];
bool isInTable;
int index, parameters;

EnterModule("CallStatement");

sprintf(line, "; **** Call statement (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);

GetNextToken(tokens);

if (tokens[0].type != IDENTIFIER)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");

// STATICSEMANTICS
index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Undefined identifier");
if (identifierTable.GetType(index) != PROCEDURE_SUBPROGRAMMODULE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting PROCEDURE identifier");
// ENDSTATICSEMANTICS

GetNextToken(tokens);
parameters = 0;
if (tokens[0].type == L_PAREN)
{
```

```
DATATYPE expressionDatatype, variableDatatype;

do
{
    GetNextToken(tokens);
    parameters++;

// CODEGENERATION
// STATICSEMANTICS
    switch (identifierTable.GetType(index + parameters))
    {
        case IN_PARAMETER:
            ParseExpression(tokens, expressionDatatype);
            if (expressionDatatype != identifierTable.GetDatatype(index + parameters))
                ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                    "Actual parameter data type does not match formal parameter data type");
            break;
        case OUT_PARAMETER:
            ParseVariable(tokens, true, variableDatatype);
            if (variableDatatype != identifierTable.GetDatatype(index + parameters))
                ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                    "Actual parameter data type does not match formal parameter data type");
            code.EmitFormattedLine("", "PUSH", "#0X0000");
            break;
        case IO_PARAMETER:
            ParseVariable(tokens, true, variableDatatype);
```

```
    if (variableDatatype != identifierTable.GetDatatype(index + parameters))
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Actual parameter data type does not match formal parameter data type");
    code.EmitFormattedLine("", "PUSH", "@SP:0D0");
    break;

    case REF_PARAMETER:
        ParseVariable(tokens, true, variableDatatype);
        if (variableDatatype != identifierTable.GetDatatype(index + parameters))
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                "Actual parameter data type does not match formal parameter data type");
        break;
    }

// ENDSTATICSEMANTICS
// ENDCODEGENERATION

} while (tokens[0].type == COMMA);

if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting )");

GetNextToken(tokens);
}

// STATICSEMANTICS

if (identifierTable.GetCountOfFormalParameters(index) != parameters)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Number of actual parameters does not match number of formal parameters");
```

```
// ENDSTATICSEMANTICS

// CODEGENERATION

code.EmitFormattedLine("", "PUSHFB");

code.EmitFormattedLine("", "CALL", identifierTable.GetReference(index));

code.EmitFormattedLine("", "POPFB");

for (parameters = identifierTable.GetCountOfFormalParameters(index); parameters >= 1; parameters--)

{

    switch (identifierTable.GetType(index + parameters))

    {

        case IN_PARAMETER:

            code.EmitFormattedLine("", "DISCARD", "#0D1");

            break;

        case OUT_PARAMETER:

            code.EmitFormattedLine("", "POP", "@SP:0D1");

            code.EmitFormattedLine("", "DISCARD", "#0D1");

            break;

        case IO_PARAMETER:

            code.EmitFormattedLine("", "POP", "@SP:0D1");

            code.EmitFormattedLine("", "DISCARD", "#0D1");

            break;

        case REF_PARAMETER:

            code.EmitFormattedLine("", "DISCARD", "#0D1");

            break;

    }

}

}
```

```
// ENDCODEGENERATION

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");

GetNextToken(tokens);

ExitModule("CallStatement");
}

//-----
void ParseReturnStatement(TOKEN tokens[])
//-----
{
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];

    EnterModule("ReturnStatement");

    sprintf(line, "; **** Return statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);

    GetNextToken(tokens);
```

```
// STATICSEMANTICS

if (code.IsInModuleBody(PROCEDURE_SUBPROGRAMMODULE))

// CODEGENERATION

    code.EmitFormattedLine("", "RETURN");

// ENDCODEGENERATION

else if (code.IsInModuleBody(FUNCTION_SUBPROGRAMMODULE))

{

    DATATYPE datatype;

    if (tokens[0].type != L_PAREN)

        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");

    GetNextToken(tokens);

    ParseExpression(tokens, datatype);

    if (datatype != identifierTable.GetDatatype(code.GetModuleIdentifierIndex()))

        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                            "Return expression data type must match FUNCTION ('func') data type");

// CODEGENERATION

    code.EmitFormattedLine("", "POP", "FB:0D0", "pop RETURN expression into function return value");
    code.EmitFormattedLine("", "RETURN");

// ENDCODEGENERATION

if (tokens[0].type != R_PAREN)

    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'"");
```

```
    GetNextToken(tokens);

}

else

    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Return statement only allowed in PROCEDURE ('proc') or FUNCTION ('func') module body");

// ENDSTATICSEMANTICS

if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ';'");

GetNextToken(tokens);

ExitModule("ReturnStatement");

}

// TODO: Replace the CODEGENERATION segments with
// service requests for file I/O
//-----
void ParseFilePrintStatement(TOKEN tokens[])
//-----
{

    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
```

```
DATATYPE datatype;

bool isInTable;
int index;

EnterModule("FilePrintStatement");

// Get the next token
GetNextToken(tokens);

// CODEGENERATION
sprintf(line, "; **** 'f_print' statement (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
// ENDCODEGENERATION

// Check for a left parenthesis: '('
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

// Get the next token
GetNextToken(tokens);

// Check if the next token is an identifier
if ((tokens[0].type != IDENTIFIER))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
```

```
"Expecting identifier");

// Check if the identifier is in the identifier table
index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Undefined identifier");

// Check if the identifier is an file variable
if (identifierTable.GetDatatype(index) != FILETYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting file variable");

// Get the next token
GetNextToken(tokens);

// Check for a comma
if (tokens[0].type != COMMA)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ','");

do
{
    GetNextToken(tokens);

    // <string>
```

```
if (tokens[0].type == STRING) {  
    // CODEGENERATION  
    /*  
     *  
     char reference[SOURCELINELENGTH + 1];  
  
     code.AddDSToStaticData(tokens[0].lexeme, "", reference);  
     code.EmitFormattedLine("", "PUSHA", reference);  
     code.EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");  
    */  
    // ENDCODEGENERATION  
  
    GetNextToken(tokens);  
}  
// <expression>  
else {  
    ParseExpression(tokens, datatype);  
  
    // CODEGENERATION  
    /*  
     *  
     switch (datatype)  
     {  
         case INTEGERTYPE:  
         case ENUMTYPE:  
             code.EmitFormattedLine("", "SVC", "#SVC_WRITE_INTEGER");  
             break;  
         case BOOLEANTYPE:  
    }  
    */  
}
```

```
        code.EmitFormattedLine("", "SVC", "#SVC_WRITE_BOOLEAN");

        break;

    case CHARACTERTYPE:

        code.EmitFormattedLine("", "SVC", "#SVC_WRITE_CHARACTER");

        break;

    case FLOATTYPE:

        code.EmitFormattedLine("", "SVC", "#SVC_WRITE_FLOAT");

        break;

    case POINTERTYPE:

        code.EmitFormattedLine("", "SVC", "#SVC_WRITE_HEXADECIMAL");

        break;

    }

}

// ENDCODEGENERATION

}

} while (tokens[0].type == COMMA);

//Check for a right parenthesis: ')'

if (tokens[0].type != R_PAREN)

    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,

        "Expecting ')\");

// Get the next token

GetNextToken(tokens);
```

```
// Check for a semicolon
if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ';'");

// Get the next token
GetNextToken(tokens);

ExitModule("FilePrintStatement");
}

//-----
void ParseFileInputStatement(TOKEN tokens[])
//-----
{
    void ParseVariable(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
    DATATYPE datatype;

    bool isInTable;
    int index;

    EnterModule("FileInputStatement");
}
```

```
// Get the next token
GetNextToken(tokens);

// CODEGENERATION
sprintf(line, "; **** 'f_input' statement (%4d)", tokens[0].sourceLineNumber);
code.EmitUnformattedLine(line);
// ENDCODEGENERATION

// Check for a left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

// Get the next token
GetNextToken(tokens);

// Check if the next token is an identifier
if ((tokens[0].type != IDENTIFIER))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting identifier");

// Check if the identifier is in the identifier table
index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Undefined identifier");
```

```
// Check if the identifier is an file variable
if (identifierTable.GetDatatype(index) != FILETYPE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting file variable");

// Get the next token
GetNextToken(tokens);

// Check for a comma
if (tokens[0].type != COMMA)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ','");

// Get the next token
GetNextToken(tokens);

// Parse the variable
ParseVariable(tokens, true, datatype);

// CODEGENERATION
/*
switch (datatype)
{
    case INTEGERTYPE:
        code.EmitFormattedLine("", "SVC", "#SVC_READ_INTEGER");
}
```

```
        break;

case BOOLEANTYPE:
    code.EmitFormattedLine("", "SVC", "#SVC_READ_BOOLEAN");
    break;

case CHARACTERTYPE:
    code.EmitFormattedLine("", "SVC", "#SVC_READ_CHARACTER");
    break;

case FLOATTYPE:
    code.EmitFormattedLine("", "SVC", "#SVC_READ_FLOAT");
    break;
}

code.EmitFormattedLine("", "POP", "@SP:0D1");
code.EmitFormattedLine("", "DISCARD", "#0D1");
*/
// ENDCODEGENERATION

//Check for a right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ')');

// Get the next token
GetNextToken(tokens);

//Check for a semicolon
if (tokens[0].type != SEMICOLON)
```

```
ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting ';'");

// Get the next token
GetNextToken(tokens);

ExitModule("FileInputStatement");
}

//-----
void ParseFileOperationStatement(TOKEN tokens[])
//-----
{
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    char line[SOURCELINELENGTH + 1];
    DATATYPE datatype;

    bool isInTable;
    int index;

    EnterModule("FileOperationStatement");

    // Get the next token
    //GetNextToken(tokens);
```

```
// Check for assignment operator: '='
if (tokens[0].type != ASSIGN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '='");

// Get the next token
GetNextToken(tokens);

// Check the file operation
// TODO: Add code generation
switch (tokens[0].type)
{
case F_CREATE:
    sprintf(line, "; **** f_create statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);

// Get the next token
GetNextToken(tokens);

// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

// Get the next token
```

```
GetNextToken(tokens);

// Check for <string>
if (tokens[0].type != STRING)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting a string argument");

// CODEGENERATION
char reference[SOURCELINELENGTH + 1];

code.AddDSToStaticData(tokens[0].lexeme, "", reference);

// Point the file variable to the new file
code.EmitFormattedLine("", "PUSHA", reference);
code.EmitFormattedLine("", "POP", "@SP:0D1");
code.EmitFormattedLine("", "DISCARD", "#0D1");

// Create the new file
code.EmitFormattedLine("", "PUSHA", reference);
code.EmitFormattedLine("", "SVC", "#SVC_CREATE_FILE");

// ENDCODEGENERATION

// Get the next token
GetNextToken(tokens);
```

```
// Check for right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                         "Expecting ')''");

break;

case F_READ:
    sprintf(line, "; **** f_read statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);

// Get the next token
GetNextToken(tokens);

// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                         "Expecting '('");

// Get the next token
GetNextToken(tokens);

switch (tokens[0].type)
{
case TRUE:
    // CODEGENERATION
    // ...
}
```

```
// ENDCODEGENERATION
break;

case FALSE:
    // CODEGENERATION
    // ...
    // ENDCODEGENERATION
    break;

default:
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting 'true' or 'false');

}

// Get the next token
GetNextToken(tokens);

// Check for right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting ')''");

break;

case F_WRITE:
    sprintf(line, "; **** f_write statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    // Get the next token
    GetNextToken(tokens);
```

```
// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '(');

// Get the next token
GetNextToken(tokens);

switch (tokens[0].type)
{
    case TRUE:
        // CODEGENERATION
        // ...
        // ENDCODEGENERATION
        break;
    case FALSE:
        // CODEGENERATION
        // ...
        // ENDCODEGENERATION
        break;
    default:
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Expecting 'true' or 'false'");
}
```

```
// Get the next token
GetNextToken(tokens);

// Check for right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ')''");

break;

case F_CLEAR:
    sprintf(line, "; **** f_clear statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    // Get the next token
    GetNextToken(tokens);

// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

// Get the next token
GetNextToken(tokens);

// Check for right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
```

```
    "Expecting ')''");

break;

case F_OPEN:
    sprintf(line, "; **** f_open statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);

    // Get the next token
    GetNextToken(tokens);

    // Check for left parenthesis
    if (tokens[0].type != L_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Expecting '('");

    // Get the next token
    GetNextToken(tokens);

    // Check for <string>
    if (tokens[0].type != STRING)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Expecting a string argument");

    // CODEGENERATION
    // char reference[SOURCELINELENGTH + 1];
    // ...
```

```
// ENDCODEGENERATION

// Get the next token
GetNextToken(tokens);

// Check for right parenthesis
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ')''");

break;

case F_CLOSE:
    sprintf(line, "; **** f_close statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    // Get the next token
    GetNextToken(tokens);

// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

// Get the next token
GetNextToken(tokens);

// Check for right parenthesis
```

```
if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting ')''");

break;

case F_DELETE:
    sprintf(line, "; **** f_delete statement (%4d)", tokens[0].sourceLineNumber);
    code.EmitUnformattedLine(line);
    // Get the next token
    GetNextToken(tokens);

    // Check for left parenthesis
    if (tokens[0].type != L_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting '('");

    // Get the next token
    GetNextToken(tokens);

    // Check for right parenthesis
    if (tokens[0].type != R_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ')''");

break;

default:
```

```
ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
    "Expecting a file operation");
}

// Get the next token
GetNextToken(tokens);

// Check for semicolon
if (tokens[0].type != SEMICOLON)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting ';'");

ExitModule("FileOperationStatement");
}

//-----
void ParseFCreate(TOKEN tokens[])
//-----
{
    /*** Add at a later date ***/
}
```

```
//-----
void ParseExpression(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    // CODEGENERATION
    /*
        An expression is composed of a collection of one or more operands (Rogue calls them
        primaries) and operators (and perhaps sets of parentheses to modify the default
        order-of-evaluation established by precedence and associativity rules).
        Expression evaluation computes a single value as the expression's result.
        The result has a specific data type. By design, the expression result is
        "left" at the top of the run-time stack for subsequent use.

        Rogue expressions must be single-mode with operators working on operands of
        the appropriate type (for example, boolean AND boolean) and not mixing
        modes. Static semantic analysis guarantees that operators are
        operating on operands of appropriate data type.
    */
    // ENDCODEGENERATION

    void ParseConjunction(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    DATATYPE datatypeLHS, datatypeRHS;
```

```
EnterModule("Expression");

ParseConjunction(tokens, datatypeLHS);

if ((tokens[0].type == OR) ||
    (tokens[0].type == NOR) ||
    (tokens[0].type == XOR))
{
    while ((tokens[0].type == OR) ||
           (tokens[0].type == NOR) ||
           (tokens[0].type == XOR))
    {
        TOKENTYPE operation = tokens[0].type;

        GetNextToken(tokens);
        ParseConjunction(tokens, datatypeRHS);

        // CODEGENERATION
        switch (operation)
        {
            case OR:

                // STATICSEMANTICS
                if (!((datatypeLHS == BOOLEANTYPE) && (datatypeRHS == BOOLEANTYPE)))
                    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operands");
        }
    }
}
```

```
// ENDSTATICSEMANTICS

code.EmitFormattedLine("", "OR");

datatype = BOOLEANTYPE;
break;

case NOR:

// STATICSEMANTICS

if (!((datatypeLHS == BOOLEANTYPE) && (datatypeRHS == BOOLEANTYPE)))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operands");
// ENDSTATICSEMANTICS

code.EmitFormattedLine("", "NOR");

datatype = BOOLEANTYPE;
break;

case XOR:

// STATICSEMANTICS

if (!((datatypeLHS == BOOLEANTYPE) && (datatypeRHS == BOOLEANTYPE)))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operands");
// ENDSTATICSEMANTICS

code.EmitFormattedLine("", "XOR");

datatype = BOOLEANTYPE;
break;

}
```

```
    }

    // CODEGENERATION

}

else

datatype = datatypeLHS;

ExitModule("Expression");

}

//-----

void ParseConjunction(TOKEN tokens[], DATATYPE& datatype)
//-----

{

void ParseNegation(TOKEN tokens[], DATATYPE & datatype);

void GetNextToken(TOKEN tokens[]);

DATATYPE datatypeLHS, datatypeRHS;

EnterModule("Conjunction");

ParseNegation(tokens, datatypeLHS);

if ((tokens[0].type == AND) ||
(tokens[0].type == NAND))

{
```

```
while ((tokens[0].type == AND) ||
       (tokens[0].type == NAND))
{
    TOKENTYPE operation = tokens[0].type;

    GetNextToken(tokens);
    ParseNegation(tokens, datatypeRHS);

    switch (operation)
    {
        case AND:
            if (!((datatypeLHS == BOOLEANTYPE) && (datatypeRHS == BOOLEANTYPE)))
                ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operands");
            code.EmitFormattedLine("", "AND");
            datatype = BOOLEANTYPE;
            break;
        case NAND:
            if (!((datatypeLHS == BOOLEANTYPE) && (datatypeRHS == BOOLEANTYPE)))
                ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operands");
            code.EmitFormattedLine("", "NAND");
            datatype = BOOLEANTYPE;
            break;
    }
}
else
```

```
datatype = datatypeLHS;

ExitModule("Conjunction");
}

//-----
void ParseNegation(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParseComparison(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    DATATYPE datatypeRHS;

    EnterModule("Negation");

    if (tokens[0].type == NOT)
    {
        GetNextToken(tokens);
        ParseComparison(tokens, datatypeRHS);

        if (!(datatypeRHS == BOOLEANTYPE))
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting boolean operand");
        code.EmitFormattedLine("", "NOT");
        datatype = BOOLEANTYPE;
    }
}
```

```
else
    ParseComparison(tokens, datatype);

    ExitModule("Negation");
}

//-----
void ParseComparison(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParseComparator(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    DATATYPE datatypeLHS, datatypeRHS;

    EnterModule("Comparison");

    ParseComparator(tokens, datatypeLHS);
    if ((tokens[0].type == LT) ||
        (tokens[0].type == LTEQ) ||
        (tokens[0].type == EQ) ||
        (tokens[0].type == GT) ||
        (tokens[0].type == GTEQ) ||
        (tokens[0].type == NOTEQ))
    )
{
```

```
TOKENTYPE operation = tokens[0].type;

GetNextToken(tokens);

ParseComparator(tokens, datatypeRHS);

/*
    CMPI           ; or CMPF (as required)
    JMPXX   T????? ; XX = L,E,G,LE,NE,GE (as required)
    PUSH    #0X0000 ; push FALSE
    JMP     E????? ;      or
    T????? PUSH    #0xFFFF  ; push TRUE (as required)
    E????? EQU    *
*/
if ( ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == INTEGERTYPE)) ||
    ((datatypeLHS == ENUMTYPE    ) && (datatypeRHS == INTEGERTYPE)) ||
    ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == ENUMTYPE    )) || 
    ((datatypeLHS == ENUMTYPE    ) && (datatypeRHS == ENUMTYPE    )) )
{
    code.EmitFormattedLine("", "CMPI");
}
else if ((datatypeLHS == FLOATTYPE) && (datatypeRHS == FLOATTYPE))
{
    code.EmitFormattedLine("", "CMPF");
}
else if ( ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == FLOATTYPE  )) ||
```

```
((datatypeLHS == FLOATTYPE) && (datatypeRHS == INTEGERTYPE)) ||
((datatypeLHS == ENUMTYPE) && (datatypeRHS == FLOATTYPE)) ||
((datatypeLHS == FLOATTYPE) && (datatypeRHS == ENUMTYPE)) )

{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Operands must be both integers, both enum types, one integer and one enum type, or both float");
}

else
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer, enum type, or float operands");
}

char Tlabel[SOURCELINELENGTH + 1], Elabel[SOURCELINELENGTH + 1];

sprintf(Tlabel, "T%04d", code.LabelSuffix());
sprintf(Elabel, "E%04d", code.LabelSuffix());
switch (operation)
{
case LT:
    code.EmitFormattedLine("", "JMPL", Tlabel);
    break;
case LTEQ:
    code.EmitFormattedLine("", "JMPLE", Tlabel);
    break;
case EQ:
```

```
        code.EmitFormattedLine("", "JMPNE", Tlabel);
        break;

    case GT:
        code.EmitFormattedLine("", "JMPG", Tlabel);
        break;

    case GTEQ:
        code.EmitFormattedLine("", "JMPGE", Tlabel);
        break;

    case NOTEQ:
        code.EmitFormattedLine("", "JMPNE", Tlabel);
        break;
    }

datatype = BOOLEANTYPE;
code.EmitFormattedLine("", "PUSH", "#0X0000");
code.EmitFormattedLine("", "JMP", Elabel);
code.EmitFormattedLine(Tlabel, "PUSH", "#0xFFFF");
code.EmitFormattedLine(Elabel, "EQU", "*");

}

else
datatype = datatypeLHS;

ExitModule("Comparison");
}

//-----
void ParseComparator(TOKEN tokens[], DATATYPE& datatype)
```

```
//-----
{

void ParseTerm(TOKEN tokens[], DATATYPE & datatype);

void GetNextToken(TOKEN tokens[]);

DATATYPE datatypeLHS, datatypeRHS;

EnterModule("Comparator");

ParseTerm(tokens, datatypeLHS);

if ((tokens[0].type == PLUS) ||
    (tokens[0].type == MINUS))
{
    while ((tokens[0].type == PLUS) ||
           (tokens[0].type == MINUS))
    {
        TOKENTYPE operation = tokens[0].type;

        GetNextToken(tokens);
        ParseTerm(tokens, datatypeRHS);

        if ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == INTEGERTYPE))
        {
            switch (operation)
            {
```

```
case PLUS:
    code.EmitFormattedLine("", "ADDI");
    break;
case MINUS:
    code.EmitFormattedLine("", "SUBI");
    break;
}
datatype = INTEGERTYPE;
}
else if ((datatypeLHS == FLOATTYPE) && (datatypeRHS == FLOATTYPE))
{
    switch (operation)
    {
        case PLUS:
            code.EmitFormattedLine("", "ADDF");
            break;
        case MINUS:
            code.EmitFormattedLine("", "SUBF");
            break;
    }
    datatype = FLOATTYPE;
}
else if ( (datatypeLHS == INTEGERTYPE) && (datatypeRHS == FLOATTYPE)
        || ((datatypeLHS == FLOATTYPE) && (datatypeRHS == INTEGERTYPE)) )
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Operands must be of the same datatype");
```

```
        }

    else
    {
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer or float operands");
    }
}

else
datatype = datatypeLHS;

ExitModule("Comparator");
}

//-----
void ParseTerm(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParseFactor(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    DATATYPE datatypeLHS, datatypeRHS;

    EnterModule("Term");

    ParseFactor(tokens, datatypeLHS);
    if ((tokens[0].type == MULTIPLY) ||
        (tokens[0].type == DIVIDE)) {
        GetNextToken(tokens);
        ParseFactor(tokens, datatypeRHS);
        datatype = datatypeLHS * datatypeRHS;
    }
    else if (tokens[0].type == SUBTRACT) {
        GetNextToken(tokens);
        ParseFactor(tokens, datatypeRHS);
        datatype = datatypeLHS - datatypeRHS;
    }
    else if (tokens[0].type == ADD) {
        GetNextToken(tokens);
        ParseFactor(tokens, datatypeRHS);
        datatype = datatypeLHS + datatypeRHS;
    }
}
```

```
(tokens[0].type == DIVIDE) ||
(tokens[0].type == MODULUS))

{
while ((tokens[0].type == MULTIPLY) ||
(tokens[0].type == DIVIDE) ||
(tokens[0].type == MODULUS))

{
    TOKENTYPE operation = tokens[0].type;

    GetNextToken(tokens);
    ParseFactor(tokens, datatypeRHS);

    if ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == INTEGERTYPE))
    {
        switch (operation)
        {
            case MULTIPLY:
                code.EmitFormattedLine("", "MULI");
                break;
            case DIVIDE:
                code.EmitFormattedLine("", "DIVI");
                break;
            case MODULUS:
                code.EmitFormattedLine("", "REMI");
                break;
        }
    }
}
```

```
datatype = INTEGERTYPE;

}

else if ((datatypeLHS == FLOATTYPE) && (datatypeRHS == FLOATTYPE))

{

    switch (operation)

    {

        case MULTIPLY:

            code.EmitFormattedLine("", "MULF");

            break;

        case DIVIDE:

            code.EmitFormattedLine("", "DIVF");

            break;

        case MODULUS: /* the % operator only accepts integer operands */

            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer operands");

            break;

    }

    datatype = FLOATTYPE;

}

else if ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == FLOATTYPE) ||
          (datatypeLHS == FLOATTYPE) && (datatypeRHS == INTEGERTYPE))

{

    switch (operation)

    {

        case MODULUS:

            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer operands");


```

```
        break;

    default:
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Operands must be of the same datatype");
        break;
    }
}
else
{
    switch (operation)
    {
        case MODULUS:
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer operands");
            break;
        default:
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer or float operands");
            break;
    }

}
}

datatype = datatypeLHS;

ExitModule("Term");
}
```

```
//-----
void ParseFactor(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParseSecondary(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    EnterModule("Factor");

    if ((tokens[0].type == ABS) ||
        (tokens[0].type == PLUS) ||
        (tokens[0].type == MINUS)
    )
    {
        DATATYPE datatypeRHS;
        TOKENTYPE operation = tokens[0].type;

        GetNextToken(tokens);
        ParseSecondary(tokens, datatypeRHS);

        if (datatypeRHS != INTEGERTYPE && datatypeRHS != FLOATTYPE)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer or float operand");

        switch (operation)
        {
```

```
case ABS:
/*
    SETNZPI          ; or SETNZPF (as required)
    JMPNN   E?????
    NEGI          ; NEGI or NEGFI (as required)
E????? EQU      *
*/
{
    char Elabel[SOURCELINELENGTH + 1];

    sprintf(Elabel, "E%04d", code.LabelSuffix());

    switch (datatypeRHS)
    {
        case INTEGERTYPE:
            code.EmitFormattedLine("", "SETNZPI");
            code.EmitFormattedLine("", "JMPNN", Elabel);
            code.EmitFormattedLine("", "NEGI");
            break;
        case FLOATTYPE:
            code.EmitFormattedLine("", "SETNZPF");
            code.EmitFormattedLine("", "JMPNN", Elabel);
            code.EmitFormattedLine("", "NEGFI");
            break;
    }
}
```

```
    code.EmitFormattedLine(Elabel, "EQU", "*");

}

break;

case PLUS:

// Do nothing (identity operator)

break;

case MINUS:

switch (datatypeRHS)

{

case INTEGERTYPE:

    code.EmitFormattedLine("", "NEGI");

    break;

case FLOATTYPE:

    code.EmitFormattedLine("", "NEGF");

    break;

}

break;

}

datatype = datatypeRHS;

}

else

ParseSecondary(tokens, datatype);

ExitModule("Factor");
```

```
}

//-----
void ParseSecondary(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParsePrefix(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    DATATYPE datatypeLHS, datatypeRHS;

    EnterModule("Secondary");

    ParsePrefix(tokens, datatypeLHS);

    if (tokens[0].type == POWER)
    {
        GetNextToken(tokens);

        ParsePrefix(tokens, datatypeRHS);

        if ((datatypeLHS == INTEGERTYPE) && (datatypeRHS == INTEGERTYPE))
        {
            code.EmitFormattedLine("", "POWI");
            datatype = INTEGERTYPE;
        }
    }
}
```

```
else if ((datatypeLHS == FLOATTYPE) && (datatypeRHS == FLOATTYPE))
{
    code.EmitFormattedLine("", "POWF");

    datatype = FLOATTYPE;
}

else
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting 2 integer operands or 2 float operands");
}

}

else
    datatype = datatypeLHS;

ExitModule("Secondary");
}

//-----
void ParsePrefix(TOKEN tokens[], DATATYPE& datatype)
//-----
{
    void ParseVariable(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void ParsePrimary(TOKEN tokens[], DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);

    EnterModule("Prefix");
```

```
if ((tokens[0].type == INC) ||
    (tokens[0].type == DEC)
)
{
    DATATYPE datatypeRHS;
    TOKENTYPE operation = tokens[0].type;

    GetNextToken(tokens);
    ParseVariable(tokens, true, datatypeRHS);

    if (datatypeRHS != INTEGERTYPE)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting integer operand");

    switch (operation)
    {
        case INC:
            code.EmitFormattedLine("", "PUSH", "@SP:0D0");
            code.EmitFormattedLine("", "PUSH", "#0D1");
            code.EmitFormattedLine("", "ADDI");
            code.EmitFormattedLine("", "POP", "@SP:0D1");           // side-effect
            code.EmitFormattedLine("", "PUSH", "@SP:0D0");
            code.EmitFormattedLine("", "SWAP");
            code.EmitFormattedLine("", "DISCARD", "#0D1");         // value
            break;
        case DEC:
            code.EmitFormattedLine("", "PUSH", "@SP:0D0");
            code.EmitFormattedLine("", "PUSH", "#0D1");
            code.EmitFormattedLine("", "SUBI");
            code.EmitFormattedLine("", "POP", "@SP:0D1");
            code.EmitFormattedLine("", "PUSH", "@SP:0D0");
            code.EmitFormattedLine("", "SWAP");
            code.EmitFormattedLine("", "DISCARD", "#0D1");
            break;
    }
}
```

```
        code.EmitFormattedLine("", "PUSH", "@SP:0D0");
        code.EmitFormattedLine("", "PUSH", "#0D1");
        code.EmitFormattedLine("", "SUBI");
        code.EmitFormattedLine("", "POP", "@SP:0D1");           // side-effect
        code.EmitFormattedLine("", "PUSH", "@SP:0D0");
        code.EmitFormattedLine("", "SWAP");
        code.EmitFormattedLine("", "DISCARD", "#0D1");         // value
        break;
    }
    datatype = INTEGERTYPE;
}
else
ParsePrimary(tokens, datatype);

ExitModule("Prefix");
}

//-----
void ParsePrimary(TOKEN tokens[], DATATYPE& datatype) /*** updated ***/
//-----
{
    void ParseVariable(TOKEN tokens[], bool asLValue, DATATYPE & datatype);
    void ParseExpression(TOKEN tokens[], DATATYPE & datatype);
    void ParseAssociativeArrayReference(TOKEN tokens[], bool asLvalue, DATATYPE & datatype);
    void GetNextToken(TOKEN tokens[]);
}
```

