Table of Contents

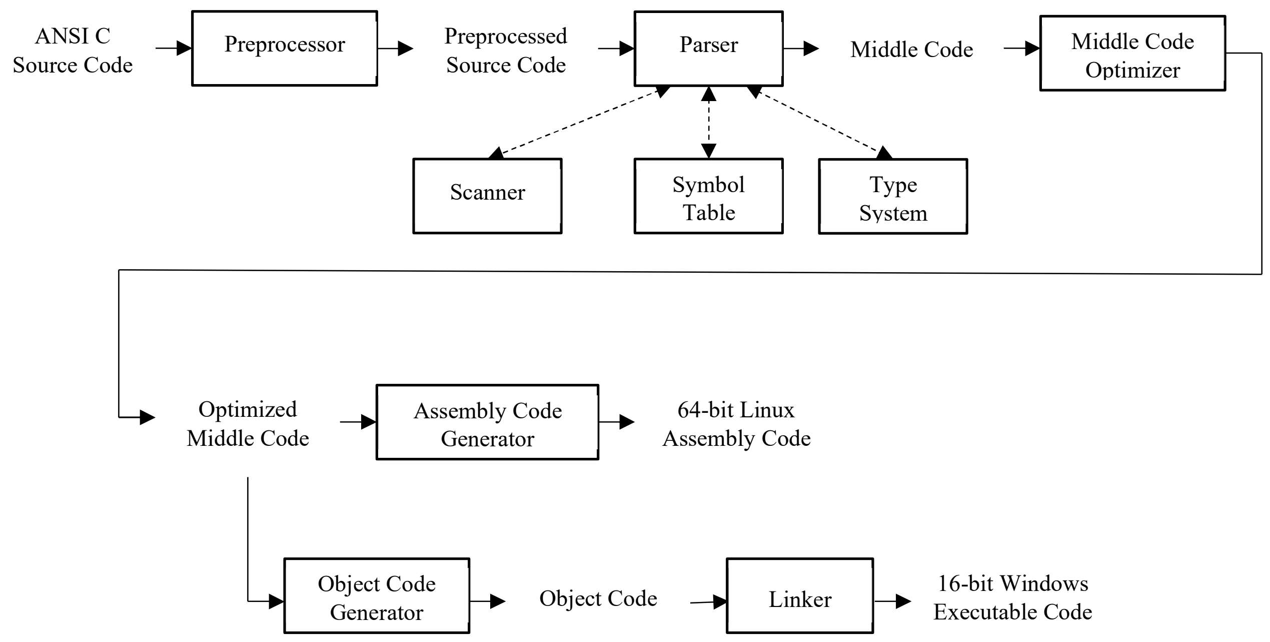
1. Introduction

This is a book about compiler construction. More specifically, about the construction of an optimized compiler for ANSI C. It can be set to generate two kinds of target code:

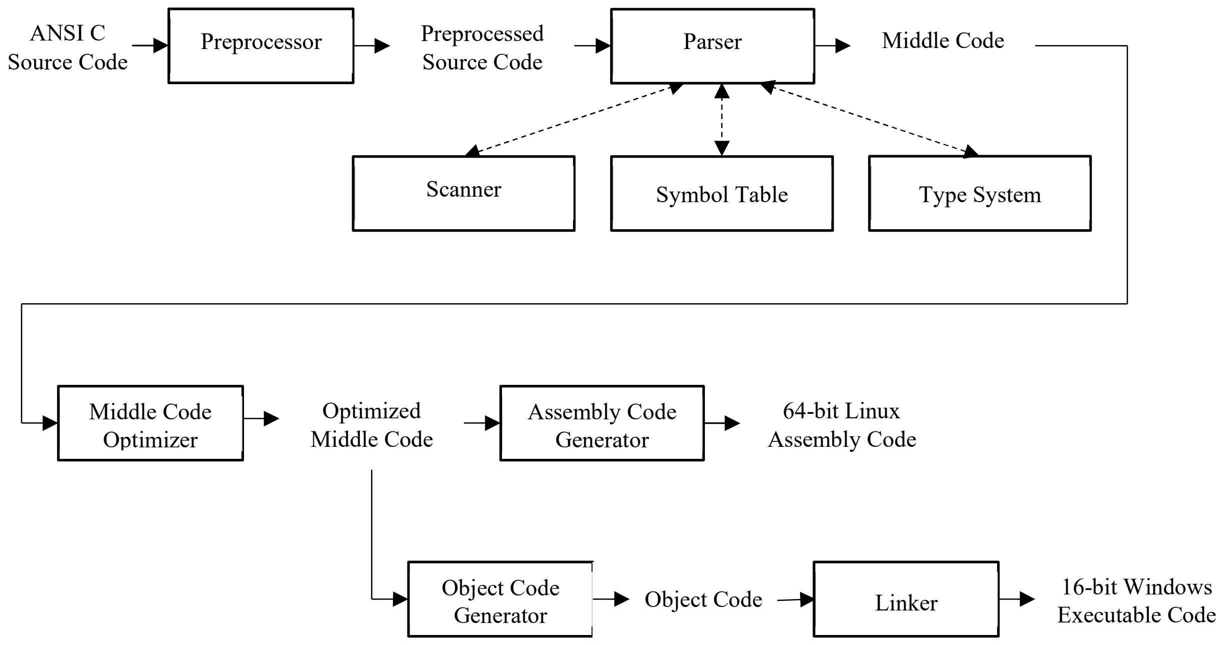
- Assembly code for the Intel 64-bit Linux system together with a makefile with instructions for further assemblering and linking.
- A file in the .com file format holding assembled and linked code the 16-bit Windows system, ready to be directly executed.

1.1. Overview

The compiler is divided into several phases. The input is ANSI C source code, the output is Linux assembly code or Windows executable code.



The compiler is made up be a sequence of phases.



1.2. The Compiler Phases

The compiler is made up be a sequence of phases, which each take code in some form and generates more refined code.

1.2.1. The Preprocessor

As the name implies, the preprocessor processes the source code before the actual compilation starts. Its task is to remove comments, include files, replace macros, and perform conditional programming. We look into the preprocessor in Appendix A.

1.2.2. Scanning

The scanner is responsible for interpreting sequences of characters into **tokens**: the least significant parts of the source code. For instance, the characters 'i' and 'f' are interpreted as the keyword **if** and the characters '3', '.', '1', and '4' are interpreted as the floating-point value 3.14.

1.2.3. Parsing and Middle Code Generation

Every programming language has a syntax, that describes the form of the code. In this book, the syntax is defined a **grammar**, described in Appendix A.3.7. The parser checks whether the source code is correct with regard to the grammar by requesting tokens from the scanner when needed. When the declarations are parsed the **symbol table** is generated, which holds information of variables, types, and functions. The **type system** is also used to perform type checking and type castings. The output of the parser is a sequence of **middle code**. As the name implies, the middle code is a simple notation holding the code between the parsing and the final target code generation. More specifically, each instruction can refer to three values at most; therefore, the middle code notation is called **three-address-code**.

Below is an example of how a sample of C code is translated into the middle code.

```
if (a < b)
   a = -b;
 else
   b = -a;
```

(a) C code

```
1. if a < b goto 3
2. goto 5
3. a = -b;
4. goto 6
5. b = -a
6. ...
```

(b) Middle code

As each middle code instruction can refer to three values at the most, we introduce **temporary values** to hold sub expressions.

```
x = (a + b) * (c - d);
```

(a) C code

```
1. temp1 = a + b
2. temp2 = c - d
3. temp3 = temp1 * temp2
4. x = temp3
```

(b) Middle code

1.2.4. Middle Code Optimization

The purpose of the middle code optimization phase is to make the middle code more effective, to provide for more effective assembly code generation. Reduction of conditional jumps is one example of such optimization. For instance, by swapping the condition in **if** statement of the example below, we can remove a **goto** instruction.

```
1. if a < b goto 3
2. goto 5
3. a = -b;
4. goto 6
5. b = -a
6. ...
```

(a) Before

```
1. if a >= b goto 4
2. a = -b;
3. goto 5
4. b = -a;
5. ...
```

(b) After

1.2.5. Initialization

A constant or variable may be initialized. The initializer may be a single value, a list of values, or a list of other lists. In the following example, and two-dimensional array is initialized by lists.

```
int a[][3] = {{1, 2, 3}, {4, 5, 6}};
```

The same array may also be initialized by one list.

```
int a[][3] = {1, 2, 3, 4, 5, 6};
```

1.2.6. Static Address

A static address is an address which value is determined in compile-time. For instance, if **a** is a static array, the **&a[3]** expression is set specific constant address by the compiler and linker.

1.2.7. Declarators and Declaration Specifiers

In the following code, **struct {int i;}** is a **specifier** while **s** and ***p** are **declarators**.

```
struct {int i;} s, *p;
```

A declarator can also be initialized. In a struct or union it can be marked as a bitfield.

```
struct {int i;} s = {1}, *p = NULL;
struct {int i : 3;};
```

1.2.8. The Symbol Table

The symbol table holds information of the symbols of the code, such as variables, constants, and functions, as well as struct and union tags. A symbol can hold extern, static, auto, register, and typedef storage. The symbol table is actually a hierarchy of tables, where the root table represent global space and the other tables represents functions, blocks in functions, or member sets of structs or unions.

1.2.9. The Type System

ANSI C holds a rather large set of types. The basic types are the integral types and floating-point types. There are also pointers, arrays, function, structs and unions. It is possible to type cast between several of the types. Moreover, in some situation is an array or function is interpreted as a pointer.

1.2.10. Assembly Code Generation

When the middle code has been generated and optimized, we generate the assembly code. In the first step, we generate the assembly code and use **tracks** to mark where in the assembly code the yet unknown registers shall be placed. In the second step we perform register allocation to find the optimal us of the register set, and finally replace the tracks with the actual registers.

1.2.1. Register Allocation

The register allocator creates a graph where the vertices hold tracks, and two vertices are connected by an edge if their tracks overlaps in the code. Then we find a match where two overlapping tracks are assigned non-overlapping registers. Let us look at the following example, we assume that **x**, **a**, **b**, **c**, and **d** are static 16-bit variables stored at specific addresses.

```
x = a + b & c - d;
```

(a) C Code

```
1. temporary0 = a + b
2. temporary1 = c - d
3. temporary2 = t0 & t1
4. x = temporary2
```

(b) Middle Code

In the assembly code generation, the registers are represented by tracks, that are replaced by proper registers by the register allocator.

```
mov track0, [aaddress]
add track0, [baddress]
mov track1, [caddress]
sub track1, [daddress]
and track0, track1
mov [xaddress], track0
```

(c) Assembly Code with Tracks

```
mov ax, [aaddress]
add ax, [baddress]
mov bx, [caddress]
sub bx, [daddress]
and ax, bx
mov [xaddress], ax
```

(d) Final Assembly Code

In the final assembly code, we cannot assign **track0** and **track1** the same register, since they overlap.

1.2.2. The Object Code Generator and Linker

At mentioned at the beginning of this chapter, there are two target systems: 64-bit Linux and 16-bit Windows. In the Windows case, we generate files with object code and link the object code together in a file executable in the .com file format. However, we will ignore this alternative until Chapter 13, and focus on the Linux case up until then.

1.2.3. The Standard Library

The C language comes with a set of functions, macros, and instructions for many services. It is mostly implemented in C. However, there are some additional non-standard elements for accessing registers and performing system calls.

1.3. Calling Forwards or Backwards

When methods call each other, in which order shall they be defined? Shall calls be made forwards or backwards? In this book I have chosen a compromise. Generally speaking, calls are made forwards. However, small methods are defined before the calling function, resulting in backwards calls.

```
void a() {
  b();
}

void b() {
}
```

(a) Forward Call

```
void b() {
}

void a() {
  b();
}
```

(b) Backward Call

1.4. The Main Class

The **Main** class handles the overall compiling and linking. It compiles the source code and generates the assembly files together with a makefile, that is used to assembly and link the code to the final executable file.

Main.cs

```
using System;
using System.IO;
using System.Text;
using System.Globalization;
using System.Collections.Generic;

namespace CCompiler {
  public class Start {
```

First of all, there is the **Linux** and **Windows** fields, that indicates whether the output of the compiler is a set of assembly files and a makefile for the Linux system, or an executable file for the Windows system. Throughout this book, there will be several statements that test whether the **Windows** or **Linux** field is true. We will look into the Linux cases through the first part of this book and wait with the Windows case until Chapter 13.

```
    public static bool Linux, Windows;
    public static string SourcePath =
      @"C:\Users\Stefan\Documents\vagrant\homestead\code\code\",
                    TargetPath = @"C:\D\";
```

As stated above, we simply ignore the **Start.Windows** it this point, an revisit them in Chapter 13.

```
  public static void Main(string[] args){
    if (Start.Windows) {
      // ...
    }
```

We set the current culture for decimal numbers to be properly interpreted.

```
    System.Threading.Thread.CurrentThread.CurrentCulture =
      CultureInfo.InvariantCulture;
```

If there are no command line arguments, we report an error.

```
    if (args.Length == 0) {
      Assert.Error("usage: compiler <filename>");
    }
```

The **rebuild** flag indicates that each source file shall be compiled, regardless of whether the generated assembly file is fresh; that is, was last modified after the source file or any of its included files was last modified. The **print** flag indicates that information about the compiling process shall be printed to **stdout**.

```
    List<string> argList = new List<string>(args);
    bool rebuild = argList.Remove("-rebuild"),
         print = argList.Remove("-print");
```

Then we start by iterating through the files of the command line arguments. The **IsGeneratedFileFresh** call decides if the assembly file is fresh; that is, if neither its source file nor any of the included files has been modified after generated assembly file. If the **rebuild** flag is set or if the source file is not fresh, it becomes compiled. The **rebuild** flag has just that effect, that the file shall be recompiled regardless of whether it is necessary or not.

```
    try {
      if (Start.Linux) {
        foreach (string arg in argList) {
          FileInfo file = new FileInfo(SourcePath + arg);

          if (rebuild || !IsGeneratedFileFresh(file, ".asm")) {
            if (print) {
              Console.Out.WriteLine("Compiling \"" +
                                    file.FullName + ".c\".");
            }

            CompileSourceFile(file);
          }
        }
```

Finally, we call the **GenerateMakeFile** method to create the makefile.

```
        GenerateMakeFile(argList);
      }

      if (Start.Windows) {
        // ...
      }
    }
```

In case of a runtime error (most likely: file error), we catch the exception and report an error.

```
      catch (Exception exception) {
        Console.Out.WriteLine(exception.StackTrace);
        Assert.Error(exception.Message, Message.Parse_error);
      }
    }
```

1.4.1. Generating the Assembly File

The **CompileSourceFile** method compiles the source file into an assembly file. We begin by preprocessing the source code by creating an object of the **Preprocessor** class. The preprocessing has two outputs: the preprocessed code and the include set. We use the include set when calling **GenerateDependencyFile** to establish the dependency file of the source file; that is, the name of the source file itself and the files included (directly or indirectly) by the source file.

```
    public static void CompileSourceFile(FileInfo file) {
      FileInfo sourceFile = new FileInfo(file.FullName + ".c");
      Preprocessor preprocessor = new Preprocessor(sourceFile);
      GenerateDependencyFile(file, preprocessor.IncludeSet);
```

We convert the preprocessed source code to a byte array, that we use as input in a memory stream, that finally become input to the scanner.

```
      byte[] byteArray =
        Encoding.ASCII.GetBytes(preprocessor.PreprocessedCode);
      MemoryStream memoryStream = new MemoryStream(byteArray);
      CCompiler_Main.Scanner scanner =
        new CCompiler_Main.Scanner(memoryStream);
```

We set the root symbol table to represent the global space of the source code and parse the source code. The **Path** and **Line** fields of the scanner is initialized in order to report the correct file and number in cases of error messages. If the parser encounters an error, it returns false and we report a syntax error.

```
      try {
        SymbolTable.CurrentTable = new SymbolTable(null, Scope.Global);
        CCompiler_Main.Parser parser = new CCompiler_Main.Parser(scanner);
        Assert.Error(parser.Parse(), Message.Syntax_error);
      }
      catch (IOException ioException) {
        Assert.Error(false, ioException.StackTrace, Message.Syntax_error);
      }
```

The result of the parsing is the **SymbolTable.GlobalStaticSet** set, holding a set of static symbols, such as functions and static variables. The resulting assembly file shall is made up by three parts:

Extern. The names of objects defined in other files and accessible in this file.

Global. The names of objects defined in this file and accessible in other files.

Code. The actual assembly code of the symbols of the global static set.

We start by defining the total extern set; that is, the union of the extern sets of the symbol in the global static set of the file.

```
      if (Start.Linux) {
        ISet<string> totalExternSet = new HashSet<string>();

        foreach (StaticSymbol staticSymbol in SymbolTable.GlobalStaticSet) {
          StaticSymbolLinux staticSymbolLinux =
            (StaticSymbolLinux) staticSymbol;
          totalExternSet.UnionWith(staticSymbolLinux.ExternSet);
        }
```

We remove the names of the symbols of the global static set, since objects defined in this file are accessible in this file and shall not be stated as extern.

```
        foreach (StaticSymbol staticSymbol in SymbolTable.GlobalStaticSet) {
          totalExternSet.Remove(staticSymbol.UniqueName);
        }
```

Now we are ready to write the actual assembly file, we start by creating a file and connect a stream to in. Is there is already a file with the same name, it will be overwritten.

```
        FileInfo assemblyFile = new FileInfo(file.FullName + ".asm");
        StreamWriter streamWriter = new StreamWriter(assemblyFile.FullName);
```

We iterate through the symbols and add the names of all symbol, with the exceptions of the symbols without external linkage (which have a separator identifier in their names) and symbols that represents constant values (which have a numeric identifier in their names). Neither symbols without external linkage nor value symbol shall be visible for other files.

```
        foreach (StaticSymbol staticSymbol in SymbolTable.GlobalStaticSet) {
          if (!staticSymbol.UniqueName.Contains(Symbol.SeparatorId) &&
              !staticSymbol.UniqueName.Contains(Symbol.NumberId)) {
            streamWriter.WriteLine("\tglobal " + staticSymbol.UniqueName);
          }
        }
        streamWriter.WriteLine();
```

Then we add the names of all extern references.

```
        foreach (string externName in totalExternSet) {
          streamWriter.WriteLine("\textern " + externName);
        }
```

If this file holds the initialization code, we state that the **_start** marker and **CallStackStart** shall marker shall be global; that is, present in this file and accessible in other files. The **_start** marker marks the start point of the execution of the code, while **CallStackStart** marks the beginning of the call stack.

```
        if (SymbolTable.InitSymbol != null) {
          streamWriter.WriteLine("\tglobal _start");
          streamWriter.WriteLine("\tglobal " + Linker.CallStackStart);
        }
```

If this file does not hold the initialization code, we state that the **_start** marker is extern; that is, present in another file and accessible in this file.

```
        else {
          streamWriter.WriteLine("\textern " + Linker.CallStackStart);
        }
        streamWriter.WriteLine();
```

Then we iterate through the global static set and add the text lists (that holds the assembly code) of the symbols.

```
        foreach (StaticSymbol staticSymbol in SymbolTable.GlobalStaticSet) {
          StaticSymbolLinux staticSymbolLinux =
            (StaticSymbolLinux) staticSymbol;
          streamWriter.WriteLine();
          foreach (string line in staticSymbolLinux.TextList) {
            streamWriter.WriteLine(line);
          }
        }
```

If this file holds the initialization code, we add the call stack for the activation records at the end of the file. We allocate one megabyte (1 048 576 bytes) for the call stack.

```
        if (SymbolTable.InitSymbol != null) {
          streamWriter.WriteLine();
          streamWriter.WriteLine("section .data");
          streamWriter.WriteLine(Linker.CallStackStart +
                                 ":\ttimes 1048576 db 0");
        }

        streamWriter.Close();
      }

      if (Start.Windows) {
        // ...
      }
    }
```

1.4.1. Generate the Make File

The **GenerateMakeFile** method generate a makefile for assembling and linking the target files into an executable file.

```
    private static void GenerateMakeFile(List<string> argList) {
      StreamWriter makeStream = new StreamWriter(SourcePath + "makefile");
```

We call the resulting executable file “main”, and we state that it is dependent of the object files; that is, the file names with the “.o” suffix.

```
      makeStream.Write("main:");
      foreach (string arg in argList) {
        makeStream.Write(" " + arg.ToLower() + ".o");
      }
      makeStream.WriteLine();
```

We link the object files together with the GNU linker (**ld**) system command, which perform the linking of the object files.

```
      makeStream.Write("\tld -o main");
      foreach (string arg in argList) {
        makeStream.Write(" " + arg.ToLower() + ".o");
      }
      makeStream.WriteLine();
      makeStream.WriteLine();
```

We assembly each file with the Netwide Assembler (**nasm**), to assembly the assembly file into an object file.

```
      foreach (string arg in argList) {
        makeStream.WriteLine(arg.ToLower() + ".o: " + arg.ToLower() + ".asm");
        makeStream.WriteLine("\tnasm -f elf64 -o " + arg.ToLower() + ".o "
                             + arg.ToLower() + ".asm");
        makeStream.WriteLine();
      }
```

Finally, we generate the clear list by listing all the object files and the final main file.

```
      makeStream.WriteLine("clear:");
      foreach (string arg in argList) {
        makeStream.WriteLine("\trm " + arg.ToLower() + ".o");
      }

      makeStream.WriteLine("\trm main");
      makeStream.Close();
    }
```

Below is an example of how the makefile may look:

```
main: main.o malloc.o ctype.o errno.o locale.o math.o setjmp.o signal.o file.o
      temp.o scanf.o printf.o stdlib.o time.o string.o
      ld -o main main.o malloc.o ctype.o errno.o locale.o math.o setjmp.o
      signal.o file.o temp.o scanf.o printf.o stdlib.o time.o string.o

main.o: main.asm
      nasm -f elf64 -o main.o main.asm

malloc.o: malloc.asm
      nasm -f elf64 -o malloc.o malloc.asm

ctype.o: ctype.asm
      nasm -f elf64 -o ctype.o ctype.asm

errno.o: errno.asm
      nasm -f elf64 -o errno.o errno.asm

locale.o: locale.asm
      nasm -f elf64 -o locale.o locale.asm

math.o: math.asm
      nasm -f elf64 -o math.o math.asm

setjmp.o: setjmp.asm
      nasm -f elf64 -o setjmp.o setjmp.asm

signal.o: signal.asm
      nasm -f elf64 -o signal.o signal.asm

file.o: file.asm
      nasm -f elf64 -o file.o file.asm

temp.o: temp.asm
      nasm -f elf64 -o temp.o temp.asm

scanf.o: scanf.asm
      nasm -f elf64 -o scanf.o scanf.asm

printf.o: printf.asm
      nasm -f elf64 -o printf.o printf.asm

stdlib.o: stdlib.asm
      nasm -f elf64 -o stdlib.o stdlib.asm
```

```
time.o: time.asm
      nasm -f elf64 -o time.o time.asm

string.o: string.asm
      nasm -f elf64 -o string.o string.asm

clear:
      rm main.o
      rm malloc.o
      rm ctype.o
      rm errno.o
      rm locale.o
      rm math.o
      rm setjmp.o
      rm signal.o
      rm file.o
      rm temp.o
      rm scanf.o
      rm printf.o
      rm stdlib.o
      rm time.o
      rm string.o
      rm main
```

1.4.2. Is the Object File Fresh?

The **GenerateDependencyFile** method defines for each source code file its dependency set; that is, the name of the source file and the names of the files (directly or indirectly) included with the **#include** preprocessor directive. We use the set of include files information extracted by the preprocessor.

```
    private static void GenerateDependencyFile(FileInfo file,
                                               ISet<FileInfo> includeSet) {
      FileInfo dependencySetFile = new FileInfo(file.FullName + ".dependency");
      StreamWriter dependencyWriter =
        new StreamWriter(File.Open(SourcePath + dependencySetFile.Name,
                                   FileMode.Create));

      dependencyWriter.Write(file.Name + ".c");
      foreach (FileInfo includeFile in includeSet) {
        dependencyWriter.Write(" " + includeFile.Name);
      }

      dependencyWriter.Close();
    }
```

The **IsGeneratedFileFresh** method decides whether a generated file is fresh; that is, if none of its dependency files has been modified since the generated file was modified.

```
    public static bool IsGeneratedFileFresh(FileInfo file, string suffix) {
      FileInfo generatedFile = new FileInfo(file.FullName + suffix),
               dependencySetFile = new FileInfo(file.FullName + ".dependency");
```

If the generated file does not exist, it is obviously not fresh, so we return false. If the dependency set file does not exist, we have to assume that it is not fresh due to dependencies, and we return false.

```
      if (!generatedFile.Exists || !dependencySetFile.Exists) {
        return false;
      }
```

If none of the cases above apply, we continue to look into the dependency files; that is, the files that the generated file is (directly or indirectly) dependent of.

```
      if (dependencySetFile.Exists) {
        try {
          StreamReader dependencySetReader =
            new StreamReader(File.OpenRead(dependencySetFile.FullName));
          string dependencySetText = dependencySetReader.ReadToEnd();
          dependencySetReader.Close();
```

We split the file text into dependency file names and check whether the files have been modified after generated file.

```
          if (dependencySetText.Length > 0) {
            string[] dependencyNameArray = dependencySetText.Split(' ');

            foreach (string dependencyName in dependencyNameArray)  {
              FileInfo dependencyFile =
                new FileInfo(SourcePath + dependencyName);
```

If the file has been modified after the generated file, the generated file is not fresh, and we return false.

```
              if (dependencyFile.LastWriteTime > generatedFile.LastWriteTime){
                return false;
              }
            }
          }
        }
        catch (IOException ioException) {
          Console.Out.WriteLine(ioException.StackTrace);
          return false;
        }
      }
```

When we have iterated through the without finding any dependencies, the generated file is fresh and we return true.

```
      return true;
    }
  }
}
```

In order for the parsers of this book to work properly, we add three parser classes. We need them in order to use the parsers.

```
namespace CCompiler_Main {
  public partial class Parser :
         QUT.Gppg.ShiftReduceParser<ValueType, QUT.Gppg.LexLocation> {
    public Parser(Scanner scanner)
     :base(scanner) {
      // Empty.
    }
  }
}
```

The expression parser is used by the preprocessor, to parse expressions in the **#if** and **#ifelse** directives. In the expression grammar there is the possibility to test whether a macro has been defined. Therefore, we let the constructor accepts the macro map as a parameter.

```
namespace CCompiler_Exp {
  public partial class Parser :
         QUT.Gppg.ShiftReduceParser<ValueType, QUT.Gppg.LexLocation> {
    public static IDictionary<string,CCompiler.Macro> m_macroMap;

    public Parser(Scanner scanner,
                  IDictionary<string,CCompiler.Macro> macroMap)
     :base(scanner) {
      m_macroMap = macroMap;
    }
  }
}

namespace CCompiler_Pre {
  public partial class Parser :
         QUT.Gppg.ShiftReduceParser<ValueType, QUT.Gppg.LexLocation> {
    public Parser(Scanner scanner)
     :base(scanner) {
      // Empty.
    }
  }
}
```

2. Scanning

In this project we use Garden Point Lex, which is a scanner-generation tool based on the classic Lex tool. For a crash course, see Appendix D.

2.1. The typedef-name Problem

Let us take look at the two source code lines below. Intuitively, the first line looks like an expression statement where **x** and **y** are variables of integral or floating types. Admittedly, the expression lacks side effects and is therefore meaningless, but it is still a valid expression statement. On the other hand, the second line looks like a pointer declaration where **MyType** is a type defined by and earlier **typedef** definition and **ptr** is a pointer to that type.

```
x * y;
MyType* ptr;
```

However, syntactically it is the same thing: a name, an asterisk, another name, and a semicolon. The parser cannot distinguish between the two cases. The solution to the problem is to give the scanner access to the symbol table, so it can look up the whether the name occurred is an earlier **typedef** definition, so that the parser can distinguish between the two cases.

2.2. The Scanner

The scanner identifies and returns the tokens of the C language; that is, keywords, operators, strings, characters, and numerical values.

Scanner.gplex

```
%namespace CCompiler_Main

%using CCompiler;
%using System.Numerics;

%{
  public static FileInfo Path = null;
  public static int Line = 1;
```

In the first part of the scanner we define the scanner rules. We need rules for names as well as octal, decimal, and hexadecimal, floating, string and character values. There are several kinds of values:

- **Octal**. An octal value starts with an optional plus or minus sign, that is followed by a zero and possible more digits between zero and seven, inclusive.

  ```
  OCTAL_VALUE [\+\-]?0[0-7]*
  ```

- **Decimal**. A decimal value is made up an optional plus or minus sign followed by at least one digit. The first digit cannot be zero.

  ```
  DECIMAL_VALUE [\+\-]?[1-9][0-9]*
  ```

- **Hexadecimal**. A hexadecimal value starts with an optional plus or minus sign, followed by zero and a lowercase ‘x’ or an uppercase ‘X’, and at least one hexadecimal digits.

```
HEXADECIMAL_VALUE [\+\-]?0[xX][0-9a-fA-F]+
```

- **Floating-point**. A floating-point is made up by the decimal part and the exponent part. If the exponent part is not empty, the decimal part can be made up by digits without a dot. If the exponent part is empty, there has to be a dot and at least one digit in the integer or fraction part.

```
DECIMAL_WITHOUT_EXPONENT (([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+))
DECIMAL_WITH_EXPONENT    [0-9]+|(([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+))
```

  The exponent part starts with a lowercase 'e' or an uppercase 'E', followed by an optional plus or minus sign, at least one digit, and an optional lowercase 'f' or 'l' or an uppercase 'F' or 'D', to indicate a value of the **float** or **long double** type.

```
EXPONENT                 [eE][\+\-]?[0-9]+
FLOATING_VALUE           [\+\-]?({DECIMAL_WITHOUT_EXPONENT}|
                         ({DECIMAL_WITH_EXPONENT}{EXPONENT}))[flFL]?
```

- The decimal values can be appended by the lowercase 'u' or an uppercase 'U' to indicate an unsigned value and the lowercase 's' or 'l' or an uppercase 'S' or 'L' to indicate a short or long (signed or unsigned) value.

```
INTEGRAL_VALUE ({DECIMAL_VALUE}|{OCTAL_VALUE}|{HEXADECIMAL_VALUE})
               ([uU]?[sSlL]?)|([sSlL]?[uU]?)
```

- A character starts with single quotation mark followed by at least one character and is terminated by another single quotation mark. However, a single quotation mark may be preceded by a backslash.

```
CHAR_VALUE \'(\\\'|[^'])*\'
```

- In the same way, a string starts with a double quotation mark followed by anything except double quotation marks and is terminated by another double quotation mark. However, the string may hold double quotations marks if they are proceeded by backslashes.

```
STRING_VALUE \"(\\\"|[^"])*\"
```

- Names are used to identify variables, constants, struct and unions, enumerations, functions, and macros. They start with a letter or an underscore ('_'), that is optionally followed by letters, digits, or underscores.

```
NAME [a-zA-Z_][a-zA-Z0-9_]*
```

- Register names are used internally only, when performing system calls. They start with the text "register_" followed by the name of the register.

```
REGISTER_NAME "register_"[a-z]+
```

- The path line is used to keeping track of the current line number. It is used by the preprocessor macro **__LINE__**, and when reporting errors. The path line starts with a dollar sign followed by anything except newline (the dot represents every character except newline) and ends with another dollar sign.

```
PATH_LINE \$.*\$
```

- A white space is a space or any character that can substitute as a space; that is horizontal tabulator, return, newline, and form feed.

```
WHITE_SPACE [ \t\r\n\f]
```

The next section holds the actions of keywords, operators, and the rules defined above.

```
%%
"auto"          { return ((int) Tokens.AUTO);          }
"break"         { return ((int) Tokens.BREAK);         }
"case"          { return ((int) Tokens.CASE);          }
"carry_flag"    { return ((int) Tokens.CARRY_FLAG);    }
"char"          { return ((int) Tokens.CHAR);          }
"const"         { return ((int) Tokens.CONSTANT);      }
"continue"      { return ((int) Tokens.CONTINUE);      }
"default"       { return ((int) Tokens.DEFAULT);       }
"do"            { return ((int) Tokens.DO);            }
"double"        { return ((int) Tokens.DOUBLE);        }
"else"          { return ((int) Tokens.ELSE);          }
"enum"          { return ((int) Tokens.ENUM);          }
"extern"        { return ((int) Tokens.EXTERN);        }
"float"         { return ((int) Tokens.FLOAT);         }
"for"           { return ((int) Tokens.FOR);           }
"goto"          { return ((int) Tokens.GOTO);          }
"int"           { return ((int) Tokens.INT);           }
"interrupt"     { return ((int) Tokens.INTERUPT);      }
"if"            { return ((int) Tokens.IF);            }
"jump_register" { return ((int) Tokens.JUMP_REGISTER); }
"long"          { return ((int) Tokens.LONG);          }
"register"      { return ((int) Tokens.REGISTER);      }
"return"        { return ((int) Tokens.RETURN);        }
"short"         { return ((int) Tokens.SHORT);         }
"signed"        { return ((int) Tokens.SIGNED);        }
"sizeof"        { return ((int) Tokens.SIZEOF);        }
"static"        { return ((int) Tokens.STATIC);        }
"struct"        { return ((int) Tokens.STRUCT);        }
"switch"        { return ((int) Tokens.SWITCH);        }
"syscall"       { return ((int) Tokens.SYSCALL);       }
"typedef"       { return ((int) Tokens.TYPEDEF);       }
"union"         { return ((int) Tokens.UNION);         }
"unsigned"      { return ((int) Tokens.UNSIGNED);      }
"while"         { return ((int) Tokens.WHILE);         }
"void"          { return ((int) Tokens.VOID);          }
"volatile"      { return ((int) Tokens.VOLATILE);      }

";" { return ((int) Tokens.SEMICOLON); }
":" { return ((int) Tokens.COLON); }
"," { return ((int) Tokens.COMMA); }

"." { return ((int) Tokens.DOT); }
"->" { return ((int) Tokens.ARROW); }
"..." { return ((int) Tokens.ELLIPSE); }

"(" { return ((int) Tokens.LEFT_PAREN); }
")" { return ((int) Tokens.RIGHT_PAREN); }
"{" { return ((int) Tokens.LEFT_BLOCK); }
"}" { return ((int) Tokens.RIGHT_BLOCK); }
"[" { return ((int) Tokens.LEFT_SQUARE); }
"]" { return ((int) Tokens.RIGHT_SQUARE); }

"*" { return ((int) Tokens.ASTERRISK); }
"?" { return ((int) Tokens.QUESTION_MARK); }

"||" { return ((int) Tokens.LOGICAL_OR); }
```

```
"&&" { return ((int) Tokens.LOGICAL_AND); }
"!" { return ((int) Tokens.LOGICAL_NOT); }
"&" { return ((int) Tokens.AMPERSAND); }
"^" { return ((int) Tokens.BITWISE_XOR); }
"|" { return ((int) Tokens.BITWISE_OR); }
"~" { return ((int) Tokens.BITWISE_NOT); }

"==" { return ((int) Tokens.EQUAL); }
"!=" { return ((int) Tokens.NOT_EQUAL); }

"<"  { return ((int) Tokens.LESS_THAN); }
"<=" { return ((int) Tokens.LESS_THAN_EQUAL); }
">"  { return ((int) Tokens.GREATER_THAN); }
">=" { return ((int) Tokens.GREATER_THAN_EQUAL); }

"<<" { return ((int) Tokens.LEFT_SHIFT); }
">>" { return ((int) Tokens.RIGHT_SHIFT); }

"+" { return ((int) Tokens.PLUS); }
"-" { return ((int) Tokens.MINUS); }

"/" { return ((int) Tokens.DIVIDE); }
"%" { return ((int) Tokens.MODULO); }

"++" { return ((int) Tokens.INCREMENT); }
"--" { return ((int) Tokens.DECREMENT); }

"="  { return ((int) Tokens.ASSIGN); }
"+="  { return ((int) Tokens.ADD_ASSIGN); }
"-="  { return ((int) Tokens.SUBTRACT_ASSIGN); }
"*="  { return ((int) Tokens.MULTIPLY_ASSIGN); }
"/="  { return ((int) Tokens.DIVIDE_ASSIGN); }
"%="  { return ((int) Tokens.MODULO_ASSIGN); }
"<<=" { return ((int) Tokens.LEFT_SHIFT_ASSIGN); }
">>=" { return ((int) Tokens.RIGHT_SHIFT_ASSIGN); }
"&="  { return ((int) Tokens.AND_ASSIGN); }
"^="  { return ((int) Tokens.XOR_ASSIGN); }
"|="  { return ((int) Tokens.OR_ASSIGN); }
```

When we encounter a register name, we look up the register, and report an error there is no register with the name.

```
{REGISTER_NAME} {
  { Register register;
    string text = yytext.Substring(9);

    if (Enum.TryParse<Register>(text, out register)) {
      yylval.register = register;
      return ((int) Tokens.REGISTER_NAME);
    }

    Assert.Error(text, Message.Unknown_register);
  }
}
```

When we encounter a name, we need to check whether it represent a **typedef** name or a regular name, which is the name of a variable, constant, struct or union, enumeration, or a function. We look up the name in the current symbol table in accordance with the typedef-name problem above and return the **typedef-name** token in case of a typedef name. In case of a regular name, we return the **name** token.

```
{NAME} {
  { string name = yytext;

    if (Start.Linux && (name.Equals("abs") ||
        name.Equals("_start") || name.Equals("section") ||
        name.Equals("extern") || name.Equals("global"))) {
      name = Symbol.FileMarker + name;
    }

    Symbol symbol = SymbolTable.CurrentTable.LookupSymbol(name);

    if ((symbol != null) && symbol.IsTypedef()) {
      yylval.type = symbol.Type;
      return ((int) Tokens.TYPEDEF_NAME);
    }
    else {
      yylval.name = name;
      return ((int) Tokens.NAME);
    }
  }
}
```

When it comes to integral values, we need to decide the sign, base, and type of the value.

```
{INTEGRAL_VALUE} {
  { string text = yytext.Trim().ToLower();
```

The **ToUInt64** method does not accept plus or minus signs. Therefore, we need to remove the sign and assign true to the **minus** variable in case of a negative value.

```
    bool minus = false;
    if (text.StartsWith("+")) {
      text = text.Substring(1);
    }
    else if (text.StartsWith("-")) {
      minus = true;
      text = text.Substring(1);
    }
```

Then we decide the base of the value. If the text start with “0x” or “0X”, the value is a hexadecimal value with the base 16. If the text start with “0”, the value is an octal value with the base 8. Otherwise, it is a decimal value with the base 10.

```
    int fromBase;
    if (text.StartsWith("0x")) {
      fromBase = 16;
      text = text.Substring(2);
    }
    else if (text.StartsWith("0")) {
      fromBase = 8;
    }
    else {
      fromBase = 10;
    }
```

Then we decide the type of the value. If the text ends with any combination of lowercase ‘s’ or ‘u’ or an uppercase ‘S’ or ‘U’, the type is unsigned small integer.

```
CCompiler.Type type;
if (text.EndsWith("us") || text.EndsWith("su")) {
  type = CCompiler.Type.UnsignedShortIntegerType;
  text = text.Substring(0, text.Length - 2);
}
```

If the text ends with any combination of lowercase ‘l’ or ‘u’ or an uppercase ‘L’ or ‘U’, the type is large unsigned integer.

```
else if (text.EndsWith("ul") || text.EndsWith("lu")) {
  type = CCompiler.Type.UnsignedLongIntegerType;
  text = text.Substring(0, text.Length - 2);
}
```

If the text ends with a single lowercase ‘s’ or an uppercase ‘S’, the type is signed short integer.

```
else if (text.EndsWith("s")) {
  type = CCompiler.Type.SignedShortIntegerType;
  text = text.Substring(0, text.Length - 1);
}
```

If the text ends with a single lowercase ‘l’ or an uppercase ‘L’, the type is signed long integer.

```
else if (text.EndsWith("l")) {
  type = CCompiler.Type.SignedLongIntegerType;
  text = text.Substring(0, text.Length - 1);
}
```

If the text ends with a single lowercase ‘u’ or an uppercase ‘U’, the type is unsigned integer.

```
else if (text.EndsWith("u")) {
  type = CCompiler.Type.UnsignedIntegerType;
  text = text.Substring(0, text.Length - 1);
}
```

Otherwise, the type is signed integer.

```
else {
  type = CCompiler.Type.SignedIntegerType;
}
```

Finally, we decide the value by calling the **ToUInt64** method in the **Convert** standard class.

```
try {
  ulong unsignedValue = Convert.ToUInt64(text, fromBase);
```

We convert the value to an object of the standard **BigInteger** class, and if **minus** is true, we make the value negative.

```
  BigInteger bigValue = new BigInteger(unsignedValue);

  if (minus) {
    bigValue = -bigValue;
  }
```

We then create a symbol of the value that we return. We also store a static symbol in the global static set.

```
  yylval.symbol = new Symbol(type, bigValue);
  yylval.symbol.StaticSymbol =
    ConstantExpression.Value(yylval.symbol.UniqueName, type, bigValue);
  SymbolTable.StaticSet.Add(yylval.symbol.StaticSymbol);
}
```

The **ToUInt64** method in the **Convert** standard class throws the **OverflowException** if the value exceeds the limits, and we report an error.

```
    catch (OverflowException) {
      Assert.Error(type + ": " + text, Message.Value_overflow);
    }
```

Finally, we return the **value** token.

```
    return ((int) Tokens.VALUE);
  }
}
```

When it comes to floating values, we do not need to look for its plus or minus sign.

```
{FLOATING_VALUE} {
  { string text = yytext.ToLower();
    CCompiler.Type type;
```

If the text ends with lowercase ‘f’ an uppercase ‘F’, the type becomes a float.

```
    if (text.EndsWith("f")) {
      type = CCompiler.Type.FloatType;
      text = text.Substring(0, text.Length - 1);
    }
```

If the text ends with lowercase ‘l’ an uppercase ‘L’, the type becomes a long double.

```
    else if (text.EndsWith("l")) {
      type = CCompiler.Type.LongDoubleType;
      text = text.Substring(0, text.Length - 1);
    }
```

Otherwise, the type is a double.

```
    else {
      type = CCompiler.Type.DoubleType;
    }
```

We call the **Parse** method of the **decimal** standard class to obtain the value. Then we create and return a value symbol. We also add a static value to the global static set.

```
    try {
      decimal value = decimal.Parse(text, NumberStyles.Float);
      yylval.symbol = new Symbol(type, value);
      yylval.symbol.StaticSymbol =
        ConstantExpression.Value(yylval.symbol.UniqueName, type, value);
      SymbolTable.StaticSet.Add(yylval.symbol.StaticSymbol);
    }
```

The **Parse** method of the **decimal** standard throws an exception if the value exceeds the limits, in which case we report an error.

```
    catch (OverflowException) {
      Assert.Error(type + ": " + text, Message.Value_overflow);
    }
```

Finally, we return the **Value** token.

```
    return ((int) Tokens.VALUE);
  }
}
```

In the case of a character value, we call the **SlashToChar** method of the **Slash** class to clear the text from escape characters. We then check that it holds three characters (including the two single quotation marks), and return a symbol holding the value.

```
{CHAR_VALUE} {
  { CCompiler.Type type = new CCompiler.Type(Sort.SignedChar);
    string text = Slash.SlashToChar(yytext);
    Assert.Error(text.Length == 3, yytext, Message.Invalid_char_sequence);
    yylval.symbol = new Symbol(type, (BigInteger) ((int) text[1]));
    return ((int) Tokens.VALUE);
  }
}
```

In the case of a string value, we also call the **SlashToChar** method of the **Slash** class to clear the text from escape characters. We then return a symbol holding the value. In this case we do not need to check the length of the string.

```
{STRING_VALUE} {
  { CCompiler.Type type = new CCompiler.Type(Sort.String);
    string text = Slash.SlashToChar(yytext);
    object value = text.Substring(1, text.Length - 2);
    yylval.symbol = new Symbol(type, value);
    yylval.symbol.StaticSymbol =
      ConstantExpression.Value(yylval.symbol.UniqueName, type, value);
    SymbolTable.StaticSet.Add(yylval.symbol.StaticSymbol);
    return ((int) Tokens.VALUE);
  }
}
```

Path lines are generated by the preprocessor for the **__FILE__** and **__LINE__** macros to hold the correct file name and line number. It is also used by the compiler when reporting errors. Its text is made up by two dollar-signs ('$') at the beginning and end, and the file name and line number separated by a comma (','). The text also holds plus signs ('+') in place of spaces.

```
{PATH_LINE} {
  { string text = yytext.Substring(1, yyleng - 2);
    int index = text.IndexOf(',');
    Path = new FileInfo(text.Substring(0, index).Replace("+", " "));
    Line = int.Parse(text.Substring(index + 1));
  }
}
```

In case of a white space, we only update the **Line** variable if the character is a newline. Note that we do not return anything in this case, which causes the scanner to proceed with the next character of the input stream.

```
{WHITE_SPACE} {
  if (yytext.Equals("\n")) {
    ++Line;
  }
}
```

Finally, when the scanner finds a character that has not been handled by the rules above, an error is reported.

```
. { Assert.Error(yytext, Message.Unknown_character); }
```

2.3. Slash Sequences

A problem for the scanner to solve is to transform the escape sequences of characters and strings into the proper characters. For instance, the character sequence '\n' shall be transformed into the newline character, with ASCII value ten. For the ASCII table, see Appendix F.

Slash.cs
```
using System.Text;
using System.Collections.Generic;

namespace CCompiler {
  class Slash {
```

The **m_escapeMap** map holds the escape sequences of ANSI C.

```
    private static IDictionary<char,char> m_slashMap =
      new Dictionary<char,char>() {
        {'0', '\0'},  // Terminator, ASCII number 0
        {'a', '\a'},  // Alert (Beep, Bell), ASCII number 7
        {'b', '\b'},  // Backspace, ASCII number 8
        {'f', '\f'},  // Form Feed (Page Break), ASCII number 12
        {'n', '\n'},  // New Line (Line Feed), ASCII number 10
        {'r', '\r'},  // Carrige Return, ASCII number 13
        {'t', '\t'},  // Horizontal Tabulator, ASCII number 9
        {'v', '\v'},  // Vertical Tabulator, ASCII number 11
        {'\'', '\''}, // Single Quotation Mark, ASCII number 39
        {'\"', '\"'}, // Double Quotation Mark, ASCII number 34
        {'?', '?'},   // Question Mark, ASCII number 63
        {'\\', '\\'}  // Backslash, ASCII number 92
      };
```

The **SlashToChar** method takes a string with slash sequences and returns the same string with the slash sequences replaced by the proper characters.

```
    public static string SlashToChar(string text) {
      StringBuilder buffer = new StringBuilder(text);
```

We add three zero-characters to the string to make sure that the **char1**, **char2**, and **char3** values are valid (that we do not index outside the string).

```
      buffer.Append("\0\0\0");
      for (int index = 0; buffer[index] != '\0'; ++index) {
        if (buffer[index] == '\\') {
          char char1 = buffer[index + 1],
               char2 = buffer[index + 2],
               char3 = buffer[index + 3];
```

If the slash is followed by a character stored in the slash map, we replace the slash and the character with the character corresponding to the slash sequence by using the **m_escapeMap** map above.

```
          if (m_escapeMap.ContainsKey(char1)) {
            buffer.Remove(index, 2);
            buffer.Insert(index, m_escapeMap[char1]);
          }
```

If the three characters following the slash are octal digits, we calculate the ASCII value for the character.

```
          else if (IsOctal(char1) && IsOctal(char2) && IsOctal(char3)) {
              int octValue = 64 * CharToOctal(char1) +
                              8 * CharToOctal(char2) +
                                  CharToOctal(char3);
```

An ASCII value is not allowed to exceed 255; if it does, we report an error.

```
              Assert.Error(octValue <= 255, Message.Invalid_octal_sequence);
```

We remove four characters; that is, the slash and its following three octal digits, and we insert the resulting character in its place.

```
              buffer.Remove(index, 4);
              buffer.Insert(index, (char) octValue);
            }
```

If the slash is followed by two octal digits only, we use them to calculate the ASCII value. In this case, we do not need to check whether the ASCII exceeds 255, since the highest possible value with two octal digits is 63 ($77_8 = 8 * 7 + 7 = 63_{10}$).

```
            else if (IsOctal(char1) && IsOctal(char2)) {
              int octValue = 8 * CharToOctal(char1) +
                                 CharToOctal(char2);
              buffer.Remove(index, 3);
              buffer.Insert(index, (char)octValue);
            }
```

If the slash is followed by one octal digit only, we use it to calculate the ASCII value.

```
            else if (IsOctal(char1)) {
              int octValue = CharToOctal(char1);
              buffer.Remove(index, 2);
              buffer.Insert(index, (char) octValue);
            }
```

If the slash is followed by a lowercase ‘x’ or an uppercase ‘X’ and two hexadecimal digits, we use them to calculate the ASCII value. Also in this case we need not check whether the ASCII value exceeds 255, since the highest possible value with two hexadecimal digits is 255 ($FF_{16} = 15 * 16 + 15 = 255_{10}$).

```
            else if (char.ToLower(char1) == 'x') {
              if (IsHex(char1) && IsHex(char2)) {
                int hexValue = 16 * CharToHex(char1) + CharToHex(char2);
                buffer.Remove(index, 3);
                buffer.Insert(index, (char) hexValue);
              }
```

If the slash is followed by a lowercase ‘x’ or an uppercase ‘X’ and hexadecimal digit only, we use it to calculate the ASCII value.

```
              else if (IsHex(char1)) {
                int hexValue = CharToHex(char1);
                buffer.Remove(index, 2);
                buffer.Insert(index, (char) hexValue);
              }
```

If the slash is followed by lowercase ‘x’ or an uppercase ‘X’ without at least one hexadecimal digit, we report an error.

```
              else {
                Assert.Error(char1.ToString(),
                             Message.Invalid_hexadecimal_code);
```

```
          }
        }
```

Finally, if the slash is not followed by a character in the **m_escapeMap** map, an octal digit, a lowercase ‘x’ or an uppercase ‘X’, we report an error.

```
        else {
          Assert.Error(buffer[index + 1].ToString(),
                       Message.Invalid_slash_sequence);
        }
      }
    }
```

When we have traversed through the string, we remove the three zero-character we added at the beginning and return the string.

```
    buffer.Remove(buffer.Length - 3, 3);
    return buffer.ToString();
  }
```

The **IsOctal** and **IsHex** methods return true if the given character is and octal or hexadecimal digit, respectively.

```
  private static bool IsOctal(char c) {
    return "01234567".Contains(c.ToString());
  }

  private static bool IsHex(char c) {
    return "0123456789abcdef".Contains(c.ToString().ToLower());
  }
```

The **CharToOctal** and **CharToHex** methods return the numerical value corresponding to the given character.

```
  private static int CharToOctal(char c) {
    return "01234567".IndexOf(c);
  }

  private static int CharToHex(char c) {
    return "0123456789abcdef".IndexOf(c.ToString().ToLower());
  }
}
}
```

3. Parsing

The parser can be considered to be the heart of the compiler. It requests tokens from the scanner, checks that the source code complies with the grammar, constructs the symbol table, performs type checking, and generates middle code.

3.1. Scope and MiddleOperator

The **Scope** enumeration holds the possible scopes of the symbol table. Each symbol table in the hierarchy holds a specific scope. The **Block** scope corresponds to a code block inside a function.

Scope.cs

```
namespace CCompiler {
  public enum Scope { Struct = (int) Sort.Struct,
                      Union = (int) Sort.Union,
                      Function, Global, Block};
}
```

Before we start the parsing, we need to look into the middle code generated by the parser. The **MiddleOperator** enumeration holds the operator of the middle code.

MiddleOperator.cs

```
namespace CCompiler {
  public enum MiddleOperator
   {AssignRegister, InspectRegister, StackTop,
    JumpRegister, SysCall, Interrupt,

    PushZero, PushOne, PushFloat, PopFloat, TopFloat, PopEmpty,

    Initializer, InitializerZero, AssignInitSize, Assign,

    LogicalOr, LogicalAnd,
    BitwiseNot, BitwiseOr, BitwiseXOr, BitwiseAnd,
    ShiftLeft, ShiftRight,

    Compare, Equal, NotEqual, LessThan,
    LessThanEqual, GreaterThan, GreaterThanEqual,

    Carry, NotCarry, Case, CaseEnd, Jump,

    Index, Dot, Increment, Decrement,
    Plus, Minus, Add, Subtract, Multiply, Divide, Modulo,

    Address, Dereference,
    PreCall, Call, PostCall, DecreaseStack,

    ParameterInitSize, Parameter,
    GetReturnValue, SetReturnValue, Return, Exit,

    IntegralToIntegral, IntegralToFloating, FloatingToIntegral,

    FunctionEnd, Empty};
}
```

MainParser.gppg

```
%namespace CCompiler_Main
%partial

%using CCompiler;
%using System.Numerics;
```

We use the

```
%{
```

The **SpecifierStack** field is used to store the references to **Specifier** objects, corresponding to specifier lists.

```
  public static Stack<Specifier> SpecifierStack = new Stack<Specifier>();
```

The **EnumValueStack** field is used to store the references to **BigInteger** objects, in order to keep track of implicitly incremented enumeration value.

```
  public static Stack<BigInteger> EnumValueStack = new Stack<BigInteger>();
```

The **ScopeStack** field is used when parsing function parameters in order to distinguish them from regular variables, since parameters cannot hold static, extern, or typedef storage.

```
  public static Stack<Scope> ScopeStack = new Stack<Scope>();
%}
```

The parser is made up by tokens and rules, where tokens correspond to operators, keywords, and characters. The following tokens are used by the parser, and they are also returned by the scanner in Chapter 2.

A variable or function has **auto**, **register**, **static**, **extern**, or **typedef** storage.

```
%token AUTO REGISTER STATIC EXTERN TYPEDEF
```

It can also be qualified as **constant** or **volatile**.

```
       CONSTANT VOLATILE
```

An integral type can be **signed** or **unsigned char**, **short int**, **int**, or **long int**.

```
       SIGNED UNSIGNED CHAR SHORT INT LONG
```

A floating type can be **float**, **double**, or **long double**.

```
       FLOAT DOUBLE
```

Moreover, there are enumerations, structs, and unions.

```
       ENUM STRUCT UNION
```

Finally, there is also the **void** type, that marks the absence of a type. It is used to mark the absence of a parameter list or return type of a function, and as pointer to void.

```
       VOID
```

The asterisk (’*’) is used both for multiplication and dereferencing pointers.

```
       PLUS MINUS ASTERRISK DIVIDE MODULO INCREMENT DECREMENT

       EQUAL NOT_EQUAL LESS_THAN LESS_THAN_EQUAL GREATER_THAN
       GREATER_THAN_EQUAL
```

The assignment is both simple and compound.

```
       ASSIGN ADD_ASSIGN SUBTRACT_ASSIGN MULTIPLY_ASSIGN
```

```
        DIVIDE_ASSIGN MODULO_ASSIGN AND_ASSIGN OR_ASSIGN XOR_ASSIGN
        LEFT_SHIFT_ASSIGN RIGHT_SHIFT_ASSIGN

        LOGICAL_OR LOGICAL_AND LOGICAL_NOT
```

The ampersand ('&') is used for both the **bitwise and** and **address** operators.

```
        AMPERSAND BITWISE_XOR BITWISE_OR BITWISE_NOT
```

The ellipse is made up by three dots ('...') and is used when defining variadic functions; that is, functions with a variable number of parameters, such as **printf** or **scanf**.

```
        QUESTION_MARK COLON COMMA SEMICOLON ELLIPSE DOT ARROW SIZEOF
```

In C, we have the regular parentheses '(' and ')', block parentheses '{' and '}', and squares '[' and ']'

```
        LEFT_PARENTHESIS RIGHT_PARENTHESIS LEFT_BLOCK RIGHT_BLOCK
        LEFT_SQUARE RIGHT_SQUARE
```

We need a set of keywords for the statements of C.

```
        IF ELSE SWITCH CASE DEFAULT FOR WHILE DO BREAK CONTINUE RETURN GOTO
```

The interrupt, jump_register, syscall, and carry flag tokens are used for internal system calls only.

```
        INTERRUPT JUMP_REGISTER SYSCALL CARRY_FLAG
```

Note that we do not need any tokens for comments, since they have already been taken care of by the preprocessor.