```
EnterModule("Primary");

char operand[SOURCELINELENGTH + 1];

bool isInTable;
int index;

switch (tokens[0].type)
{
case INTEGER:
    sprintf(operand, "#0D%s", tokens[0].lexeme);
    code.EmitFormattedLine("", "PUSH", operand);
    datatype = INTEGERTYPE;
    GetNextToken(tokens);
    break;
case TRUE:
    code.EmitFormattedLine("", "PUSH", "#0xFFFF");
    datatype = BOOLEANTYPE;
    GetNextToken(tokens);
    break;
case FALSE:
    code.EmitFormattedLine("", "PUSH", "#0X0000");
    datatype = BOOLEANTYPE;
    GetNextToken(tokens);
    break;
case CHARACTER:
```

```
sprintf(operand, "#%s", tokens[0].lexeme);
code.EmitFormattedLine("", "PUSH", operand);
datatype = CHARCTERTYPE;
GetNextToken(tokens);
break;

case FLOATVAL:
    sprintf(operand, "#0F%s", tokens[0].lexeme);
    code.EmitFormattedLine("", "PUSH", operand);
    datatype = FLOATTTYPE;
    GetNextToken(tokens);
    break;

case L_PAREN:
    GetNextToken(tokens);
    ParseExpression(tokens, datatype);
    if (tokens[0].type != R_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting )");
    GetNextToken(tokens);
    break;
// addr(<variable>)

case ADDR:
    GetNextToken(tokens);

    // Check for left parenthesis
    if (tokens[0].type != L_PAREN)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting (");
```

```
GetNextToken(tokens);

if (tokens[0].type != IDENTIFIER)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
}

index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Undefined identifier");

//=====
// MUST be a variable reference
//=====

if (identifierTable.GetType(index) == FUNCTION_SUBPROGRAMMODULE || identifierTable.GetDatatype(index) == ASSOCTYPE)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Identifier must refer to a variable");
}

// Get the datatype of the referenced variable, and log it as the
//      datatype of the CONTENTS OF the pointer(s)
DATATYPE contentDatatype;

contentDatatype = identifierTable.GetDatatype(index);

for (int ptr : ptr_indexes)
```

```
{  
    identifierTable.SetContentDatatype(ptr, contentDatatype);  
}  
  
GetNextToken(tokens);  
  
// Check for right parenthesis  
if (tokens[0].type != R_PAREN)  
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting )");  
GetNextToken(tokens);  
  
// set the datatype to POINTERTYPE  
datatype = POINTERTYPE;  
  
// CODEGENERATION  
// push the ADDRESS OF the variable  
code.EmitFormattedLine("", "PUSHA", identifierTable.GetReference(index));  
// END CODEGENERATION  
  
break;  
  
// cont(<ptr variable>)  
case CONT:  
  
GetNextToken(tokens);
```

```
// Check for left parenthesis
if (tokens[0].type != L_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting (");
GetNextToken(tokens);

if (tokens[0].type != IDENTIFIER)
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting identifier");
}

index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!isInTable)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Undefined identifier");

//=====
// MUST be a pointer reference
//=====

if (identifierTable.GetDatatype(index) != POINTERTYPE )
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Identifier must refer to a pointer variable.");
}

GetNextToken(tokens);

// Check for right parenthesis
if (tokens[0].type != R_PAREN)
```

```
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting )");

GetNextToken(tokens);

// Get the datatype of the contents of the pointer
datatype = identifierTable.GetContentDatatype(index);

// CODEGENERATION
sprintf(operand, "@%s", identifierTable.GetReference(index));
code.EmitFormattedLine("", "PUSH", operand);
// ENDCODEGENERATION

break;

// <AssociativeArrayReference> | <variable> | <enumConstant> | <FunctionReference>
case IDENTIFIER:
{
    bool isInTable;
    int index;

    // Variables used to check if the identifier is an enumConstant
    bool isEnumConstant = false;
    vector<string> v;
    vector<string>::iterator it;

    // see if the identifier can be found in the enumMap
```

```
if (lValueEnumType != "")  
{  
    v = enumMap[lValueEnumScope][lValueEnumType];  
    it = find(v.begin(), v.end(), tokens[0].lexeme);  
  
    // if found, set isEnumConstant to true  
    if (it != v.end())  
    {  
        isEnumConstant = true;  
    }  
  
    // Check if the enum type of the l-value can be found in the global scope  
    else if (lValueEnumScope == PROGRAMMODULESCOPE || lValueEnumScope == SUBPROGRAMMODULESCOPE)  
    {  
        if (enumMap[GLOBALSCOPE].find(lValueEnumType) != enumMap[GLOBALSCOPE].end())  
        {  
            v = enumMap[GLOBALSCOPE][lValueEnumType];  
            it = find(v.begin(), v.end(), tokens[0].lexeme);  
  
            // if found, set isEnumConstant to true  
            if (it != v.end())  
            {  
                isEnumConstant = true;  
            }  
        }  
    }  
}
```

```
}

// If identifier is undefined if it's not in the identifier table AND it's not an enum constant
// (enum constants are not stored in the identifier table)
index = identifierTable.GetIndex(tokens[0].lexeme, isInTable);
if (!(isInTable || isEnumConstant))
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Undefined identifier");

if (identifierTable.GetType(index) != FUNCTION_SUBPROGRAMMODULE)
{

//=====
// Associative array reference
//=====

if (identifierTable.GetDatatype(index) == ASSOCTYPE)
{

    ParseAssociativeArrayReference(tokens, false, datatype);

    // CODEGENERATION
    /** Get the value for the given key in the associative array ***/
    code.EmitFormattedLine("", "GETAAE", identifierTable.GetReference(index));
    // ENDCODEGENERATION
}

//=====
```

```
// Enumeration constant
//=====
else if (isEnumConstant)
{
    // Convert the enumConstant to an integer by getting its index in its enumMap vector
    int enumToInt = it - v.begin();

    sprintf(operand, "#0D%d", enumToInt);
    code.EmitFormattedLine("", "PUSH", operand);

    datatype = ENUMTYPE;
    GetNextToken(tokens);

}

//=====
// variable reference
//=====

else
{
    ParseVariable(tokens, false, datatype);

    // When assigning a pointer to another pointer, set the contents
    //   datatype of the l-value pointer to that of the r-value pointer.
    if (datatype == POINTERTYPE)
```

```
{  
    contentDatatype = identifierTable.GetContentDatatype(index);  
  
    for (int ptr : ptr_indexes)  
    {  
        identifierTable.SetContentDatatype(ptr, contentDatatype);  
    }  
}  
}  
  
//=====  
// FUNCTION_SUBPROGRAMMODULE reference  
//=====  
else  
{  
    char operand[MAXIMUMLENGTHIDENTIFIER + 1];  
    int parameters;  
  
    GetNextToken(tokens);  
    if (tokens[0].type != L_PAREN)  
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '('");  
  
    // CODEGENERATION  
    code.EmitFormattedLine("", "PUSH", "#0X0000", "reserve space for function return value");  
    // ENDCODEGENERATION
```

```
datatype = identifierTable.GetDatatype(index);

parameters = 0;

if (tokens[1].type == R_PAREN)
{
    GetNextToken(tokens);
}

else
{
    do
    {
        DATATYPE expressionDatatype;

        GetNextToken(tokens);
        ParseExpression(tokens, expressionDatatype);
        parameters++;

        // STATICSEMANTICS
        if (expressionDatatype != identifierTable.GetDatatype(index + parameters))
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
                "Actual parameter data type does not match formal parameter data type");
        // ENDSTATICSEMANTICS

    } while (tokens[0].type == COMMA);
}
```

```
// STATICSEMANTICS

if (identifierTable.GetCountOfFormalParameters(index) != parameters)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Number of actual parameters does not match number of formal parameters");

// ENDSTATICSEMANTICS

if (tokens[0].type != R_PAREN)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ')'");
GetNextToken(tokens);

// CODEGENERATION

code.EmitFormattedLine("", "PUSHFB");
code.EmitFormattedLine("", "CALL", identifierTable.GetReference(index));
code.EmitFormattedLine("", "POPFB");
sprintf(operand, "#0D%d", parameters);
code.EmitFormattedLine("", "DISCARD", operand);

// ENDCODEGENERATION

}

}

break;

default:

    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Expecting literal, '(', variable, associative array reference, or FUNCTION ('func') identifier");
    break;
}
```

```
    ExitModule("Primary");
}

//-----
void ParseVariable(TOKEN tokens[],bool asLValue,DATATYPE &datatype) /*** updated ***/
//-----
{
/*
Syntax "locations"          l- or r-value
-----
<expression>                r-value
<prefix>                     l-value
<InputStatement>             l-value
LHS of <assignmentStatement> l-value
<ForStatement>               l-value
OUT <formalParameter>        l-value
IO <formalParameter>         l-value
REF <formalParameter>        l-value

r-value ( read-only): value is pushed on run-time stack
l-value (read/write): address of value is pushed on run-time stack
*/
void GetNextToken(TOKEN tokens[]);

bool isInTable;
```

```
int index;
int dimensions;
IDENTIFIERTYPE identifierType;

EnterModule("Variable");

if ( tokens[0].type != IDENTIFIER )
    ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,"Expecting identifier");

// STATICSEMANTICS
index = identifierTable.GetIndex(tokens[0].lexeme,isInTable);
if ( !isInTable )
    ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,"Undefined identifier");

identifierType = identifierTable.GetType(index);
datatype = identifierTable.GetDatatype(index);

// Add the indexes of pointer l-values to the ptr_indexes vector
if (datatype == POINTERTYPE)
{
    if (asLValue)
    {
        // Add to pointer vector
        ptr_indexes.push_back(index);
    }
}
```

```
}

if (!((identifierType == GLOBAL_VARIABLE) ||
(identifierType == GLOBAL_CONSTANT) ||
(identifierType == PROGRAMMODULE_VARIABLE) ||
(identifierType == PROGRAMMODULE_CONSTANT) ||
(identifierType == SUBPROGRAMMODULE_VARIABLE) ||
(identifierType == SUBPROGRAMMODULE_CONSTANT) ||
(identifierType == IN_PARAMETER) ||
(identifierType == OUT_PARAMETER) ||
(identifierType == IO_PARAMETER) ||
(identifierType == REF_PARAMETER)))

ProcessCompilerError(tokens[0].sourceLineNumber,tokens[0].sourceLineIndex,"Expecting variable or constant identifier");

if (asLValue && ((identifierType == GLOBAL_CONSTANT) ||
(identifierType == PROGRAMMODULE_CONSTANT) ||
(identifierType == SUBPROGRAMMODULE_CONSTANT)))
ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Constant may not be l-value");

if (asLValue && (identifierType == GLOBAL_VARIABLE) && code.IsInModuleBody(FUNCTION_SUBPROGRAMMODULE))
ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "FUNCTION ('func') may not modify global variable");

// ENDSTATICSEMANTICS

// CODEGENERATION
if (identifierTable.GetDimensions(index) == 0)
{
```

```
if (asLValue)
    code.EmitFormattedLine("", "PUSHA", identifierTable.GetReference(index));
else
    code.EmitFormattedLine("", "PUSH", identifierTable.GetReference(index));
}
else
{
    GetNextToken(tokens);
    if (tokens[0].type != L_BRACKET)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '['");
    dimensions = 0;
    do
    {
        DATATYPE expressionDatatype;

        GetNextToken(tokens);
        ParseExpression(tokens, expressionDatatype);
        dimensions++;
        if (expressionDatatype != INTEGERTYPE)
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Index expression must be integer");
    } while (tokens[0].type == COMMA);

    if (identifierTable.GetDimensions(index) != dimensions)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
            "Number of index expressions does not match array dimensions");
```

```
if (tokens[0].type != R_BRACKET)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting ']'");

if (asLValue)
    code.EmitFormattedLine("", "ADRAE", identifierTable.GetReference(index));
else
    code.EmitFormattedLine("", "GETAE", identifierTable.GetReference(index));

}

// ENDCODEGENERATION

GetNextToken(tokens);

ExitModule("Variable");
}

//-----
void ParseAssociativeArrayReference(TOKEN tokens[], bool asLvalue, DATATYPE &datatype)    /*** new ***/
//-----
{
    void GetNextToken(TOKEN tokens[]);
    DATATYPE keyDatatype;

    EnterModule("AssociativeArrayReference");
}
```

```
// Get the next token
GetNextToken(tokens);

// Check for a left brace: '{'
if (tokens[0].type != L_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '{'");
GetNextToken(tokens);

// Parse the expression between the braces
ParseExpression(tokens, keyDatatype);

// Check that the key is a scalar datatype
if ((keyDatatype != INTEGERTYPE) && (keyDatatype != FLOATTYPE) &&
    (keyDatatype != CHARACTERTYPE) && (keyDatatype != BOOLEANTYPE))
{
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex,
        "Key must be of a scalar datatype (int, bool, char, or float)");
}

// Check for a right brace: '}'
if (tokens[0].type != R_BRACE)
    ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting '}');

// If the reference is an r-value, use type-casting operator 'as' to cast the value
// to one of the four scalar datatypes (This must be done since STM does not offer
// run-time type checking.)
```

```
if (!asLvalue)
{
    GetNextToken(tokens);
    if (tokens[0].type != AS)
        ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting type-cast operator 'as'");

    GetNextToken(tokens);
    switch (tokens[0].type)
    {
        case INT:
            datatype = INTEGERTYPE;
            break;
        case BOOL:
            datatype = BOOLEANTYPE;
            break;
        case CHAR:
            datatype = CHARACTERTYPE;
            break;
        case FLOAT:
            datatype = FLOATTYPE;
            break;
        default:
            ProcessCompilerError(tokens[0].sourceLineNumber, tokens[0].sourceLineIndex, "Expecting 'int', 'bool', 'char', or 'float'");
            break;
    }
}
```

```
}

// End of module
GetNextToken(tokens);

ExitModule("AssociativeArrayReference");

}

//-----
void Callback1(int sourceLineNumber,const char sourceLine[])
//-----
{
    cout << setw(4) << sourceLineNumber << " " << sourceLine << endl;
}

//-----
void Callback2(int sourceLineNumber,const char sourceLine[])
//-----
{
    char line[SOURCELINELENGTH+1];

// CODEGENERATION
    sprintf(line,"; %4d %s",sourceLineNumber,sourceLine);
    code.EmitUnformattedLine(line);
```

```
// ENDCODEGENERATION
}

//-----
void GetNextToken(TOKEN tokens[])
//-----
{
    const char* TokenDescription(TOKENTYPE type);

    int i;
    TOKENTYPE type;
    char lexeme[SOURCELINELENGTH + 1];
    int sourceLineNumber;
    int sourceLineIndex;
    char information[SOURCELINELENGTH + 1];

//=====
// Move look-ahead "window" to make room for next token-and-lexeme
//=====

    for (int i = 1; i <= LOOKAHEAD; i++)
        tokens[i - 1] = tokens[i];

    char nextCharacter = reader.GetLookAheadCharacter(0).character;

//=====
// "Eat" white space and comments
```

```
//=====
do
{
    // "Eat" any white-space (blanks and EOLCs and TABCs)
    while ((nextCharacter == ' ')
        || (nextCharacter == READER<CALLBACKSUSED>::EOLC)
        || (nextCharacter == READER<CALLBACKSUSED>::TABC))
        nextCharacter = reader.GetNextCharacter().character;

    // "Eat" line comment
    if ((nextCharacter == '$') && (reader.GetLookAheadCharacter(1).character == '$'))
    {

#define TRACESCANNER
        sprintf(information, "At (%4d:%3d) begin line comment",
            reader.GetLookAheadCharacter(0).sourceLineNumber,
            reader.GetLookAheadCharacter(0).sourceLineIndex);
        lister.ListInformationLine(information);
#endif

        do
            nextCharacter = reader.GetNextCharacter().character;
            while (nextCharacter != READER<CALLBACKSUSED>::EOLC);
    }

    // "Eat" block comments (nesting allowed)
```

```
if ((nextCharacter == '$') && (reader.GetLookAheadCharacter(1).character == '*'))
{
    int depth = 0;

    do
    {
        if ((nextCharacter == '$') && (reader.GetLookAheadCharacter(1).character == '*'))
        {
            depth++;

#ifdef TRACESCANNER
            sprintf(information, "At (%4d:%3d) begin block comment depth = %d",
                    reader.GetLookAheadCharacter(0).sourceLineNumber,
                    reader.GetLookAheadCharacter(0).sourceLineIndex,
                    depth);
            lister.ListInformationLine(information);
#endif

            nextCharacter = reader.GetNextCharacter().character;
            nextCharacter = reader.GetNextCharacter().character;
        }
        else if ((nextCharacter == '*') && (reader.GetLookAheadCharacter(1).character == '$'))
        {

#ifdef TRACESCANNER
            sprintf(information, "At (%4d:%3d)    end block comment depth = %d",

```

```
    reader.GetLookAheadCharacter(0).sourceLineNumber,
    reader.GetLookAheadCharacter(0).sourceLineIndex,
    depth);
    lister.ListInformationLine(information);

#endif

    depth--;
    nextCharacter = reader.GetNextCharacter().character;
    nextCharacter = reader.GetNextCharacter().character;
}

else
    nextCharacter = reader.GetNextCharacter().character;

} while ((depth != 0) && (nextCharacter != READER<CALLBACKSUSED>::EOPC));

if (depth != 0)
    ProcessCompilerError(reader.GetLookAheadCharacter(0).sourceLineNumber,
        reader.GetLookAheadCharacter(0).sourceLineIndex,
        "Unexpected end-of-program");
}

} while ((nextCharacter == ' ')
    || (nextCharacter == READER<CALLBACKSUSED>::EOLC)
    || (nextCharacter == READER<CALLBACKSUSED>::TABC)
    || ((nextCharacter == '$') && (reader.GetLookAheadCharacter(1).character == '$'))
    || ((nextCharacter == '$') && (reader.GetLookAheadCharacter(1).character == '*')));

//=====
// Scan token
```

```
//=====
sourceLineNumber = reader.GetLookAheadCharacter(0).sourceLineNumber;
sourceLineIndex = reader.GetLookAheadCharacter(0).sourceLineIndex;

// reserved words (and <identifier>)
if (isalpha(nextCharacter))
{
    char UClexeme[SOURCELINELENGTH + 1];

    i = 0;
    lexeme[i++] = nextCharacter;
    nextCharacter = reader.GetNextCharacter().character;
    while (isalpha(nextCharacter) || isdigit(nextCharacter) || (nextCharacter == '_'))
    {
        lexeme[i++] = nextCharacter;
        nextCharacter = reader.GetNextCharacter().character;
    }
    lexeme[i] = '\0';
    for (i = 0; i <= (int)strlen(lexeme); i++)
        UClexeme[i] = toupper(lexeme[i]);

    // check if lexeme is the keyword FILE
    if (strcmp(UClexeme, "FILE") == 0)
    {
        type = FILE_T;
```

```
}

// check if lexeme is an instance of an END-token or an ELSE-token
bool isEndToken = false;

if (strcmp(UCLexeme, "END") == 0) {

    isEndToken = true;

    // search for a single space followed by the phrase "main"
    if (nextCharacter == ' ') {

        int secondPart = strlen(lexeme); //marking index of second part
        i = secondPart;
        lexeme[i++] = ' ';
        nextCharacter = reader.GetNextCharacter().character;

        while (isalpha(nextCharacter)) {
            lexeme[i++] = nextCharacter;
            nextCharacter = reader.GetNextCharacter().character;
        }
        lexeme[i] = '\0';

        for (i = secondPart; i <= (int)strlen(lexeme); i++)
            UCLexeme[i] = toupper(lexeme[i]);
    }
}
```

```
if (strcmp(UCLexeme, "END MAIN") == 0) {
    type = ENDMAIN;
}
else if (strcmp(UCLexeme, "END IF") == 0) {
    type = ENDIF;
}
else if (strcmp(UCLexeme, "END WHILE") == 0) {
    type = ENDWHILE;
}
else if (strcmp(UCLexeme, "END FOR") == 0) {
    type = ENDFOR;
}
else if (strcmp(UCLexeme, "END PROC") == 0) {
    type = ENDPROC;
}
else if (strcmp(UCLexeme, "END FUNC") == 0) {
    type = ENDFUNC;
}
}

if (!isEndToken) {
    bool isFound = false;

    i = 0;
```

```
        while (!isFound && (i <= (sizeof(TOKENTABLE) / sizeof(TOKENTABLERECORD)) - 1))
        {
            if (TOKENTABLE[i].isReservedWord && (strcmp(UCLexeme, TOKENTABLE[i].description) == 0))
                isFound = true;
            else
                i++;
        }
        if (isFound)
            type = TOKENTABLE[i].type;
        else
            type = IDENTIFIER;
    }

}

// <integer> and <float>
else if (isdigit(nextCharacter))
{
    i = 0;
    lexeme[i++] = nextCharacter;
    nextCharacter = reader.GetNextCharacter().character;
    while (isdigit(nextCharacter))
    {
        lexeme[i++] = nextCharacter;
        nextCharacter = reader.GetNextCharacter().character;
    }
}
```

```
// if floating point does not exist, set as <integer>
if (nextCharacter != '.')
{
    lexeme[i] = '\0';
    type = INTEGER;
}

// if floating point exists, set as <float>
else
{
    // Add nextCharacter to lexeme, increment index
    lexeme[i++] = nextCharacter;

    // Get next character
    nextCharacter = reader.GetNextCharacter().character;

    // Check if there is a digit after the floating point
    if (!isdigit(nextCharacter))
    {
        ProcessCompilerError(sourceLineNumber, sourceLineIndex,
            "Digit must follow floating point");
    }
    else
    {
        while (isdigit(nextCharacter))
        {
            // Add nextCharacter to lexeme, increment index
            lexeme[i++] = nextCharacter;
        }
    }
}
```

```
lexeme[i++] = nextCharacter;

// Get next character
nextCharacter = reader.GetNextCharacter().character;
}

// Check if 'e' is written after the decimal places
if (nextCharacter == 'e')
{
    // Add nextCharacter to lexeme, increment index
    lexeme[i++] = nextCharacter;

    // Get next character
    nextCharacter = reader.GetNextCharacter().character;

    // Check if a '-' is after the 'e'
    if (nextCharacter == '-')
    {
        // Add nextCharacter to lexeme, increment index
        lexeme[i++] = nextCharacter;

        // Get next character
        nextCharacter = reader.GetNextCharacter().character;
    }

    // Check if there is an integer value after e
```

```
if (!isdigit(nextCharacter))
{
    ProcessCompilerError(sourceLineNumber, sourceLineIndex,
        "e must be followed by an integer value");
}

else
{
    while (isdigit(nextCharacter))
    {
        // Add nextCharacter to lexeme, increment index
        lexeme[i++] = nextCharacter;

        // Get next character
        nextCharacter = reader.GetNextCharacter().character;
    }
}

lexeme[i] = '\0';
type = FLOATVAL;
}

}

else
{
switch (nextCharacter)
```

```
{  
// <character>  
case '\\':  
;  
// Set index for lexeme to 0  
i = 0;  
lexeme[i++] = nextCharacter;  
// Get nextChar  
nextCharacter = reader.GetNextCharacter().character;  
  
// If nextChar is '  
if (nextCharacter == '\\')  
{  
// Give error: "Expecting ASCII character"  
ProcessCompilerError(sourceLineNumber, sourceLineIndex,  
"Expecting ASCII character");  
}  
// Else  
else  
{  
// Add nextChar to lexeme, increment index  
lexeme[i++] = nextCharacter;  
  
// Get nextChar  
nextCharacter = reader.GetNextCharacter().character;
```

```
// If nextChar != '  
if (nextCharacter != '\\')  
{  
    // Give error: "Invalid char literal"  
    ProcessCompilerError(sourceLineNumber, sourceLineIndex,  
        "Invalid char literal");  
}  
// Else  
else  
{  
    // Add nextChar to lexeme, increment index  
    lexeme[i++] = nextCharacter;  
  
    // Add null character to lexeme  
    lexeme[i] = '\\0';  
  
    // Set type to CHARACTER  
    type = CHARACTER;  
  
    // Call reader.GetNextCharacter()  
    reader.GetNextCharacter();  
}  
}  
// break  
break;  
// <string>
```

```
case '':
    i = 0;
    nextCharacter = reader.GetNextCharacter().character;
    while ((nextCharacter != '') && (nextCharacter != READER<CALLBACKSUSED>::EOLC))
    {
        if ((nextCharacter == '\\') && (reader.GetLookAheadCharacter(1).character == '''))
        {
            lexeme[i++] = nextCharacter;
            nextCharacter = reader.GetNextCharacter().character;
        }
        else if ((nextCharacter == '\\') && (reader.GetLookAheadCharacter(1).character == '\\'))
        {
            lexeme[i++] = nextCharacter;
            nextCharacter = reader.GetNextCharacter().character;
        }
        lexeme[i++] = nextCharacter;
        nextCharacter = reader.GetNextCharacter().character;
    }
    if (nextCharacter == READER<CALLBACKSUSED>::EOLC)
        ProcessCompilerError(sourceLineNumber, sourceLineIndex,
                            "Invalid string literal");
    lexeme[i] = '\0';
    type = STRING;
    reader.GetNextCharacter();
    break;
case READER<CALLBACKSUSED>::EOPC:
```

```
{  
    static int count = 0;  
  
    if (++count > (LOOKAHEAD + 1))  
        ProcessCompilerError(sourceLineNumber, sourceLineIndex,  
            "Unexpected end-of-program");  
    else  
    {  
        type = EOPTOKEN;  
        reader.GetNextCharacter();  
        lexeme[0] = '\0';  
    }  
}  
break;  
case ',':  
    type = COMMA;  
    lexeme[0] = nextCharacter; lexeme[1] = '\0';  
    reader.GetNextCharacter();  
    break;  
case ';':  
    type = SEMICOLON;  
    lexeme[0] = nextCharacter; lexeme[1] = '\0';  
    reader.GetNextCharacter();  
    break;  
case '(':  
    type = L_PAREN;
```

```
lexeme[0] = nextCharacter; lexeme[1] = '\0';
reader.GetNextCharacter();
break;

case ')':
    type = R_PAREN;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case '[':
    type = L_BRACKET;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case ']':
    type = R_BRACKET;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case '{':
    type = L_BRACE;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case '}':
    type = R_BRACE;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
```

```
    reader.GetNextCharacter();

    break;

case '<':
    lexeme[0] = nextCharacter;
    nextCharacter = reader.GetNextCharacter().character;
    if (nextCharacter == '=')
    {
        type = LTEQ;
        lexeme[1] = nextCharacter; lexeme[2] = '\0';
        reader.GetNextCharacter();
    }
    else if (nextCharacter == '>')
    {
        type = NOTEQ;
        lexeme[1] = nextCharacter; lexeme[2] = '\0';
        reader.GetNextCharacter();
    }
    else
    {
        type = LT;
        lexeme[1] = '\0';
    }
    break;

// use character look-ahead to "find" other '='

case '=':
    lexeme[0] = nextCharacter;
```

```
if (reader.GetLookAheadCharacter(1).character == '=')
{
    nextCharacter = reader.GetNextCharacter().character;
    lexeme[1] = nextCharacter; lexeme[2] = '\0';
    reader.GetNextCharacter();
    type = EQ;
}

else // a single '=' is an assignment operator
{
    type = ASSIGN;
    lexeme[1] = '\0';
    reader.GetNextCharacter();
}

break;

case '>':
    lexeme[0] = nextCharacter;
    nextCharacter = reader.GetNextCharacter().character;
    if (nextCharacter == '=')
    {
        type = GTEQ;
        lexeme[1] = nextCharacter; lexeme[2] = '\0';
        reader.GetNextCharacter();
    }
    else
    {
        type = GT;
    }
}
```

```
lexeme[1] = '\0';

}

break;

// use character look-ahead to "find" '='

case '!':
    lexeme[0] = nextCharacter;

    if (reader.GetLookAheadCharacter(1).character == '=')

    {
        nextCharacter = reader.GetNextCharacter().character;

        lexeme[1] = nextCharacter; lexeme[2] = '\0';

        reader.GetNextCharacter();

        type = NOTEQ;
    }

else

{
    type = UNKTOKEN;

    lexeme[1] = '\0';

    reader.GetNextCharacter();
}

break;

case '+':
    lexeme[0] = nextCharacter;

    if (reader.GetLookAheadCharacter(1).character == '+')

    {
        nextCharacter = reader.GetNextCharacter().character;

        lexeme[1] = nextCharacter; lexeme[2] = '\0';
```

```
    type = INC;
}
else
{
    type = PLUS;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
}
reader.GetNextCharacter();
break;

case '-':
    lexeme[0] = nextCharacter;
    if (reader.GetLookAheadCharacter(1).character == '-')
    {
        nextCharacter = reader.GetNextCharacter().character;
        lexeme[1] = nextCharacter; lexeme[2] = '\0';
        type = DEC;
    }
    else
    {
        type = MINUS;
        lexeme[0] = nextCharacter; lexeme[1] = '\0';
    }
    reader.GetNextCharacter();
    break;

// use character look-ahead to "find" other '*'
case '*':
```

```
lexeme[0] = nextCharacter;

if (reader.GetLookAheadCharacter(1).character == '*')
{
    nextCharacter = reader.GetNextCharacter().character;
    lexeme[1] = nextCharacter; lexeme[2] = '\0';
    type = POWER;
}

else
{
    type = MULTIPLY;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
}

reader.GetNextCharacter();
break;

case '/':
    type = DIVIDE;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case '%':
    type = MODULUS;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    reader.GetNextCharacter();
    break;

case '^':
    type = POWER;
```

```
lexeme[0] = nextCharacter; lexeme[1] = '\0';

reader.GetNextCharacter();

break;

case ':':

    type = COLON;

    lexeme[0] = nextCharacter; lexeme[1] = '\0';

    reader.GetNextCharacter();

    break;

default:

    type = UNKTOKEN;

    lexeme[0] = nextCharacter; lexeme[1] = '\0';

    reader.GetNextCharacter();

    break;

}

}

tokens[LOOKAHEAD].type = type;

strcpy(tokens[LOOKAHEAD].lexeme, lexeme);

tokens[LOOKAHEAD].sourceLineNumber = sourceLineNumber;

tokens[LOOKAHEAD].sourceLineIndex = sourceLineIndex;

#endif TRACE_SCANNER

sprintf(information, "At (%4d:%3d) token = %12s lexeme = |%s|",

tokens[LOOKAHEAD].sourceLineNumber,

tokens[LOOKAHEAD].sourceLineIndex,

TokenDescription(type), lexeme);
```

```
    lister.ListInformationLine(information);

#endif

}

//-----
const char *TokenDescription(TOKENTYPE type)
//-----
{
    int i;
    bool isFound;

    isFound = false;
    i = 0;

    while ( !isFound && (i <= (sizeof(TOKENTABLE)/sizeof(TOKENTABLERECORD))-1) )
    {
        if ( TOKENTABLE[i].type == type )
            isFound = true;
        else
            i++;
    }

    return ( isFound ? TOKENTABLE[i].description : "???????" );
}
```

Rogue.h

```
//-----  
// Mujeeb Adelekan  
// Rogue compiler "global" definitions and the common classes  
// ROGUEEXCEPTION, LISTER, READER, CODE, and IDENTIFIERTABLE  
  
//  
// *Note* Several common classes are commented to indicate their  
// required evolution to support incremental development of the  
// Rogue compiler  
  
//  
// Rogue.h  
//-----  
  
#define CALLBACKSUSED 2  
  
const int SOURCELINELENGTH      = 512;  
const int LOOKAHEAD             = 2;  
const int LINESPERPAGE          = 60;  
const int MAXIMUMLENGTHIDENTIFIER = 64;  
const int MAXIMUMIDENTIFIERS    = 500;  
  
enum DATATYPE { NOTYPE, INTEGERTYPE, BOOLEANTYPE, CHARACTERTYPE, FLOATTYPE, POINTERTYPE, ASSOCTYPE, ENUMTYPE, FILETYPE };  
  
enum IDENTIFIERSCOPE { GLOBALSCOPE, PROGRAMMODULESCOPE, SUBPROGRAMMODULESCOPE };  
  
enum IDENTIFIERTYPE
```

```
{  
    GLOBAL_VARIABLE,           // static global variable data  
    GLOBAL_CONSTANT,          // static global constant data  
    PROGRAMMODULE_VARIABLE,   // static PROGRAM module local variable data  
    PROGRAMMODULE_CONSTANT,   // static PROGRAM module local constant data  
  
    //-----  
    // ADDED FOR SPL6  
    //-----  
    SUBPROGRAMMODULE_VARIABLE, // automatic, frame-based data  
    SUBPROGRAMMODULE_CONSTANT, // automatic, frame-based data  
    IN_PARAMETER,             // automatic, frame-based data  
    OUT_PARAMETER,            // automatic, frame-based data  
    IO_PARAMETER,             // automatic, frame-based data  
    REF_PARAMETER,            // automatic, frame-based data  
    PROCEDURE_SUBPROGRAMMODULE, // module name  
  
    //-----  
    // ADDED FOR SPL7  
    //-----  
    FUNCTION_SUBPROGRAMMODULE, // module name  
  
    //-----  
    // ADDED to support enumerations  
    //-----  
    GLOBAL_ENUMTYPE,          // name of global enumeration type  
    PROGRAMMODULE_ENUMTYPE,    // name of PROGRAM module enumeration type  
    SUBPROGRAMMODULE_ENUMTYPE // name of SUBPROGRAM module enumeration type  
};
```

```
//=====
class ROGUEEXCEPTION
//=====
{
private:
    char description[80+1];

public:
    //-----
    ROGUEEXCEPTION(const char description[])
    //-----
    {
        strcpy(this->description,description);
    }
    //-----
    char *GetDescription()
    //-----
    {
        return(description);
    }
};

//=====
class LISTER
//=====
```

```
{  
private:  
    const int LINESPERPAGE;  
  
    ofstream LIST;  
    int pageNumber;  
    int linesOnPage;  
    char sourceFileName[80+1];  
  
public:  
    LISTER(const int LINESPERPAGE = 55);  
    ~LISTER();  
    void OpenFile(const char sourceFileName[]);  
    void ListSourceLine(int sourceLineNumber,const char sourceLine[]);  
    void ListInformationLine(const char information[]);  
  
private:  
    void ListTopOfPageHeader();  
};  
  
//-----  
LISTER::LISTER(const int LINESPERPAGE): LINESPERPAGE(LINESPERPAGE)  
//-----  
{  
    pageNumber = 0;  
    linesOnPage = 0;
```

```
}

//-----
LISTER::~LISTER()
//-----
{

    if ( LIST.is_open() )
    {
        LIST.flush();
        LIST.close();
    }
}

//-----
void LISTER::OpenFile(const char sourceFileName[])
//-----
{
    char fullFileName[80+1];

    strcpy(this->sourceFileName,sourceFileName);
    strcat(this->sourceFileName,".rog");
    strcpy(fullFileName,sourceFileName);
    strcat(fullFileName,".list");
    LIST.open(fullFileName,ios::out);
    if ( !LIST.is_open() ) throw( ROGUEEXCEPTION("Unable to open list file") );
    ListTopOfPageHeader();
}
```

```
}

//-----
void LISTER::ListSourceLine(int sourceLineNumber,const char sourceLine[])
//-----
{
    if ( linesOnPage >= LINESPERPAGE )
    {
        ListTopOfPageHeader();
        linesOnPage = 0;
    }
    LIST << setw(4) << sourceLineNumber << " " << sourceLine << endl;
    linesOnPage++;
}

//-----
void LISTER::ListInformationLine(const char information[])
//-----
{
    if ( linesOnPage >= LINESPERPAGE )
    {
        ListTopOfPageHeader();
        linesOnPage = 0;
    }
    LIST << information << endl;
    linesOnPage++;
}
```

```
}

//-----
void LISTER::ListTopOfPageHeader()
//-----
{
/*
"Source file name" Page XXXX
Line Source Line
-----
*/
const char FF = 0X0C;

pageNumber++;

LIST << FF << '""' << sourceFileName << "\" Page " << setw(4) << pageNumber << endl;
LIST << "Line Source Line" << endl;
LIST << "-----" << endl;
}

//=====
struct NEXTCHARACTER
//=====
{
    char character;
    int sourceLineNumber;
    int sourceLineIndex;
```

```
};

//=====
template <int CALLBACKSALLOWED = 5>
class READER
//=====

{
public:
    static const char EOPC = 0;
    static const char EOLC = '\n';
    static const char TABC = '\t';

private:
    const int SOURCELINELENGTH;
    const int LOOKAHEAD;

    char *sourceLine;
    int sourceLineNumber;
    int sourceLineIndex;
    NEXTCHARACTER *nextCharacters;
    ifstream SOURCE;
    LISTER *lister;
    bool atEOP;
    int numberCallbacks;
    void (*CallbackFunctions[CALLBACKSALLOWED+1])
        (int sourceLineNumber,const char sourceLine[]);
}
```

```
public:  
    READER(const int SOURCELINELENGTH = 512,const int LOOKAHEAD = 0);  
    ~READER();  
    void OpenFile(const char sourceFileName[]);  
    void SetLister(LISTER *lister);  
    NEXTCHARACTER GetNextCharacter();  
    NEXTCHARACTER GetLookAheadCharacter(int index);  
    void AddCallbackFunction(void (*CallbackFunction)  
        (int sourceLineNumber,const char sourceLine[]));  
  
private:  
    void ReadSourceLine();  
};  
  
//-----  
template<int CALLBACKSALLOWED>  
READER<CALLBACKSALLOWED>::READER(const int SOURCELINELENGTH,const int LOOKAHEAD):  
    SOURCELINELENGTH(SOURCELINELENGTH),LOOKAHEAD(LOOKAHEAD)  
//-----  
{  
    sourceLine = new char [ SOURCELINELENGTH+2 ];  
    nextCharacters = new NEXTCHARACTER [ LOOKAHEAD+1 ];  
    sourceLineNumber = 0;  
    atEOP = false;  
    numberCallbacks = 0;  
}
```

```
//-----
template<int CALLBACKSALLOWED>
READER<CALLBACKSALLOWED>::~READER()
//-----
{
    delete [] sourceLine;
    delete [] nextCharacters;
    if ( SOURCE.is_open() ) SOURCE.close();
}

//-----
template<int CALLBACKSALLOWED>
void READER<CALLBACKSALLOWED>::OpenFile(const char sourceFileName[])
//-----
{
    char fullFileName[80+1];

    strcpy(fullFileName,sourceFileName);
    strcat(fullFileName,".rog");
    SOURCE.open(fullFileName,ios::in);
    if ( !SOURCE.is_open() ) throw( ROGUEEXCEPTION("Unable to open source file") );

// Read first source line and "fill" nextCharacters[]
    ReadSourceLine();
    for ( int i = 0; i <= LOOKAHEAD; i++)

```

```
{  
    nextCharacters[i].character = READER::EOPC;  
    nextCharacters[i].sourceLineNumber = 0;  
    nextCharacters[i].sourceLineIndex = 0;  
}  
  
for (int i = 0; i <= LOOKAHEAD; i++)  
    GetNextCharacter();  
}  
  
//-----  
template<int CALLBACKSALLOWED>  
void READER<CALLBACKSALLOWED>::SetLister(LISTER *lister)  
//-----  
{  
    this->lister = lister;  
}  
  
//-----  
template<int CALLBACKSALLOWED>  
NEXTCHARACTER READER<CALLBACKSALLOWED>::GetNextCharacter()  
//-----  
{  
    char character;  
  
    // Move look-ahead "window" to make room for next character  
    for (int i = 1; i <= LOOKAHEAD; i++)
```

```
nextCharacters[i-1] = nextCharacters[i];

nextCharacters[LOOKAHEAD].sourceLineNumber = sourceLineNumber;
nextCharacters[LOOKAHEAD].sourceLineIndex = sourceLineIndex;

if ( atEOP )
    character = READER::EOPC;
//    character = EOPC;

else
{
    if ( sourceLineIndex <= ((int) strlen(sourceLine)-1) )
    {
        character = sourceLine[sourceLineIndex];
        sourceLineIndex += 1;
    }
    else
    {
        character = READER::EOLC;
        ReadSourceLine();
    }
}

// Only non-printable characters allowed are EOPC, '\n', and '\t', others are changed to ' '
if ( iscntrl(character)
&& !(character == READER::EOPC)
|| (character == READER::EOLC)
```

```
    || (character == READER::TABC)
    )
}

character = ' ';

nextCharacters[LOOKAHEAD].character = character;

#ifndef TRACEREADER
{
    char information[80+1];

    if      ( isprint(character) )
        sprintf(information,"At (%4d:%3d) %02X = %c",
            nextCharacters[LOOKAHEAD].sourceLineNumber,
            nextCharacters[LOOKAHEAD].sourceLineIndex,
            character,character);
    else if ( character == READER::EOPC )
        sprintf(information,"At (%4d:%3d) %02X = EOPC",
            nextCharacters[LOOKAHEAD].sourceLineNumber,
            nextCharacters[LOOKAHEAD].sourceLineIndex,
            character);
    else if ( character == READER::EOLC )
        sprintf(information,"At (%4d:%3d) %02X = EOLC",
            nextCharacters[LOOKAHEAD].sourceLineNumber,
            nextCharacters[LOOKAHEAD].sourceLineIndex,
            character);
}
```

```
else if ( character == READER::TABC )
    sprintf(information,"At (%4d:%3d) %02X = TABC",
    nextCharacters[LOOKAHEAD].sourceLineNumber,
    nextCharacters[LOOKAHEAD].sourceLineIndex,
    character);
else
    sprintf(information,"At (%4d:%3d) %02X = ???",
    nextCharacters[LOOKAHEAD].sourceLineNumber,
    nextCharacters[LOOKAHEAD].sourceLineIndex,
    character);
lister->ListInformationLine(information);
}

#endif

return( nextCharacters[0] );
}

//-----
template<int CALLBACKSALLOWED>
NEXTCHARACTER READER<CALLBACKSALLOWED>::GetLookAheadCharacter(int index)
//-----
{
// index in [ 0,LOOKAHEAD ] where index = 0 means last GetNextCharacter() returned
if ( (0 <= index) && (index <= LOOKAHEAD) )
    return( nextCharacters[index] );
else
```

```
throw( ROGUEEXCEPTION("GetLookAheadCharacter() index out-of-range") );  
}  
  
//-----  
template<int CALLBACKSALLOWED>  
void READER<CALLBACKSALLOWED>::AddCallbackFunction(  
    void (*CallbackFunction)(int sourceLineNumber,const char sourceLine[]))  
//-----  
{  
    if ( numberCallbacks <= CALLBACKSALLOWED )  
        CallbackFunctions[++numberCallbacks] = CallbackFunction;  
    else  
        throw( ROGUEEXCEPTION("Too many callback functions") );  
}  
  
//-----  
template<int CALLBACKSALLOWED>  
void READER<CALLBACKSALLOWED>::ReadSourceLine()  
//-----  
{  
    if ( SOURCE.eof() )  
        atEOP = true;  
    else  
    {  
        SOURCE.getline(sourceLine,SOURCELINELENGTH+2);  
        sourceLineNumber++;
```

```
    strcat(sourceLine, "\n");

    if ( SOURCE.fail() && !SOURCE.eof() )
    {

        lister->ListInformationLine("***** Source line too long!");

        SOURCE.clear();

    }

    // Erase *ALL* control characters at end of source line (if any)

    while ( (0 <= (int) strlen(sourceLine)-1) &&

            iscntrl(sourceLine[(int) strlen(sourceLine)-1]) )

        sourceLine[(int) strlen(sourceLine)-1] = '\0';

    sourceLineIndex = 0;

    lister->ListSourceLine(sourceLineNumber,sourceLine);

    // Give each callback function the opportunity to process newly-read source line

    for (int i = 1; i <= numberCallbacks; i++)

        (*CallbackFunctions[i])(sourceLineNumber,sourceLine);

    }

}

//=====

class IDENTIFIERTABLE

//=====

{

private:

    struct IDENTIFIERRECORD
```

```
{  
    int scope;  
    char lexeme[MAXIMUMLENGTHIDENTIFIER+1];  
    IDENTIFIERTYPE identifierType;  
    char reference[MAXIMUMLENGTHIDENTIFIER+1];  
    DATATYPE datatype;  
//-----  
// ADDED FOR SPL8  
//-----  
    int dimensions;  
//-----  
// **** ADDED for pointer support ****  
//-----  
    DATATYPE contentDatatype;  
//-----  
// **** ADDED for enumeration support ****  
//-----  
    char enumType[MAXIMUMLENGTHIDENTIFIER+1];  
};  
  
private:  
    static const char IDENTIFIERTYPENAMES[][][26+1];  
    static const char DATATYPENAMES[][][9+1];  
  
private:  
    int capacity;
```

```
int identifiers;
IDENTIFIERRECORD *identifierTable;
int scopes;
int *scopeTable;
LISTER *lister;

public:
    IDENTIFIERTABLE(LISTER *lister,int capacity);
    ~IDENTIFIERTABLE();
    int GetIndex(const char lexeme[],bool &isInTable);
//-----
// MODIFIED FOR SPL8
//-----

/*** AddToTable() edited to support pointers ***/
void AddToTable(const char lexeme[],IDENTIFIERTYPE identifierType,
                DATATYPE datatype,const char reference[],int dimensions = 0, DATATYPE contentDatatype = NOTYPE);
void EnterNestedStaticScope();
void ExitNestedStaticScope();
void DisplayTableContents(const char description[]);

/*
Assume index was determined by a prior successful call to GetIndex() as
precondition of the remaining accessor member functions.
*/
bool IsInCurrentScope(int index)
```

```
{  
    return( identifierTable[index].scope == scopes );  
}  
  
int GetScope(int index)  
{  
    return( identifierTable[index].scope );  
}  
  
IDENTIFIERTYPE GetType(int index)  
{  
    return( identifierTable[index].identifierType );  
}  
  
char *GetLexeme(int index)  
{  
    return( identifierTable[index].lexeme );  
}  
  
char *GetReference(int index)  
{  
    return( identifierTable[index].reference );  
}  
  
DATATYPE GetDatatype(int index)  
{  
    return( identifierTable[index].datatype );  
}  
  
//-----  
// ADDED FOR SPL6  
//-----
```

```
int GetCountOfFormalParameters(int index);

//-----
// ADDED FOR SPL8
//-----

int GetDimensions(int index)
{
    return( identifierTable[index].dimensions );
}

//-----
// **** ADDED for pointer support ****
//-----

void SetContentDatatype(int index, DATATYPE contentDatatype)
{
    identifierTable[index].contentDatatype = contentDatatype;
}

DATATYPE GetContentDatatype(int index)
{
    return (identifierTable[index].contentDatatype);
}

//-----
// **** ADDED for enumeration support ****
//-----

void SetEnumType(int index, char enumType[])
{
    strcpy(identifierTable[index].enumType, enumType);
```

```
}

string GetEnumType(int index)
{
    return (identifierTable[index].enumType);
}

};

//-----

const char IDENTIFIERTABLE::IDENTIFIERTYPENAMES[][26+1] =
//-----
{

    "GLOBAL_VARIABLE",
    "GLOBAL_CONSTANT",
    "PROGRAMMODULE_VARIABLE",
    "PROGRAMMODULE_CONSTANT",
//-----
// ADDED FOR SPL6
//-----
    "SUBPROGRAMMODULE_VARIABLE",
    "SUBPROGRAMMODULE_CONSTANT",
    "IN_PARAMETER",
    "OUT_PARAMETER",
    "IO_PARAMETER",
    "REF_PARAMETER",
    "PROCEDURE_SUBPROGRAMMODULE",
```

```
//-----
// ADDED FOR SPL7
//-----
"FUNCTION_SUBPROGRAMMODULE",
//-----
// ADDED to support enumerations
//-----
"GLOBAL_ENUMTYPE",
"PROGRAMMODULE_ENUMTYPE",
"SUBPROGRAMMODULE_ENUMTYPE"
};

//-----
const char IDENTIFIERTABLE::DATATYPENAMES[ ][9+1] =
//-----
{
    "N/A",
    "INTEGER",
    "BOOLEAN",
    "CHARACTER",
    "FLOAT",
    "POINTER",
    "ASSOC",
    "ENUMTYPE",
    "FILE"
};
```

```
//-----
IDENTIFIERTABLE::IDENTIFIERTABLE(LISTER *lister,int capacity)
//-----
{
    this->lister = lister;
    this->capacity = capacity;
    identifierTable = new IDENTIFIERRECORD [ capacity+1 ];
    identifiers = 0;
    scopeTable = new int [ capacity+1 ];
    scopes = 0;
}

//-----
IDENTIFIERTABLE::~IDENTIFIERTABLE()
//-----
{
    delete [] identifierTable;
    delete [] scopeTable;
}

//-----
int IDENTIFIERTABLE::GetIndex(const char lexeme[],bool &isInTable)
//-----
{
    /*
```

Try to find identifier's lexeme in identifier table working from the end of the table toward the beginning.

```
/*
char UClexeme1[SOURCELINELENGTH+1],UClexeme2[SOURCELINELENGTH];
int index;
bool isInCurrentScope;

for (int i = 0; i <= (int) strlen(lexeme); i++)
    UClexeme1[i] = toupper(lexeme[i]);
isInTable = false;
index = identifiers;
while ( (index >= 1) && !isInTable )
{
    for (int i = 0; i <= (int) strlen(identifierTable[index].lexeme); i++)
        UClexeme2[i] = toupper(identifierTable[index].lexeme[i]);
    if ( strcmp(UClexeme1,UClexeme2) == 0 )
    {
        isInTable = true;
        isInCurrentScope = (identifierTable[index].scope == scopes);
    }
    else
        index -= 1;
}

#ifndef TRACEIDENTIFIERTABLE
{
```

```

char information[SOURCELINELENGTH+1];

if ( isInTable )
    sprintf(information,"Found identifier \"%s\" at index = %d (%s)",
        lexeme,index,((isInCurrentScope) ? "is in current scope" : "not in current scope"));
else
    sprintf(information,"Did not find identifier \"%s\"",lexeme);
lister->ListInformationLine(information);
}

#endif

return( index );
}

//-----

void IDENTIFIERTABLE::AddToTable(const char lexeme[],IDENTIFIERTYPE identifierType,
        DATATYPE datatype,const char reference[],int dimensions /* = 0*/,
        DATATYPE contentDatatype) /** contentDatatype added **/;

//-----
{
/*
Assumes a prior reference to GetIndex() has already guaranteed that the
identifier being added is *NOT* in the identifier table
*/
if ( identifiers > capacity )
    throw( ROGUEEXCEPTION("Identifier table capacity exceeded") );

```

```
else
{
    identifiers++;

    identifierTable[identifiers].scope = scopes;
    strncpy(identifierTable[identifiers].lexeme, lexeme, MAXIMUMLENGTHIDENTIFIER);
    identifierTable[identifiers].identifierType = identifierType;
    strcpy(identifierTable[identifiers].reference, reference);
    identifierTable[identifiers].datatype = datatype;
    identifierTable[identifiers].dimensions = dimensions;
    identifierTable[identifiers].contentDatatype = contentDatatype; /*** added ***/

    strcpy(identifierTable[identifiers].enumType, ""); /*** added ***/
}

#ifndef TRACEIDENTIFIERTABLE
{
    char information[SOURCELINELENGTH+1];

    sprintf(information, "Added identifier \"%s\" at index = %d, reference = %s, identifier type = %s, data type = %s, dimensions = %d, contents type
= %s",
            lexeme, identifiers,
            strcmp(identifierTable[identifiers].reference, "\0") == 0 ? "N/A" : identifierTable[identifiers].reference,
            IDENTIFIERTYPENAMES[identifierTable[identifiers].identifierType],
            DATATYPENAMES[identifierTable[identifiers].datatype],
            dimensions,
            DATATYPENAMES[identifierTable[identifiers].contentDatatype]); /*** added ***/
}
```

```
    lister->ListInformationLine(information);

}

#endif

}

//-----

void IDENTIFIERTABLE::EnterNestedStaticScope()
//-----
{
    scopeTable[++scopes] = identifiers;

#endif TRACEIDENTIFIERTABLE
{
    char information[SOURCELINELENGTH+1];

    sprintf(information,"Begin nested scope #%d, identifier table index = %d",
            scopes,identifiers);
    lister->ListInformationLine(information);
}

#endif

}

//-----
// UPDATED FOR SPL6, SPL7
```

```
//-----
void IDENTIFIERTABLE::ExitNestedStaticScope()
//-----
{
/*
Remove the identifiers of *ALL* local variables/constants in the module scope
just ended *EXCEPT* for subprogram module formal parameters. Mark the formal
parameters so they will not be "found" by subsequent searches. The formal
parameters are retained so references to their module can be compiled.

>Note* The subprogram module identifier is retained because it is in the
scope just re-entered.

*/
identifiers = scopeTable[scopes--];
if ( (identifierTable[identifiers].identifierType == PROCEDURE_SUBPROGRAMMODULE) ||
     (identifierTable[identifiers].identifierType == FUNCTION_SUBPROGRAMMODULE) )
    while ( (identifierTable[identifiers+1].identifierType == IN_PARAMETER) ||
            (identifierTable[identifiers+1].identifierType == OUT_PARAMETER) ||
            (identifierTable[identifiers+1].identifierType == IO_PARAMETER) ||
            (identifierTable[identifiers+1].identifierType == REF_PARAMETER) )
    {
        identifiers++;
/*
A null-string lexeme ensures subprogram module formal parameters are
not found when out-of-scope, but still allows identifier type to
remain available for subprogram reference semantic analysis.
*/
    }
}
```

```

        identifierTable[identifiers].lexeme[0] = '\0';

    }

#ifndef TRACEIDENTIFIERTABLE
{
    char information[SOURCELINELENGTH+1];

    sprintf(information,"End nested scope #%-d, identifier table index now = %d",
            scopes+1,identifiers);
    lister->ListInformationLine(information);
}
#endif

}

//-----
void IDENTIFIERTABLE::DisplayTableContents(const char description[]) /** updated to include contents datatype and enum types ***/
//-----
{
/*
=====
description

# Scope Data type Contents type Dimensions Type          Reference          Lexeme
----- -----
XXX   XXX XXXXXXXXXX XXXXXXXXXXXXXXXX      XXX XXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXX XX...

```

```
    .  
    .  
    .  
=====  
*/  
  
lister->ListInformationLine  
( "=====");  
  
lister->ListInformationLine(description);  
  
lister->ListInformationLine  
( "# Scope Data type Contents type Dimensions Type Reference Lexeme (Enum Type)");  
  
lister->ListInformationLine  
( "--- ----- ----- ----- ----- ----- ----- -----");  
  
for (int i = 1; i <= identifiers; i++)  
{  
  
    char information[SOURCELINELENGTH+1];  
  
    char enumTypeName[MAXIMUMLENGTHIDENTIFIER + 1 + 3];  
  
  
    sprintf(information,"%3d %3d %-9s %-13s %3d %-26s %-20.20s %s",  
            i,identifierTable[i].scope,  
            ((identifierTable[i].datatype == NOTYPE) ? " " : DATATYPENAMES[identifierTable[i].datatype]),  
            ((identifierTable[i].contentDatatype == NOTYPE) ? " " : DATATYPENAMES[identifierTable[i].contentDatatype]),  
            identifierTable[i].dimensions,  
            IDENTIFIERTYPENAMES[identifierTable[i].identifierType],  
            identifierTable[i].reference,  
            identifierTable[i].lexeme);  
  
  
    // Print enumType if needed
```

```
if (identifierTable[i].datatype == ENUMTYPE)
{
    sprintf(enumTypeName, " (%s)", identifierTable[i].enumType);
    strcat(information, enumTypeName);
}

lister->ListInformationLine(information);
}

lister->ListInformationLine
("=====");
}

//-----
// ADDED FOR SPL6
//-----

int IDENTIFIERTABLE::GetCountOfFormalParameters(int index)
//-----
{

/*
Assumes compiler has verified that

(1) index represents a subprogram module identifier; and
(2) all the subprogram module formal parameters immediately follow the
    subprogram module identifier in identifier table
*/
int count;
```

```
index++;
count = 0;

while ( (identifierTable[index].identifierType == IN_PARAMETER) ||
        (identifierTable[index].identifierType == OUT_PARAMETER) ||
        (identifierTable[index].identifierType == IO_PARAMETER) ||
        (identifierTable[index].identifierType == REF_PARAMETER) )

{
    count++;
    index++;
}

#endif TRACEIDENTIFIERTABLE

{
    char information[SOURCELINELENGTH+1];

    sprintf(information,"Subprogram module \"%s\" has %d formal parameters",
            identifierTable[index].lexeme,count);
    lister->ListInformationLine(information);
}

#endif

return( count );
}

//=====
```

```
class CODE
//=====
{
/*
SPL static data--global variable/constant definitions, string literals, and
PROGRAM module variables/constants--are parsed both very early in the
compile and near the end of the compile. As a result, their STM code must
be "stored" because the code cannot be emitted until *AFTER* the entire
SPL program has been parsed.

SPL dynamic (frame-resident) data--subprogram module formal parameters and
local variables/constants--have storage space accounted for when their
definitions are parsed. The frame space is then allocated using STM
statements emitted as part of the subprogram module prolog code.