The next part of the parser is the union section, where we define the attributes of the rules below.

```
%union {
  public string name;
```

The **Register** enumeration holds the registers of the target machine.

```
  public Register register;
```

Each symbol has a type, which is described by the **Type** class.

```
  public CCompiler.Type type;
  public List<CCompiler.Type> type_list;
  public Sort sort;
  public Symbol symbol;
  public IDictionary<string,Symbol> symbol_map;
  public ISet<Pair<Symbol,bool>> symbol_bool_pair_set;
  public Pair<Symbol,bool> symbol_bool_pair;
  public List<Symbol> symbol_list;
  public List<string> string_list;
  public Declarator declarator;
  public List<Declarator> declarator_list;
  public MiddleOperator middle_operator;
  public Expression expression;
  public List<Expression> expression_list;
  public Statement statement;
  public Pair<List<Symbol>,Boolean> parameter_pair;
  public List<MiddleCode> middle_code_list;
  public object obj;
  public List<object> object_list;
}
```

```
%token <name> NAME
%token <register> REGISTER_NAME
%token <type> TYPEDEF_NAME
%token <symbol> VALUE

%type <obj> declaration_specifier declaration_specifier_list_x
%type <object_list> declaration_specifier_list

%type <name> optional_name
%type <type> struct_or_union_specifier
%type <sort> struct_or_union

%type <type> enum_specifier
%type <symbol_bool_pair_set> enum_list
%type <symbol_bool_pair> enum
%type <middle_code_list> declarator_list declaration
                         declaration_list optional_declaration_list
                         initialization_bitfield_simple_declarator

%type <declarator> optional_simple_declarator declarator direct_declarator
%type <type_list> optional_pointer_list pointer_list
%type <type> pointer_marker

%type <parameter_pair> optional_parameter_ellipse_list
                       parameter_ellipse_list
%type <symbol_list> parameter_list
%type <symbol> parameter_declaration

%type <string_list> optional_name_list identifier_list
%type <object_list> initializer_list
%type <obj> initializer

%type <type> type_name

%type <declarator> abstract_declarator direct_abstract_declarator

%type <middle_operator> assignment_operator equality_operator
                        relation_operator shift_operator
                        multiply_operator prefix_add_operator
                        increment_operator

%type <expression> optional_constant_integral_expression
                   constant_integral_expression optional_expression
                   expression assignment_expression
                   condition_expression logical_or_expression
                   logical_and_expression BITWISE_OR_expression
                   bitwise_xor_expression bitwise_and_expression
                   equality_expression relation_expression
                   shift_expression add_expression
                   multiply_expression type_cast_expression
                   prefix_expression postfix_expression
                   primary_expression

%type <statement> optional_statement_list statement
                  closed_statement opened_statement

%type <expression_list> optional_argument_expression_list
```

```
                              argument_expression_list
```

3.2. Declarations

Now we have reached the last (and largest) part of the parser, where the rules are defined. The **source_code_file** rule is the start of the grammar.

```
%start source_code_file

%%
```

A source code file is made up of a sequence of external declarations, which may be function definitions or regular declarations.

```
source_code_file:
    external_declaration
  | source_code_file external_declaration;

external_declaration:
    function_definition
 |  declaration;
```

3.2.1. Function Definition

A function definition is made up by a declarator, possible preceded by a declaration specifier list, followed by an optional declaration list, and a block with an optional statement list. For instance, in the function below **unsigned long int** is the declaration specifier list, **square(int value)** is the declarator, and **return value * value ;** is the only statement in the statement list.

```
unsigned long int square(int value) { return value * value; }
```

In C, there are two ways to define a function:

- **The old way**. The parameter list hold holds the names of the parameters and their types are defined afterwards.
- **The new way**. The parameter list holds the names and types of the parameters.

In the following examples, **f** is defined in the old way and **g** is defined in the new way.

```
double f(a, b)
int a;
double b; {
  return a + b;
}

double f(int a, double b) {
  return a + b;
}
```

The first part of the function definition rule handles the case where there is specifier list of at least one specifier. There are three methods to handle the function definition: **FunctionHeader** that handles the header of the function, **FunctionDefinition** that makes sure the function is defined in either the old or new way, and **FunctionEnd** that generates the assembly code of the function and saves it in the global static set.

MainParser.gppg

```
function_definition:
    declaration_specifier_list_x declarator {
      MiddleCodeGenerator.FunctionHeader
        (SpecifierStack.Pop(), $2);
    }
    optional_declaration_list {
      MiddleCodeGenerator.FunctionDefinition();
    }
    LEFT_BLOCK optional_declaration_list optional_statement_list RIGHT_BLOCK {
      $8.CodeList.InsertRange(0, $7);
      MiddleCodeGenerator.BackpatchGoto();
      MiddleCodeGenerator.FunctionEnd($8);
    }
```

The second rule handles the case where there is no specifier list. In that case, the function return type becomes a signed integer.

```
  | declarator {
      MiddleCodeGenerator.FunctionHeader(null, $1);
    }
    optional_declaration_list {
      MiddleCodeGenerator.FunctionDefinition();
    }
    LEFT_BLOCK optional_declaration_list optional_statement_list RIGHT_BLOCK {
      $7.CodeList.InsertRange(0, $6);
      MiddleCodeGenerator.BackpatchGoto();
      MiddleCodeGenerator.FunctionEnd($7);
    };

optional_declaration_list:
    /* Empty */      { $$ = new List<MiddleCode>(); }
  | declaration_list { $$ = $1; };
```

3.2.2. Specifier List

A declaration is made up by a declaration specifier list followed by an optional declarator list, and a semicolon. For instance, in the following declaration, **static long int** is the specifier list while ***p**, and **a[3]** are declarators.

```
static long int *p, a[3];
```

There is possible to define a declaration without a declarator list. This feature is useful when declare a named struct or union:

```
struct _s {int i};
```

However, this feature makes it also possible to declare unnamed specifier lists. The following declaration are syntactically correct but without meaning, they will result in no generated code.

```
unsigned short int;
union {short s;}
```

But the following declaration has meaning since the enumeration values can be used.

```
enum {ZERO = 0, ONE = 1, TEN = 10};
```

MainParser.gppg

```
declaration:
    declaration_specifier_list SEMICOLON {
      SpecifierStack.Push(Specifier.SpecifierList($1));
```

```
        $$ = new List<MiddleCode>();
      }
    | declaration_specifier_list_x declarator_list SEMICOLON {
        SpecifierStack.Pop();
        $$ = $2;
      };

optional_declaration_specifier_list_x:
      /* empty */
    | declaration_specifier_list_x;
```

Due to parser-specific limitations we add an extra rule for the specifier list and push the list at specifier stack. The stack is inspected for each of the following declarator. It is necessary to decide the type completely in case of initialization of the declarator.

```
declaration_specifier_list_x:
    declaration_specifier_list {
      SpecifierStack.Push(Specifier.SpecifierList($1));
    };
```

A declaration specifier list is made up by one or more declaration specifiers, which are stored in a list.

```
declaration_specifier_list:
      declaration_specifier {
        $$ = new List<object>();
        $$.Add($1);
      }
    | declaration_specifier declaration_specifier_list {
        $2.Add($1);
        $$ = $2;
      };
```

There is a set of different kind of specifiers. The following specifiers return a value of the **Mask** enumeration.

- Storage specifiers: **auto**, **register**, **static**, **extern**, or **typedef**
- Type qualifiers: **constant** or **volatile**
- Type specifiers: **void**, **char**, **short**, **int**, **long**, **float**, **double**, **signed**, or **unsigned**

The following specifiers return a reference to the **Type** class. Note that a specifier can be a struct, union, or enumeration declaration, or a typedef name as well as a keyword.

```
declaration_specifier:
      AUTO                      { $$ = Mask.Auto;     }
    | REGISTER                  { $$ = Mask.Register; }
    | STATIC                    { $$ = Mask.Static;   }
    | EXTERN                    { $$ = Mask.Extern;   }
    | TYPEDEF                   { $$ = Mask.Typedef;  }
    | CONSTANT                  { $$ = Mask.Constant; }
    | VOLATILE                  { $$ = Mask.Volatile; }
    | VOID                      { $$ = Mask.Void;     }
    | CHAR                      { $$ = Mask.Char;     }
    | SHORT                     { $$ = Mask.Short;    }
    | INT                       { $$ = Mask.Int;      }
    | LONG                      { $$ = Mask.Long;     }
    | FLOAT                     { $$ = Mask.Float;    }
    | DOUBLE                    { $$ = Mask.Double;   }
    | SIGNED                    { $$ = Mask.Signed;   }
```

```
  | UNSIGNED                  { $$ = Mask.Unsigned; }
  | struct_or_union_specifier { $$ = $1;            }
  | enum_specifier            { $$ = $1;            }
  | TYPEDEF_NAME              { $$ = $1;            };
```

3.2.3. Structs and Unions

A struct or union specifier holds an optional name and a declaration list within brackets. It can also refer to a declared struct or union by its name. When the struct or union members are parsed, the struct or union is given a symbol table of its own and each member becomes added to symbol table.

MainParser.gppg

The call to **StructOrUnionHeader** adds the struct or union to the symbol table if the optional tag name is not null. We add the tag name to the current symbol table before we parse the members, for recursive pointers to work. For instance,

```
struct Cell {
  int value;
  struct Cell* next;
}
```

If we do not add the tag, there is a risk that the **next** field in the code above will point to another struct with the same name, defined in a surrounding block or in global space.

```
struct_or_union_specifier:
    struct_or_union optional_name {
      if ($2 != null) {
        SymbolTable.CurrentTable.AddTag($2, new CCompiler.Type($1));
      }
```

Before the parsing of the members we assign them a symbol table of their own.

```
      SymbolTable.CurrentTable =
        new SymbolTable(SymbolTable.CurrentTable, (Scope) $1);
    }
    LEFT_BLOCK declaration_list RIGHT_BLOCK {
      $$ = MiddleCodeGenerator.StructUnionSpecifier($2, $1);
```

After the parsing of the member list of the struct, we restore the original symbol table.

```
      SymbolTable.CurrentTable = SymbolTable.CurrentTable.ParentTable;
    }
```

In case of a struct of union without a declaration list, but with an obligatory name, we look up the name.

```
  | struct_or_union NAME {
      $$ = MiddleCodeGenerator.LookupStructUnionSpecifier($2, $1);
    };

struct_or_union:
    STRUCT { $$ = Sort.Struct; }
  | UNION  { $$ = Sort.Union;  };

optional_name:
    /* Empty */ { $$ = null; }
  | NAME        { $$ = $1;   };

declaration_list:
    declaration {
      $$ = $1;
    }
  | declaration_list declaration {
      $1.AddRange($2);
      $$ = $1;
    };
```

3.2.4. Enumeration

An enumeration declaration may hold a list of enumeration items, which are assigned values. Each item is declared as a constant signed integer, with a value implicitly or explicitly assigned.

MainParser.gppg

```
enum_specifier:
    ENUM optional_name {
      EnumValueStack.Push(BigInteger.Zero);
    }
    LEFT_BLOCK enum_list RIGHT_BLOCK {
      EnumValueStack.Pop();
      $$ = MiddleCodeGenerator.EnumSpecifier($2, $5);
    }
```

An enumeration declaration may also hold a name without an enumeration item list. In that case, we look up the name and returns its type, which is a constant signed integer. If the name of the enumeration does not exist, we report an error.

```
  | ENUM NAME {
      $$ = MiddleCodeGenerator.LookupEnum($2);
    };
```

The result of **enum_list** is a set holding the symbols of the enumeration items and a Boolean value indicating whether the item was explicitly assigned. We need the Boolean value in case the enumeration symbol is given external storage, since assignment of extern values are not allowed.

```
enum_list:
    enum {
      ISet<Pair<Symbol,bool>> memberSet = new HashSet<Pair<Symbol,bool>>();
      memberSet.Add($1);
      $$ = memberSet;
    }
  | enum_list COMMA enum {
      ISet<Pair<Symbol,bool>> memberSet = $1;
      memberSet.Add($3);
      $$ = memberSet;
    };
```

When it comes to enumerations, we have a potential problem. Each enumeration item is stored in the symbol table as a signed integer with a value. As stated above, the value may be explicitly assigned or implicitly given a value. If the storage of the enumeration is extern, it is not allowed to assign values to the enumeration items. However, it is in C allowed to state the declaration specifiers in arbitrary order. For instance, the following declaration is valid:

```
enum {a, b} extern;
```

The following declaration is invalid, since it has extern storage and **a** is assigned a value:

```
enum {a = 1, b} extern;
```

The problem is that when we parse enumeration declaration we might not yet know if its storage is extern. Therefore, we must also store, for each item, whether it has been explicitly assigned. The handling of the storage is managed by the **Specifier** class (see Chapter 4) after all declaration specifiers have been parsed. For each item a pair is returned, holding the symbol of the item and a Boolean value indicating whether the item has been explicitly assigned a value.

```
enum:
    NAME {
      Symbol symbol = MiddleCodeGenerator.EnumItem($1, null);
      $$ = new Pair<Symbol,bool>(symbol, false);
    }
  | NAME ASSIGN constant_integral_expression {
      Symbol symbol = MiddleCodeGenerator.EnumItem($1, $3.Symbol);
      $$ = new Pair<Symbol,bool>(symbol, true);
    };
```

3.2.5. Declarators

As mentioned in the previous section, a declaration is made up by a declaration specifier list, an optional declarator list, and a semicolon. In the following declaration, **static long int** is the specifier list while ***p**, and **a[3]** are declarators.

```
static long int *p, a[3];
```

A declarator list is made up declarators separated by commas. There are three kinds of declarators:

- Simple, such as **int i;**
- With initialization, such as **int i = 3;**
- With bitfield, only allowed in structs and unions, such as **int i : 3;**. It also possible to omit the declarator; in the **int : 3;** declaration, it is only stated that three bits shall be unused.

Note that it is not possible to combine initializations with bitfield markers, since bitfields are only allowed as struct or union members that cannot be initialized.

MainParser.gppg

```
declarator_list:
    initialization_bitfield_declarator {
      $$ = $1;
    }
  | declarator_list COMMA initialization_bitfield_declarator {
      $1.AddRange($3);
      $$ = $1;
    };
```

When calling the declarator methods in the **MiddleCodeDeclarator** class, we submit the specifier list at the top of the specifier list stack, in order to build the complete type. In some cases, the specifier list constitutes the type; for instance, in **int i**, the integer is the complete type. In other cases, such as **int *p** or **f(int)**, the pointer to integer and the function return integer are the complete types.

```
initialization_bitfield_declarator:
    declarator {
      MiddleCodeGenerator.Declarator(SpecifierStack.Peek(), $1);
      $$ = new List<MiddleCode>();
    }
  | declarator ASSIGN initializer {
      $$ = MiddleCodeGenerator.InitializedDeclarator
```

```
            (SpecifierStack.Peek(), $1, $3);
    }
  | optional_declarator COLON constant_integral_expression {
      MiddleCodeGenerator.BitfieldDeclarator
        (SpecifierStack.Peek(), $1, $3.Symbol);
      $$ = new List<MiddleCode>();
    };
```

3.2.6. Pointer Declarators

A pointer declarator is made up by an optional list of pointer markers and a direct declarator. A pointer marker is made up by an optional qualifier (constant or volatile) list and an asterisk ('*'). In other words, each asterisk can be qualified by the word **const** and **volatile**, and it is possible to add more than one pointer marker to establish a pointer-to-pointer effect.

MainParser.gppg

```
declarator:
    optional_pointer_list direct_declarator {
      $$ = MiddleCodeGenerator.PointerDeclarator($1, $2);
    };

optional_pointer_list:
    /* Empty */  { $$ = new List<CCompiler.Type>(); }
  | pointer_list { $$ = $1; };

pointer_list:
    pointer_marker {
      $$ = new List<CCompiler.Type>();
      $$.Add($1);
    }
  | pointer_list pointer_marker {
      $1.Add($2);
      $$ = $1;
    };

pointer_marker:
    ASTERRISK optional_qualifier_list {
      $$ = Specifier.QualifierList($2);
    };
```

A pointer can be qualified with **constant** and **volatile**. Note the difference between a constant pointer and a pointer to a constant value. In case of a constant pointer the pointer itself is constant, it cannot be assigned another address. In case of a pointer to a constant value the pointer can be assigned to another address, but the value it points to is always constant. Naturally, a constant pointer can also point at a constant value.

`int * const p;`	`const int * p;`	`const int const * p;`
(a) Constant pointer	(b) Pointer to constant value	(c) Constant pointer to constant value

```
optional_qualifier_list:
    /* Empty */    {
      $$ = new List<Mask>();
    }
  | optional_qualifier_list qualifier {
      $$ = $1;
      $$.Add($2);
    };
```

```
qualifier:
   CONSTANT { $$ = Mask.Constant; }
   VOLATILE { $$ = Mask.Volatile; };
```

3.2.7. Direct Declarator

The next step after we have handled pointer markers is the direct declarator, which may hold an array or function declaration, or just a single name.

```
direct_declarator:
    NAME {
      $$ = new Declarator($1);
    }
```

A declarator may also another declarator enclosed by parenthesis, in which case we simply return the declarator. The parentheses are present only to change the precedence of the declarator.

```
  | LEFT_PARENTHESIS declarator RIGHT_PARENTHESIS {
      $$ = $2;
    }
```

A direct declarator can be an array declarator, with a constant expression enclosed by brackets.

```
  | direct_declarator LEFT_SQUARE
    optional_constant_integral_expression RIGHT_SQUARE {
      $$ = MiddleCodeGenerator.ArrayType($1, $3);
    }
```

Similar to the pointer case above, it is possible to omit the declarator of an array declaration. It is also possible to omit the expression within the brackets. However, the brackets must always be present in an array declaration.

MainParser.gppg

```
  | direct_declarator
    LEFT_PARENTHESIS parameter_ellipse_list RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.
           NewFunctionDeclaration($1, $3.First, $3.Second);
    }
```

The next declarator is the old-style function declarator, where the parameter list is a list of strings.

MainParser.gppg

```
  | direct_declarator LEFT_PARENTHESIS
    optional_name_list RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.OldFunctionDeclaration($1, $3);
    };

optional_parameter_ellipse_list:
    /* Empty */            { $$ = null; }
  | parameter_ellipse_list { $$ = $1;   };
```

The **parameter_ellipse_list** rule returns a pair of the parameter list and a Boolean value representing the presence of the ellipse marker.

```
parameter_ellipse_list:
    parameter_list {
      $$ = new Pair<List<Symbol>,Boolean>($1, false);
    }
  | parameter_list COMMA ELLIPSE {
      $$ = new Pair<List<Symbol>,Boolean>($1, true);
    };
```

For each parameter declaration, we push the scope of the current symbol table on a stack that we pop after the parsing of the declaration. This is because there are special rules for the storage specifiers of parameters (only **auto** or **register** allowed) for the **Specifier** class to handle (see Chapter 5).

```
parameter_list:
    { ScopeStack.Push(SymbolTable.CurrentTable.Scope);
      SymbolTable.CurrentTable.Scope = Scope.Parameter;
    }
    parameter_declaration {
      SymbolTable.CurrentTable.Scope = ScopeStack.Pop();
      $$ = new List<Symbol>();
      $$.Add($2);
    }
  | parameter_list COMMA {
      ScopeStack.Push(SymbolTable.CurrentTable.Scope);
      SymbolTable.CurrentTable.Scope = Scope.Parameter;
    }
    parameter_declaration {
      SymbolTable.CurrentTable.Scope = ScopeStack.Pop();
      $1.Add($4);
      $$ = $1;
    };
```

A parameter declaration can be made up by a declaration specifier list together with a declarator or an abstract declarator, or only a declaration specifier list since function parameters do not need to be named.

```
parameter_declaration:
    declaration_specifier_list {
      $$ = MiddleCodeGenerator.Parameter(Specifier.SpecifierList($1), null);
    }
  | declaration_specifier_list_x declarator {
      $$ = MiddleCodeGenerator.Parameter(SpecifierStack.Pop(), $2);
    }
  | declaration_specifier_list_x abstract_declarator {
      $$ = MiddleCodeGenerator.Parameter(SpecifierStack.Pop(), $2);
    };

optional_name_list:
    /* Empty */ { $$ = new List<string>(); }
  | name_list   { $$ = $1;                  };

name_list:
    NAME {
      $$ = new List<string>();
      $$.Add($1);
    }
  | name_list COMMA NAME {
      $1.Add($3);
      $$ = $1;
    };
```

3.2.8. Initialization

The initializer can be either an assignment expression or a block, which is made up by assignment expressions or other blocks. The assignment expression makes sure that the expression cannot hold assignment or comma.

```
initializer:
    assignment_expression {
      $$ = $1;
    }
```

The block holds a list of initializers, which means that initializers can be nested. However, at the deepest level, there are always expressions in the initialization lists.

```
  | LEFT_BLOCK initializer_list optional_comma RIGHT_BLOCK {
      $$ = $2;
    };

optional_comma:
    /* Empty */
  | COMMA;

initializer_list:
    initializer {
      $$ = new List<object>();
      $$.Add($1);
    }
  | initializer_list COMMA initializer {
      $1.Add($3);
      $$ = $1;
    };
```

3.2.9. Abstract Declarator

An abstract declarator is a direct declarator without a name, which is allowed in function parameters and as operand of the **sizeof** operator. It works in the same way as the regular declarator. The only difference is that an abstract declarator cannot be made up by a name.

```
abstract_declarator:
    pointer_list {
      $$ = MiddleCodeGenerator.PointerDeclarator ($1, null);
    }
  | optional_pointer_list direct_abstract_declarator {
      $$ = MiddleCodeGenerator.PointerDeclarator($1, $2);
    };

direct_abstract_declarator:
    LEFT_PARENTHESIS abstract_declarator RIGHT_PARENTHESIS {
      $$ = $2;
    }
  | LEFT_SQUARE optional_constant_integral_expression RIGHT_SQUARE {
      $$ = MiddleCodeGenerator.ArrayType(null, $2);
    }
  | direct_abstract_declarator
    LEFT_SQUARE optional_constant_integral_expression RIGHT_SQUARE {
      $$ = MiddleCodeGenerator.ArrayType($1, $3);
    }
  | LEFT_PARENTHESIS optional_parameter_ellipse_list RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.
           NewFunctionDeclaration(null, $2.First, $2.Second);
    }
  | direct_abstract_declarator
    LEFT_PARENTHESIS optional_parameter_ellipse_list RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.
           NewFunctionDeclaration($1, $3.First, $3.Second);
    };
```

3.3. Statements

The next part of the parser is the result for statements.

3.3.1. The if-else Problem

The **if-else** problem is the problem of syntactically interpret the leftmost source code below. Semantically, the middle interpretation of the left statement is the correct one, each **else** shall be connected to the latest preceding **if**.

```
if (a < b)
  if (c < d)
    e = 1;
  else
    f = 2;
```

(a) Ambiguous C code

```
if (a < b) {
  if (c < d)
    e = 1;
  else
    f = 2;
}
```

(b) Correct Interpretation

```
if (a < b)
  if (c < d) {
    e = 1;
  }
else
  f = 2;
```

(c) Incorrect interpretation

Below is a simple set of statement rules. Unfortunately, they are ambiguous in that way that the an **else** does not have to be connected to the latest preceding **if**, resulting in both the middle and rightmost semantically interpretation above, depending in which order the rules are applied.

```
statement ::=
    IF LEFT_PAREN logical_expression RIGHT_PAREN statement
  | IF LEFT_PAREN logical_expression RIGHT_PAREN statement ELSE statement
  | ...
```

To solve the problem, we need a more complicated set of rules that works with open and closed statements. The following set is unambiguous in that way that it always connects each **else** with the latest preceding **if**.

```
statement:
    opened_statement
  | closed_statement

opened_statement:
    if ( expression ) statement
  | if ( expression ) closed_statement else opened_statement
  | switch ( expression ) opened_statement
  | case constant_integral_expression : opened_statement
  | while ( expression ) opened_statement
  | for ( optional_expression ; optional_expression ; optional_expression )
    opened_statement
  | name : opened_statement

closed_statement:
    if ( expression ) closed_statement else closed_statement
  | switch ( expression ) closed_statement
  | while ( expression ) closed_statement
  | for ( optional_expression ; optional_expression ; optional_expression )
    closed_statement
  | do statement while ( expression ) ;
  | case constant_integral_expression : closed_statement
  | default : closed_statement
  | continue ;
  | break ;
  | { optional_statement_list }
  | goto name ;
  | return optional_expression ;
  | optional_expression ;
  | declaration ;
  | jump_register ( register_name ) ;
  | interrupt ( constant_integral_expression ) ;
  | syscall ( ) ;
```

3.3.2. The Statement Class

Let us look at the **Statement** class. A statement holds a list of middle code instructions, and the **next-set**, which is a set of jump instructions that shall be backpatched to jump to the next statement after the current statement.

Statement.cs

```
using System.Collections.Generic;

namespace CCompiler {
  public class Statement {
    private List<MiddleCode> m_codeList;
    private ISet<MiddleCode> m_nextSet;

    public Statement(List<MiddleCode> codeList,
                     ISet<MiddleCode> nextSet = null) {
      Assert.ErrorXXX(codeList != null);
      m_codeList = codeList;
      m_nextSet = (nextSet != null) ? nextSet : (new HashSet<MiddleCode>());
    }

    public List<MiddleCode> CodeList {
      get { return m_codeList; }
    }

    public ISet<MiddleCode> NextSet {
      get { return m_nextSet; }
    }
  }
}
```

3.3.3. Statements

As stated above, a statement can be an open or a closed statement.

MainParser.cs

```
statement:
    opened_statement { $$ = $1; }
  | closed_statement { $$ = $1; };
```

3.3.4. The If Statement

```
opened_statement:
    IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement {
      $$ = MiddleCodeGenerator.IfStatement($3, $5);
    };
```

3.3.1. The If-Else Statement

```
opened_statement:
    IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS closed_statement
    ELSE opened_statement {
      $$ = MiddleCodeGenerator.IfElseStatement($3, $5, $7);
    };

closed_statement:
    IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS closed_statement
    ELSE closed_statement {
      $$ = MiddleCodeGenerator.IfElseStatement($3, $5, $7);
    };
```

3.3.2. The Switch Statement

The switch statement needs a header the is parsed before the statement to initialize the switch statement parsing.

```
switch_header:
    /* Empty. */ { MiddleCodeGenerator.SwitchHeader(); };
```

The switch statement occours in both the opened and closed statements.

```
opened_statement:
    SWITCH switch_header LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
    opened_statement {
      $$ = MiddleCodeGenerator.SwitchStatement($4, $6);
    };

closed_statement:
    SWITCH switch_header LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
    closed_statement {
      $$ = MiddleCodeGenerator.SwitchStatement($4, $6);
    };

opened_statement:
    CASE constant_integral_expression COLON opened_statement {
      $$ = MiddleCodeGenerator.CaseStatement($2, $4);
    };

closed_statement:
    CASE constant_integral_expression COLON closed_statement {
      $$ = MiddleCodeGenerator.CaseStatement($2, $4);
    };

closed_statement:
    DEFAULT COLON closed_statement {
      $$ = MiddleCodeGenerator.DefaultStatement($3);
    };
```

3.3.1. The Loop Statements

Similar to the switch statement above, the while, do, and for statements need a header to initialize the statements parsing.

```
loop_header:
    /* Empty. */ { MiddleCodeGenerator.LoopHeader(); };

opened_statement:
    WHILE loop_header LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
    opened_statement {
      $$ = MiddleCodeGenerator.WhileStatement($4, $6);
    };

closed_statement:
    WHILE loop_header LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
    closed_statement {
      $$ = MiddleCodeGenerator.WhileStatement($4, $6);
    };

closed_statement:
    DO loop_header statement WHILE
    LEFT_PARENTHESIS expression RIGHT_PARENTHESIS SEMICOLON {
      $$ = MiddleCodeGenerator.DoStatement($3, $6);
    };

opened_statement:
    FOR loop_header LEFT_PARENTHESIS optional_expression SEMICOLON
    optional_expression SEMICOLON optional_expression RIGHT_PARENTHESIS
    opened_statement {
      $$ = MiddleCodeGenerator.ForStatement($4, $6, $8, $10);
    };

closed_statement:
    FOR loop_header LEFT_PARENTHESIS optional_expression SEMICOLON
    optional_expression SEMICOLON optional_expression RIGHT_PARENTHESIS
    closed_statement {
      $$ = MiddleCodeGenerator.ForStatement($4, $6, $8, $10);
    };
```

3.3.2. Label and Jump Statement

```
opened_statement:
    NAME COLON opened_statement {
      $$ = MiddleCodeGenerator.LabelStatement($1, $3);
    };

closed_statement:
    GOTO NAME SEMICOLON {
      $$ = MiddleCodeGenerator.GotoStatement($2);
    };
```

3.3.3. The Return Statement

The return statement may have an optional expression.

```
closed_statement:
    RETURN optional_expression SEMICOLON {
      $$ = MiddleCodeGenerator.ReturnStatement($2);
    };
```

3.3.4. Optional Expression Statements

A statement can be made up by an optional expression; that is, the statement is an expression followed by a semicolon, or simply a semicolon.

MainParser.gppg

```
closed_statement:
    optional_expression SEMICOLON {
      $$ = MiddleCodeGenerator.ExpressionStatement($1);
    };
```

3.3.5. Block Statements

A statement can be an optional sequence of statements enclosed in brackets. The sequence is parsed with a new symbol table.

```
closed_statement:
    LEFT_BLOCK {
      SymbolTable.CurrentTable =
        new SymbolTable(SymbolTable.CurrentTable, Scope.Block);
    }
    optional_declaration_list optional_statement_list RIGHT_BLOCK {
      SymbolTable.CurrentTable =
        SymbolTable.CurrentTable.ParentTable;
      $4.CodeList.InsertRange(0, $3);
      $$ = $4;
    }
```

In case of an empty statement list, we return a statement with an empty code list and an empty next-set.

```
optional_statement_list:
    /* Empty */ {
      $$ = new Statement(new List<MiddleCode>(),
                         new HashSet<MiddleCode>());
    }
```

In case of a non-empty statement list, we add the code list of the statements of the statement list. For each statement in the list, except the last statement, we backpatch the next-set to the beginning of the code list of the next-set. The result becomes a statement with the total middle code list and the next-set of the last statement.

```
  | optional_statement_list statement {
      MiddleCodeGenerator.Backpatch($1.NextSet, $2.CodeList);
      List<MiddleCode> codeList = new List<MiddleCode>();
      codeList.AddRange($1.CodeList);
      codeList.AddRange($2.CodeList);
      $$ = new Statement(codeList, $2.NextSet);
    };
```

3.3.6. Jump Register Statements

When performing a call to a function which address is stored in a pointer variable, we need to store the address in a register and jump to that register.

```
closed_statement:
    JUMP_REGISTER LEFT_PARENTHESIS REGISTER_NAME RIGHT_PARENTHESIS SEMICOLON {
      $$ = MiddleCodeGenerator.JumpRegisterStatement($3);
    };
```

3.3.7. Interrupt Statements

When making system calls in the Windows environment, an interrupt occurs. The operand is an integral value of short size (1 byte).

```
closed_statement:
    INTERRUPT LEFT_PARENTHESIS constant_integral_expression RIGHT_PARENTHESIS
    SEMICOLON {
      $$ = MiddleCodeGenerator.InterruptStatement($3);
    };
```

3.3.8. System Call Statements

The system calls in the Linux environment do not take any operands.

```
closed_statement:
    SYSCALL LEFT_PARENTHESIS RIGHT_PARENTHESIS SEMICOLON {
      $$ = MiddleCodeGenerator.SystemCallStatement();
    };
```

3.4. Expressions

The third part of the parser is the expressions. We start with the expression of lowest precedence and add a new rule for each new level of precedence.

3.4.1. The Expression Class (Short and Long List)

Similar to the statement case above, we have the Expression class to handle expressions. It handles a symbol, a register, and the **short list** and the **long list**. The short list and long list both hold the middle code of the expression. The difference is that the short list holds only the side effects of the expression, while the long holds the whole expression. For instance, in the following expression the long list holds the function call, decrement, and addition. The short list holds the only the side effects: the function call and decrement, but not the addition.

```
f(x) + a--;
```

The following statement is correct, but since in holds no side-effects its short list will be empty. In the end, no code will be generated.

```
a + (b * c);
```

The register parameter is used in system calls when we need to access or assign a specific register.

Expression.cs
```
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  public class Expression {
    private Symbol m_symbol;
    private List<MiddleCode> m_shortList;
    private List<MiddleCode> m_longList;
    private Register? m_register;

    public Expression(Symbol symbol, List<MiddleCode> shortList,
                      List<MiddleCode> longList, Register? register = null) {
      m_symbol = symbol;
      m_shortList = (shortList != null) ? shortList : (new List<MiddleCode>());
      m_longList = (longList != null) ? longList : (new List<MiddleCode>());
      m_register = register;
    }

    public Symbol Symbol {
      get { return m_symbol; }
    }

    public List<MiddleCode> ShortList {
      get { return m_shortList; }
    }

    public List<MiddleCode> LongList {
      get { return m_longList; }
    }

    public Register? Register {
      get { return m_register; }
    }

    public override string ToString() {
      return m_symbol.ToString();
    }
  }
}
```

3.4.2. Optional Expressions

In expression statements and the **for** statement, the expressions are optional. Therefore, we have a rule for an optional expression. In the absence of an expression, the result is null.

MainParser.cs

```
optional_expression:
    /* Empty */ { $$ = null; }
  | expression  { $$ = $1;   };
```

3.4.3. The Comma Expression

We begin the parsing of the expressions with the operator of lowest precedence, which is the comma operator. A comma expression may be an assignment expression, or two expressions separated by a comma.

```
expression:
   assignment_expression {
      $$ = $1;
    }
  | expression COMMA assignment_expression {
      $$ = MiddleCodeGenerator.CommaExpression($1, $3);
    };
```

3.4.4. The Assignment Expression

There are two kinds of assignment: simple and compound. In the case of compound assignment, we first evaluate the corresponding binary expression (addition in the example below), assign its result to a temporary variable that we then assign to original left operand.

(a) Compound assignment	(b) Equivalent simple assignment

`x += y;`	`t = x + y`
	`x = t`

An assignment expression can be a conditional expression, a simple assignment, or a compound assignment.

MainParser.cs
```
assignment_expression:
    condition_expression {
      $$ = $1;
    }
  | prefix_expression assignment_operator assignment_expression {
      $$ = MiddleCodeGenerator.AssignmentExpression($2, $1, $3);
    };

assignment_operator:
    ASSIGN             { $$ = MiddleOperator.Assign;          }
  | ADD_ASSIGN         { $$ = MiddleOperator.Add;      }
  | SUBTRACT_ASSIGN    { $$ = MiddleOperator.Subtract; }
  | MULTIPLY_ASSIGN    { $$ = MiddleOperator.Multiply; }
  | DIVIDE_ASSIGN      { $$ = MiddleOperator.Divide;   }
  | MODULO_ASSIGN      { $$ = MiddleOperator.Modulo;   }
  | AND_ASSIGN         { $$ = MiddleOperator.BitwiseAnd;      }
  | OR_ASSIGN          { $$ = MiddleOperator.BitwiseOr;       }
  | XOR_ASSIGN         { $$ = MiddleOperator.BitwiseXOr;      }
  | LEFT_SHIFT_ASSIGN  { $$ = MiddleOperator.ShiftLeft;       }
  | RIGHT_SHIFT_ASSIGN { $$ = MiddleOperator.ShiftRight;      };
```

3.4.1. The Condition Expression

The conditional operator is the only operator in C that takes three operands. It applies lazy evaluation, which means that only one of the second or third expression will be evaluated, depending on the value of the first expression.

MainParser.gppg
```
condition_expression:
    logical_or_expression {
      $$ = $1;
    }
  | logical_or_expression QUESTION_MARK expression COLON condition_expression{
      $$ = MiddleCodeGenerator.ConditionalExpression($1, $3, $5);
    };
```

3.4.1. Constant Expression

On several occasions, such array limits and enumeration values, we need to parse constant integral expressions. The expression is a conditional expression since it cannot hold commas or assignments.

MainParser.cs

```
optional_constant_integral_expression:
    /* Empty */                    { $$ = null; }
  | constant_integral_expression { $$ = $1;   };

constant_integral_expression:
    condition_expression {
      $$ = MiddleCodeGenerator.ConstantIntegralExpression($1);
    };
```

3.4.1. Logical Expressions

The logical **and** and **or** operators have one rule each, since **and** have higher precedence than **or**.

MainParser.cs

```
logical_or_expression:
    logical_and_expression {
      $$ = $1;
    }
  | logical_or_expression LOGICAL_OR logical_and_expression {
      $$ = MiddleCodeGenerator.LogicalOrExpression($1, $3);
    };

logical_and_expression:
    bitwise_or_expression {
      $$ = $1;
    }
  | logical_and_expression LOGICAL_AND bitwise_or_expression {
      $$ = MiddleCodeGenerator.LogicalAndExpression($1, $3);
    };
```

3.4.2. Bitwise Expressions

Unlike the logical expressions in the previous section, the bitwise operators do not apply lazy evaluation, which means that the right expression is always evaluated regardless of the value of the left expression. There are three bitwise operators: **or** (inclusive or), **xor** (exclusive or), and **and**. The expressions have one rule each since they have different precedence. They take two integer values and perform operations on each bit:

- **or**: the resulting bit is one if at least one bit is one, zero otherwise
- **xor**: the resulting bit is one if exact one bit (but not both bits) is one, zero otherwise
- **and**: the resulting bit is one if at both values are one, zero otherwise

MainParser.gppg

```
bitwise_or_expression:
    bitwise_xor_expression {
      $$ = $1;
    }
  | bitwise_or_expression BITWISE_OR bitwise_xor_expression {
      $$ = MiddleCodeGenerator.BitwiseExpression
           (MiddleOperator.BitwiseOr, $1, $3);
    };

bitwise_xor_expression:
    bitwise_and_expression {
      $$ = $1;
    }
  | bitwise_xor_expression BITWISE_XOR bitwise_and_expression {
      $$ = MiddleCodeGenerator.BitwiseExpression
           (MiddleOperator.BitwiseXOr, $1, $3);
    };

bitwise_and_expression:
    equality_expression {
      $$ = $1;
    }
  | bitwise_and_expression AMPERSAND equality_expression {
      $$ = MiddleCodeGenerator.BitwiseExpression
           (MiddleOperator.BitwiseAnd, $1, $3);
    };
```

3.4.1. Shift Expression

The left and right shift expression must hold integral types. The right expression is type cast to a one-byte integer value.

MainParser.gppg

```
shift_expression:
    add_expression {
      $$ = $1;
    }
  | shift_expression shift_operator add_expression {
      $$ = MiddleCodeGenerator.ShiftExpression($2, $1, $3);
    };

shift_operator:
    LEFT_SHIFT  { $$ = MiddleOperator.ShiftLeft;  }
  | RIGHT_SHIFT { $$ = MiddleOperator.ShiftRight; };
```

3.4.2. Equality and Relation Expressions

Values of all types except structs or unions can be compared with equality and inequality operator, even though arrays and functions are converted to pointers before the comparation. The equality operators have higher precedence than the relation operators, but they both call **RelationalExpression** in **MiddleCodeGenerator**.

MainParser.gppg

```
equality_expression:
    relation_expression {
      $$ = $1;
    }
  | equality_expression equality_operator relation_expression {
      $$ = MiddleCodeGenerator.RelationalExpression($2, $1, $3);
    };

equality_operator:
    EQUAL     { $$ = MiddleOperator.Equal;    }
  | NOT_EQUAL { $$ = MiddleOperator.NotEqual; };

relation_expression:
    shift_expression {
      $$ = $1;
    }
  | relation_expression relation_operator shift_expression {
      $$ = MiddleCodeGenerator.RelationalExpression ($2, $1, $3);
    };

relation_operator:
    LESS_THAN          { $$ = MiddleOperator.LessThan;          }
  | LESS_THAN_EQUAL    { $$ = MiddleOperator.LessThanEqual;     }
  | GREATER_THAN       { $$ = MiddleOperator.GreaterThan;       }
  | GREATER_THAN_EQUAL { $$ = MiddleOperator.GreaterThanEqual; };
```

3.4.3. Addition and Subtraction Expression

The addition and subtraction expression are a bit more complicated since we need to take pointer arithmetic into consideration.

MainParser.gppg

```
add_expression:
    multiply_expression {
      $$ = $1;
    }
  | add_expression PLUS multiply_expression {
      $$ = MiddleCodeGenerator.AdditionExpression($1, $3);
    }
  | add_expression MINUS multiply_expression {
      $$ = MiddleCodeGenerator.SubtractionExpression($1, $3);
    };
```

3.4.4. Multiplication Expressions

The multiplication expression takes the multiply, divide, and module operators.

MainParser.gppg

```
multiply_expression:
    type_cast_expression {
      $$ = $1;
    }
  | multiply_expression multiply_operator type_cast_expression {
      $$ = MiddleCodeGenerator.MultiplyExpression($2, $1, $3);
    };
```

The multiplication operators are multiply, division, and modulo.

```
multiply_operator:
    ASTERRISK { $$ = MiddleOperator.Multiply; }
  | DIVIDE    { $$ = MiddleOperator.Divide;   }
  | MODULO    { $$ = MiddleOperator.Modulo;   };
```

3.4.5. Type Cast Expressions

A type cast expression is a type name within parentheses followed by any kind of expression.

MainParser.gppg

```
type_cast_expression:
    prefix_expression {
      $$ = $1;
    }
  | LEFT_PARENTHESIS type_name RIGHT_PARENTHESIS type_cast_expression {
      $$ = MiddleCodeGenerator.CastExpression($2, $4);
    };
```

The type name is a declaration specifier list with or without an abstract declarator.

```
type_name:
    declaration_specifier_list {
      $$ = MiddleCodeGenerator.TypeName(Specifier.SpecifierList($1), null);
    }
  | declaration_specifier_list {
      SpecifierStack.Push(Specifier.SpecifierList($1));
    }
    abstract_declarator {
      $$ = MiddleCodeGenerator.TypeName(SpecifierStack.Pop(), $3);
    };
```

3.4.6. Prefix Expression

In C, there are several prefix expressions:

- Unary add and minus
- Logical not
- Bitwise not
- The **sizeof** operator
- The address operator
- The dereference operator
- Prefix increment and decrement

```
prefix_expression:
    postfix_expression {
      $$ = $1;
    }
  | prefix_add_operator type_cast_expression {
      $$ = MiddleCodeGenerator.UnaryExpression($1, $2);
    }
  | LOGICAL_NOT type_cast_expression {
      $$ = MiddleCodeGenerator.LogicalNotExpression($2);
    }
  | BITWISE_NOT type_cast_expression {
      $$ = MiddleCodeGenerator.BitwiseNotExpression($2);
    }
```

The **sizeof** operator can be applied to an expression or a type name within parentheses. It does not apply to functions or bitfields. When applied to an array, it gives the total size of the array, not the size of its address.

```
  | SIZEOF prefix_expression {
      $$ = MiddleCodeGenerator.SizeOfExpression($2);
    }
  | SIZEOF LEFT_PARENTHESIS type_name RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.SizeOfType($3);
    }
```

The address operator does not apply to bitfields symbols or symbols with register storage.

```
    AMPERSAND type_cast_expression {
      $$ = MiddleCodeGenerator.AddressExpression($2);
    };
```

The dereference operator takes a pointer or array as operand.

```
  | ASTERRISK type_cast_expression {
      $$ = MiddleCodeGenerator.DereferenceExpression($2);
    }
```

The postfix increment and decrement change the operand and return the original value. The operand has to be a left-value of arithmetic or pointer type.

```
  | increment_operator prefix_expression {
      $$ = MiddleCodeGenerator.PrefixIncrementExpression($1, $2);
    };

prefix_add_operator:
    PLUS  { $$ = MiddleOperator.Plus;      }
  | MINUS { $$ = MiddleOperator.Minus; };

increment_operator:
    INCREMENT { $$ = MiddleOperator.Increment; }
  | DECREMENT { $$ = MiddleOperator.Decrement; };
```

3.4.7. Postfix Expression

In C, there are several postfix expressions

- Prefix increment and decrement
- Dot and Arrow
- Array indexing
- Function calls

```
postfix_expression:
    primary_expression {
      $$ = $1;
    }
  | postfix_expression increment_operator {
      $$ = MiddleCodeGenerator.PostfixIncrementExpression($2, $1);
    }
```

The dot and arrow operator takes a pointer to a struct or union, and the name of one of their members as operands.

```
  | postfix_expression DOT NAME {
      $$ = MiddleCodeGenerator.DotExpression($1, $3);
    }
  | postfix_expression ARROW NAME {
      $$ = MiddleCodeGenerator.ArrowExpression($1, $3);
    }
  | postfix_expression LEFT_SQUARE expression RIGHT_SQUARE {
      $$ = MiddleCodeGenerator.IndexExpression($1, $3);
    }
  | postfix_expression {
      MiddleCodeGenerator.FunctionPreCall($1);
    }
    LEFT_PARENTHESIS optional_argument_expression_list RIGHT_PARENTHESIS {
      $$ = MiddleCodeGenerator.CallExpression($1, $4);
    };
```

The arguments of the function call are gathered in a list.

```
argument_expression_list:
    assignment_expression {
      $$ = new List<Expression>();
      $$.Add(MiddleCodeGenerator.ArgumentExpression(0, $1));
    }
  | argument_expression_list COMMA assignment_expression {
      $1.Add(MiddleCodeGenerator.ArgumentExpression($1.Count, $3));
      $$ = $1;
    };
```

3.4.8. Primary Expressions

We have finally reached the last kind of expression. A primary expression may be another expression within parentheses, a value, a symbol, a register, the carry flag, or the stack top position.

In case of an expression within parentheses, the result is simply the expression. The parentheses are only present to change the precedence of the expression.

```
primary_expression:
    LEFT_PARENTHESIS expression RIGHT_PARENTHESIS {
      $$ = $2;
    }
  | VALUE {
      $$ = MiddleCodeGenerator.ValueExpression($1);
    }
  | NAME {
      $$ = MiddleCodeGenerator.NameExpression($1);
    }
  | REGISTER_NAME {
      $$ = MiddleCodeGenerator.RegisterExpression($1);
    }
```

The carry flag is also only used internally in conjunction with system calls. On some occasion the interrupt call returns information stored in the carry flag.

```
  | CARRY_FLAG {
      $$ = MiddleCodeGenerator.CarryFlagExpression();
    }
```

The address of the top position of the call stack is needed by the **malloc** function in the C standard library, see Appendix 14.

```
  | STACK_TOP {
      $$ = MiddleCodeGenerator.StackTopExpression();
    };
```

4. Middle Code Generation

The methods of this chapter are called by the parser of the previous chapter. The rules of the parser of the previous chapter call methods of the **MiddleCodeGenerator** class of this chapter to perform type checking, construct the symbol table, and generate the middle code.

4.1. The MiddleCode Class

The middle code of the compiler in this book is **three-address code**. As the name implies, each instruction is made up by one operator and at most three addresses, which are references to objects of the **Object** class, or its sub classes, such as **Symbol**, **Register**, **String**, **Integer**, or **Double**.

MiddleCode.cs
```
using System;
using System.Text;
using System.Collections.Generic;
```

The **MiddleCode** class holds the operator and the array of the three operands. It also holds a set of test methods.

```
namespace CCompiler {
  public class MiddleCode {
    private MiddleOperator m_middleOperator;
    private object[] m_operandArray = new object[3];

    public MiddleCode(MiddleOperator middleOp, object operand0 = null,
                      object operand1 = null, object operand2 = null) {
      m_middleOperator = middleOp;
      m_operandArray[0] = operand0;
      m_operandArray[1] = operand1;
      m_operandArray[2] = operand2;
    }

    public MiddleOperator Operator {
      get { return m_middleOperator; }
      set { m_middleOperator = value; }
    }

    public object this[int index] {
      get { return m_operandArray[index]; }
      set { m_operandArray[index] = value;  }
    }

    public void Clear() {
      m_middleOperator = MiddleOperator.Empty;
      m_operandArray[0] = null;
      m_operandArray[1] = null;
      m_operandArray[2] = null;
    }
```

The **IsGoto**, **IsCarry**, and **IsRelation** methods test whether the middle code is a jump instruction.

```
    public bool IsGoto() {
      return (m_middleOperator == MiddleOperator.Jump);
    }

    public bool IsCarry() {
      return (m_middleOperator == MiddleOperator.Carry) ||
             (m_middleOperator == MiddleOperator.NotCarry);
    }

    public static bool IsRelation(MiddleOperator middleOperator) {
      switch (middleOperator) {
        case MiddleOperator.Case:
        case MiddleOperator.Equal:
        case MiddleOperator.NotEqual:
        case MiddleOperator.LessThan:
        case MiddleOperator.LessThanEqual:
        case MiddleOperator.GreaterThan:
        case MiddleOperator.GreaterThanEqual:
          return true;

        default:
          return false;
      }
    }

    public bool IsRelation() {
      return IsRelation(m_middleOperator);
    }

    public bool IsRelationCarryOrGoto() {
      return IsRelation() || IsCarry() || IsGoto();
    }
```

The **IsBinary**, **IsCommutative**, and **IsShift** methods test the operator of the middle code.

```
    public bool IsBinary() {
      switch (m_middleOperator) {
        case MiddleOperator.Add:
        case MiddleOperator.Subtract:
        case MiddleOperator.Multiply:
        case MiddleOperator.Divide:
        case MiddleOperator.Modulo:
        case MiddleOperator.LogicalOr:
        case MiddleOperator.LogicalAnd:
        case MiddleOperator.BitwiseOr:
        case MiddleOperator.BitwiseXOr:
        case MiddleOperator.BitwiseAnd:
        case MiddleOperator.ShiftLeft:
        case MiddleOperator.ShiftRight:
          return true;

        default:
          return false;
      }
    }

    public bool IsCommutative() {
      switch (m_middleOperator) {
        case MiddleOperator.Add:
```

```
        case MiddleOperator.Multiply:
        case MiddleOperator.BitwiseOr:
        case MiddleOperator.BitwiseXOr:
        case MiddleOperator.BitwiseAnd:
          return true;

        default:
          return false;
      }
    }

    public static bool IsMultiply(MiddleOperator middleOp) {
      switch (middleOp) {
        case MiddleOperator.Multiply:
        case MiddleOperator.Divide:
        case MiddleOperator.Modulo:
          return true;

        default:
          return false;
      }
    }

    public static bool IsShift(MiddleOperator middleOp) {
      switch (middleOp) {
        case MiddleOperator.ShiftLeft:
        case MiddleOperator.ShiftRight:
          return true;

        default:
          return false;
      }
    }
```

The **ToString** method simply returns a string holding the operator and the operand array. It is only used when adding comments to the final assembly code.

```
    public override string ToString() {
      return m_middleOperator + ToString(m_operandArray[0]) +
             ToString(m_operandArray[1]) + ToString(m_operandArray[2]);
    }

    private static string ToString(object value) {
      if (value != null) {
        return (" "  + value.ToString().Replace("\n", "\\n"));
      }
      else {
        return "";
      }
    }
  }
}
```

4.2. The MiddleCodeGenerator Class

The **MiddleCodeGenerator** class is a large class that generates the middle code from the source code.

MiddleCodeGenerator.cs

```
using System;
using System.IO;
using System.Linq;
using System.Numerics;
using System.Collections.Generic;
```

To begin with, we have the **AddMiddleCode** methods that add a **MiddleCode** object to the given code list.

```
namespace CCompiler {
  public class MiddleCodeGenerator {
    public static MiddleCode AddMiddleCode(List<MiddleCode> codeList,
                             MiddleOperator op, object operand0 = null,
                             object operand1 = null, object operand2 = null) {
      MiddleCode middleCode = new MiddleCode(op, operand0, operand1, operand2);
      codeList.Add(middleCode);
      return middleCode;
    }
```

4.2.1. The Forward-Jump Problem (Backpatching)

When generating middle code instructions for expressions or statements, a common situation is that we generate forward jump instructions without yet knowing the target address. One way to solve the problem is to work with **backpatching**, which means that we use sets to keep track of jump instructions with yet unknown target addresses that eventually becomes filled in when the target addresses have become known.

For instance, let us look at the while statement to the left. The parsing of expression **a < b** will generate the middle code in the middle and the sets to the right. The middle code line will a have undefined targets since we not yet know where to jump when the expression becomes evaluated to true or false.

```
while (a < b)              1. if a < b goto ?         a < b: true_set = {1}
  ++a;                     2. goto ?                         false_set = {2}
```

(a) C code (a) Middle code (a) True and false-sets

When parsing the while statement, the sets of the expression becomes backpatched: If the **a < b** expression is true, line 1 shall jump to line 3. If the expression is false, line 2 shall jump to line 5, which is the middle code instruction following the middle code for the while statement.

```
backpatch(true_set, 3); => backpatch({1}, 3);
backpatch(false_set, 5); => backpatch({2}, 5);

1. if a < b goto 3
2. goto 5
3. ++a
4. goto 1
5. ...
```

If we let the while statement above be surrounded by an **if-else** statement, we have both the true and false-sets of the **if** expression **c < d** and the sets of the while expression **a < b**. Note that line 6 and 7 do not need to be backpatched, since they jump backwards to known line numbers.

```
if (c < d)                 1. if c < d goto ?         c < d: true_set1: {1}
  while (a < b)            2. goto ?                         false_set1: {2}
    ++a;                   3. if a < b goto ?         a < b: true_set2: {3}
else                       4. goto ?                         false_set2: {4}
  ++b;                     5. ++a
                           6. goto 3
```

```
7. goto 9
8. ++b
9. ...
```

When the **if** and **while** statements become parsed the following call will occur:

```
backpatch(true_set_1, 3); => backpatch({1}, 3);
backpatch(false_set_1, 8); => backpatch({2}, 8);

backpatch(true_set_2, 5); => backpatch({3}, 5);
backpatch(false_set_2, 7); => backpatch({4}, 7);
```

```
1. if c < d goto 3
2. goto 8
3. if a < b goto 5
4. goto 7
5. ++a
6. goto 3
7. goto 9
8. ++b
9. ...
```

The generated middle code above is ineffective, but the middle code optimizer of Chapter 11 will change it into the following middle code.

```
1. if c >= d goto 5
2. if a >= b goto 6
3. ++a
4. goto 2
5. ++b
6. ...
```

Another example is the evaluation of a logical **and** expression. Since C takes advantage of lazy evaluation, we have to insert jump instructions in the middle code.

```
if ((a < b) && (c < d))    1. if a < b goto ?
  ++a;                     2. goto ?
else                       3. if c < d goto ?
  ++b;                     4. goto ?
                           5. ++a
                           6. goto 7
                           7. ++b
                           8. ...
```

(a) C code (b) Middle code before backpatching

```
1. if a < b goto ?         1. if a < b goto ?
2. goto ?                  2. goto ?
3. if c < d goto ?         3. if c < d goto ?
4. goto ?                  4. goto ?
5. ++a                     5. ++a
6. goto 7                  6. goto 7
7. ++b                     7. ++b
8. ...                     8. ...
```

(c) Middle code after (c) Middle code after backpatching   optimization

When the first expression (**a < b**) is evaluated, its true-set is {1} and its false-set is {2}. When the second expression is evaluated the true-set of the first expression is backpatched to the beginning of the second expression. This means that if the first expression is true we continue to evaluate the second expression. However, if the first expression is false the second expression will never be evaluated since it not necessary, and the total expression will be false regardless of the value of the second expression. The result will be that the true-set of the total expression is the one of the second expression and the false-set is the union of the false sets of both expressions.

```
1. if a < b goto 3          true-set: {3}
2. goto ?                   false-set: {2, 4}
3. if c < d goto ?
4. goto ?
5. ++a
6. goto 7
7. ++b
```

In a similar way, the true-set of an expression with logical-or will be the union of both true-sets while the false-set will the false-set of the second expression.

```
if ((a < b) || (c < d))   1. if a < b goto ?        true-set: {1, 3}
  ++a;                    2. goto 3                 false-set: {4}
else                      3. if c < d goto ?
  ++b;                    4. goto ?
                          5. ++a
                          6. goto 7
                          7. ++b
                          8. ...
```

The first **Backpatch** method takes set of jump instructions and a list of instructions, where its first instruction is the target of the jump instructions, and calls the second **Backpatch** method with the set of jump instruction and the first instruction in the list, which becomes the target instruction.