*/
private:
    struct DATARECORD
    {
        char mnemonic[SOURCELINELENGTH+1];
        char operand[SOURCELINELENGTH+1];
        char comment[SOURCELINELENGTH+1];
    };

private:
    ofstream STM;
    char codeFileName[80+1];
```

```
vector<DATARECORD> staticdata;
int SBOffset;
int labelsuffix;
//-----
// ADDED FOR SPL6
//-----
vector<DATARECORD> framedata;
int FBOffset;
bool isInModuleBody;
int moduleIdentifierIndex;
IDENTIFIERTYPE moduleIdentifierType;
//-----

public:
CODE();
~CODE();
void OpenFile(const char sourceFileName[]);
void EmitBeginningCode(const char sourceFileName[]);
void EmitEndingCode();
int GetSBOffset();
void AddRWToStaticData(int operand,const char comment[],char reference[]);
void AddDWToStaticData(const char operand[],const char comment[],char reference[]);
void AddDSToStaticData(const char operand[],const char comment[],char reference[]);
void EmitStaticData();
int LabelSuffix();
void EmitFormattedLine(const char label[],const char mnemonic[],const char operand[] = "",const char comment[] = "");
```

```
void EmitUnformattedLine(const char line[]);

//-----
// ADDED FOR SPL6
//-----

void ResetFrameData();
void IncrementFBOffset(int increment = 1);
int GetFBOffset();
void AddInstructionToInitializeFrameData(const char mnemonic[], const char operand[], const char comment[] = "");
void EmitFrameData();
void EnterModuleBody(IDENTIFIERTYPE moduleIdentifierType, int moduleIdentifierIndex);
void ExitModuleBody();
bool IsInModuleBody(IDENTIFIERTYPE moduleIdentifierType);
int GetModuleIdentifierIndex();

//-----
};

//-----
CODE::CODE()
//-----
{

    staticdata.clear();
    SBOffset = 0;
    labelsuffix = 0;

//-----
// ADDED FOR SPL6
//-----
```

```
    isInModuleBody = false;  
}  
  
//-----  
CODE::~CODE()  
//-----  
{  
    if ( STM.is_open() )  
    {  
        STM.flush();  
        STM.close();  
    }  
}  
  
//-----  
void CODE::OpenFile(const char sourceFileName[])  
//-----  
{  
    strcpy(codeFileName,sourceFileName);  
    strcat(codeFileName,".stm");  
    STM.open(codeFileName,ios::out);  
    if ( !STM.is_open() ) throw( ROGUEEXCEPTION("Unable to open code file") );  
}  
  
//-----  
void CODE::EmitBeginningCode(const char sourceFileName[])
```

```
//-----
{
    char line[SOURCELINELENGTH+1];

    EmitUnformattedLine(";-----");
    sprintf(line, "; %s.stm", sourceFileName); EmitUnformattedLine(line);
    EmitUnformattedLine(";-----");
    EmitUnformattedLine("; SVC numbers");

    EmitFormattedLine("SVC_DONOTHING"      , "EQU", "0D0", "force context switch");
    EmitFormattedLine("SVC_TERMINATE"       , "EQU", "0D1");
    EmitFormattedLine("SVC_READ_INTEGER"   , "EQU", "0D10");
    EmitFormattedLine("SVC_WRITE_INTEGER"  , "EQU", "0D11");
    EmitFormattedLine("SVC_WRITE_HEXADECIMAL", "EQU", "0D12", "write integer in hexdecimal");
    EmitFormattedLine("SVC_READ_FLOAT"     , "EQU", "0D20");
    EmitFormattedLine("SVC_WRITE_FLOAT"    , "EQU", "0D21");
    EmitFormattedLine("SVC_READ_BOOLEAN"   , "EQU", "0D30");
    EmitFormattedLine("SVC_WRITE_BOOLEAN"  , "EQU", "0D31");
    EmitFormattedLine("SVC_READ_CHARACTER" , "EQU", "0D40");
    EmitFormattedLine("SVC_WRITE_CHARACTER", "EQU", "0D41");
    EmitFormattedLine("SVC_WRITE_ENDL"     , "EQU", "0D42");
    EmitFormattedLine("SVC_READ_STRING"    , "EQU", "0D50");
    EmitFormattedLine("SVC_WRITE_STRING"   , "EQU", "0D51");
    EmitFormattedLine("SVC_CREATE_FILE"    , "EQU", "0D60", "create a new file");
    EmitFormattedLine("SVC_SET_READ_ACCESS", "EQU", "0D61", "set file read access");
    EmitFormattedLine("SVC_INITIALIZE_HEAP" , "EQU", "0D90");
    EmitFormattedLine("SVC_ALLOCATE_BLOCK"  , "EQU", "0D91");
```

```
EmitFormattedLine("SVC DEALLOCATE_BLOCK", "EQU", "0D92");

EmitUnformattedLine("");

EmitFormattedLine("", "ORG", "0X0000");

EmitUnformattedLine("");

EmitFormattedLine("", "JMP", "MAINPROGRAM");

}

//-----

void CODE::EmitEndingCode()
//-----
{

    char reference[SOURCELINELENGTH+1];

    EmitUnformattedLine("");
    EmitUnformattedLine(";-----");
    EmitUnformattedLine("; Issue \"Run-time error #X..X near line #X..X\" to handle run-time errors");
    EmitUnformattedLine(";-----");
    EmitFormattedLine("HANDLERUNTIMEERROR", "EQU", "*");
    EmitFormattedLine("", "SVC", "#SVC_WRITE_ENDL");
    AddDSToStaticData("Run-time error #","",reference);
    EmitFormattedLine("", "PUSHA",reference);
    EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");
    EmitFormattedLine("", "SVC", "#SVC_WRITE_INTEGER");
    AddDSToStaticData(" near line #","",reference);
    EmitFormattedLine("", "PUSHA",reference);
    EmitFormattedLine("", "SVC", "#SVC_WRITE_STRING");
```

```
EmitFormattedLine("", "SVC", "#SVC_WRITE_INTEGER");

EmitFormattedLine("", "SVC", "#SVC_WRITE_ENDL");

EmitFormattedLine("", "PUSH", "#0D1");

EmitFormattedLine("", "SVC", "#SVC_TERMINATE");

EmitUnformattedLine("");
EmitUnformattedLine(";-----");
EmitUnformattedLine("; Static allocation of global data and PROGRAM module data");
EmitUnformattedLine(";-----");
EmitFormattedLine("STATICDATA", "EQU", "*");
EmitStaticData();

EmitUnformattedLine("");
EmitUnformattedLine(";-----");
EmitUnformattedLine("; Heap space for dynamic memory allocation (to support future SPL syntax)");
EmitUnformattedLine(";-----");
EmitFormattedLine("HEAPBASE", "EQU", "*");
EmitFormattedLine("HEAPSIZE", "EQU", "0B0001000000000000", "8K bytes = 4K words");

EmitUnformattedLine("");
EmitUnformattedLine(";-----");
EmitUnformattedLine("; Run-time stack");
EmitUnformattedLine(";-----");
EmitFormattedLine("RUNTIMESTACK", "EQU", "0xFFFFE");

}
```

```
-----  
int CODE::GetSBOffset()  
-----  
{  
    return( SBOffset );  
}  
  
-----  
void CODE::AddRWToStaticData(int operand,const char comment[],char reference[])  
-----  
{  
    DATARECORD r;  
  
    strcpy(r.mnemonic,"RW");  
    sprintf(r.operand,"0D%d",operand);  
    strcpy(r.comment,comment);  
    staticdata.push_back( r );  
    sprintf(reference,"SB:0D%d",SBOffset);  
    SBOffset += operand;  
}  
  
-----  
void CODE::AddDWToStaticData(const char operand[],const char comment[],char reference[])  
-----  
{  
    DATARECORD r;
```

```
strcpy(r.mnemonic,"DW");
strcpy(r.operand,operand);
strcpy(r.comment,comment);
staticdata.push_back( r );
sprintf(reference,"SB:0D%d",SBOffset);
SBOffset += 1;

}

//-----
void CODE::AddDSToStaticData(const char operand[],const char comment[],char reference[])
//-----
{

DATARECORD r;

strcpy(r.mnemonic,"DS");
sprintf(r.operand,"\"%s\"",operand);
strcpy(r.comment,comment);
staticdata.push_back( r );
sprintf(reference,"SB:0D%d",SBOffset);
SBOffset += 2 + (int) strlen(operand);

}

//-----
void CODE::EmitStaticData()
//-----
{
```

```
for(int i = 0; i <= (int) staticdata.size()-1; i++)
    EmitFormattedLine("",staticdata[i].mnemonic,staticdata[i].operand,staticdata[i].comment);
}

//-----
int CODE::LabelSuffix()
//-----
{
    labelsuffix += 10;
    return( labelsuffix );
}

//-----
void CODE::EmitFormattedLine(const char label[],const char mnemonic[],const char operand[],const char comment[])
//-----
{
/*
    1         2         3         4         5         6         7         8
1234567890123456789012 ^56789012 ^5678901234567890123 ^6789012345678901234567890
*/
    char line[110+1];

    if ( (int) strlen(comment) > 0 )
        sprintf(line,"%-22s %-9s %-20s ; %s",label,mnemonic,operand,comment);
    else
        sprintf(line,"%-22s %-9s %s",label,mnemonic,operand);
```

```
STM << line << endl;
}

//-----
void CODE::EmitUnformattedLine(const char line[])
//-----
{
    STM << line << endl;
}

//-----
void CODE::ResetFrameData()
//-----
{
    framedata.clear();
    FBOffset = 0;
}

//-----
void CODE::IncrementFBOffset(int increment/* = 1*/)
//-----
{
    this->FBOffset += increment;
}

//-----
```

```
int CODE::GetFBOffset()
//-----
{
    return( FBOffset );
}

//-----
void CODE::AddInstructionToInitializeFrameData(const char mnemonic[],const char operand[],const char comment[])
//-----
{
    DATARECORD r;

    strcpy(r.mnemonic,mnemonic);
    strcpy(r.operand,operand);
    strcpy(r.comment,comment);
    framedata.push_back( r );
}

//-----
void CODE::EmitFrameData()
//-----
{
    for(int i = 0; i <= (int) framedata.size()-1; i++)
        EmitFormattedLine("",framedata[i].mnemonic,framedata[i].operand,framedata[i].comment);
}
```

```
-----  
void CODE::EnterModuleBody(IDENTIFIERTYPE moduleIdentifierType,int moduleIdentifierIndex)  
-----  
{  
    isInModuleBody = true;  
    this->moduleIdentifierType = moduleIdentifierType;  
    this->moduleIdentifierIndex = moduleIdentifierIndex;  
}  
  
-----  
void CODE::ExitModuleBody()  
-----  
{  
    isInModuleBody = false;  
}  
  
-----  
bool CODE::IsInModuleBody(IDENTIFIERTYPE moduleIdentifierType)  
-----  
{  
    return( isInModuleBody && (this->moduleIdentifierType == moduleIdentifierType) );  
}  
  
-----  
int CODE::GetModuleIdentifierIndex()  
-----
```

```
{  
    return( moduleIdentifierIndex );  
}  
  
STM.c  
-----  
// Dr. Art Hanna  
// Modified by Mujeeb Adelekan  
// Simple Target Machine (STM) for SPL featuring a two-pass  
// assembler and simulation of a stack-based ISA  
// 9.25.2018 Added CONCATS (concatenate strings)  
// 9.15.2020 Added escape sequence codes to <string> scanning in  
//           GetNextToken()  
  
#define VERSION "September 15, 2020"  
  
// STM.c  
-----  
#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
#include <cctype.h>  
#include <string.h>  
#include <math.h>  
  
#define SOURCELINELENGTH      512  
#define LINESPERPAGE          55
```

```

#define EOPC          (1)
#define EOLC          (2)
#define FF            0X0C

#define SIZEOFIDENTIERTABLE    500
#define MAXIMUMLENGTHIDENTIFIER 64

#define HIBYTE(word) ((0xFF00u & word) >> 8)
#define LOBYTE(word) ((0X00FFu & word)      )
#define TORF(word)   ((word == 0xFFFFu) ? 'T' : ((word == 0X0000u)? 'F' : '?'))
#define SIGNED(x)   (((x) <= 0X7FFF) ? (int) (x) : ( (int) ((x)-65536)) )
#define UNSIGNED(x) (((x) <= 32767) ? (WORD) (x) : ((WORD) ((x)-65536)) )

typedef unsigned char BYTE;
typedef unsigned short int WORD;

/*
=====
STM assembly-language grammar expressed in BNF
=====

<STMProgram>      ::= { [ <statement> ] EOLC }* EOPC

<statement>        ::= <instruction> [ <comment> ]
                      | <comment>

<instruction>      ::= [ <identifier> ] <HWMnemonic> [ <operand> ]

```

```

| [ <identifier> ] ORG             <I16>
| <identifier> EQU             (( <W16> | * ))
| [ <identifier> ] RW            [ <I16> ]
| [ <identifier> ] DW            <W16>
| [ <identifier> ] DS            <string>

<HWMnemonic> ::= || See list of H/W operations

<operand> ::= <memory>
| #<W16>
| <A16>           || for jump/CALL instructions *ONLY*

<memory> ::= #<W16>           || mode = 0X00, immediate
| <A16>           || mode = 0X01, memory direct
| @<A16>          || mode = 0X02, memory indirect
| $<A16>          || mode = 0X03, memory indexed
| SP:<I16>         || mode = 0X04, SP-relative direct
| @SP:<I16>        || mode = 0X05, SP-relative indirect
| $SP:<I16>        || mode = 0X06, SP-relative indexed
| FB:<I16>         || mode = 0X07, FB-relative direct
| @FB:<I16>        || mode = 0X08, FB-relative indirect
| $FB:<I16>        || mode = 0X09, FB-relative indexed
| SB:<I16>          || mode = 0X0A, SB-relative direct
| @SB:<I16>         || mode = 0X0B, SB-relative indirect
| $SB:<I16>         || mode = 0X0C, SB-relative indexed

```

```

<W16>      ::= (( <I16> | <F16> | true | false | <character> | <identifier> ))

<A16>      ::= <identifier>

<identifier> ::= <letter> { (( <letter> | <dit> | _ )) }*

<I16>      ::= [ (( + | - )) ] (( 0D <dit> { <dit> }*
|          0X <hit> { <hit> }*
|          0B <bit> { <bit> }* ))

<F16>      ::= [ (( + | - )) ] 0F <dit> { <dit> }* . <dit> { <dit> }* [ E [ - ] <dit> { <dit> }* ]

<bit>      ::= 0 | 1
<dit>      ::= <bit> | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hit>      ::= <dit> | A | B | C | D | E | F
<character> ::= '<ASCIICharacter>'      || *Note* escape sequences \\ and \
<string>    ::= "{ <ASCIICharacter> }*"   || *Note* escape sequences \n, \t, \v, \b, \r, \a, \\, \
<letter>    ::= A | B | ... | Z | a | b | ... | z
<ASCIICharacter> ::= || Every printable ASCII character in range [ ' ', '~' ]

```

```
<comment>     :: ; { <ASCIICharacter> }*
```

```
=====
=====
```

```
STM Instruction Set Architecture (ISA)
```

```
=====
=====
```

```
OpCode Assembly Syntax      Instruction Format  Semantics (meaning at run-time)
```

0X00	NOOP		OpCode	Do nothing
0X01	PUSH	memory	OpCode:mode:016	Push word memory[EA] on run-time stack
0X02	PUSHA	memory	OpCode:mode:016	Push EA on run-time stack
0X03	POP	memory	OpCode:mode:016	Pop run-time stack and store word in memory[EA]
0X04	DISCARD #W16		OpCode:016	Discard 016U words from top of run-time stack
0X05	SWAP		OpCode	Pop RHS,LHS; push RHS,LHS
0X06	MAKEDUP		OpCode	Read TOS; push TOS (duplicate TOS)
0X07	PUSHSP		OpCode	Push SP
0X08	PUSHFB		OpCode	Push FB
0X09	PUSHSB		OpCode	Push SB
0X0A	POSPS		OpCode	Pop SP
0X0B	POPFB		OpCode	Pop FB
0X0C	POPSB		OpCode	Pop SB
0X0D	SETAAE	memory	OpCode:mode:016	Pop key,value; add (key,value) to array in memory[EA] (Note 5)
0X0E	GETAAE	memory	OpCode:mode:016	Pop key; find (key,value) in array in memory[EA]; push value (Note 6)
0X0F	ADRAAE	memory	OpCode:mode:016	Pop key; find (key,value) in array in memory[EA]; push address of value (Note 7)
0X10	COPYAA		OpCode	Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in [0,2*capacity+1] (Note 9)
0X11	SETSE	memory	OpCode:mode:016	Pop character,index; store character in memory[EA+4+2*(index-1)] (Note 8)
0X12	GETSE	memory	OpCode:mode:016	Pop index; push memory[EA+4+2*(index-1)] (Note 8)

0X13	ADRSE	memory	OpCode:mode:016	Pop index; push address (EA+4+2*(index-1)) (Note 8)
0X14	ADDSE	memory	OpCode:mode:016	Pop character; store character in memory[EA+4+2*length]; increment length (Note 8)
0X15	COPYS		OpCode	Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in [0, capacity+1] (Note 9)
0X1D	CONCATS		OpCode	Pop RES,RHS,LHS; memory[RES] = memory[LHS] concatenate memory[RHS]; push RES (Note 12)
0X16	SETAE	memory	OpCode:mode:016	Pop value,index(n),index(n-1),...,index(1); store value at offset in array (Note 10)
0X17	GETAE	memory	OpCode:mode:016	Pop index(n),index(n-1),...,index(1); push value found at offset in array (Note 10)
0X18	ADRAE	memory	OpCode:mode:016	Pop index(n),index(n-1),...,index(1); push address of value found at offset in array (Note 10)
0X19	COPYA		OpCode	Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in [0, 2*n+capacity] (Note 9)
0X1A	GETAN	memory	OpCode:mode:016	Push n (# of dimensions)
0X1B	GETALB	memory	OpCode:mode:016	Pop dimension # i; push lower-bound, LBi (Note 11)
0X1C	GETAUB	memory	OpCode:mode:016	Pop dimension # i; push upper-bound, UBi (Note 11)
0X20	ADDI		OpCode	Pop RHS,LHS; push integer (LHS+RHS)
0X21	ADDF		OpCode	Pop RHS,LHS; push float (LHS+RHS)
0X22	SUBI		OpCode	Pop RHS,LHS; push integer (LHS-RHS)
0X23	SUBF		OpCode	Pop RHS,LHS; push float (LHS-RHS)
0X24	MULI		OpCode	Pop RHS,LHS; push integer (LHS*RHS)
0X25	MULF		OpCode	Pop RHS,LHS; push float (LHS*RHS)

0X26	DIVI	OpCode	Pop RHS,LHS; push integer (LHS÷RHS)
0X27	DIVF	OpCode	Pop RHS,LHS; push float (LHS÷RHS)
0X28	REMI	OpCode	Pop RHS,LHS; push integer (LHS rem RHS)
0X29	POWI	OpCode	Pop RHS,LHS; push integer pow(LHS,RHS)
0X2A	POWF	OpCode	Pop RHS,LHS; push float pow(LHS,RHS)
0X2B	NEGI	OpCode	Pop RHS; push integer -RHS
0X2C	NEGFI	OpCode	Pop RHS; push float -RHS
0X2D	AND	OpCode	Pop RHS,LHS; push boolean (LHS and RHS)
0X2E	NAND	OpCode	Pop RHS,LHS; push boolean (LHS nand RHS)
0X2F	OR	OpCode	Pop RHS,LHS; push boolean (LHS or RHS)
0X30	NOR	OpCode	Pop RHS,LHS; push boolean (LHS nor RHS)
0X31	XOR	OpCode	Pop RHS,LHS; push boolean (LHS xor RHS)
0X32	NXOR	OpCode	Pop RHS,LHS; push boolean (LHS nxor RHS)
0X33	NOT	OpCode	Pop RHS; push boolean (not RHS)
0X34	BITAND	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-and RHS)
0X35	BITNAND	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-nand RHS)
0X36	BITOR	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-or RHS)
0X37	BITNOR	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-nor RHS)
0X38	BITXOR	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-xor RHS)
0X39	BITNXOR	OpCode	Pop RHS,LHS; push boolean (LHS bitwise-nxor RHS)
0X3A	BITNOT	OpCode	Pop RHS; push boolean (bitwise-not RHS)
0X3B	BITSL #W16	OpCode:016	Pop LHS; push (LHS shifted-left 016U bits)
0X3C	BITLSR #W16	OpCode:016	Pop LHS; push (LHS logically shifted-right 016U bits)
0X3D	BITASR #W16	OpCode:016	Pop LHS; push (LHS arithmetically shifted-right 016U bits)
0X60	CITOF	OpCode	Pop integer RHS; push float RHS (integer-to-float)

0X61	CFTOI	OpCode	Pop float RHS; push integer RHS (float-to-integer)
0X70	CMPI	OpCode	Pop RHS,LHS; set LEG in FLAGS based on (LHS ? RHS) (integer)
0X71	CMPF	OpCode	Pop RHS,LHS; set LEG in FLAGS based on (LHS ? RHS) (float)
0X72	SETNZPI	OpCode	Set NZP in FLAGS based on sign of TOS (integer)
0X73	SETNZPF	OpCode	Set NZP in FLAGS based on sign of TOS (float)
0X74	SETT	OpCode	Set T in FLAGS based on true/false value of TOS (boolean)
0X80	JMP A16	OpCode:016	PC <- 016U
0X81	JMPL A16	OpCode:016	if (L) PC <- 016U
0X82	JMPE A16	OpCode:016	if (E) PC <- 016U
0X83	JMPG A16	OpCode:016	if (G) PC <- 016U
0X84	JMPLE A16	OpCode:016	if (L or E) PC <- 016U
0X85	JMPNE A16	OpCode:016	if (L or G) PC <- 016U (JMPLG)
0X86	JMPGE A16	OpCode:016	if (G or E) PC <- 016U
0X87	JMPN A16	OpCode:016	if (N) PC <- 016U
0X88	JMPNN A16	OpCode:016	if (not N) PC <- 016U
0X89	JMPZ A16	OpCode:016	if (Z) PC <- 016U
0X8A	JMPNZ A16	OpCode:016	if (not Z) PC <- 016U
0X8B	JMPP A16	OpCode:016	if (P) PC <- 016U
0X8C	JMPNP A16	OpCode:016	if (not P) PC <- 016U
0X8D	JMPT A16	OpCode:016	if (T) PC <- 016U
0X8E	JMPNT A16	OpCode:016	if (not T) PC <- 016U (JMPF)
0XA0	CALL A16	OpCode:016	Push PC; PC <- 016U
0XA1	RETURN	OpCode	Pop PC

0xFF	SVC #W16	OpCode:016	Execute service request 016U (parameters are passed on run-time stack)
------	----------	------------	------------------------------------------------------------------------

Notes

Note 1: Opcode is an 8-bit code used to uniquely identify each STM machine instruction.

Note 2: mode is an 8-bit code used to specify a memory operand's addressing mode.

Note 3: 016 is the instruction's operand field and is interpreted as either a 16-bit two's complement integer, 016, or as a 16-bit unsigned integer, 016U. The interpretation used depends on the OpCode.

Note 4: EA is the effective address of a memory word. The EA is computed using 016U and the instruction's addressing mode.

Note 5: When (key,value) pair is found, the existing value is replaced with the value popped from run-time stack. When (key,value) pair is not found, the pair is stored in the next available (key,value) slot. A fatal run-time error occurs when (key,value) pair is not found and (size = capacity).

Note 6: A fatal run-time error occurs when (key,value) pair is not found.

Note 7: When (key,value) pair is not found, the pair is stored in the next available (key,value) slot. A fatal run-time error occurs when (key,value) pair is not found and (size = capacity).

Note 8: A STM string is composed of 2+capacity words of contiguous memory. Word 1 is the length of the string (the number of characters contained in the string); word 2 is the string's capacity; and words 3-to-(capacity+2) are reserved for the string's characters. When empty, length = 0, otherwise

$(1 \leq \text{length} \leq \text{capacity})$. A fatal error occurs when $(1 \leq \text{index} \leq \text{length})$ is not true for SETSE, GETSE, and ADRSE. A fatal error occurs when $(\text{length} = \text{capacity})$ when ADDSE begins execution.

Note 9: Assumes that memory block pointed to by LHS is large enough to accommodate structure stored in memory block pointed to by RHS.

Note 10: A fatal error occurs when the following equation is not true for SETAE, GETAE, and ADRAE.
 $((LB1 \leq \text{index1} \leq UB1) \text{ AND } (LB2 \leq \text{index2} \leq UB2) \text{ AND...AND } (LBn \leq \text{indexn} \leq UBn))$

Note 11: A fatal error occurs when dimension # i is not in [1,n].

Note 12: Assumes that memory block pointed to by RES is large enough to accommodate the concatenation of strings pointed to by LHS and RHS. A fatal error occurs when
 $(\text{length-of-LHS} + \text{length-of-RHS}) > \text{capacity-of-RES}$

*/

```
//-----
typedef enum
//-----
{
// Pseudo-terminals
    IDENTIFIER,
    INTEGER,
    FLOAT,
    CHARACTER,
    STRING,
```

```
EOPTOKEN,  
EOLTOKEN,  
UNKNOWN,  
// Punctuation  
POUND,  
ATSIGN,  
COLON,  
DOLLAR,  
ASTERISK,  
// Assembler mnemonics  
ORG,  
EQU,  
DW,  
DS,  
RW,  
// Boolean literals  
FALSE,  
TRUE,  
// Registers  
SP,  
FB,  
SB,  
// H/W mnemonics  
NOOP,  
PUSH,  
PUSHA,
```

POP,
DISCARD,
SWAP,
MAKEDUP,
PUSHSP,
PUSHFB,
PUSHSB,
POPSP,
POPFB,
POPSB,
SETAAE,
GETAAE,
ADRAAE,
COPYAA,
SETSE,
GETSE,
ADRSE,
ADDSE,
COPYS,
CONCATS,
SETAE,
GETAE,
ADRAE,
COPYA,
GETAN,
GETALB,

GETAUB,
ADDI,
ADDF,
SUBI,
SUBF,
MULTI,
MULF,
DIVI,
DIVF,
REMI,
POWI,
POWF,
NEGI,
NEGFI,
AND,
NAND,
OR,
NOR,
XOR,
NXOR,
NOT,
BITAND,
BITNAND,
BITOR,
BITNOR,
BITXOR,

BITNXOR,
BITNOT,
BITSL,
BITLSR,
BITASR,
CITO_F,
CFTOI,
CMPI,
CMPF,
SETNZPI,
SETNZPF,
SETT,
JMP,
JMPL,
JMPE,
JMPG,
JMPEL,
JMPNE,
JMPGE,
JMPN,
JMPNN,
JMPZ,
JMPNZ,
JMPP,
JMPNP,
JMPT,

```
JMPNT,  
CALL,  
RETURN,  
SVC  
} TOKENTYPE;  
  
typedef enum { NONE, MEMORY, A16, IMMW16 } OPERANDTYPE;  
  
//-----  
typedef struct  
//-----  
{  
    int opCode;  
    int sizeInBytes;  
    char mnemonic[8+1]; // *LOOK* increase size for longer-than-8-character mnemonics  
    TOKENTYPE token;  
    OPERANDTYPE operandType;  
} HWOPERATIONRECORD;  
  
//-----  
const HWOPERATIONRECORD HWOperationTable[] =  
//-----  
{  
    { 0X00,1,"NOOP" ,NOOP ,NONE },  
  
    { 0X01,4,"PUSH" ,PUSH ,MEMORY },
```

```
{ 0X02,4,"PUSHA" ,PUSHA ,MEMORY },  
{ 0X03,4,"POP" ,POP ,MEMORY },  
{ 0X04,3,"DISCARD" ,DISCARD ,IMMW16 },  
{ 0X05,1,"SWAP" ,SWAP ,NONE },  
{ 0X06,1,"MAKEDUP" ,MAKEDUP ,NONE },  
{ 0X07,1,"PUSHSP" ,PUSHSP ,NONE },  
{ 0X08,1,"PUSHFB" ,PUSHFB ,NONE },  
{ 0X09,1,"PUSHSB" ,PUSHSB ,NONE },  
{ 0X0A,1,"POPSP" ,POPSP ,NONE },  
{ 0X0B,1,"POPFB" ,POPFB ,NONE },  
{ 0X0C,1,"POPSB" ,POPSB ,NONE },  
{ 0X0D,4,"SETAAE" ,SETAAE ,MEMORY },  
{ 0X0E,4,"GETAAE" ,GETAAE ,MEMORY },  
{ 0X0F,4,"ADRAAE" ,ADRAAE ,MEMORY },  
{ 0X10,1,"COPYAA" ,COPYAA ,NONE },  
{ 0X11,4,"SETSE" ,SETSE ,MEMORY },  
{ 0X12,4,"GETSE" ,GETSE ,MEMORY },  
{ 0X13,4,"ADRSE" ,ADRSE ,MEMORY },  
{ 0X14,4,"ADDSE" ,ADDSE ,MEMORY },  
{ 0X15,1,"COPYS" ,COPYS ,NONE },  
{ 0X1D,1,"CONCATS" ,CONCATS ,NONE },  
{ 0X16,4,"SETAE" ,SETAE ,MEMORY },  
{ 0X17,4,"GETAE" ,GETAE ,MEMORY },  
{ 0X18,4,"ADRAE" ,ADRAE ,MEMORY },  
{ 0X19,1,"COPYA" ,COPYA ,NONE },  
{ 0X1A,4,"GETAN" ,GETAN ,MEMORY },
```

```
{ 0X1B,4,"GETALB" ,GETALB ,MEMORY  },
{ 0X1C,4,"GETAUB" ,GETAUB ,MEMORY  },

{ 0X20,1,"ADDI"   ,ADDI   ,NONE  },
{ 0X21,1,"ADDF"   ,ADDF   ,NONE  },
{ 0X22,1,"SUBI"   ,SUBI   ,NONE  },
{ 0X23,1,"SUBF"   ,SUBF   ,NONE  },
{ 0X24,1,"MULI"   ,MULI   ,NONE  },
{ 0X25,1,"MULF"   ,MULF   ,NONE  },
{ 0X26,1,"DIVI"   ,DIVI   ,NONE  },
{ 0X27,1,"DIVF"   ,DIVF   ,NONE  },
{ 0X28,1,"REMI"   ,REMI   ,NONE  },
{ 0X29,1,"POWI"   ,POWI   ,NONE  },
{ 0X2A,1,"POWF"   ,POWF   ,NONE  },
{ 0X2B,1,"NEGI"   ,NEGI   ,NONE  },
{ 0X2C,1,"NEGF"   ,NEGF   ,NONE  },
{ 0X2D,1,"AND"    ,AND    ,NONE  },
{ 0X2E,1,"NAND"   ,NAND   ,NONE  },
{ 0X2F,1,"OR"     ,OR     ,NONE  },
{ 0X30,1,"NOR"    ,NOR    ,NONE  },
{ 0X31,1,"XOR"    ,XOR    ,NONE  },
{ 0X32,1,"NXOR"   ,NXOR   ,NONE  },
{ 0X33,1,"NOT"    ,NOT    ,NONE  },
{ 0X34,1,"BITAND" ,BITAND ,NONE  },
{ 0X35,1,"BITNAND" ,BITNAND ,NONE  },
{ 0X36,1,"BITOR"   ,BITOR   ,NONE  },
```

```
{ 0X37,1,"BITNOR" ,BITNOR ,NONE   },
{ 0X38,1,"BITXOR" ,BITXOR ,NONE   },
{ 0X39,1,"BITNXOR" ,BITNXOR ,NONE   },
{ 0X3A,1,"BITNOT" ,BITNOT ,NONE   },
{ 0X3B,3,"BITSL"   ,BITSL  ,IMMW16 },
{ 0X3C,3,"BITLSR"  ,BITLSR ,IMMW16 },
{ 0X3D,3,"BITASR"  ,BITASR ,IMMW16 },

{ 0X60,1,"CITOF"   ,CITOF  ,NONE   },
{ 0X61,1,"CFTOI"   ,CFTOI  ,NONE   },

{ 0X70,1,"CMPI"    ,CMPI   ,NONE   },
{ 0X71,1,"CMPF"    ,CMPF   ,NONE   },
{ 0X72,1,"SETNZPI" ,SETNZPI ,NONE   },
{ 0X73,1,"SETNZPF" ,SETNZPF ,NONE   },
{ 0X74,1,"SETT"    ,SETT   ,NONE   },

{ 0X80,3,"JMP"     ,JMP    ,A16   },
{ 0X81,3,"JMPL"    ,JMPL   ,A16   },
{ 0X82,3,"JMPE"    ,JMPE   ,A16   },
{ 0X83,3,"JMPG"    ,JMPG   ,A16   },
{ 0X84,3,"JMPLE"   ,JMPLE  ,A16   },
{ 0X85,3,"JMPNE"   ,JMPNE  ,A16   },
{ 0X86,3,"JMPGE"   ,JMPGE  ,A16   },
{ 0X87,3,"JMPN"    ,JMPN   ,A16   },
{ 0X88,3,"JMPNN"   ,JMPNN  ,A16 }
```

```
{ 0X89,3,"JMPZ" ,JMPZ ,A16   },
{ 0X8A,3,"JMPNZ" ,JMPNZ ,A16   },
{ 0X8B,3,"JMPP"  ,JMPP ,A16   },
{ 0X8C,3,"JMPNP" ,JMPNP ,A16   },
{ 0X8D,3,"JMPT"  ,JMPT ,A16   },
{ 0X8E,3,"JMPNT" ,JMPNT ,A16   },

{ 0XA0,3,"CALL"  ,CALL  ,A16   },
{ 0XA1,1,"RETURN" ,RETURN ,NONE  },

{ 0xFF,3,"SVC"   ,SVC   ,IMMW16 }

};

//-----
// GLOBAL VARIABLES
//-----

FILE *SOURCE,*LOG;

char sourceFileName[SOURCELINELENGTH+1];
char sourceLine[SOURCELINELENGTH+1],nextCharacter;
int sourceLineIndex;
bool atEOP;

// ***** identifierTable
typedef struct
{
    int value;
```

```
int numberOfDefinitions;
char identifier[MAXIMUMLENGTHIDENTIFIER+1];
} IDENTIFIERTABLERECORD;

int sizeOfIdentifierTable;
IDENTIFIERTABLERECORD identifierTable[SIZEOFIDENTIFIERTABLE+1];

// ***** mainMemory
BYTE mainMemory[0xFFFF+1];

// ***** syntaxErrors (up to 10 for each source line)
int numberOfSyntaxErrors;
char syntaxErrors[10][80+1];

//-----
void ProcessRunTimeError(const char error[],bool isFatalError)
//-----
{
    fprintf(LOG,"Run-time error %s\n",error); fflush(LOG);
    printf("Run-time error %s\n",error);
    if ( isFatalError )
    {
        fclose(LOG);
        system("PAUSE");
        exit(1);
    }
}
```

```
}

}

//-----
int main()
//-----
{
    void DoPass1();
    void DoPass2(bool *noSyntaxErrors);
    void ExecuteProgram();

    char fullFileName[SOURCELINELENGTH+1];
    bool noSyntaxErrors;

    printf("Version %s\n\n",VERSION);

    printf("Source filename? "); scanf("%s",sourceFileName);
    strcpy(fullFileName,sourceFileName);
    strcat(fullFileName,".stm");
    if ( (SOURCE = fopen(fullFileName,"r")) == NULL )
    {
        printf("Error opening source file %s\n",fullFileName);
        system("PAUSE");
        exit( 1 );
    }
}
```

```
strcpy(fullFileName,sourceFileName);
strcat(fullFileName,".log");
if ( (LOG = fopen(fullFileName,"w")) == NULL )
{
    printf("Error opening log file %s\n",fullFileName);
    system("PAUSE");
    exit( 1 );
}

printf("Log file is %s\n",fullFileName);

/*
Assemble source program then, if no syntax errors are discovered
during translation execute resulting machine program
*/
DoPass1();
DoPass2(&noSyntaxErrors);
fclose(SOURCE);

if ( noSyntaxErrors )
    ExecuteProgram();
else
    printf("Source file contains syntax errors\n");

fclose(LOG);
system("PAUSE");
```

```
    return( 0 );
}

//-----
void DoPass1()
//-----
{

    void GetNextToken(TOKENTYPE *token,char lexeme[]);
    void GetNextCharacter();
    void ReadSourceLine();
    void DefineIdentifierInTable(const char lexeme[],int value);
    bool IdentifierIsInTable(const char lexeme[]);
    int FindIdentifierInTable(const char lexeme[]);
    int ATOI16(const char lexeme[]);
    WORD ATOF16(const char lexeme[]);

    TOKENTYPE token;
    char lexeme[SOURCELINELENGTH+1];
    int LC;
    bool defineLineLabel;
    char labelLexeme[SOURCELINELENGTH+1];
    int labelValue;

    sizeOfIdentifierTable = 0;
    LC = 0X0000;
```

```
/*
Each statement *MUST BE* wholly contained on a single source line.
Read through source file line-by-line to build identifierTable.
Ignore *ALL* syntax errors because they will be "caught by" pass #2.

*/
ReadSourceLine();
while ( !atEOP )
{
    GetNextCharacter();
    GetNextToken(&token,lexeme);
    if ( token != EOLTOKEN )
    {
        if ( token == IDENTIFIER )
        {
            defineLineLabel = true;
            strcpy(labelLexeme,lexeme);
            labelValue = LC;      // default, with labelValue changed by ORG and EQU instructions
            GetNextToken(&token,lexeme);
        }
        else
            defineLineLabel = false;
        switch ( token )
        {
            case ORG:
                GetNextToken(&token,lexeme);
                if ( token == INTEGER )
```

```
labelValue = LC = ATOI16(lexeme);

else          // *ERROR*
    labelValue = LC = 0X0000;
break;

case EQU:
    GetNextToken(&token,lexeme);
    switch ( token )
    {
        case INTEGER:
            labelValue = ATOI16(lexeme);
            break;
        case FLOAT:
            labelValue = ATOF16(lexeme);
            break;
        case TRUE:
            labelValue = 0xFFFFu;
            break;
        case FALSE:
            labelValue = 0X0000u;
            break;
        case CHARACTER:
            labelValue = (WORD) lexeme[0];
            break;
        case IDENTIFIER:
            if ( IdentifierIsInTable(lexeme) )
                labelValue = identifierTable[FindIdentifierInTable(lexeme)].value;
    }
}
```

```
        else
            defineLineLabel = false;
        break;

    case ASTERISK:
        labelValue = LC;
        break;

    default:
        defineLineLabel = false;
        break;
    }

    break;

case RW:
    GetNextToken(&token,lexeme);
    if ( token == INTEGER )
        LC += 2*ATOI16(lexeme);
    else           // *ERROR* unless token is EOLTOKEN
        LC += 2;
    break;

case DW:          // *always* only one word of object
    LC += 2;
    break;

case DS:          // 2-byte UNICODE characters (prepend length and capacity)
    GetNextToken(&token,lexeme);
    if ( token == STRING )
        LC += 2*((int) strlen(lexeme)+2);
    else           // *ERROR*
```

```
    LC += 2;

    break;

default:           // should be a hardware operation token

{

    bool found = false;

    int i = 0;

    while ( (i <= (sizeof(HWOperationTable)/sizeof(HWOPERATIONRECORD))-1) && !found )

        if ( token != HWOperationTable[i].token )

            i++;

        else

    {

        LC += HWOperationTable[i].sizeInBytes;

        found = true;

    }

    if ( !found ) // *ERROR*

        LC += 1;

    }

    break;

}

/*
When first token is an identifier add its lexeme and its value
to the identifierTable if it's not already there.

*/
if ( defineLineLabel )

    DefineIdentifierInTable(labelLexeme,labelValue);
```

```
}

// Ignore remainder of source line
ReadSourceLine();

}

}

//-----

void DoPass2(bool *noSyntaxErrors)
//-----

{

void GetNextToken(TOKENTYPE *token,char lexeme[]);
void GetNextCharacter();
void ReadSourceLine();
void ListTopOfPageHeader(int *pageNumber,int *lines);
void RecordSyntaxError(const char syntaxError[]);
int ATOI16(const char lexeme[]);
WORD ParseW16(TOKENTYPE *token,char lexeme[]);
WORD ParseA16(TOKENTYPE *token,char lexeme[]);
bool IdentifierIsInTable(const char lexeme[]);
int FindIdentifierInTable(const char lexeme[]);
void WriteBYTEToMainMemory(int address,BYTE byte);

TOKENTYPE token;
char lexeme[SOURCELINELENGTH+1];
bool lineIsLabeled;
BYTE objectCode[256+1];
```

```
int objectBytes;

int oldLC,LC,lineNumber,linesOnPage,pageNumber;
int i,j;

*noSyntaxErrors = true;
LC = 0X0000;
lineNumber = 0;
pageNumber = 0;
ListTopOfPageHeader(&pageNumber,&linesOnPage);

/*
   Each statement *MUST BE* wholly contained on a single source line.
   Read through source file line-by-line to assemble into object code.

*/
rewind(SOURCE);
atEOP = false;
ReadSourceLine(); lineNumber++;
while ( !atEOP )
{
    oldLC = LC;
    numberOfSyntaxErrors = 0;
    objectBytes = 0;
    GetNextCharacter();
    GetNextToken(&token,lexeme);
    if ( token != EOLTOKEN )
    {
        // Is first token an identifier? If so, is it multiply defined?
```

```
if ( token == IDENTIFIER )
{
    int index = FindIdentifierInTable(lexeme);

    if ( identifierTable[index].numberOfDefinitions != 1 )
        RecordSyntaxError("Multiply-defined identifier");
    GetNextToken(&token,lexeme);
    lineIsLabeled = true;
}

else
    lineIsLabeled = false;

switch ( token )
{
    case IDENTIFIER: // assume misspelled hardware mnemonic, so insert a NOOP
        LC++;
        objectCode[1] = 0X00u; objectBytes = 1;
        RecordSyntaxError("Misplaced identifier");
        GetNextToken(&token,lexeme);
        break;

    case UNKNOWN: // does not affect LC
        objectBytes = 0;
        RecordSyntaxError("Unknown token");
        GetNextToken(&token,lexeme);
        break;

/*
<instruction> ::= [ <identifier> ] ORG          <I16>
```

```
|   <identifier> EQU      (( <W16> | * ))
| [ <identifier> ] RW      [ <I16> ]
| [ <identifier> ] DW      <W16>
| [ <identifier> ] DS      <string>

*/
case ORG:
    objectBytes = 0;
    GetNextToken(&token,lexeme);
    if ( token == INTEGER )
        oldLC = LC = ATOI16(lexeme);
    else
    {
        RecordSyntaxError("ORG statement <I16> operand missing");
        oldLC = LC = 0X0000;
    }
    GetNextToken(&token,lexeme);
    break;
case EQU:
    if ( !lineIsLabeled )
        RecordSyntaxError("EQU statement must be labeled");
    objectBytes = 0;
    GetNextToken(&token,lexeme);
    if ( token == ASTERISK )
        GetNextToken(&token,lexeme);
    else
        ParseW16(&token,lexeme);
```

```
break;

case RW:
    GetNextToken(&token,lexeme);
    objectBytes = 0;
    if      ( token == INTEGER )
    {
        LC += 2*ATOI16(lexeme);
        GetNextToken(&token,lexeme);
    }
    else if ( token == EOLTOKEN )
        LC += 2;
    else
    {
        RecordSyntaxError("Invalid RW statement <I16> operand");
        LC += 2;
        GetNextToken(&token,lexeme);
    }
    break;

case DW:           // one word of object
{
    WORD W16;

    objectBytes = 2;
    GetNextToken(&token,lexeme);
    W16 = ParseW16(&token,lexeme);
    objectCode[1] = HIBYTE(W16);
```

```
    objectCode[2] = LOBYTE(W16);
}
break;
case DS:
    GetNextToken(&token,lexeme);
    if ( token == STRING )
    {
        objectBytes = 0;
        // length
        objectCode[++objectBytes] = HIBYTE((WORD) strlen(lexeme));
        objectCode[++objectBytes] = LOBYTE((WORD) strlen(lexeme));
        // capacity
        objectCode[++objectBytes] = HIBYTE((WORD) strlen(lexeme));
        objectCode[++objectBytes] = LOBYTE((WORD) strlen(lexeme));
        for ( i = 0; i <= (int) strlen(lexeme)-1; i++ )
        {
            objectCode[++objectBytes] = 0X00u;
            objectCode[++objectBytes] = (BYTE) lexeme[i];
        }
    }
    else
    {
        objectBytes = 2;
        RecordSyntaxError("DS statement operand must be string");
        objectCode[1] = 0X00u;
        objectCode[2] = 0X00u;
```

```
}

GetNextToken(&token,lexeme);

break;

/*
<instruction>    ::= [ <identifier> ] <HWMnemonic> [ <operand> ]

<operand>        ::= <memory>

                   | #<W16>

                   | <A16>

<memory>         ::= #<W16>           || mode = 0X00, immediate
                   | <A16>            || mode = 0X01, memory direct
                   | @<A16>          || mode = 0X02, memory indirect
                   | $<A16>          || mode = 0X03, memory indexed
                   | SP:<I16>         || mode = 0X04, SP-relative direct
                   | @SP:<I16>        || mode = 0X05, SP-relative indirect
                   | $SP:<I16>        || mode = 0X06, SP-relative indexed
                   | FB:<I16>          || mode = 0X07, FB-relative direct
                   | @FB:<I16>         || mode = 0X08, FB-relative indirect
                   | $FB:<I16>         || mode = 0X09, FB-relative indexed
                   | SB:<I16>          || mode = 0X0A, SB-relative direct
                   | @SB:<I16>         || mode = 0X0B, SB-relative indirect
                   | $SB:<I16>         || mode = 0X0C, SB-relative indexed

<W16>           ::= (( <I16> | <F16> | true | false | <character> | <identifier> ))
```

```
<A16>      ::= <identifier>

<identifier>  ::= <letter> { (( <letter> | <dit> | _ )) }*
*/
default:
{
// find hardware token in HWOperationTable
TOKENTYPE operation = token;
bool found = false;
int i = 0;

while ( (i <= (sizeof(HWOperationTable)/sizeof(HWOPERATIONRECORD))-1) && !found )
if ( operation != HWOperationTable[i].token )
    i++;
else
    found = true;
if ( !found )
{
    RecordSyntaxError("Invalid hardware mnemonic");
    LC += 1;
}
else
{
    objectCode[1] = HWOperationTable[i].opCode;
    switch ( HWOperationTable[i].operandType )
{
```

```
case NONE:
    objectBytes = 1;
    GetNextToken(&token,lexeme);
    break;

case MEMORY:
    GetNextToken(&token,lexeme);
    switch ( token )
    {
        case POUND:
            {
                WORD W16;

                if ( (operation == POP) || (operation == PUSHA) )
                    RecordSyntaxError("POP and PUSHA cannot have an immediate operand");
                objectCode[2] = 0;
                GetNextToken(&token,lexeme);
                W16 = ParseW16(&token,lexeme);
                objectCode[3] = HIBYTE(W16);
                objectCode[4] = LOBYTE(W16);
            }
            break;

        case IDENTIFIER:
            {
                WORD A16;

                objectCode[2] = 1;
```

```
A16 = ParseA16(&token,lexeme);

objectCode[3] = HIBYTE(A16);

objectCode[4] = LOBYTE(A16);

}

break;

case ATSIGN:

{

WORD A16;

GetNextToken(&token,lexeme);

if ( token == IDENTIFIER )

{

objectCode[2] = 2;

A16 = ParseA16(&token,lexeme);

objectCode[3] = HIBYTE(A16);

objectCode[4] = LOBYTE(A16);

}

else

{

TOKENTYPE R = token;

if ( !((R == SP) || (R == FB) || (R == SB)) )

{

RecordSyntaxError("Expecting register SP, FB, or SB");

R = SP;

}
```

```
GetNextToken(&token,lexeme);

if ( token != COLON )
    RecordSyntaxError("Expecting :");

GetNextToken(&token,lexeme);

if ( token != INTEGER )
{
    RecordSyntaxError("Expecting <I16>");

    objectCode[2] = 5;
    objectCode[3] = 0X00u;
    objectCode[4] = 0X00u;

}
else
{
    switch ( R )
    {

        case SP:
            objectCode[2] = 5;
            break;

        case FB:
            objectCode[2] = 8;
            break;

        case SB:
            objectCode[2] = 11;
            break;
    }
    objectCode[3] = HIBYTE(ATOI16(lexeme));
}
```

```
    objectCode[4] = LOBYTE(ATOI16(lexeme));

}

GetNextToken(&token,lexeme);

}

break;

case DOLLAR:

{

    GetNextToken(&token,lexeme);

    if ( token == IDENTIFIER )

    {

        WORD A16;

        objectCode[2] = 3;

        A16 = ParseA16(&token,lexeme);

        objectCode[3] = HIBYTE(A16);

        objectCode[4] = LOBYTE(A16);

    }

    else

    {

        TOKENTYPE R = token;

        if ( !((R == SP) || (R == FB) || (R == SB)) )

        {

            RecordSyntaxError("Expecting register SP, FB, or SB");

            R = SP;

        }

    }

}
```

```
}

GetNextToken(&token,lexeme);

if ( token != COLON )

    RecordSyntaxError("Expecting :");

GetNextToken(&token,lexeme);

if ( token != INTEGER )

{

    RecordSyntaxError("Expecting <I16>");

    objectCode[2] =  6;

    objectCode[3] = 0X00u;

    objectCode[4] = 0X00u;

}

else

{

    switch ( R )

    {

        case SP:

            objectCode[2] =  6;

            break;

        case FB:

            objectCode[2] =  9;

            break;

        case SB:

            objectCode[2] = 12;

            break;

    }

}
```

```
    objectCode[3] = HIBYTE(ATOI16(lexeme));
    objectCode[4] = LOBYTE(ATOI16(lexeme));
}
GetNextToken(&token,lexeme);
}
break;
case SP:
case FB:
case SB:
{
    TOKENTYPE R = token;

    GetNextToken(&token,lexeme);
    if ( token != COLON )
        RecordSyntaxError("Expecting :");
    GetNextToken(&token,lexeme);
    if ( token != INTEGER )
    {
        RecordSyntaxError("Expecting <I16>");
        objectCode[3] = 0X00u;
        objectCode[4] = 0X00u;
    }
    else
    {
        objectCode[3] = HIBYTE(ATOI16(lexeme));
```

```
    objectCode[4] = LOBYTE(ATOI16(lexeme));

}

switch ( R )

{

    case SP:

        objectCode[2] =  4;

        break;

    case FB:

        objectCode[2] =  7;

        break;

    case SB:

        objectCode[2] = 10;

        break;

    }

}

GetNextToken(&token,lexeme);

break;

default:

    RecordSyntaxError("Invalid <memory> operand");

    objectCode[2] =  0;

    objectCode[3] = 0X00u;

    objectCode[4] = 0X00u;

    GetNextToken(&token,lexeme);

    break;

}

objectBytes = 4;
```

```
        break;

case A16:
    GetNextToken(&token,lexeme);
    if ( token != IDENTIFIER )
    {
        RecordSyntaxError("Expecting <identifier>");
        objectCode[2] = 0X00u;
        objectCode[3] = 0X00u;
        GetNextToken(&token,lexeme);
    }
    else
    {
        WORD A16;

        A16 = ParseA16(&token,lexeme);
        objectCode[2] = HIBYTE(A16);
        objectCode[3] = LOBYTE(A16);
    }
    objectBytes = 3;
    break;

case IMMW16:
    GetNextToken(&token,lexeme);
    if ( token != POUND )
    {
        RecordSyntaxError("Expecting #");
        objectCode[2] = 0X00u;
```

```
    objectCode[3] = 0X00u;
    GetNextToken(&token,lexeme);
}
else
{
    WORD W16;

    GetNextToken(&token,lexeme);
    W16 = ParseW16(&token,lexeme);
    objectCode[2] = HIBYTE(W16);
    objectCode[3] = LOBYTE(W16);
}
objectBytes = 3;
break;
}
}
}
break;
}
}

if ( token != EOLTOKEN )
    RecordSyntaxError("Expecting end-of-line");
/*
LC Object     Line  Source Line
-----
0XXXXX  XXXXXXXX  XXXX  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
**** XXX...XXX

*/
if ( linesOnPage > LINESPERPAGE )
    ListTopOfPageHeader(&pageNumber,&linesOnPage);
fprintf(LOG,"0X%04hX ",oldLC);
for (j = 1; j <= 4; j++)
{
    if ( j <= objectBytes )
        fprintf(LOG,"%02X",objectCode[j]);
    else
        fprintf(LOG,"  ");

}

fprintf(LOG," %4d %s\n",lineNumber,sourceLine); fflush(LOG);
linesOnPage++;
i = 5;
while ( i <= objectBytes )
{
    if ( linesOnPage > LINESPERPAGE )
        ListTopOfPageHeader(&pageNumber,&linesOnPage);
    fprintf(LOG,"0X%04hX ",(oldLC+i-1));
    for (j = 1; j <= 4; j++)
    {
        if ( j+i-1 <= objectBytes )
            fprintf(LOG,"%02X",objectCode[j+i-1]);
        else
```

```
    fprintf(LOG,"  ");
}

fprintf(LOG,"\n"); fflush(LOG);

linesOnPage++;
i = i+4;
}

LC = LC+objectBytes;

if ( LC-1 > 0xFFFF )
{
    RecordSyntaxError("Location counter overflow");
    LC = 0X0000;
}

for (i = 1; i <= numberOfSyntaxErrors; i++)
{
    *noSyntaxErrors = false;
    if ( linesOnPage > LINESPERPAGE )
        ListTopOfPageHeader(&pageNumber,&linesOnPage);
    fprintf(LOG, "***** %s\n",syntaxErrors[i]); fflush(LOG);
    linesOnPage++;
    printf("Error on line %4d %s\n",lineNumber,syntaxErrors[i]);
}

for (i = 1; i <= objectBytes; i++)
    WriteBYTEToMainMemory(oldLC+i-1,objectCode[i]);

ReadSourceLine(); lineNumber++;

}
```

```
//-----
WORD ParseW16(TOKENTYPE *token,char lexeme[])
//-----
{
    WORD ParseA16(TOKENTYPE *token,char lexeme[]);
    void GetNextToken(TOKENTYPE *token,char lexeme[]);
    int ATOI16(const char lexeme[]);
    WORD ATOF16(const char lexeme[]);
    void RecordSyntaxError(const char syntaxError[]);

    WORD W16;

    switch ( *token )
    {
        case INTEGER:
            W16 = ATOI16(lexeme);
            GetNextToken(token,lexeme);
            break;
        case FLOAT:
            W16 = ATOF16(lexeme);
            GetNextToken(token,lexeme);
            break;
        case TRUE:
            W16 = 0xFFFFu;
            GetNextToken(token,lexeme);
    }
}
```

```
        break;

    case FALSE:
        W16 = 0X0000u;
        GetNextToken(token,lexeme);
        break;

    case CHARACTER:
        W16 = (WORD) lexeme[0];
        GetNextToken(token,lexeme);
        break;

    case IDENTIFIER:
        W16 = ParseA16(token,lexeme);
        break;

    default:
        RecordSyntaxError("Expecting <I16>, <F16>, true, false, <character>, or <identifier>");
        W16 = 0X0000u;
        GetNextToken(token,lexeme);
        break;
    }

    return( W16 );
}

//-----
WORD ParseA16(TOKENTYPE *token,char lexeme[])
//-----
{
    void GetNextToken(TOKENTYPE *token,char lexeme[]);
```

```
bool IdentifierIsInTable(const char lexeme[]);

int FindIdentifierInTable(const char lexeme[]);

void RecordSyntaxError(const char syntaxError[]);

WORD A16;

if ( IdentifierIsInTable(lexeme) )

    A16 = (WORD) identifierTable[FindIdentifierInTable(lexeme)].value;

else

{

    RecordSyntaxError("Undefined <identifier>");

    A16 = 0X0000u;

}

GetNextToken(token,lexeme);

return( A16 );

}

//-----

bool IsValidDigit(char digit,char base)

//-----

{

    bool r;

    switch ( base )

    {

        case 'B':
```

```

r = ( (digit == '0') || (digit == '1') );
break;
case 'D':
    r = isdigit(digit);
break;
case 'X':
    r = isxdigit(digit);
break;
}
return( r );
}

//-----
int ATOI16(const char lexeme[])
//-----
{
/*
Safely assume lexeme "obeys" the following syntax rule because the scanner
recognized TOKENTYPE as INTEGER

<I16>      ::= [ (( + | - )) ] (( 0D <dit> { <dit> }*
|                  0X <hit> { <hit> }*
|                  0B <bit> { <bit> }* ))
*/
char sign,base;
int i,digit,I16;

```

```
i = 0;
sign = '+';

if ( (lexeme[i] == '+') || (lexeme[i] == '-') )
{
    sign = lexeme[i];
    i++;
}

i++; base = lexeme[i]; i++;
I16 = 0;

while ( i <= (int) strlen(lexeme)-1 )

{
    if ( lexeme[i] == '0' ) digit = 0;
    if ( lexeme[i] == '1' ) digit = 1;
    if ( lexeme[i] == '2' ) digit = 2;
    if ( lexeme[i] == '3' ) digit = 3;
    if ( lexeme[i] == '4' ) digit = 4;
    if ( lexeme[i] == '5' ) digit = 5;
    if ( lexeme[i] == '6' ) digit = 6;
    if ( lexeme[i] == '7' ) digit = 7;
    if ( lexeme[i] == '8' ) digit = 8;
    if ( lexeme[i] == '9' ) digit = 9;
    if ( lexeme[i] == 'A' ) digit = 10;
    if ( lexeme[i] == 'B' ) digit = 11;
    if ( lexeme[i] == 'C' ) digit = 12;
    if ( lexeme[i] == 'D' ) digit = 13;
```

```
if ( lexeme[i] == 'E' ) digit = 14;
if ( lexeme[i] == 'F' ) digit = 15;

switch ( base )
{
    case 'B':
        I16 = I16* 2 + digit;
        break;
    case 'D':
        I16 = I16*10 + digit;
        break;
    case 'X':
        I16 = I16*16 + digit;
        break;
}
i++;
}

return( (sign == '-')? -I16 : I16 );
}

//-----
WORD ATOF16(const char lexeme[])
//-----
{
/*
Safely assume lexeme "obeys" the following syntax rule because the scanner
```

```
recognized TOKENTYPE as FLOAT

<F16>      ::= [ (( + | - )) ] 0F <dit> { <dit> }* . <dit> { <dit> }* [ E [ - ] <dit> { <dit> }* ]

*/
void ConvertFloatToHalfFloat(float F,WORD *HF);

char sign;
WORD F16;
float item;
int i;

i = 0;
sign = '+';
if ( (lexeme[i] == '+') || (lexeme[i] == '-') )
{
    sign = lexeme[i];
    i++;
}
i += 2;
sscanf(&lexeme[i],"%f",&item);
item = ((sign == '-') ? -item : item);
ConvertFloatToHalfFloat(item,&F16);
return( F16 );
}

//-----
```

```
void GetNextToken(TOKENTYPE *token, char lexeme[])
//-----
{
    void GetNextCharacter();
    bool IsValidDigit(char digit, char base);
    void RecordSyntaxError(const char syntaxError[]);

    int i;

// "Eat" blanks and tabs (if any) at beginning-of-line
    while ( (nextCharacter == ' ') || (nextCharacter == '\t') )
        GetNextCharacter();

/*
"Eat" comments (if any). Comments are always extend to end-of-line, but *DO NOT* include EOLC.
<comment>      :: ; { <ASCIICharacter> }*
*/
    if ( nextCharacter == ';' )
    {
        do
            GetNextCharacter();
        while ( nextCharacter != EOLC );
    }
/*
If an identifier-like lexeme is not an assembler mnemonic, a hardware mnemonic,
a boolean literal, or a register, then it is--by default--an identifier
```

```
<identifier>      ::= <letter> { (( <letter> | <dit> | _ )) }*
```

```
*/
```

```
if ( isalpha(nextCharacter) )
```

```
{
```

```
    char UClexeme[SOURCELINELENGTH+1];
```

```
    i = 0;
```

```
    lexeme[i++] = nextCharacter;
```

```
    GetNextCharacter();
```

```
    while ( isalpha(nextCharacter) ||
```

```
           isdigit(nextCharacter) ||
```

```
           (nextCharacter == '_') )
```

```
    {
```

```
        lexeme[i++] = nextCharacter;
```

```
        GetNextCharacter();
```

```
    }
```

```
    lexeme[i] = '\0';
```

```
    *token = IDENTIFIER;
```

```
    for (i = 0; i <= (int) strlen(lexeme); i++)
```

```
        UClexeme[i] = toupper(lexeme[i]);
```

```
    if      ( strcmp(UClexeme,"ORG"     ) == 0 )
```

```
        *token = ORG;
```

```
    else if ( strcmp(UClexeme,"EQU"     ) == 0 )
```

```
        *token = EQU;
```

```
    else if ( strcmp(UClexeme,"DW"      ) == 0 )
```

```

*token = DW;
else if ( strcmp(UClexeme,"DS"      ) == 0 )
*token = DS;
else if ( strcmp(UClexeme,"RW"      ) == 0 )
*token = RW;
else if ( strcmp(UClexeme,"TRUE"    ) == 0 )
*token = TRUE;
else if ( strcmp(UClexeme,"FALSE"   ) == 0 )
*token = FALSE;
else if ( strcmp(UClexeme,"SP"      ) == 0 )
*token = SP;
else if ( strcmp(UClexeme,"FB"      ) == 0 )
*token = FB;
else if ( strcmp(UClexeme,"SB"      ) == 0 )
*token = SB;
else
{
    for (i = 0; i <= (sizeof(HWOperationTable)/sizeof(HWOPERATIONRECORD))-1; i++)
        if ( strcmp(UClexeme,HWOperationTable[i].mnemonic) == 0 )
            *token = HWOperationTable[i].token;
}
/*
<I16>      ::= [ (( + | - )) ] (( 0D <dit> { <dit> }*
|                  0X <hit> { <hit> }*
|                  0B <bit> { <bit> }* ))

```

```
<F16>      ::= [ (( + | - )) ] 0F <dit> { <dit> }* . <dit> { <dit> }* [ E [ - ] <dit> { <dit> }* ]
<bit>       ::= 0 | 1
<dit>       ::= <bit> | 2 | 3| 4| 5 | 6 | 7 | 8 | 9
<hit>       ::= <dit> | A | B | C | D | E | F

*/
else if ( (nextCharacter == '0')
    || (nextCharacter == '+')
    || (nextCharacter == '-') )
{
    i = 0;
    if ( (nextCharacter == '+') || (nextCharacter == '-') )
    {
        lexeme[i++] = nextCharacter;
        GetNextCharacter();
    }
    if ( nextCharacter != '0' )
    {
        RecordSyntaxError("Invalid <I16> or <F16> prefix\n");
        *token = INTEGER;
        strcpy(lexeme,"0D0");
    }
    else
    {
        lexeme[i++] = nextCharacter;
        GetNextCharacter();
        if      ( (toupper(nextCharacter) == 'B')
```

```
    || (toupper(nextCharacter) == 'D')
    || (toupper(nextCharacter) == 'X') )

{
    char base = toupper(nextCharacter);

    lexeme[i++] = base;
    GetNextCharacter();
    while ( IsValidDigit(nextCharacter,base) )
    {
        lexeme[i++] = toupper(nextCharacter);
        GetNextCharacter();
    }
    *token = INTEGER;
    lexeme[i] = '\0';
}

else if ( toupper(nextCharacter) == 'F' )
{
    lexeme[i++] = nextCharacter;
    GetNextCharacter();
    if ( !isdigit(nextCharacter) )
    {
        RecordSyntaxError("Invalid <F16> literal");
        strcpy(lexeme,"0F0.0");
        *token = FLOAT;
    }
    else
```

```
{  
    lexeme[i++] = nextCharacter;  
    GetNextCharacter();  
    while ( isdigit(nextCharacter) )  
    {  
        lexeme[i++] = nextCharacter;  
        GetNextCharacter();  
    }  
    if ( nextCharacter != '.' )  
    {  
        RecordSyntaxError("Invalid <F16> literal");  
        strcpy(lexeme,"0F0.0");  
        *token = FLOAT;  
    }  
    else  
    {  
        lexeme[i++] = nextCharacter;  
        GetNextCharacter();  
        if ( !isdigit(nextCharacter) )  
        {  
            RecordSyntaxError("Invalid <F16> literal");  
            strcpy(lexeme,"0F0.0");  
            *token = FLOAT;  
        }  
        else  
        {
```

```
lexeme[i++] = nextCharacter;
GetNextCharacter();
while ( isdigit(nextCharacter) )
{
    lexeme[i++] = nextCharacter;
    GetNextCharacter();
}
if ( toupper(nextCharacter) != 'E' )
{
    lexeme[i] = '\0';
    *token = FLOAT;
}
else
{
    lexeme[i++] = nextCharacter;
    GetNextCharacter();
    if ( nextCharacter == '-' )
    {
        lexeme[i++] = nextCharacter;
        GetNextCharacter();
    }
    if ( !isdigit(nextCharacter) )
    {
        RecordSyntaxError("Invalid <F16> literal");
        strcpy(lexeme,"0F0.0");
        *token = FLOAT;
    }
}
```