MiddleCodeGenerator.cs

```
public static void Backpatch(ISet<MiddleCode> sourceSet,
                             List<MiddleCode> list) {
  Backpatch(sourceSet, GetFirst(list));
}
```

The second **Backpatch** method iterates through the set of jump instructions and set its first reference (index 0) to the target. The first reference of each jump instruction is its target address.

```
public static void Backpatch(ISet<MiddleCode> sourceSet,
                             MiddleCode target) {
  foreach (MiddleCode source in sourceSet) {
    source[0] = target;
  }
}
```

The **GetFirst** method returns the first instruction in the middle code instruction list. It also adds an empty instruction if the list is empty. This is done to make sure that the backpatching calls always have an target instruction.

```
private static MiddleCode GetFirst(List<MiddleCode> list) {
  if (list.Count == 0) {
    AddMiddleCode(list, MiddleOperator.Empty);
  }
```

```
      return list[0];
    }
```

4.3. Declarations

4.3.1. Function Definition

This section covers the following methods:

- **FunctionHeader**, is called before the parsing of the function definition.
- **FunctionDefinition**, is called before the parsing of the function definition.
- **FunctionEnd**, is called before the parsing of the function definition.

The **FunctionHeader** method is called before the parsing of the function body. Its task is to store the symbol of the function in the symbol table, and to create a new symbol table for the symbols of the function.

MiddleCodeGenerator.cs

```
    public static void FunctionHeader(Specifier specifier,
                                      Declarator declarator) {
      Type returnType;
      bool externalLinkage;
```

If **specifier** is not null, we extract the external linkage, storage and return type of the function from the specifier.

```
      if (specifier != null) {
        externalLinkage = specifier.ExternalLinkage;
        returnType = specifier.Type;
      }
```

If **specifier** is null, there was no storage specifier or return type defined, in which case we state that the function has external linkage, the extern storage, and the signed integer return type.

```
      else {
        externalLinkage = true;
        returnType = Type.SignedIntegerType;
      }
```

We add the return type to the pointer declarator. In this way the type become a complete function with a return type.

```
      declarator.Add(returnType);
```

Then we do some error checking. Technically, the declaration may have any type, and we must check that it in fact is a function declaration.

```
      Assert.Error(declarator.Type.IsFunction(), declarator.Name,
                   Message.Not_a_function);
```

A function may lack a name in a function declaration, but in a function definition it must have a name.

```
      Assert.Error(declarator.Name != null,
                   Message.Unnamed_function_definition);
```

The public and static member variable **CurrentFunction** is a reference to the symbol holding the current function, which is used on every several occasions when returning values from the function.

A function declarator is given extern storage in the Declarator method. However, since this function declaration is in fact a function definition, with a function body, then its storage is set to static when **CurrentFunction** is set. The reason is that it is allowed to have many extern function declarations, but only one function definition, with the same name. The **AddSymbol** method of the **SymbolTable** class makes sure there are no two function definitions with the same name in the same scope.

```
      SymbolTable.CurrentFunction=new Symbol(declarator.Name, externalLinkage,
                                             Storage.Static, declarator.Type);
```

Every function definition, as well as static variables, are added to the global static set. They will eventually be translated into assembly code.

```
      SymbolTable.CurrentTable.AddSymbol(SymbolTable.CurrentFunction);
```

If the function is the **main** function of the C code, the return type must be void or signed or unsigned integer.

```
      if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
        Assert.Error(returnType.IsVoid() || returnType.IsInteger(), "main",
                     Message.Function_main_must_return_void_or_integer);
      }
```

Finally, we create a new symbol for the function. Every symbol defined in this function will be stored in the new symbol table, or in one of its sub tables.

```
      SymbolTable.CurrentTable =
        new SymbolTable(SymbolTable.CurrentTable, Scope.Function);
    }
```

The **FunctionDefinition** method checks that the definition is correct, especially in case of old-style definition. In the old-style function definition, the parameter list must match the declaration list.

```
    public static void FunctionDefinition() {
      Type funcType = SymbolTable.CurrentFunction.Type;

      if (funcType.Style == Type.FunctionStyle.Old) {
        List<string> nameList = funcType.NameList;
        IDictionary<string,Symbol> entryMap =
          SymbolTable.CurrentTable.EntryMap;
```

In the old-style case, we must make sure the number of parameters equals the number of declarations.

```
        Assert.Error(nameList.Count == entryMap.Count,
                     SymbolTable.CurrentFunction.Name, Message.
          Unmatched_number_of_parameters_in_old__style_function_definition);
```

Then we iterate through the parameter list to make sure they are all declared, and to assigned them the correct offset. When the parameters were declared they were given offsets. However, since the parameters and declaration may come in different order, we need to reassign offsets to match the parameters' order.

```
        int offset = SymbolTable.FunctionHeaderSize;
        foreach (string name in nameList) {
          Symbol symbol;
```

If one of the parameters has not been declared, we report an error.

```
          if (!entryMap.TryGetValue(name, out symbol)) {
            Assert.Error(name, Message.
                      Undefined_parameter_in_old__style_function_definition);
          }
```

```
        symbol.Offset = offset;
        offset += symbol.Type.SizeArray();
      }
    }
```

In case of a new-style function definition, we need to make sure the entry map of the current function is empty. If it is not empty, we have a mix of old-style and new-style function definition, which is not allowed.

```
    else {
      Assert.Error(SymbolTable.CurrentTable.EntryMap.Count == 0,
        Message.New_and_old_style_mixed_function_definition);
```

Then we iterate through the parameters list and add each parameter to the current symbol table of the function. In this way, each parameter is located at a memory offset so they can be assigned in function calls and be treated like regular variables inside the function. The **AddSymbol** method will also report an error if two parameters have the same name.

```
      foreach (Symbol symbol in funcType.ParameterList) {
        SymbolTable.CurrentTable.AddSymbol(symbol);
      }
    }
```

When the parameter list has been settled, we check if the function is the **main** function of the C code, in which case some special rules apply.

```
    if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
```

First, we call the **InitializationCodeList** method that adds a few lines of code that initializes the system and will be placed before the code of the main function.

```
      AssemblyCodeGenerator.InitializationCodeList();
```

We extract the parameter type list from the current function. In case of old-style definition, this list will be null. In case of new-style definition, there is only two allowed parameter list:

- No parameters, marked with void.

  ```
  int main(void) { /* ... */ }
  ```

- Command line arguments. The two cases below are actually the same case, since arrays are changed to pointers in parameter types. Note that we do not check the names of the parameters, the parameters may have other names.

  ```
  int main(int argc, char *argv[]) { /* ... */ }
  ```

  or

  ```
  int main(int argc, char **argv) { /* ... */ }
  ```

MiddleCodeGenerator.cs

```
      List<Type> typeList =
        SymbolTable.CurrentFunction.Type.TypeList;
```

If **typelist** is not null, and there are two parameters, we check if the command line argument case applies. The first parameter shall be a signed or unsigned integer while the second parameter shall be a pointer to pointer to a signed or unsigned character.

```
      if ((typeList != null) && (typeList.Count == 2)) {
        Assert.Error(typeList[0].IsInteger() &&
                     typeList[1].IsPointer() &&
                     typeList[1].PointerType.IsPointer() &&
```

```
                  typeList[1].PointerType.PointerType.IsChar(),
                  "main", Message.Invalid_parameter_list);
      AssemblyCodeGenerator.ArgumentCodeList();
    }
```

If **typelist** is null, we have the old-style function definition, in which case we do not perform any type checking at all. If **typelist** is empty, we have the void-marked empty parameter list.

```
    else {
      Assert.Error((typeList == null) || (typeList.Count == 0),
                  "main", Message.Invalid_parameter_list);
    }
  }
}
```

The **FunctionEnd** method is called when the statements of the function have been parsed. Its task is to check the return statement at the end of the function and generate a static symbol holding the assembly code of the function.

The first step is to backpatch the **next-set** of the statement. Each statement has a next-set that holds jump instructions. For instance, **for** or **while** loops have next-sets holding jump instructions jumping out of the loop. We add a new empty statement and backpatch the next-set to it.

```
public static void FunctionEnd(Statement statement) {
  MiddleCode nextCode =
    AddMiddleCode(statement.CodeList, MiddleOperator.Empty);
  Backpatch(statement.NextSet, nextCode);
```

If the return type of the function is void, we need to add a **Return** instruction at the end of the function.

```
  if (SymbolTable.CurrentFunction.Type.ReturnType.IsVoid()) {
    AddMiddleCode(statement.CodeList, MiddleOperator.Return);
```

If the function is the main function of C, we also add an **Exit** instruction.

```
    if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
      AddMiddleCode(statement.CodeList, MiddleOperator.Exit);
    }
  }
```

We then add a **FunctionEnd** middle code instruction. Its only purpose is that the middle code optimizer will report an error if it is possible to reach the **EndFunction** instruction in a function not returning void.

```
  AddMiddleCode(statement.CodeList, MiddleOperator.FunctionEnd,
                SymbolTable.CurrentFunction);
```

When we have added all middle code instructions, we optimize the middle code of the function. The optimizer transforms the middle code in several ways, see Chapter 11.

```
  MiddleCodeOptimizer middleCodeOptimizer =
    new MiddleCodeOptimizer(statement.CodeList);
  middleCodeOptimizer.Optimize();
```

When the middle code has been optimized, we generate assembly code of the function, see Chapter 12.

```
  List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();
  AssemblyCodeGenerator.GenerateAssembly(statement.CodeList,
                                         assemblyCodeList);
```

As mention in the introduction chapter of this book, we generate two kinds of target code. In the first part of the book we generate assembly code text for 64-bit Linux on Intel processors, which is than assembled and linked in traditional manner. In the second part of the book we generate an executable target file for 16-bit Windows. In the case where we need to distinguish between the two cases, we test whether the **Start**.**Linux** or **Start**.**Windows** static variable is true. In this chapter, we perform the actions for the Linux target system, see Chapter 13 for the Windows target system.

In the Linux case, we need to generate the assembly text from the assembly code list. We also need to generate the extern set; that is, the set of accesses of static variables with external linkage and the calls to all functions with external linkage.

```
      if (Start.Linux) {
        List<string> textList = new List<string>();

        if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
          textList.AddRange(SymbolTable.InitSymbol.TextList);

          if (SymbolTable.ArgsSymbol != null) {
            textList.AddRange(SymbolTable.ArgsSymbol.TextList);
          }
        }
        else {
          textList.Add("section .text");
        }

        ISet<string> externSet = new HashSet<string>();
        AssemblyCodeGenerator.LinuxTextList(assemblyCodeList, textList,
                                            externSet);
```

When the text list and access set has been generated, we create a static symbol that we add to the global static set. The symbols of the global static set are then handled by the **Main** method of Section 1.3

```
        StaticSymbol staticSymbol =
          new StaticSymbolLinux(SymbolTable.CurrentFunction.UniqueName,
                                textList, externSet);
        SymbolTable.StaticSet.Add(staticSymbol);
      }
```

As mentioned above, we ignore to code generation for the Windows envirnoment at this point, but will will look into it in Chapter 13.

```
      if (Start.Windows) {
        // ...
      }
```

Finally, we set the former symbol table, which is the parent table of this function's table, to be the current table. We also set the current function to null, which strictly speaking in not necessary. However, it is logical to set it to null when the parser does not parse the code of a function.

```
      SymbolTable.CurrentTable = SymbolTable.CurrentTable.ParentTable;
      SymbolTable.CurrentFunction = null;
    }
```

4.3.2. Structs and Unions

The **StructUnionSpecifier** method adds a struct or union. The **sort** parameter is either **Sort.Struct** or **Sort.Union**. If the optional name is not null, we look up the type (it has been added by **StructOrUnionHeader** above) and sets its member map, which is given by the entry map of the current symbol table.

```
    public static Type StructUnionSpecifier(string optionalName, Sort sort) {
      if (optionalName != null) {
        Type type = SymbolTable.CurrentTable.LookupTag(optionalName, sort);
        type.MemberMap = SymbolTable.CurrentTable.EntryMap;
        type.MemberList = SymbolTable.CurrentTable.EntryList;
        return type;
      }
```

If the optional name is null, we create and return a type with the entry map of the current symbol table.

```
      else {
        return (new Type(sort, SymbolTable.CurrentTable.EntryMap,
                         SymbolTable.CurrentTable.EntryList));
      }
    }
```

The **LookupStructUnion** method looks a struct or union. The **sort** parameter is either **Sort.Struct** or **Sort.Union**.

```
    public static Type LookupStructUnion(string name, Sort sort) {
      Type type = SymbolTable.CurrentTable.LookupTag(name, sort);
```

If the struct or union exists, we simply return its type.

```
      if (type != null) {
        return type;
      }
```

If the struct or union does not exist, we create a new type and add it to the tag map. However, the type lacks a member map, which means that it is not yet possible to define variables of the type. It is only when the struct or union becomes properly defined with a member map that it will be possible to define variables.

```
      else {
        type = new Type(sort);
        SymbolTable.CurrentTable.AddTag(name, type);
        return type;
      }
    }
```

4.3.3. Enumeration

The **EnumItem** method stores the item in the symbol table and checks the optional initialization symbol. The type of the item is constant signed integer.

MiddleCodeGenerator.cs

```
    public static Symbol EnumItem(string itemName,
                                  Symbol optInitializerSymbol) {
      Type itemType = new Type(Sort.SignedInt);
      itemType.Constant = true;
```

The value of the item is either set by the initialization symbol value, if present, or by the enumeration value stack. The stack holds the value of the current enumeration list. It is popped and pushed with the item value plus one. In that way will each item hold the value of the previous item plus one, the first item holds the value zero.

```
      BigInteger value;
      if (optInitializerSymbol != null) {
        Assert.Error(optInitializerSymbol.Type.IsIntegral(), itemName,
                     Message.Non__integral_enum_value);
        Assert.Error(optInitializerSymbol.IsValue(), itemName,
                     Message.Non__constant_enum_value);
        CCompiler_Main.Parser.EnumValueStack.Pop();
        value = (BigInteger) optInitializerSymbol.Value;
      }
      else {
        value = CCompiler_Main.Parser.EnumValueStack.Pop();
      }

      Symbol itemSymbol = new Symbol(itemName, null, itemType, value);
      SymbolTable.CurrentTable.AddSymbol(itemSymbol);
      CCompiler_Main.Parser.EnumValueStack.Push(value + 1);
      return itemSymbol;
    }
```

The **EnumSpecifier** method add the enumerator to the current symbol table if optional name is not null.

```
  public static Type EnumSpecifier(string optionalName,
                                   ISet<Pair<Symbol,bool>> enumSet) {
    Type enumType = new Type(enumSet);

    if (optionalName != null) {
      SymbolTable.CurrentTable.AddTag(optionalName, enumType);
    }

    return enumType;
  }
```

The **LookupEnum** method looks up the name of an enumeration. Technically, there is no meaning by looking up an enumeration name since an enumeration value has the signed integer type. However, if an unknown name is refereed, an error shall be reported. For instance, the following code is valid only if an enumeration with the name **CarMake** has earlier been defined.

```
Enum CarMake car;
```

MiddleCodeGenerator.cs

```
  public static Type LookupEnum(string name) {
    Type type = SymbolTable.CurrentTable.LookupTag(name, Sort.Enumeration);
    Assert.Error(type != null, name, Message.Tag_not_found);
    return type;
  }
```

4.3.4. Declarator

The **Declarator** method handles the cases without assignment or bitfield. The declarator is possibly made up by a sequence of pointer, array, or functions declarations, and we add the type of the specifier as the end type of the declarator.

MiddleCodeGenerator.cs

```
  public static void Declarator(Specifier specifier, Declarator declarator){
    declarator.Add(specifier.Type);
```

A function can only hold extern or static storage. This is in fact the only storage check that is performed outside the **Specifier** class of Chapter 4, since we need the declarator to decide that it is in fact a function declaration.

```
    Storage storage = specifier.Storage;
    if (declarator.Type.IsFunction()) {
      Assert.Error((storage == Storage.Static) ||
                   (storage == Storage.Extern),  storage, Message.
      Only_extern_or_static_storage_allowed_for_functions);
```

When declared, a function may be extern or static. After the external linkage has been set in the Specifier class, we do not need the storage of a function declarator. Therefore, we always set extern storage for a function declarator. If this function declaration should in fact be a function definition, with a function body, then its storage will be set to static by the **FunctionHeader**.

```
            storage = Storage.Extern;
        }
```

We create the symbol holding the name, external linkage, storage, and type of the declarator, and add it to the current symbol table.

```
        Symbol symbol = new Symbol(declarator.Name, specifier.ExternalLinkage,
                                   storage, declarator.Type);
        SymbolTable.CurrentTable.AddSymbol(symbol);
```

If the symbol has static storage and is not a function, we add it to the global static set. We call the Value method in the **ConstantExpression** class to generate the global static set. If it is a function definition, it becomes added by the **FunctionEnd** method of Section 3.2.1. If it is function declaration, is shall not be stored in the global static set at all. Declarations are stored in the symbol table only, not in the global static set.

```
        if ((storage == Storage.Static) && !type.IsFunction()) {
          SymbolTable.StaticSet.Add(ConstantExpression.Value(symbol));
        }
      }
```

The **InitializedDeclarator** method handles the cases where the declarator becomes initialized with a value. Similar to the **Declarator** case above, we begin by adding the type of the specifier to the declarator.

```
      public static List<MiddleCode> InitializedDeclarator(Specifier specifier,
                                 Declarator declarator, object initializer){
        declarator.Add(specifier.Type);
        Type type = declarator.Type;
        Storage storage = specifier.Storage;
        string name = declarator.Name;
```

A function cannot be initialized, neither can a symbol of extern or typedef storage, or a member of a struct or union.

```
        Assert.Error(!type.IsFunction(), null,
                     Message.Functions_cannot_be_initialized);
        Assert.Error(storage != Storage.Extern, name,
                     Message.Extern_cannot_be_initialized);
        Assert.Error(storage != Storage.Typedef, name,
                     Message.Typedef_cannot_be_initialized);
        Assert.Error((SymbolTable.CurrentTable.Scope != Scope.Struct) &&
                     (SymbolTable.CurrentTable.Scope != Scope.Union),
                     name, Message.Struct_or_union_field_cannot_be_initialized);
```

In case of static storage, we call the **GenerateStatic** method in the **GenerateStaticInitializer** class to generate a middle code list holding the initialization code.

```
        if (storage == Storage.Static) {
          List<MiddleCode> middleCodeList =
            GenerateStaticInitializer.GenerateStatic(type, initializer);
```

We create a symbol with the name and type of the declarator and the external linkage and storage of the specifier and add it to the current symbol table.

```
        Symbol symbol = new Symbol(name, specifier.ExternalLinkage,
                                   storage, type);
        SymbolTable.CurrentTable.AddSymbol(symbol);
```

Since the storage is static, we also create a static symbol that we add to the global static set. Similar to the **Declarator** case above, we call the **Value** method in the **ConstantExpression** class to generate the value. However, we return an empty code list after we have added the static symbol to the global static set.

```
        StaticSymbol staticSymbol =
          ConstantExpression.Value(symbol.UniqueName, type, middleCodeList);
        SymbolTable.StaticSet.Add(staticSymbol);
        return (new List<MiddleCode>());
      }
```

In the non-static cases, we call the **GenerateAuto** method in the **GenerateAutoInitializer** class, which return a list of middle code instructions. The value in this case is not a static value. Instead, it is a list of middle code instructions that assign the value to the symbol in run-time.

```
      else {
        Symbol symbol =
          new Symbol(name, specifier.ExternalLinkage, storage, type);
        symbol.Offset = SymbolTable.CurrentTable.CurrentOffset;
        List<MiddleCode> codeList =
          GenerateAutoInitializer.GenerateAuto(symbol, initializer);
        SymbolTable.CurrentTable.AddSymbol(symbol);
        return codeList;
      }
    }
```

The **BitfieldDeclarator** handles the case with bitfields. What is special in this case is that the declarator can actually be omitted.

```
    public static void BitfieldDeclarator(Specifier specifier,
                             Declarator declarator, Symbol bitsSymbol) {
      Storage storage = specifier.Storage;
```

Bitfields are only allowed in structs or unions and can only have auto or register storage.

```
      Assert.Error((SymbolTable.CurrentTable.Scope == Scope.Struct) ||
                   (SymbolTable.CurrentTable.Scope == Scope.Union),bitsSymbol,
                   Message.Bitfields_only_allowed_in_structs_or_unions);
      Assert.Error((storage == Storage.Auto) || (storage == Storage.Register),
                   null, Message.
                   Only_auto_or_register_storage_allowed_in_struct_or_union);
```

We calculate the number of bits by parsing the text of the

```
      object bitsValue = bitsSymbol.Value;
      int bits = int.Parse(bitsValue.ToString());
```

If the declarator is not null (it has been present in the code) we add the specifier type to it and extract its final type, which must be integral. The number of bits must be a value between one and the type size in bites (eight times the type size in bytes), inclusive.

```
      if (declarator != null) {
        declarator.Add(specifier.Type);
        Type type = declarator.Type;
        Assert.Error(type.IsIntegral(), type,
                     Message.Non__integral_bits_expression);
        Assert.Error((bits >= 1) && (bits <= (8 * type.Size())),
                      bitsValue, Message.Bits_value_out_of_range);
```

If the number of bits is less then the type size in bits, we need to define the bit mask that is used to delete the bits not covered by the mask.

```
        if (bits < (8 * type.Size())) {
          type.SetBitfieldMask(bits);
        }

        Symbol symbol = new Symbol(declarator.Name, specifier.ExternalLinkage,
                                   storage, type);
        SymbolTable.CurrentTable.AddSymbol(symbol);
```

Contrary to the previous cases, we do not add the symbol to the global static set since it cannot be static.

```
      }
```

If the declarator is null (it has been omitted in the code) we just check that the number of bits is in range: at least one and at most the type size in bits.

```
      else {
        Assert.Error((bits >= 1) && (bits <= (8 * 4)), bitsValue,
                      Message.Bits_value_out_of_range);
      }
    }
```

4.3.5. Pointer declarator

The **PointerDeclarator** method takes a type list and a declarator.

MiddleCodeGenerator.cs

```
    public static Declarator PointerDeclarator(List<Type> typeList,
                                               Declarator declarator) {
```

If the declarator is null, we simply create a new declarator since we always need a declarator, even in the cases where is has been omitted.

```
      if (declarator == null) {
        declarator = new Declarator(null);
      }
```

We iterate through the pointer type list and add each pointer type to the declarator. In this way, the declarator is pointer.

```
      foreach (Type pointerType in typeList) {
        Type pointerType = new Type((Type) null);
        pointerType.Constant = type.Constant;
        pointerType.Volatile = type.Volatile;
        declarator.Add(pointerType);
      }
```

We will always have at least a declarator with a certain type or a pointer type, since it is syntactically impossible to omit both the declarator and the pointer list.

```
      return declarator;
    }
```

4.3.6. Direct Declarator

The **ArrayType** method handles the declaration of an array.

MiddleCodeGenerator.cs
```
    public static Declarator ArrayType(Declarator declarator,
                                       Expression optionalSizeExpression) {
      if (declarator == null) {
        declarator = new Declarator(null);
      }
```

If the array size expression is present, the parser has already made sure it is a constant expression of integral type. But we must also check that the array (the value of the expression) is positive.

```
      Type arrayType;
      if (optionalSizeExpression != null) {
        Symbol optSizeSymbol = optionalSizeExpression.Symbol;
        int arraySize = (int) ((BigInteger) optSizeSymbol.Value);
        Assert.Error(arraySize > 0, arraySize,
                     Message.Non__positive_array_size);
        arrayType = new Type(arraySize, null);
      }
```

If the array size expression is not present, we create a new array type of zero size.

```
      else {
        arrayType = new Type(0, null);
      }
```

We add the array type to the declarator, which we return. Note that the type of the array is yet unknown, which is why it is null.

```
      declarator.Add(arrayType);
      return declarator;
    }
```

The next kind of direct declarator is the new-style function declarator, with parentheses enclosing a variable parameter list. A variable parameter list is a parameter list that may be finished with the variable marker ('…').

The **NewFunctionDeclaration** handles a new-style function declaration.

MiddleCodeGenerator.cs
```
    public static Declarator NewFunctionDeclaration(Declarator declarator,
                                                    List<Symbol> parameterList,
                                                    bool variadic) {
```

It is not allowed to add a variable marker to an empty parameter list, it must always be at least one parameter.

```
      if (parameterList.Count == 0) {
        Assert.Error(!variadic, "...",
            Message.An_variadic_function_must_have_at_least_one_parameter);
      }
```

It is allowed to have a **void** parameter if it is the only parameter in the list, and there is no variable marker.

```
      else if ((parameterList.Count == 1) && parameterList[0].Type.IsVoid()) {
        Assert.Error(parameterList[0].Name == null,
                     parameterList[0].Name,
                     Message.A_void_parameter_cannot_be_named);
        Assert.Error(!variadic, "...", Message.
                     An_variadic_function_cannot_have_a_void_parameter);
        parameterList.Clear();
      }
```

Otherwise, a **void** parameter is not allowed.

```
      else {
        foreach (Symbol symbol in parameterList) {
          Assert.Error(!symbol.Type.IsVoid(),
                       Message.Invalid_void_parameter);
        }
      }
```

Finally, we create a new-style function type with the parameter list and variadic status, add it to the declarator, and return the declarator. Note that we do not add a return type (the first parameter is null), since it will be added to the declarator by the specifier.

```
      declarator.Add(new Type(null, parameterList, variadic));
      return declarator;
    }
```

The **OldFunctionDeclaration** method handles the old-style function declaration. The only thing we need to test is that the same name does not occur twice in the name list.

MiddleCodeGenerator.cs

```
    public static Declarator OldFunctionDeclaration(Declarator declarator,
                                                    List<string> nameList) {
```

We create a set and add the names of the name list to the set. The **Add** method returns true as long as the name does not already occur in the set.

```
      ISet<string> nameSet = new HashSet<string>();
      foreach (string name in nameList) {
        Assert.Error(nameSet.Add(name), name, Message.Name_already_defined);
      }
```

Finally, we create an old-style function type with the name list, add it to the declarator, and return the declarator. Again, note that we do not add a return type since it will be added to the declarator by the specifier.

```
      declarator.Add(new Type(null, nameList));
      return declarator;
    }
```

4.3.7. Parameters

MiddleCodeGenerator.cs

```
    public static Symbol Parameter(Specifier specifier,
                                   Declarator declarator) {
      Storage storage = specifier.Storage;
      Type specifierType = specifier.Type;

      string name;
      Type type;

      if (declarator != null) {
        name = declarator.Name;
        declarator.Add(specifierType);
        type = declarator.Type;
      }
      else {
        name = null;
        type = specifierType;
      }
```

If the parameter is an array, we set the type to a pointer to the array type, and if it is a function, we set the type to a pointer to the function. In both cases the type becomes constant.

```
      if (type.IsArray()) {
        type = new Type(type.ArrayType);
        type.Constant = true;
      }
      else if (type.IsFunction()) {
        type = new Type(type);
        type.Constant = true;
      }
```

We create and return a symbol, with the name, storage, and type of the parameter. The second value is false since parameters do not have external linkage. The last value is true since it is a parameter, which we need to know that in the assembly code generation phase in Chapter 12.

```
      return (new Symbol(name, false, storage, type, true));
    }
```

4.4. Statements

4.4.1. The If Statement

The **IfStatement** method handles the if statement of the parser. Since the if-statement accepts logical expressions only, we need to start with making sure the expression is a logical expression.

MiddleCodeGenerator.cs

```
    public static Statement IfStatement(Expression expression,
                                        Statement innerStatement){
      expression = TypeCast.ToLogical(expression);
```

Since we use the value of the expression in the if-statement, we only use the long list.

```
      List<MiddleCode> codeList = expression.LongList;
      AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);
```

We backpatch the true value of the expression to the first instruction in the middle code list of the inner statement. This means that when the expression is true, the program will jump to the beginning of the code list of the inner statement.

```
      Backpatch(expression.Symbol.TrueSet, innerStatement.CodeList);
      codeList.AddRange(innerStatement.CodeList);
```

We add a jump instruction at the end of the middle code that shall jump to the instruction following this if-statement.

```
      MiddleCode nextCode = AddMiddleCode(codeList, MiddleOperator.Jump);
```

We define the next-set of the if-statement as the union of the next-set of the inner statement, the false-set of the expression, and the jump instruction added to the code list of the inner statement.

```
      ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
      nextSet.UnionWith(innerStatement.NextSet);
      nextSet.UnionWith(expression.Symbol.FalseSet);
      nextSet.Add(nextCode);
```

Finally, we return on object of the **Statement** class holding the code list and next-set of the if-statement.

```
      return (new Statement(codeList, nextSet));
    }
```

4.4.2. The If-Else Statement

The **IfElseStatement** method is similar to **IfStatement** above. The difference is that we have two inner statements, that shall be executed in case of a true or false expression.

```
    public static Statement IfElseStatement(Expression expression,
                                            Statement trueStatement,
                                            Statement falseStatement) {
      expression = TypeCast.ToLogical(expression);
      List<MiddleCode> codeList = expression.LongList;
      AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);
```

We backpatch the true-set and false-set to the beginning of the code list of the true and false statement, respectively.

```
      Backpatch(expression.Symbol.TrueSet, trueStatement.CodeList);
      Backpatch(expression.Symbol.FalseSet, falseStatement.CodeList);
```

We add the code list for the true and false statements. We also add a jump instruction after each list, to jump out of the statement. The jump instruction are then added to the next-set of the else-if-statement.

```
      codeList.AddRange(trueStatement.CodeList);
      MiddleCode trueNextCode = AddMiddleCode(codeList, MiddleOperator.Jump);
      codeList.AddRange(falseStatement.CodeList);
      MiddleCode falseGotoCode = AddMiddleCode(codeList, MiddleOperator.Jump);
```

The next-set of the if-else-statement is the union of the next-sets of the true and false statement, and their extra jump instructions.

```
      ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
      nextSet.UnionWith(trueStatement.NextSet);
      nextSet.UnionWith(falseStatement.NextSet);
      nextSet.Add(trueNextCode);
      nextSet.Add(falseGotoCode);
```

Finally, we return on object of the **Statement** class holding the code list and next-set of the if-statement.

```
      return (new Statement(codeList, nextSet));
    }
```

4.4.3. The Switch Statement

We need a map to keep track of the case statements. Each entry of the map holds the value of the case statement, and the first instruction of its middle code list (if the list is empty, we add an empty instruction). Since the switch statements can be nested, we store the maps for each switch statement in the **m_caseMapStack** map.

```
    private static Stack<IDictionary<BigInteger, MiddleCode>> m_caseMapStack =
      new Stack<IDictionary<BigInteger, MiddleCode>>();
```

In the same way, we need a stack to keep track of the default statement of the switch statement. For each default statement connected to a switch statement, we store the first instruction of the default middle code list in the **m_defaultStack**. If there is no default statement, we add null to the stack.

```
private static Stack<MiddleCode> m_defaultStack = new Stack<MiddleCode>();
```

Finally, we also need the **m_breakSetStack** stack for the break statements connected to the switch statement. We add the first instruction of the break statement to the sets of the stack. They are later backpatched to jump out of the switch statement.

```
private static Stack<ISet<MiddleCode>> m_breakSetStack =
  new Stack<ISet<MiddleCode>>();
```

The **SwitchHeader** method is called before the parsing of the switch statement. It pushes an empty map to the case map stack, a null reference to the default stack, and an empty set to the break set stack.

```
public static void SwitchHeader() {
  m_caseMapStack.Push(new Dictionary<BigInteger,MiddleCode>());
  DefaultStack.Push(null);
  BreakSetStack.Push(new HashSet<MiddleCode>());
}
```

The **SwitchStatement** method is called after the parsing of the switch statement. Its task is to generate middle code for the case and default statements, and to backpatch the break statements.

```
public static Statement SwitchStatement(Expression switchExpression,
                                        Statement innerStatement) {
```

Since the switch statement requires an integral value rather than a logical value we need to, contrary to the if-statement above, type cast a potential logical value to an integral value.

```
  switchExpression = TypeCast.LogicalToIntegral(switchExpression);
```

Since we need the value of the switch statement, rather than its side-effects, we are only interested in the long list of the switch expression.

```
  List<MiddleCode> codeList = switchExpression.LongList;
```

Each case value is type casted to the type of the switch expression.

```
  Type switchType = switchExpression.Symbol.Type;
  foreach (KeyValuePair<BigInteger,MiddleCode> entry
           in m_caseMapStack.Pop()) {
    BigInteger caseValue = entry.Key;
    MiddleCode caseTarget = entry.Value;
    Symbol caseSymbol = new Symbol(switchType, caseValue);
```

We add code where we compare the case value with the switch value and jump to the matching middle code instruction if the values are equal. We use the **Case** instruction rather than **Equal**, since we want the value to be kept in a register when we generate the assembly code in Chapter 12.

```
    AddMiddleCode(codeList, MiddleOperator.Case, caseTarget,
                  switchExpression.Symbol, caseSymbol);
  }
```

After the case values, we add a case end instruction. The reason for this is that the register used for the case comparison in the generated assembly code shall be unallocated.

```
  AddMiddleCode(codeList, MiddleOperator.CaseEnd,
                switchExpression.Symbol);
```

If there is a default statement, we jump to it if the switch statement does not match any of the case statement. In that case, the next-set becomes empty.

```
      MiddleCode defaultCode = DefaultStack.Pop();

      if (defaultCode != null) {
        AddMiddleCode(codeList, MiddleOperator.Jump, defaultCode);
      }
```

Similar to the if-statement and if-else-statement, the switch-statement has a next-set; that is, a set of middle code instruction that shall jump out of the switch statement.

```
      ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
```

If there is no default statement, the execution shall jump out of the switch statement after the last case statement, and we add a jump instruction to the next-set.

```
      if (defaultCode == null) {
        nextSet.Add(AddMiddleCode(codeList, MiddleOperator.Jump));
      }
```

We add the code list of the inner statement to the resulting code list.

```
      codeList.AddRange(innerStatement.CodeList);
```

Finally, we add the next-set of the inner statement and the set of break statements to the next-set of the switch statement.

```
      nextSet.UnionWith(innerStatement.NextSet);
      nextSet.UnionWith(BreakSetStack.Pop());
      return (new Statement(codeList, nextSet));
    }
```

4.4.4. The Case Statement

For the case statement the **Main.CaseMapStack** is not allowed to be empty. If it is empty, the case statement misses an enclosed switch statement, and we report an error.

```
    public static Statement CaseStatement(Expression expression,
                                          Statement statement) {
      Assert.Error(m_caseMapStack.Count > 0, Message.Case_without_switch);
      expression = TypeCast.LogicalToIntegral(expression);
```

Moreover, the value of the case expression must be integral and constant, and thereby possible to evaluate in compile-time. If the value of the symbol is null, we have a non-constant case value, and we report an error.

```
      Assert.Error(expression.Symbol.Value != null, expression.Symbol.Name,
                   Message.Non__constant_case_value);
      BigInteger caseValue = (BigInteger) expression.Symbol.Value;
```

We extract the current case map from the top of the case map stack and adds the current case value to the map.

```
      IDictionary<BigInteger, MiddleCode> caseMap = m_caseMapStack.Peek();
```

If the map already contains the value, we have two case statements with the same value and we report an error.

```
      Assert.Error(!caseMap.ContainsKey(caseValue), caseValue,
                   Message.Repeated_case_value);
```

Finally, we add the case value and the first instruction of the statement's code to the case map of the enclosed switch statements

```
  caseMap.Add(caseValue, GetFirst(statement.CodeList));
  return statement;
}
```

4.4.5. The Default Statement

If the default stack is empty, we have a default statement without an enclosing switch statement and we report an error.

```
public static Statement DefaultStatement(Statement statement) {
  Assert.Error(DefaultStack.Count > 0, Message.Default_without_switch);
```

There can be only one default statement in a switch statement. Therefore, if the value on the top of the default stack is not null, we have a switch statement with two default statement and we report an error.

```
  Assert.Error(DefaultStack.Pop() == null, Message.Repeted_default);
```

Finally, we add the default middle code instruction at the top of the stack.

```
  DefaultStack.Push(GetFirst(statement.CodeList));
  return statement;
}
```

4.4.6. The While Statement

Similar to the break set stack above, we also need a continue set stack.

```
private static Stack<ISet<MiddleCode>> m_continueSetStack =
  new Stack<ISet<MiddleCode>>();
```

The **LoopHeader** method is called before the parsing of a while, do, or for statement. Its task is to add empty sets at the top of the break set stack and continue set stack.

```
public static void LoopHeader() {
  m_breakSetStack.Push(new HashSet<MiddleCode>());
  m_continueSetStack.Push(new HashSet<MiddleCode>());
}
```

The **WhileStatement** method is called after the while statement has been parsed. Similar to the if statement above, it starts by type casting the expression into logical type.

```
public static Statement WhileStatement(Expression expression,
                                        Statement innerStatement) {
  expression = TypeCast.ToLogical(expression);
  List<MiddleCode> codeList = expression.LongList;
  AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);
```

We backpatch the true-set of the expression to the code list of the inner statement. If the expression is true, the execution shall jump to the beginning of the inner statement.

```
  Backpatch(expression.Symbol.TrueSet, innerStatement.CodeList);
  codeList.AddRange(innerStatement.CodeList);
```

We define the next-set of the while statement, and add a jump instruction that jumps back to the beginning of the while statement. Since we jump backwards, we do not need to perform backpatching.

```
  ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
  nextSet.Add(AddMiddleCode(codeList, MiddleOperator.Jump,
                                           GetFirst(codeList)));
```

We also add the false-set of the expression and the break set of the inner statement to the next-set.

```
      nextSet.UnionWith(expression.Symbol.FalseSet);
      nextSet.UnionWith(m_breakSetStack.Pop());
```

We backpatch the next-set of the inner statement and its continue set to the code list. The jump instruction of the sets shall jump back to the beginning of the while statement.

```
      Backpatch(innerStatement.NextSet, codeList);
      Backpatch(m_continueSetStack.Pop(), codeList);
      return (new Statement(codeList, nextSet));
    }
```

4.4.7. The Do Statement

The do statement starts with a statement followed by an expression. We start by backpatching the next set of the inner statement to the beginning of the middle code of the do statement.

```
    public static Statement DoStatement(Statement innerStatement,
                                        Expression expression) {
      List<MiddleCode> codeList = innerStatement.CodeList;
      Backpatch(innerStatement.NextSet, codeList);
      codeList.AddRange(expression.LongList);
```

We backpatch the true-set of the expression and the continue set of the inner statement to the beginning of the do statement.

```
      Backpatch(expression.Symbol.TrueSet, codeList);
      Backpatch(m_continueSetStack.Pop(), codeList);
```

The next-set of the do statement is the union of the false-set of the expression, the break-set of the inner statement, and the jump instruction out of the inner statement.

```
      ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
      nextSet.UnionWith(expression.Symbol.FalseSet);
      nextSet.UnionWith(m_breakSetStack.Pop());
      return (new Statement(codeList, nextSet));
    }
```

4.4.8. The For Statement

In the **ForStatement** method, we make difference of the short and long list of the expressions. If the test expression is present, we want its value, and therefore we are interested in its long list. On the other hand, if the initializer or next expressions are present, we are only interested in their potential side effects, which are stored in their short lists.

```
    public static Statement ForStatement(Expression initializerExpression,
                         Expression testExpression, Expression nextExpression,
                         Statement innerStatement) {
      List<MiddleCode> codeList = new List<MiddleCode>();
      ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();
```

We add an empty instruction as the target of the test expression, since we do not know if their in fact is a test expression.

```
      MiddleCode testTarget = AddMiddleCode(codeList, MiddleOperator.Empty);
```

If the initializer expression is not null, we add its short list. Note that we add the short list instead of the long list since we are only interested of the side effects of the expression, not its value. We also backpatch both its true-set and false-set to the test target; that is, the beginning of the test expression code.

```
      if (initializerExpression != null) {
        codeList.AddRange(initializerExpression.ShortList);
        Backpatch(initializerExpression.Symbol.TrueSet, testTarget);
        Backpatch(initializerExpression.Symbol.FalseSet, testTarget);
      }
```

If the test expression is not null, we start by type casting it into a logical value and add its long list to the code. Note that we add the long list rather than the short list in this case, since we need the value of the expression, rather than its side effects only.

```
      if (testExpression != null) {
        testExpression = TypeCast.ToLogical(testExpression);
        codeList.AddRange(testExpression.LongList);
        AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);
```

We backpatch the true-set of the test expression to the beginning or the code list of the inner statement.

```
        Backpatch(testExpression.Symbol.TrueSet, innerStatement.CodeList);
        nextSet.UnionWith(testExpression.Symbol.FalseSet);
      }
```

We add the code list of the inner statement. We then add the next target instruction and backpatch the next-set of the code of the inner statement since we do not know if the next expression is not null.

```
      codeList.AddRange(innerStatement.CodeList);
      MiddleCode nextTarget = AddMiddleCode(codeList, MiddleOperator.Empty);
      Backpatch(innerStatement.NextSet, nextTarget);
```

If the next expression is not null, we add its short list and backpatch both its true-set and false-set to the beginning of the test expression.

```
      if (nextExpression != null) {
        codeList.AddRange(nextExpression.ShortList);
        Backpatch(nextExpression.Symbol.TrueSet, testTarget);
        Backpatch(nextExpression.Symbol.FalseSet, testTarget);
      }
```

Finally, like the while case, the continue set is backpatched to the beginning of the test expression and the break set is added to the next-set of the for statement.

```
      AddMiddleCode(codeList, MiddleOperator.Jump, testTarget);
      Backpatch(m_continueSetStack.Pop(), nextTarget);
      nextSet.UnionWith(m_breakSetStack.Pop());

      return (new Statement(codeList, nextSet));
    }
```

4.4.9. The Label and Goto Statements

In C, is is allowed to use goto statements, jumping to labels, even though it is not recommended. For that we need the **m_labelMap** and **m_gotoSetMap** maps, where **m_labelMap** stores the labels of the goto targets and **m_gotoSetMap** stores sets of instructions jumping to a target.

```
    public static IDictionary<string, MiddleCode> m_labelMap =
      new Dictionary<string, MiddleCode>();
    public static IDictionary<string, ISet<MiddleCode>> m_gotoSetMap =
      new Dictionary<string, ISet<MiddleCode>>();
```

The **LabelStatement** add a label to the label map. If the label map already holds a label with the name, we have two labels with the same name, which is not allowed and we report an error.

```
    public static Statement LabelStatement(string labelName,
                                           Statement statement) {
      Assert.Error(!m_labelMap.ContainsKey(labelName),
                   labelName, Message.Defined_twice);
      m_labelMap.Add(labelName, GetFirst(statement.CodeList));
      return statement;
    }
```

In the **GotoStatement** method we start by adding middle code holding a goto instruction.

```
    public static Statement GotoStatement(string labelName) {
      List<MiddleCode> gotoList = new List<MiddleCode>();
      MiddleCode gotoCode = AddMiddleCode(gotoList, MiddleOperator.Jump);
```

If the label name already exists the goto-set map, we look up the goto-set and add the goto instruction. This situation occurs in when there has been previous jumps to the label.

```
      if (m_gotoSetMap.ContainsKey(labelName)) {
        ISet<MiddleCode> gotoSet = m_gotoSetMap[labelName];
        gotoSet.Add(gotoCode);
      }
```

If the label name is not already a key in the goto-set map, we create a new goto-set, to which we add the goto instruction, and then we add the label name and goto set to the goto set map. This situation occurs in when we encounter the first jump to the label. Note that we do not check whether the label is present in the label map, since it may be a forward jump. The **BackpatchGoto** method below backpatches all jumps.

```
      else {
        ISet<MiddleCode> gotoSet = new HashSet<MiddleCode>();
        gotoSet.Add(gotoCode);
        m_gotoSetMap.Add(labelName, gotoSet);
      }

      return (new Statement(gotoList));
    }
```

The **BackpatchGoto** method is called at the end of each function definition. It iterates through the goto-set map and, for each label name, looks up the label instruction and backpatch the goto-set to that instruction.

```
    public static void BackpatchGoto() {
      foreach (KeyValuePair<string,ISet<MiddleCode>> entry in m_gotoSetMap) {
        string labelName = entry.Key;
        ISet<MiddleCode> gotoSet = entry.Value;
```

If the label name does not exist in the label map, we have a goto statement to an unknown label and we report an error.

```
        MiddleCode labelCode;
        Assert.Error(m_labelMap.TryGetValue(labelName, out labelCode),
                     labelName, Message.Missing_goto_address);
        Backpatch(gotoSet, labelCode);
```

```
    }
  }
```

4.4.10. Return Statement

The **GenerateReturnStatement** method generates the code for setting the return value (if present) and return the execution to the calling function. If the function is main function it also adds code for exiting the execution. In that case, the execution shall exit only in case of the original main function. Note thet is allowed to recursively call the main function.

```
    public static Statement ReturnStatement(Expression expression) {
      List<MiddleCode> codeList;
```

If the expression is not null, we need the check that the function does not return void. If it does, we report an error.

```
      if (expression != null) {
        Assert.Error(!SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),
                     Message.Non__void_return_from_void_function);
```

We cast the return expression to the return type of the function.

```
        expression = TypeCast.ImplicitCast(expression,
                               SymbolTable.CurrentFunction.Type.ReturnType);
        codeList = expression.LongList;
        AddMiddleCode(codeList, MiddleOperator.SetReturnValue);
        AddMiddleCode(codeList, MiddleOperator.Return,
                      null, expression.Symbol);
      }
```

If the expression is null, we check that the function returns void. If it does not, we report an error.

```
      else {
        Assert.Error(SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),
                     Message.Void_returned_from_non__void_function);
        codeList = new List<MiddleCode>();
        AddMiddleCode(codeList, MiddleOperator.Return);
      }
```

If the function is the main function, we exit the program execution.

```
      if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
        AddMiddleCode(codeList, MiddleOperator.Exit);
      }

      return (new Statement(codeList));
    }
```

4.4.11. Optional Expression Statement

If there is an expression, we add its short list to the code list. We choose the short list rather than the long list since we are only interested in the side effects of the expression.

MiddleCodeGenerator.cs
```
    public static Statement ExpressionStatement(Expression expression) {
      List<MiddleCode> codeList = new List<MiddleCode>();

      if (expression != null) {
        codeList.AddRange(expression.ShortList);
      }

      return (new Statement(codeList));
    }
```

4.4.12. Jump Register Statements

The **JumpRegisterStatement** method adds an instruction for jumping to the address stores in a register. It is used by the **setjmp** function in the standard library.

```
  public static Statement JumpRegisterStatement(Register register) {
    List<MiddleCode> codeList = new List<MiddleCode>();
    AddMiddleCode(codeList, MiddleOperator.JumpRegister, register);
    return (new Statement(codeList));
  }
```

4.4.1. Interrupt Statements

The **InterruptStatement** method adds an instruction for performing an interrupt call. It is used by the standard library for the Windows environment.

```
  public static Statement InterruptStatement(Expression expression) {
    List<MiddleCode> codeList = new List<MiddleCode>();
    AddMiddleCode(codeList, MiddleOperator.Interrupt,
                  expression.Symbol.Value);
    return (new Statement(codeList));
  }
```

4.4.1. System Call Statements

The **SystemCallStatement** method adds an instruction for performing a system call. It is used by the standard library for the Linux environment.

```
  public static Statement SystemCallStatement() {
    List<MiddleCode> codeList = new List<MiddleCode>();
    AddMiddleCode(codeList, MiddleOperator.SysCall);
    return (new Statement(codeList));
  }
```

4.5. Expressions

The next section handles the middle code generation of the expressions. Similar to the parsing, we start with the comma operator, which holds lowest precedence.

4.5.1. The Comma Expression

The value of a comma expression is the value of the right expression. The value of the left expression is discarded, we only keep the side effects of the left expression.

```
  public static Expression CommaExpression(Expression leftExpression,
                                           Expression rightExpression) {
    List<MiddleCode> shortList = new List<MiddleCode>();
    shortList.AddRange(leftExpression.ShortList);
    shortList.AddRange(rightExpression.ShortList);
```

Note that we add the short list of the left expression to the middle code long list, because we are noted in the value of the left expression, only its side effects.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.ShortList);
      longList.AddRange(rightExpression.LongList);
```

We return the symbol of the right expression and discard the symbol of the left expression.

```
      return (new Expression(rightExpression.Symbol, shortList, longList));
    }
```

4.5.2. The Assignment Expression

In C, there are both simple and compound assignment. The **AssignmentExpression** method calls **Assignment** with different arguments depending on the operator.

```
    public static Expression AssignmentExpression(MiddleOperator middleOp,
                                                  Expression leftExpression,
                                                  Expression rightExpression) {
      switch (middleOp) {
        case MiddleOperator.Assign:
          return Assignment(leftExpression, rightExpression, true);
```

In cases of compound assignment, we call the suitable method to perform the operation, and the **Assignment** method for the assignment.

```
        case MiddleOperator.Add:
          return Assignment(leftExpression, AdditionExpression
                            (leftExpression, rightExpression));

        case MiddleOperator.Subtract:
          return Assignment(leftExpression,
            SubtractionExpression(leftExpression, rightExpression));

        case MiddleOperator.Multiply:
        case MiddleOperator.Divide:
        case MiddleOperator.Modulo:
          return Assignment(leftExpression,
            MultiplyExpression(middleOp, leftExpression, rightExpression));

        case MiddleOperator.BitwiseAnd:
        case MiddleOperator.BitwiseOr:
        case MiddleOperator.BitwiseXOr:
          return Assignment(leftExpression,
            BitwiseExpression(middleOp, leftExpression, rightExpression));

        default: // shift left, shift right
          return Assignment(leftExpression,
            ShiftExpression(middleOp, leftExpression, rightExpression));
      }
    }
```

The **Assignment** method performs simple or compound assignment.

```
    public static Expression Assignment(Expression leftExpression,
                                        Expression rightExpression,
                                        bool simpleAssignment = false) {
```

In cases of system calls, a specific register is assigned a value.

```
      Register? register = leftExpression.Register;
      if (register != null) {
        Symbol rightSymbol = rightExpression.Symbol;
```

We check that the register has the same size (in bytes) as the expression it is assigned to. If it is not, we report an error.

```
        Assert.Error(AssemblyCode.SizeOfRegister(register.Value) ==
                     rightExpression.Symbol.Type.Size(),
                     Message.Unmatched_register_size);
```

We add the long list of the right expression, since we need its value, to the final middle code list.

```
        List<MiddleCode> longList = new List<MiddleCode>();
        longList.AddRange(rightExpression.LongList);
```

We add the assignment of the value of right expression to the register.

```
        AddMiddleCode(longList, MiddleOperator.AssignRegister,
                      register, rightExpression.Symbol);
```

Since this an assignment, the code list holds both the assignment of the register and the side effect of the assignment. Therefore, we add the long list as both the short list and the long list of the result expression.

```
        return (new Expression(rightExpression.Symbol, longList, longList));
      }
```

If the left expression is not a register, we first check that the left expression is assignable. If it is not, we report an error.

```
      else {
        Assert.Error(leftExpression.Symbol.Assignable,
                     leftExpression, Message.Not_assignable);
        List<MiddleCode> longList = new List<MiddleCode>();
```

In case of a simple assignment we add the long list of the left expression to the code list. In case of a compound assignment, the long list of the left expression has already been added to long list of the right expression. If we added the long list in case of a compound assignment, we would add the same code twice.

```
        if (simpleAssignment) {
          longList.AddRange(leftExpression.LongList);
```

If the left expression is of floating type, we need to pop the floating value stack in case of a simple assignment. In case of a compound assignment, we let the value stay on the stack.

```
          if (leftExpression.Symbol.Type.IsFloating()) {
            AddMiddleCode(longList, MiddleOperator.PopFloat);
          }
        }
```

We add the long list of the right expression to the final code list.

```
        longList.AddRange(rightExpression.LongList);
```

If the expressions hold floating type, we really do not perform an assignment, we simply top the current value on the floating value stack to the left expression.

```
        if (leftExpression.Symbol.Type.IsFloating()) {
          AddMiddleCode(longList, MiddleOperator.TopFloat,
                        leftExpression.Symbol);
          List<MiddleCode> shortList = new List<MiddleCode>();
          shortList.AddRange(longList);
```

In case of floating type, there is a difference between the short list and long list. In the long list, the resulting value of the assignment (which equals the new value of the left expression) shall be used. Therefore, it is preserved on the stack. In the short list, the value is popped from the stack, since the value shall not be used, only the side effect of the assignment.

```
          AddMiddleCode(shortList, MiddleOperator.PopFloat);
          return (new Expression(leftExpression.Symbol, shortList, longList));
        }
```

In case of integral type, we add an assign instruction to the final code.

```
        else {
          if (leftExpression.Symbol.Type.IsStructOrUnion()) {
            AddMiddleCode(longList, MiddleOperator.AssignInit,
                          leftExpression.Symbol, rightExpression.Symbol);
          }

          AddMiddleCode(longList, MiddleOperator.Assign,
                        leftExpression.Symbol, rightExpression.Symbol);
          BigInteger? bitFieldMask =
                     leftExpression.Symbol.Type.GetBitfieldMask();
```

In case of bitfield, we add an add operation to set the non-relevant bits to zero.

```
          if (bitFieldMask != null) {
            Symbol maskSymbol = new Symbol(leftExpression.Symbol.Type,
                                           bitFieldMask);
            AddMiddleCode(longList, MiddleOperator.BitwiseAnd,
                          leftExpression.Symbol, leftExpression.Symbol,
                          maskSymbol);
          }
```

In the integral case, we set the code list as both the short list and long list in the resulting expression since the assignment is a side effect.

```
          return (new Expression(leftExpression.Symbol, longList, longList));
        }
      }
    }
```

4.5.3. The Condition Expression

The condition operator is rather complicated.

```
    public static Expression ConditionalExpression(Expression testExpression,
                                                 Expression trueExpression,
                                                 Expression falseExpression) {
```

We type cast the test expression to logical type.

```
      testExpression = TypeCast.ToLogical(testExpression);
```

If the test expression is constant, we simple return the true expression if the test expression is true and the false expression if it is false.

```
      if (ConstantExpression.IsConstant(testExpression)) {
        return ConstantExpression.IsTrue(testExpression)
               ? falseExpression : trueExpression;
      }
```

If both the true and false expressions hold logical types, we keep their types.

```
      if (trueExpression.Symbol.Type.IsLogical() &&
          falseExpression.Symbol.Type.IsLogical()) {
```

We start by backpatching the true-set and false-set of the test expression to the beginning of the true and false expression's code list.

```
        Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);
        Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);
```

The resulting true-set is the union of the true-sets of the true and false expression, and the resulting false-set is the union of the false-sets of the true and false expression.

```
        ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),
                         falseSet = new HashSet<MiddleCode>();
        trueSet.UnionWith(trueExpression.Symbol.TrueSet);
        trueSet.UnionWith(falseExpression.Symbol.TrueSet);
        falseSet.UnionWith(trueExpression.Symbol.FalseSet);
        falseSet.UnionWith(falseExpression.Symbol.FalseSet);

        List<MiddleCode> shortList = new List<MiddleCode>();
```

If the short lists of both the true and false expression is empty, it does not matter if the test expression is true or false, and we let the resulting short list be the short list of the true expression (which may be empty).

```
        if (IsEmpty(trueExpression.ShortList) &&
            IsEmpty(falseExpression.ShortList)) {
          shortList.AddRange(testExpression.ShortList);
        }
```

If the short list of the test expression is not empty, the situation becomes a bit more complicated. We add the long list, rather than the short list, of the test expression to the final short list since we need the value of the test expression in order to jump to the beginning of the short list of either the true of false expression.

```
        else {
          shortList.AddRange(testExpression.LongList);
          shortList.AddRange(trueExpression.ShortList);
          shortList.AddRange(falseExpression.ShortList);
        }
```

We add the long list of the test, true, and false expression to the final long list.

```
        List<MiddleCode> longList = new List<MiddleCode>();
        longList.AddRange(testExpression.LongList);
        longList.AddRange(trueExpression.LongList);
        longList.AddRange(falseExpression.LongList);
```

Finally, we create a new symbol with logical type and the resulting true-sets and false-sets.