```
        }

        else
        {

            lexeme[i++] = nextCharacter;
            GetNextCharacter();
            while ( isdigit(nextCharacter) )
            {
                lexeme[i++] = nextCharacter;
                GetNextCharacter();
            }
            lexeme[i] = '\0';
            *token = FLOAT;
        }
    }
}

}

}

else
{
    RecordSyntaxError("Invalid <I16> or <F16> prefix\n");
    strcpy(lexeme, "0D0");
    *token = INTEGER;
}
}
```

```
else
{
    switch ( nextCharacter )
    {
        // <string>

        case '':
            i = 0;
            GetNextCharacter();
            while ( nextCharacter != '"' )
            {
                if ( nextCharacter == '\\' )
                {
                    lexeme[i++] = nextCharacter;
                    GetNextCharacter();
                }
                lexeme[i++] = nextCharacter;
                GetNextCharacter();
            }
            lexeme[i] = '\0';
            *token = STRING;
            GetNextCharacter();
            break;

        case '\\':
            *token = CHARACTER;
            GetNextCharacter();
            if ( nextCharacter == '\\' )
```

```
GetNextCharacter();

lexeme[0] = nextCharacter; lexeme[1] = '\0';

GetNextCharacter();

GetNextCharacter();

break;

case ':':

*token = COLON;

lexeme[0] = nextCharacter; lexeme[1] = '\0';

GetNextCharacter();

break;

case '#':

*token = POUND;

lexeme[0] = nextCharacter; lexeme[1] = '\0';

GetNextCharacter();

break;

case '@':

*token = ATSIGN;

lexeme[0] = nextCharacter; lexeme[1] = '\0';

GetNextCharacter();

break;

case '$':

*token = DOLLAR;

lexeme[0] = nextCharacter; lexeme[1] = '\0';

GetNextCharacter();

break;

case '*':
```

```
*token = ASTERISK;

lexeme[0] = nextCharacter; lexeme[1] = '\0';
GetNextCharacter();
break;

case EOLC:
    *token = EOLTOKEN;
    lexeme[0] = '\0';
    break;

case EOPC:
    *token = EOPTOKEN;
    lexeme[0] = '\0';
    break;

default:
    RecordSyntaxError("Unknown token\n");
    *token = UNKNOWN;
    lexeme[0] = nextCharacter; lexeme[1] = '\0';
    GetNextCharacter();
    break;
}

}

//-----
void RecordSyntaxError(const char syntaxError[])
//-----
{
```

```
if ( numberOfSyntaxErrors <= 10 )
    strcpy(syntaxErrors[+numberOfSyntaxErrors],syntaxError);
}

//-----
void GetNextCharacter()
//-----
{
    if ( atEOP )
        nextCharacter = EOPC;
    else if ( sourceLineIndex <= ((int) strlen(sourceLine)-1) )
    {
        nextCharacter = sourceLine[sourceLineIndex];
        sourceLineIndex += 1;
    }
    else
        nextCharacter = EOLC;
}

//-----
void ReadSourceLine()
//-----
{
    if ( feof(SOURCE) )
        atEOP = true;
    else
```

```
{  
    if ( fgets(sourceLine,SOURCELINELENGTH,SOURCE) == NULL )  
        atEOP = true;  
    else  
    {  
        if ( (strchr(sourceLine,'\n') == NULL) && !feof(SOURCE) )  
        {  
            fprintf(LOG,"***** Source line is too long!");  
            fflush(LOG);  
        }  
        // Erase *ALL* control characters at end of source line (if any)  
        while ( (0 <= (int) strlen(sourceLine)-1) &&  
               iscntrl(sourceLine[(int) strlen(sourceLine)-1]) )  
            sourceLine[(int) strlen(sourceLine)-1] = '\0';  
        sourceLineIndex = 0;  
    }  
}  
  
//-----  
void ListTopOfPageHeader(int *pageNumber,int *linesOnPage)  
//-----  
{  
/*  
111111112222222233333334444444455555555666666667777777778  
123456789012345678901234567890123456789012345678901234567890
```

Page XXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
LC Object Line Source Line
-----
0XXXXX XXXXXXXX XXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*/
*pageNumber += 1;
*linesOnPage = 0;
fprintf(LOG,"%cPage %3d %s.stm\n\n",FF,*pageNumber,sourceFileName);
fprintf(LOG,"LC Object Line Source Line\n");
fprintf(LOG,"-----\n");
fflush(LOG);

}

//-----
void DefineIdentifierInTable(const char lexeme[],int value)
//-----
{
    bool IdentifierIsInTable(const char lexeme[]);
    int FindIdentifierInTable(const char lexeme[]);

    if ( IdentifierIsInTable(lexeme) )
    {
        int index = FindIdentifierInTable(lexeme);
        identifierTable[index].numberOfDefinitions++;
    }
}
```

```
else
{
    if ( sizeOfIdentifierTable <= SIZEOFIDENTIFIERTABLE )
    {
        sizeOfIdentifierTable++;
        strncpy(identifierTable[sizeOfIdentifierTable].identifier,lexeme,MAXIMUMLENGTHIDENTIFIER);
        identifierTable[sizeOfIdentifierTable].numberOfDefinitions = 1;
        identifierTable[sizeOfIdentifierTable].value = value;
    }
    else
        ProcessRunTimeError("Identifier table overflow",true);
}

//-----
bool IdentifierIsInTable(const char lexeme[])
//-----
{
    bool isInTable;
    int i,index;
    char UClexeme[SOURCELINELENGTH+1];

    for (i = 0; i <= (int) strlen(lexeme); i++)
        UClexeme[i] = toupper(lexeme[i]);
    isInTable = false;
    index = 1;
```

```
while ( !isInTable && (index <= sizeOfIdentifierTable) )
{
    char UCidentifier[MAXIMUMLENGTHIDENTIFIER+1];

    for (i = 0; i <= (int) strlen(identifierTable[index].identifier); i++)
        UCidentifier[i] = toupper(identifierTable[index].identifier[i]);
    if ( strncmp(UCidentifier,UClexeme,MAXIMUMLENGTHIDENTIFIER) == 0 )
        isInTable = true;
    else
        index += 1;
}

return( isInTable );
}

//-----
int FindIdentifierInTable(const char lexeme[])
//-----
{

    bool isFound;
    int i,index;
    char UClexeme[SOURCELINELENGTH+1];

    for (i = 0; i <= (int) strlen(lexeme); i++)
        UClexeme[i] = toupper(lexeme[i]);
    index = 1;
    isFound = false;
```

```
while ( !isFound )
{
    char UCidentifier[MAXIMUMLENGTHIDENTIFIER+1];

    for (i = 0; i <= (int) strlen(identifierTable[index].identifier); i++)
        UCidentifier[i] = toupper(identifierTable[index].identifier[i]);
    if ( strncmp(UCidentifier,UCLexeme,MAXIMUMLENGTHIDENTIFIER) == 0 )
        isFound = true;
    else
        index += 1;
}

return( index );
}

//-----
void InitializeMainMemory()
//-----
{
    int address;

    for (address = 0X0000; address <= 0xFFFF; address++)
        mainMemory[address] = 0X00u;
}

//-----
void WriteBYTEToMainMemory(int address,BYTE byte)
```

```
-----  
{  
    if ( (0X0000 <= address) && (address <= 0xFFFF) )  
        mainMemory[address] = byte;  
    else  
    {  
        char information[SOURCELINELENGTH+1];  
  
        sprintf(information,"Write main memory address 0X%08X is not in [ 0X0000,0xFFFF ]",address);  
        ProcessRunTimeError(information,true);  
    }  
}  
  
-----  
void ReadBYTEFromMainMemory(int address,BYTE *byte)  
-----  
{  
    if ( (0X0000 <= address) && (address <= 0xFFFF) )  
        *byte = mainMemory[address];  
    else  
    {  
        char information[SOURCELINELENGTH+1];  
  
        sprintf(information,"Read main memory address 0X%08X is not in [ 0X0000,0xFFFF ]",address);  
        ProcessRunTimeError(information,true);  
    }  
}
```

```
}

//-----
void WriteWORDToMainMemory(int address,WORD word)
//-----
{

    void WriteBYTEToMainMemory(int address,BYTE byte);

    BYTE hiByte = HIBYTE(word);
    BYTE loByte = LOBYTE(word);

    // STM is a big-endian machine
    WriteBYTEToMainMemory( address,hiByte);
    WriteBYTEToMainMemory(address+1,loByte);

}

//-----
void ReadWORDFromMainMemory(int address,WORD *word)
//-----
{
    void ReadBYTEToMainMemory(int address,BYTE *byte);

    BYTE loByte,hiByte;

    // STM is a big-endian machine
    ReadBYTEToMainMemory(address ,&hiByte);
```

```
ReadBYTEFromMainMemory(address+1,&loByte);

*word = (hiByte << 8) | loByte;

}

//-----
void ExecuteProgram()
//-----

{

void WriteBYTEToMainMemory(int address,BYTE byte);
void WriteWORDToMainMemory(int address,WORD word);
void ReadBYTEFromMainMemory(int address,BYTE *byte);
void ReadWORDFromMainMemory(int address,WORD *word);

WORD MemoryOperandEA(BYTE mode,WORD O16,WORD PC,WORD *SP,WORD FB,WORD SB,char information[]);
void ConvertFloatToHalfFloat(float F,WORD *HF);
void ConvertHalfFloatToFloat(WORD HF,float *F);
void ConvertHalfFloatToBase10(WORD HF,char base10[]);
void TraceFREEnodes(WORD FREEblocks);

WORD PC,SP,FB,SB;      // CPU registers
char N,Z,P,T,L,E,G,R; // FLAGS "register" (R is reserved for future use)
bool running;

char IN[SOURCELINELENGTH+1],OUT[SOURCELINELENGTH+1];

WORD heapBase,heapSize,FREEnodes;
```

```
PC = 0X0000u;
SP = 0xFFFFEu; // address of first available word on run-time stack (locations 0xFFFFE:0xFFFF)
FB = 0X0000u; // default address that *MUST* be changed by machine program before use
SB = 0X0000u; // default address that *MUST* be changed by machine program before use
N = Z = P = T = L = E = G = 0;

running = true;

OUT[0] = '\0';

/*
PC  SP  TOS0  TOS1  TOS2 mnemonic  information
-----
XXXX XXXX XXXX XXXX XXXX XXXXXXXXX XXXXXX...
*/
fprintf(LOG, "\n\n");
fprintf(LOG, " PC  SP  TOS0  TOS1  TOS2 mnemonic  information\n");
fprintf(LOG, "-----\n");
fflush(LOG);
do
{
    WORD O16,W16,RHS,LHS,TOS,EA,memoryOperand;
    BYTE opCode,mode;
    char traceLine[SOURCELINELENGTH+1],information[SOURCELINELENGTH+1];

    sprintf(traceLine,"%04hX %04hX",PC,SP);
```

```
if ( SP <= 0XFFFC )
{
    ReadWORDFromMainMemory(SP+2,&W16);
    sprintf(information," %04hX",W16);
    strcat(traceLine,information);

}
else
    strcat(traceLine,"      ");

if ( SP <= 0XFFFA )
{
    ReadWORDFromMainMemory(SP+4,&W16);
    sprintf(information," %04hX",W16);
    strcat(traceLine,information);

}
else
    strcat(traceLine,"      ");
    strcat(traceLine," ");

if ( SP <= 0XFFF9 )
{
    ReadWORDFromMainMemory(SP+6,&W16);
    sprintf(information," %04hX",W16);
    strcat(traceLine,information);

}
else
    strcat(traceLine,"      ");
    strcat(traceLine," ");
```

```
ReadBYTEFromMainMemory(PC,&opCode); PC += 1;

switch ( opCode )
{
    // 0X00      NOOP           OpCode           Do nothing
    case 0X00:
        strcat(traceLine,"NOOP      ");
        break;

    // 0X01      PUSH      memory     OpCode:mode:016      Push word memory[EA] on run-time stack
    case 0X01:
        strcat(traceLine,"PUSH      ");
        ReadBYTEFromMainMemory(PC,&mode); PC += 1;
        ReadWORDFromMainMemory(PC,&016); PC += 2;
        EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
        strcat(traceLine,information);
        ReadWORDFromMainMemory(EA,&memoryOperand);
        WriteWORDToMainMemory(SP,memoryOperand); SP -= 2;
        sprintf(information," = 0X%04hX",memoryOperand);
        strcat(traceLine,information);
        break;

    // 0X02      PUSHA     memory     OpCode:mode:016      Push EA on run-time stack
    case 0X02:
        strcat(traceLine,"PUSHA     ");
}
```

```
ReadBYTEFromMainMemory(PC,&mode); PC += 1;

ReadWORDFromMainMemory(PC,&016); PC += 2;

EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);

strcat(traceLine,information);

WriteWORDToMainMemory(SP,EA); SP -= 2;

break;

// 0X03      POP      memory      OpCode:mode:016      Pop word from run-time stack and store in memory[EA]

case 0X03:

    strcat(traceLine,"POP      ");

    ReadBYTEFromMainMemory(PC,&mode); PC += 1;

    ReadWORDFromMainMemory(PC,&016); PC += 2;

    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);

    strcat(traceLine,information);

    ReadWORDFromMainMemory(SP+2,&memoryOperand); SP += 2;

    WriteWORDToMainMemory(EA,memoryOperand);

    sprintf(information," = 0X%04hX",memoryOperand);

    strcat(traceLine,information);

    break;

// 0X04      DISCARD #W16      OpCode:016      Discard 016U words at top of run-time stack

case 0X04:

    strcat(traceLine,"DISCARD  ");

    ReadWORDFromMainMemory(PC,&016); PC += 2;

    SP += 2*016;

    sprintf(information," #%hd words from top-of-stack",016);
```

```
        strcat(traceLine,information);

        break;

// 0X05      SWAP           OpCode          Pop RHS,LHS; push RHS,LHS

case 0X05:
        strcat(traceLine,"SWAP      ");
        ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
        ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
        WriteWORDToMainMemory(SP,RHS); SP -= 2;
        WriteWORDToMainMemory(SP,LHS); SP -= 2;
        sprintf(information," 0X%04hX <-> 0X%04hX",LHS,RHS);
        strcat(traceLine,information);
        break;

// 0X06      MAKEDUP         OpCode          Read TOS; push TOS (duplicate TOS)

case 0X06:
        strcat(traceLine,"MAKEDUP   ");
        ReadWORDFromMainMemory(SP+2,&TOS);
        WriteWORDToMainMemory(SP,TOS); SP -= 2;
        sprintf(information," duplicate 0X%04hX",TOS);
        strcat(traceLine,information);
        break;

// 0X07      PUSHSP          OpCode          Push SP

case 0X07:
        strcat(traceLine,"PUSHSP   ");
```

```
    WriteWORDToMainMemory(SP,SP); SP -= 2;
    sprintf(information," 0X%04hX",SP+2);
    strcat(traceLine,information);
    break;

// 0X08      PUSHFB          OpCode          Push FB
case 0X08:
    strcat(traceLine,"PUSHFB    ");
    WriteWORDToMainMemory(SP,FB); SP -= 2;
    sprintf(information," 0X%04hX",FB);
    strcat(traceLine,information);
    break;

// 0X09      PUSHSB          OpCode          Push SB
case 0X09:
    strcat(traceLine,"PUSHSB    ");
    WriteWORDToMainMemory(SP,SB); SP -= 2;
    sprintf(information," 0X%04hX",SB);
    strcat(traceLine,information);
    break;

// 0X0A      POPSP           OpCode          Pop  SP
case 0X0A:
    strcat(traceLine,"POPSP    ");
    ReadWORDFromMainMemory(SP+2,&SP); // ***SP += 2; NOT REQUIRED***
    sprintf(information," SP = 0X%04hX",SP);
```

```
        strcat(traceLine,information);

        break;

// 0X0B      POPFB          OpCode          Pop FB

case 0X0B:
    strcat(traceLine,"POPFB      ");
    ReadWORDFromMainMemory(SP+2,&FB); SP += 2;
    sprintf(information," FB = 0X%04hX",FB);
    strcat(traceLine,information);
    break;

// 0X0C      POPSB          OpCode          Pop SB

case 0X0C:
    strcat(traceLine,"POPSB      ");
    ReadWORDFromMainMemory(SP+2,&SB); SP += 2;
    sprintf(information," SB = 0X%04hX",SB);
    strcat(traceLine,information);
    break;

// 0X0D      SETAAE memory   OpCode:mode:016   Pop key,value; add (key,value) to array in memory[EA] (Note 1)
// Note 1: When (key,value) pair is found, the existing value is replaced with the value popped
//         from run-time stack. When (key,value) pair is not found, the pair is stored in the next available
//         (key,value) slot. A fatal run-time error occurs when (key,value) pair is not found and (size == capacity).

case 0X0D:
{
```

```
WORD size, capacity, keyS, valueS, keyM, valueM;
int i;
bool isFound;

strcat(traceLine,"SETAAE    ");
ReadBYTEFromMainMemory(PC,&mode); PC += 1;
ReadWORDFromMainMemory(PC,&016); PC += 2;
EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
strcat(traceLine,information);
ReadWORDFromMainMemory(EA,&size);
ReadWORDFromMainMemory(EA+2,&capacity);
ReadWORDFromMainMemory(SP+2,&keyS); SP += 2;
ReadWORDFromMainMemory(SP+2,&valueS); SP += 2;
i = 1;
isFound = false;
while ( (i <= size) && !isFound )
{
    ReadWORDFromMainMemory(EA+4+4*(i-1),&keyM);
    if ( keyM == keyS )
        isFound = true;
    else
        i++;
}
if ( isFound )
    WriteWORDToMainMemory(EA+4+4*(i-1)+2,valueS);
else
```

```

{
    if ( size == capacity ) ProcessRunTimeError("Associative array overflow",true);
    size++;
    WriteWORDToMainMemory(EA,size);
    WriteWORDToMainMemory(EA+4+4*(size-1) ,keyS);
    WriteWORDToMainMemory(EA+4+4*(size-1)+2,valueS);
}

sprintf(information," pair = (0X%04hX,0X%04hX)",keyS,valueS);
strcat(traceLine,information);

}
break;

// 0X0E    GETAAE   memory      OpCode:mode:016      Pop key; find (key,value) in array in memory[EA]; push value (Note 2)
// Note 2: A fatal run-time error occurs when (key,value) pair is not found.

case 0X0E:
{
    WORD size,capacity,keyS,valueS,keyM,valueM;
    int i;
    bool isFound;

    strcat(traceLine,"GETAAE   ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(EA,&size);
}

```

```

ReadWORDFromMainMemory(EA+2,&capacity);

ReadWORDFromMainMemory(SP+2,&keyS); SP += 2;

i = 1;

isFound = false;

while ( (i <= size) && !isFound )

{

    ReadWORDFromMainMemory(EA+4+4*(i-1),&keyM);

    if ( keyM == keyS )

        isFound = true;

    else

        i++;

}

if ( isFound )

{

    ReadWORDFromMainMemory(EA+4+4*(i-1)+2,&valueM);

    WriteWORDToMainMemory(SP,valueM); SP -= 2;

}

else

    ProcessRunTimeError("Associative array key not found",true);

sprintf(information," pair = (0X%04hX,0X%04hX)",keyS,valueM);

strcat(traceLine,information);

}

break;

// 0X0F ADRAAE memory      OpCode:mode:016      Pop key; find (key,value) in array in memory[EA]; push address of value (Note 3)

// Note 3: When (key,value) pair is not found, the pair is stored in the next available (key,value) slot.

```

```
//      A fatal run-time error occurs when (key,value) pair is not found and (size == capacity).

case 0X0F:
{
    WORD size, capacity, keyS, valueS, keyM, valueM, addressValueM;
    int i;
    bool isFound;

    strcat(traceLine, "ADRAAE    ");

    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&O16); PC += 2;
    EA = MemoryOperandEA(mode,O16,PC,&SP,FB,SB,information);
    strcat(traceLine,information);

    ReadWORDFromMainMemory(EA,&size);
    ReadWORDFromMainMemory(EA+2,&capacity);
    ReadWORDFromMainMemory(SP+2,&keyS); SP += 2;
    i = 1;
    isFound = false;

    while ( (i <= size) && !isFound )
    {
        ReadWORDFromMainMemory(EA+4+4*(i-1),&keyM);
        if ( keyM == keyS )
            isFound = true;
        else
            i++;
    }
    if ( isFound )

```

```

{
    addressValueM = EA+4+4*(i-1)+2;
}
else
{
    if ( size == capacity ) ProcessRunTimeError("Associative array overflow",true);
    size++;
    WriteWORDToMainMemory(EA,size);
    WriteWORDToMainMemory(EA+4+4*(size-1) ,keyS);
    addressValueM = EA+4+4*(size-1)+2;
}

WriteWORDToMainMemory(SP,addressValueM); SP -= 2;
sprintf(information," key = 0X%04hX, address = 0X%04hX",keyS,addressValueM);
strcat(traceLine,information);

}
break;

// Note 9: Assumes that memory block pointed to by LHS is large enough to accommodate structure stored
// in memory block pointed to by RHS.

// 0X10      COPYAA          OpCode          Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in
//                                         [ 0,2*capacity+1 ] (Note 9)

case 0X10:
{
    int i;
    WORD capacity;
}

```

```

        strcat(traceLine,"COPYAA    ");
        ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
        ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
        ReadWORDFromMainMemory(RHS+2,&capacity);
        for (i = 0; i <= 2*capacity+1; i++)
        {
            ReadWORDFromMainMemory(RHS+2*i,&W16);
            WriteWORDToMainMemory(LHS+2*i,W16);
        }
        sprintf(information," %4u words from 0X%04hX to 0X%04hX",1+(2*capacity+1),RHS,LHS);
        strcat(traceLine,information);
    }
    break;

// Note 8: A STM string is composed of 2+capacity words of contiguous memory. Word 1 is the length of
// the string (the number of characters contained in the string); word 2 is the string's capacity; and
// words 3-to-(capacity+2) are reserved for the string's characters. When empty, length = 0, otherwise
// (1 <= length <= capacity). A fatal error occurs when (1 <= index <= length) is not true for SETSE,
// GETSE, and ADRSE and a fatal error occurs when (length = capacity) when ADDSE begins execution.

// 0X11    SETSE    memory      OpCode:mode:016      Pop character,index; store character in
//                                         memory[EA+4+2*(index-1)] (Note 8)

case 0X11:
{
    WORD length,capacity,index,character;
}