```
        Symbol symbol = new Symbol(trueSet, falseSet);
        return (new Expression(symbol, shortList, longList));
      }
```

If at least one of the true or false expression does not hold logical type, we define **maxType** as their largest type, and type cast both expressions to that type.

```
      else {
        Type maxType = TypeCast.MaxType(trueExpression.Symbol.Type,
                                        falseExpression.Symbol.Type);
        trueExpression = TypeCast.ImplicitCast(trueExpression, maxType);
        Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);
```

We create a new temporary symbol to hold the result of the condition expression.

```
Symbol symbol = new Symbol(maxType);
```

In case of non-floating type, we add the assignment instruction. In case of a floating type, the value is already placed at the floating value stack and we do not need to do anything.

```
if (maxType.IsFloating()) {
  AddMiddleCode(trueExpression.LongList,
                MiddleOperator.DecreaseStack);
}
else { // XXX
  if (trueExpression.Symbol.IsTemporary()) {
    foreach (MiddleCode middleCode in trueExpression.LongList) {
      if (middleCode[0] == trueExpression.Symbol) {
        middleCode[0] = symbol;
      }
    }
  }
  else {
    AddMiddleCode(trueExpression.LongList, MiddleOperator.Assign,
                  symbol, trueExpression.Symbol);
  }
}
```

We add the jump to a target code to both the short and long list of the true expression. The target code will be added after the code of the false expression. Its purpose is to jump over the false expression code.

```
MiddleCode targetCode = new MiddleCode(MiddleOperator.Empty);
AddMiddleCode(trueExpression.ShortList,
              MiddleOperator.Jump, targetCode);
AddMiddleCode(trueExpression.LongList,
              MiddleOperator.Jump, targetCode);
```

Similar to the true expression, we type cast and false expression, and backpatch the true-set of the test expression to the beginning of the false expression code.

```
falseExpression = TypeCast.ImplicitCast(falseExpression, maxType);
Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);
```

We also assign the value of the false expression to the temporary symbol if it does not hold floating type. If it holds floating type, the value is already placed at the top of the floating value stack, and we do nothing.

```
if (!maxType.IsFloating()) {
  if (falseExpression.Symbol.IsTemporary()) {
    foreach (MiddleCode middleCode in falseExpression.LongList) {
      if (middleCode[0] == falseExpression.Symbol) {
        middleCode[0] = symbol;
      }
    }
  }
  else {
    AddMiddleCode(falseExpression.LongList, MiddleOperator.Assign,
                  symbol, falseExpression.Symbol);
  }
}
```

If both the short lists of the true and false expressions are empty, we just add the short list of the test expression (which may be empty).

```
        List<MiddleCode> shortList = new List<MiddleCode>();
        if (IsEmpty(trueExpression.ShortList) &&
            IsEmpty(falseExpression.ShortList)) {
          shortList.AddRange(testExpression.ShortList);
        }
```

If not both the short lists of the true and false expressions are empty, we add the long list of the test expression, rather than the short list, since we need to value of the test expression, as well as the short lists of the true and false expressions.

```
        else {
          shortList.AddRange(testExpression.LongList);
          shortList.AddRange(trueExpression.ShortList);
          shortList.AddRange(falseExpression.ShortList);
          shortList.Add(targetCode);
        }
```

Finally, add the long lists and return the expression.

```
        List<MiddleCode> longList = new List<MiddleCode>();
        longList.AddRange(testExpression.LongList);
        longList.AddRange(trueExpression.LongList);
        longList.AddRange(falseExpression.LongList);
        longList.Add(targetCode);

        return (new Expression(symbol, shortList, longList));
      }
    }
```

4.5.4. Constant Integral Expression

The **ConstantIntegralExpression** methods makes sure the expression is indeed constant and either integral or pointer.

```
    public static Expression ConstantIntegralExpression(Expression expression)
    { expression = ConstantExpression.Cast(expression,
                                           Type.SignedLongIntegerType);
      Assert.Error(expression != null, expression,
                   Message.Non__constant_expression);
      Assert.Error(expression.Symbol.Type.IsIntegralOrPointer(),
                   expression.Symbol, Message.Non__integral_expression);
      return expression;
    }
```

4.5.5. The Logical Or Expression

The **LogicalOrExpression** method backpatches the true-set of the first operand to the beginning of the right operand. This can be interpreted as if the left expression is true, we try the right expression thereafter. If both the expressions are true, the result is true. However, if the left expression is false, the result is determined to be false without evaluating the right expression.

```
MiddleCodeGenerator.cs
    public static Expression LogicalOrExpression(Expression leftExpression,
                                                 Expression rightExpression) {
```

We check if the expression is constant. If it is constant, we return the constant expression.

```
      Expression constantExpression =
        ConstantExpression.Logical(MiddleOperator.LogicalOr,
                                         leftExpression, rightExpression);

      if (constantExpression != null) {
        return constantExpression;
      }
```

We type cast both the expressions to the logical type.

```
      leftExpression = TypeCast.ToLogical(leftExpression);
      rightExpression = TypeCast.ToLogical(rightExpression);
```

For the resulting expression to be true, it is enough that one of the left or right expression is true. Therefore, the true-set of the resulting expression is the union of the true-set of the left and right expression. If the left expression is evaluated to true, the right expression (including its side effects) shall not be evaluated. The false-sets, on the other hand, are different. If the left expression is evaluated to false, we need to evaluate the right expression. Therefore, we backpatch the false-set of the left expression to the beginning of the right expression code. The false-set of the resulting expression is the false-set of the right expression.

```
      ISet<MiddleCode> trueSet = new HashSet<MiddleCode>();
      trueSet.UnionWith(leftExpression.Symbol.TrueSet);
      trueSet.UnionWith(rightExpression.Symbol.TrueSet);
      Backpatch(leftExpression.Symbol.FalseSet, rightExpression.LongList);
      Symbol symbol = new Symbol(trueSet, rightExpression.Symbol.FalseSet);

      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);

      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);

      return (new Expression(symbol, shortList, longList));
    }
```

4.5.1. The Logical And Expression

The **LogicalAndExpression** method works in the opposite way compared to the logical **or** expression case above. If the left expression is true, we shall continue to evaluate the false expression. The resulting expression is true only if both the left and right expression is true. However, if the left expression is false the resulting expression is false, without evaluation of the right expression. Therefore, we backpatch the false-set of the first operand to the beginning of the right operand. The true-set of the resulting expression is the true-set of the right expression, and the false set is the union of the false sets of the left and right expression.

```
    public static Expression LogicalAndExpression(Expression leftExpression,
                                                  Expression rightExpression){
      Expression constantExpression =
        ConstantExpression.Logical(MiddleOperator.LogicalAnd,
                                   leftExpression, rightExpression);

      if (constantExpression != null) {
        return constantExpression;
      }

      leftExpression = TypeCast.ToLogical(leftExpression);
      rightExpression = TypeCast.ToLogical(rightExpression);

      ISet<MiddleCode> falseSet = new HashSet<MiddleCode>();
      falseSet.UnionWith(leftExpression.Symbol.FalseSet);
      falseSet.UnionWith(rightExpression.Symbol.FalseSet);

      Backpatch(leftExpression.Symbol.TrueSet, rightExpression.LongList);
      Symbol symbol = new Symbol(rightExpression.Symbol.TrueSet, falseSet);

      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);

      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);

      return (new Expression(symbol, shortList, longList));
    }
```

1.1.1. Bitwise Expressions

There are three bitwise operators: inclusive **or**, exclusive **or** (**xor**), and **and**.

```
    public static Expression BitwiseExpression(MiddleOperator middleOp,
                                               Expression leftExpression,
                                               Expression rightExpression) {
```

If the expression can be evaluated to a constant value, we return the constant expression.

```
      Expression constantExpression = ConstantExpression.
         Arithmetic(middleOp, leftExpression, rightExpression);

      if (constantExpression != null) {
        return constantExpression;
      }
```

We find the maximal type of the left and right expression.

```
      Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,
                                      rightExpression.Symbol.Type);
      Symbol resultSymbol = new Symbol(maxType);
```

The type of the left and right expression must be integral or pointer. Array, string, and functions are converted to pointers.

```
      Assert.Error(maxType.IsIntegralPointerArrayStringOrFunction(),
                   maxType, Message.Invalid_type_in_bitwise_expression);
```

We type cast both expressions into the maximal type.

```
      leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);
      rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);
```

The short list of the resulting expression is simply the short list of the expressions.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);
```

The long list of the resulting expression is the long list of the expressions, and the bitwise operation.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);

      AddMiddleCode(longList, middleOp, resultSymbol,
                    leftExpression.Symbol, rightExpression.Symbol);
      return (new Expression(resultSymbol, shortList, longList));
    }
```

4.5.2. Shift Expression

In C, there are both left shift and right shift.

```
    public static Expression ShiftExpression(MiddleOperator middleOp,
                                             Expression leftExpression,
                                             Expression rightExpression) {
```

First, we check that the left expression holds integral or pointer type. An array, a string, or a function is cast to a pointer.

```
      Assert.Error(leftExpression.Symbol.Type.
                   IsIntegralPointerArrayStringOrFunction(),
                   leftExpression, Message.Invalid_type_in_shift_expression);
```

If the expression can be evaluated to a constant expression, we return the constant expression.

```
      Expression constantExpression =
        ConstantExpression.Arithmetic(middleOp, leftExpression,
                                      rightExpression);
      if (constantExpression != null) {
        return constantExpression;
      }
```

The right expression is type cast to an unsigned character, which always have a size of one byte.

```
      rightExpression =
        TypeCast.ImplicitCast(rightExpression, Type.UnsignedCharType);
```

The final short list is simple the short lists of the left and right expression.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);
```

The final short list is simple the short lists of the left and right expression.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);
```

The final long list is the short lists of the left and right expression, and the shift operation.

```
      Symbol resultSymbol = new Symbol(leftExpression.Symbol.Type);
      AddMiddleCode(longList, middleOp, resultSymbol,
                    leftExpression.Symbol, rightExpression.Symbol);
      return (new Expression(resultSymbol, shortList, longList));
    }
```

4.5.3. Equality and Relation Expressions

In C, there are the two equality operators **equal** and **not equal**, as well as the relational operators **less than**, **less than or equal**, **greater than**, and **greater than or equal**.

```
public static Expression RelationalExpression(MiddleOperator middleOp,
                                              Expression leftExpression,
                                              Expression rightExpression){
```

First, we check the types of the expression. Everything except struct or union can be compared.

```
  Assert.Error(!leftExpression.Symbol.Type.IsStructOrUnion(),
               leftExpression,
               Message.Invalid_type_in_expression);
  Assert.Error(!rightExpression.Symbol.Type.IsStructOrUnion(),
               rightExpression,
               Message.Invalid_type_in_expression);

  Expression constantExpression =
    ConstantExpression.Relation(middleOp, leftExpression,
                                rightExpression);
  if (constantExpression != null) {
    return constantExpression;
  }
```

The find the maximal type of the left and right expression types, and type cast both the expressions to that type.

```
  Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,
                                  rightExpression.Symbol.Type);

  leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);
  rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);
```

The final short list is simply the short lists of the left end right expression.

```
  List<MiddleCode> shortList = new List<MiddleCode>();
  shortList.AddRange(leftExpression.ShortList);
  shortList.AddRange(rightExpression.ShortList);
```

If one of the expression types is signed and the other is unsigned, we make sure they are both signed.

```
  if (leftExpression.Symbol.Type.IsSigned() &&
      rightExpression.Symbol.Type.IsUnsigned()) {
    rightExpression =
      TypeCast.ImplicitCast(rightExpression, leftExpression.Symbol.Type);
  }
  else if (leftExpression.Symbol.Type.IsUnsigned() &&
           rightExpression.Symbol.Type.IsSigned()) {
    leftExpression =
      TypeCast.ImplicitCast(leftExpression, rightExpression.Symbol.Type);
  }
```

The final long list is the long lists of the left end right expression, to begin with.

```
  List<MiddleCode> longList = new List<MiddleCode>();
  longList.AddRange(leftExpression.LongList);
  longList.AddRange(rightExpression.LongList);
```

We add two instruction to the long list. The first is an if-goto instruction that we add to the true-set. If the expression is true, this instruction will jump to the target that later will be backpatched into the instruction. In the same way, we add a goto instruction to the false-set. If the expression is false, the instruction will jump to the target later backpatched to the instruction.

```
      ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),
                       falseSet = new HashSet<MiddleCode>();
      trueSet.Add(AddMiddleCode(longList, middleOp, null,
                                leftExpression.Symbol,
                                rightExpression.Symbol));
      falseSet.Add(AddMiddleCode(longList, MiddleOperator.Jump));
```

The final symbol holds logical type with true and false-set.

```
      Symbol symbol = new Symbol(trueSet, falseSet);
      return (new Expression(symbol, shortList, longList));
    }
```

4.5.4. Addition Expression

```
    public static Expression MultiplySize(Expression arrayExpression,
                                          Expression indexExpression) {
      Type arrayType = arrayExpression.Symbol.Type;

      if (arrayType.PointerOrArrayType.Size() > 1) {
        int size = arrayType.PointerOrArrayType.Size();
        Symbol sizeSymbol =
          new Symbol(indexExpression.Symbol.Type, new BigInteger(size));
        Expression sizeExpression = new Expression(sizeSymbol);
        indexExpression = MultiplyExpression(MiddleOperator.Multiply,
                                             indexExpression, sizeExpression);
      }

      return TypeCast.ImplicitCast(indexExpression, arrayType);
    }
```

The addition and subtraction expressions are a little bit complicated, since it is possible to perform pointer arithmetic.

```
    public static Expression AdditionExpression(Expression leftExpression,
                                                Expression rightExpression) {
      Type leftType = leftExpression.Symbol.Type,
           rightType = rightExpression.Symbol.Type;
```

In an addition expression, we have three allowed type combinations.

- Both types are arithmetic.
- The left type is a pointer or an array, not to void or function, and the right type is an integral.
- The left type is an integral, and the right type is a pointer or an array, not to void or function.

```
      Assert.Error((leftType.IsArithmetic() && rightType.IsArithmetic()) ||
                   (leftType.IsPointerOrArray() && rightType.IsIntegral() &&
                    !leftType.PointerOrArrayType.IsVoid() &&
                    !leftType.PointerOrArrayType.IsFunction()) ||
                   (leftType.IsIntegral() && rightType.IsPointerOrArray() &&
                    !rightType.PointerOrArrayType.IsVoid() &&
                    !rightType.PointerOrArrayType.IsFunction()),
                   leftExpression, Message.Non__arithmetic_expression);
```

We check if the expression is a constant expression, in which case we return the constant expression.

```
      Expression constantExpression =
        ConstantExpression.Arithmetic(MiddleOperator.Add,
                                      leftExpression, rightExpression);
      if (constantExpression != null) {
        return constantExpression;
      }
```

We also check if the expression is a static expression, in which case we return the static expression.

```
      Expression staticExpression =
        StaticExpression.Binary(MiddleOperator.Subtract,
                                leftExpression, rightExpression);
      if (staticExpression != null) {
        return staticExpression;
      }
```

If one of the expressions is a pointer or an array, we multiply the other expression with the size of the pointer type or array type in order to provide pointer arithmetic.

```
      if (leftType.IsPointerOrArray()) {
        rightExpression = MultiplySize(leftExpression, rightExpression);
      }

      if (rightType.IsPointerOrArray()) {
        leftExpression = MultiplySize(rightExpression, leftExpression);
      }
```

We call **MaxType** in **TypeCast** to obtain the largest type of the operands, and type cast both the operands to that type.

```
      Type maxType = TypeCast.MaxType(leftType, rightType);
      leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);
      rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);
```

The final short list is simple the short lists of the left and right expression.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);
```

The final long list is the long lists of the left and right expression, and the addition operation.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);
```

The resulting expression if the sum of the left and right expression.

```
      Symbol resultSymbol = new Symbol(maxType);
      AddMiddleCode(longList, MiddleOperator.Add, resultSymbol,
                    leftExpression.Symbol, rightExpression.Symbol);
      return (new Expression(resultSymbol, shortList, longList));
    }
```

4.5.5. Subtraction Expression

Similar to the addition case above, we have to take pointer arithmetic in consideration in the subtraction case also.

```
    public static Expression SubtractionExpression(Expression leftExpression,
                                                   Expression rightExpression)
    { Type leftType = leftExpression.Symbol.Type,
           rightType = rightExpression.Symbol.Type;
```

In a subtraction expression, we have three allowed type combinations.

- Both types are arithmetic.
- The left type is a pointer or array, and the right type is an integral.
- Both types are pointers or arrays, and their pointer or array types have the same size.

```
      Assert.Error((leftType.IsArithmetic() && rightType.IsArithmetic()) ||
                   (leftType.IsPointerOrArray() && rightType.IsIntegral()) ||
                   (leftType.IsPointerOrArray() &&
                    rightType.IsPointerOrArray() &&
                    (leftType.PointerOrArrayType.Size() ==
                     rightType.PointerOrArrayType.Size())),
                   leftExpression, Message.Non__arithmetic_expression);
```

If the left or right operand is a pointer or array, their pointer or array types may not be void or function.

```
      if (leftType.IsPointerOrArray()) {
        Assert.Error(!leftType.PointerOrArrayType.IsVoid() &&
                     !leftType.PointerOrArrayType.IsFunction(),
                     leftExpression, Message.Non__arithmetic_expression);
      }

      if (rightType.IsPointerOrArray()) {
        Assert.Error(!rightType.PointerOrArrayType.IsVoid() &&
                     !rightType.PointerOrArrayType.IsFunction(),
                     rightExpression, Message.Non__arithmetic_expression);
      }
```

In case of a constant expression we return the constant expression, and in case of a static expression we return the static expression.

```
      Expression constantExpression =
        ConstantExpression.Arithmetic(MiddleOperator.Subtract,
                                      leftExpression, rightExpression);
      if (constantExpression != null) {
        return constantExpression;
      }

      Expression staticExpression =
        StaticExpression.Binary(MiddleOperator.Subtract,
                                leftExpression, rightExpression);
      if (staticExpression != null) {
        return staticExpression;
      }
```

We call **MaxType** in **TypeCast** to obtain the largest type of the left and right expression, and type cast both the expressions to that type.

```
      Type maxType = TypeCast.MaxType(leftType, rightType);
      leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);
      rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);
```

The resulting short list is simply the short lists of the left and right expressions.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);
```

The resulting long list is the long lists of the left and right expressions, and the subtraction operation.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);
```

The resulting expression if the difference of the left and right expression.

```
      Symbol resultSymbol = new Symbol(maxType);
      AddMiddleCode(longList, MiddleOperator.Subtract, resultSymbol,
                    leftExpression.Symbol, rightExpression.Symbol);
      Expression resultExpression =
        new Expression(resultSymbol, shortList, longList);
```

If both the left and right expression hold pointer or array types, the resulting type shall be signed integer. Moreover, we need to divide the result with the size or their pointer or array size, unless it is one byte.

```
      if (leftType.IsPointerOrArray() && rightType.IsPointerOrArray()) {
        resultExpression =
          TypeCast.ImplicitCast(resultExpression, Type.SignedIntegerType);

        if (leftType.PointerOrArrayType.Size() > 1) {
          int size = leftType.PointerOrArrayType.Size();
          Symbol sizeSymbol =
            new Symbol(Type.SignedIntegerType, new BigInteger(size));
          Expression sizeExpression = new Expression(sizeSymbol);
          resultExpression = MultiplyExpression(MiddleOperator.Divide,
                                         resultExpression, sizeExpression);
        }
      }

      return resultExpression;
    }
```

4.5.6. Multiplication Expressions

A multiplication expression is a multiplication, division, or modulo.

```
    public static Expression MultiplyExpression(MiddleOperator middleOp,
                                                Expression leftExpression,
                                                Expression rightExpression) {
```

Similar to the addition and subtraction cases above, we check if the expression is a constant expression. However, we do not check whether the expression static, since there is are no static expression with the multiplication operators.

```
      Expression constantExpression =
        ConstantExpression.Arithmetic(middleOp, leftExpression,
                                      rightExpression);
      if (constantExpression != null) {
        return constantExpression;
      }

      Type leftType = leftExpression.Symbol.Type,
           rightType = rightExpression.Symbol.Type;
```

In case of signed or unsigned module, we check that the left and right expression holds integral types.

```
      if (middleOp == MiddleOperator.Modulo) {
        Assert.Error(leftType.IsIntegral() && rightType.IsIntegral(),
                     Message.Invalid_type_in_expression);
      }
```

In case of signed or unsigned multiplication or division, we check that the left and right expression holds arithmetic types.

```
      else {
        Assert.Error(leftType.IsArithmetic() && rightType.IsArithmetic(),
                     Message.Invalid_type_in_expression);
      }
```

We call **MaxType** in **TypeCast** to obtain the largest type, and we type cast both expressions to that type.

```
      Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,
                                      rightExpression.Symbol.Type);
      leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);
      rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);
```

If one of the expression types is signed and the other is unsigned, we make sure they are both signed.

```
      if (leftExpression.Symbol.Type.IsSigned() &&
          rightExpression.Symbol.Type.IsUnsigned()) {
        rightExpression =
          TypeCast.ImplicitCast(rightExpression, leftExpression.Symbol.Type);
      }
      else if (leftExpression.Symbol.Type.IsUnsigned() &&
               rightExpression.Symbol.Type.IsSigned()) {
        leftExpression =
          TypeCast.ImplicitCast(leftExpression, rightExpression.Symbol.Type);
      }
```

The short list and the long list are made up by the short and long lists of the left and right expression.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(leftExpression.ShortList);
      shortList.AddRange(rightExpression.ShortList);

      List<MiddleCode> longList = new List<MiddleCode>();
      longList.AddRange(leftExpression.LongList);
      longList.AddRange(rightExpression.LongList);
```

The resulting expression is the operations.

```
      Symbol resultSymbol = new Symbol(maxType);
      AddMiddleCode(longList, middleOp, resultSymbol,
                    leftExpression.Symbol, rightExpression.Symbol);
      return (new Expression(resultSymbol, shortList, longList));
    }
```

4.5.7. Cast Expressions

The TypeName method returns the type of the expression. The **specifier** parameter holds the resulting type of the declaration specifier list.

```
    public static Type TypeName(Specifier specifier, Declarator declarator) {
      Type specifierType = specifier.Type;
```

If the declarator is not null, we add the specifier's type to the declarator, and returns its type.

```
      if (declarator != null) {
        declarator.Add(specifierType);
        return declarator.Type;
      }
```

If the declarator is null, we just return the specifier's type.

```
      else {
        return specifierType;
      }
    }
```

4.5.8. Unary Addition Expressions

The unary addition operators are plus and minus. The operand must be a logical or arithmetic value.

```
    public static Expression UnaryExpression(MiddleOperator middleOp,
                                             Expression expression) {
      Type type = expression.Symbol.Type;
      Assert.Error(type.IsLogical() || type.IsArithmetic (),
                   expression, Message.Non__arithmetic_expression);

      Expression constantExpression =
        ConstantExpression.Arithmetic(middleOp, expression);
      if (constantExpression != null) {
        return constantExpression;
      }

      Symbol resultSymbol = new Symbol(expression.Symbol.Type);
      AddMiddleCode(expression.LongList, middleOp,
                    resultSymbol, expression.Symbol);
      return (new Expression(resultSymbol, expression.ShortList,
                             expression.LongList));
    }
```

4.5.9. Logical Not Expression

The **LogicalNotExpression** method swap the true-set and false-set of the operand. Note that the method does not actually add any middle code, it just swaps the sets.

```
    public static Expression LogicalNotExpression(Expression expression) {
      Expression constantExpression =
        ConstantExpression.LogicalNot(expression);
      if (constantExpression != null) {
        return constantExpression;
      }
```

The resulting expression is the original expression with swapped true and false-sets.

```
      expression = TypeCast.ToLogical(expression);
      Symbol notSymbol =
        new Symbol(expression.Symbol.FalseSet, expression.Symbol.TrueSet);
      return (new Expression(notSymbol, expression.ShortList,
                             expression.LongList));
    }
```

4.5.10. Bitwise Not Expression

The **BitwiseNotExpression** method generates code for the bitwise not unary operator.

```
public static Expression BitwiseNotExpression(Expression expression) {
  expression = TypeCast.LogicalToIntegral(expression);
  Assert.Error(expression.Symbol.Type.IsIntegral(),
               Message.Only_integral_values_for_bitwise_not);
  Expression constantExpression =
    ConstantExpression.Arithmetic(MiddleOperator.BitwiseNot, expression);
  if (constantExpression != null) {
    return constantExpression;
  }

  expression = TypeCast.LogicalToIntegral(expression);
  Symbol resultSymbol = new Symbol(expression.Symbol.Type);
  AddMiddleCode(expression.LongList, MiddleOperator.BitwiseNot,
                resultSymbol, expression.Symbol);
  return (new Expression(resultSymbol, expression.ShortList,
                         expression.LongList));
}
```

4.5.11. The sizeof Expression

The first **sizeof** operator takes a type or an expression as operand.

```
public static Expression SizeOfExpression(Expression expression) {
```

We check that the storage of the expression is not **register**, since it is not allowed.

```
  Assert.Error(!expression.Symbol.IsRegister(), expression,
               Message.Register_storage_not_allowed_in_sizof_expression);
```

Moreover, we check that it is not a function or a bitfield, since they do not have sizes.

```
  Type type = expression.Symbol.Type;
  Assert.Error(!type.IsFunction(),
               Message.Sizeof_applied_to_function_not_allowed);
  Assert.Error(!type.IsBitfield(),
               Message.Sizeof_applied_to_bitfield_not_allowed);
```

We create a symbol of signed integer types with the size as its value.

```
  Symbol symbol = new Symbol(Type.SignedIntegerType,
                             (BigInteger) (expression.Symbol.Type.Size()));
  return (new Expression(symbol, new List<MiddleCode>(),
                         new List<MiddleCode>()));
}
```

In the case of the type name, we also check that the expression is not a function or a bitfield.

```
public static Expression SizeOfType(Type type) {
  Assert.Error(!type.IsFunction(),
               Message.Sizeof_applied_to_function_not_allowed);
  Assert.Error(!type.IsBitfield(),
               Message.Sizeof_applied_to_bitfield_not_allowed);
  Symbol symbol =
    new Symbol(Type.SignedIntegerType, (BigInteger) type.Size());
  return (new Expression(symbol, new List<MiddleCode>(),
                         new List<MiddleCode>()));
}
```

4.5.12. Prefix Increment and Decrement Expression

MiddleCodeGenerator.cs

```
    private static IDictionary<MiddleOperator,MiddleOperator> m_incrementMap =
      new Dictionary<MiddleOperator, MiddleOperator>() {
        {MiddleOperator.Increment, MiddleOperator.Add},
        {MiddleOperator.Decrement, MiddleOperator.Subtract}};

    private static IDictionary<MiddleOperator,MiddleOperator>
      m_incrementInverseMap = new Dictionary<MiddleOperator,MiddleOperator>(){
        {MiddleOperator.Increment, MiddleOperator.Subtract},
        {MiddleOperator.Decrement, MiddleOperator.Add}};
    public static Expression PrefixIncrementExpression
                         (MiddleOperator middleOp, Expression expression){
      Symbol symbol = expression.Symbol;
      Assert.Error(symbol.IsAssignable(),  Message.Not_assignable);
      Assert.Error(symbol.Type.IsArithmeticOrPointer(),
                   expression, Message.Invalid_type_in_increment_expression);

      if (symbol.Type.IsIntegralOrPointer()) {
        Symbol oneSymbol = new Symbol(symbol.Type, BigInteger.One);
        AddMiddleCode(expression.ShortList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.LongList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);

        BigInteger? bitFieldMask = symbol.Type.GetBitfieldMask();
        if (bitFieldMask != null) {
          Symbol maskSymbol = new Symbol(symbol.Type, bitFieldMask.Value);
          MiddleCode maskCode = new MiddleCode(MiddleOperator.BitwiseAnd,
                                     symbol, symbol, maskSymbol);
          expression.ShortList.Add(maskCode);
          expression.LongList.Add(maskCode);
        }

        Symbol resultSymbol = new Symbol(symbol.Type);
        AddMiddleCode(expression.LongList, MiddleOperator.Assign,
                      resultSymbol, symbol);

        return (new Expression(resultSymbol, expression.ShortList,
                               expression.LongList));
      }
```

The increment and decrement operator apply not only to integral values, but also to floating values. The operation. We start by pushing the value one at the floating value stack, the value to be incremented or decremented has already been pushed at the stack by earlier operations. We perform the operation, which is addition or subtraction. We preform the operation on both the short list and list of the expression. The difference is that in the short list case we pop the value off the stack since we do not need it anymore. In the long list case we do nothing, we just let the value stay on the stack to be used by later operations.

```
      else {
        AddMiddleCode(expression.ShortList, MiddleOperator.PushOne);
        Symbol oneSymbol = new Symbol(symbol.Type, (decimal) 1);
        AddMiddleCode(expression.ShortList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.ShortList, MiddleOperator.PopFloat, symbol);
        AddMiddleCode(expression.LongList, MiddleOperator.PushOne);
        AddMiddleCode(expression.LongList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.LongList, MiddleOperator.TopFloat, symbol);

        Symbol resultSymbol = new Symbol(symbol.Type);
        return (new Expression(resultSymbol, expression.ShortList,
                               expression.LongList));
      }
    }
```

4.5.13. Postfix Increment and Decrement Expression

MiddleCodeGenerator.cs

```
    public static Expression PostfixIncrementExpression
                             (MiddleOperator middleOp, Expression expression){
      Symbol symbol = expression.Symbol;
      Assert.Error(symbol.IsAssignable(), Message.Not_assignable);
      Assert.Error(symbol.Type.IsArithmeticOrPointer(),
                   expression, Message.Invalid_type_in_increment_expression);

      if (symbol.Type.IsIntegralOrPointer()) {
        Symbol resultSymbol = new Symbol(symbol.Type);
        AddMiddleCode(expression.LongList, MiddleOperator.Assign,
                      resultSymbol, symbol);
        Symbol oneSymbol = new Symbol(symbol.Type, BigInteger.One);
        AddMiddleCode(expression.ShortList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.LongList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);

        BigInteger? bitFieldMask = symbol.Type.GetBitfieldMask();
        if (bitFieldMask != null) {
          Symbol maskSymbol = new Symbol(symbol.Type, bitFieldMask.Value);
          AddMiddleCode(expression.ShortList, MiddleOperator.BitwiseAnd,
                        symbol, symbol, maskSymbol);
          AddMiddleCode(expression.LongList, MiddleOperator.BitwiseAnd,
                        symbol, symbol, maskSymbol);
        }

        return (new Expression(resultSymbol, expression.ShortList,
                               expression.LongList));
      }
      else {
        AddMiddleCode(expression.ShortList, MiddleOperator.PushOne);
        Symbol oneSymbol = new Symbol(symbol.Type, (decimal) 1);
        AddMiddleCode(expression.ShortList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.ShortList, MiddleOperator.PopFloat, symbol);
```

In the long list case, the situation becomes a little more complicated, since the result of the operations shall be the original value, not the resulting value. Therefore, we must perform the inverse operation on the value on the stack in order to return to the original value.

```
        AddMiddleCode(expression.LongList, MiddleOperator.PushOne);
        AddMiddleCode(expression.LongList, m_incrementMap[middleOp],
                      symbol, symbol, oneSymbol);
        AddMiddleCode(expression.LongList, MiddleOperator.TopFloat, symbol);

        AddMiddleCode(expression.LongList, MiddleOperator.PushOne);
        AddMiddleCode(expression.LongList, m_incrementInverseMap[middleOp],
                      symbol, symbol, oneSymbol);

        Symbol resultSymbol = new Symbol(symbol.Type);
        return (new Expression(resultSymbol, expression.ShortList,
                               expression.LongList));
      }
    }
```

4.5.14. Address Expression // XXX

```
    public static Expression AddressExpression(Expression expression) {
      Assert.Error(!symbol.IsRegister() && !symbol.Type.IsBitfield(),
                   expression,  Message.Not_addressable);
```

The address operator may result in a static address.

```
      Expression staticExpression =
        StaticExpression.Unary(MiddleOperator.Address, expression);
      if (staticExpression!= null) {
        return staticExpression ;
      }
```

If the expression has floating type, we need to pop the value from the floating value stack.

```
      if (expression.Symbol.Type.IsFloating()) {
        AddMiddleCode(expression.LongList, MiddleOperator.PopFloat);
      }
```

The type of the resulting expression is a pointer to the type of the original expression

```
      Symbol resultSymbol = new Symbol(new Type(expression.Symbol.Type));
      AddMiddleCode(expression.LongList, MiddleOperator.Address,
                    resultSymbol, expression.Symbol);
      return (new Expression(resultSymbol, expression.ShortList,
                             expression.LongList));
    }
```

4.5.15. Dereference Expression

To begin with, we look into the Dereference method, which is called by **DereferenceExpression**, **ArrowExprerssion**, and **IndexExpression**. It generates code for the dereference operator where the offset is constant, which it is in case of dereference expressions, arrow expressions, and index expressions with constant index values. The only exception is index expressions with non-constant index values, in which case we need to generate different code.

```
    private static Expression Dereference(Expression expression,
                                          Symbol resultSymbol, int offset) {
      resultSymbol.AddressSymbol = expression.Symbol;
      resultSymbol.AddressOffset = offset;
      AddMiddleCode(expression.LongList, MiddleOperator.Dereference,
                    resultSymbol, expression.Symbol, 0);
```

If the resulting expression holds floating type, we need to pop the value from the floating value stack.

```
    if (resultSymbol.Type.IsFloating()) {
      AddMiddleCode(expression.LongList, MiddleOperator.PushFloat,
                    resultSymbol);
    }

    return (new Expression(resultSymbol, expression.ShortList,
                           expression.LongList));
  }
```

The dereference operator applies to pointers, arrays, strings, and functions. In the function case, the expression is regarded as a pointer to a function.

```
  public static Expression DereferenceExpression(Expression expression) {
    Assert.Error(expression.Symbol.Type.IsPointerArrayOrString(),
                 Message.Invalid_dereference_of_non__pointer);
    Symbol resultSymbol =
      new Symbol(expression.Symbol.Type.PointerOrArrayType);
    return Dereference(expression, resultSymbol, 0);
  }
```

4.5.16. Arrow Expression

The **ArrowExpression** method does also call **Dereference** after checking that the expression type and member name is valid.

```
  public static Expression ArrowExpression(Expression expression,
                                           string memberName) {
    Assert.Error(expression.Symbol.Type.IsPointer() &&
                 expression.Symbol.Type.PointerType.IsStructOrUnion(),
                 expression, Message.
                 Not_a_pointer_to_a_struct_or_union_in_arrow_expression);
    Assert.Error(expression.Symbol.Type.PointerType.MemberMap != null,
                 expression, Message.
                 Member_access_of_uncomplete_struct_or_union);

    Symbol memberSymbol;
    Assert.Error(expression.Symbol.Type.PointerType.MemberMap.
                 TryGetValue(memberName, out memberSymbol),
                 memberName, Message.Unknown_member_in_arrow_expression);

    Symbol resultSymbol = new Symbol(memberSymbol.Type);
    return Dereference(expression, resultSymbol, memberSymbol.Offset);
  }
```

4.5.17. Index Expression

In an index expression one of the expressions shall be a pointer or an array, while the other expression is an integral value. For instance, if **a** is an array and **i** an integer value, both **a[i]** and **i[a]** are valid index expressions.

```
  public static Expression IndexExpression(Expression leftExpression,
                                           Expression rightExpression) {
    Type leftType = leftExpression.Symbol.Type,
         rightType = rightExpression.Symbol.Type;
```

On of the types must be a non-void pointer, array, or string, and the other type must be integral.

```
    Assert.Error((leftType.IsPointerArrayOrString() &&
                 !leftType.PointerOrArrayType.IsVoid() &&
                 rightType.IsIntegral()) ||
                (leftType.IsIntegral() &&
                 rightType.IsPointerArrayOrString() &&
                 !rightType.PointerOrArrayType.IsVoid()),
                null, Message.Invalid_type_in_index_expression);
```

Similar to the cases above we check for static expression. However, an index expression cannot be constant.

```
Expression staticExpression =
  StaticExpression.Binary(MiddleOperator.Index, leftExpression,
                          rightExpression);
if (staticExpression != null) {
  return staticExpression;
}
```

Note that either of the left and right expression may be the array or index expression. Therefore, we check which expression is a pointer or an array.

```
Expression arrayExpression, indexExpression;

if (leftExpression.Symbol.Type.IsPointerOrArray()) {
  arrayExpression = leftExpression;
  indexExpression = rightExpression;
}
else {
  indexExpression = leftExpression;
  arrayExpression = rightExpression;
}

Type arrayType = arrayExpression.Symbol.Type,
     indexType = indexExpression.Symbol.Type;
Symbol resultSymbol = new Symbol(arrayType.PointerOrArrayType);
```

If the index value is constant, we can call **Dereference** with the index value multiplied with the size of the pointer or array type as offset.

```
if (indexExpression.Symbol.Value is BigInteger) {
  int indexValue = (int) ((BigInteger) indexExpression.Symbol.Value),
      indexSize = arrayType.PointerOrArrayType.Size();
  return Dereference(arrayExpression, resultSymbol,
                     indexValue * indexSize);
}
```

If the index value is not constant, we begin by generating code for multiplying the index expression with the size of the pointer or array type.

```
else {
  indexExpression = MultiplySize(arrayExpression, indexExpression);

  List<MiddleCode> shortList = new List<MiddleCode>();
  shortList.AddRange(arrayExpression.ShortList);
  shortList.AddRange(indexExpression.ShortList);

  List<MiddleCode> longList = new List<MiddleCode>();
  longList.AddRange(arrayExpression.LongList);
  longList.AddRange(indexExpression.LongList);

  Symbol addSymbol = new Symbol(arrayType);
```

```
      AddMiddleCode(longList, MiddleOperator.Add,
                    addSymbol, arrayExpression.Symbol,
                    indexExpression.Symbol);

      Expression addExpression =
        new Expression(addSymbol, shortList, longList);
      return Dereference(addExpression, resultSymbol, 0);
    }
  }
```

4.5.1. Dot Expression

The dot expression is a bit more complicated than the arrow expression, depending on whether the address symbol is null.

```
public static Expression DotExpression(Expression expression,
                                      string memberName) {
  Symbol parentSymbol = expression.Symbol;
  Assert.Error(parentSymbol.Type.IsStructOrUnion(), expression,
               Message.Not_a_struct_or_union_in_dot_expression);
  Assert.Error(parentSymbol.Type.MemberMap != null, expression,
               Message.Member_access_of_uncomplete_struct_or_union);

  Symbol memberSymbol;
  Assert.Error(parentSymbol.Type.MemberMap.
               TryGetValue(memberName, out memberSymbol),
               memberName, Message.Unknown_member_in_dot_expression);
```

If the address symbol is not null, we create a new symbol with the same its address symbol, and the offset of the parent symbol and member.

```
  Symbol resultSymbol;
  if (parentSymbol.AddressSymbol != null) {
    string name = parentSymbol.Name + "." + memberSymbol.Name +
                  Symbol.SeparatorId + memberSymbol.Offset;
    resultSymbol = new Symbol(name, parentSymbol.ExternalLinkage,
                              parentSymbol.Storage, memberSymbol.Type,
                              parentSymbol.IsParameter());
    resultSymbol.UniqueName = parentSymbol.UniqueName;
    resultSymbol.AddressSymbol = parentSymbol.AddressSymbol;
    resultSymbol.AddressOffset = parentSymbol.AddressOffset;
    resultSymbol.Offset = parentSymbol.Offset + memberSymbol.Offset;
  }
```

If the address symbol is null, we create a new symbol with the same storage and the offset of the struct or union plus the member.

```
  else {
    resultSymbol = new Symbol(memberSymbol.Type);
    resultSymbol.Name = parentSymbol.Name + Symbol.SeparatorId +
                        memberName;
    resultSymbol.UniqueName = parentSymbol.UniqueName;
    resultSymbol.Storage = parentSymbol.Storage;
    resultSymbol.Offset = parentSymbol.Offset + memberSymbol.Offset;
  }

  return (new Expression(resultSymbol, expression.ShortList,
                                   expression.LongList));
    }
```

4.5.2. Function Call Expression

The **FunctionPreCall** method is called before the argument list is parsed. It checks that the expression is either a function or a pointer to a function, adds the parameter type list of the function to the type list stack, adds zero to the current offset stack, and adds the call header instruction to the short and long list of the expression.

MiddleCodeGenerator.cs

```
    public static void CallHeader(Expression expression) {
      Type type = expression.Symbol.Type;
      Assert.Error(type.IsFunction() || type.IsFunctionPointer(),
                   expression.Symbol, Message.Not_a_function);
      Type functionType = type.IsFunction() ? type : type.PointerType;
      TypeListStack.Push(functionType.TypeList);
      ParameterOffsetStack.Push(0);
      AddMiddleCode(expression.LongList, MiddleOperator.PreCall,
                    SymbolTable.CurrentTable.CurrentOffset);
    }
```

The **CallExpression** method is called after the argument list is parsed. It checks that the expression is either a function or a pointer to a function, adds the parameter type list of the function to the type list stack, adds zero to the current offset stack, and adds the call header instruction to the short and long list of the expression.

```
    public static Expression CallExpression(Expression functionExpression,
                                            List<Expression> argumentList){
      List<Type> typeList = TypeListStack.Pop();
      ParameterOffsetStack.Pop();
```

The reason we keep track of the current offset is that we in case of nested function calls need to allocate space for the previous arguments of the functions call outside the nested function. If the current offset is more than zero, we add it together with the size of the standard function top the total offset. If the current offset is zero, no argument has yet been passed, and we do not need to increase the total offset.

```
      int totalOffset = 0;
      foreach (int currentOffset in ParameterOffsetStack) {
        if (currentOffset > 0) {
          totalOffset += (SymbolTable.FunctionHeaderSize + currentOffset);
        }
      }

      Type functionType = functionExpression.Symbol.Type.IsPointer() ?
                          functionExpression.Symbol.Type.PointerType :
                          functionExpression.Symbol.Type;
```

We check that there are not too few parameters in the call. If the type list is null, the number of arguments does not matter. However, if the type list is not null, the number of arguments must be at least the number of parameters in the type list.

```
      Assert.Error((typeList == null) ||
                   (argumentList.Count >= typeList.Count),
                   functionExpression,
                   Message.Too_few_actual_parameters_in_function_call);
```

We also check that there are not too many parameters in the call. If the type list is null or if the callee function has a variadic parameter list, the number of arguments does not matter. There may be more arguments than parameters, we already know that the arguments are not fewer than the parameters. However, if the type list is not null and the function does not have a variadic parameter list, the number of arguments must equal the number of parameters.

```
Assert.Error(functionType.IsVariadic() || (typeList == null) ||
             (argumentList.Count == typeList.Count),
             functionExpression,
             Message.Too_many_parameters_in_function_call);

List<MiddleCode> longList = new List<MiddleCode>();
longList.AddRange(functionExpression.LongList);
```

When iterate through the, we keep track of the extra size. In variadic function calls we need to allocate memory for the extra parameters that is not included in the regular parameter list.

```
int count = 0, offset = SymbolTable.FunctionHeaderSize, extra = 0;
foreach (Expression argumentExpression in argumentList) {
  Type type;
  if ((typeList != null) && (count++ < typeList.Count)) {
    type = typeList[count];
  }
  else {
    type = ParameterType(argumentExpression.Symbol);
    extra += type.Size();
  }
```

We add the argument to the parameter list and update the current offset.

```
  Expression parameterExpression =
    TypeCast.ImplicitCast(argumentExpression, parameterType);
  longList.AddRange(parameterExpression.LongList);

  if (parameterType.IsStructOrUnion()) {
    AddMiddleCode(longList, MiddleOperator.ParameterInitSize,
                  SymbolTable.CurrentTable.CurrentOffset + totalOffset +
                  offset, parameterType, parameterExpression.Symbol);
  }

  AddMiddleCode(longList, MiddleOperator.Parameter,
                SymbolTable.CurrentTable. CurrentOffset + totalOffset +
                offset, parameterType, parameterExpression.Symbol);
  offset += parameterType.Size();
}
```

We add both the function call and post call instructions to the long list. The post call instruction is for

```
Symbol functionSymbol = functionExpression.Symbol;
AddMiddleCode(longList, MiddleOperator.Call,
              SymbolTable.CurrentTable.CurrentOffset + totalOffset,
              functionSymbol, extra);
AddMiddleCode(longList, MiddleOperator.PostCall,
              SymbolTable.CurrentTable.CurrentOffset + totalOffset);

Type returnType = functionType.ReturnType;
Symbol returnSymbol = new Symbol(returnType);
```

We make the short list be a copy of the long list. However, there is one difference between the lists if the function returns a floating value. In that case, we pop the value off the floating value stack in the short list, since the value is not interesting in the short list, only the side effect of the function call. However, we keep the value on the stack in the long list, since the value shall be used in the long list.

```
      List<MiddleCode> shortList = new List<MiddleCode>();
      shortList.AddRange(longList);

      if (!returnType.IsVoid()) {
        if (returnType.IsStructOrUnion()) {
          Type pointerType = new Type(returnType);
          Symbol addressSymbol = new Symbol(pointerType);
          returnSymbol.AddressSymbol = addressSymbol;
        }

        AddMiddleCode(longList, MiddleOperator.GetReturnValue,
                      returnSymbol);

        if (returnType.IsFloating()) {
          AddMiddleCode(shortList, MiddleOperator.PopEmpty);
        }
      }

      return (new Expression(returnSymbol, shortList, longList));
    }
```

4.5.3. Argument Expression List

The **ArgumentExpression** method type casts the argument into an appropriate type.

MiddleCodeGenerator.cs

```
    public static Expression ArgumentExpression(int index,
                                                Expression expression) {
```

We need to type list on top of the type list stack to compare the argument expression with the parameter type.

```
      List<Type> typeList = TypeListStack.Peek();
```

If the argument is within the parameter list, we type cast the argument into the parameter type.

```
      if ((typeList != null) && (index < typeList.Count)) {
        expression = TypeCast.ImplicitCast(expression, typeList[index]);
      }
      else {
```

If the argument is not within the parameter list (variadic call), we type cast the argument into an appropriate type: character and short integer are cast into (signed or unsigned) integer, and float is cast into double. These type casts apply to the variadic function call rules of ANSI C.

```
        Type type = expression.Symbol.Type;

        if (type.IsChar() || type.IsShort()) {
          if (type.IsSigned()) {
            expression =
              TypeCast.ImplicitCast(expression, Type.SignedIntegerType);
          }
          else {
```

```
          expression =
            TypeCast.ImplicitCast(expression, Type.UnsignedIntegerType);
        }
      }
      else if (type.IsFloat()) {
        expression = TypeCast.ImplicitCast(expression, Type.DoubleType);
      }
    }
```

We need to update current offset to allocate space in case of nested function calls.

```
    int offset = ParameterOffsetStack.Pop();
    ParameterOffsetStack.Push(offset +
                              ParameterType(expression.Symbol).Size());
    return (new Expression(expression.Symbol, expression.LongList,
                           expression.LongList));
  }
```

The **ParameterType** method returns the type of the parameter. In case of array, string, or function, it returns pointer types. Otherwise, it returns the regular type.

```
  private static Type ParameterType(Symbol symbol) {
    switch (symbol.Type.Sort) {
      case Sort.Array:
        return (new Type(symbol.Type.ArrayType));

      case Sort.Function:
        return (new Type(symbol.Type));

      case Sort.String:
        return (new Type(new Type(Sort.SignedChar)));

      case Sort.Logical:
        return Type.SignedIntegerType;

      default:
        return symbol.Type;
    }
  }
```

4.5.4. Primary Expressions

The value holds floating type, we push it at the floating value stack.

MiddleCodeGenerator.cs

```
  public static Expression ValueExpression(Symbol symbol) {
    List<MiddleCode> longList = new List<MiddleCode>();

    if (symbol.Type.IsFloating()) {
      AddMiddleCode(longList, MiddleOperator.PushFloat, symbol);
    }

    return (new Expression(symbol, new List<MiddleCode>(), longList));
  }
```

Given a name, we look it up in the current symbol table. If there is no symbol, we add a function without a parameter list, that returns a signed integer, to the symbol table.

MiddleCodeGenerator.cs
```
    public static Expression NameExpression(string name) {
      Symbol symbol = SymbolTable.CurrentTable.LookupSymbol(name);
      Assert.Error(symbol != null, name, Message.Unknown_name);

      List<MiddleCode> shortList = new List<MiddleCode>(),
                       longList = new List<MiddleCode>();

      if (symbol.Type.IsFloating()) {
        AddMiddleCode(shortList, MiddleOperator.PushFloat, symbol);
        AddMiddleCode(longList, MiddleOperator.PushFloat, symbol);
      }

      return (new Expression(symbol, shortList, longList));
    }
```

The registers are only used internally, in conjunction with system calls. On some occasion the interrupt call returns information stored in a register.

The closed statements do also include a set of internal statements; that is, statements not included in standard C, but necessary for the standard library functionality. The first statement is the load register statement, which stores a value in a register. The operands are the name of a register and an expression of integral or pointer type with same size as the register.

MiddleCodeGenerator.cs
```
    public static Expression RegisterExpression(Register register) {
      List<MiddleCode> longList = new List<MiddleCode>();
      int size = AssemblyCode.SizeOfRegister(register);
      Type type = TypeSize.SizeToUnsignedType(size);
      Symbol symbol = new Symbol(type);
      AddMiddleCode(longList, MiddleOperator.InspectRegister,
                    symbol, register);
      return (new Expression(symbol, new List<MiddleCode>(),
                             longList, register));
    }
```

MiddleCodeGenerator.cs
```
    public static Expression CarryFlagExpression() {
      ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),
                       falseSet = new HashSet<MiddleCode>();
      List<MiddleCode> longList = new List<MiddleCode>();
      trueSet.Add(AddMiddleCode(longList, MiddleOperator.Carry));
      falseSet.Add(AddMiddleCode(longList, MiddleOperator.Jump));
      Symbol symbol = new Symbol(trueSet, falseSet);
      return (new Expression(symbol, new List<MiddleCode>(), longList));
    }
```

MiddleCodeGenerator.cs
```
    public static Expression StackTopExpression() {
      List<MiddleCode> longList = new List<MiddleCode>();
      Symbol symbol = new Symbol(new Type(Type.UnsignedCharType));
      AddMiddleCode(longList, MiddleOperator.StackTop, symbol);
      return (new Expression(symbol, new List<MiddleCode>(), longList));

  }
```

5. Declaration Specifiers and Declarators

The **Storage** enumeration holds the storage specifiers of a symbol.

Storage.cs
```
namespace CCompiler {
  public enum Storage {Auto     = (int) Mask.Auto,
                       Register = (int) Mask.Register,
                       Static   = (int) Mask.Static,
                       Extern   = (int) Mask.Extern,
                       Typedef  = (int) Mask.Typedef};
}
```

A symbol definition is made up by a **specifier** and a **declarator**. The **Mask** class is used to distinguish between the different specifiers and is used by the **Specifier** class below.

Mask.cs
```
namespace CCompiler {
  public enum Mask {StorageMask = 0x0000FFF,
                    Auto        = 0x0000010,
                    Register    = 0x0000020,
                    Static      = 0x0000040,
                    Extern      = 0x0000080,
                    Typedef     = 0x0000100,

                    QualifierMask = 0x000F000,
                    Constant      = 0x0001000,
                    Volatile      = 0x0002000,

                    SortMask  = 0xFFF0000,
                    Signed    = 0x0010000,
                    Unsigned  = 0x0020000,
                    Char      = 0x0040000,
                    Short     = 0x0100000,
                    Int       = 0x0200000,
                    Long      = 0x0400000,
                    Float     = 0x0800000,
                    Double    = 0x1000000,
                    Void      = 0x2000000,
```

We use the bitwise **or** operator to mask simply masks to compounded masks.

```
                    SignedChar = Signed | Char,
                    UnsignedChar = Unsigned |  Char,
                    ShortInt = Short |  Int,
                    SignedShort = Signed |  Short,
                    SignedShortInt = Signed |  Short |  Int,
                    UnsignedShort = Unsigned |  Short,
                    UnsignedShortInt = Unsigned |  Short |  Int,
                    SignedInt = Signed |  Int,
                    UnsignedInt = Unsigned |  Int,
                    LongInt = Long |  Int,
                    SignedLong = Signed |  Long,
                    SignedLongInt = Signed |  Long |  Int,
                    UnsignedLong = Unsigned |  Long,
                    UnsignedLongInt = Unsigned |  Long |  Int,
                    LongDouble = Long |  Double};
}
```

The **Specifier** class is used to generate the type specified by a list of specifiers. The declaration specifiers are made up by keywords or struct, union, or enumeration declarations.

Specifier.cs

```
using System;
using System.Text;
using System.Collections.Generic;

namespace CCompiler {
  public class Specifier {
```

The specifier decides the storage and type of the symbol, it also decide whether the symbol has external linkage.

```
    private bool m_externalLinkage;
    private Storage? m_storage;
    private Type m_type;
```

The **m_maskToSortMap** map maps the masks of the **Mask** enumeration to the types of the **Sort** enumeration. However, the masks are represented by unsigned integer values since we use unsigned integer values to mask the keywords of the declaration specifiers in the **SpecifierList** method below.

```
    private static IDictionary<int, Sort> m_maskToSortMap =
      new Dictionary<int, Sort>() {
        {(int) Mask.Void, Sort.Void},
        {(int) Mask.Char, Sort.SignedChar},
        {(int) Mask.SignedChar, Sort.SignedChar},
        {(int) Mask.UnsignedChar, Sort.UnsignedChar},
        {(int) Mask.Short, Sort.SignedShortInt},
        {(int) Mask.ShortInt, Sort.SignedShortInt},
        {(int) Mask.SignedShort, Sort.SignedShortInt},
        {(int) Mask.SignedShortInt, Sort.SignedShortInt},
        {(int) Mask.UnsignedShort, Sort.UnsignedShortInt},
        {(int) Mask.UnsignedShortInt, Sort.UnsignedShortInt},
        {(int) Mask.Int, Sort.SignedInt},
        {(int) Mask.Signed, Sort.SignedInt},
        {(int) Mask.SignedInt, Sort.SignedInt},
        {(int) Mask.Unsigned, Sort.UnsignedInt},
        {(int) Mask.UnsignedInt, Sort.UnsignedInt},
        {(int) Mask.Long, Sort.SignedLongInt},
        {(int) Mask.LongInt, Sort.SignedLongInt},
        {(int) Mask.SignedLong, Sort.SignedLongInt},
        {(int) Mask.SignedLongInt, Sort.SignedLongInt},
        {(int) Mask.UnsignedLong, Sort.UnsignedLongInt},
        {(int) Mask.UnsignedLongInt, Sort.UnsignedLongInt},
        {(int) Mask.Float, Sort.Float},
        {(int) Mask.Double, Sort.Double},
        {(int) Mask.LongDouble, Sort.LongDouble}
      };
```

A specifier is made up by the external linkage, storage, and type.

```
    public Specifier(bool externalLinkage, Storage? storage, Type type) {
      m_externalLinkage = externalLinkage;
      m_storage = storage;
      m_type = type;
    }

    public bool ExternalLinkage {
      get { return m_externalLinkage; }
    }

    public Storage Storage {
      get { Assert.ErrorXXX(m_storage != null);
            return m_storage.Value; }
    }

    public Type Type {
      get { return m_type; }
    }
```

The static **SpecifierList** method takes a list of declaration specifiers and returns a **Specifier** object, holding the storage, type, and external linkage of the symbol.

```
    public static Specifier SpecifierList(List<object> specifierList) {
      int totalMaskValue = 0;
      Type compoundType = null;
```

We iterate through the declaration specifier list and gather the keywords of the **Mask** enumeration. If a keyword occurs twice, we report an error.

```
      foreach (object obj in specifierList) {
        if (obj is Mask) {
          int maskValue = (int) obj;

          if ((maskValue & totalMaskValue) != 0) {
            Assert.Error(MaskToString(maskValue),
                         Message.Keyword_defined_twice);
          }

          totalMaskValue |= maskValue;
        }
```

If the element of the list is not a **Mask** enumeration, it must be a type. If more than one type occurs in the list, we report an error.

```
        else {
          if (compoundType != null) {
            Assert.Error(MaskToString(totalMaskValue),
                         Message.Invalid_specifier_sequence);
          }

          compoundType = (Type) obj;
        }
      }
```

When we have the final mask, we begin extracting the storage. The storage is initially set to null, and we mask out the storage part of the total mask. It is possible that there was no storage in the specifier list. However, if there is a storage it must match one of the storage masks. If it does not match, more than one storage specifiers were present in the declaration specifier list, and we report an error.

```
Storage? storage = null;
{ int totalStorageValue = totalMaskValue & ((int) Mask.StorageMask);

  if (totalStorageValue != 0) {
    Assert.Error(Enum.IsDefined(typeof(Mask), totalStorageValue),
                 MaskToString(totalStorageValue),
                 Message.Invalid_specifier_sequence);
    storage = (Storage) totalStorageValue;
  }
}
```

The next step is to determine the external linkage. The symbol has externa linkage if the scope of the current symbol table is global (the symbol is defined in global space). Moreover, the storage shall be null (is has not been stated in the declaration specifier list) or extern. If the symbol is static in global scope it has no external linkage.

```
bool externalLinkage = (SymbolTable.CurrentTable.Scope == Scope.Global)
                       && ((storage == null) ||
                           (storage == CCompiler.Storage.Extern));

if (storage == null) {
  if (SymbolTable.CurrentTable.Scope == Scope.Global) {
    storage = Storage.Static;
  }
  else {
    storage = Storage.Auto;
  }
}
```

When the storage and external linkage has been taken care of, we perform a series of error checks. If the symbol is a function parameter (parameter scope) the storage must be auto or register.

```
if (SymbolTable.CurrentTable.Scope == Scope.Parameter) {
  Assert.Error((storage == Storage.Auto) ||
               (storage == Storage.Register), storage, Message.
      Only_auto_or_register_storage_allowed_in_parameter_declaration);
}
```

If the scope of the current symbol table is a struct or union, the storage must also be auto or register.

```
else if ((SymbolTable.CurrentTable.Scope == Scope.Struct) ||
         (SymbolTable.CurrentTable.Scope == Scope.Union)) {
    Assert.Error((storage == Storage.Auto) ||
                 (storage == Storage.Register), storage, Message.
      Only_auto_or_register_storage_allowed_for_struct_or_union_scope);
}
```

If the scope of the current symbol table is global, the storage must also be extern, static, or typedef.

```
else if (SymbolTable.CurrentTable.Scope == Scope.Global) {
    Assert.Error((storage == Storage.Extern) ||
                 (storage == Storage.Static) ||
                 (storage == Storage.Typedef), storage, Message.
  Only_extern____static____or_typedef_storage_allowed_in_global_scope);
}
```

When we have checked the storages, we continue to check whether there is a compound type and if it is an enumeration. In that case we iterate through the enumeration items and take action in accordance with the storage of the enumeration.

The reason we do this at this point rather than when the items are defined is that it is possible to state the storage are the enumeration definition. For instance: **enum Values {One, Two, Three} static;**.

```
      if ((compoundType != null) && (compoundType.EnumItemSet != null)){
```

If the enumeration has typedef storage, we need to change its type to signed integer, since enumeration value has that type. We also need to change the storage of the enumeration items. If the typedef definition occurs in global space, the enumeration item shall have static storage. If it occurs in local space, they shall have auto storage.

```
        if (storage == Storage.Typedef) {
          compoundType = Type.SignedIntegerType;
          storage = (SymbolTable.CurrentTable.Scope == Scope.Global)
                    ? Storage.Static : Storage.Auto;
        }

        foreach (Pair<Symbol,bool> pair in compoundType.EnumItemSet){
          Symbol itemSymbol = pair.First;
```

We set the storage of the item to the storage of the enumeration. At this point we are sure that the storage is not null, and that we can safely access its value.

```
          itemSymbol.Storage = storage.Value;
```

If the enumeration has static storage, we add the value of the item to the global static set.

```
          switch (itemSymbol.Storage) {
            case CCompiler.Storage.Static:
              SymbolTable.StaticSet.Add(ConstantExpression.
                                          Value(itemSymbol));
              break;
```

If the enumeration has auto or register storage, we set the offset of the item in the local symbol table.

```
            case CCompiler.Storage.Auto:
            case CCompiler.Storage.Register:
              SymbolTable.CurrentTable.SetOffset(itemSymbol);
              break;
```

If the enumeration has extern storage, it is not allowed to initialize its members with an explicit value.

```
            case CCompiler.Storage.Extern: {
                bool enumInitializer = pair.Second;
                Assert.Error(!enumInitializer,
                             itemSymbol + " = " + itemSymbol.Value,
                  Message.Extern_enumeration_item_cannot_be_initialized);
              }
              break;
          }
        }
      }
```

We check if the total mask holds the masks representing the constant or volatile keywords.

```
      { bool isConstant = (totalMaskValue & ((int) Mask.Constant)) != 0;
        bool isVolatile = (totalMaskValue & ((int) Mask.Volatile)) != 0;
```

If the type is constant or volatile (or both) and is a struct or union, we create a new compound type. The reason for this is that we do not want to mark the original compound type as constant or volatile since it may have a tag name.

```
if ((compoundType != null) && compoundType.IsStructOrUnion() &&
    (isConstant || isVolatile)) {
  compoundType = new Type(compoundType.Sort, compoundType.MemberMap,
                          compoundType.MemberList);
}
```

Finally, we extract the sort mask and test whether it represent a valid type. If the declaration specifier list does not represent a valid combination, we report an error. For instance, **signed short double** is an invalid combination of declaration specifiers.

```
Sort? sort = null;
int sortMaskValue = totalMaskValue & ((int) Mask.SortMask);

if (sortMaskValue != 0) {
  if (!m_maskToSortMap.ContainsKey(sortMaskValue)) {
    Assert.Error(MaskToString(sortMaskValue),
                 Message.Invalid_specifier_sequence);
  }

  sort = m_maskToSortMap[sortMaskValue];
}
```

When defining the final type, there are four cases. If the compound type is not null and the sort is null, the result type is the compound type, with the possible addition of constant or volatile qualifiers.

```
Type type = null;
if ((compoundType != null) && (sort == null)) {
  compoundType.Constant = (compoundType.Constant || isConstant);
  compoundType.Volatile = (compoundType.Volatile || isVolatile);
  type = compoundType;
}
```

If the compound type is null and the sort is not null, the result type is based on the sort.

```
else if ((compoundType == null) && (sort != null)) {
  type = new Type(sort.Value);
  type.Constant = isConstant;
  type.Volatile = isVolatile;
}
```

If the compound type and sort are both null, there is no type. This is valid in function definitions, in which case the type shall be a signed integer.

```
else if ((compoundType == null) && (sort == null)) {
  type = new Type(Sort.SignedInt);
  type.Constant = isConstant;
  type.Volatile = isVolatile;
}
```

If both the compound type and the sort are not null, the type specification is invalid. For instance, **unsigned short struct {int i;}** is an invalid combination.

```
else {
  Assert.Error(MaskToString((int) sortMaskValue), Message.
             Invalid_specifier_sequence_together_with_type);
}
```

Finally, the specifier with the external linkage, storage, and type is returned.

```
return (new Specifier(externalLinkage, storage, type));
```

```
      }
    }
```

The **QualifierList** method is called when a pointer is declared. It exams whether the pointer is constant or volatile.

```
    public static Type QualifierList(List<Mask> qualifierList) {
      int totalMaskValue = 0;

      foreach (Mask mask in qualifierList) {
        int maskValue = (int) mask;

        if ((maskValue & totalMaskValue) != 0) {
          Assert.Error(MaskToString(maskValue),
                       Message.Keyword_defined_twice);
        }

        totalMaskValue |= maskValue;
      }

      Type type = Type.VoidPointerType;
      type.Constant = (totalMaskValue & ((int) Mask.Constant)) != 0;
      type.Volatile = (totalMaskValue & ((int) Mask.Volatile)) != 0;
      return type;
    }
```

The **MaskToString** method is called when reporting an error. It returns a string holding the declaration specifiers as text.

```
    private static string MaskToString(int totalMaskValue) {
      StringBuilder maskBuffer = new StringBuilder();
```

We iterate through the bits of the mask and add the text of the **Mask** enumeration matching each true bit (bit with value one).

```
      for (int maskValue = 1; maskValue != 0; maskValue <<= 1) {
        if ((maskValue & totalMaskValue) != 0) {
          string maskName = Enum.GetName(typeof(Mask), maskValue).ToLower();
          maskBuffer.Append(((maskBuffer.Length > 0) ? " " : "") + maskName);
        }
      }

      return maskBuffer.ToString();
    }
  }
}
```

5.1. Declarators

A declarator follows the declaration specifiers, it can be a simple variable, a pointer, an array, or an old-style or new-style function. A declarator can be initialized with a value or marked with its size in bits. The bold part of the following code are examples of declarators. Only members of a struct can be marked with bits. Struct or union members, as well as function parameters, cannot be initialized.

```
int i, j = 2;
double x = 3.14;
int f(a, b, c), g(int i, double x);
char s[] = "Hello";
```

```
int *p, *q = &i;
long int value : 7; // Struct members only
```

The **Declarator** class handles a declarator. For each new element of the declarator, the **Add** method is called. For instance, if the declaration specifier is an integer and the declarator is a pointer **int *p;**, **Add** is called with a pointer first, and then with an integer. The second call attaches the integer to the pointer, so that the pointer points at the integer. In the same way, the array type becomes attached to an array and the return type becomes attached to a function.

```
namespace CCompiler {
  public class Declarator {
    private string m_name;
    private Type m_firstType, m_lastType;

    public Declarator(string name) {
      m_name = name;
    }

    public string Name {
      get { return m_name; }
      set { m_name = value; }
    }

    public Type Type {
      get { return m_firstType; }
    }
```

If we add a type to the declarator where there is no previous type, we set both the first and last type to that type.

```
    public void Add(Type newType) {
      if (m_firstType == null) {
        m_firstType = m_lastType = newType;
      }
      else {
        switch (m_lastType.Sort) {
```

If the last type of the declarator is a pointer, we set it to point at the new type.

```
          case Sort.Pointer:
            m_lastType.PointerType = newType;
            m_lastType = newType;
            break;
```

If the last type of the declarator is an array, we set the type of the array to be the new type. However, the array must be complete; that is, it must have a stated size. Moreover, the new type cannot be a function since arrays of functions are not allowed.

```
          case Sort.Array:
            Assert.Error(newType.IsComplete(),
                         Message.Array_of_incomplete_type_not_allowed);
            Assert.Error(!newType.IsFunction(),
                         Message.Array_of_function_not_allowed);
            m_lastType.ArrayType = newType;
            m_lastType = newType;
            break;
```

If the last type of the declarator is a function, set the return type of the function to be the new type. However, the new type cannot be an array or a function since functions are not allowed to return an arrays or functions.

```
          case Sort.Function:
            Assert.Error(!newType.IsArray(),
                         Message.Function_cannot_return_array);
            Assert.Error(!newType.IsFunction(),
                         Message.Function_cannot_return_function);
            m_lastType.ReturnType = newType;
            m_lastType = newType;
            break;
        }
      }
    }
  }
}
```

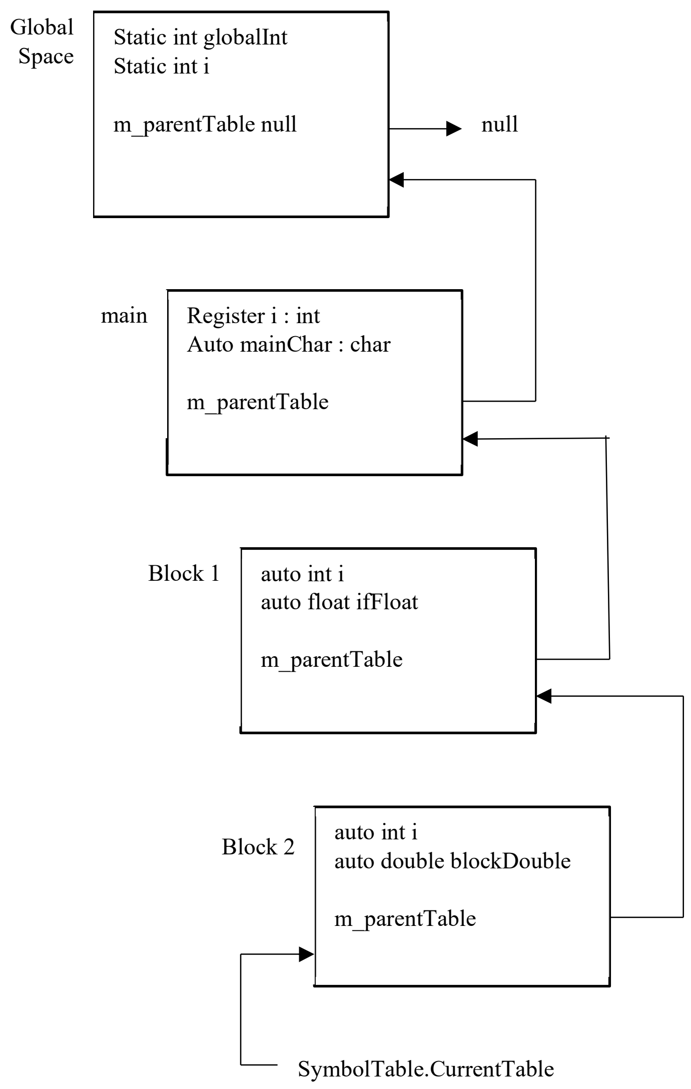
6. The Symbol Table

The symbol table keeps track of the values, types, functions, and variables of the code, defined by the programmer and well as temporary variables introduced by the compiler. It also holds struct and union tags (enumerations do not have tags). In fact, there are actually several symbol tables. There are symbol tables for the global space, for each function, for each block in the functions, and for the members of each struct or union. In this way, the symbol tables form a hierarchy where each table holds a reference to its parent table (except the table for the global space, which table parent reference is null).

For example, let us look at the following code.

```
int globalInt;
int i;

void main() {
  register int i;
  char mainChar;

  if (globalCount > 0) { // Block 1
    int i;
    float ifFloat;

    { int i; // Block 2
      double blockDouble;
      // Point p
    }
  }
}
```

When the parsing reaches point **p**, the symbol table has the form below. Note that the variable **i** defined in every block and is present in every table. When a symbol is looked up, we start searching in the current table and continue to search up through the hierarchy until we found a symbol with the given name or the parent symbol reference is null, in which case the table representing the global space.



The **SymbolTable** class holds a symbol table. The scope of a table is **Global**, **Function**, **Block**, **Struct**, or **Union** as of the **Scope** enumeration.

SymbolTable.cs
```
using System;
using System.Collections.Generic;

namespace CCompiler {
  public class SymbolTable {
    private Scope m_scope;
```

For each function call, an activation record is allocated at the call stack. The first three entries are the return jump address (the address of the instruction following the function call), the regular frame pointer (the address of the current activation record), and the variadic frame pointer (the regular frame pointer plus the size of potential extra parameters in a variadic call). The function header size is the sum of the size of those three entries. As the compiler in this book can be set to generate code for different target machines, the offsets are also given different values as the pointer size varies. See chapter 12 for more information about the frame pointers.

```
public static int ReturnAddressOffset = 0;
public static int RegularFrameOffset = TypeSize.PointerSize;
public static int VariadicFrameOffset = 2 * TypeSize.PointerSize;
public static int FunctionHeaderSize = 3 * TypeSize.PointerSize;
```

Each table holds a reference to its parent table, except the table of global space, which table parent reference is null.

```
private SymbolTable m_parentTable;
```

The activation record is organized in the following way:

- The offsets of the return jump address, the regular frame pointer, and the variadic frame pointer.
- The regular function parameters.
- The extra parameters in case of a variadic call.
- Variables of auto or register storage.
- Temporary integral values are stored are loaded from the registers and stored to the activation record during function calls.
- Temporary floating-point values are loaded from the floating-point stack and to the activation record during function calls.

```
private int m_currentOffset;
```

All parameters and variables of a function or a block, as well as the members of a struct or a union, are stored in the **m_entryMap** and **m_entryList**. We use **m_entryMap** to look up symbols and **m_entryList** when initializing values in a struct or union, in which case the symbols need to be stored in the order they were declared in the code. The standard C# **Dictionary** class provides fast searching but does not guarantee any order among its key-value pairs.

```
private IDictionary<string,Symbol> m_entryMap =
  new Dictionary<string,Symbol>();
private List<Symbol> m_entryList = new List<Symbol>();
```

The tag map holds the types of the structs or unions that are marked with a tag.

```
private IDictionary<string,Type> m_tagMap = new Dictionary<string,Type>();
```

CurrentTable and **CurrentFunction** hold references to the current symbol table and the function currently processed by the parser. We need the current symbol table when adding or looking up symbols, and the current function when generating code for calling a function or returning a value. In global scope, the **CurrentFunction** is null.

```
public static SymbolTable CurrentTable = null;
public static Symbol CurrentFunction = null;
```

GlobalStaticSet holds all static symbols. Only symbols of auto or register storage are given offsets. Symbols of static, extern, or typedef storage are not given offsets, since they are not stored at the activation record in runtime.

```
public static ISet<StaticSymbol> GlobalStaticSet;
public static StaticSymbolLinux InitSymbol, ArgsSymbol;
```

The constructor of the **SymbolTable** class takes the parent table (which is null for global space) and the scope of the table.

```
    public SymbolTable(SymbolTable parentTable, Scope scope) {
      m_parentTable = parentTable;
```

In case of global scope, we create the global static set. There is only one table of global scope and one static set in each source code file. In this case we do not need to set the offset field since there are no variables of auto or register storage in global scope.

```
      switch (m_scope = scope) {
        case Scope.Global:
          GlobalStaticSet = new HashSet<StaticSymbol>();
          InitSymbol = ArgsSymbol = null;
          break;
```

In case of struct of union scope, the current offset is initialized to zero.

```
        case Scope.Struct:
        case Scope.Union:
          m_currentOffset = 0;
          break;
```

In case of function scope, the current offset is initialized to the size of the header size. The return address, regular frame pointer, and variadic frame pointer holds the first entries, and the parameters are given the offsets following the initialization.

```
        case Scope.Function:
          m_currentOffset = FunctionHeaderSize;
          break;
```

In case of block scope, the current offset is initialized to the current offset of its parent table. When the parsing leaves a block the parent table again becomes the current table, and a following block is again initialized to the current offset of the parent table. In this way, symbols of different blocks can share the same space on the activation record.

```
        case Scope.Block:
          m_currentOffset = m_parentTable.m_currentOffset;
          break;
      }
    }

    public Scope Scope {
      get { return m_scope; }
    }

    public SymbolTable ParentTable {
      get { return m_parentTable; }
    }

    public IDictionary<string,Symbol> EntryMap {
      get { return m_entryMap; }
    }

    public List<Symbol> EntryList {
      get { return m_entryList; }
    }

    public int CurrentOffset {
      get { return m_currentOffset; }
    }
```

The **AddSymbol** method adds a symbol to the symbol table. It is possible to add two symbols with the same name if they have equal types and at least one of them holds extern storage. When adding a symbol with a name we have to check a few features. If the symbol is extern, it cannot be initialized (its value must be null).

```
  public void AddSymbol(Symbol newSymbol) {
    string name = newSymbol.Name;
```

The name of the symbol may be null in case of unnamed function parameters.

```
    if (name != null) {
      Symbol oldSymbol;
```

We look up the entry list, and if we find another symbol with the same name, we check whether at least one of the symbols is extern and they have equals types. Otherwise, we have two non-extern symbols with the same name, and we report an error.

```
      if (m_entryMap.TryGetValue(name, out oldSymbol)) {
        Assert.Error(oldSymbol.IsExtern() || newSymbol.IsExtern(),
                     name, Message.Name_already_defined);
        Assert.Error(oldSymbol.Type.Equals(newSymbol.Type),
                     name, Message.Different_types_in_redeclaration);
```

The new symbol is given the same unique name as the old symbol, in order for the linker to find the symbol. Since the symbols are referred to by the same name, they also need to have the same unique name,

```
        newSymbol.UniqueName = oldSymbol.UniqueName;
```

If the new symbol is not extern, we remove the old symbol and replace it with the new symbol. We do not need to modify **m_entryList**, since we use int in struct and unions only, and struct or union members cannot be extern.

```
        if (!newSymbol.IsExtern()) {
          m_entryMap[name] = newSymbol;
        }
      }
```

If there is no previous symbol with the same name, we just add the new symbol to the entry map and entry list.

```
      else {
        m_entryMap[name] = newSymbol;
        m_entryList.Add(newSymbol);
      }
    }
```

If the variable is auto or register and not a function, it shall be given an offset in the activation record. Unless in union scope, in which each member always has offset zero.

```
    if (!newSymbol.Type.IsFunction()) {
      if (newSymbol.IsAutoOrRegister()) {
        if (m_scope == Scope.Union) {
          newSymbol.Offset = 0;
        }
```

If the new symbol is an enumeration constant item, is shall not be given an offset. This is because we do not know at this point if the item holds auto or register scope. An item is given auto storage from the beginning, but that may change when the whole enumeration is given its storage, which is done after the item is stored in the symbol table.

```
          else if (!newSymbol.Type.IsEnumerator()) {
            newSymbol.Offset = m_currentOffset;
            m_currentOffset += newSymbol.Type.Size();
          }
        }
      }
    }
```

The **SetOffset** method sets the offset of a symbol. It is called when an enumeration constant item has finally been given auto or register storage.

```
    public void SetOffset(Symbol symbol) {
      symbol.Offset = m_currentOffset;
      m_currentOffset += symbol.Type.Size();
    }
```

The **LookupSymbol** looks up the symbol with the given in the current symbol table and its parent tables, recursively. If no symbol with the given name is found, null is returned. Note that this method searches the parent tables while **AddSymbol** only checks for symbols with the same name in the current table.

```
    public Symbol LookupSymbol(string name) {
      Symbol symbol;

      if (m_entryMap.TryGetValue(name, out symbol)) {
        return symbol;
      }
      else if (m_parentTable != null) {
        return m_parentTable.LookupSymbol(name);
      }

      return null;
    }
```

The **AddTag** method adds a struct or union tag to the current symbol table. It is possible to add a new tag with an already taken name. However, in that case, the old and the new tag must have the same sort (they must both be structs or both be unions) and the member map of at least one of them must be null. This order is necessary to provide linked lists in C.

```
    public void AddTag(string name, Type newType) {
      if (m_tagMap.ContainsKey(name)) {
        Type oldType = m_tagMap[name];
        Assert.Error(!oldType.IsEnumerator() &&
                     (oldType.Sort == newType.Sort), name,
                     Message.Name_already_defined);
```

If the member map of the old tag is null, we assign it the member map of the new tag.

```
        if (oldType.MemberMap == null) {
          oldType.MemberMap = newType.MemberMap;
          oldType.MemberList = newType.MemberList;
        }
```

If the neither the old nor the new member map is null, we report an error since it is not allowed to have two structs or unions with the same name tags with non-null member maps.

```
        else {
          Assert.Error(newType.MemberMap == null, name,
                       Message.Name_already_defined);
        }
```

```
      }
```

If there is no struct or union tag previous added to the tag map, we simply add the new tag to the map.

```
      else {
        m_tagMap.Add(name, newType);
      }
    }
```

The **LookUpTag** method looks up a tag in the current table or its parent tables. If the tag is not found, null is returned. Similar to **LookUpSymbol**, **LookUpTag** searches the parent tables.

```
    public Type LookupTag(string name, Sort sort) {
      Type type;

      if (m_tagMap.TryGetValue(name, out type) && (type.Sort == sort)) {
        return type;
      }
      else if (m_parentTable != null) {
        return m_parentTable.LookupTag(name, sort);
      }

      return null;
    }
  }
}
```

6.1. The Symbol

The symbol table is made up of symbols, which have a storage, type, and potential value. The **Symbol** class describes a variable, a value, a function, or a type specified with **typedef**.

Symbol.cs

```
using System;
using System.IO;
using System.Text;
using System.Numerics;
using System.Globalization;
using System.Collections.Generic;

namespace CCompiler {
  public class Symbol {
    public const string NumberId = "#";
    public const string TemporaryId = "£";
    public const string SeparatorId = "$";
    public const string SeparatorDot = ".";
    public const string FileMarker = "@";
```

A symbol may have external linkage; that is, is visible from other source code files. A function may have be a definition (with a body of code) or be a declaration (with only the name, return type, and parameter list).

```
    private bool m_externalLinkage;
```

A symbol actually has two names: the regular name it has been defined with and is looked up by the symbol table, and a unique name to distinguish between two symbols with the same name in different scopes. For symbols with external linkage, the regular and unique names are the same. For symbols without external linkage, the unique name is really unique. The idea is that the no two names without external linkage in any file shall have the same name. For instance, two static symbols in two different functions, or two global symbols in two different files, shall have different unique names.

```
    private string m_name, m_uniqueName;
```

A symbol may be a parameter in a function definition.

```
    private bool m_parameter;
```

A symbol has a storage and a type, it may also have a value.

```
    private Storage m_storage;
    private Type m_type;
    private object m_value;
```

Symbols with auto and register storage are given an offset on the current activation record.

```
    private int m_offset;
```

When dereferencing a symbol with the dereference, arrow, and index (with a constant index value) expression, the dereferenced symbol is stored in the address symbol, with an offset. The offset is zero in the dereference case, the offset of the field in the arrow case, and the constant index value times the size of the array type in the index case. In case of the index operator with a non-constant index value, the value of the address offset is set to zero and is calculated in run-time.

```
    private Symbol m_addressSymbol;
    private int m_addressOffset;
```

A symbol may hold logical type, in which case the true-set and false-set are stored in the symbol.

```
    private ISet<MiddleCode> m_trueSet, m_falseSet;
```

Finally, we have static counters for unique names and temporary names.

```
    private static int UniqueNameCount = 0, TemporaryNameCount = 0;
```

The first constructor is used for defined symbols, variables, constants, and functions. In case of enumeration constants, they also hold a value.

```
    public Symbol(string name, bool externalLinkage, Storage storage,
                  Type type, bool parameter = false, object value = null) {
      m_name = name;
      m_externalLinkage = externalLinkage;
      m_storage = storage;
```

If the symbol has external linkage its unique name is its regular name.

```
      if (m_externalLinkage) {
        m_uniqueName = m_name;
      }
```

If the symbol does not have external linkage, its unique name is a file marker, a unique number, and the regular name. The file marker will, by the linker, be replaced by a number unique for the object file. In this way, every symbol of the same file will have a unique number and each file will also have a unique number.

```
      else {
        m_uniqueName = Symbol.FileMarker + (UniqueNameCount++) +
                       Symbol.SeparatorId + m_name;
      }

      m_parameter = parameter;
      m_value = CheckValue(m_type = type, value);
```

```
  }
```

If the value of the symbol is a **BigInteger** value, we check that it does not exceed its limits, which is defined for each integral type.

```
  private static object CheckValue(Type type, object value) {
    if (value is BigInteger) {
      BigInteger bigValue = (BigInteger) value;

      if (type.IsUnsigned() && (bigValue < 0)) {
        bigValue += TypeSize.GetMaxValue(type.Sort) + 1;
      }

      Assert.Error((bigValue >= TypeSize.GetMinValue(type.Sort)) &&
                   (bigValue <= TypeSize.GetMaxValue(type.Sort)),
                   type + ": " + value, Message.Value_overflow);
      return bigValue;
    }

    return value;
  }
```

In many expressions, the resulting value is a temporary symbol. In case of a dereference, arrow, dot, or index expression, the resulting symbol may be assignable and addressable.

```
  public Symbol(Type type) {
    m_name = Symbol.TemporaryId + "temporary" + (TemporaryNameCount++);
    m_externalLinkage = false;
    m_storage = Storage.Auto;
    m_type = type;
    m_parameter = false;
  }
```

The resulting symbol of a logical expression.

```
  public Symbol(ISet<MiddleCode> trueSet, ISet<MiddleCode> falseSet) {
    m_name = Symbol.TemporaryId + "logical" + (TemporaryNameCount++);
    m_storage = Storage.Auto;
    m_type = new Type(Sort.Logical);
    m_trueSet = (trueSet != null) ? trueSet : (new HashSet<MiddleCode>());
    m_falseSet = (falseSet != null) ? falseSet
                                    : (new HashSet<MiddleCode>());
    m_parameter = false;
  }
```

We need to store a constant value as the denominator in an integral multiplication and division, and when pushing an integral or floating value to the floating value stack.

```
  public Symbol(Type type, object value) {
    Assert.ErrorXXX(!(value is bool));
    m_name = ValueName(type, value);
    m_uniqueName = Symbol.FileMarker + (UniqueNameCount++) +
                   Symbol.SeparatorId + m_name;
    m_storage = Storage.Static;
    m_parameter = false;
    m_value = CheckValue(m_type = type, value);
  }
```

The value is given a name the reflects its type and its actual value.

```
    public static string ValueName(CCompiler.Type type, object value) {
      Assert.ErrorXXX(value != null);
```

If the value is a string, its name becomes the text with each character that is not a letter or digit replaced by its hex code.

```
      if (value is string) {
        string text = (string) value;
        StringBuilder buffer = new StringBuilder();

        for (int index = 0; index < text.Length; ++index) {
          if (char.IsLetterOrDigit(text[index]) ||
              (text[index] == '_')) {
            buffer.Append(text[index]);
          }
          else if (text[index] != '\0') {
            int asciiValue = (int) text[index];
            char hex1 = "0123456789ABCDEF"[asciiValue / 16],
                 hex2 = "0123456789ABCDEF"[asciiValue % 16];
            buffer.Append(hex1.ToString() + hex2.ToString());
          }
        }

        return "string_" + buffer.ToString() + Symbol.NumberId;
      }
```

If the value is a static address, the name is the text “staticaddress” with the unique name and offset.

```
      else if (value is StaticAddress) {
        StaticAddress staticAddress = (StaticAddress) value;
        return "staticaddress" + Symbol.SeparatorId + staticAddress.UniqueName
               + Symbol.SeparatorId + staticAddress.Offset + Symbol.NumberId;
      }
```

An array, floating, or integral value is given the name of the values’s string with the minus sign replaced by the text “minus”.

```
      else if (type.IsArray()) {
        return "Array_" + value.ToString() + Symbol.NumberId;
      }
      else if (type.IsFloating()) {
        return "float" + type.Size().ToString() + Symbol.SeparatorId +
               value.ToString().Replace("-", "minus") + Symbol.NumberId;
      }
      else {
        return "int" + type.Size().ToString() + Symbol.SeparatorId +
               value.ToString().Replace("-", "minus") + Symbol.NumberId;
      }
    }

    public string Name {
      get { return m_name; }
      set { m_name = value; }
    }

    public string UniqueName {
      get { return m_uniqueName; }
      set { m_uniqueName = value; }
```

```
}

public bool ExternalLinkage {
  get { return m_externalLinkage; }
}

public Storage Storage {
  get { return m_storage; }
  set { m_storage = value; }
}

public Type Type {
  get { return m_type; }
  set { m_type = value; }
}

public int Offset {
  get { return m_offset; }
  set { m_offset = value; }
}

public bool IsExtern() {
  return (m_storage == Storage.Extern);
}

public bool IsStatic() {
  return (m_storage == Storage.Static);
}

public bool IsExternOrStaticArray() {
  return IsExternOrStatic() && m_type.IsArray();
}

public bool IsExternOrStatic() {
  return IsExtern() || IsStatic();
}

public bool IsTypedef() {
  return (m_storage == Storage.Typedef);
}

public bool IsAuto() {
  return (m_storage == Storage.Auto);
}

public bool IsRegister() {
  return (m_storage == Storage.Register);
}

public bool IsAutoOrRegister() {
  return IsAuto() || IsRegister();
}

public ISet<MiddleCode> TrueSet {
  get { return m_trueSet; }
  set { m_trueSet = value; }
}
```

```
    public ISet<MiddleCode> FalseSet {
      get { return m_falseSet; }
      set { m_falseSet = value; }
    }

    public bool Parameter {
      get { return m_parameter; }
    }

    public object Value {
      get { return m_value; }
      set { m_value = value; }
    }

    public Symbol AddressSymbol {
      get { return m_addressSymbol; }
      set { m_addressSymbol = value; }
    }

    public int AddressOffset {
      get { return m_addressOffset; }
      set { m_addressOffset = value; }
    }
```

A symbol is a value if its name is not null (it may be null if it is parameter) and contains the number identifier.

```
    public bool IsValue() {
      return (m_name != null) && m_name.Contains(NumberId);
    }
```

A symbol is temporary if its name is not null (it may be null if it is parameter) and contains the temporary identifier, and its address symbol is null.

```
    public bool IsTemporary() {
      return (m_name != null) && m_name.Contains(TemporaryId) &&
             (m_addressSymbol == null);
    }
```

The symbol is assignable if it is not a value, its type not recursively constant, and is not an array, function, or string. A type is recursively constant if it is constant or, in case of a struct or union, any of its members is recursively constant.

```
    public bool IsAssignable() {
      return !IsValue() && !m_type.IsConstantRecursive() &&
             !m_type.IsArrayFunctionOrString();
    }

    public override string ToString() {
      if (m_value != null) {
        return m_value.ToString();
      }
      else if (m_name != null) {
        if (m_addressSymbol != null) {
          return m_name + " -> " + m_addressSymbol.ToString();
        }
        else {
          return m_name;
        }
      }
      else {
        if (m_addressSymbol != null) {
          return m_addressSymbol.ToString();
        }
        else {
          return "";
        }
      }
    }

    public static string SimpleName(string name) {
      int index = name.LastIndexOf(Symbol.SeparatorId);
      return (index != -1) ? name.Substring(index + 1).Replace("#", "")
                           : name;
    }
  }
}
```

6.2. The Static Symbol

The **StaticSymbol** class is the base class of **StaticSymbolLinux** and **StaticSymbolWindows**.

StaticSymbol.cs

```
using System.IO;

namespace CCompiler {
  public abstract class StaticSymbol {
```

The class holds a unique name.

```
    private string m_uniqueName;

    public StaticSymbol() {
      // Empty.
    }

    public StaticSymbol(string uniqueName) {
      m_uniqueName = uniqueName;
    }

    public string UniqueName {
      get { return m_uniqueName; }
    }
```

Two static symbols are considered equal if they share the same unique name. This may happen if they hold values, in which case only the first of them shall be added to the global static set.

```
    public override bool Equals(object obj) {
      if (obj is StaticSymbol) {
        return m_uniqueName.Equals(((StaticSymbol) obj).m_uniqueName);
      }

      return false;
    }
    public override int GetHashCode() {
      return m_uniqueName.GetHashCode();
    }
```

The **Write** and **Read** methods as intended to be overridden by sub classes. In this class, they only write and read the unique name.

```
    public virtual void Write(BinaryWriter outStream) {
      outStream.Write(m_uniqueName);
    }

    public virtual void Read(BinaryReader inStream) {
      m_uniqueName = inStream.ReadString();
    }
  }
}
```

The **StaticSymbolLinux** class is a sub class of **StaticSymbol**. The **m_textList** field hold the assembly code instructions, and **m_externSet** holds the extern references of the assembly code.

StaticSymbolLinux.cs

```
using System;
using System.IO;
using System.Collections.Generic;

namespace CCompiler {
  public class StaticSymbolLinux : StaticSymbol {
    private List<String> m_textList;
    private ISet<string> m_externSet;
```

The constructor calls the base class constructor with the unique name.

```
    public StaticSymbolLinux(string uniqueName, List<string> textList,
                             ISet<string> externSet)
     :base(uniqueName) {
      m_textList = textList;
      m_externSet = externSet;
    }

    public List<string> TextList {
      get { return m_textList; }
    }

    public ISet<string> ExternSet {
      get { return m_externSet; }
    }
  }
}
```

There is also the **StaticSymbolWindows** class for the Windows environment. We will look into it in Chapter 13.

StaticSymbolWindows.cs

```
namespace CCompiler {
  public class StaticSymbolWindows : StaticSymbol {
    // ...
  }
}
```

7. The Type System

C has a rather large set of types. Values of enumeration types (**enum**) are stored as signed integer. A type can be constant or volatile. The **Sort** enumeration holds the simple and compound types of C. The **String** and **Logical** are present, even though they are not types in C. However, temporary vale may hold these types. We also have the **void** type, which technically is not a type, but rather mark the absence of a type.

Sort.cs
```
namespace CCompiler {
  public enum Sort {Void, SignedChar, UnsignedChar, SignedShortInt,
                    UnsignedShortInt, SignedInt, UnsignedInt,
                    SignedLongInt, UnsignedLongInt, Float, Double,
                    LongDouble, String, Pointer, Array, Struct, Union,
                    Function, Logical};
}
```

7.1. The Type Class

The **Type** class holds methods for create types with different features. A pointer has a pointer type, an array has an array size and type, a function has a return type and parameter list, a struct or union has a member map, and a struct or union member may have a bitfield.

Type.cs
```
using System;
using System.Linq;
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  public class Type {
    private Sort m_sort;
```

Each type holds a sort, it may be an integral sort (signed or unsigned character, short integer, integer, or long integer) a floating sort (float, double, or long double), a pointer or an array, a struct or a union, or a function.

```
    public Sort Sort {
      get { return m_sort; }
    }
```

The **IsSigned** method return true if the type a signed character, short integer, integer, or long integer.

```
    public static bool IsSigned(Sort sort) {
      return (sort == Sort.SignedChar) || (sort == Sort.SignedShortInt) ||
             (sort == Sort.SignedInt) || (sort == Sort.SignedLongInt);
    }

    public bool IsVoid() {
      return (m_sort == Sort.Void);
    }

    public bool IsChar() {
      return (m_sort == Sort.SignedChar) || (m_sort == Sort.UnsignedChar);
    }
```

```
public bool IsShort() {
  return (m_sort == Sort.SignedShortInt) ||
         (m_sort == Sort.UnsignedShortInt);
}

public bool IsInteger() {
  return (m_sort == Sort.SignedInt) || (m_sort == Sort.UnsignedInt);
}

public bool IsIntegral() {
  return IsSigned() || IsUnsigned();
}

public bool IsSigned() {
  switch (m_sort) {
    case Sort.SignedChar:
    case Sort.SignedShortInt:
    case Sort.SignedInt:
    case Sort.SignedLongInt:
      return true;

    default:
      return false;
  }
}

public bool IsUnsigned() {
  switch (m_sort) {
    case Sort.UnsignedChar:
    case Sort.UnsignedShortInt:
    case Sort.UnsignedInt:
    case Sort.UnsignedLongInt:
      return true;

    default:
      return false;
  }
}

public bool IsFloat() {
  return (m_sort == Sort.Float);
}

public bool IsFloating() {
  switch (m_sort) {
    case Sort.Float:
    case Sort.Double:
    case Sort.LongDouble:
      return true;

    default:
      return false;
  }
}

public bool IsArithmetic() {
  return IsIntegral() || IsFloating();
}

public bool IsString() {
  return (m_sort == Sort.String);
}
```

The following constructor takes an integral or arithmetic type, or void.

```
  public Type(Sort sort) { // arithmetic or logical
    m_sort = sort;
  }
```

Contrary to some other languages, there is no logical type in C. However, as C applies **lazy evaluation**, we need a logical type. Lazy evaluation means that an expression shall be not be evaluated more than necessary to determine its value. More specifically, if the left operand of the logical **or** operator is true the right operand shall not be evaluated. In the same way, if the left operand of the logical **and** operator is false the right operand shall not be evaluated. The same goes for the conditional operator, depending on whether the first expression is true or false, only the second or the third expression becomes evaluated. The logical type holds the two sets **m_trueSet** and **m_falseSet**, holding the middle code addresses to jumps instructions that will later be filled with the addresses to jump to if the expression is true or false.

```
  public bool IsLogical() {
    return (m_sort == Sort.Logical);
  }
```

7.1.1. Bitfields

In case of a bitfield member in a struct or union, we store its bitfield mask. The mask is technically two to the power of the bits minus one or. Informally, it is the bits number of ones, counting from the least significant bit. The bitfield type is an integral type, with the addition of the bitfield mask. The mask is used to set the unused bits to zero when a bitfield variable is assigned a value.

```
  private BigInteger? m_bitfieldMask = null;

  public void SetBitfieldMask(int bits) {
    m_bitfieldMask = (BigInteger) (Math.Pow(2, bits) - 1);
  }

  public bool IsBitfield() {
    return (m_bitfieldMask != null);
  }

  public BigInteger? GetBitfieldMask() {
    return m_bitfieldMask;
  }
```

7.1.2. Pointers

The type is a null pointer when it is created, it will later be set by the **Declarator** class. The pointer type is given as a constructor parameter.

```
  private Type m_pointerType;

  public Type(Type pointerType) {
    m_sort = Sort.Pointer;
    m_pointerType = pointerType;
```

```
}

public Type PointerType {
  get { return m_pointerType; }
  set { m_pointerType = value; }
}

public bool IsPointer() {
  return (m_sort == Sort.Pointer);
}

public bool IsIntegralOrPointer() {
  return IsIntegral() || IsPointer();
}

public bool IsArithmeticOrPointer() {
  return IsArithmetic() || IsPointer();
}

public bool IsIntegralLogicalOrPointer() {
  return IsIntegralOrPointer() || IsLogical();
}
```

7.1.3. Arrays

In case of an array, the constructor takes its size and the array type. When the type is created, the array size can be zero. In that case it will later be set by the length of its initialization list or become marked as uncomplete, in which case is can only be used as function parameter.

```
private int m_arraySize;
private Type m_arrayType;

public Type(int arraySize, Type arrayType) {
  m_sort = Sort.Array;
  m_arraySize = arraySize;
  m_arrayType = arrayType;
}

public int ArraySize {
  get { return m_arraySize; }
  set { m_arraySize = value; }
}

public Type ArrayType {
  get { return m_arrayType; }
  set { m_arrayType = value; }
}

public bool IsArray() {
  return (m_sort == Sort.Array);
}

public bool IsPointerOrArray() {
  return IsPointer() || IsArray();
}

public bool IsPointerArrayOrString() {
  return IsPointerOrArray() || IsString();
}

public bool IsIntegralPointerOrArray() {
  return IsIntegral() || IsPointerOrArray();
}
```

The **PointerOrArrayType** property gets the pointer or array type, which case it may be.

```
  public Type PointerOrArrayType {
    get { return (m_sort == Sort.Pointer) ? m_pointerType : m_arrayType; }
  }
```

7.1.4. Functions

C support both old-style and new-style function declarations. The old-style declaration takes a parameter list of names, that is matched to a set of declarations, while the new-style declaration takes a parameter list of declarations. However, both styles have in common that the function has a return type.

```
  public enum FunctionStyle {Old, New};
  private FunctionStyle m_functionStyle;
  private Type m_returnType;
```

The old-style function has name list, while the new style function has a parameter list, made up by name-symbol pairs.

```
  private List<string> m_nameList;
  private List<Symbol> m_parameterList;
```

The parameter list is transformed into a type list, holding the types of each parameter.

```
  private List<Type> m_typeList;
```

Some functions, like **printf** and **scanf**, are variadic, which means that they can take a various number of parameters.

```
  private bool m_variadic;
```

The following constructor takes an old-style function. The names in the list must be unique; that is, the same name may not appear twice in the list.

```
  public Type(Type returnType, List<string> nameList) {
    m_sort = Sort.Function;
    m_functionStyle = FunctionStyle.Old;
    m_returnType = returnType;
    m_nameList = nameList;
    m_parameterList = null;
    m_variadic = false;
```

We add the names of the list to a set, and check that they hold the same size. If they do not, there is at least two names with the same name, which is not allowed. If the same name is added to set twice, only the first is stored in the set, and the set is smaller than the list.

```
    Assert.Error(nameList.Count == new HashSet<string>(nameList).Count,
                 null, Message.Duplicate_name_in_parameter_list);
  }
```

The following constructor takes a new-style function, with return type, parameter list, and variadic status.

```
  public Type(Type returnType, List< Symbol> parameterList,
              bool variadic) {
      m_sort = Sort.Function;
      m_functionStyle = FunctionStyle.New;
      m_returnType = returnType;
      m_nameList = null;
      m_parameterList = parameterList;
      m_variadic = variadic;
      m_typeList = null;
```

If the parameter list is not null, we iterate through it and add the types to the type list. If the parameter list is null, the type list also becomes null. In that case the function lacks a parameter list and, when called, accepts all arguments.

```
      if (parameterList != null) {
        m_typeList = new List<Type>();

        foreach (Symbol symbol in parameterList) {
          m_typeList.Add(symbol.Type);
        }
      }
    }

    public bool IsFunction() {
      return (m_sort == Sort.Function);
    }

    public bool IsFunctionPointer() {
      return (m_sort == Sort.Pointer) && m_pointerType.IsFunction();
    }

    public bool IsArrayFunctionOrString() {
      return IsArray() || IsFunction() || IsString();
    }

    public bool IsIntegralPointerOrFunction() {
      return IsIntegralOrPointer() || IsFunction();
    }

    public bool IsPointerArrayStringOrFunction() {
      return IsPointer() || IsArrayFunctionOrString ();
    }

    public bool IsIntegralPointerOrFunction() {
      return IsIntegralOrPointer() || IsFunction();
    }

    public bool IsIntegralPointerArrayOrFunction() {
      return IsIntegralPointerOrArray() || IsFunction();
    }

    public bool IsIntegralPointerArrayStringOrFunction() {
      return IsIntegralPointerArrayOrFunction() || IsString();
    }

    public bool IsArithmeticPointerArrayStringOrFunction() {
      return IsArithmeticOrPointer() || IsArray() ||
             IsFunction() || IsString();
    }
```

The following properties gets the style, name list, parameter list, type list, return type, and whether the function is variadic.

```
    public FunctionStyle Style {
      get { return m_functionStyle; }
    }

    public List<string> NameList {
      get { return m_nameList; }
    }

    public List<Symbol> ParameterList {
      get { return m_parameterList; }
    }

    public List<Type> TypeList {
      get { return m_typeList; }
    }

    public Type ReturnType {
      get { return m_returnType; }
      set { m_returnType = value; }
    }

    public bool IsVariadic() {
      return m_variadic;
    }
```

7.1.5. Structs and Unions

A struct and union takes a list of symbols. A struct or union is **tagged** if is given a name:

```
struct {
  int a, b;
};
```

(a) An untagged struct

```
struct s {
  int a, b;
};
```

(b) A struct with tag name **s**

A struct or union definition can be meaningful or meaningless. The declaration **struct s {int i;};** is meaningful, since **s** can later be used to define variables. The declaration **struct {int i;} t;** is also meaningful, since **t** is a defined variable. However, the declaration **struct {int i;};** is meaningless, but allowed.

The member map may be null, in which case the declaration is uncomplete. Then it can only be used when defining a pointer to it, which is useful when constructing linked lists.

```
    private IDictionary<string,Symbol> m_memberMap;
    private List<Symbol> m_memberList;

    public Type(Sort sort, IDictionary<string,Symbol> symbolMap,
                List<Symbol> symbolList) {
      m_sort = sort;
      m_memberMap = symbolMap;
      m_memberList = symbolList;
    }

    public IDictionary<string,Symbol> MemberMap {
```

```
  get { return m_memberMap; }
  set { m_memberMap = value; }
}

public List<Symbol> MemberList {
  get { return m_memberList; }
  set { m_memberList = value; }
}

public bool IsStruct() {
  return (m_sort == Sort.Struct);
}

public bool IsUnion() {
  return (m_sort == Sort.Union);
}

public bool IsStructOrUnion() {
  return IsStruct() || IsUnion();
}

public bool IsArrayFunctionStringStructOrUnion() {
  return IsArray() || IsFunction() || IsString() || IsStructOrUnion();
}
```

7.1.6. Enumerations

The enumeration type (**enum**) is stored as a signed integer with a value, explicitly stated or implicitly assigned. However, the **Specifier** class of Chapter 5 needs to know if the type is **enum** to initialize its value. Therefore, we add the **m_enumItem** field.

```
private ISet<Pair<Symbol,bool>> m_enumItemSet;

public Type(ISet<Pair<Symbol,bool>> enumItemSet) {
  m_sort = Sort.SignedInt;
  m_enumItemSet = enumItemSet;
}

public ISet<Pair<Symbol,bool>> EnumItemSet {
  get { return m_enumItemSet; }
}
```

7.1.7. Type Size

Each type has a size, even though void and function as well as incomplete arrays, struct, and union sizes are defined to zero. The size of an array is its size times the size of its type, the size of a struct is the sum of the sizes of its members, and the size of a union is the size of its largest member. Note that a pointer always has the same size, regardless of what it points at.

```
public int Size() {
  switch (m_sort) {
    case Sort.Array:
      return m_arraySize * m_arrayType.Size();

    case Sort.Struct:
        if (m_memberMap != null) {
          int size = 0;

          foreach (Symbol symbol in m_memberMap.Values) {
            size += symbol.Type.Size();
          }

          return size;
        }
        else {
          return 0;
        }

      case Sort.Union:
        if (m_memberMap == null) {
          int size = 0;

          foreach (Symbol symbol in m_memberMap.Values) {
            size = Math.Max(size, symbol.Type.Size());
          }

          return size;
        }
        else {
          return 0;
        }

      case Sort.Logical:
        return TypeSize.SignedIntegerSize;

      default:
        return TypeSize.Size(m_sort);
    }
  }
```

The **SizeArray** works as **Size** above, with the difference that arrays, functions, and strings is given pointer size. This method is called when arguments are passed in function calls.

```
  public int SizeArray() {
    switch (m_sort) {
      case Sort.Array:
      case Sort.Function:
      case Sort.String:
        return TypeSize.PointerSize;

      default:
        return Size();
    }
  }
```

7.1.8. Complete Types

It is possible to define an array without stating its size, in which case the array is given the size zero. In that case, the array size must be determined by the size of its initialization list. However, if the array definition lacks an initialization list, the array keeps the size zero and is considered incomplete. In the same way, it is possible to define only the name tag of a struct or union, with its member map to be defined later. In that case, the member map is given the value null, which is keep if the member map is not defined, and the struct or union is consider incomplete. Variables can only have complete types, and the pointer type, array type or function return type must be also complete.

```
public bool IsComplete() {
  switch (m_sort) {
    case Sort.Array:
      return (m_arraySize > 0);

    case Sort.Struct:
    case Sort.Union:
      return (m_memberMap != null);

    default:
      return true;
  }
}
```

7.1.9. Constant and Volatile

The idea of the volatile qualifier is to prevent optimization, and since this book is focused on optimization techniques, we have no real use for the volatile qualifier. However, for the sake of completeness we include the **m_volatile** field in the **Type** class. It makes no difference if a type is volatile or not.

```
private bool m_constant, m_volatile;

public bool Constant {
  get { return m_constant; }
  set { m_constant = value; }
}

public bool Volatile {
  get { return m_volatile; }
  set { m_volatile = value; }
}
```

A type is constant if its field **m_constant** is true. However, a struct or union is constant if it is constant in itself, or if is any of its members if recursively constant.

```
public bool IsConstantRecursive() {
  if (m_constant) {
    return true;
  }
  else if (IsStructOrUnion() && (m_memberMap != null)) {
    foreach (Symbol symbol in m_memberMap.Values) {
      if (symbol.Type.IsConstantRecursive()) {
        return true;
      }
    }
  }

  return false;
}
```

7.1.10. Hash Code and Equals

The **GetHashCode** method simply return the hash code of its base class. It has been included for the sake of completeness since we also include the **Equals** method.

```
    public override int GetHashCode() {
      return base.GetHashCode();
    }
```

The **Equals** method return true if the types are equal.

```
    public override bool Equals(object obj) {
      if (obj is Type) {
        Type type = (Type) obj;

        if ((m_constant == type.m_constant) &&
            (m_volatile == type.m_volatile) && (m_sort == type.m_sort)) {
```

Two pointers are considered to be equal if the types they point at are equal.

```
          switch (m_sort) {
            case Sort.Pointer:
              return m_pointerType.Equals(type.m_pointerType);
```

Two arrays are equal if their types are equal and they have the same size, or if at least one of the arrays have size zero.

```
            case Sort.Array:
              return ((m_arraySize == 0) || (type.m_arraySize == 0) ||
                      (m_arraySize == type.m_arraySize)) &&
                     m_arrayType.Equals(type.m_arrayType);
```

Two structs or unions are equal if they both are incomplete (their member maps are null) or if their member maps are equal. Note that they must not only have the same members, the members must also appear in the same order.

```
            case Sort.Struct:
            case Sort.Union:
              return ((m_memberMap == null) && (type.m_memberMap == null)) ||
                     ((m_memberMap != null) && (type.m_memberMap != null) &&
                     m_memberMap.Equals(type.m_memberMap));
```

Two functions are equal if their return types are equal, they have the same style (old or new) and their type lists are equal. Note that two functions can be equal even if their parameters have different names, at long as their types are equal.

```
            case Sort.Function:
              return m_returnType.Equals(type.m_returnType) &&
                     (((m_typeList == null) && (type.m_typeList == null)) ||
                      ((m_typeList != null) && (type.m_typeList != null) &&
                     m_typeList.SequenceEqual(type.m_typeList))) &&
                     (m_variadic == type.m_variadic);
```

Finally, if none of the cases above apply, we compare the sorts of the types.

```
            default:
              return true;
          }
        }
      }

      return false;
    }
```

7.1.11. Predefined Types

We have a set of predefined types.

```
    public static Type SignedShortIntegerType =
      new Type(Sort.SignedShortInt);
    public static Type UnsignedShortIntegerType =
      new Type(Sort.UnsignedShortInt);
    public static Type SignedIntegerType = new Type(Sort.SignedInt);
    public static Type UnsignedIntegerType = new Type(Sort.UnsignedInt);
    public static Type SignedLongIntegerType = new Type(Sort.SignedLongInt);
    public static Type UnsignedLongIntegerType =
      new Type(Sort.UnsignedLongInt);
    public static Type FloatType = new Type(Sort.Float);
    public static Type DoubleType = new Type(Sort.Double);
    public static Type LongDoubleType = new Type(Sort.LongDouble);
    public static Type SignedCharType = new Type(Sort.SignedChar);
    public static Type UnsignedCharType = new Type(Sort.UnsignedChar);
    public static Type StringType = new Type(Sort.String);
    public static Type VoidType = new Type(Sort.Void);
    public static Type PointerTypeX = new Type(SignedIntegerType);
    public static Type VoidPointerType = new Type(new Type(Sort.Void));
    public static Type LogicalType = new Type(Sort.Logical);
```

7.1.12. ToString

Finally, we the **ToString** method. It returns the sort of the type as a string, with underlines replaced but spaces (' ').For instance, The sort **signed_short_int** becomes the string "signed short int". It is only called in error messages.

```
   public override string ToString() {
      return Enum.GetName(typeof(Sort), m_sort).Replace("_", " ").ToLower();
    }
  }
}
```

7.2. Type Size

The **TypeSize** class holds the sizes of the types and the minimum and maximum values of each type.