```

```
strcat(traceLine,"SETSE      ");

ReadBYTEFromMainMemory(PC,&mode); PC += 1;

ReadWORDFromMainMemory(PC,&016); PC += 2;

EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);

strcat(traceLine,information);

ReadWORDFromMainMemory(EA,&length);

ReadWORDFromMainMemory(EA+2,&capacity);

ReadWORDFromMainMemory(SP+2,&character); SP += 2;

ReadWORDFromMainMemory(SP+2,&index); SP += 2;

if ( !(1 <= index) &&(index <= length) ) ProcessRunTimeError("Invalid string index",true);

WriteWORDToMainMemory(EA+4+2*(index-1),character);

sprintf(information,", set string[0X%04hX] to '%c'",index,LOBYTE(character));

strcat(traceLine,information);

}

break;

// 0X12    GETSE   memory      OpCode:mode:016      Pop index; push memory[EA+4+2*(index-1)] (Note 8)

case 0X12:

{
    WORD length,capacity,index,character;

    strcat(traceLine,"GETSE      ");

    ReadBYTEFromMainMemory(PC,&mode); PC += 1;

    ReadWORDFromMainMemory(PC,&016); PC += 2;

    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
```

```
        strcat(traceLine,information);

        ReadWORDFromMainMemory(EA,&length);
        ReadWORDFromMainMemory(EA+2,&capacity);
        ReadWORDFromMainMemory(SP+2,&index); SP += 2;

        if ( !((1 <= index) &&(index <= length)) ) ProcessRunTimeError("Invalid string index",true);

        ReadWORDFromMainMemory(EA+4+2*(index-1),&character);
        WriteWORDToMainMemory(SP,character); SP -= 2;
        sprintf(information,", get string[0X%04hX] = '%c'",index,LOBYTE(character));
        strcat(traceLine,information);

    }

    break;

// 0X13      ADRSE    memory      OpCode:mode:016      Pop index; push address (EA+4+2*(index-1)) (Note 8)

case 0X13:
{
    WORD length,capacity,index,character;

    strcat(traceLine,"ADRSE      ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(EA,&length);
    ReadWORDFromMainMemory(EA+2,&capacity);
    ReadWORDFromMainMemory(SP+2,&index); SP += 2;

    if ( !((1 <= index) &&(index <= length)) ) ProcessRunTimeError("Invalid string index",true);
```

```
    WriteWORDToMainMemory(SP,EA+4+2*(index-1)); SP -= 2;
    sprintf(information,", address of string[0X%04hX] = 0X%04hX",index,EA+4+2*(index-1));
    strcat(traceLine,information);

}

break;

// 0X14    ADDSE    memory      OpCode:mode:016      Pop character; store character in memory[EA+4+2*length];
//                                         increment length (Note 8)

case 0X14:
{
    WORD length,capacity,index,character;

    strcat(traceLine,"ADDSE    ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(EA,&length);
    ReadWORDFromMainMemory(EA+2,&capacity);
    ReadWORDFromMainMemory(SP+2,&character); SP += 2;
    if ( length == capacity ) ProcessRunTimeError("String overflow",true);
    length++;
    WriteWORDToMainMemory(EA+4+2*(length-1),character);
    WriteWORDToMainMemory(EA,length);
    sprintf(information,", add string[0X%04hX] to '%c'",length,LOBYTE(character));
    strcat(traceLine,information);
```

```
}

break;

// Note 9: Assumes that memory block pointed to by LHS is large enough to accommodate structure stored
// in memory block pointed to by RHS.

// 0X15    COPYS          OpCode          Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in
//                                     [ 0,capacity+1 ] (Note 9)

case 0X15:
{
    int i;
    WORD capacity;

    strcat(traceLine,"COPYS      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ReadWORDFromMainMemory(RHS+2,&capacity);
    for (i = 0; i <= capacity+1; i++)
    {
        ReadWORDFromMainMemory(RHS+2*i,&W16);
        WriteWORDToMainMemory(LHS+2*i,W16);
    }
    sprintf(information," %4u words from 0X%04hX to 0X%04hX",1+(capacity+1),RHS,LHS);
    strcat(traceLine,information);
}
break;
```

```
// Note 12: Assumes that memory block pointed to by RES is large enough to accommodate the concatenation
//          of strings pointed to by LHS and RHS. A fatal error occurs when
//          (length-of-LHS + length-of-RHS) > capacity-of-RES

// 0X1D    CONCATS           OpCode          Pop RES,RHS,LHS; memory[RES] = memory[LHS] concatenate memory[RHS];
//                                         push RES (Note 12)

case 0X1D:
{
    int i;
    WORD RES,capacityRES,lengthLHS,lengthRHS;

    strcat(traceLine,"CONCATS  ");
    ReadWORDFromMainMemory(SP+2,&RES); SP += 2;
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ReadWORDFromMainMemory(RES+2,&capacityRES);
    ReadWORDFromMainMemory(RHS,&lengthRHS);
    ReadWORDFromMainMemory(LHS,&lengthLHS);

    if ( lengthRHS+lengthLHS > capacityRES ) ProcessRunTimeError("String overflow",true);
    WriteWORDToMainMemory(RES,lengthRHS+lengthLHS);
    for (i = 0; i <= lengthLHS-1; i++)
    {
        ReadWORDFromMainMemory(LHS+4+2*i,&W16);
        WriteWORDToMainMemory(RES+4+2*i,W16);
    }
}
```

```
for (i = 0; i <= lengthRHS-1; i++)
{
    ReadWORDFromMainMemory(RHS+4+2*i,&W16);
    WriteWORDToMainMemory(RES+4+2*(i+lengthLHS),W16);
}
WriteWORDToMainMemory(SP,RES); SP -= 2;
sprintf(information," 0X%04hX = 0X%04hX + 0X%04hX (%4u words)",RES,LHS,RHS,lengthLHS+lengthRHS);
strcat(traceLine,information);
}

break;

// Note 10: A fatal error occurs when the following equation is not true for SETAE, GETAE, and ADRAE.
// ((LB1 <= index1 <= UB1) AND (LB2 <= index2 <= UB2) AND...AND (LBn <= indexn <= UBn))

// 0X16    SETAE    memory      OpCode:mode:016      Pop value,index(n),index(n-1),...,index(1); store value at offset
//                                         in array (Note 10)

case 0X16:
{
    int i,offset;
    WORD n,capacity,indexi,productOfSizes;
    WORD LBi,UBi,value;

    strcat(traceLine,"SETAE    ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
```

```
        strcat(traceLine,information);

        ReadWORDFromMainMemory(EA,&n);

        ReadWORDFromMainMemory(SP+2,&value); SP += 2;

        offset = 0;

        productOfSizes = 1;

        for (i = n; i >= 1; i--)

        {

            ReadWORDFromMainMemory(SP+2,&indexi); SP += 2;

            ReadWORDFromMainMemory(EA+2*(2*i-1),&LBi);

            ReadWORDFromMainMemory(EA+2*(2*i+0),&UBi);

            if ( !((SIGNED(LBi) <= SIGNED(indexi)) && (SIGNED(indexi) <= SIGNED(UBi))) )

                ProcessRunTimeError("Invalid array index",true);

            offset += productOfSizes*(SIGNED(indexi)-SIGNED(LBi));

            productOfSizes *= (SIGNED(UBi)-SIGNED(LBi)+1);

        }

        WriteWORDToMainMemory(EA+2*(1+2*n+offset),value);

        sprintf(information,", set 0X%04hX at 0X%04hX (offset %d)",value,EA+2*(1+2*n+offset),offset);

        strcat(traceLine,information);

    }

    break;

// 0X17    GETAE   memory      OpCode:mode:016      Pop index(n),index(n-1),...,index(1); push value found at offset
//                                         in array (Note 10)

case 0X17:

{



    int i,offset;
```

```
WORD n,capacity,indexi,productOfSizes;
WORD LBi,UBi,value;

strcat(traceLine,"GETAE      ");
ReadBYTEFromMainMemory(PC,&mode); PC += 1;
ReadWORDFromMainMemory(PC,&O16); PC += 2;
EA = MemoryOperandEA(mode,O16,PC,&SP,FB,SB,information);
strcat(traceLine,information);
ReadWORDFromMainMemory(EA,&n);
offset = 0;
productOfSizes = 1;
for (i = n; i >= 1; i--)
{
    ReadWORDFromMainMemory(SP+2,&indexi); SP += 2;
    ReadWORDFromMainMemory(EA+2*(2*i-1),&LBi);
    ReadWORDFromMainMemory(EA+2*(2*i+0),&UBi);
    if ( !((SIGNED(LBi) <= SIGNED(indexi)) && (SIGNED(indexi) <= SIGNED(UBi))) )
        ProcessRunTimeError("Invalid array index",true);
    offset += productOfSizes*(SIGNED(indexi)-SIGNED(LBi));
    productOfSizes *= (SIGNED(UBi)-SIGNED(LBi)+1);
}
ReadWORDFromMainMemory(EA+2*(1+2*n+offset),&value);
WriteWORDToMainMemory(SP,value); SP -= 2;
sprintf(information,", get 0X%04hX at 0X%04hX (offset %d)",value,EA+2*(1+2*n+offset),offset);
strcat(traceLine,information);
}
```

```
break;

// 0X18    ADRAE    memory      OpCode:mode:016      Pop index(n),index(n-1),...,index(1); push address of value found
//                                         at offset in array (Note 10)

case 0X18:
{
    int i,offset;
    WORD n,capacity,indexi,productOfSizes;
    WORD LBi,UBi;

    strcat(traceLine,"ADRAE      ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(EA,&n);
    offset = 0;
    productOfSizes = 1;
    for (i = n; i >= 1; i--)
    {
        ReadWORDFromMainMemory(SP+2,&indexi); SP += 2;
        ReadWORDFromMainMemory(EA+2*(2*i-1),&LBi);
        ReadWORDFromMainMemory(EA+2*(2*i+0),&UBi);
        if ( !((SIGNED(LBi) <= SIGNED(indexi)) && (SIGNED(indexi) <= SIGNED(UBi))) )
            ProcessRunTimeError("Invalid array index",true);
        offset += productOfSizes*(SIGNED(indexi)-SIGNED(LBi));
    }
}
```

```
    productOfSizes *= (SIGNED(UBi)-SIGNED(LBi)+1);

}

WriteWORDToMainMemory(SP,EA+2*(1+2*n+offset)); SP -= 2;
sprintf(information,", address of array value (offset %d) = 0X%04hX",offset,EA+2*(1+2*n+offset));
strcat(traceLine,information);

}

break;

// Note 9: Assumes that memory block pointed to by LHS is large enough to accommodate structure stored
// in memory block pointed to by RHS.

// 0X19      COPYA          OpCode          Pop RHS,LHS; memory[LHS+2*i] = memory[RHS+2*i], i in
//                                         [ 0,2*n+capacity ] (Note 9)

case 0X19:

{
    int i;
    WORD n,capacity;
    WORD LBi,UBi;

    strcat(traceLine,"COPYA    ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ReadWORDFromMainMemory(RHS,&n);

    capacity = 1;
    for (i = n; i >= 1; i--)
    {
```

```
ReadWORDFromMainMemory(RHS+2*(2*i-1),&LBi);
ReadWORDFromMainMemory(RHS+2*(2*i+0),&UBi);
capacity *= (SIGNED(UBi)-SIGNED(LBi)+1);
}
for (i = 0; i <= 2*n+capacity; i++)
{
    ReadWORDFromMainMemory(RHS+2*i,&W16);
    WriteWORDToMainMemory(LHS+2*i,W16);
}
sprintf(traceLine," %4u words from 0X%04hX to 0X%04hX",1+(2*n+capacity),RHS,LHS);
strcat(traceLine,information);
}

break;

// 0X1A      GETAN      memory      OpCode:mode:016      Push n (# of dimensions)

case 0X1A:
{
    WORD n;

    strcat(traceLine,"GETAN      ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&O16); PC += 2;
    EA = MemoryOperandEA(mode,O16,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(EA,&n);
    WriteWORDToMainMemory(SP,n); SP -= 2;
```

```
sprintf(information," # dimensions n = %u",n);
strcat(traceLine,information);

}

break;

// Note 11: A fatal error occurs when dimension # i is not in [ 1,n ].

// 0X1B    GETALB  memory      OpCode:mode:016      Pop dimension # i; push lower-bound, LBi (Note 11)

case 0X1B:

{
WORD n,i,LBi;

strcat(traceLine,"GETALB    ");

ReadBYTEFromMainMemory(PC,&mode); PC += 1;
ReadWORDFromMainMemory(PC,&016); PC += 2;
EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
strcat(traceLine,information);
ReadWORDFromMainMemory(SP+2,&i); SP += 2;
ReadWORDFromMainMemory(EA,&n);
if ( !( (1 <= SIGNED(i)) && (SIGNED(i) <= SIGNED(n)) ) )
    ProcessRunTimeError("Invalid array dimension #",true);
ReadWORDFromMainMemory(EA+2*(2*i-1),&LBi);
WriteWORDToMainMemory(SP,LBi); SP -= 2;
sprintf(information," LB#%u = %u",i,LBi);
strcat(traceLine,information);

}

break;
```

```
// 0X1C    GETAUB  memory      OpCode:mode:016      Pop dimension # i; push upper-bound, UBi (Note 11)
case 0X1C:
{
    WORD n,i,UBi;

    strcat(traceLine,"GETAUB    ");
    ReadBYTEFromMainMemory(PC,&mode); PC += 1;
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    EA = MemoryOperandEA(mode,016,PC,&SP,FB,SB,information);
    strcat(traceLine,information);
    ReadWORDFromMainMemory(SP+2,&i); SP += 2;
    ReadWORDFromMainMemory(EA,&n);
    if ( !( (1 <= SIGNED(i)) && (SIGNED(i) <= SIGNED(n)) ) )
        ProcessRunTimeError("Invalid array dimension #",true);
    ReadWORDFromMainMemory(EA+2*(2*i+0),&UBi);
    WriteWORDToMainMemory(SP,UBi); SP -= 2;
    sprintf(information,", UB#%u = %u",i,UBi);
    strcat(traceLine,information);
}
break;

// 0X20    ADDI           OpCode           Pop RHS,LHS; push integer ( LHS+RHS )
case 0X20:
{
    strcat(traceLine,"ADDI    ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
```

```
ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

TOS = UNSIGNED(SIGNED(LHS)+SIGNED(RHS));

WriteWORDToMainMemory(SP,TOS); SP -= 2;

sprintf(information," 0X%04hX = 0X%04hX + 0X%04hX",TOS,LHS,RHS);

strcat(traceLine,information);

break;

// 0X21      ADDF          OpCode           Pop RHS,LHS; push   float ( LHS+RHS )

case 0X21:

{

float FLHS,FRHS;

char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

strcat(traceLine,"ADDF      ");

ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

ConvertHalfFloatToFloat(LHS,&FLHS);

ConvertHalfFloatToFloat(RHS,&FRHS);

ConvertFloatToHalfFloat((FLHS+FRHS),&TOS);

WriteWORDToMainMemory(SP,TOS); SP -= 2;

ConvertHalfFloatToBase10(TOS,base10TOS);

ConvertHalfFloatToBase10(LHS,base10LHS);

ConvertHalfFloatToBase10(RHS,base10RHS);

sprintf(information," %s = %s + %s",base10TOS,base10LHS,base10RHS);

strcat(traceLine,information);

}
```

```
break;

// 0X22    SUBI          OpCode      Pop RHS,LHS; push integer ( LHS-RHS )

case 0X22:
    strcat(traceLine,"SUBI      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = UNSIGNED(SIGNED(LHS)-SIGNED(RHS));
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX - 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;

// 0X23    SUBF          OpCode      Pop RHS,LHS; push float ( LHS-RHS )

case 0X23:
{
    float FLHS,FRHS;
    char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

    strcat(traceLine,"SUBF      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ConvertHalfFloatToFloat(LHS,&FLHS);
    ConvertHalfFloatToFloat(RHS,&FRHS);
    ConvertFloatToHalfFloat((FLHS-FRHS),&TOS);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
}
```

```
ConvertHalfFloatToBase10(TOS,base10TOS);
ConvertHalfFloatToBase10(LHS,base10LHS);
ConvertHalfFloatToBase10(RHS,base10RHS);
sprintf(information," %s = %s - %s",base10TOS,base10LHS,base10RHS);
strcat(traceLine,information);

}

break;

// 0X24      MULI          OpCode           Pop RHS,LHS; push integer ( LHS*RHS )
case 0X24:
    strcat(traceLine,"MULI      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = UNSIGNED(SIGNED(LHS)*SIGNED(RHS));
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX * 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;

// 0X25      MULF          OpCode           Pop RHS,LHS; push float ( LHS*RHS )
case 0X25:
{
    float FLHS,FRHS;
    char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

    strcat(traceLine,"MULF      ");
```

```

ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

ConvertHalfFloatToFloat(LHS,&FLHS);

ConvertHalfFloatToFloat(RHS,&FRHS);

ConvertFloatToHalfFloat((FLHS*FRHS),&TOS);

WriteWORDToMainMemory(SP,TOS); SP -= 2;

ConvertHalfFloatToBase10(TOS,base10TOS);

ConvertHalfFloatToBase10(LHS,base10LHS);

ConvertHalfFloatToBase10(RHS,base10RHS);

sprintf(information," %s = %s * %s",base10TOS,base10LHS,base10RHS);

strcat(traceLine,information);

}

break;

// 0X26      DIVI          OpCode           Pop RHS,LHS; push integer ( LHS÷RHS )

case 0X26:

    strcat(traceLine,"DIVI      ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

    TOS = UNSIGNED(SIGNED(LHS)/SIGNED(RHS));

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    sprintf(information," 0X%04hX = 0X%04hX / 0X%04hX",TOS,LHS,RHS);

    strcat(traceLine,information);

    break;

// 0X27      DIVF          OpCode           Pop RHS,LHS; push float ( LHS÷RHS )

```

```

case 0X27:
{
    float FLHS,FRHS;
    char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

    strcat(traceLine,"DIVF      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ConvertHalfFloatToFloat(LHS,&FLHS);
    ConvertHalfFloatToFloat(RHS,&FRHS);
    ConvertFloatToHalfFloat((FLHS/FRHS),&TOS);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    ConvertHalfFloatToBase10(TOS,base10TOS);
    ConvertHalfFloatToBase10(LHS,base10LHS);
    ConvertHalfFloatToBase10(RHS,base10RHS);
    sprintf(information, " %s = %s / %s",base10TOS,base10LHS,base10RHS);
    strcat(traceLine,information);
}
break;

// 0X28      REMI          OpCode          Pop RHS,LHS; push integer ( LHS rem RHS )
case 0X28:
{
    strcat(traceLine,"REMI      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = UNSIGNED(SIGNED(LHS)%SIGNED(RHS));
}

```

```
WriteWORDToMainMemory(SP,TOS); SP -= 2;

sprintf(information," 0X%04hX = 0X%04hX % 0X%04hX",TOS,LHS,RHS);
strcat(traceLine,information);

break;

// 0X29      POWI          OpCode           Pop RHS,LHS; push integer pow(LHS,RHS)
case 0X29:

    strcat(traceLine,"POWI      ");

    {

        int p,i;

        ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
        ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

        if ( SIGNED(RHS) < 0 )
            p = 0;
        else
        {
            p = 1;
            for (i = 1; i <= SIGNED(RHS); i++)
                p = p*SIGNED(LHS);
        }
        TOS = UNSIGNED(p);

    }

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    sprintf(information," 0X%04hX = 0X%04hX ^ 0X%04hX",TOS,LHS,RHS);
```

```
        strcat(traceLine,information);

        break;

// 0X2A      POWF          OpCode          Pop RHS,LHS; push float pow(LHS,RHS)

case 0X2A:

{

    float FLHS,FRHS;

    char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

    strcat(traceLine,"POWF      ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

    ConvertHalfFloatToFloat(LHS,&FLHS);

    ConvertHalfFloatToFloat(RHS,&FRHS);

    ConvertFloatToHalfFloat((float) pow(FLHS,FRHS),&TOS);

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    ConvertHalfFloatToBase10(TOS,base10TOS);

    ConvertHalfFloatToBase10(LHS,base10LHS);

    ConvertHalfFloatToBase10(RHS,base10RHS);

    sprintf(information, " %s = %s ^ %s",base10TOS,base10LHS,base10RHS);

    strcat(traceLine,information);

}

        break;

// 0X2B      NEGI          OpCode          Pop RHS; push integer -RHS

case 0X2B:
```

```
    strcat(traceLine,"NEGI      ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

    TOS = UNSIGNED(-SIGNED(RHS));

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    sprintf(information," 0X%04hX = -(0X%04hX)",TOS,RHS);

    strcat(traceLine,information);

    break;

// 0X2C      NEGF          OpCode           Pop RHS; push  float -RHS
case 0X2C:
{
    float FRHS;
    char base10TOS[80+1],base10RHS[80+1];

    strcat(traceLine,"NEGF      ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

    ConvertHalfFloatToFloat(RHS,&FRHS);

    ConvertFloatToHalfFloat(-FRHS,&TOS);

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    ConvertHalfFloatToBase10(TOS,base10TOS);

    ConvertHalfFloatToBase10(RHS,base10RHS);

    sprintf(information," %s = -(%s)",base10TOS,base10RHS);

    strcat(traceLine,information);

}
break;
```

```

// 0X2D      AND           OpCode          Pop RHS,LHS; push boolean ( LHS and RHS )
case 0X2D:
    strcat(traceLine,"AND      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( (LHS == 0xFFFFu) && (RHS == 0xFFFFu) )
        TOS = 0xFFFFu;
    else
        TOS = 0X0000u;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," %c = %c AND %c",TORF(TOS),TORF(LHS),TORF(RHS));
    strcat(traceLine,information);
    break;

// 0X2E      NAND          OpCode          Pop RHS,LHS; push boolean ( LHS nand RHS )
case 0X2E:
    strcat(traceLine,"NAND      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( (LHS == 0xFFFFu) && (RHS == 0xFFFFu) )
        TOS = 0X0000u;
    else
        TOS = 0xFFFFu;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," %c = %c NAND %c",TORF(TOS),TORF(LHS),TORF(RHS));
    strcat(traceLine,information);

```

```
break;

// 0X2F      OR           OpCode          Pop RHS,LHS; push boolean ( LHS or RHS )
case 0X2F:
    strcat(traceLine,"OR      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( (LHS == 0X0000u) && (RHS == 0X0000u) )
        TOS = 0X0000u;
    else
        TOS = 0xFFFFu;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," %c = %c OR %c",TORF(TOS),TORF(LHS),TORF(RHS));
    strcat(traceLine,information);
    break;

// 0X30      NOR          OpCode          Pop RHS,LHS; push boolean ( LHS nor RHS )
case 0X30:
    strcat(traceLine,"NOR      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( (LHS == 0X0000u) && (RHS == 0X0000u) )
        TOS = 0xFFFFu;
    else
        TOS = 0X0000u;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
```

```

sprintf(information," %c = %c NOR %c",TORF(TOS),TORF(LHS),TORF(RHS));
strcat(traceLine,information);
break;

// 0X31      XOR           OpCode          Pop RHS,LHS; push boolean ( LHS xor RHS )
case 0X31:
    strcat(traceLine,"XOR      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( ((LHS == 0xFFFFFu) && (RHS == 0X0000u)) || ((LHS == 0X0000u) && (RHS == 0xFFFFFu)) )
        TOS = 0xFFFFFu;
    else
        TOS = 0X0000u;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," %c = %c XOR %c",TORF(TOS),TORF(LHS),TORF(RHS));
    strcat(traceLine,information);
    break;

// 0X32      NXOR          OpCode          Pop RHS,LHS; push boolean ( LHS nxor RHS )
case 0X32:
    strcat(traceLine,"NXOR      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    if ( ((LHS == 0xFFFFFu) && (RHS == 0X0000u)) || ((LHS == 0X0000u) && (RHS == 0xFFFFFu)) )
        TOS = 0X0000u;
    else

```

```

TOS = 0xFFFFFu;

WriteWORDToMainMemory(SP,TOS); SP -= 2;
sprintf(information," %c = %c NXOR %c",TORF(TOS),TORF(LHS),TORF(RHS));
strcat(traceLine,information);
break;

// 0X33    NOT           OpCode          Pop RHS; push boolean ( not RHS )
case 0X33:
    strcat(traceLine,"NOT      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    if ( RHS == 0X0000u )
        TOS = 0xFFFFFu;
    else
        TOS = 0X0000u;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," %c = NOT %c",TORF(TOS),TORF(RHS));
    strcat(traceLine,information);
    break;

// 0X34    BITAND         OpCode          Pop RHS,LHS; push boolean ( LHS bitwise-AND RHS )
case 0X34:
    strcat(traceLine,"BITAND   ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = LHS&RHS;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;

```

```
sprintf(information," 0X%04hX = 0X%04hX bitwise-AND 0X%04hX",TOS,LHS,RHS);
strcat(traceLine,information);
break;

// 0X35      BITNAND          OpCode           Pop RHS,LHS; push boolean ( LHS bitwise-NAND RHS )
case 0X35:
    strcat(traceLine,"BITNAND  ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = ~(LHS&RHS);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX bitwise-NAND 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;

// 0X36      BITOR           OpCode           Pop RHS,LHS; push boolean ( LHS bitwise-OR   RHS )
case 0X36:
    strcat(traceLine,"BITOR   ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = LHS|RHS;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX bitwise-OR 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;
```

```
// 0X37      BITNOR           OpCode          Pop RHS,LHS; push boolean ( LHS bitwise-NOR  RHS )
case 0X37:
    strcat(traceLine,"BITNOR   ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = ~(LHS|RHS);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX bitwise-NOR 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;

// 0X38      BITXOR           OpCode          Pop RHS,LHS; push boolean ( LHS bitwise-XOR   RHS )
case 0X38:
    strcat(traceLine,"BITXOR   ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = LHS^RHS;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX bitwise-XOR 0X%04hX",TOS,LHS,RHS);
    strcat(traceLine,information);
    break;

// 0X39      BITNXOR          OpCode          Pop RHS,LHS; push boolean ( LHS bitwise-NXOR RHS )
case 0X39:
    strcat(traceLine,"BITNXOR  ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
```

```

ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

TOS = ~(LHS^RHS);

WriteWORDToMainMemory(SP,TOS); SP -= 2;

sprintf(information," 0X%04hX = 0X%04hX bitwise-NXOR 0X%04hX",TOS,LHS,RHS);

strcat(traceLine,information);

break;

// 0X3A      BITNOT          OpCode           Pop RHS; push boolean ( bitwise-NOT RHS )

case 0X3A:

    strcat(traceLine,"BITNOT    ");

    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;

    TOS = ~RHS;

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    sprintf(information," 0X%04hX = bitwise-NOT 0X%04hX",TOS,RHS);

    strcat(traceLine,information);

    break;

// 0X3B      BITSL     #W16      OpCode:016      Pop LHS; push ( LHS shifted-left 016U bits ) )

case 0X3B:

    strcat(traceLine,"BITSL    ");

    ReadWORDFromMainMemory(PC,&016); PC += 2;

    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;

    TOS = LHS << 016;

    WriteWORDToMainMemory(SP,TOS); SP -= 2;

    sprintf(information," 0X%04hX = 0X%04hX SL %5hd",TOS,RHS,016);

    strcat(traceLine,information);

```

```

break;

// 0X3C    BITLSR  #W16      OpCode:016      Pop LHS; push ( LHS logically shifted-right 016U bits      )

case 0X3C:
    strcat(traceLine,"BITLSR    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = LHS >> 016;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX SL %5hd",TOS,RHS,016);
    strcat(traceLine,information);
    break;

// 0X3D    BITASR  #W16      OpCode:016      Pop LHS; push ( LHS arithmetically shifted-right 016U bits )

case 0X3D:
    strcat(traceLine,"BITASR    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    TOS = UNSIGNED(SIGNED(LHS) >> 016);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," 0X%04hX = 0X%04hX SL %5hd",TOS,RHS,016);
    strcat(traceLine,information);
    break;

// 0X60    CTOF      OpCode          Pop integer RHS; push   float RHS (integer-to-float)

case 0X60:

```

```

{
    char base10TOS[80+1];

    strcat(traceLine,"CITOF      ");
    ReadWORDFromMainMemory(SP+2,&TOS); SP += 2;
    ConvertFloatToHalfFloat((float) SIGNED(TOS),&TOS);
    ConvertHalfFloatToBase10(TOS,base10TOS);
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," TOS = %s",base10TOS);
    strcat(traceLine,information);

}

break;

// 0X61      CFTOI          OpCode          Pop   float RHS; push integer RHS (float-to-integer)
case 0X61:
{
    float FTOS;
    char base10TOS[80+1];

    strcat(traceLine,"CFTOI      ");
    ReadWORDFromMainMemory(SP+2,&TOS); SP += 2;
    ConvertHalfFloatToFloat(TOS,&FTOS);
    TOS = (WORD) FTOS;
    WriteWORDToMainMemory(SP,TOS); SP -= 2;
    sprintf(information," TOS = 0X%04hX",TOS);
    strcat(traceLine,information);
}

```

```

}

break;

// 0X70      CMPI          OpCode           Pop RHS,LHS; set LEG in FLAGS based on ( LHS ? RHS ) (integer)
case 0X70:
    strcat(traceLine,"CMPI      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    L = (SIGNED(LHS) < SIGNED(RHS)) ? 1 : 0;
    E = (SIGNED(LHS) == SIGNED(RHS)) ? 1 : 0;
    G = (SIGNED(LHS) > SIGNED(RHS)) ? 1 : 0;
    sprintf(information," 0X%04hX ? 0X%04hX LEG = %d%d%d",LHS,RHS,L,E,G);
    strcat(traceLine,information);
    break;

// 0X71      CMPF          OpCode           Pop RHS,LHS; set LEG in FLAGS based on ( LHS ? RHS ) (float)
case 0X71:
{
    float FLHS,FRHS;
    char base10TOS[80+1],base10LHS[80+1],base10RHS[80+1];

    strcat(traceLine,"CMPF      ");
    ReadWORDFromMainMemory(SP+2,&RHS); SP += 2;
    ReadWORDFromMainMemory(SP+2,&LHS); SP += 2;
    ConvertHalfFloatToFloat(LHS,&FLHS);
    ConvertHalfFloatToFloat(RHS,&FRHS);
}

```

```

L = (FLHS < FRHS) ? 1 : 0;
E = (FLHS == FRHS) ? 1 : 0;
G = (FLHS > FRHS) ? 1 : 0;
ConvertHalfFloatToBase10(LHS,base10LHS);
ConvertHalfFloatToBase10(RHS,base10RHS);
sprintf(information," %s ? %s LEG = %d%d%d",base10LHS,base10RHS,L,E,G);
strcat(traceLine,information);

}

break;

// 0X72      SETNZPI          OpCode           Set NZP in FLAGS based on sign of TOS (integer)
case 0X72:
    strcat(traceLine,"SETNZPI  ");
    ReadWORDFromMainMemory(SP+2,&TOS);
    N = (SIGNED(TOS) < 0) ? 1 : 0;
    Z = (SIGNED(TOS) == 0) ? 1 : 0;
    P = (SIGNED(TOS) > 0) ? 1 : 0;
    sprintf(information," TOS = 0X%04hX NZP = %d%d%d",TOS,N,Z,P);
    strcat(traceLine,information);
    break;

// 0X73      SETNZPF          OpCode           Set NZP in FLAGS based on sign of TOS (float)
case 0X73:
{
    float FTOS;
    char base10TOS[80+1];

```

```

        strcat(traceLine,"SETNZPF  ");
        ReadWORDFromMainMemory(SP+2,&TOS);
        ConvertHalfFloatToFloat(TOS,&FTOS);
        N = (FTOS < 0.0) ? 1 : 0;
        Z = (FTOS == 0.0) ? 1 : 0;
        P = (FTOS > 0.0) ? 1 : 0;
        ConvertHalfFloatToBase10(TOS,base10TOS);
        sprintf(information," TOS = %s NZP = %d%d%d",base10TOS,N,Z,P);
        strcat(traceLine,information);
    }
    break;

// 0X74      SETT          OpCode           Set T in FLAGS based on true/false value of TOS (boolean)
case 0X74:
    strcat(traceLine,"SETT      ");
    ReadWORDFromMainMemory(SP+2,&TOS);
    if ( TOS == 0xFFFFu )
        T = 1;
    else
        T = 0;
    sprintf(information," T = %d",T);
    strcat(traceLine,information);
    break;

// 0X80      JMP      A16          OpCode:016          PC <- 016U

```

```
case 0X80:
    strcat(traceLine,"JMP      ");
    ReadWORDFromMainMemory(PC,&016);
    sprintf(information," 0X%04hX",016);
    strcat(traceLine,information);
    PC = 016;
    break;

// 0X81    JMPL    A16          OpCode:016      if (      L ) PC <- 016U
case 0X81:
    strcat(traceLine,"JMPL      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
    strcat(traceLine,information);
    if ( L == 1 )
        PC = 016;
    break;

// 0X82    JMPE    A16          OpCode:016      if (      E ) PC <- 016U
case 0X82:
    strcat(traceLine,"JMPE      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
    strcat(traceLine,information);
    if ( E == 1 )
        PC = 016;
```

```
break;

// 0X83    JMPG    A16          OpCode:016      if (      G ) PC <- 016U
case 0X83:
    strcat(traceLine,"JMPG      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
    strcat(traceLine,information);
    if ( G == 1 )
        PC = 016;
    break;

// 0X84    JMPL E   A16          OpCode:016      if ( L or E ) PC <- 016U
case 0X84:
    strcat(traceLine,"JMPL E     ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
    strcat(traceLine,information);
    if ( (L == 1) || (E == 1) )
        PC = 016;
    break;

// 0X85    JMPNE   A16          OpCode:016      if ( L or G ) PC <- 016U (JMPLG)
case 0X85:
    strcat(traceLine,"JMPNE   ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
```

```
sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
strcat(traceLine,information);
if ( !(E == 1) )
    PC = 016;
break;

// 0X86    JMPGE   A16          OpCode:016      if ( G or E ) PC <- 016U
case 0X86:
    strcat(traceLine,"JMPGE    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX LEG = %d%d%d",016,L,E,G);
    strcat(traceLine,information);
    if ( (G == 1) || (E == 1) )
        PC = 016;
    break;

// 0X87    JMPN    A16          OpCode:016      if (      N ) PC <- 016U
case 0X87:
    strcat(traceLine,"JMPN    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
    if ( N == 1 )
        PC = 016;
    break;
```

```
// 0X88    JMPNN   A16          OpCode:016      if (  not N ) PC <- 016U
case 0X88:
    strcat(traceLine,"JMPNN    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
    if ( !(N == 1) )
        PC = 016;
    break;

// 0X89    JMPZ    A16          OpCode:016      if (      Z ) PC <- 016U
case 0X89:
    strcat(traceLine,"JMPZ    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
    if ( Z == 1 )
        PC = 016;
    break;

// 0X8A    JMPNZ   A16          OpCode:016      if (  not Z ) PC <- 016U
case 0X8A:
    strcat(traceLine,"JMPNZ    ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
```

```
if ( !(Z == 1) )
    PC = 016;
break;

// 0X8B    JMPP    A16          OpCode:016      if (      P ) PC <- 016U
case 0X8B:
    strcat(traceLine,"JMPP      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
    if ( P == 1 )
        PC = 016;
    break;

// 0X8C    JMPNP   A16          OpCode:016      if (  not P ) PC <- 016U
case 0X8C:
    strcat(traceLine,"JMPNP      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information," 0X%04hX NZP = %d%d%d",016,N,Z,P);
    strcat(traceLine,information);
    if ( !(P == 1) )
        PC = 016;
    break;

// 0X8D    JMPT    A16          OpCode:016      if (      T ) PC <- 016U
case 0X8D:
```

```
        strcat(traceLine,"JMPT      ");

        ReadWORDFromMainMemory(PC,&016); PC += 2;

        sprintf(information," 0X%04hX T = %d",016,T);

        strcat(traceLine,information);

        if ( T == 1 )

            PC = 016;

        break;

// 0X8E    JMPNT   A16          OpCode:016           if (  not T ) PC <- 016U (JMPF)

case 0X8E:

        strcat(traceLine,"JMPNT      ");

        ReadWORDFromMainMemory(PC,&016); PC += 2;

        sprintf(information," 0X%04hX T = %d",016,T);

        strcat(traceLine,information);

        if ( !(T == 1) )

            PC = 016;

        break;

// 0XA0    CALL    A16          OpCode:016           Push PC; PC <- 016U

case 0XA0:

        strcat(traceLine,"CALL      ");

        ReadWORDFromMainMemory(PC,&016); PC += 2;

        WriteWORDToMainMemory(SP,PC); SP -= 2;

        sprintf(information," 0X%04hX return to 0X%04hX",016,PC);

        strcat(traceLine,information);

        PC = 016;
```

```

break;

// 0XA1      RETURN          OpCode          Pop PC

case 0XA1:
    strcat(traceLine,"RETURN    ");
    ReadWORDFromMainMemory(SP+2,&PC); SP += 2;
    sprintf(information," to 0X%04hX",PC);
    strcat(traceLine,information);
    break;

/*
=====
STMOS Service Requests
=====

#      Description          Parameters
---  -----
 0  Force context switch   (none) Do nothing
 1  Terminate process     Pop termination status
10  Read integer          Input W16 as integer; push W16
11  Write integer         Pop W16; output W16 as integer
12  Write hexadeciml integer (*added*) Pop W16; output W16 as hexadecimal integer
20  Read float             Input W16 as float; push W16
21  Write float            Pop W16; output W16 as float
30  Read boolean           Input W16 as boolean; push W16 { 't','T','f','F' }
31  Write boolean          Pop W16; output W16 as boolean { 'T','F' }
40  Read character         Input W16 as character; push W16
41  Write character        Pop W16; output W16 as character

```

```

42 Write ENDL character      (none) Output ENDL (end-of-line) character
50 Read string               Pop A16 as address of string; input string into A16
51 Write string              Pop A16 as address of string; output string from A16
60 Create new file           Pop A16 as address of the file name; create new file
61 Set file read access     Pop W16 as boolean; set read access to 'T' or 'F'.
90 Initialize heap            Pop heapSize and heapBase; initialize heap
91 Allocate heap block        Pop blockSize; allocate block; push blockAddress
92 Deallocate heap block      Pop blockAddress; deallocate block
*/
// 0xFF    SVC #W16          OpCode:016      Execute service request 016U (parameters are passed on run-time stack)
case 0xFF:
    strcat(traceLine,"SVC      ");
    ReadWORDFromMainMemory(PC,&016); PC += 2;
    sprintf(information,"%#hd",016);
    strcat(traceLine,information);
    switch ( 016 )
    {
        case  0: //  0   Force context switch      (none) Do nothing
            strcat(traceLine," force context switch");
            break;
        case  1: //  1   Terminate process          Pop termination status
            if ( strlen(OUT) > 0 )
            {
                fprintf(LOG,"%s\n",OUT);
                printf("%s\n",OUT);
            }
    }
}

```

```
ReadWORDFromMainMemory(SP+2,&W16); SP += 2;
sprintf(information," terminate program with status %hd, SP = 0X%04hX\n",W16,SP);
strcat(traceLine,information);
running = false;
break;

case 10: // 10      Read integer           Input W16 as integer; push W16
{
    if ( strlen(OUT) > 0 )
    {
        printf("%s",OUT);
        scanf("%hu",&W16);
        fprintf(LOG,"%s%hd\n",OUT,W16);
        OUT[0] = '\0';
    }
    else
    {
        printf("?");
        scanf("%hu",&W16);
        fprintf(LOG,"? %hd\n",W16);
    }
    WriteWORDToMainMemory(SP,W16); SP -= 2;
    sprintf(information," read integer 0X%04hX",W16);
    strcat(traceLine,information);
    break;

case 11: // 11      Write integer          Pop W16; output W16 as integer
{
    char datum[SOURCELINELENGTH+1];
```

```
    ReadWORDFromMainMemory(SP+2,&W16); SP += 2;
    sprintf(datum,"%hd",W16);
    strcat(OUT,datum);
}
strcat(traceLine," write integer");
break;

case 12: // 12 Write hexadeciml integer /*ADDED*/ Pop W16; output W16 as hexadecimal integer
{
    char datum[SOURCELINELENGTH + 1];

    ReadWORDFromMainMemory(SP + 2, &W16); SP += 2;
    sprintf(datum, "0x%04x", W16);
    strcat(OUT, datum);
}
strcat(traceLine, " write hexadeciml integer");
break;

case 20: // 20     Read float                               Input W16 as float; push W16
{
    float item;
    char base10W16[80+1];

    if ( strlen(OUT) > 0 )
    {
        printf("%s",OUT);
        scanf("%f",&item);
    }
}
```

```
    ConvertFloatToHalfFloat(item,&W16);

    ConvertHalfFloatToBase10(W16,base10W16);

    fprintf(LOG,"%s%s (approximation)\n",OUT,base10W16);

    OUT[0] = '\0';

}

else

{

    printf("? ");

    scanf("%f",&item);

    ConvertFloatToHalfFloat(item,&W16);

    ConvertHalfFloatToBase10(W16,base10W16);

    fprintf(LOG,"? %s (approximation)\n",base10W16);

}

WriteWORDToMainMemory(SP,W16); SP -= 2;

sprintf(information," read float %s",base10W16);

strcat(traceLine,information);

}

break;

case 21: // 21    Write float          Pop W16; output W16 as float

{

    char datum[SOURCELINELENGTH+1];

    ReadWORDFromMainMemory(SP+2,&W16); SP += 2;

    ConvertHalfFloatToBase10(W16,datum);

    strcat(OUT,datum);

}
```

```
    strcat(traceLine," write float");

    break;

case 30: // 30      Read boolean                         Input W16 as boolean; push W16 { 't','T','f','F' }

    if ( strlen(OUT) > 0 )

    {

        printf("%s",OUT);

        scanf(" %c",&IN[0]);

        fprintf(LOG,"%s%c\n",OUT,LOBYTE(IN[0]));

        OUT[0] = '\0';

    }

    else

    {

        printf("? ");

        scanf(" %c",&IN[0]);

        fprintf(LOG,"? %c\n",LOBYTE(IN[0]));

    }

    if      ( toupper(IN[0]) == 'T' )

        W16 = 0xFFFFu;

    else if ( toupper(IN[0]) == 'F' )

        W16 = 0x0000u;

    else

    {

        ProcessRunTimeError("Boolean must be in { t,T,f,F }",false);

        W16 = 0x0000u;

    }

    WriteWORDToMainMemory(SP,W16); SP -= 2;
```

```
sprintf(information," read boolean 0X%04hX",W16);
strcat(traceLine,information);
break;

case 31: // 31      Write boolean                               Pop W16; output W16 as boolean { 'T','F' }

{
    char datum[SOURCELINELENGTH+1];

    ReadWORDFromMainMemory(SP+2,&W16); SP += 2;
    sprintf(datum,"%c",((W16 == 0xFFFFu) ? 'T' : 'F'));
    strcat(OUT,datum);

}

sprintf(information," write boolean 0X%04hX",W16);
strcat(traceLine,information);
break;

case 40: // 40      Read character                            Input W16 as character; push W16

if ( strlen(OUT) > 0 )
{
    printf("%s",OUT);
    scanf(" %c",&IN[0]);
    fprintf(LOG,"%s%c\n",OUT,LOBYTE(IN[0]));
    OUT[0] = '\0';
}

else
{
    printf("? ");
    scanf(" %c",&IN[0]);
}
```

```
    fprintf(LOG,"? %c\n",LOBYTE(IN[0]));

}

W16 = LOBYTE(IN[0]);

WriteWORDToMainMemory(SP,W16); SP -= 2;

sprintf(information," read character 0X%04hX = '%c'",W16,LOBYTE(W16));

strcat(traceLine,information);

break;

case 41: // 41      Write character                  Pop W16, output W16 as character

{

const BYTE LF = 0X0Au;

const BYTE CR = 0X0Du;

char datum[SOURCELINELENGTH+1];

ReadWORDFromMainMemory(SP+2,&W16); SP += 2;

if ( (LOBYTE(W16) == LF) || (LOBYTE(W16) == CR) )

{

    fprintf(LOG,"%s\n",OUT);

    printf("%s\n",OUT);

    OUT[0] = '\0';

    sprintf(information," write character 0X%04hX",W16);

    strcat(traceLine,information);

}

else

{

    sprintf(datum,"%c",LOBYTE(W16));

}
```

```
        strcat(OUT,datum);

        sprintf(information," write character 0X%04hX = '%c'",W16,LOBYTE(W16));
        strcat(traceLine,information);

    }

}

break;

case 42: // 42      Write ENDL character          (none) Output ENDL (end-of-line) character
{
    char datum[SOURCELINELENGTH+1];

    fprintf(LOG,"%s\n",OUT);
    printf("%s\n",OUT);
    OUT[0] = '\0';

}

strcat(traceLine," write ENDL");

break;

case 50: // 50      Read string                  Pop A16 as address of string; input string into A16
{
    int i;

    WORD A16,length,capacity;

    // "flush" stdin
    while ( getc(stdin) != '\n' );
    ReadWORDFromMainMemory(SP+2,&A16); SP += 2;
    if ( strlen(OUT) > 0 )
    {
```

```
    printf("%s",OUT);

    gets(IN);

    fprintf(LOG,"%s%s",OUT,IN);

    OUT[0] = '\0';

}

else

{

    printf("? ");

    gets(IN);

    fprintf(LOG,"? %s",IN);

}

// Ensure there *IS* something to "flush" when 2 or more string inputs occur in a row ***KLUDGE***

ungetc('\n',stdin);

ReadWORDFromMainMemory(A16+2,&capacity);

if (strlen(IN) <= capacity)

{

    length = strlen(IN);

    fprintf(LOG,"\n");

}

else

{

    length = capacity;

    fprintf(LOG," (truncated) \n");

}

WriteWORDToMainMemory(A16,length);

for (i = 1; i <= length; i++)
```

```
        WriteWORDToMainMemory(A16+4+(i-1)*2,(WORD) IN[i-1]);  
    }  
    break;  
  
case 51: // Write string  
{  
    const BYTE NUL = 0X00u;  
    const BYTE LF = 0X0Au;  
    const BYTE CR = 0X0Du;  
  
    int i;  
    char datum[SOURCELINELENGTH+1];  
    WORD A16,length,capacity;  
  
    ReadWORDFromMainMemory(SP+2,&A16); SP += 2;  
    ReadWORDFromMainMemory(A16,&length);  
    ReadWORDFromMainMemory(A16+2,&capacity);  
    i = 1;  
    while ( i <= length )  
    {  
        ReadWORDFromMainMemory(A16+4+(i-1)*2,&W16);  
        i += 1;  
        // substitute for legal escape sequences  
        if ( LOBYTE(W16) == '\\\\' )  
        {  
            ReadWORDFromMainMemory(A16+4+(i-1)*2,&W16);  
            i += 1;
```

```
switch ( LOBYTE(W16) )
{
    case 'n': // LF (newline)
        W16 = LF;
        break;

    case 't': // HT (horizontal tab)
        W16 = 0X09u;
        break;

    case 'v': // VT (vertical tab)
        W16 = 0X0Bu;
        break;

    case 'b': // BS (backspace)
        W16 = 0X08u;
        break;

    case 'r': // CR (carriage return)
        W16 = CR;
        break;

    case 'a': // BEL (alert)
        W16 = 0X07u;
        break;

    case '\\': // \ (backslash)
        W16 = '\\';
        break;

    case '\"': // " (double quote)
        W16 = '"';
        break;
}
```

```
        default:
            W16 = '?';
            RecordSyntaxError("Illegal escape character sequence\n");
        }
    }

    if ( (LOBYTE(W16) == LF) || (LOBYTE(W16) == CR) )
    {
        fprintf(LOG,"%s\n",OUT);
        printf("%s\n",OUT);
        OUT[0] = '\0';
    }
    else
    {
        sprintf(datum,"%c",LOBYTE(W16));
        strcat(OUT,datum);
    }
}

strcat(traceLine," write string");
break;

case 60: // Create a new file /* ADDED */           Pop A16 as address of the file name; create new file
{
    // Copied from case 51 (STM_WRITE_STRING)
```

```
const BYTE NUL = 0X00u;
const BYTE LF = 0X0Au;
const BYTE CR = 0X0Du;

int i;
char filename[SOURCELINELENGTH + 1];
char datum[SOURCELINELENGTH + 1];
WORD A16, length, capacity;

ReadWORDFromMainMemory(SP + 2, &A16); SP += 2;
ReadWORDFromMainMemory(A16, &length);
ReadWORDFromMainMemory(A16 + 2, &capacity);

filename[0] = '\0';
i = 1;
while (i <= length)
{
    ReadWORDFromMainMemory(A16 + 4 + (i - 1) * 2, &W16);
    i += 1;
    // substitute for legal escape sequences
    if (LOBYTE(W16) == '\\')
    {
        ReadWORDFromMainMemory(A16 + 4 + (i - 1) * 2, &W16);
        i += 1;
        switch (LOBYTE(W16))
        {
```

```
case 'n': // LF (newline)
    W16 = LF;
    break;
case 't': // HT (horizontal tab)
    W16 = 0X09u;
    break;
case 'v': // VT (vertical tab)
    W16 = 0X0Bu;
    break;
case 'b': // BS (backspace)
    W16 = 0X08u;
    break;
case 'r': // CR (carriage return)
    W16 = CR;
    break;
case 'a': // BEL (alert)
    W16 = 0X07u;
    break;
case '\\': // \ (backslash)
    W16 = '\\';
    break;
case '\"': // " (double quote)
    W16 = '"';
    break;
default:
    W16 = '?';
```

```
        RecordSyntaxError("Illegal escape character sequence\n");
    }

}

/*
if ((LOBYTE(W16) == LF) || (LOBYTE(W16) == CR))
{
    fprintf(LOG, "%s\n", OUT);
    printf("%s\n", OUT);
    OUT[0] = '\0';
}
else
{
    //sprintf(datum, "%c", LOBYTE(W16));
    //strcat(OUT, datum);

}

*/
sprintf(datum, "%c", LOBYTE(W16));
strcat(filename, datum);
}

// Create the file as "write-only" (Note: Rogue will not be able to read the file just yet.)
FILE* file;

strcat(filename, ".txt");
```

```
file = fopen(filename, "w");

if (file == NULL)
{
    printf("Failed to open %s", filename);
}

fclose(file);
}

// *add to traceLine here*
strcat(traceLine, " create file");
break;

case 61: // Set file read access /***WORK IN PROGRESS***/           Pop W16 as boolean; set read access to 'T' or 'F'.
{
    char datum[SOURCELINELENGTH + 1];

    ReadWORDFromMainMemory(SP + 2, &W16); SP += 2;

}

/*** add traceline information ***/
break;

/*
FREEnodes is a pointer to a singly-linked list of nodes representing contiguous blocks of
```

heap space that are "free." The FREEnodes list is initialized with SVC #90 to contain a single, large node which represents a block consisting of *ALL* allocateable heap space. After a series of SVC #91 (allocate block) and SVC #92 (deallocate block) service requests, the FREEnodes list will consist of (1) the remnants of the original, large "free" node; and (2) nodes representing deallocated blocks which cannot be "absorbed by" (are not on the "edge" of) the free space of the other free nodes.

The strategy "find-first-FREE-node-that-fits-the-requested-blockSize" is used to satisfy SVC #91 requests.

The FREEnodes list is maintained in order of increasing address of the node.

```

structure FREENODE
    // metadata (offset from beginning of node measured in bytes)
    (0) WORD FLink          address of logically-next node in list
    (2) WORD blockSize      measured in bytes
    // block of allocated heap space
    (4) BYTE block[1:blockSize] *NOTE* blockAddress is address-of block[1]
endstructure

/*
case 90: // 90 Initialize heap          Pop heapSize and heapBase; initialize heap
{
    // Pop headSize and heapBase
    ReadWORDFromMainMemory(SP+2,&heapSize); SP += 2;
    ReadWORDFromMainMemory(SP+2,&heapBase); SP += 2;
    // Create one large node FLink = 0X0000 and (blockSize = heapSize-4) and initialize the FREE node list
    FREEnodes = heapBase;
    WriteWORDToMainMemory(FREEnodes+0,0X0000u);
    WriteWORDToMainMemory(FREEnodes+2,heapSize-4);
}

```

```
sprintf(information," initialize heap, heapBase = 0X%04hX, heapSize = 0X%04hX words",heapBase,heapSize);
strcat(traceLine,information);

TraceFREEnodes(FREEnodes);

}

break;

case 91: // 91  Allocate heap block          Pop blockSize; allocate block; push blockAddress
{
    WORD allocateNodeAddress,blockSize,blockAddress;
    WORD nodeAddress,nodeFLink,nodeBlockSize;
    bool nodeFound;

    // Pop blockSize
    ReadWORDFromMainMemory(SP+2,&blockSize); SP += 2;
    // Use first-fit algorithm to find a nodeBlockSize >= (blockSize+4)
    nodeAddress = FREEnodes;
    nodeFound = false;
    do
    {
        ReadWORDFromMainMemory(nodeAddress+0,&nodeFLink);
        ReadWORDFromMainMemory(nodeAddress+2,&nodeBlockSize);
        if ( nodeBlockSize >= (blockSize+4) )
            nodeFound = true;
        else
            nodeAddress = nodeFLink;
    } while ( !nodeFound && (nodeFLink != 0X0000u) );
    if ( !nodeFound )
```

```
{  
    ProcessRunTimeError("Heap space exhausted",false);  
    // Push 0X0000 to indicate allocation request failed  
    WriteWORDToMainMemory(SP,0X0000u); SP -= 2;  
    sprintf(information," allocate heap block failed\n");  
}  
  
else  
{  
    // "Break-off" (blockSize+4) bytes at rear of "fitting" node's block  
    allocateNodeAddress = (nodeAddress+4+nodeBlockSize) - (blockSize+4);  
    WriteWORDToMainMemory(nodeAddress+2,nodeBlockSize-(blockSize+4));  
    // Set allocate node FLink = 0X0000 and blockSize  
    WriteWORDToMainMemory(allocateNodeAddress+0,0X0000u);  
    WriteWORDToMainMemory(allocateNodeAddress+2,blockSize);  
    // Push blockAddress  
    blockAddress = allocateNodeAddress+4;  
    WriteWORDToMainMemory(SP,blockAddress); SP -= 2;  
    sprintf(information," allocate heap block, block address = 0X%04hX, block size= 0X%04hX words",blockAddress,blockSize);  
}  
strcat(traceLine,information);  
TraceFREEnodes(FREEnodes);  
}  
break;  
  
case 92: // 92  Deallocate heap block          Pop blockAddress; deallocate block  
{  
    WORD nodeAddress,nodeFLink,nodeBlockSize;
```

```
WORD deallocateNodeAddress,blockSize,blockAddress;
bool nodeFound;

// Pop blockAddress
ReadWORDFromMainMemory(SP+2,&blockAddress); SP += 2;
deallocateNodeAddress = blockAddress-4;
ReadWORDFromMainMemory(deallocateNodeAddress+2,&blockSize);

// Find node which should precede the deallocate node
nodeAddress = FREEnodes;
nodeFound = false;
do
{
    ReadWORDFromMainMemory(nodeAddress+0,&nodeFLink);
    ReadWORDFromMainMemory(nodeAddress+2,&nodeBlockSize);
    if ( (nodeFLink == 0X0000u) || (nodeFLink > deallocateNodeAddress) )
        nodeFound = true;
    else
        nodeAddress = nodeFLink;
} while ( !nodeFound );

// If possible, "absorb" deallocate block into one or both of its surrounding FREE list nodes
if ( nodeAddress+4+nodeBlockSize == deallocateNodeAddress )
{
    nodeBlockSize += (blockSize+4);
    WriteWORDToMainMemory(nodeAddress+2,nodeBlockSize);
    if ( (nodeFLink != 0X0000u) && (nodeAddress+4+nodeBlockSize == nodeFLink) )
    {

```

```
WORD nextNodeFLink,nextNodeBlockSize;

ReadWORDFromMainMemory(nodeFLink+0,&nextNodeFLink);
ReadWORDFromMainMemory(nodeFLink+2,&nextNodeBlockSize);
nodeBlockSize += (nextNodeBlockSize+4);
WriteWORDToMainMemory(nodeAddress+0,nextNodeFLink);
WriteWORDToMainMemory(nodeAddress+2,nodeBlockSize);

}

}

// otherwise, insert deallocate block into FREE list after nodeAddress node

else

{

    WriteWORDToMainMemory(deallocateNodeAddress+0,nodeFLink);
    WriteWORDToMainMemory(nodeAddress+0,deallocateNodeAddress);

}

sprintf(information," deallocate heap block, block address = 0X%04hX, block size= 0X%04hX words",blockAddress,blockSize);
strcat(traceLine,information);
TraceFREEnodes(FREEnodes);

}

break;

default:

    strcat(traceLine," Invalid SVC #");

    ProcessRunTimeError("Invalid SVC #",false);

    break;

}

break;
```

```
// *UNKNOWN* opCode

default:
    strcat(traceLine,"???????  ");
    ProcessRunTimeError("Invalid opcode",false);
    break;
}

fprintf(LOG,"%s\n",traceLine);

} while ( running );

}

//-----
void TraceFREEnodes(WORD FREEnodes)
//-----
{
/*
-----
FREE nodes list
0X????:0X????(0X????)
0X????:0X????(0X????)
...
0X????:0X????(0X????)

*/
void ReadWORDFromMainMemory(int address,WORD *word);

WORD nodeAddress,nodeFLink,nodeBlockSize;
```

```

fprintf(LOG,"-----\n");
fprintf(LOG,"FREE nodes list\n");
nodeAddress = FREEnodes;
while ( nodeAddress != 0X0000u )
{
    ReadWORDFromMainMemory(nodeAddress+0,&nodeFLink);
    ReadWORDFromMainMemory(nodeAddress+2,&nodeBlockSize);
    fprintf(LOG, " 0X%04hX:0X%04hX(0X%04hX)\n",nodeAddress,(nodeAddress+4+nodeBlockSize-1),nodeBlockSize);
    nodeAddress = nodeFLink;
}
fprintf(LOG,"-----\n");
fflush(LOG);

}

//-----
WORD MemoryOperandEA(BYTE mode,WORD O16,WORD PC,WORD *SP,WORD FB,WORD SB,char information[])
//-----
{

void ReadWORDFromMainMemory(int address,WORD *word);

/*
The <memory> syntax (shown below) allows the programmer to specify a 16-bit STM
word value in 1 of 3 diffent ways

(1) a <A16> (which must be an <identifier>);

(2) as a <W16> (which can be <I16> (signed integer), <F16> (signed floating point),

```

```
true or false (boolean literals), <character> or an <identifier>;
(3) a <I16> (which is *always* interpreted as an unsigned integer)
```

this 16-bit value--regardless of the way it is written in assembly language--is referred to in the parameter list as 016.

```
<memory>      ::= #<W16>           || mode = 0X00, immediate
                | <A16>            || mode = 0X01, memory direct
                | @<A16>           || mode = 0X02, memory indirect
                | $<A16>           || mode = 0X03, memory indexed
                | SP:<I16>          || mode = 0X04, SP-relative direct
                | @SP:<I16>          || mode = 0X05, SP-relative indirect
                | $SP:<I16>          || mode = 0X06, SP-relative indexed
                | FB:<I16>           || mode = 0X07, FB-relative direct
                | @FB:<I16>          || mode = 0X08, FB-relative indirect
                | $FB:<I16>          || mode = 0X09, FB-relative indexed
                | SB:<I16>           || mode = 0X0A, SB-relative direct
                | @SB:<I16>          || mode = 0X0B, SB-relative indirect
                | $SB:<I16>          || mode = 0X0C, SB-relative indexed
```

*/

```
WORD EA,IEA,TOS;
```

```
switch ( mode )
{
    case 0X00: // EA = PC-2 ***ASSUMES PC IS ADDRESS OF NEXT OPCODE***
        EA = PC-2;
```

```
sprintf(information," #memory[EA = 0X%04hX]",EA);
break;

case 0X01: // EA = A16
EA = 016;
sprintf(information," memory[EA = 0X%04hX]",EA);
break;

case 0X02: // EA = mainMemory[ A16 ]
IEA = 016;
ReadWORDFromMainMemory(IEA,&EA);
sprintf(information," @memory[EA = 0X%04hX = memory[0X%04hX]]",EA,IEA);

case 0X03: // Pop TOS; EA = A16 + TOS
ReadWORDFromMainMemory(*SP+2,&TOS); *SP += 2;
EA = 016 + TOS;
sprintf(information," $0X%04hX+(%5hd) memory[EA = 0X%04hX]",016,TOS,EA);
break;

case 0X04: // EA = (SP+2)+2*I16
EA = (*SP+2) + 2*016;
sprintf(information," SP(%3hd) memory[EA = 0X%04hX]",016,EA);
break;

case 0X05: // EA = mainMemory[ (SP+2)+2*I16 ]
IEA = (*SP+2) + 2*016;
ReadWORDFromMainMemory(IEA,&EA);
sprintf(information," @SP(%3hd) memory[EA = 0X%04hX = memory[0X%04hX]]",016,EA,IEA);
break;

case 0X06: // Pop TOS; EA = (SP+2)+2*I16 + TOS
ReadWORDFromMainMemory(*SP+2,&TOS); *SP += 2;
```

```
EA = (*SP+2) + 2*016 + TOS;
sprintf(information," SP(%3hd)+(%5hd) memory[EA = 0X%04hX]",016,TOS,EA);
break;

case 0X07: // EA = FB-2*I16 ***NOTICE SUBTRACTION***
EA = FB - 2*016;
sprintf(information," FB(%3hd) memory[EA = 0X%04hX]",016,EA);
break;

case 0X08: // EA = mainMemory[ FB-2*I16 ] ***NOTICE SUBTRACTION***
IEA = FB - 2*016;
ReadWORDFromMainMemory(IEA,&EA);
sprintf(information," @FB(%3hd) memory[EA = 0X%04hX = memory[0X%04hX]]",016,EA,IEA);
break;

case 0X09: // Pop TOS; EA = FB-2*I16 + TOS ***NOTICE SUBTRACTION***
ReadWORDFromMainMemory(*SP+2,&TOS); *SP += 2;
EA = FB - 2*016 + TOS;
sprintf(information," FB(%3hd)+(%5hd) memory[EA = 0X%04hX]",016,TOS,EA);
break;

case 0X0A: // EA = SB+2*I16
EA = SB + 2*016;
sprintf(information," SB(%3hd) memory[EA = 0X%04hX]",016,EA);
break;

case 0X0B: // EA = mainMemory[ SB+2*I16 ]
IEA = SB + 2*016;
ReadWORDFromMainMemory(IEA,&EA);
sprintf(information," @SB(%3hd) memory[EA = 0X%04hX = memory[0X%04hX]]",016,EA,IEA);
break;
```

```

case 0X0C: // Pop TOS; EA = SB+2*I16 + TOS
    ReadWORDFromMainMemory(*SP+2,&TOS); *SP += 2;
    EA = SB + 2*016 + TOS;
    sprintf(information," SB(%3hd)+(%5hd) memory[EA = 0X%04hX]",016,TOS,EA);
    break;
default:
    ProcessRunTimeError("Invalid addressing mode (not in [ 0X00,0X0C ])",true);
    break;
}
return( EA );
}

/*
IEEE-754 16-bit half-precision float format
SEEE EEMM MMMM MMMM
1000 0000 0000 0000 = 0X8000 mask for sign
0111 1100 0000 0000 = 0X7C00 exponent
0000 0011 1111 1111 = 0X03FF mantissa

IEEE-754 32-bit single-precision float format
SEEE EEEE EMMM MMMM MMMM MMMM MMMM
1000 0000 0000 0000 0000 0000 0000 = 0X80000000 mask for sign
0111 1111 1000 0000 0000 0000 0000 = 0X7F800000 exponent
0000 0000 0111 1111 1111 1111 1111 = 0X007FFFFF mantissa
*/

```

```
//-----

void ConvertFloatToHalfFloat(float F,WORD *HF)
//-----

{

    union
    {

        float F;
        unsigned int UI;
    } X;

    int Fs,Fe,Fm;
    int HFs,HFe,HFm;

    X.F = F;

    Fs = (X.UI & 0X80000000) >> 31;
    Fe = (X.UI & 0X7F800000) >> 23;
    Fm = (X.UI & 0X007FFFFF);

    // +-float (denormalized) --> +-0 half-float
    if      ( (Fe == 0) && (Fm != 0) )
    {
        HFs = Fs;
        HFe = 0;
        HFm = 0;
    }
    // +-0 float --> +-0 half-float
```

```
else if ( (Fe == 0) && (Fm == 0) )
{
    HFs = Fs;
    HFe = 0;
    HFm = 0;
}

// +-Inf float --> +-Inf half-float

else if ( (Fe == 255) && (Fm == 0) )
{
    HFs = Fs;
    HFe = 31;
    HFm = 0;
}

// +-NaN float --> +-NaN half-float

else if ( (Fe == 255) && (Fm != 0) )
{
    HFs = Fs;
    HFe = 31;
    HFm = Fm >> 13;
}

// | float | > 2^15 --> +-Inf half-float

else if ( Fe-127 > 15 )
{
    HFs = Fs;
    HFe = 31;
    HFm = 0;
```

```
}

// | float | < 2^-24 --> +-0 half-float
else if ( Fe-127 < -24 )
{
    HFs = Fs;
    HFe = 0;
    HFm = 0;
}

// 2^-24 <= | +-float (normalized) | < 2^-14 --> +-half-float (denormalized)
else if ( (-24 <= Fe-127) && (Fe-127 < -14) )
{
    HFs = Fs;
    HFe = 0;
    HFm = (Fm | 0X00800000) >> (13 + (-14-(Fe-127)));
}

// +-float (normalized) --> +-half-float (normalized)
else
{
    HFs = Fs;
    HFe = Fe-127+15;
    HFm = Fm >> 13;
}

*HF = (WORD) ((HFs << 15) | (HFe << 10) | HFm);
}
```

```
//-----  
void ConvertHalfFloatToFloat(WORD HF,float *F)  
//-----  
{  
    union  
    {  
        float F;  
        unsigned int UI;  
    } X;  
  
    int Fs,Fe,Fm;  
    int HFs,HFe,HFm;  
  
    HFs = (HF & 0X8000) >> 15;  
    HFe = (HF & 0X7C00) >> 10;  
    HFm = HF & 0X03FF;  
  
    // +-0 half-float --> +-0 float  
    if      ( (HFe == 0) && (HFm == 0) )  
    {  
        Fs = HFs;  
        Fe = 0;  
        Fm = 0;  
    }  
    // +-Inf half-float --> +-Inf float  
    else if ( (HFe == 31) && (HFm == 0) )
```

```
{  
    Fs = HFs;  
    Fe = 255;  
    Fm = 0;  
}  
  
// +-NaN half-float --> +-NaN float  
  
else if ( (HFe == 31) && (HFm != 0) )  
{  
    Fs = HFs;  
    Fe = 255;  
    Fm = HFm << 13;  
}  
  
// +-half-float(denormalized) --> +-float (normalized)  
  
else if ( (HFe == 0) && (HFm != 0) )  
{  
    Fs = HFs;  
    Fe = -14+127;  
    Fm = HFm << 13;  
    do  
    {  
        Fm <= 1;  
        Fe -= 1;  
    } while ( (Fm & 0X00800000) == 0 );  
    Fm = Fm & 0X007FFFFF;  
}  
  
// +-half-float (normalized) --> +-float (normalized)
```

```
else
{
    Fs = HFs;
    Fe = HFe-15+127;
    Fm = HFm << 13;
}

X.UI = (Fs << 31) | (Fe << 23) | Fm;
*F = X.F;
}

//-----
void ConvertHalfFloatToBase10(WORD HF,char base10[])
//-----
{
    int HFs = (HF & 0X8000) >> 15;
    int HFe = (HF & 0X7C00) >> 10;
    int HFm = HF & 0X03FF;

    // +-0 half-float
    if      ( (HFe == 0) && (HFm == 0) )
        sprintf(base10,"0.0");

    // +-Inf half-float
    else if ( (HFe == 31) && (HFm == 0) )
        sprintf(base10,"%sInf",((HFs == 1) ? "-" : "+"));

    // +-NaN half-float
```

```
else if ( (HFe == 31) && (HFm != 0) )
    sprintf(base10,"NaN");
// +-half-float (normalized and denormalized)
else
{
    float F;

    ConvertHalfFloatToFloat(HF,&F);

    // Display 4 significant digits with no exponent when 0.1000 <= | F | <= 999.9
    if      ( ( 0.1000f <= fabs(F)) && (fabs(F) <= 0.9999) )
        sprintf(base10,"%6.4f",F);
    else if ( ( 1.000f  <= fabs(F)) && (fabs(F) <= 9.999 ) )
        sprintf(base10,"%5.3f",F);
    else if ( ( 10.00f   <= fabs(F)) && (fabs(F) <= 99.99 ) )
        sprintf(base10,"%5.2f",F);
    else if ( ( 100.0f    <= fabs(F)) && (fabs(F) <= 999.9    ) )
        sprintf(base10,"%5.1f",F);
    // otherwise display -X.XXXESX or X.XXXESX
    else
    {
        int Ei;

        // Formats as -X.XXXESXXX or X.XXXESXXX with no format control of exponent!
        sprintf(base10,"%10.3E",F);
        // Shorten format to -X.XXXESX or X.XXXESX by erasing 00 in exponent
```

```
Ei = (int) strlen(base10)-5;  
base10[Ei+2] = base10[Ei+4];  
base10[Ei+3] = '\0';  
}  
}  
}
```