TypeSize.cs

```
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  class TypeSize {
    public static int PointerSize;
    public static int SignedIntegerSize;
```

The **m_maskMap** map holds the bit masks for integer values of one, two, and four bytes. The values are the same for the Linux and the Windows environment.

```
    private static IDictionary<int, BigInteger>
      m_maskMap = new Dictionary<int, BigInteger>() {
        {1, (BigInteger) 0x000000FF},
        {2, (BigInteger) 0x0000FFFF},
        {4, (BigInteger) 0xFFFFFFFF}};
```

The **m_sizeMap** map holds the size of type.

```
public static IDictionary<Sort,int> m_sizeMap =
  new Dictionary<Sort,int>();
```

The **m_signedMap** and **m_unsignedMap** maps hold the signed and unsigned integer types of each size.

```
public static IDictionary<int,Sort>
  m_signedMap = new Dictionary<int,Sort>(),
  m_unsignedMap = new Dictionary<int,Sort>();
```

The **m_minValueMap** and **m_maxValueMap** hold the minimum and maximum values of each integral type.

```
private static IDictionary<Sort,BigInteger>
  m_minValueMap = new Dictionary<Sort,BigInteger>(),
  m_maxValueMap = new Dictionary<Sort,BigInteger>();
```

The sizes and values of the types depends on whether the compiler generates code for the Linux or the Windows environment. In the Linux environment, a pointer is eight bytes and an integer is four bytes.

```
  if (Start.Linux) {
    PointerSize = 8;
    SignedIntegerSize = 4;

    m_sizeMap.Add(Sort.Void, 0);
    m_sizeMap.Add(Sort.Function, 0);
    m_sizeMap.Add(Sort.Logical, 1);
    m_sizeMap.Add(Sort.Pointer, 8);
    m_sizeMap.Add(Sort.Array, 8);
    m_sizeMap.Add(Sort.String, 8);
    m_sizeMap.Add(Sort.SignedChar, 1);
    m_sizeMap.Add(Sort.UnsignedChar, 1);
    m_sizeMap.Add(Sort.SignedShortInt, 2);
    m_sizeMap.Add(Sort.UnsignedShortInt, 2);
    m_sizeMap.Add(Sort.SignedInt, 4);
    m_sizeMap.Add(Sort.UnsignedInt, 4);
    m_sizeMap.Add(Sort.SignedLongInt, 8);
    m_sizeMap.Add(Sort.UnsignedLongInt, 8);
    m_sizeMap.Add(Sort.Float, 4);
    m_sizeMap.Add(Sort.Double, 8);
    m_sizeMap.Add(Sort.LongDouble, 8);

    m_signedMap.Add(1, Sort.SignedChar);
    m_signedMap.Add(2, Sort.SignedShortInt);
    m_signedMap.Add(4, Sort.SignedInt);
    m_signedMap.Add(8, Sort.SignedLongInt);

    m_unsignedMap.Add(1, Sort.UnsignedChar);
    m_unsignedMap.Add(2, Sort.UnsignedShortInt);
    m_unsignedMap.Add(4, Sort.UnsignedInt);
    m_unsignedMap.Add(8, Sort.UnsignedLongInt);

    m_minValueMap.Add(Sort.Logical, 0);
    m_minValueMap.Add(Sort.SignedChar, -128);
    m_minValueMap.Add(Sort.UnsignedChar, 0);
    m_minValueMap.Add(Sort.SignedShortInt, -32768);
    m_minValueMap.Add(Sort.UnsignedShortInt, 0);
    m_minValueMap.Add(Sort.SignedInt, -2147483648);
```

```
        m_minValueMap.Add(Sort.UnsignedInt, 0);
        m_minValueMap.Add(Sort.Array, 0);
        m_minValueMap.Add(Sort.Pointer, 0);
        m_minValueMap.Add(Sort.SignedLongInt, -9223372036854775808);
        m_minValueMap.Add(Sort.UnsignedLongInt, 0);

        m_maxValueMap.Add(Sort.Logical, 1);
        m_maxValueMap.Add(Sort.SignedChar, 127);
        m_maxValueMap.Add(Sort.UnsignedChar, 255);
        m_maxValueMap.Add(Sort.SignedShortInt, 32767);
        m_maxValueMap.Add(Sort.UnsignedShortInt, 65535);
        m_maxValueMap.Add(Sort.SignedInt, 2147483647);
        m_maxValueMap.Add(Sort.UnsignedInt, 4294967295);
        m_maxValueMap.Add(Sort.Array, 4294967295);
        m_maxValueMap.Add(Sort.Pointer, 4294967295);
        m_maxValueMap.Add(Sort.SignedLongInt, 9223372036854775807);
        m_maxValueMap.Add(Sort.UnsignedLongInt, 18446744073709551615);
      }
```

The type sizes for the Windows environment are defined in Chapter 13.

```
      if (Start.Windows) {
        // ...
      }
    }

    public static BigInteger GetMinValue(Sort sort) {
      return m_minValueMap[sort];
    }

    public static BigInteger GetMaxValue(Sort sort) {
      return m_maxValueMap[sort];
    }

    public static BigInteger GetMask(Sort sort) {
      return m_maskMap[m_sizeMap[sort]];
    }

    public static Type SizeToSignedType(int size) {
      return new Type(m_signedMap[size]);
    }

    public static Type SizeToUnsignedType(int size) {
      return new Type(m_unsignedMap[size]);
    }

    public static int Size(Sort sort) {
      return m_sizeMap[sort];
    }
  }
}
```

7.3. Type Casting

Type casting occurs on several occasions in C. There are two types of cast: explicit (manual) conversation where type cast is stated and implicit (automatic) where it the type cast is omitted:

```
double x = 3.1;
int y = x,       // implicit cast
    z = (int) x; // explicit cast
```

TypeCast.cs

```
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  public class TypeCast {
```

The **LogicalToIntegral** method makes sure the expression holds integral type. If it holds logical type, it is cast from logical type to signed integer type.

```
    public static Expression LogicalToIntegral(Expression expression) {
      if (expression.Symbol.Type.IsLogical()) {
        return ImplicitCast(expression, Type.SignedIntegerType);
      }

      return expression;
    }
```

The **LogicalToFloating** method makes sure the expression holds floating type. If it holds logical type, it is cast from logical type to double type.

```
    public static Expression LogicalToFloating(Expression expression) {
      if (expression.Symbol.Type.IsLogical()) {
        return ImplicitCast(expression, Type.DoubleType);
      }

      return expression;
    }
```

The **ToLogical** method makes sure the expression has a logical type. If it does not already hold logical type, the expression is cast to logical type.

```
    public static Expression ToLogical(Expression expression) {
      if (!expression.Symbol.Type.IsLogical()) {
        return ImplicitCast(expression, Type.LogicalType);
      }

      return expression;
    }
```

7.3.1. Implicit Cast

The **ImplicitCast** method performs an implicit cast. To begin with, if the expression is constant, we return the constant casted expression.

```
    public static Expression ImplicitCast(Expression fromExpression,
                                          Type toType) {
      Expression constantExpression =
        ConstantExpression.Cast(fromExpression, toType);
      if (constantExpression != null) {
        return constantExpression;
      }
      Type fromType = fromExpression.Symbol.Type;

      if (fromType.Equals(toType) ||
          (fromType.IsLogical() && toType.IsLogical()) ||
          (fromType.IsPointerArrayStringOrFunction() &&
           toType.IsPointerOrArray()) ||
          (fromType.IsPointerArrayStringOrFunction() &&
           toType.IsIntegral() && (fromType.Size() == toType.Size())) ||
          (((fromType.IsFloating() && toType.IsFloating()) ||
            (fromType.IsIntegralPointerOrFunction() &&
             toType.IsIntegralPointerArrayOrFunction())) &&
           (fromType.SizeArray() == toType.SizeArray()))) {
        return fromExpression;
      }
      else {
        return ExplicitCast(fromExpression, toType);
      }
    }
```

7.3.2. Explicit Cast

The **ExplicitCast** method perform an explicit cast between the different types of C.

```
  public static Expression ExplicitCast(Expression sourceExpression,
                                        Type targetType) {
    Expression constantExpression =
      ConstantExpression.ConstantCast(sourceExpression, targetType);
    if (constantExpression != null) {
      return constantExpression;
    }

    Symbol sourceSymbol = sourceExpression.Symbol, targetSymbol = null;
    Type sourceType = sourceSymbol.Type;
    List<MiddleCode> shortList = sourceExpression.ShortList,
                     longList = sourceExpression.LongList;

    if (targetType.IsVoid()) {
      targetSymbol = new Symbol(targetType);
    }
    else if (sourceType.IsStructOrUnion() && targetType.IsStructOrUnion()) {
      Assert.Error(sourceType.Equals(targetType), sourceType + " to " +
                   targetType, Message.Invalid_type_cast);
      targetSymbol = new Symbol(targetType);
    }
```

If the source type is logical and the target type is integral, array, or pointer, we need to backpatch the true-set and false-set of the expression's symbol.

The true-set becomes backpatched to a statement that assign the value one to a temporary variable in the

```
    else if (sourceType.IsLogical() &&
             targetType.IsIntegralPointerOrArray()) {
      targetSymbol = new Symbol(targetType);
```

We backpatch the true-set to a statement that assign the value one to a temporary variable.

```
      Symbol oneSymbol = new Symbol(targetType, BigInteger.One);
      MiddleCode trueCode =
        new MiddleCode(MiddleOperator.Assign, targetSymbol, oneSymbol);
      MiddleCodeGenerator.Backpatch(sourceSymbol.TrueSet, trueCode);
      longList.Add(trueCode);
```

In the middle of backpatching of the true-set and false-set, we add code that jumps to the zero value assignment.

```
        MiddleCode targetCode = new MiddleCode(MiddleOperator.Empty);
        longList.Add(new MiddleCode(MiddleOperator.Jump, targetCode));
```

We backpatch the false-set to a statement that assign the value one to a temporary variable.

```
        Symbol zeroSymbol = new Symbol(targetType, BigInteger.Zero);
        MiddleCode falseCode =
          new MiddleCode(MiddleOperator.Assign, targetSymbol, zeroSymbol);
        MiddleCodeGenerator.Backpatch(sourceSymbol.FalseSet, falseCode);
        longList.Add(falseCode);
```

Finally, we add the target of the jump code.

```
        longList.Add(targetCode);
      }
```

If the source type is logical and the target type is floating, we push the values one and zero on the floating-point stack instead of assigning a variable.

```
      else if (sourceType.IsLogical() && targetType.IsFloating()) {
        targetSymbol = new Symbol(targetType);
```

We backpatch the true-set of the source symbol to the code that pushes one at the stack.

```
        MiddleCode trueCode = new MiddleCode(MiddleOperator.PushOne);
        MiddleCodeGenerator.Backpatch(sourceSymbol.TrueSet, trueCode);
        longList.Add(trueCode);
```

Similar to the integral case above, in the middle of backpatching of the true-set and false-set, we add code that jumps to the push-zero assignment.

```
        MiddleCode targetCode = new MiddleCode(MiddleOperator.Empty);
        longList.Add(new MiddleCode(MiddleOperator.Jump, targetCode));
```

We backpatch the true-set of the source symbol to the code that pushes one at the stack.

```
        MiddleCode falseCode = new MiddleCode(MiddleOperator.PushZero);
        MiddleCodeGenerator.Backpatch(sourceSymbol.FalseSet, falseCode);
        longList.Add(falseCode);
```

Finally, we add the target of the jump code.

```
        longList.Add(targetCode);
      }
```

In case of an arithmetic (integral or floating), array, string, or function source type, and logical target type, we test whether the value of the source expression is non-zero, and generate a true-set and a false-set from the test.

```
      else if (sourceType.IsArithmeticPointerArrayStringOrFunction() &&
               targetType.IsLogical()) {
```

We begin by creating the zero value to compare the source value with. It is **decimal** in case of floating type, and **BigInteger** in case of integral, pointer, array, string, of function.

```
        object zeroValue = sourceType.IsLogical() ? ((object) decimal.Zero)
                                                  : ((object)BigInteger.Zero);
        Symbol zeroSymbol = new Symbol(targetType, zeroValue);
```

We add code for comparing the source value with the zero value. If the source value is not zero we jump to yet unknown address. We add that first instruction to the true-set.

```
        MiddleCode testCode =
          new MiddleCode(MiddleOperator.NotEqual, null,
                         sourceSymbol, zeroSymbol);
        ISet<MiddleCode> trueSet = new HashSet<MiddleCode>();
        trueSet.Add(testCode);
        longList.Add(testCode);
```

If the source value is zero, we jump to another yet unknown address. We add that instruction to the false-set.

```
        MiddleCode gotoCode = new MiddleCode(MiddleOperator.Jump);
        ISet<MiddleCode> falseSet = new HashSet<MiddleCode>();
        falseSet.Add(gotoCode);
        longList.Add(gotoCode);
```

Finally, the target symbol holds the true-set and false-set.

```
        targetSymbol = new Symbol(trueSet, falseSet);
      }
```

In case of a cast from an integral type of one byte (pointers are always two bytes) to a floating type, it becomes a little bit more complicated since there is no assembly converting operator. Instead, we need to first convert the small integral value (one byte) to a larger integral value (two bytes), which we in turn convert to a floating value.

In case of cast between two floating values, we only create the target symbol. We do not need to do anything else, since the floating-point value is already stored at the floating-point stack.

```
      else if (sourceType.IsFloating() && targetType.IsFloating()) {
        targetSymbol = new Symbol(targetType);
      }
```

The same goes for the other way around, when converting a floating value to a small (one byte) integral value, we first need to convert the floating value to an integral value of two bytes, which we in turn convert to an integral value of one byte.

```
      else if (sourceType.IsFloating() &&
               targetType.IsIntegralPointerOrArray()) {
        targetSymbol = new Symbol(targetType);
```

Due to technical limitations, there is no assembly instruction for popping a one-byte integral value from the floating-point stack. Therefore, when casting a floating value to a one-byte integral value, we first need to add a floating-to-integral instruction to a temporary integer value, which we in turn cast to the one-byte integral value.

```
        if (targetType.Size() == 1) {
          Type tempType = sourceType.IsSigned() ? Type.SignedIntegerType
                                                : Type.UnsignedIntegerType;
          Symbol tempSymbol = new Symbol(tempType);
          MiddleCode tempCode =
            new MiddleCode(MiddleOperator.FloatingToIntegral, tempSymbol,
                           sourceSymbol);
          longList.Add(tempCode);
          MiddleCode resultCode =
            new MiddleCode(MiddleOperator.IntegralToIntegral, targetSymbol,
                           tempSymbol);
          longList.Add(resultCode);
        }
```

If the target type size is more than one byte, we simply add a floating-to-integral instruction.

```
        else {
          MiddleCode resultCode =
            new MiddleCode(MiddleOperator.FloatingToIntegral, targetSymbol,
                           sourceSymbol);
          longList.Add(resultCode);
        }
      }
```

If the source type is integral, pointer, array, string, or function, and the target type is floating, we have the opposite situation if the source type size is one. We cast the one-byte value to a temporary integer value, which we in turn cast to a floating value.

```
      else if (sourceType.IsIntegralPointerArrayStringOrFunction () &&
               targetType.IsFloating()) {
        targetSymbol = new Symbol(targetType);

        if (sourceType.Size() == 1) {
          Type tempType = sourceType.IsSigned() ? Type.SignedIntegerType
                                                : Type.UnsignedIntegerType;
          Symbol tempSymbol = new Symbol(tempType);
          MiddleCodeGenerator.
            AddMiddleCode(longList, MiddleOperator.IntegralToIntegral,
                          tempSymbol, sourceSymbol);
          MiddleCodeGenerator.
            AddMiddleCode(longList, MiddleOperator.IntegralToFloating,
                          targetSymbol, tempSymbol);
        }
```

If the target type size is more than one byte, we simply add an integral-to-floating instruction.

```
        else {
          MiddleCodeGenerator.
            AddMiddleCode(longList, MiddleOperator.IntegralToFloating,
                          targetSymbol, sourceSymbol);
        }
      }
```

If the source type is integral, pointer, array, string, or function, and the target type is integral, array, or pointer, we simply add an integral-to-integral instruction.

```
      else if (sourceType.IsIntegralPointerArrayStringOrFunction () &&
               targetType.IsIntegralPointerOrArray()) {
        targetSymbol = new Symbol(targetType);
        MiddleCodeGenerator.
          AddMiddleCode(longList, MiddleOperator.IntegralToIntegral,
                        targetSymbol, sourceSymbol);
      }
```

In none of the cases above applies, the target symbol is still null, and we report an error.

```
      Assert.Error(targetSymbol != null, sourceType + " to " +
                   targetType, Message.Invalid_type_cast);
      return (new Expression(targetSymbol, shortList, longList));
    }
```

7.3.3. Type Promotion

Type promotion is the process of converting a “smaller” type to a “larger” type in a binary expression. For instance, in the expression below, the integer **i** shall be converted to a double value, since double is larger than integer.

```
int i = 3;
double x = 3.14, y;
y = x + i;
```

Below is a table holding the relations of the types. Note that **float** is considered to be larger than **long int**, even if they have the same size. In the same way, **short** is considered to the larger than **char**.

Largest type	long double
	double
	float
	signed long int unsigned long int
	signed int unsigned int
	signed short int unsigned short int
Smallest type	signed char unsigned char

In accordance with the Ansi C standard the signed and unsigned types have the same size, and a character is always one byte. This due to historical reasons. In the early versions of C, there was no void type. Instead, **char** was used a generic type in pointer expressions. Therefore, a character always holds one byte. However, the standard does not state whether a signed value shall be converted to an unsigned value or the other way around in the expression below. However, in this book we always convert the unsigned value to the matching signed type (if the hold the same size).

```
signed int i = -3;
unsigned int u = 6;
i + u;
```

TypeCast.cs

```
    public static Type MaxType(Type leftType, Type rightType) {
      if ((leftType.IsFloating() && !rightType.IsFloating()) ||
          ((leftType.Size() == rightType.Size()) &&
           leftType.IsSigned() && !rightType.IsSigned())) {
        return leftType;
      }
      else if ((!leftType.IsFloating() && rightType.IsFloating()) ||
               ((leftType.Size() == rightType.Size()) &&
               !leftType.IsSigned() && rightType.IsSigned())) {
        return rightType;
      }
      else {
        return (leftType.Size() > rightType.Size()) ? leftType : rightType;
      }
    }
  }
}
```

8. Constant Expression

There are advantages by calculating the values of constant expression during the compilation rather than the execution. Besides the optimization benefits, resulting in smaller and faster code, there are some occasions when we need to calculate the value. In array definitions, we need to evaluate the size of the array, in initialized enumeration values we need to evaluate the value to be able to assign the next uninitialized enumeration its correct value, in case statement we need to decide the value in order to make sure the same case value does not occur twice in the same switch statement, and the preprocessor need to evaluate the values of **if** directives.

8.1. Unary and Binary Expressions

The **ConstantExpression** class holds methods that calculates the value of unary and binary logical and arithmetic expression, if their operands are constant.

ConstantExpression.cs
```
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  public class ConstantExpression {
```

The **IsConstant** method returns true if the expression can be evaluated to a value, which it can if it holds logical type with the true-set or the false-set empty (both sets cannot be empty). If one set is empty, the expression holds no **if** statement, only a **goto** statement. It can also be evaluated to a constant value if it holds integral, pointer type, or floating type with a not-null value.

```
  public class ConstantExpression {
    public static bool IsConstant(Expression expression) {
      Symbol resultSymbol = expression.Symbol;

      if (resultSymbol.Type.IsLogical()) {
        return (resultSymbol.TrueSet.Count == 0) ||
               (resultSymbol.FalseSet.Count == 0);
      }
      else {
        return (resultSymbol.Value is BigInteger) ||
               (resultSymbol.Value is decimal);
      }
    }
```

The **IsTrue** method returns true if the (assumed constant) expression can be evaluated to a true value, which it can if the value is a reference to a **BigInteger** object that is not zero, or a **decimal** object that is not zero, or if the true-set is not empty.

```
    public static bool IsTrue(Expression expression) {
      Symbol symbol = expression.Symbol;

      if (symbol.Type.IsLogical()) {
        return (expression.Symbol.TrueSet.Count > 0);
      }
      else {
        return !symbol.Value.Equals(BigInteger.Zero) &&
            !symbol.Value.Equals((decimal) 0);
      }
    }

    private static Expression LogicalToIntegral(Expression expression) {
      if (expression.Symbol.Type.IsLogical()) {
        return ConstantCast(expression, Type.SignedIntegerType);
      }

      return expression;
    }

    private static Expression LogicalToFloating(Expression expression) {
      if (expression.Symbol.Type.IsLogical()) {
        return ConstantCast(expression, Type.DoubleType);
      }

      return expression;
    }

    private static Expression ToLogical(Expression expression) {
      if (!expression.Symbol.Type.IsLogical()) {
        return ConstantCast(expression, Type.LogicalType);
      }

      return expression;
    }
```

8.1.1. Relation Expressions

The **Relation** method returns the calculated value of the expression, if possible. Otherwise, it returns null.

```
    public static Expression Relation(MiddleOperator middleOp,
                                      Expression leftExpression,
                                      Expression rightExpression) {
      if (!IsConstant(leftExpression) || !IsConstant(rightExpression)) {
        return null;
      }

      leftExpression = LogicalToIntegral(leftExpression);
      rightExpression = LogicalToIntegral(rightExpression);

      List<MiddleCode> longList = new List<MiddleCode>();
      MiddleCode jumpCode = new MiddleCode(MiddleOperator.Jump);
      longList.Add(jumpCode);
      ISet<MiddleCode> jumpSet = new HashSet<MiddleCode>();
      jumpSet.Add(jumpCode);

      object leftValue = leftExpression.Symbol.Value,
             rightValue = leftExpression.Symbol.Value;

      Assert.ErrorXXX(((leftValue is BigInteger) && (rightValue is BigInteger))
||
                      ((leftValue is decimal) && (rightValue is decimal)));

      int compareTo = 0;
      if ((leftValue is BigInteger) && (rightValue is BigInteger)) {
```

```
    BigInteger leftBigInteger = (BigInteger) leftValue,
               rightBigInteger = (BigInteger) leftValue;
    compareTo = leftBigInteger.CompareTo(rightBigInteger);
  }
  else {
    decimal leftDecimal, rightDecimal;

    if (leftValue is BigInteger) {
      leftDecimal = (decimal) ((BigInteger) leftValue);
      rightDecimal = (decimal) rightValue;
    }
    else if (rightValue is BigInteger) {
      leftDecimal = (decimal) leftValue;
      rightDecimal = (decimal) ((BigInteger)rightValue);
    }
    else {
      leftDecimal = (decimal) leftValue;
      rightDecimal = (decimal) rightValue;
    }

    compareTo = leftDecimal.CompareTo(rightDecimal);
  }

  bool resultValue = false;
  switch (middleOp) {
    case MiddleOperator.Equal:
      resultValue = (compareTo == 0);
      break;

    case MiddleOperator.NotEqual:
      resultValue = (compareTo != 0);
      break;

    case MiddleOperator.LessThan:
      resultValue = (compareTo < 0);
      break;

    case MiddleOperator.LessThanEqual:
      resultValue = (compareTo <= 0);
      break;

    case MiddleOperator.GreaterThan:
      resultValue = (compareTo > 0);
      break;

    case MiddleOperator.GreaterThanEqual:
      resultValue = (compareTo >= 0);
      break;
  }

  Symbol resultSymbol = resultValue ? (new Symbol(jumpSet, null))
                                    : (new Symbol(null, jumpSet));
  return (new Expression(resultSymbol, null, longList));
}
```

8.1.2. Logical Expressions

The **Logical** method evaluate the value of two constant logical expressions, with the **or** or **and** operator.

```
public static Expression Logical(MiddleOperator middleOp,
                                 Expression leftExpression,
                                 Expression rightExpression) {
  if (!IsConstant(leftExpression) || !IsConstant(rightExpression)) {
    return null;
  }

  bool resultValue = false;
  switch (middleOp) {
    case MiddleOperator.LogicalAnd:
      resultValue = IsTrue(leftExpression) && IsTrue(rightExpression);
      break;

    case MiddleOperator.LogicalOr:
      resultValue = IsTrue(leftExpression) || IsTrue(rightExpression);
      break;
  }
```

The long list of the final expression holds only one unconditional jump instruction that is added to the jump set, which will be the true-set or false of the final expression.

```
  List<MiddleCode> longList = new List<MiddleCode>();
  MiddleCode jumpCode = new MiddleCode(MiddleOperator.Jump);
  longList.Add(jumpCode);
  ISet<MiddleCode> jumpSet = new HashSet<MiddleCode>();
  jumpSet.Add(jumpCode);
```

The resulting symbol has a logical value. The jump set becomes its true-set or false-set, depending on whether the result value is true.

```
  Symbol resultSymbol = new Symbol(null, null);
  if (resultValue) {
    resultSymbol.TrueSet = jumpSet;
  }
  else {
    resultSymbol.FalseSet = jumpSet;
  }
```

The final expression does not have a short list, and its long list is made up by the unconditional jump instruction only.

```
  return (new Expression(resultSymbol, null, longList));
}
```

8.1.3. Arithmetic Expressions

The **Arithmetic** method evaluates the value of a constant arithmetic expression. If at least one of the expressions has floating type, we call **ArithmeticFloating**, If neither of the expressions has floating type, we call **ArithmeticIntegral**.

```
public static Expression Arithmetic(MiddleOperator middleOp,
                                    Expression leftExpression,
                                    Expression rightExpression) {
  if (!IsConstant(leftExpression) || !IsConstant(rightExpression)) {
    return null;
  }
  else if (leftExpression.Symbol.Type.IsFloating() ||
           rightExpression.Symbol.Type.IsFloating()) {
    return ArithmeticFloating(middleOp, leftExpression, rightExpression);
  }
  else {
    return ArithmeticIntegral(middleOp, leftExpression, rightExpression);
  }
}
```

The **ArithmeticIntegral** method evaluate the value of a constant integral arithmetic expression. It calls in turn the **ArithmeticIntegral** method that takes two symbols as parameters. The reason for using two methods is that the latter method is called by the middle code optimizer.

```
  private static Expression ArithmeticIntegral(MiddleOperator middleOp,
                                               Expression leftExpression,
                                               Expression rightExpression){
    leftExpression = LogicalToIntegral(leftExpression);
    rightExpression = LogicalToIntegral(rightExpression);
    Symbol symbol = ArithmeticIntegral(middleOp, leftExpression.Symbol,
                                       rightExpression.Symbol);
    return (new Expression(symbol, null, null));
  }
```

The integral arithmetic evaluation is a bit more complicated since we must take pointer arithmetic in consideration.

```
  public static Symbol ArithmeticIntegral(MiddleOperator middleOp,
                                          Symbol leftSymbol,
                                          Symbol rightSymbol) {
    Type leftType = leftSymbol.Type,
         rightType = rightSymbol.Type;
```

Since this method has been called, we know that none of the expression have floating type, and the potential logical types has been cast to integral types. Therefore, we can assume that their values are **BigInteger** objects.

```
    BigInteger leftValue = (BigInteger) leftSymbol.Value,
               rightValue = (BigInteger) rightSymbol.Value,
               resultValue = 0;
```

In case of addition, we must check whether the type of the left expression is a pointer or array type. In that case we need to multiply the value of the right expression with the size of the pointer or array type.

We must also check whether the right type of the right expression is a pointer or array, in which case we multiply the left value with the size of the pointer or array type.

If neither the left nor the right expression holds pointer or array types, we simply add the values.

```
    switch (middleOp) {
      case MiddleOperator.Add:
        if (leftType.IsPointerOrArray()) {
          resultValue = leftValue +
                        (rightValue * leftType.PointerOrArrayType.Size());
        }
        else if (leftType.IsPointerOrArray()) {
          resultValue = (leftValue * rightType.PointerOrArrayType.Size()) +
                        rightValue;
        }
        else {
          resultValue = leftValue + rightValue;
        }
        break;
```

In case of subtraction, to begin with we must check if both the expressions hold pointer or array types. If they do, we subtract their values and divide the difference with size of the pointer or array type.

If the left expression (but not the right expression) holds pointer or array type, we must multiply the right value with the size of the pointer or array type before we subtract t it from the left value.

If neither the left nor the right expression holds pointer or array type, we simply subtract the values.

```
        case MiddleOperator.Subtract:
          if (leftType.IsPointerOrArray() && rightType.IsPointerOrArray()) {
            resultValue = (leftValue - rightValue) /
                          leftType.PointerOrArrayType.Size();
          }
          else if (leftType.IsPointerOrArray()) {
            resultValue = leftValue -
                          (rightValue * leftType.PointerOrArrayType.Size());
          }
          else {
            resultValue = leftValue - rightValue;
          }
          break;
```

In case of multiplication, division, or modulo, bitwise, or shift operators, we simply perform the operation.

```
        case MiddleOperator.Multiply:
          resultValue = leftValue * rightValue;
          break;

        case MiddleOperator.Divide:
          resultValue = leftValue / rightValue;
          break;

        case MiddleOperator.Modulo:
          resultValue = leftValue % rightValue;
          break;

        case MiddleOperator.BitwiseOr:
          resultValue = leftValue | rightValue;
          break;

        case MiddleOperator.BitwiseXOr:
          resultValue = leftValue ^ rightValue;
          break;

        case MiddleOperator.BitwiseAnd:
          resultValue = leftValue & rightValue;
          break;
```

In case of shift operators, we need to type cast the right value from **BigInteger** to **int**, simply because the **BigInteger** operation wants an integer in those cases.

```
        case MiddleOperator.ShiftLeft:
          resultValue = leftValue << ((int) rightValue);
          break;

        case MiddleOperator.ShiftRight:
          resultValue = leftValue >> ((int) rightValue);
          break;
      }
```

When we have evaluated the value, we need to find its type, which we do by calling the **MaxType** method in the **TypeCast** class.

```
      Type maxType = TypeCast.MaxType(leftSymbol.Type, rightSymbol.Type);
```

Finally, we return a symbol holding the maximum type and the resulting value.

```
      return (new Symbol(maxType, resultValue));
    }
```

The evaluation of a floating constant expression is simpler than the integral expression, since we do not need to take pointer arithmetic in consideration.

```
    private static Expression ArithmeticFloating(MiddleOperator middleOp,
                                                 Expression leftExpression,
                                                 Expression rightExpression) {
      leftExpression = TypeCast.LogicalToFloating(leftExpression);
      rightExpression = TypeCast.LogicalToFloating(rightExpression);
```

The value of each expression can be **BigInteger** or **decimal**. In the case of **BigInteger**, we first cast the value to **BigInteger** and then to **decimal**.

```
      Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,
                                      rightExpression.Symbol.Type);
      leftExpression = ConstantCast(leftExpression, maxType);
      rightExpression = ConstantCast(rightExpression, maxType);

      decimal leftDecimal = (decimal) leftExpression.Symbol.Value,
              rightDecimal = (decimal) rightExpression.Symbol.Value,
              resultDecimal = 0;
```

Since the values are object of the **decimal** class, we can user the regular operators.

```
      switch (middleOp) {
        case MiddleOperator.Add:
          resultDecimal = leftDecimal + rightDecimal;
          break;

        case MiddleOperator.Subtract:
          resultDecimal = leftDecimal - rightDecimal;
          break;

        case MiddleOperator.Multiply:
          resultDecimal = leftDecimal * rightDecimal;
          break;

        case MiddleOperator.Divide:
          resultDecimal = leftDecimal / rightDecimal;
          break;
      }

      Symbol resultSymbol = new Symbol(maxType, resultDecimal);
      List<MiddleCode> longList = new List<MiddleCode>();
```

```
    MiddleCodeGenerator.AddMiddleCode(longList, MiddleOperator.PushFloat,
                                      resultSymbol);
    return (new Expression(resultSymbol, null, longList));
  }
```

The **LogicalNot** method evaluates the logical invers of the expression if it is constant.

```
  public static Expression LogicalNot(Expression expression) {
    if (IsConstant(expression)) {
      expression = TypeCast.ToLogical(expression);
```

The resulting symbol is simply the original symbol with its true-set and false-set swapped.

```
      Symbol resultSymbol = new Symbol(expression.Symbol.FalseSet,
                                       expression.Symbol.TrueSet);
      return (new Expression(resultSymbol, null, expression.LongList));
    }

    return null;
  }
```

The **Arithmetic** method performs a unary arithmetic operation on a constant expression. If the expression is constant and floating **ArithmeticFloating** is called, and if it is constant and integral **ArithmeticIntegral** is called.

```
  public static Expression Arithmetic(MiddleOperator middleOp,
                                      Expression expression) {
    if (!IsConstant(expression)) {
      return null;
    }
    else if (expression.Symbol.Type.IsFloating()) {
      return ArithmeticFloating(middleOp, expression);
    }
    else {
      return ArithmeticIntegral(middleOp, expression);
    }
  }
```

The **ArithmeticIntegral** method performs unary integral arithmetic operations: unary addition and subtraction as well as bitwise not.

```
  private static Expression ArithmeticIntegral(MiddleOperator middleOp,
                                               Expression expression) {
    expression = TypeCast.LogicalToIntegral(expression);
    BigInteger value = (BigInteger) expression.Symbol.Value;
```

The unary addition operator does in fact nothing. The unary subtraction operator and bitwise not-operator perform the corresponding operations.

```
    Symbol resultSymbol = null;
    switch (middleOp) {
      case MiddleOperator.Plus:
        resultSymbol = new Symbol(expression.Symbol.Type, value);
        break;

      case MiddleOperator.Minus:
        resultSymbol = new Symbol(expression.Symbol.Type, -value);
        break;

      case MiddleOperator.BitwiseNot:
        resultSymbol = new Symbol(expression.Symbol.Type, ~value);
        break;
    }
```

The final expression does not have short list and long list, since the integral operations do not produce any middle code.

```
      return (new Expression(resultSymbol, null, null));
    }
```

The **ArithmeticFloating** method performs unary addition and subtraction. In case of addition, nothing happens, a new symbol with the same value is created.

```
    private static Expression ArithmeticFloating(MiddleOperator middleOp,
                                                 Expression expression) {
      expression = TypeCast.LogicalToFloating(expression);
      decimal value = (decimal) expression.Symbol.Value;
      Symbol resultSymbol = null;

      switch (middleOp) {
        case MiddleOperator.Plus:
          resultSymbol = new Symbol(expression.Symbol.Type, value);
          break;

        case MiddleOperator.Minus:
          resultSymbol = new Symbol(expression.Symbol.Type, -value);
          break;
      }
```

Since the result is floating value, we push it on the floating value stack.

```
      List<MiddleCode> longList = new List<MiddleCode>();
      MiddleCodeGenerator.AddMiddleCode(longList, MiddleOperator.PushFloat,
                                        resultSymbol);
```

The resulting expression has no short list, and the long list holds only the floating stack push instruction.

```
      return (new Expression(resultSymbol, null, longList));
    }
```

8.1.4. Constant Type Cast

The **ConstantCast** method cast a constant expression into a new type. If the expression if not constant, we return null.

```
    public static Expression ConstantCast(Expression sourceExpression,
                                          Type targetType) {
      if (!IsConstant(sourceExpression)) {
        return null;
      }

      Symbol sourceSymbol = sourceExpression.Symbol, targetSymbol;
      Type sourceType = sourceSymbol.Type;
      object sourceValue = sourceSymbol.Value;
      List<MiddleCode> longList = new List<MiddleCode>();
```

If the source and target types have the same size and they both are either integral, array, or pointers, or they both are floating, we just return a new expression with the target symbol and the short list and long list of the source expression.

```
      if (sourceType.IsIntegralOrPointer() && targetType.IsFloating()) {
        decimal targetValue = ((decimal) ((BigInteger) sourceValue));
        targetSymbol = new Symbol(targetType, targetValue);
      }
      else if (sourceType.IsFloating() && targetType.IsIntegralOrPointer()) {
        BigInteger targetValue = ((BigInteger)((decimal)sourceValue));
        targetSymbol = new Symbol(targetType, targetValue);
      }
```

If the source type is logical and the target type is in integral, array, or pointer. We look into the true and false-sets of the expression.

If the true-set is not empty, the expression is true and we return a new expression with the value one. We do not need to exam the false-set, since we know that the expression is constant, in which case one of the sets is always empty.

If the true-set is empty, we return a new expression with the value zero, since the false-set must be non-empty.

```
      else if (sourceType.IsLogical() && targetType.IsArithmeticOrPointer()) {
        bool isTrue = (sourceSymbol.TrueSet.Count > 0);
        object targetValue;

        if (targetType.IsIntegralOrPointer()) {
          targetValue = isTrue ? BigInteger.One : BigInteger.Zero;
        }
        else {
          targetValue = isTrue ? decimal.One : decimal.Zero;
        }

        targetSymbol = new Symbol(targetType, targetValue);
      }
      else if (sourceType.IsArithmeticOrPointer() && targetType.IsLogical()) {
        bool isTrue = !sourceValue.Equals(BigInteger.Zero) &&
                      !sourceValue.Equals(decimal.Zero);

        MiddleCode gotoCode = new MiddleCode(MiddleOperator.Jump);
        longList.Add(gotoCode);

        ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),
                         falseSet = new HashSet<MiddleCode>();
        if (isTrue) {
          trueSet.Add(gotoCode);
        }
        else {
          falseSet.Add(gotoCode);
        }

        targetSymbol = new Symbol(trueSet, falseSet);
      }
      else {
        targetSymbol = new Symbol(targetType, sourceValue);
      }

      if (targetType.IsFloating()) {
        longList.Add(new MiddleCode(MiddleOperator.PushFloat, targetSymbol));
      }
      return (new Expression(targetSymbol, null, longList));
    }
```

8.1.5. Constant Value

The **Value** method takes a symbol and returns a static symbol.

```
    public static StaticSymbol Value(Symbol symbol) {
      return Value(symbol.UniqueName, symbol.Type, symbol.Value);
    }

    public static StaticSymbol Value(string uniqueName, Type type,
                                     object value) {
      List<MiddleCode> middleCodeList = new List<MiddleCode>();
```

If the value is not null, we add the initializer instruction to the code list.

```
      if (value != null) {
        middleCodeList.Add(new MiddleCode(MiddleOperator.Initializer,
                                          type.Sort, value));
      }
```

If the value is null, we add the initializer ero instruction, with the type size.

```
      else {
        middleCodeList.Add(new MiddleCode(MiddleOperator.InitializerZero,
                                          type.Size()));
      }
```

We translate the middle code instructions to assembly code.

```
      List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();
      AssemblyCodeGenerator.GenerateAssembly(middleCodeList,
                                             assemblyCodeList);
```

For the Linux target machine, we generate a list of text holding the final assembly code. For Windows target code generation, se Chapter 13.

```
      if (Start.Linux) {
        List<string> textList = new List<string>();
        textList.Add("section .data");
        textList.Add("\n" + uniqueName + ":");
        ISet<string> externSet = new HashSet<string>();
        AssemblyCodeGenerator.LinuxTextList(assemblyCodeList, textList,
                                            externSet);
        return (new StaticSymbolLinux(uniqueName, textList, externSet));
      }

      if (Start.Windows) {
        // ...
      }

      return null;
    }
  }
}
```

9. Static Addresses

A static expression is an expression that is located at a specific place in the memory. The address can be identified by its name and offset. In the following example where **i** and **p** are static variables, **p** is static, and the linker eventually decides its value.

```
int i;
int *p = &i;
```

9.1.1. Static Value and Address

The **StaticValue** and **StaticAddress** classes are identical sub classes of **StaticBase**, which holds a name and an offset. In the end, only static addresses are allowed, but static values can hold intermediate values during the parsing. For instance, in the static address **&a[3]**, **a[3]** is temporary stored as a static value.

StaticBase.cs

```
namespace CCompiler {
  public abstract class StaticBase {
    private string m_uniqueName;
    private int m_offset;

    public StaticBase(string name, int offset) {
      m_uniqueName = name;
      m_offset = offset;
    }

    public string UniqueName {
      get { return m_uniqueName; }
    }

    public int Offset {
      get { return m_offset; }
    }

    public override string ToString() {
      if (m_offset > 0) {
        return m_uniqueName + " + " + m_offset;
      }
      else if (m_offset < 0) {
        return m_uniqueName + " - " + (-m_offset);
      }
      else {
        return m_uniqueName;
      }
    }
  }

  public class StaticValue : StaticBase {
    public StaticValue(string name, int offset)
     :base(name, offset) {
      // Empty.
    }
  }
  public class StaticAddress : StaticBase {
    public StaticAddress(string name, int offset)
     :base(name, offset) {
      // Empty.
    }
  }
}
```

9.1.2. Static Expression

The **StaticExpression** class holds the two methods **Binary** and **Unary**, which takes a binary or unary expression and returns a static expression if there is one, or null otherwise.

StaticExpression.cs

```
using System.Numerics;

namespace CCompiler {
  public class StaticExpression {
```

The **Binary** method exams the expressions if the operator is binary addition or subtraction, index, or dot.

```
    public static Expression Binary(MiddleOperator middleOp,
                                    Expression leftExpression,
                                    Expression rightExpression) {
      Type leftType = leftExpression.Symbol.Type,
           rightType = rightExpression.Symbol.Type;
      object leftValue = leftExpression.Symbol.Value,
             rightValue = rightExpression.Symbol.Value;
```

In the addition case, the operand values must be a static address and a constant integer value, or an extern or static array and a constant value. For instance, **&i + 2**, **2 + &i**, **a + 2** or **2 + a**, where **i** an integer and **a** is an array. In case of static address and an integral value on either side, we call **GenerateAddition** to generate the resulting static address.

```
          if ((leftValue is StaticAddress) && // &i + 2
              (rightValue is BigInteger)) {
            return GenerateAddition(leftSymbol, (BigInteger) rightValue);
          }
          else if ((leftValue is BigInteger) && // 2 + &i
                   (rightValue is StaticAddress)){
            return GenerateAddition(rightSymbol, (BigInteger) leftValue);
          }
```

In case of a static or extern array and an integer value on either side, we also call **GenerateAddition**.

```
          else if (leftSymbol.IsExternOrStaticArray() && // a + 2
                   (rightValue is BigInteger)) {
            return GenerateAddition(leftSymbol, (BigInteger)rightValue);
          }
          else if ((leftValue is BigInteger) && // 2 + a
                   rightSymbol.IsExternOrStaticArray()) {
            return GenerateAddition(rightSymbol, (BigInteger) leftValue);
          }
          break;
```

In the subtraction case the left operand must a static address or a static or extern array, and the right operand must be a constant integer value. For instance, **&i - 2** or **a - 2**. Unlik the addition case above, we cannot swap the operands, the **2 - &i** and **2 - a** cases are not allowed.

```
        case MiddleOperator.Subtract:
          if ((leftValue is StaticAddress) && // &i - 2
              (rightValue is BigInteger)) {
            return GenerateAddition(leftSymbol, -((BigInteger) rightValue));
          }
          else if (leftSymbol.IsExternOrStaticArray() && // a - 2
                   (rightValue is BigInteger)) {
            return GenerateAddition(leftSymbol, -((BigInteger) rightValue));
          }
          break;
```

In the index case, the operands must be a static address or a static or extern array, and a constant integer value, on either side. For instance, **&i[2]** and **a[2]** as well as **2[&i]** are **2[a]** are allowed. We call **GenerateIndex** to generate the static address.

```
        case MiddleOperator.Index:
          if ((leftValue is StaticAddress)  && (rightValue is BigInteger)) {
            return GenerateIndex(leftSymbol, (BigInteger) rightValue); // &i[2]
          }
          else if ((leftValue is BigInteger) && (rightValue is StaticAddress)){
            return GenerateIndex(rightSymbol, (BigInteger) leftValue); // 2[&i]
          }
          else if (leftSymbol.IsExternOrStaticArray() &&
                   (rightValue is BigInteger)) {
            return GenerateIndex(leftSymbol, (BigInteger) rightValue); // a[2]
          }
          else if ((leftValue is BigInteger) &&
                   rightSymbol.IsExternOrStaticArray()) {
            return GenerateIndex(rightSymbol, (BigInteger) leftValue); // 2[a]
          }
          break;
```

In the dot case, the operands must be an extern or static struct or union, or a static address. For instance, **s.i** where **s** is a static or extern struct and **i** is one of its members. Note that the resulting value is an object of the **StaticValue** class rather than the **StaticAddress** class.

```
        case MiddleOperator.Dot:
          if (leftSymbol.IsExternOrStatic()) {
            object resultValue =
              new StaticValue(leftSymbol.UniqueName, rightSymbol.Offset);
            Symbol resultSymbol = new Symbol(leftType, resultValue);
            return (new Expression(resultSymbol, null, null));
          }
          break;
      }

      return null;
    }
```

The **GenerateAddition** method generates a static address for an addition expression.

```
    private static Expression GenerateAddition(Symbol symbol,
                                               BigInteger value) {
      int offset = ((int) value) * symbol.Type.PointerOrArrayType.Size();
      StaticAddress resultValue;

      if (symbol.Value is StaticAddress) {
        StaticAddress staticAddress = (StaticAddress) symbol.Value;
        resultValue = new StaticAddress(staticAddress.UniqueName,
                                        staticAddress.Offset + offset);
      }
      else {
        resultValue = new StaticAddress(symbol.UniqueName, offset);
      }

      Symbol resultSymbol = new Symbol(symbol.Type, resultValue);
      return (new Expression(resultSymbol, null, null));
    }
```

The **GenerateIndex** method generates the static address for an index expression.

```
    private static Expression GenerateIndex(Symbol symbol,
                                            BigInteger value) {
      int offset = ((int) value) * symbol.Type.ArrayType.Size();
      StaticValue resultValue;

      if (symbol.Value is StaticAddress) {
        StaticAddress staticAddress = (StaticAddress) symbol.Value;
        resultValue = new StaticValue(staticAddress.UniqueName,
                                      staticAddress.Offset + offset);
      }
      else {
        resultValue = new StaticValue(symbol.UniqueName, offset);
      }

      Symbol resultSymbol = new Symbol(symbol.Type, resultValue);
      return (new Expression(resultSymbol, null, null));
    }
```

Finally, we have to unary case. There is only one relevant operator: the address operator ('&').

```
    public static Expression Unary(MiddleOperator middleOp,
                                   Expression expression) {
      Symbol symbol = expression.Symbol;
```

If the symbol of the address operator is a static value, we create a static address with the same name and offset. For instance, **&a[i]** or **&s.i**.

```
      if (middleOp == MiddleOperator.Address) {
        if (symbol.Value is StaticValue) { // &a[i], &s.i
          StaticValue staticValue = (StaticValue) symbol.Value;
          StaticAddress staticAddress =
            new StaticAddress(staticValue.UniqueName, staticValue.Offset);
          Symbol resultSymbol =
            new Symbol(new Type(symbol.Type), staticAddress);
          return (new Expression(resultSymbol, null, null));
        }
```

If the symbol is not a static value, but holds extern or static storage, we create a static address, with the symbol name and offset zero. For instance, **&i**, where **i** holds static or extern storage.

```
        else if (symbol.IsExternOrStatic()) {
          StaticAddress staticAddress =
            new StaticAddress(symbol.UniqueName, 0);
          Symbol resultSymbol =
            new Symbol(new Type(symbol.Type), staticAddress);
          return (new Expression(resultSymbol, null, null));
        }
      }

      return null;
    }
  }
}
```

10. Initialization

In C, it is possible to initialize simple and compound variables. Therefore, we need to check that the initialized value has the correct type. There are two kinds of initialization: static and auto. They perform the same task: matching a type against an initializer. However, the result of a static initialization is a memory block while the result of an auto initialization is a sequence of assignment instructions. The initializer may be an expression or a recursive list of expressions; that is, the list may hold sub lists.

There are several cases to consider:

Variable	**Initializer**	**Example**
Pointer	Address	`static int array[3];` `static int* p = &a[2];`
Pointer to signed or unsigned char	String	`static char *p = "Hello";`
Pointer	List	`int *p = {1,2,3};`
Array	String	`char s[] = "World";`
Array	List	`Char st[] = {'a', 'b', 'c'};`
Struct	List	
Union	List	
Integral or Pointer		
Floating		

If the variable is a pointer and the initializer is an address we add its offset to the block and store the name of the address in the access map, the address value will later be looked up and added by the linker.

If the variable type is a pointer to a (signed or unsigned) character and the initializator is a string, the address is stored in the access map and a zero address is stored in the block, it will later be properly looked up and set by the linker.

10.1.1. Auto Initialization

Auto initialization occurs when an auto or register variable becomes initialized. Since the initialization value can be non-constant, no memory block is created. Instead, a sequence of assignment instructions is added to the middle code.

GenerateAutoInitializer.cs

```
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  class GenerateAutoInitializer {
    public static List<MiddleCode> GenerateAuto(Symbol toSymbol,
                                            object fromInitializer) {
      Assert.ErrorXXX((fromInitializer is Expression) ||
                      (fromInitializer is List<object>));
      Type toType = toSymbol.Type;
      List<MiddleCode> codeList = new List<MiddleCode>();
```

If the initializer is an expression, we have two cases. If the type is an array of characters and the initializer is a string, we change the initializer from a string to a list of characters, and call **GenerateAutoInitializer** recursively with the list of characters instream of the string. In C, when initializing an array of characters, a string or a list of characters are equivalent. For instance, the initializations **char s[] = "Hi"** and **char s[] = {'H', 'i', '\0'}** are equivalent. Note the terminating zero-character is added implicitly and the string case.

```
      if (fromInitializer is Expression) {
        Expression fromExpression = (Expression) fromInitializer;

        if (toType.IsArray() && toType.ArrayType.IsChar() &&
            fromExpression.Symbol.Type.IsString()) {
          string text = ((string) fromExpression.Symbol.Value) + "\0";
          List<object> list = new List<object>();

          foreach (char c in text) {
            Symbol charSymbol =
              new Symbol(toType.ArrayType, (BigInteger) ((int) c));
            Expression charExpression = new Expression(charSymbol, null, null);
            list.Add(charExpression);
          }

          return GenerateAuto(toSymbol, list);
        }
```

In all other cases, we type cast the expression into the defined type and generate middle code instructions. In case of a floating value, we pop the value from the floating value stack to the symbol to be initialized. In case of any other type we generate an assignment instruction.

```
        else {
          fromExpression = TypeCast.ImplicitCast(fromExpression, toType);
          codeList.AddRange(fromExpression.LongList);

          if (toSymbol.Type.IsFloating()) {
            codeList.Add(new MiddleCode(MiddleOperator.PopFloat, toSymbol));
          }
          else {
            if (fromExpression.Symbol.Type.IsStructOrUnion()) {
              codeList.Add(new MiddleCode(MiddleOperator.AssignInit,
                                          toSymbol, fromExpression.Symbol));
            }

            codeList.Add(new MiddleCode(MiddleOperator.Assign, toSymbol,
                                        fromExpression.Symbol));
          }
        }
      }
```

If the initializer is not an expression it must be a list of expressions or other lists, recursively. In that case the type must be an array, a struct, or a union, since only arrays, structs, and unions can be initialized with a list.

```
      else {
        Assert.Error(toType.IsArray() ||toType.IsStructOrUnion(),
                     toType, Message.
            Only_array_struct_or_union_can_be_initialized_by_a_list);
        List<object> fromList = (List<object>) fromInitializer;
```

If the type is an array, we set its size if it has not yet been defined or check that the list size does not exceed the array size if the size has been defined. Note that in the case of an array we do not add a terminating zero character.

```
        switch (toType.Sort) {
          case Sort.Array: {
              fromList = ModifyInitializer.ModifyArray(toType, fromList);

              if (toType.ArraySize == 0) {
                toType.ArraySize = fromList.Count;
              }
              else {
                Assert.Error(fromList.Count <= toType.ArraySize,
                             toType, Message.Too_many_initializers);
              }
```

We iterate through the list and, for each element in the list, define a symbol for the array index and call **GenerateAuto** recursively, so that potential sub list are properly processed. If the list does not cover the whole array, the remaining part of the array remains uninitialized. The offset of each index symbol is sum of the offset of the symbol and the offset of the index symbol. Note that we must multiply the index of the array with the size of the array type to determine the correct index. However, it is not strictly necessary give the index symbol a name. That is for readability reasons only.

```
              for (int index = 0; index < fromList.Count; ++index) {
                Symbol indexSymbol = new Symbol(toType.ArrayType);
                indexSymbol.Offset = toSymbol.Offset +
                                     (index * toType.ArrayType.Size());
                indexSymbol.Name = toSymbol.Name + "[" + index + "]";
                codeList.AddRange(GenerateAuto(indexSymbol, fromList[index]));
              }
            }
            break;
```

In case of a struct or a union, we check that the list size does not exceed its allowed size. In case of a struct, the list size must not exceed the number of struct members. In case of a union, the list must hold exactly on element.

```
          case Sort.Struct:
          case Sort.Union: {
              List<Symbol> memberList = toType.MemberList;
              Assert.Error((toType.IsStruct() &&
                           (fromList.Count <= memberList.Count)) ||
                           (toType.IsUnion() && (fromList.Count == 1)),
                           toType, Message.Too_many_initializers);
```

Like the array case, we iterate through the list and, for each element in the list, create a symbol for the member and call **GenerateAuto** recursively, so that potential sub lists are properly processed. The offset of each member symbol is the sum of the offset of the struct or union and the member.

```
              IEnumerator<Symbol> enumerator = memberList.GetEnumerator();
              foreach (object fromInitializor in fromList) {
                enumerator.MoveNext();
                Symbol memberSymbol = enumerator.Current;
                Symbol subSymbol = new Symbol(memberSymbol.Type);
                subSymbol.Name = toSymbol.Name + "." + memberSymbol.Name;
                subSymbol.Offset = toSymbol.Offset + memberSymbol.Offset;
                codeList.AddRange(GenerateAuto(subSymbol, fromInitializor));
              }
            }
            break;
        }
      }

      return codeList;
    }
  }
}
```

10.1.1. Static Initialization

Static initialization occurs when a static variable becomes initialized. The initialization value has to be constant and known at compile time. No code is generated, instead a memory block holding the value is created.

The **GenerateStaticInitializer** class basically performs the same task as the **GenerateAutoInitializer** class above: it interprets the initialization and compare it to the defined type. However, it does not generate assignment instruction, but rather initialization instructions. One difference compared to the auto initialization case is that if the array, struct, or union is not completely initialized we need add zeros for the remaining part.

GenerateStaticInitializer.cs
```
using System.Collections.Generic;

namespace CCompiler {
  public class GenerateStaticInitializer {
    public static List<MiddleCode> GenerateStatic(Type toType,
                                               object fromInitializer) {
      Assert.ErrorXXX((fromInitializer is Expression) ||
                    (fromInitializer is List<object>));
      List<MiddleCode> codeList = new List<MiddleCode>();
```

If the initializer is an expression, we have two special cases: that the defined type is a character array and the initializer is a string, the defined type is a pointer and the initializer is an array, a function, or a string.

```
      if (fromInitializer is Expression) {
        Expression fromExpression = (Expression) fromInitializer;
        Symbol fromSymbol = fromExpression.Symbol;
        Assert.Error(fromSymbol.IsExternOrStatic(), fromSymbol,
                     Message.Non__static_initializer);
        Type fromType = fromSymbol.Type;
```

If the defined type is a character array and the initializer is a string, we first check the size of the string. If the array size is undefined, we set its size to the size of the string, plus one for the terminating zero-character. If the array size is defined, we check that the string (including the terminating zero-character) fits withing the array.

```
        if (toType.IsArray() && toType.ArrayType.IsChar() &&
            fromType.IsString()) {
```

```
          string text = (string) fromSymbol.Value;
```

Note that we add space for an extra character, the terminating zero-character.

```
          if (toType.ArraySize == 0) {
            toType.ArraySize = text.Length + 1;
          }
```

Note that the string length must be strictly less than the array size, for the terminating zero-character to fit in the string.

```
          else {
            Assert.Error(text.Length < toType.ArraySize,
                         toType, Message.Too_many_initializers);
          }

          codeList.Add(new MiddleCode(MiddleOperator.Initializer,
                                      fromSymbol.Type.Sort, text));
        }
```

If the defined type is a pointer and the initializer is an array, a function, or a string, we create a static address and add an initialization middle code instruction.

```
        else if (toType.IsPointer() && fromType.IsArrayFunctionOrString()) {
          Assert.ErrorXXX((fromType.IsString() && toType.PointerType.IsChar())
                        ||(fromType.IsArray() &&
                           fromType.ArrayType.Equals(toType.PointerType)) ||
                        (fromType.IsFunction() &&
                         fromType.Equals(toType.PointerType)));
          StaticAddress staticAddress =
            new StaticAddress(fromSymbol.UniqueName, 0);
          codeList.Add(new MiddleCode(MiddleOperator.Initializer,
                                      toType.Sort, staticAddress));
        }
```

In all other cases, we type cast the expression into the defined type, check that the expression is constant, and add an initialization middle code instruction.

```
        else {
          Expression toExpression =
            TypeCast.ImplicitCast(fromExpression, toType);
          Symbol toSymbol = toExpression.Symbol;
```

If the value is not null, the expression is constant. However, the value may be a static address, which we consider to be constant in this context.

```
          Assert.Error(toSymbol.Value != null, toSymbol,
                       Message.Non__constant_expression);
          codeList.Add(new MiddleCode(MiddleOperator.Initializer,
                                      toSymbol.Type.Sort, toSymbol.Value));
        }
      }
```

If the initializer is a list, the defined type must be an array, a struct, or a union.

```
      else {
        Assert.Error(toType.IsArray() || toType.IsStructOrUnion(),
                     toType, Message.
                    Only_array_struct_or_union_can_be_initialized_by_a_list);
        List<object> fromList = (List<object>) fromInitializer;
```

In case of an array, we set the array size to the list size if undefined. If the array is defined, we check that the list size does not exceed the array size.

```
switch (toType.Sort) {
  case Sort.Array: {
      fromList = ModifyInitializer.ModifyArray(toType, fromList);

      if (toType.ArraySize == 0) {
        toType.ArraySize = fromList.Count;
      }
      else {
        Assert.Error(fromList.Count <= toType.ArraySize,
                     toType, Message.Too_many_initializers);
      }
```

We iterate the array values and call **GenerateStatic** for each value. In the static case we do not need to create index symbol since the code to be generated are initializations rather than assignments.

```
      foreach (object value in fromList) {
        codeList.AddRange(GenerateStatic(toType.ArrayType, value));
      }
```

However, in the static case we must add an instruction for the initialization of the potential remaining part of the array. The **InitializerZero** instruction adds a sequence of zero-bytes.

```
      int restSize = toType.Size() -
                     (fromList.Count * toType.ArrayType.Size());
      if (restSize > 0) {
        codeList.Add(new MiddleCode(MiddleOperator.InitializerZero,
                                    restSize));
      }
    }
    break;
```

Like the auto case, the initializer list size must not exceed the number of members in a struct. The initializer list must hold exactly one element in case of a union.

```
  case Sort.Struct:
  case Sort.Union: {
      List<Symbol> memberList = toType.MemberList;
      Assert.Error((toType.IsStruct() &&
                   (fromList.Count <= memberList.Count)) ||
                   (toType.IsUnion() && (fromList.Count == 1)),
                   toType, Message.Too_many_initializers);
```

Like the auto case, we iterate through the initialization list and call **GenerateStatic** recursively for each element in the list. However, in the static case we must sum the size of the initializer list in order to add zero-bytes if the list does not cover the struct or union.

```
      int size = 0;
      IEnumerator<Symbol> enumerator = memberList.GetEnumerator();
      foreach (object fromInitializor in fromList) {
        enumerator.MoveNext();
        Symbol memberSymbol = enumerator.Current;
        codeList.AddRange(GenerateStatic(memberSymbol.Type,
                                         fromInitializor));
        size += memberSymbol.Type.Size();
      }
```

```
            int restSize = toType.Size() - size;
            if (restSize > 0) {
              codeList.Add(new MiddleCode(MiddleOperator.InitializerZero,
                                          restSize));
            }
          }
          break;
      }
    }

    return codeList;
  }
 }
}
```

10.1.2. Modify Initializer

The **ModifyInitializer** class changes an initialization list into the form of the defined type. For instance: **int [3][2] = {1, 2, 3, 4, 5, 6}** is changed to **int [3][2] = {{1, 2}, {3, 4}, {5, 6}}**.

ModifyInitializer.cs

```
using System;
using System.IO;
using System.Collections.Generic;

namespace CCompiler {
  class ModifyInitializer {
    public static List<object> ModifyArray(Type type, List<object> list) {
```

First, we define the dimension-to-size map; that is, each dimension in the defined array is assigned an array size. For instance, in **int [2][3][4]** dimension 1 has size 4, dimension 2 has size 3, and dimension 3 has size 2. Dimension zero refers to the final array type, and its size is always zero. In **int [][3][4]**, dimension 3 is undefined and set to zero, but that does not matter since we do not exam the highest dimension.

```
      IDictionary<int,int> dimensionToSizeMap = new Dictionary<int,int>();
      int maxDimension = DimensionToSizeMap(type, dimensionToSizeMap);
```

Then we define the initializer-to-dimension map; that is, each list and sub list in the initializer is assigned a dimension.

```
      IDictionary<object,int> initializerToDimensionMap =
        new Dictionary<object,int>();
      InitializerToDimensionMap(list, initializerToDimensionMap);
```

Then we iterate through the dimensions of the array, from dimension one to the second-highest dimension, inclusive. For each dimension we define the total list, where each element holds the current dimension.

```
      for (int dimension = 1; dimension < maxDimension; ++dimension) {
        List<object> totalList =
          new List<object>(), currentList = new List<object>();
        int arraySize = dimensionToSizeMap[dimension];
        Assert.ErrorXXX(arraySize > 0);

        foreach (object member in list) {
          if (initializerToDimensionMap[member] < dimension) {
            currentList.Add(member);
          }
          else {
```

```
        if (currentList.Count > 0) {
          initializerToDimensionMap[currentList] = dimension;
          totalList.Add(currentList);
        }

        totalList.Add(member);
        currentList = new List<object>();
      }

      if (currentList.Count == arraySize) {
        initializerToDimensionMap[currentList] = dimension;
        totalList.Add(currentList);
        currentList = new List<object>();
      }
    }

    if (currentList.Count > 0) {
      initializerToDimensionMap[currentList] = dimension;
      totalList.Add(currentList);
    }

    list = totalList;
  }

  return list;
}
```

The **DimensionToSizeMap** method assigns the array size of each dimension in the nested array.

```
private static int DimensionToSizeMap(Type type,
                              IDictionary<int,int> dimensionToSizeMap) {
  if (type.IsArray()) {
    int dimension =
      DimensionToSizeMap(type.ArrayType, dimensionToSizeMap) + 1;
    dimensionToSizeMap[dimension] = type.ArraySize;
  }

  return 0;
}
```

The **InitializerToDimensionMap** method assigns the dimension to each list in the nested list.

```
private static int InitializerToDimensionMap(object initializer,
                      IDictionary<object,int> initializerToDimensionMap) {
  if (initializer is List<object>) {
    List<object> list = (List<object>) initializer;
    int maxDimension = 0;
```

If the initializer is a list, we iterate through the list, and assign the list the dimension of the sub list with the highest dimension, plus one.

```
    foreach (object member in list) {
      int dimension =
        InitializerToDimensionMap(member, initializerToDimensionMap);
      maxDimension = Math.Max(maxDimension, dimension);
    }

    initializerToDimensionMap[list] = maxDimension + 1;
    return maxDimension + 1;
```

```
      }

      return 0;
    }
  }
}
```

11. Middle Code Optimization

When the middle code has been generated, we need to perform several optimizations since the code may be ineffective. Some parts may be introduced by the programmer, while other parts are introduced by the parser. Since one optimization may reveal new optimization opportunities, we need to repeat the optimizations until we do not detect any more opportunities. The field **m_update** is set to true if we find an optimization opportunity.

MiddleCodeOptimizor.cs

```
using System.Numerics;
using System.Collections.Generic;
```

The **m_update** field is set to true each time an optimization has occurred, and the middle code list is stored in **m_middleCodeList**.

```
namespace CCompiler {
  public class MiddleCodeOptimizer {
    private bool m_update;
    private List<MiddleCode> m_middleCodeList;

    public MiddleCodeOptimizer(List<MiddleCode> middleCodeList) {
      m_middleCodeList = middleCodeList;
    }
```

First of all, we need to change the addresses of jump instructions, from middle code objects to their index in the code list, by calling **ObjectToIntegerAddresses**.

```
    public void Optimize() {
      ObjectToIntegerAddresses();
```

Then follows the main loop of the optimization process. We continue to iterate as long as there are optimization possibilities; that is, as long as **m_update** is true. The **m_update** field is set to false at the beginning and will be set to true in case of an optimization.

```
      do {
        m_update = false;
        ClearGotoNextStatements();
        ClearDoubleRelationStatements();
        TraceGotoChains();
        ClearUnreachableCode();
        RemovePushPop();
        MergePopPushToTop();
        MergeTopPopToPop();
        MergeBinary();
        //MergeDoubleAssign(); // XXX
        SematicOptimization();
        OptimizeRelation();
        OptimizeCommutative();
        OptimizeBinary();
        CheckIntegral(); // XXX
        CheckFloating(); // XXX
        RemoveClearedCode();
      } while (m_update);
    }
```

11.1.1. Object to Integer Addresses

When we generated the middle code, we use middle code objects as target for jump instructions. The first we need to do is to change the middle code targets to integer targets. The reason for this is that the optimization process of this chapter will result in the removal of middle code instruction, among which some may be jump targets.

```
    public void ObjectToIntegerAddresses() {
```

We start by defining the address map, which we load with the index of each instruction.

```
      IDictionary<MiddleCode,int> addressMap =
        new Dictionary<MiddleCode,int>();

      for (int index = 0; index < m_middleCodeList.Count; ++index) {
        addressMap.Add(m_middleCodeList[index], index);
      }
```

We then iterate through the middle code list and, with the help of the address map, change the targets from middle code instructions to integer values.

```
      for (int index = 0; index < m_middleCodeList.Count; ++index) {
        MiddleCode sourceCode = m_middleCodeList[index];

        if (sourceCode.IsGoto() || sourceCode.IsCarry() ||
            sourceCode.IsRelation()) {
          Assert.ErrorXXX(sourceCode[0] is MiddleCode);
          MiddleCode targetCode = (MiddleCode) sourceCode[0];
          Assert.ErrorXXX(addressMap.ContainsKey(targetCode));
          sourceCode[0] = addressMap[targetCode];
        }
      }
    }
```

11.1.2. Jump Next Instructions

In some cases, there may be a jump instruction that just jumps to the next instruction. Those instructions are meaningless and shall be removed.

```
  1. goto 2
  2. ...

    private void ClearGotoNextStatements() {
      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode middleCode = m_middleCodeList[index];

        if (middleCode.IsRelationCarryOrGoto()) {
          int target = (int) middleCode[0];
```

When we encounter a jump instruction that jumps to the next instruction, we remove it

```
          if (target == (index + 1)) {
            middleCode.Clear();
            m_update = true;
          }
```

```
      }
    }
  }
```

11.1.3. Next-Double Jump Statements

A conditional jump instruction that jumps two steps ahead and is followed by an unconditional jump instruction can be modified in that we reverse the condition and the target.

(a) Before optimization	(b) After optimization
`1. if a < b goto 3`	`1. if a >= b goto 10`
`2. goto 10`	`2. `*`removed`*
`3. ...`	`3. ...`

To begin with, we need the inverse map to change the condition.

```
  public static IDictionary<MiddleOperator, MiddleOperator> m_inverseMap =
    new Dictionary<MiddleOperator, MiddleOperator>() {
      {MiddleOperator.Equal, MiddleOperator.NotEqual},
      {MiddleOperator.NotEqual, MiddleOperator.Equal},
      {MiddleOperator.Carry, MiddleOperator.NotCarry},
      {MiddleOperator.NotCarry, MiddleOperator.Carry},
      {MiddleOperator.LessThan, MiddleOperator.GreaterThanEqual},
      {MiddleOperator.LessThanEqual, MiddleOperator.GreaterThan},
      {MiddleOperator.GreaterThan, MiddleOperator.LessThanEqual},
      {MiddleOperator.GreaterThanEqual, MiddleOperator.LessThan}
    };
```

We iterate through the middle code list, from the first to the next-to-last instruction since we inspect the current instruction and the instruction following it.

```
  private void ClearDoubleRelationStatements() {
    for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
      MiddleCode thisCode = m_middleCodeList[index],
                 nextCode = m_middleCodeList[index + 1];

      if ((thisCode.IsRelation() || thisCode.IsCarry()) &&
          nextCode.IsGoto()) {
        int target1 = (int) thisCode[0],
            target2 = (int) nextCode[0];
```

If the instruction jumps over the next instruction, we use the inverse map to change the current instruction, and we clear the next instruction.

```
        if (target1 == (index + 2)) {
          MiddleOperator operator1 = thisCode.Operator;
          thisCode.Operator = m_inverseMap[operator1];
          thisCode[0] = target2;
          nextCode.Clear();
          m_update = true;
        }
      }
    }
  }
```

11.1.4. Jump-Chains

A jump chain is a sequence of unconditional jump instructions where each instruction in the chain, except the last one, jumps to another unconditional jump instruction. It would be more effective if they all jump to the same target as the last jump instruction.

```
1. goto 3                 1. goto 7
2. ..                     2. ..
3. goto 5                 3. goto 7
4. ...                    4. ...
5. goto 7                 5. goto 7
6. ...                    6. ...
7. a = 1                  7. a = 1
```

(a) Before optimization (b) After optimization

First, we trace all the unconditional jump instructions by calling **TraceGotoChains**, that in turn calls **TraceGoto**. Then we change all the jump targets.

```
private void TraceGotoChains() {
  for (int index = 1; index < m_middleCodeList.Count; ++index) {
    MiddleCode middleCode = m_middleCodeList[index];

    if (middleCode.IsRelationCarryOrGoto()) {
      ISet<int> sourceSet = new HashSet<int>();
      sourceSet.Add(index);
```

For each jump instruction we trace the jump-chain and if the first and last jump instructions differs, we replace all the jumps in the chain with the last jump.

```
      int firstTarget = (int) middleCode[0];
      int finalTarget = TraceGoto(firstTarget, sourceSet);

      if (firstTarget != finalTarget) {
        foreach (int source in sourceSet) {
          MiddleCode sourceCode = m_middleCodeList[source];
          sourceCode[0] = finalTarget;
        }

        m_update = true;
      }
    }
  }
}
```

The **TraceGoto** method follows the jump instructions as long as they continue to jump to new targets.

```
private int TraceGoto(int target, ISet<int> sourceSet) {
  MiddleCode objectCode = m_middleCodeList[target];

  if (!sourceSet.Contains(target) && objectCode.IsGoto()) {
    sourceSet.Add(target);
    int nextTarget = (int) objectCode[0];
    return TraceGoto(nextTarget, sourceSet);
  }
  else {
    return target;
  }
```

```
    }
```

11.1.5. Remove Unreachable Code

Code that is not reachable from the first instruction of the function shall be removed. We call **SearchReachableCode** that follows all conditional and unconditional jumps instructions and save their line numbers in the visited set. We also report an error if we reach the last instruction of a function that do not return void.

(a) C code	(b) Before optimization	(b) After optimization
`if (1)`	`1. goto 1`	`1.` *`removed`*
`  a = 1;`	`2. a = 1`	`2. a = 1`
`else`	`3. goto 5`	`3.` *`removed`*
`  b = 2;`	`4. b = 2`	`4.` *`removed`*
	`5. ...`	`5. ...`

```
    private void ClearUnreachableCode() {
      ISet<MiddleCode> visitedSet = new HashSet<MiddleCode>();
      SearchReachableCode(0, visitedSet);

      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode middleCode = m_middleCodeList[index];
        if (!visitedSet.Contains(middleCode)) {
          m_middleCodeList[index].Clear();
          m_update = true;
        }
      }
    }

    private void SearchReachableCode(int index, ISet<MiddleCode> visitedSet) {
      for (; index < m_middleCodeList.Count; ++index) {
        MiddleCode middleCode = m_middleCodeList[index];

        if (visitedSet.Contains(middleCode)) {
          return;
        }

        visitedSet.Add(middleCode);

        if (middleCode.IsRelation() || middleCode.IsCarry()) {
          int target = (int) middleCode[0];
          SearchReachableCode(target, visitedSet);
        }
        else if (middleCode.IsGoto()) {
          int target = (int) middleCode[0];
          SearchReachableCode(target, visitedSet);
          return;
        }
        else if (middleCode.Operator == MiddleOperator.Return) {
          if (m_middleCodeList[index + 1].Operator == MiddleOperator.Exit) {
            visitedSet.Add(m_middleCodeList[index + 1]);
          }

          return;
        }
        else if (middleCode.Operator == MiddleOperator.FunctionEnd) {
          Symbol funcSymbol = (Symbol) middleCode[0];
          Assert.Error(funcSymbol.Type.ReturnType.IsVoid(),
                       funcSymbol.Name,
                       Message.Reached_the_end_of_a_non__void_function);
          return;
        }
      }
    }
```

11.1.1. Remove Push-Pop Chains

Sometimes there may be a push followed by a pop of the same symbol, or no symbol at all, which in meaningless and shall be removed.

```
1. push x              1. removed
2. pop x               2. removed

1. push x              1. removed
2. pop                 2. removed
```

However, we do not remove code where different symbols are pushed and popped.

```
1. push x
2. pop y
```

```
    public void RemovePushPop() {
      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode thisCode = m_middleCodeList[index],
                   nextCode = m_middleCodeList[index + 1];
        if ((thisCode.Operator == MiddleOperator.PushFloat) &&
            (nextCode.Operator == MiddleOperator.PopFloat) &&
            ((thisCode[0] == nextCode[0]) || (nextCode[0] == null))) {
          thisCode.Clear();
          nextCode.Clear();
          m_update = true;
        }
      }
    }
```

11.1.1. Merge Pop-Push Chains

Sometimes there may be a pop followed by a push of the same symbol, which shall be replaced by a top.

```
1. pop x               1. top x
2. push x              2. removed
```

```
    public void MergePopPushToTop() {
      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode thisCode = m_middleCodeList[index],
                   nextCode = m_middleCodeList[index + 1];

        if ((thisCode.Operator == MiddleOperator.PopFloat) &&
            (nextCode.Operator == MiddleOperator.PushFloat) &&
            (thisCode[0] == nextCode[0])) {
          thisCode.Operator = MiddleOperator.TopFloat;
          nextCode.Clear();
          m_update = true;
        }
      }
    }
```

11.1.2. Change Top-Pop to Pop

Sometimes there may be a push followed by a pop of the same symbol, or no symbol at all, which in meaningless and shall be removed.

```
1. top x                  1. pop x
2. pop                    2. empty
```

```
    public void MergeTopPopToPop() {
      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode thisCode = m_middleCodeList[index],
                   nextCode = m_middleCodeList[index + 1];

        if ((thisCode.Operator == MiddleOperator.TopFloat) &&
            (nextCode.Operator == MiddleOperator.PopFloat) &&
            (nextCode[0] == null)) {
          thisCode.Operator = MiddleOperator.PopFloat;
          nextCode.Clear();
          m_update = true;
        }
      }
    }
```

11.1.3. Merge Binary

When a variable I assigned a binary expression, the resulting middle code is two instructions where the first instruction assign a temporary variable to the expression, and then assign the temporary variable to the resulting variable. We restore the expression to its original form, which allows for generation of more effective assembly code in Chapter 12.

```
a = b + c;                1. t0 = b + c             1. a = b + c
                          2. a = t0                 2. removed
```

(a) C code (b) Before optimization (b) After optimization

```
    private void MergeBinary() {
      for (int index = 0; index < (m_middleCodeList.Count - 1); ++index) {
        MiddleCode thisCode = m_middleCodeList[index],
                   nextCode = m_middleCodeList[index + 1];

        if ((/*thisCode.IsUnary() ||*/ thisCode.IsBinary()) &&
            (nextCode.Operator == MiddleOperator.Assign) &&
            ((Symbol) thisCode[0]).IsTemporary() &&
            (thisCode[0] == nextCode[1])) {
          thisCode[0] = nextCode[0];
          nextCode.Clear();
          m_update = true;
        }
      }
    }
```

11.1.4. Semantic Optimization

The semantic optimization simplifies expression such add addition of zero or multiplication of one, it handles a number of situations:

Category	**Example**	**Result**
Constant Expression	2 * 3	6
Binary Addtition	0 + i i + 0	i i
Binary Subtraction	0 - i i - 0	-i i
Binary Multiplication	0 * i 1 * i i * 0 i * 1	0 i 0 i
Binary Division and Modulo	i / 1 i % 1	i i

```
private void SematicOptimization() {
  for (int index = 0; index < m_middleCodeList.Count; ++index) {
    MiddleCode thisCode = m_middleCodeList[index];

    if (thisCode.IsBinary()) {
      Symbol resultSymbol = (Symbol) thisCode[0],
             leftSymbol = (Symbol) thisCode[1],
             rightSymbol = (Symbol) thisCode[2],
             newSymbol = null;
```

Even though the **ConstantExpression** class evaluates all constant expressions caused by the programmer, the pointer arithmetic during the parsing may have cause more constant expressions, expressions where both operands are constant values. In that case we call **ArithmeticIntegral** in **ConstantExpression** to evaluate the value of the expression.

```
      if ((leftSymbol.Value is BigInteger) && // t0 = 2 * 3
          (rightSymbol.Value is BigInteger)) {
        newSymbol =
          ConstantExpression.ArithmeticIntegral(thisCode.Operator,
                                                leftSymbol, rightSymbol);
      }
```

In case of additions where the left operand is zero, 0 + i, we keep the right operand.

```
      // t0 = 0 + i
      else if ((thisCode.Operator == MiddleOperator.Add) &&
                  (leftSymbol.Value is BigInteger) &&
                  (leftSymbol.Value.Equals(BigInteger.Zero))) {
        newSymbol = rightSymbol;
      }
      // t0 = 0 - i; t0 = -i
      else if ((thisCode.Operator == MiddleOperator.Subtract) &&
                  (leftSymbol.Value is BigInteger) &&
                  (leftSymbol.Value.Equals(BigInteger.Zero))) {
        thisCode.Operator = MiddleOperator.Minus;
        thisCode[0] = thisCode[1];
        thisCode[1] = null;
      }
```

In case of additions or subtractions where the right operand is zero, i + 0 or i - 0, we keep the left operand.

```
    // t0 = i + 0
    // t0 = i - 0
    else if (((thisCode.Operator == MiddleOperator.Add) ||
                (thisCode.Operator == MiddleOperator.Subtract)) &&
                (rightSymbol.Value is BigInteger) &&
                (rightSymbol.Value.Equals(BigInteger.Zero))) {
      newSymbol = leftSymbol;
    }
```

In case of multiplications where the left operand is zero, the result is zero.

```
    // t0 = 0 * i
    else if ((thisCode.Operator == MiddleOperator.Multiply) &&
                (leftSymbol.Value is BigInteger) &&
                (leftSymbol.Value.Equals(BigInteger.Zero))) {
      newSymbol = new Symbol(resultSymbol.Type, BigInteger.Zero);
    }
```

In case of multiplications where the left operand is one, we keep the right operand.

```
    // t0 = 1 * i
    else if ((thisCode.Operator == MiddleOperator.Multiply) &&
                (leftSymbol.Value is BigInteger) &&
                (leftSymbol.Value.Equals(BigInteger.One))) {
      newSymbol = rightSymbol;
    }
```

In case of multiplications where the right operand is zero, the result is zero.

```
    // t0 = i * 0
    else if ((thisCode.Operator == MiddleOperator.Multiply) &&
                (rightSymbol.Value is BigInteger) &&
                (rightSymbol.Value.Equals(BigInteger.Zero))) {
      newSymbol = new Symbol(resultSymbol.Type, BigInteger.Zero);
    }
```

In case of multiplications or division where the right operand is one, we keep the left operand.

```
    // t0 = i * 1
    // t0 = i / 1
    else if (((thisCode.Operator == MiddleOperator.Multiply) ||
                (thisCode.Operator == MiddleOperator.Divide) ||
                (thisCode.Operator == MiddleOperator.Modulo)) &&
                (rightSymbol.Value is BigInteger) &&
                (rightSymbol.Value.Equals(BigInteger.One))) {
```

```
          newSymbol = leftSymbol;
        }
```

If the new symbol is not null, we replace the expression with the new symbol. If the result symbol is a temporary symbol. We iterate until we find the place where the result symbol is accessed and replace the result symbol with the new symbol at that place.

```
t1 = i + 0                  j = i
j = t1                      k = i
k = t1
```

(a) Before optimization (b) After optimization

```
        if (newSymbol != null) {
          if (resultSymbol.IsTemporary()) {
            thisCode.Operator = MiddleOperator.Empty;

            if (newSymbol.IsTemporary()) {
              int index2;
              for (index2 = (index - 1); index2 >= 0; --index2) {
                MiddleCode previousCode = m_middleCodeList[index2];

                if (previousCode[0] == resultSymbol) {
                  previousCode[0] = newSymbol;
                  break;
                }
              }
            }

            { int index2;
              for (index2 = index + 1; index2 < m_middleCodeList.Count;
                   ++index2) {
                MiddleCode nextCode = m_middleCodeList[index2];

                if (nextCode[0] == resultSymbol) {
                  nextCode[0] = newSymbol;
                }

                if (nextCode[1] == resultSymbol) {
                  nextCode[1] = newSymbol;
                }

                if (nextCode[2] == resultSymbol) {
                  nextCode[2] = newSymbol;
                }
              }
            }
          }
```

If the result symbol is not temporary symbol, we change the instruction into an assignment, where the result symbol is assigned to the new symbol. For instance, we change **i** = **0** + **j** to **i** = **j**.

```
          else {
            thisCode.Operator = MiddleOperator.Assign; // i = 0 + j;
            thisCode[1] = newSymbol;                   // i = j;
            thisCode[2] = null;
          }
```

```
            m_update = true;
          }
        }
      }
    }
```

11.1.5. Optimize Relation Expression

In order to optimize the final assembly code generation, we shall swap the operator in expression where the left operand is a value. This operation does not decrease the number of middle code instructions. Instead, it will decrease the number of assembly code instructions, since an integer value cannot be the left-most operands.

(a) Before optimization	(b) After optimization
`1. if 10 < i goto 20`	`1. if i > 10 goto 20`

When generating the final assembly code, we cannot have an integer value as the left expression in a relational expression. Therefore, we swap the operands if the left operand holds an integer value. The expression cannot hold two integer values, in that case the **ConstantExpression** class of Chapter 8 would have reduced the expression to its resulting value.

Moreover, if the left expression is an array, function, or string, we want to use its address rather than its value. Similar to the integer value case, we cannot use the address directly in the assembly code. Therefore, if the left expression is an array, function or string, and the right expression is not an array, function, or string, or an integer value, we also swap the expressions.

We use the **m_swapMap** map to swap the operator. Note that that map is not the same map as the **m_inverseMap** we used in the Next-Jump-Double case above. When inversing an expression, we change its meaning, when we swap the expression it still has the same meaning, we just arrange the operands so that we can generate more efficient assembly code.

```
public static IDictionary<MiddleOperator, MiddleOperator> m_swapMap =
   new Dictionary<MiddleOperator, MiddleOperator>() {
    {MiddleOperator.Equal, MiddleOperator.Equal},
    {MiddleOperator.NotEqual, MiddleOperator.NotEqual},
    {MiddleOperator.LessThan, MiddleOperator.GreaterThan},
    {MiddleOperator.GreaterThan, MiddleOperator.LessThan},
    {MiddleOperator.LessThanEqual, MiddleOperator.GreaterThanEqual},
    {MiddleOperator.GreaterThanEqual, MiddleOperator.LessThanEqual}
  };

private void SwapRelation() {
  foreach (MiddleCode middleCode in m_middleCodeList) {
    if (middleCode.IsRelation()) {
      Symbol leftSymbol = (Symbol) middleCode[1],
             rightSymbol = (Symbol) middleCode[2];
      if ((leftSymbol.Value is BigInteger) ||
          (leftSymbol.Type.IsArrayFunctionOrString()) &&
          !rightSymbol.Type.IsArrayFunctionOrString() &&
          !(rightSymbol.Value is BigInteger)) {
        middleCode.Operator = m_swapMap[middleCode.Operator];
        middleCode[1] = rightSymbol;
        middleCode[2] = leftSymbol;
      }
    }
    }
  }
```

11.1.1. Optimize Communicative Expression

```
  private void OptimizeCommutative() {
    foreach (MiddleCode middleCode in m_middleCodeList) {
      if (middleCode.IsCommutative()) {
        Symbol leftSymbol = (Symbol) middleCode[1],
               rightSymbol = (Symbol) middleCode[2];

        // not 1 - i
        if (leftSymbol.Type.IsIntegralPointerArrayOrFunction() &&
            (leftSymbol.Value is BigInteger)) {
          middleCode[1] = rightSymbol;
          middleCode[2] = leftSymbol;
        }
      }
    }
  }
```

11.1.2. Remove Trivial Assignment

Trivial assignment, where a symbol is assigned the value of the same symbol, such as **x = x;**, shall be removed.

```
  private void RemoveTrivialAssign() {
    foreach (MiddleCode middleCode in m_middleCodeList) {
      MiddleOperator middleOperator = middleCode.Operator;

      if (middleOperator == MiddleOperator.Assign) {
        Symbol resultSymbol = (Symbol) middleCode[0],
               assignSymbol = (Symbol) middleCode[1];

        if (resultSymbol == assignSymbol) {
          middleCode.Operator = MiddleOperator.Empty;
          m_update = true;
        }
      }
    }
  }
```

11.1.3. Remove Cleared Code

Technically, it is not strictly necessary to remove cleared middle code instructions since they do not generate any target code. However, it simplifies the optimization methods of this section if we remove cleared code at the end of each iteration.

```
  public void RemoveClearedCode() {
```

We iterate backwards through the middle code list for performance reasons.

```
    for (int index = (m_middleCodeList.Count - 1); index >= 0;--index){
      if (m_middleCodeList[index].Operator == MiddleOperator.Empty) {
```

For each empty instruction we must go through all other instruction and decrease all target by one that is greater than the index of the instruction to be removed.

```
        foreach (MiddleCode middleCode in m_middleCodeList) {
```

```
        if (middleCode.IsRelationCarryOrGoto()) {
          int target = (int) middleCode[0];

          if (target > index) {
            middleCode[0] = target - 1;
          }
        }
      }

      m_middleCodeList.RemoveAt(index);
    }
  }
}
```

12. Assembly Code Generation

When we have generated and optimized the middle code, we continue to generate the final assembly code. The first step is to generate the assembly code with **tracks**. A track is a place holder for a register and follows a (yet unknown) register through the code. The track is then replaced by a proper register by the register allocator. Finally, the assembly code instructions are written in plain text.

12.1. Runtime Management

When it comes to runtime management, we use the classic style where we allocate an activation record for each function call, beginning with the initial call to **main**. Each activation record holds the data for the functions' parameters and local variables and constants. We use the **regular frame pointer** to hold the address of the current function call. The activation record also holds the frame pointer of the previous activation record and the return address; that is, the address to jump to when the execution of the current function returns to the calling function. More specifically: the address of the instruction following the call in the calling function.

What makes it a little bit more complicated is that a function may be variadic; that is, it has a variable number of parameters, like **printf** and **scanf**. We introduce a second frame pointer: the **variadic frame pointer** for variadic functions. It holds the address of the activation record plus the size of the extra parameters in the call to a variadic function.

To sum it up: the activation record holds the return address, regular frame pointer, and variadic frame pointer of the calling function as well as the parameters, potential extra parameter in case of a variadic function, and auto and register variables and constants of the current function. The variadic frame pointer is undefined in case of a regular calling function. However, we must always have the variadic frame pointer in the activation record since the function can be called by both variadic and regular functions.

When a function calls another function (or itself recursively) the activation record of the called function is located at the address above the activation record of the calling function. The return address, regular and variadic frame pointers of the calling function are stored at the beginning of the activation record of the called function. When a function returns to the calling function, the regular and variadic frame pointers are restored to the values of the calling function, and a jump back to the return address occurs.

The return value is not part of the activation record. It is stored in a register for integral and pointer types and at the floating-point stack for floating types. In case of a struct or union, its address is returned in a register.

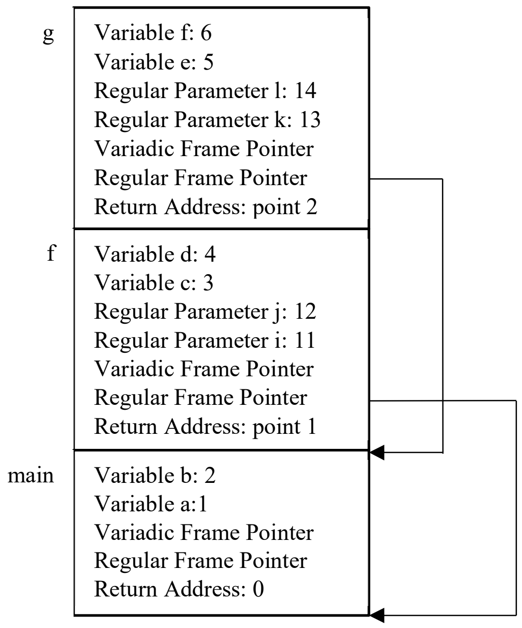
The return address of the activation record for the initial **main** call is initialized to zero. When main return and the return address is zero, an exit of the execution occurs instead of a return. But only for the first call, subsequent recursive calls to **main** have their own regular return addresses since it is quite possible to make recursive calls to **main**. For instance, the following program writes the number one to ten.

```
void main() {
  static count = 1;
  if (count <= 10) {
    printf("%d ", count++);
    main();
  }
}
```

Let us look at the following example.

```
void main() {
  int a = 1, b = 2;
  f(11, 12);
}

void f(int i, int j) {
  int c = 3, d = 4;
  g(13, 14);
}

void g(int k, int l) {
  int e = 5, f = 6;
  // point
}
```

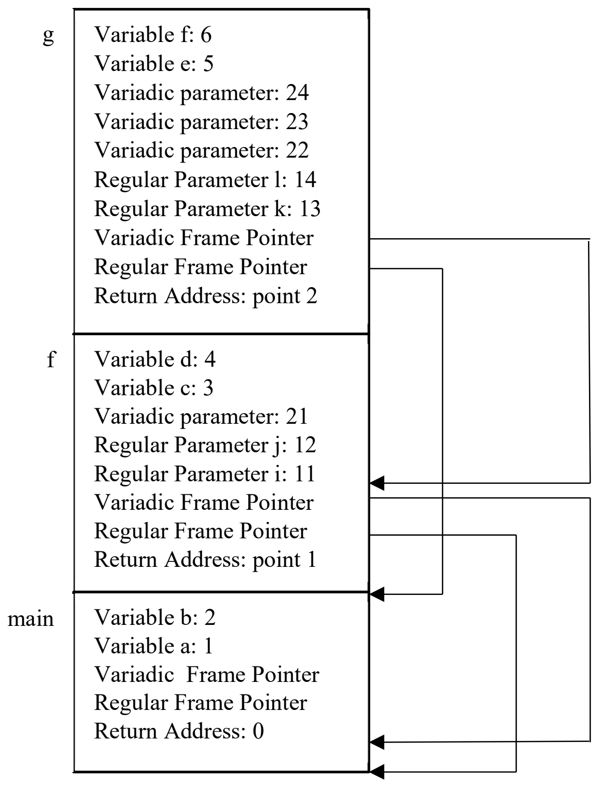
The program above has the call stack as follows when the execution of the program reached the point. The regular parameters refer to parameters defined in the parameter list. In the next example, we look into variadic parameter: extra parameters on calls to variadic functions.



In the example above, the variadic frame pointer is ignored. In the following example, however, we let **f** and **g** be variadic functions. The variadic status is marked by three dots ('…').

```
void main() {
  int a = 1, b = 2;
  f(11, 12, 21);
}
```

```
void f(int i, int j, ...) {
  int c = 3, d = 4;
  g(13, 14, 22, 23, 24);
}

void g(int k, int l, ...) {
  int e = 5, f = 6;
  // point
}
```



The variadic parameters are stored after the regular parameters in the activation record. Similar to the previous example, the regular frame pointers point at the beginning of the activation record of the calling function. However, the variadic frame pointers points at the beginning of the activation record of the calling function, plus the size of the variadic parameters. For the sake of argument, let us assume that an integer value allocates four bytes. Then the variadic frame pointer of **f** is the regular frame pointer plus four bytes (one integer value). The variadic frame pointer of **g** the regular frame pointer plus twelve bytes (three integer values). When refereeing to parameters in a variadic function, the regular frame pointer is used, and when refereeing to variables, the variadic frame pointer is used. The variadic parameters are accessible by using the address of the last regular parameter (it has to be at least one) and reading the values by increase the address, which is the task of the macros **va_list**, **va_start**, and **va_arg** in the **stdarg** standard library.

12.2. Assembly Operator

Like the middle code operators, we also have the assembly code operators. Several operators come in several varieties. For instance, the **add** instruction comes in the base form, which is used when a register is involved, like **add ax, 123**, **sub ax, [bp + 2]** or **xor [count], bx**. The other varieties are used when there is no register involved, and we therefore have to specify the size of the value to be assigned, like **add word [bp + 2], 123**.

AssemblyOperator.cs
```
namespace CCompiler {
  public enum AssemblyOperator {
    add, add_byte, add_dword, add_qword, add_word, return_address,
    and, and_byte, and_dword, and_qword, and_word, call,
    cmp, cmp_byte, cmp_dword, cmp_qword, cmp_word, comment,
    dec, dec_byte, dec_dword, dec_qword, dec_word,
    define_address, define_value, define_zero_sequence,
    div, div_byte, div_dword, div_qword, div_word, empty,
    fabs, fadd, faddp, fchs, fcompp, fdiv, fdivp, fdivr,
    fdivrp, fild_dword, fild_qword, fild_word,
    fist_dword, fist_qword, fist_word,
    fistp_dword, fistp_qword, fistp_word,
    fld1, fld_dword, fld_qword, fldcw, fldz,
    fmul, fmulp, fst_dword, fst_qword, fstcw,
    fstp_dword, fstp_qword, fstsw,
    fsub, fsubp, fsubr, fsubrp, ftst,
    idiv, idiv_byte, idiv_dword, idiv_qword, idiv_word,
    imul, imul_byte, imul_dword, imul_qword, imul_word,
    inc, inc_byte, inc_dword, inc_qword, inc_word, interrupt,
    ja, jae, jb, jbe, jc, je, jg, jge, jl, jle, jmp, jnc, jne, jnz, jz,
    label, lahf, mov, mov_byte, mov_dword, mov_qword, mov_word,
    mul, mul_byte, mul_dword, mul_qword, mul_word,
    neg, neg_byte, neg_dword, neg_qword, neg_word, new_middle_code,
    nop, not, not_byte, not_dword, not_qword, not_word,
    or, or_byte, or_dword, or_qword, or_word, pop, ret, sahf, set_track_size,
    shl, shl_byte, shl_dword, shl_qword, shl_word,
    shr, shr_byte, shr_dword, shr_qword, shr_word,
    sub, sub_byte, sub_dword, sub_qword, sub_word, syscall,
    xor, xor_byte, xor_dword, xor_qword, xor_word
  };
};
```

12.3. Assembly Code

The **AssemblyCode** class handles one assembly code instruction. It holds methods for initialization and optimization of a single instruction, a set of methods for testing and conversion of values and register, and the **ToString** method that writes the instruction in plain text.

AssemblyCode.cs
```
using System;
using System.Text;
using System.Numerics;
using System.Collections.Generic;
```

First, there is a set of special registers:

- **RegularFrameRegister**. The regular frame pointer: the address of the beginning of the current activation record.
- **VariadicFrameRegister**. The variadic frame pointer: the address of the beginning of the current activation record plus the size (in bytes) of the variadic parameters, in order to correctly access the

local variables of the function, which are located after the variadic parameters. Ignored in regular functions.

- **ReturnValueRegister**. Holds the function return value in case of integral or pointer type, or its address in case of struct or union type. Return values of floating types are not stored in a register or on the activation record, but rather at the floating-point stack. In C, it is not allowed to return values or array of function types.
- **ReturnAddressRegister**. Holds the address of the function return value in case of struct or union type.
- **ShiftRegister**. Holds the value of the right operand in a shift operation. Due to conditions of the architecture, it must always be **cl**.

```
namespace CCompiler {
  public class AssemblyCode {
    public static Register RegularFrameRegister, VariadicFrameRegister,
                           ReturnAddressRegister;
    public const Register ReturnValueRegister = Register.bx,
                          ShiftRegister = Register.cl;
```

We chose **bp**, **di**, and **bx** as the registers for the regular frame pointer, variadic frame pointer, and return value, since they (together with **si**) can be used as addresses in assembly operations.

```
    static AssemblyCode() {
      RegularFrameRegister =
        RegisterToSize(Register.bp, TypeSize.PointerSize);
      VariadicFrameRegister =
        RegisterToSize(Register.di, TypeSize.PointerSize);
      ReturnAddressRegister =
        RegisterToSize(Register.bx, TypeSize.PointerSize);
    }
```

An assembly instruction is made up by an operator and at most three operands. The operands are defined as objects, and they can be **Register**, **int**, **string** and **BigInteger**. All offsets are **int**, and all integer values are **BigInteger**.

```
    private AssemblyOperator m_operator;
    private object[] m_operandArray = new object[3];
```

The constructor initializes the instruction, and calls **FromAdditionToIncrement**, which changes addition and subtraction with one to increment and decrement, and **CheckSize**, which changes the operators in accordance with the given size, unless it is zero.

```
    public AssemblyCode(AssemblyOperator objectOp, object operand0,
                        object operand1, object operand2 = null,
                        int size = 0) {
      m_operator = objectOp;
      m_operandArray[0] = operand0;
      m_operandArray[1] = operand1;
      m_operandArray[2] = operand2;
      FromAdditionToIncrement();
      CheckSize(size);
    }

    public AssemblyOperator Operator {
      get { return m_operator; }
      set { m_operator = value; }
    }
```

The index property handles the operand array.

```
public object this[int index] {
  get { return m_operandArray[index]; }
  set { m_operandArray[index] = value; }
}
```

12.3.1. Assembly Code Optimization

Similar to the middle code optimization in Chapter 11, there is also some optimization of the assembly code. The **FromAdditionToIncrement** method changes the addition of one or subtraction of minus one to increment, and subtraction of one and addition of minus on to decrement. The first condition is that operator indeed is addition or subtraction and that the first operand is a track, a register, or a string.

```
private void FromAdditionToIncrement() {
  if (((Operator == AssemblyOperator.add) ||
       (Operator == AssemblyOperator.sub)) &&
      ((m_operandArray[0] is Track) || (m_operandArray[0] is Register) ||
       (m_operandArray[0] is string))) {
```

Then we check whether the second operand is a **BigInteger** and the third operand is null. In that case the value to be inspected has index one and we call **CheckIncrement** for further processing.

```
    if ((m_operandArray[1] is BigInteger) && (m_operandArray[2] == null)){
      CheckIncrement(1);
    }
```

If the second operand is an integer and operand is a **BigInteger**, the value to be inspected has index two.

```
    else if ((m_operandArray[1] is int) &&
             (m_operandArray[2] is BigInteger)) {
      CheckIncrement(2);
    }
  }
}
```

The **CheckIncrement** method inspects the value to be added or subtracted. If the operation is addition of one or subtraction of minus one, we replace the operator with increment.

```
private void CheckIncrement(int valueIndex) {
  int value = (int) ((BigInteger) m_operandArray[valueIndex]);

  if (((Operator == AssemblyOperator.add) && (value == 1)) ||
      ((Operator == AssemblyOperator.sub) && (value == -1))) {
    m_operator = AssemblyOperator.inc;
    m_operandArray[valueIndex] = null;
  }
```

In the same way, if the operation is addition of minus one subtraction of one, we replace the operator with decrement.

```
  else if (((Operator == AssemblyOperator.add) && (value == -1)) ||
           ((Operator == AssemblyOperator.sub) && (value == 1))) {
    m_operator = AssemblyOperator.dec;
    m_operandArray[valueIndex] = null;
  }
}
```

The **CheckSize** method changes the operator to a size operator. For instance, **add [bp + 2], 3** shall be changed to **add_word [bp + 2], 3** if the address represents a 2-byte value (**size** is two). Note that this method does not perform optimization, it just set the correct operator.

```
private void CheckSize(int size) {
  if ((size != 0) && ((m_operandArray[0] is Register) ||
       (m_operandArray[0] is Track) || (m_operandArray[0] is String)) &&
      (m_operandArray[1] is int) &&
      (IsUnary() || ((m_operandArray[2] is BigInteger) ||
                     (m_operandArray[2] is String)))) {
    m_operator = OperatorToSize(m_operator, size);
  }
}
```

12.3.2. Operator Test Methods

There is a set of methods that test whether the operator holds certain properties. The **IsUnary** method return true is the operator is unary; that is, it takes one operand. Note that we regard multiplication and division as unary operators, even though it takes two operands. That is because the left operand is stored in a register, only the second operand is explicitly stated. We also regard loading and storing of floating-point values as unary operators.

```
public bool IsUnary() {
  string operatorName = Enum.GetName(typeof(AssemblyOperator), Operator);
  return operatorName.StartsWith("neg") ||
         operatorName.StartsWith("not") ||
         operatorName.StartsWith("inc") ||
         operatorName.StartsWith("dec") ||
         operatorName.StartsWith("mul") ||
         operatorName.StartsWith("imul") ||
         operatorName.StartsWith("div") ||
         operatorName.StartsWith("idiv") ||
         operatorName.StartsWith("fst") ||
         operatorName.StartsWith("fld") ||
         operatorName.StartsWith("fist") ||
         operatorName.StartsWith("fild");
}
```

The jump and call test methods tests whether the first operand is a register. The jump instruction usually jumps to a specifically stated address. However, when returning from a function call, the address is stored in a register.

```
public bool IsJumpRegister() {
  return (Operator == AssemblyOperator.jmp) &&
         (m_operandArray[0] is Register);
}

public bool IsJumpNotRegister() {
  return (Operator == AssemblyOperator.jmp) &&
         !(m_operandArray[0] is Register);
}
```

A regular function call is to a specifically stated address. However, it is also possible to call a function whose address is stored in a function pointer.

```
public bool IsCallRegister() {
  return (Operator == AssemblyOperator.call) &&
         (m_operandArray[0] is Register);
}

public bool IsCallNotRegister() {
  return (Operator == AssemblyOperator.call) &&
         (m_operandArray[0] is string);
}
```

The **IsRelationNotRegister** method returns true if the operator is a relational jump operator. Note that there is no **IsRelationRegister**, since there are only jumps to specific addresses, not to addresses stored in registers.

```
    public bool IsRelationNotRegister() {
      switch (Operator) {
        case AssemblyOperator.je:
        case AssemblyOperator.jne:
        case AssemblyOperator.jl:
        case AssemblyOperator.jle:
        case AssemblyOperator.jg:
        case AssemblyOperator.jge:
        case AssemblyOperator.jb:
        case AssemblyOperator.jbe:
        case AssemblyOperator.ja:
        case AssemblyOperator.jae:
        case AssemblyOperator.jc:
        case AssemblyOperator.jnc:
          return true;

        default:
          return false;
      }
    }
```

12.3.3. Register Overlapping

When allocating registers, we have to make sure the registers do not overlap. To begin with, the **m_registerOverlapSet** set holds sets of all overlapping registers. Note that the **al** and **ah** registers are stored in separate sets, since they do not overlap.

```
    private static ISet<ISet<Register>> m_registerOverlapSet =
      new HashSet<ISet<Register>>() {
        new HashSet<Register>() {Register.al, Register.ax,
                                 Register.eax, Register.rax},
        new HashSet<Register>() {Register.ah, Register.ax,
                                 Register.eax, Register.rax},
        new HashSet<Register>() {Register.bl, Register.bx,
                                 Register.ebx, Register.rbx},
        new HashSet<Register>() {Register.bh, Register.bx,
                                 Register.ebx, Register.rbx},
        new HashSet<Register>() {Register.cl, Register.cx,
                                 Register.ecx, Register.rcx},
        new HashSet<Register>() {Register.ch, Register.cx,
                                 Register.ecx, Register.rcx},
        new HashSet<Register>() {Register.dl, Register.dx,
                                 Register.edx, Register.rdx},
        new HashSet<Register>() {Register.dh, Register.dx,
                                 Register.edx, Register.rdx},
        new HashSet<Register>() {Register.si, Register.esi, Register.rsi},
        new HashSet<Register>() {Register.di, Register.edi, Register.rdi},
        new HashSet<Register>() {Register.bp, Register.ebp, Register.rbp},
        new HashSet<Register>() {Register.sp, Register.esp, Register.rsp}
      };
```

The **RegisterOverlaps** methods returns true if the two registers overlap. If at least one register is null, they do not overlap, and we return false.

```
    public static bool RegisterOverlap(Register? register1,
                                       Register? register2) {
      if ((register1 == null) || (register2 == null)) {
        return false;
      }
```

Otherwise, we iterate through the **m_registerOverlapSet**. If one of the register sets holds both the registers, they overlap, and we return true.

```
      foreach (ISet<Register> registerSet in m_registerOverlapSet) {
        if (registerSet.Contains(register1.Value) &&
            registerSet.Contains(register2.Value)) {
          return true;
        }
      }
```

If we have iterated through the **m_registerOverlapSet** without finding a set holding both the registers, the registers do not overlap, and we return false.

```
      return false;
    }
```

12.3.4. Register Size

The **SizeOfRegister** method uses the **m_registerSizeMap** map to look up and return the size of the register.

```
    private static IDictionary<Register,int> m_registerSizeMap =
      new Dictionary<Register,int>() {
       {Register.al, 1}, {Register.bl, 1}, {Register.cl, 1}, {Register.dl, 1},
       {Register.ah, 1}, {Register.bh, 1}, {Register.ch, 1}, {Register.dh, 1},
       {Register.ax, 2}, {Register.bx, 2}, {Register.cx, 2}, {Register.dx, 2},
       {Register.eax,4}, {Register.ebx,4}, {Register.ecx,4}, {Register.edx,4},
       {Register.rax,8}, {Register.rbx,8}, {Register.rcx,8}, {Register.rdx,8},
       {Register.si, 2}, {Register.di, 2}, {Register.sp, 2}, {Register.bp, 2},
       {Register.esi,4}, {Register.edi,4}, {Register.esp,4}, {Register.ebp,4},
       {Register.rsi,8}, {Register.rdi,8}, {Register.rsp,8}, {Register.rbp, 8}
      };

    public static int SizeOfRegister(Register register) {
      return m_registerSizeMap[register];
    }
```

The **RegisterToSize** method converts a register to the given size. The **m_registerSet** set holds lists of registers of increasing sizes. Note that there are lists rather than sets, because the order of the registers is significant.

```
    private static ISet<IList<Register>> m_registerSet =
      new HashSet<IList<Register>>() {
        new Register[] {Register.al, Register.ax, Register.eax, Register.rax},
        new Register[] {Register.bl, Register.bx, Register.ebx, Register.rbx},
        new Register[] {Register.cl, Register.cx, Register.ecx, Register.rcx},
        new Register[] {Register.dl, Register.dx, Register.edx, Register.rdx},
        new Register[] {default(Register), Register.si,
                        Register.esi, Register.rsi},
        new Register[] {default(Register), Register.di,
                        Register.edi, Register.rdi},
        new Register[] {default(Register), Register.bp,
                        Register.ebp, Register.rbp},
        new Register[] {default(Register), Register.sp,
                        Register.esp, Register.rsp}
      };
```

The **m_sizeToIndexMap** map maps the size of the registers to the index in the register lists of the **m_registerSet** set above.

```
    private static IDictionary<int,int> m_sizeToIndexMap =
      new Dictionary<int, int>() {{1, 0}, {2, 1}, {4, 2}, {8, 3}};
```

The **RegisterToSize** method returns the register with the given size. If the register already holds the given size, we just return the register.

```
    public static Register RegisterToSize(Register register, int size) {
      if (m_registerSizeMap[register] == size) {
        return register;
      }
```

Otherwise, we iterate through the **m_registerSet** set, and check if any of the register sets contains the register. If it contains the register, we return the register on the index in the list corresponding to the size of the register.

```
      foreach (IList<Register> registerList in m_registerSet) {
        if (registerList.Contains(register)) {
          int index = m_sizeToIndexMap[size];
          Assert.ErrorXXX((index >= 0) && (index < registerList.Count));
          return registerList[index];
        }
      }

      Assert.ErrorXXX(false);
      return default(Register);
    }
```

The **SizeOfOperator** method returns the size of the operator. We obtain the name of the operator by calling the **GetName** method in the **Enum** class, and test whether is holds the suffix “byte”, “word”, “dword”, or “qword”, in which cases the operator size is one, two, four, or eight bytes.

```
    public static int SizeOfOperator(AssemblyOperator objectOp) {
      string name = Enum.GetName(typeof(AssemblyOperator), objectOp);
      string suffix = name.Substring(name.IndexOf("_") + 1);

      switch (suffix) {
        case "byte":
          return 1;

        case "word":
          return 2;

        case "dword":
```

```
        return 4;

      default: // "qword":
        return 8;
    }
  }
```

The **OperatorToSize** method changes a plain operator to a size operator. If the operator size is one, two, four, or eight, its name shall be suffixed by "_byte", "_word", "_dword", or "_qword".

```
  public static AssemblyOperator OperatorToSize
                               (AssemblyOperator objectOp, int size) {
    string name = Enum.GetName(typeof(AssemblyOperator), objectOp);
    Assert.ErrorXXX(objectOp != AssemblyOperator.interrupt);

    switch (size) {
      case 1:
        name = name + "_byte";
        break;

      case 2:
        name = name + "_word";
        break;

      case 4:
        name = name + "_dword";
        break;

      case 8:
        name = name + "_qword";
        break;
    }
```

We use the **Parse** method of the **Enum** class to obtain the resulting operator from the name.

```
    Assert.ErrorXXX(name.Contains("_"));
    return ((AssemblyOperator) Enum.Parse(typeof(AssemblyOperator), name));
  }
```

The **SizeOfValue** method return a size of a value, with two exceptions. In case of a **mov** operator, the size is given by the operator, and in case of a **cmp** operator with value zero, the size is one. Otherwise, we just call the second **SizeOfValue** to get the value size.

```
  public static int SizeOfValue(BigInteger value, AssemblyOperator op) {
    string name = Enum.GetName(typeof(AssemblyOperator), op);

    if (name.StartsWith("mov")) {
      return SizeOfOperator(op);
    }
    else if (name.StartsWith("cmp") && (value == 0)) {
      return 1;
    }
    else {
      return SizeOfValue(value);
    }
  }
```

The second version of **SizeOfValue** return the size of a value. The minimum and maximum value is -2^{b-1} and $2^{b-1} - 1$, where b is the number of bits of the type.

```
public static int SizeOfValue(BigInteger value) {
  if (value == 0) {
    return 0;
  }
  else if ((-128 <= value) && (value <= 127)) {
    return 1;
  }
  else if ((-32768 <= value) && (value <= 32767)) {
    return 2;
  }
  else if ((-2147483648 <= value) && (value <= 2147483647)) {
    return 4;
  }
  else {
    return 8;
  }
}
```

12.3.5. ToString

The **ToString** method returns the instruction in plain text. We have six categories of operations:

```
public override string ToString() {
  object operand0 = m_operandArray[0],
         operand1 = m_operandArray[1],
         operand2 = m_operandArray[2];
  string operatorName = Enum.GetName(typeof(AssemblyOperator),
                                     Operator).Replace("_", " ");
```

Instructions that perform computations are nullary (without operands), unary (with one operand), or binary (with two operands). In binary operations, the left operand must be a register or an address; a static name or an integer value is not allowed. The operations can be further divided into six categories:

Type	**Category**	**Description**	**Examples**
Nullary	1	operator	fadd
Unary	2	operator register	neg ax
	3	operator [register + offset] operator [static name + offset]	inc [bp + 2] dec [stdin + 4]
Binary	5	operator register, register operator register, static name operator register, integer value	mov ax, bx add ax, stdin sub ax, 123
	6	operator [register + offset], register operator [register + offset], static name operator [register + offset], integer value operator [static name + offset], register operator [static name + offset], static name operator [static name + offset], integer value	mov [bp + 2], bx add [bp + 2], stdin sub [bp + 2], 123 mov [stdin + 4], bx add [stdin + 4], stdin sub [stdin + 4], 123
	6	operator register, [register + offset] operator register, [static name + offset]	mov ax, [bp + 2] add ax, [stdin +4]

In case of a nullary operator, we simply return its name.

```
if (IsNullary()) {
  return "\t" + operatorName;
}
```

Unary operations of category two and three, where the operand is an address or a register. In case of an address, the first operand is a register or a string (static name), the second operand is an integer (offset), and the third operand is null. In case of a register, the second and third operands are null.

```
else if (IsUnary()) {
  if (((operand0 is Register) || (operand0 is string)) &&
           (operand1 is int) && (operand2 == null)) {
    return "\t" + operatorName + " [" + operand0 +
           WithSign(operand1) + "]";
  }
  else if ((operand0 is Register) && (operand1 == null) &&
           (operand2 == null)) {
    return "\t" + operatorName + " " + operand0;
  }
```

Next, we handle binary operations. In category four, the first operand is a register, and the second operand is a register, a string (static name), or a **BigInteger** (integer value).

```
else if (IsBinary()) {
  if ((operand0 is Register) && ((operand1 is Register) ||
      (operand1 is string) || (operand1 is BigInteger)) &&
      (operand2 == null)) {
    Assert.ErrorXXX(!(operand0 is string));
    return "\t" + operatorName + " " + operand0 + ", " + operand1;
  }
```

In category five, the first operand is a register or a string (static name), the second operand is an integer (offset), and the third operand is register, a string (static name), or a **BigInteger** (integer value).

```
  else if (((operand0 is Register) || (operand0 is string)) &&
           (operand1 is int) && ((operand2 is Register) ||
            (operand2 is string) || (operand2 is BigInteger))) {
    return "\t" + operatorName +
           " [" + operand0 + WithSign(operand1) + "], " + operand2;
  }
```

In category six, the first operand is a register, the second operand is register or string (static name), and the third operand is an integer (offset).

```
  else if ((operand0 is Register) && ((operand1 is Register) ||
           (operand1 is string)) && (operand2 is int)) {
    return "\t" + operatorName + " " + operand0 +
           ", [" + operand1 + WithSign(operand2) + "]";
  }
}
```

In case of a label or comment, we simple return them. Note that we add a colon (':') to the label and insert a semicolon (';').

```
else if (Operator == AssemblyOperator.label) {
  return "\n " + operand0 + ":";
}
```

```
      else if (Operator == AssemblyOperator.comment) {
        return "\t; " + operand0;
      }
```

When defining a value, we call the **ToVisibleString** method in case of a string.

```
      else if (Operator == AssemblyOperator.define_value) {
        Sort sort = (Sort) operand0;
        object value = operand1;

        if (sort == Sort.String) {
          return "\tdb " + ToVisibleString((string) operand1);
        }
```

If the value is not a string, it is an integral or floating value. In case of floating value, we add a dot ('.'), unless it already has a dot.

```
        else {
          string text = operand1.ToString();

          if (((sort == Sort.Float) || (sort == Sort.Double) ||
               (sort == Sort.LongDouble)) && !text.Contains(".")) {
            text += ".0";
          }
```

When we define the value, we use different directives depending on the size of the value: **db** (define byte) for one byte, **dw** (define word) for two bytes, **dd** (define double-word) for four bytes, and **dq** (define quarto) for eight bytes.

```
          switch (TypeSize.Size(sort)) {
            case 1:
              return "\tdb " + text;

            case 2:
              return "\tdw " + text;

            case 4:
              return "\tdd " + text;

            case 8:
              return "\tdq " + text;
          }
        }
      }
```

When defining an address, we use the **dq** directive, since the size of addresses is eight bytes. The **WithSign** method write a positive or negative offset, unless the offset is zero.

```
      else if (Operator == AssemblyOperator.define_address) {
        string name = (string) operand0;
        int offset = (int) operand1;
        return "\tdq " + name + WithSign(offset);
      }
```

When defining a sequence of zeros, we use the **times** and **db** directive, which repeats the following definition, which is the zero value of one-byte size.

```
      else if (Operator == AssemblyOperator.define_zero_sequence) {
        int size = (int) operand0;
        return "\ttimes " + size + " db 0";
      }
```

For a jump or function call, we use the **jmp** instruction.

```
      else if (IsJumpRegister() || IsCallRegister() ||
               IsCallNotRegister()) {
        return "\tjmp " + operand0;
      }
```

For address return, we load the target into the address.

```
      else if (Operator == AssemblyOperator.return_address) {
        string target = SymbolTable.CurrentFunction.UniqueName +
                        Symbol.SeparatorId + operand2;
        return "\tmov qword [" + operand0 + WithSign(operand1) + "], " +
               target;
      }
```

In case of conditional and unconditional jump instructions, we have three cases. If the third operand (**operand2**) is an integer, we jump to the address of a middle code instruction. More specific, we jump to a label, which is made up by the name of the current function, and the middle code index given by the third operand.

```
      else if (IsRelationNotRegister() || IsJumpNotRegister()) {
```

If the third operand is string, we have a special case: the jump instruction is part of the handling of the command line arguments. We simply jump to the address given by the string.

Otherwise, we have jump in a memory copy of a struct or union.

```
        Assert.ErrorXXX(operand2 is int);
        string label = SymbolTable.CurrentFunction.UniqueName +
                        Symbol.SeparatorId + operand2;
        return "\t" + operatorName + " " + label;
      }
    }
```

The **ToVisibleString** method returns the text with non-graphical character written with their ascii numbers. For instance, the text “Hello\nWorld\r”, will be translated to “Hello”, 10, “World”, 13.

```
    private static string ToVisibleString(string text) {
      StringBuilder buffer = new StringBuilder();
```

The **insideString** variable is true as long as the current character is located inside a string.

```
      bool insideString = false;
```

We iterate through the text and, for each character, check if it is not a graphical character. In that case we end the string, if necessary, and add the ascii value of the character.

```
      foreach (char c in text) {
        if (Char.IsControl(c) || (c == '\"') || (c == '\'')) {
          if (insideString) {
            buffer.Append("\", " + ((int) c).ToString() + ", ");
            insideString = false;
          }
          else {
            buffer.Append(((int) c).ToString() + ", ");
          }
        }
```

If the character is a graphical character, we begin the string, if necessary, and add the character to the string.

```
        else {
          if (insideString) {
            buffer.Append(c);
          }
          else {
            buffer.Append("\"" + c.ToString());
            insideString = true;
          }
        }
      }
```

We end the string by adding the terminating zero character.

```
      if (insideString) {
        buffer.Append("\", 0");
      }
      else {
        buffer.Append("0");
      }

      return buffer.ToString();
    }
```

The **MakeMemory** method return a label for the memory copy method.

```
    public static string MakeLabel(int labelIndex) {
      return "label" + Symbol.SeparatorId + labelIndex;
    }
```

The **WithSign** method returns the offset of an address preceded by a plus or minus sign. In case of a zero value, an empty string is returned.

```
    private string WithSign(object value) {
      int offset = (int) value;

      if (offset > 0) {
        return " + " + offset;
      }
      else if (offset < 0) {
        return " - " + (-offset);
      }
      else {
        return "";
      }
    }
  }
}
```

12.4. Tracks

A track is a place holder for a register through the assembly code. Let us look at the following example.

```
 return a + b & c - d;      1. temp0 = a + b
                            2. temp1 = c - d
                            3. temp2 = temp0 & temp1
                            4. return temp2
```

(a) C code (b) Middle code

The following middle code is generated.

Let us assume that **a**, **b**, **c**, **d**, and **e** are placed on the current activation record of a regular function with offsets 6, 8, 10, and 12, and the **bp** and **di** registers are the regular frame pointer and the variadic frame pointer.

The following assembly code is generated, with the tracks **track0**, **track1**, and **track2** as place holders for the registers. Each track represents a register, but we do not yet know which one.

```
mov track0,[bp + 6]
add track0,[bp + 8]

mov track1,[bp + 10]
sub track1,[bp + 12]

and track0, track1
```

In the return sequence, we reset the variadic and regular frame pointers to the values of the calling function, which is placed at offset 4 and 2 on the activation record, and then jump to the return address, which is placed at the beginning of the activation record. Note the order between the two middle lines. We cannot swap them, since **bp** is set on the third line.

```
mov track2, [bp]
mov di, [bp + 4]
mov bp, [bp + 2]
jmp track2
```

The return value is stored in **track0**, and shall thereby be assigned the return value register. I have chosen **bx** for the return value register, since it can be used both as an operand in arithmetic operations and as an address register. So **track0** is assigned **bx** when the return instruction is added to the code.

The register allocator assigns the remaining tracks **track1** and **track2** to suitable registers. Since **track0** and **track2** do not overlap, it is quite possible to assign them the same register, such as **ax**.

```
mov bx,[bp + 6]
add bx,[bp + 8]

mov ax,[bp + 10]
sub ax,[bp + 12]

and bx, ax

mov ax, [bp]
mov di, [bp + 4]
mov bp, [bp + 2]
jmp ax
```

The **Track** class holds a track representing a known or unknown register.

Track.cs

```
using System;
using System.Collections.Generic;

namespace CCompiler {
  public class Track {
```

The first constructor takes a symbol and a potential register as parameters, and initializes the current and maximum sizes.

```
public Track(Symbol symbol, Register? register = null) {
  Register = register;
  Assert.ErrorXXX(symbol != null);
  Assert.ErrorXXX(!symbol.Type.IsStructOrUnion());
  CurrentSize = m_maxSize = symbol.Type.ReturnSize();
}
```

The second constructor takes a type and initializes the current and maximum sizes.

```
public Track(Type type) {
  Assert.ErrorXXX(type != null);
  Assert.ErrorXXX(!type.IsStructOrUnion());
  Assert.ErrorXXX(!type.IsArrayFunctionOrString());
  CurrentSize = m_maxSize = type.Size();
}
```

A track has a current size and a maximal size. The current size may vary between assembly code instructions. We need the maximal size for the register allocator. It the size more than one byte, a reduced set of register is available.

```
public int CurrentSize {get;set;}

private int m_maxSize;
public int MaxSize {
  get { return m_maxSize; }
  set { m_maxSize = Math.Max(m_maxSize, value); }
}
```

The min and max index hold he indexes of the first and last assembly code instruction where the track occurs. We need them to decide whether tracks overlap.

```
private int m_minIndex = -1, m_maxIndex = -1;
public int Index {
  set {
    m_minIndex = (m_minIndex != -1) ? Math.Min(m_minIndex, value) : value;
    m_maxIndex = Math.Max(m_maxIndex, value);
  }
}
```

Although the idea is that a track does not hold a register but is rather assigned a register by the register allocator, there are some cases where the track is assigned a register from the beginning. A track can hold a specific register. In multiplication and division, the left operands must be stored in a specific register. In left or right shift, the right operands must be stored in the **cl** register. When returning a value, it must be stored in a specific register.

```
public Register? Register {get; set;}
```

If the track is used as a pointer, a reduced set of registers are available.

```
public bool Pointer {get; set;}
```

The **Overlaps** method returns true if the tracks overlaps. Two tracks overlap if the min and max line numbers overlap.

```
public static bool Overlaps(Track track1, Track track2) {
  Assert.ErrorXXX((track1.m_minIndex != -1) && (track1.m_maxIndex != -1));
  Assert.ErrorXXX((track2.m_minIndex != -1) && (track2.m_maxIndex != -1));
```

```
      return !(((track1.m_maxIndex < track2.m_minIndex) ||
                (track2.m_maxIndex < track1.m_minIndex)));
    }
  }
}
```

12.5. Register Allocation

The architecture holds a set of registers. One particular feature of the registers is that they to some extent overlaps. See Appendix A.3.7. for a more detailed description.

Register.cs

```
namespace CCompiler {
  public enum Register {al, ah, ax, eax, rax, bl, bh, bx, ebx, rbx,
                        cl, ch, cx, ecx, rcx, dl, dh, dx, edx, rdx,
                        si, esi, rsi, di, edi, rdi,
                        sp, esp, rsp, bp, ebp, rbp};
}
```

The **RegisterAllocator** method take the total track set and total assembly list; that is, the track set for the whole function, and the assembly list for the whole function. The first task is the find out which tracks overlaps each other. We construct the graph in the following way: the tracks are the vertices of the graph and two vertices have an edge if their tracks overlaps. The register allocation is then performed as a graph coloring, two vertices connected by an edge cannot be given the same color. In our case: two tracks that overlaps cannot be assigned the same register.

Graph coloring is a NP-complete problem, which means that there is no known algorithm that works on polynomial time. To be sure that we have found the best solution, we have to exam every possible solution, which would require an unrealistic amount of time. Instead, in this book we perform a **deep search**, where we sort the vertices into a list and for each vertex try to find a register not already allocated by any of its neighbors. If we cannot find such register, we backtrack and try another combination. If we finally find a solution where each vertex is mapped to a register not mapped by any of its neighbors, we have found an optimal solution. If we do not find a solution, we report an error: shortage of registers.

RegisterAllocator.cs

```
using System.Collections.Generic;

namespace CCompiler {
  public class RegisterAllocator {
    public RegisterAllocator(ISet<Track> totalTrackSet,
                             List<AssemblyCode> assemblyCodeList) {
      Graph<Track> totalTrackGraph = new Graph<Track>(totalTrackSet);
```

First, we build the total track graph, with all track as vertices and an edge between two vertices if their tracks overlaps. Two tracks that overlap shall not be assigned the same register, or two overlapping registers.

```
      foreach (Track track1 in totalTrackSet) {
        foreach (Track track2 in totalTrackSet) {
          if (!track1.Equals(track2) && Track.Overlaps(track1, track2)) {
            totalTrackGraph.AddEdge(track1, track2);
          }
        }
      }
```

Then we split the total graph into independent subgraphs. In this way, we can start by any vertex in each of the subgraph. Otherwise, we would have to start at every vertex to be sure that we have reached all the vertexes.

```
      ISet<Graph<Track>> split = totalTrackGraph.Split();
```

When the graph has been split, we iterate through the subgraphs and perform a deep-first search on each of them.

```
      foreach (Graph<Track> trackGraph in split) {
        List<Track> trackList = new List<Track>(trackGraph.VertexSet);
        Assert.Error(DeepFirstSearch(trackList, 0, trackGraph),
                     Message.Out_of_registers);
```

If the **DeepFirstSearch** method returns true, we have a graph coloring; that is, the tracks have been assigned registers that do not overlap. If **DeepFirstSearch** return false, we have not found a graph coloring, and we report an error, that we are out of registers.

```
      ISet<Graph<Track>> split = totalTrackGraph.Split();
      foreach (Graph<Track> trackGraph in split) {
        List<Track> trackList = new List<Track>(trackGraph.VertexSet);
        Assert.Error(DeepFirstSearch(trackList, 0, trackGraph),
                     Message.Out_of_registers);
      }
```

If we have iterated through the track graphs without encountering shortage of register, each track has been assigned a register. We call **SetRegistersInCodeList** that iterates through the code and replace each track with its assigned register.

```
      SetRegistersInCodeList(assemblyCodeList);
    }

    private static void SetRegistersInCodeList(List<AssemblyCode>
                                            assemblyCodeList) {
      foreach (AssemblyCode assemblyCode in assemblyCodeList) {
        if (assemblyCode.Operator == AssemblyOperator.set_track_size) {
          Track track = (Track) assemblyCode[0];
          object operand1 = assemblyCode[1];
```

When iterating through the code list, we also need to keep track of the current size of the tracks, in order to replace the track with a register of correct size.

```
          if (operand1 is int) {
            track.CurrentSize = (int) operand1;
          }
          else {
            track.CurrentSize = ((Track) operand1).CurrentSize;
          }

          assemblyCode.Operator = AssemblyOperator.empty;
        }
        else {
          Check(assemblyCode, 0);
          Check(assemblyCode, 1);
          Check(assemblyCode, 2);
        }
      }
    }
```

The **Check** method checks for each position in the assembly code is a reference a track is replaced by a register of the correct size.

```
private static void Check(AssemblyCode assemblyCode, int position) {
  if (assemblyCode[position] is Track) {
    Track track = (Track) assemblyCode[position];
    Assert.ErrorXXX(track.Register != null);
    assemblyCode[position] =
     AssemblyCode.RegisterToSize(track.Register.Value, track.CurrentSize);
  }
}
```

The **DeepFirstSearch** method searches the graph in a deep-first manner. It takes the track list and the current index in that list as well as the track graph.

```
private bool DeepFirstSearch(List<Track> trackList, int listIndex,
                             Graph<Track> trackGraph) {
```

If the index equals the size of the track list, we return true because we have iterated through the list and found a match; that is, each track has been assigned a register and no overlapping tracks have the same register.

```
  if (listIndex == trackList.Count) {
    return true;
  }
```

If the current track has already been assigned a register, we just call **DeepFirstSearch** with the next index.

```
  Track track = trackList[listIndex];
  if (track.Register != null) {
    return DeepFirstSearch(trackList, listIndex + 1, trackGraph);
  }
```

If the current track has not been assigned a register, we look up the set of possible register and the set of neighbor vertices; that is, the set of overlapping tracks.

```
  ISet<Register> possibleSet = GetPossibleSet(track);
  ISet<Track> neighbourSet = trackGraph.GetNeighbourSet(track);
```

We iterate through the set of possible register and, for each register that does not cause an overlapping, we assign the register to the track and call **DeepFirstSearch** recursively with the next index. If the call returns true, we have found a total mapping of registers to the track, and we just keep returning true. However, if the call does return false, we just try with another of the possible registers. If none of the registers causes a match, we clear the register of the track and return false.

```
  foreach (Register possibleRegister in possibleSet) {
    if (!OverlapNeighbourSet(possibleRegister, neighbourSet)) {
      track.Register = possibleRegister;

      if (DeepFirstSearch(trackList, listIndex + 1, trackGraph)) {
        return true;
      }

      track.Register = null;
    }
  }

  track.Register = null;
  return false;
```

```
    }
```

The **OverlapNeighbourSet** method return true if the register overlaps any of its neighbors. The **RegisterOverlap** method in the **AssemblyCode** class test whether two register overlaps.

```
    private bool OverlapNeighbourSet(Register register,
                                     ISet<Track> neighbourSet) {
      foreach (Track neighbourTrack in neighbourSet) {
        if (AssemblyCode.RegisterOverlap(register, neighbourTrack.Register)) {
          return true;
        }
      }

      return false;
    }
```

The **VariadicFunctionPointerRegisterSet** set holds the possible pointer registers of a variadic function while **RegularFunctionPointerRegisterSet** holds the possible pointer registers of a regular function. The **Byte1RegisterSet** set holds all registers of one byte while **Byte2RegisterSet** holds all registers of two bytes.

```
    public static ISet<Register>
      VariadicFunctionPointerRegisterSet = new HashSet<Register>() {
        AssemblyCode.RegisterToSize(Register.bp, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.si, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.di, TypeSize.PointerSize),
        AssemblyCode.RegisterToSize(Register.bx, TypeSize.PointerSize)
      },
      RegularFunctionPointerRegisterSet =
        new HashSet<Register>(VariadicFunctionPointerRegisterSet),
      Byte1RegisterSet = new HashSet<Register>() {
        Register.al,Register.ah, Register.bl, Register.bh,
        Register.cl, Register.ch, Register.dl, Register.dh
      },
      Byte2RegisterSet = new HashSet<Register>() {
        Register.ax, Register.bx, Register.cx, Register.dx
      };
```

We remove the frame register from **RegularFunctionPointerRegisterSet** and **VariadicFunctionPointer-RegisterSet**, since we need it to point at the current activation record. We also remove the variadic register from **VariadicFunctionPointerRegisterSet**, since we need it to point at the activation record in a variadic function.

```
    static RegisterAllocator() {
      VariadicFunctionPointerRegisterSet.
        Remove(AssemblyCode.RegularFrameRegister);
      RegularFunctionPointerRegisterSet.
        Remove(AssemblyCode.RegularFrameRegister);
      RegularFunctionPointerRegisterSet.
        Remove(AssemblyCode.VariadicFrameRegister);
    }
```

The **GetPossibleSet** method returns the possible set a track, depending on whether the track holds a pointer, or the size of the track.

```
    private static ISet<Register> GetPossibleSet(Track track) {
      if (track.Pointer) {
        if (SymbolTable.CurrentFunction.Type.IsVariadic()) {
          return RegularFunctionPointerRegisterSet;
        }
        else {
          return VariadicFunctionPointerRegisterSet;
        }
      }
```

If the track does not hold a pointer wee look into its size. If the size is one, we have a larger set to choose from. There are eight non-pointer registers of size one while there are four registers of the other sizes.

```
      else if (track.MaxSize == 1) {
        return Byte1RegisterSet;
      }
```

We return the set of registers of size two, even if the size is actually larger than two. In that case, the **RegisterToSize** method in the **AssemblyCode** class will convert the register to the correct size.

```
      else {
        return Byte2RegisterSet;
      }
    }
  }
}
```

12.6. Assembly Code Generation

The **AssemblyCodeGenerator** class holds methods for generating the final assembly code. When a value is stored in a register, to be used by a later instruction, its symbol and track is added to the **m_trackMap** map. The generated assembly code instructions are stored in the **m_assemblyCodeList** list.

AssemblyCodeGenerator.cs

```
using System.Linq;
using System.Numerics;
using System.Collections.Generic;

namespace CCompiler {
  public class AssemblyCodeGenerator {
    public IDictionary<Symbol,Track> m_trackMap =
      new Dictionary<Symbol,Track>();
    public List<AssemblyCode> m_assemblyCodeList;
```

The **m_floatStackSize** field keeps track of the size of the current floating-point stack. It can hold at most seven values.

```
    private int m_floatStackSize = 0;
    public const int FloatingStackMaxSize = 7;
```

The **MainName** field holds the name of the main function.

```
    public static string InitializerName = Symbol.SeparatorId + "initializer";
    public static string ArgsName = Symbol.SeparatorId + "args";
    public static string PathName = Symbol.SeparatorId + "PathName";
```

The constructor takes the assembly code list.

```
    public AssemblyCodeGenerator(List<AssemblyCode> assemblyCodeList) {
      m_assemblyCodeList = assemblyCodeList;
    }
```

The **RegisterAllocation** method performs the register allocation by creating an object of the **RegisterAllocator** class. It may seem strange, but it actually works, since the constructor performs the register allocation and adds the register to the assembly code list.

```
private void RegisterAllocation(ISet<Track> trackSet) {
  new RegisterAllocator(trackSet, m_assemblyCodeList);
}
```

The static **GenerateAssembly** method generates the assembly code by creating an object of the **AssemblyCodeGenerator** class that generates the assembly code with tracks, and replacing the tracks with proper registers by performing register allocation.

```
public static void GenerateAssembly(List<MiddleCode> middleCodeList,
                                    List<AssemblyCode> assemblyCodeList) {
  AssemblyCodeGenerator objectCodeGenerator =
    new AssemblyCodeGenerator(assemblyCodeList);
  objectCodeGenerator.AssemblyCodeList(middleCodeList);
  ISet<Track> trackSet = objectCodeGenerator.TrackSet();
  objectCodeGenerator.RegisterAllocation(trackSet);
}
```

The first **AddAssemblyCode** method adds a new assembly instruction to the assembly code list **m_assemblyCodeList**.

```
public AssemblyCode AddAssemblyCode(AssemblyOperator objectOp,
                      object operand0 = null, object operand1 = null,
                      object operand2 = null, int size = 0) {
  AssemblyCode assemblyCode =
    new AssemblyCode(objectOp, operand0, operand1, operand2, size);
  m_assemblyCodeList.Add(assemblyCode);
  return assemblyCode;
}
```

The second **AddAssemblyCode** method adds a new assembly instruction to the given assembly code list. It is static and is called when generating special assembly code, such as initialization code and code for command line arguments.

```
public static AssemblyCode AddAssemblyCode(List<AssemblyCode> list,
                AssemblyOperator objectOp, object operand0 = null,
                object operand1 = null, object operand2 = null,
                int size = 0) {
  AssemblyCode assemblyCode =
    new AssemblyCode(objectOp, operand0, operand1, operand2, size);
  list.Add(assemblyCode);
  return assemblyCode;
}
```

12.6.1. The Long Switch Statement

The constructor iterates through the middle code list and calls an appropriate method for each kind of middle code instruction.

```
public void AssemblyCodeList(List<MiddleCode> middleCodeList){
  for (int middleIndex = 0; middleIndex < middleCodeList.Count;
      ++middleIndex) {
    MiddleCode middleCode = middleCodeList[middleIndex];
    AddAssemblyCode(AssemblyOperator.new_middle_code, middleIndex);
```

If the current function is not null (if the code is located in function scope rather global scope), we add a label in order to make the assembly code more legible. If the middle code instruction is an initializer, we just add a label with the text of the middle code instruction.

```
        if (SymbolTable.CurrentFunction != null) {
          if ((middleCode.Operator == MiddleOperator.Initializer) ||
              (middleCode.Operator == MiddleOperator.InitializerZero)) {
            AddAssemblyCode(AssemblyOperator.label, null,
                            middleCode.ToString());
          }
```

If the middle code instruction is not an initializer, we add a label with the unique name of the function. If the middle code index is greater the zero, we also add it to the label.

```
          else {
            string label = SymbolTable.CurrentFunction.UniqueName;

            if (middleIndex > 0) {
              label += Symbol.SeparatorId + middleIndex;
            }

            AddAssemblyCode(AssemblyOperator.label, label,
                            middleCode.ToString());
          }
        }
```

Now follows the long switch statement, where each middle code instruction is matched to a function call that generates the corresponding assembly code.

```
        switch (middleCode.Operator) {
          case MiddleOperator.CallHeader:
            FunctionPreCall(middleCode);
            break;

          case MiddleOperator.Call:
            FunctionCall(middleCode, middleIndex);
            break;

          case MiddleOperator.PostCall:
            FunctionPostCall(middleCode);
            break;

          case MiddleOperator.Return:
            Return(middleCode, middleIndex);
            break;

          case MiddleOperator.Exit:
            Exit(middleCode);
            break;

          case MiddleOperator.Jump:
            Jump(middleCode);
            break;

          case MiddleOperator.AssignRegister:
            LoadToRegister(middleCode);
            break;
          case MiddleOperator.InspectRegister:
            InspectRegister(middleCode);
            break;

          case MiddleOperator.JumpRegister:
            JumpToRegister(middleCode);
            break;

          case MiddleOperator.Interrupt:
            Interrupt(middleCode);
            break;

          case MiddleOperator.SysCall:
            SystemCall(middleCode);
            break;

          case MiddleOperator.Initializer:
            Initializer(middleCode);
            break;

          case MiddleOperator.InitializerZero:
            InitializerZero(middleCode);
            break;
```

In case of assignment, we inspect the type of the symbol to be assigned. If it is a struct or union, we call **StructUnionAssign**. Otherwise, the type must be integral, and we call **IntegralAssign**. Note that the assignment middle code instruction is not used for assignment of floating values. In that case, we instead top or pop the floating value stack.

```
          case MiddleOperator.AssignInit:
            StructUnionAssignInit(middleCode);
            break;

          case MiddleOperator.Assign: {
              Symbol symbol = (Symbol) middleCode[0];

              if (symbol.Type.IsStructOrUnion()) {
                StructUnionAssign(middleCode, middleIndex);
              }
              else {
                IntegralAssign(middleCode);
              }
            }
            break;

          case MiddleOperator.BitwiseAnd:
          case MiddleOperator.BitwiseOr:
          case MiddleOperator.BitwiseXOr:
          case MiddleOperator.ShiftLeft:
          case MiddleOperator.ShiftRight:
            IntegralBinary(middleCode);
            break;
```

In case of addition or subtract, we check the type of the left operand. If it is a floating value, we call **FloatingBinary**. Otherwise, the type is integral, and we call **IntegralBinary**.

```
          case MiddleOperator.Add:
```

```
          case MiddleOperator.Subtract: {
              Symbol resultSymbol = (Symbol) middleCode[1];

              if (resultSymbol.Type.IsFloating()) {
                FloatingBinary(middleCode);
              }
              else {
                IntegralBinary(middleCode);
              }
            }
            break;
```

We have a similar case when performing multiplication or division. If case of floating values, we call **FloatingBinary** (the same method as in the floating addition and subtracting case). Otherwise, the type is integral, and we call **IntegralMultiply**.

```
          case MiddleOperator.Multiply:
          case MiddleOperator.Divide:
          case MiddleOperator.Modulo: {
              Symbol resultSymbol = (Symbol) middleCode[0];

              if (resultSymbol.Type.IsFloating()) {
                FloatingBinary(middleCode);
              }
              else {
                IntegralMultiply(middleCode);
              }
            }
            break;

          case MiddleOperator.Carry:
          case MiddleOperator.NotCarry:
            CarryExpression(middleCode);
            break;
```

In case of equality and relational operators, we call **FloatingRelation** in case of floating values and **IntegralBinary** otherwise.

```
          case MiddleOperator.Equal:
          case MiddleOperator.NotEqual:
          case MiddleOperator.LessThan:
          case MiddleOperator.LessThanEqual:
          case MiddleOperator.GreaterThan:
          case MiddleOperator.GreaterThanEqual: {
              Symbol leftSymbol = (Symbol) middleCode[1];

              if (leftSymbol.Type.IsFloating()) {
                FloatingRelation(middleCode);
              }
              else {
                IntegralBinary(middleCode);
              }
            }
            break;

          case MiddleOperator.Case:
            Case(middleCode);
            break;
          case MiddleOperator.CaseEnd:
            CaseEnd(middleCode);
            break;
```

The same goes for the unary operations. In case of floating type we call **FloatingUnary**, and in case of integral type we call **InategralUnary**.

```
          case MiddleOperator.Plus:
          case MiddleOperator.Minus:
          case MiddleOperator.BitwiseNot: {
              Symbol resultSymbol = (Symbol) middleCode[0];

              if (resultSymbol.Type.IsFloating()) {
                FloatingUnary(middleCode);
              }
              else {
                IntegralUnary(middleCode);
              }
            }
            break;

          case MiddleOperator.Address:
            Address(middleCode);
            break;

          case MiddleOperator.Dereference: {
              Symbol symbol = (Symbol) middleCode[1];

              if (symbol.Type.IsFloating()) {
                FloatingDereference(middleCode);
              }
              else {
                IntegralDereference(middleCode);
              }
            }
            break;

          case MiddleOperator.DecreaseStack:
            Assert.ErrorXXX((--m_floatStackSize) >= 0);
            break;

          case MiddleOperator.CheckTrackMapFloatStack:
            Assert.ErrorXXX((m_trackMap.Count == 0) &&
                           (m_floatStackSize == 0));
            break;

          case MiddleOperator.PushZero:
            PushSymbol(new Symbol(Type.DoubleType, (decimal) 0));
            break;

          case MiddleOperator.PushOne:
            PushSymbol(new Symbol(Type.DoubleType, (decimal) 1));
            break;

          case MiddleOperator.PushFloat:
            PushSymbol((Symbol) middleCode[0]);
```

```
  break;

case MiddleOperator.TopFloat:
  TopPopSymbol((Symbol) middleCode[0], TopOrPop.Top);
  break;

case MiddleOperator.PopFloat:
  TopPopSymbol((Symbol) middleCode[0], TopOrPop.Pop);
  break;

case MiddleOperator.PopEmpty:
  PopEmpty();
  break;

case MiddleOperator.IntegralToIntegral:
  IntegralToIntegral(middleCode, middleIndex);
  break;

case MiddleOperator.IntegralToFloating:
  IntegralToFloating(middleCode);
  break;

case MiddleOperator.FloatingToIntegral:
  FloatingToIntegral(middleCode);
  break;

case MiddleOperator.ParameterInitSize:
  StructUnionParameterInit(middleCode);
  break;

case MiddleOperator.Parameter: {
    Symbol paramSymbol = (Symbol) middleCode[2];

    if (paramSymbol.Type.IsFloating()) {
      FloatingParameter(middleCode);
    }
    else if (paramSymbol.Type.IsStructOrUnion()) {
      StructUnionParameter(middleCode, middleIndex);
    }
    else {
      IntegralParameter(middleCode);
    }
  }
  break;

case MiddleOperator.GetReturnValue: {
    Symbol returnSymbol = (Symbol) middleCode[0];

    if (returnSymbol.Type.IsStructOrUnion()) {
      StructUnionGetReturnValue(middleCode);
    }
    else if (returnSymbol.Type.IsFloating()) {
      Assert.Error((++m_floatStackSize) <= FloatingStackMaxSize,
                   null, Message.Floating_stack_overflow);
    }
    else {
      IntegralGetReturnValue(middleCode);
```

```
          }
        }
        break;
      }
    }
  }
```

12.6.2. Track Set Generation

When the assembly code has been generated, a lot of tracks has been added to the assembly code. The task of the **TrackSet** method is to pick up the tracks from the assembly code and generate the set of tracks. Each track holds a list of entries, specifying the positions of the track in the code.

```
private ISet<Track> TrackSet() {
  ISet<Track> trackSet = new HashSet<Track>();

  for (int index = 0; index < m_assemblyCodeList.Count; ++index) {
    AssemblyCode assemblyCode = m_assemblyCodeList[index];
```

In case of the set-track-size instruction, we set the current size of its track and replace the instruction with an empty instruction. The current size is then used when adding entries to the track so that the registers in the end are given the correct size.

```
    if (assemblyCode.Operator == AssemblyOperator.set_track_size) {
      Track track = (Track)assemblyCode[0];

      if (assemblyCode[1] is int) {
        track.MaxSize = (int) assemblyCode[1];
      }
      else {
        track.MaxSize = ((Track) assemblyCode[1]).MaxSize;
      }
    }
```

In all other cases, we exam the operators of the instruction by calling **CheckTrack**.

```
    else {
      CheckTrack(trackSet, assemblyCode, 0, index);
      CheckTrack(trackSet, assemblyCode, 1, index);
      CheckTrack(trackSet, assemblyCode, 2, index);
    }
  }

  return trackSet;
}
```

The **CheckTrack** method takes the track set, one of the operands in an assembly code instruction, the position of the operand (zero, one, or two), and the index of the assembly code instruction in the assembly code list. We need to set the index of the assembly code list in the track in order to later decide whether tracks overlap.

```
private void CheckTrack(ISet<Track> trackSet, AssemblyCode assemblyCode,
                        int position, int index) {
  if (assemblyCode[position] is Track) {
    Track track = (Track) assemblyCode[position];
    trackSet.Add(track);
    track.Index = index;
  }
```

```
    }
```

12.6.3. Function Calls

The **BaseRegister** method returns the base register of a symbol representing an auto or register variable or a parameter. However, the symbol can also be null, in case of an unnamed parameter. The method returns the regular frame pointer or the variadic frame pointer. Note that we always use the regular frame pointer in regular functions.

```
    public Register BaseRegister(Symbol symbol) {
      Assert.ErrorXXX((symbol == null) || symbol.IsAutoOrRegister());
```

We return the variadic frame pointer if the function is variadic and the symbol is not a parameter. If the symbol is null, it represents an unnamed parameter in which case the variadic frame pointer is returned.

```
      if (SymbolTable.CurrentFunction.Type.IsVariadic() &&
          (symbol != null) && !symbol.IsParameter()) {
        return AssemblyCode.VariadicFrameRegister;
      }
```

If the function is not variadic, or if the symbol is a parameter, we return the regular frame pointer. The activation record is organized so that the parameters are located before the variables, with potential extra parameter between the regular parameters and the variables. Therefore, we use the regular frame pointer for parameters in both variadic and regular functions, and the variadic frame pointer for variables in variadic functions since they are located after the extra parameters.

```
      else {
        return AssemblyCode.RegularFrameRegister;
      }
    }
```

The **FunctionPreCall** methods is called before the actual function call, and before the parameter list of the function call. The idea is that each expression in the parameters shall have access to all the registers and the whole floating-point value stack, that no parameter values are stored in the registers or at the floating-point stack. Therefore, **FunctionPreCall** storer the values currently stored in registers and on the floating-point value stack at the activation record. The **FunctionPostCall** method below restores the values after the function call.

The **m_totalExtraSize** field keeps track of the current record size, which increases with nested calls. The **m_recordSizeStack** field holds the record sizes of nested calls. Technically, we could manage with only the stack, and sum the record sizes to find the total record size. However, let us include **m_totalExtraSize** for efficiency reasons.

```
    private int m_totalExtraSize = 0;
    private Stack<int> m_recordSizeStack = new Stack<int>();
```

As mentioned above, each parameter expression shall have access to all registers. Therefore, we need to save the track map of each nested call in the **m_trackMapStack** field. We need also keep track of the position of each saved register on the activation record in **m_registerMapStack**. Both stacks are popped in the **PostFunctionCall** method below.

```
    private Stack<IDictionary<Symbol,Track>> m_trackMapStack =
      new Stack<IDictionary<Symbol,Track>>();
    private Stack<IDictionary<Track,int>> m_registerMapStack =
      new Stack<IDictionary<Track,int>>();
```

In **FunctionPreCall**, we start by obtaining the base register, which is the regular or variadic frame pointer, depending on whether the function is variadic. The **recordSize** variable is the original size of the activation record while the **extraSize** field holds the size of the extra values in the registers and on the floating-point value stack.

```
public void FunctionPreCall(MiddleCode middleCode) {
  Register baseRegister = BaseRegister(null);
  int recordSize = (int) middleCode[0], extraSize = 0;
```

We load the post map with the tracks of the register with their locations on the activation record. For each register to be stored on the activation record, we add a **mov** instruction. The **extraSize** field is increased for each register.

```
  int totalSize = 0;
  int doubleTypeSize = Type.DoubleType.Size();
  foreach (int size in m_topStack) {
    totalSize += size;
  }
  extraSize += totalSize * doubleTypeSize;

  IDictionary<Track,int> registerMap = new Dictionary<Track,int>();
  foreach (KeyValuePair<Symbol, Track> pair in m_trackMap) {
    Track track = pair.Value;
    AddAssemblyCode(AssemblyOperator.mov, baseRegister,
                    recordSize + extraSize, track);
    registerMap.Add(track, recordSize + extraSize);
    Symbol symbol = pair.Key;
    extraSize += symbol.Type.Size();
  }
```

After the registers have been stored on the activation record, we store the values of the floating-point value stack on the activation record. Again, the **extraSize** field is being increased with the size of the values.

```
  int topSize = m_floatStackSize - totalSize;
  m_topStack.Push(topSize);

  for (int count = 0; count < topSize; ++count) {
    AddAssemblyCode(AssemblyOperator.fstp_qword, baseRegister,
                    recordSize + extraSize);
    extraSize += doubleTypeSize;
  }
```

When the values have been stored on the activation record, we need to store the register map for the registers to be restored after the function call in the **PostFunctionCall** method below.

```
  m_registerMapStack.Push(registerMap);
  m_recordSizeStack.Push(extraSize);
```

We also use the **m_totalExtraSize** field to store the total extra size for the in case of nested calls. Technically, we do not need this field since we could obtain the same value by summarizing the values pushed on the **m_recordSizeStack** stack. However, it has been included for efficiency reasons.

```
  m_totalExtraSize += extraSize;
```

The track map is pushed on the stack, and a new object is instantiated as the new track map. The track map will be restored after the function call.

```
  m_trackMapStack.Push(m_trackMap);
  m_trackMap = new Dictionary<Symbol, Track>();
```

```
    }
```

The **m_returnFloating** is set to true if the function returns a floating value. In case that the value is stored on the floating-point value stack and need to be considered by the **PostCallFunction** method below.

```
    private bool m_returnFloating = false;
```

When calling a function there are several cases to consider: the function symbol holding the function to be called may be a proper function or a pointer to a function, and the caller or callee function (or both) may be variadic.

```
    public void FunctionCall(MiddleCode middleCode, int index) {
      int recordSize = ((int) middleCode[0]) + m_totalExtraSize;
      Symbol calleeSymbol = (Symbol) middleCode[1];
      int extraSize = (int) middleCode[2];
```

The type is the function or, in case of a pointer the pointer type, the function the pointer points at.

```
      Type calleeType = calleeSymbol.Type.IsFunction()
                      ? calleeSymbol.Type : calleeSymbol.Type.PointerType;
```

Both the caller and callee function may be variadic.

```
      bool callerVariadic = SymbolTable.CurrentFunction.Type.IsVariadic(),
           calleeVariadic = calleeType.IsVariadic();
```

The frame register is the variadic frame pointer register if the caller function is variadic. If it not, it is the regular frame pointer register.

```
      Register frameRegister = callerVariadic
                             ? AssemblyCode.VariadicFrameRegister
                             : AssemblyCode.RegularFrameRegister;
```

We start by adding the assembly code instruction for the return address. To begin with, we set it to the index of the next middle code instruction. The address will later be changed by the **WindowsJumpInfo** method to a proper assembly code address.

```
      AddAssemblyCode(AssemblyOperator.return_address, frameRegister,
                      recordSize + SymbolTable.ReturnAddressOffset,
                      (BigInteger) (index + 1));
```

We set the current regular frame pointer and, in case of a variadic caller function, the variadic pointer to their offset in the activations record of the callee function. In this way, we can reset their values in accordance with the caller function returning from the function call.

```
      AddAssemblyCode(AssemblyOperator.mov, frameRegister,
                      recordSize + SymbolTable.RegularFrameOffset,
                      AssemblyCode.RegularFrameRegister);

      if (callerVariadic) {
        AddAssemblyCode(AssemblyOperator.mov, frameRegister,
                        recordSize + SymbolTable.VariadicFrameOffset,
                        AssemblyCode.VariadicFrameRegister);
      }
```

If the callee function is a pointer to a function, we load its value into the **jumpTrack** track. We must load the value into the track before we increase the frame register below.

```
      Track jumpTrack = null;
      if (!calleeSymbol.Type.IsFunction()) {
        jumpTrack = LoadValueToRegister(calleeSymbol);
```

```
    }
```

We add the size of the caller function's activation record to the frame pointer, so that it pointes at the activation record of the callee function.

```
    AddAssemblyCode(AssemblyOperator.add, frameRegister, // add di, 10
                    (BigInteger) recordSize);
```

If the caller function is variadic, the frame pointer that we just added is the variadic frame pointer, and we also need to set the regular frame pointer to the same value.

```
    if (callerVariadic) { // mov bp, di
      AddAssemblyCode(AssemblyOperator.mov,
                      AssemblyCode.RegularFrameRegister,
                      AssemblyCode.VariadicFrameRegister);
    }
```

If the callee function, but not the caller function, is variadic, the frame pointer we just added is the variadic frame pointer, and we also need to set the regular frame pointer to the same value.

```
    else if (calleeVariadic) {
      AddAssemblyCode(AssemblyOperator.mov,
                      AssemblyCode.VariadicFrameRegister,
                      AssemblyCode.RegularFrameRegister);
    }
```

If the callee function is variadic, and the extra size is more than zero (if there are extra arguments in the function calls that are not matched to the declared parameters), we add the size to the variadic frame pointer. The callee function will have both a regular frame pointer and a variadic frame pointer. However, the variadic pointer will point to a higher address to give space to the extra arguments.

```
    if (calleeVariadic && (extraSize > 0)) {
      AddAssemblyCode(AssemblyOperator.add,
                      AssemblyCode.VariadicFrameRegister,
                      (BigInteger) extraSize);
    }
```

Finally. if the callee function is a function (not a pointer to a function) we a middle code call instruction. The **m_returnFloating** field is set to true if the callee function returns a floating value. It will later be inspected by the **FunctionPostCall** method.

```
    if (calleeSymbol.Type.IsFunction()) {
      AddAssemblyCode(AssemblyOperator.call, calleeSymbol.UniqueName);
      m_returnFloating = calleeSymbol.Type.ReturnType.IsFloating();
    }
```

If the callee function is a pointer to a function and jump to the address stored in **jumpTrack**.

```
    else {
      AddAssemblyCode(AssemblyOperator.jmp, jumpTrack);
      m_returnFloating =
        calleeSymbol.Type.PointerType.ReturnType.IsFloating();
    }
  }
```

The **FunctionPostCall** method is called after a function call. Its task is to restore the actions performed by **FunctionPreCall** above.

```
  public void FunctionPostCall(MiddleCode middleCode) {
    Register baseRegister = BaseRegister(null);
```

```
      m_trackMap = m_trackMapStack.Pop();
      IDictionary<Track,int> registerMap = m_registerMapStack.Pop();
```

First, we iterate through the post map defined in **FunctionPreCall** before the function call. We load the values stored in the map into the registers.

```
      foreach (KeyValuePair<Track,int> pair in registerMap) {
        Track track = pair.Key;
        int offset = pair.Value;
        AddAssemblyCode(AssemblyOperator.mov, track, baseRegister,offset);
      }
```

Then we investigate the floating value stack. If floating stack value before the function call was non-empty, we need to restore it.

```
      Assert.ErrorXXX(m_topStack.Count > 0);
      int topSize = m_topStack.Pop();

      if (topSize > 0) {
        int recordOffset = (int) middleCode[0];
        int doubleTypeSize = Type.DoubleType.Size();
        int recordSize = m_recordSizeStack.Pop();
```

If the return value of the previous function call is a floating value, we need to temporary restore it when we restore the floating value stack, since the return value shall be placed at the top of the floating stack.

```
        if (m_returnFloating) {
          AddAssemblyCode(AssemblyOperator.fstp_qword, baseRegister,
                          recordOffset + recordSize);
        }
```

We iterate through the floating values, as they are located above the activation record. We begin by pushing the top-most value and iterate to the bottom-most value.

```
        int currentOffset = recordOffset + recordSize;
        for (int count = 0; count < topSize; ++count) {
          currentOffset -= doubleTypeSize;
          AddAssemblyCode(AssemblyOperator.fld_qword, baseRegister,
                          currentOffset);
        }
```

If the function returned a floating value, we push it from its temporary location to the top of the floating value stack.

```
        if (m_returnFloating) {
          AddAssemblyCode(AssemblyOperator.fld_qword, baseRegister,
                          recordOffset + recordSize);
        }

        m_totalExtraSize -= recordSize;
      }
      else {
        m_totalExtraSize -= m_recordSizeStack.Pop();
      }
    }
```

12.6.4. Loading Values into Registers

The **LoadValueToRegister** method loads the value of a symbol into a register. A specific register to be loaded may be given. However, the returned register is represented by a track that will be replaced by a proper register by the register allocator.

```
    public Track LoadValueToRegister(Symbol symbol,
                                     Register? register = null) {
```

If the register is specified, we must check that no other value is stored in that register. If it is, we must move it to another register.

```
      if (register != null) {
        CheckRegister(symbol, register.Value);
      }
```

Then we check if the value is already stored in a register. If it is, we look up it track and compare the specified register with the register of the track.

```
      Track track;
      if (m_trackMap.TryGetValue(symbol, out track)) {
        m_trackMap.Remove(symbol);
```

If the registers are not null do not overlap, we need to remove the previous register from the track. If we find that the register is associated to a track, we add a **mov** instruction that moves the value to a new (yet unknown) register. The track holding that register is then returned.

```
        if ((register != null) && (track.Register != null) &&
            !AssemblyCode.RegisterOverlap(register, track.Register)) {
          Track newTrack = new Track(symbol, register.Value);
          AddAssemblyCode(AssemblyOperator.set_track_size,
                          newTrack, track);
          AddAssemblyCode(AssemblyOperator.mov, newTrack, track);
          return newTrack;
        }
```

If there is no non-overlapping register, we set the register of the track, if it is not null, and return the track.

```
        else {
          if (register != null) {
            track.Register = register;
          }

          return track;
        }
      }
```

If the track register is null, we assign it the specified register (which may or may not be null).

However, if both the specified register and the track register is non-null, and they do not overlap each other, we need to move the value of the track register to specified register. We create and return a new track for the specified register.

If the symbol does not already have a track, we create a new track for it.

Basically, we have three different cases: the symbol holds an integer value, it holds and address, or a value stored at an address. In case of an integer value, we simply load it int to the register.

```
        if (symbol.Value is BigInteger)  {
          AddAssemblyCode(AssemblyOperator.mov, track, symbol.Value);
        }
```

If the symbol is an array, function, or string, or if it holds a static address, we load the address of the value into the register, not the value itself. We load the base of the symbol into the register. The base is the symbol's unique name if it is extern or static, the regular or variadic frame register in case of an auto or register array (a function or string is always static), and the name of a static address. If the offset of the address is non-zero, we add it to the register.

```
        else if (symbol.Type.IsArrayFunctionOrString() ||
                 (symbol.Value is StaticAddress)) {
          AddAssemblyCode(AssemblyOperator.mov, track,
                          Base(symbol));

          int offset = Offset(symbol);
          if (offset != 0) {
            AddAssemblyCode(AssemblyOperator.add, track,
                            (BigInteger) offset);
          }
        }
```

In all other cases, we load the value of the symbol into the register. The base is the symbol's unique name if it is extern or static, and the regular or variadic frame register is it is auto or register.

```
        else {
          AddAssemblyCode(AssemblyOperator.mov, track,
                          Base(symbol), Offset(symbol));
        }

        return track;
      }
    }
```

If the symbol is a static address, we load its address into the register. At the moment, we load its name into the register and add its offset, if more than zero. The name will later be replaced by the address by the linker.

If the symbol is an array, we load its base into the register and add its offset, if more than zero. The base may be the unique name of a static symbol, or the regular or variadic frame pointer in case of a parameter or local variable.

If the symbol is not a constant, an array, or a static function, struct, or union, we load the value of the symbol to the register with the help of the symbol's base and offset. The base may be the unique name of a static symbol, or the regular or variadic frame pointer.

The **CheckRegister** method checks whether a symbol value is stored in the register

```
    public void CheckRegister(Symbol symbol, Register register) {
      foreach (KeyValuePair<Symbol,Track> entry in m_trackMap) {
        Symbol oldSymbol = entry.Key;
        Track oldTrack = entry.Value;
```

We iterate through the track map and check if any track holds a register overlapping the specified register.

```
        if (!oldSymbol.Equals(symbol) &&
            AssemblyCode.RegisterOverlap(register, oldTrack.Register)) {
```

If the find a match, we create a new track and insert instructions for setting the size of the track and moving the value from the previous track to the new track.

```
          Track newTrack = new Track(oldSymbol);
          m_trackMap[oldSymbol] = newTrack;

          int lastLine;
          for (lastLine = m_assemblyCodeList.Count - 1; lastLine >= 0;
               --lastLine) {
            AssemblyCode assemblyCode = m_assemblyCodeList[lastLine];
            if (oldTrack.Equals(assemblyCode[0])) {
              break;
            }
          }
          Assert.ErrorXXX(lastLine >= 0);

          AssemblyCode setCode =
            new AssemblyCode(AssemblyOperator.set_track_size,
                             newTrack, oldTrack);
          AssemblyCode movCode =
            new AssemblyCode(AssemblyOperator.mov, newTrack, oldTrack);
          m_assemblyCodeList.Insert(lastLine + 1, setCode);
          m_assemblyCodeList.Insert(lastLine + 2, movCode);
          break;
        }
      }
    }
```

The **LoadAddressToRegister** loads the address of value, rather the value itself, into a register.

```
    public Track LoadAddressToRegister(Symbol symbol,
                                       Register? register = null) {
```

If the address symbol of the symbol is not null, we simply call **LoadValueToRegister** with the address symbol, and returns the track. However, we must mark the track as a pointer, which means that the set of possible register re will be restricted.

```
      if (symbol.AddressSymbol != null) {
        Track addressTrack = LoadValueToRegister(symbol.AddressSymbol);
        addressTrack.Pointer = true;
        return addressTrack;
      }
```

If the address symbol is null, we need to obtain the address manually. We start by mov the base of the symbol to the register. The base is the regular or variadic frame pointer in case of an auto or register storage, since the value is located on the activation record, or the name of the symbol in case of extern or static storage. However, the symbol may have a static address its value. In that case, the base is the name of the static address.

```
      else {
        Symbol addressSymbol = new Symbol(new Type(symbol.Type));
        Track addressTrack = new Track(addressSymbol, register);
        Assert.ErrorXXX((addressTrack.Register == null) ||
                        RegisterAllocator.VariadicFunctionPointerRegisterSet.
                        Contains(addressTrack.Register.Value));
        addressTrack.Pointer = true;
        Assert.ErrorXXX(!(symbol.Value is BigInteger));
        AddAssemblyCode(AssemblyOperator.mov, addressTrack, Base(symbol));
```

Then we add the offset of the symbol, which is always non-zero in case of auto or register storage and zero in case of extern or static storage. In case a static address, the offset may be zero or non-zero.

```
        int offset = Offset(symbol);
        if (offset != 0) {
          AddAssemblyCode(AssemblyOperator.add, addressTrack,
                          (BigInteger) offset);
        }

        return addressTrack;
      }
    }
```

12.6.5. Return, Exit, and Jump

The **Return** method generates the code for returning from a function call. We catch the return address (which we must do before we reset the pointers), reset the regular frame pointer and variadic frame pointer, and jump back to the return address.

```
    public void Return(MiddleCode middleCode, int middleIndex) {
      if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {
        Assert.ErrorXXX(m_floatStackSize == 0);
        AddAssemblyCode(AssemblyOperator.cmp,
                        AssemblyCode.RegularFrameRegister,
                        SymbolTable.ReturnAddressOffset,
                        BigInteger.Zero, TypeSize.PointerSize);

        AssemblyCode jumpCode =
          AddAssemblyCode(AssemblyOperator.je, null, null, middleIndex + 1);
        Return();
      }
      else {
        SetReturnValue(middleCode);
        Assert.ErrorXXX(m_floatStackSize == 0);
        Return();
      }
    }
```

The **SetReturnValue** method calls **StructUnionSetReturnValue** or **IntegralSetReturnValue** in case of a struct or union value, or an integral value. In case of a floating-point value we do nothing, since the return value is already properly placed on the top of the floating-point stack.

```
    private void SetReturnValue(MiddleCode middleCode) {
      if (middleCode[1] != null) {
        Symbol returnSymbol = (Symbol) middleCode[1];

        if (returnSymbol.Type.IsStructOrUnion()) {
          StructUnionSetReturnValue(middleCode);
        }
        else if (returnSymbol.Type.IsFloating()) {
          Assert.ErrorXXX((--m_floatStackSize) == 0);
        }
        else {
          IntegralSetReturnValue(middleCode);
        }
      }
    }
```

The **Return** method adds assembly code for return control back to the calling function. We restore the frame and variadic pointers of the calling function and jump back to the calling function. Note that we have to load the return jump address into register before we restore the frame pointer, otherwise we would obtain the return address of the calling function.

```
private void Return() {
  Track track = new Track(Type.VoidPointerType);
  AddAssemblyCode(AssemblyOperator.mov, track,
                  AssemblyCode.RegularFrameRegister,
                  SymbolTable.ReturnAddressOffset);
  AddAssemblyCode(AssemblyOperator.mov,
                      AssemblyCode.VariadicFrameRegister,
                      AssemblyCode.RegularFrameRegister,
                      SymbolTable.VariadicFrameOffset);
  AddAssemblyCode(AssemblyOperator.mov, AssemblyCode.RegularFrameRegister,
                      AssemblyCode.RegularFrameRegister,
                      SymbolTable.RegularFrameOffset);
  AddAssemblyCode(AssemblyOperator.jmp, track);
}
```

The **IntegralGetReturnValue** method stores the return register in a track.

```
public void IntegralGetReturnValue(MiddleCode middleCode) {
  Symbol returnSymbol = (Symbol) middleCode[0];
  Register returnRegister =
    AssemblyCode.RegisterToSize(AssemblyCode.ReturnValueRegister,
                                returnSymbol.Type.Size());
```

We call **CheckRegister** to make sure that if the register already has a value, it is moved to another register.

```
  CheckRegister(returnSymbol, returnRegister);
  Track returnTrack = new Track(returnSymbol, returnRegister);
  m_trackMap.Add(returnSymbol, returnTrack);
  AddAssemblyCode(AssemblyOperator.empty, returnTrack);
}
```

The **IntegralSetReturnValue** loads the symbol of the expression into the return register.

```
public void IntegralSetReturnValue(MiddleCode middleCode) {
  Symbol returnSymbol = (Symbol) middleCode[1];
  Register returnRegister =
    AssemblyCode.RegisterToSize(AssemblyCode.ReturnValueRegister,
                                returnSymbol.Type.SizeArray());
  LoadValueToRegister(returnSymbol, returnRegister);
  m_trackMap.Remove(returnSymbol);
}
```

The **Exit** method generates code for exit the execution of the program. However, the code is different depending on whether we generate Windows or Linux code, and whether there is an exit symbol given.

```
public void Exit(MiddleCode middleCode) {
  Symbol exitSymbol = (Symbol) middleCode[0];
```

If the exit symbol is not null, we load its value into the **rdi** register If it is null, we just load zero into the register. This value will be returned to the surrounding operation system.

```
  if (Start.Linux) {
    if (exitSymbol != null) {
      LoadValueToRegister(exitSymbol, Register.rdi);
    }
```

```
        else {
          AddAssemblyCode(AssemblyOperator.mov, Register.rdi,
                          BigInteger.Zero);
        }
```

The Linux exit code is to load value 60 to register **rax** and do a system call.

```
        AddAssemblyCode(AssemblyOperator.mov, Register.rax,
                        (BigInteger) 60); // 0x3C
        AddAssemblyCode(AssemblyOperator.syscall);
      }
```

The code for the Windows envisonment is described in Chapter 13.

```
      if (Start.Windows) {
        // ...
      }
    }
```

The **Jump** method simply add a jump instruction with the index a middle code instruction as target.

```
    public void Jump(MiddleCode middleCode) {
      int jumpTarget = (int) middleCode[0];
      AddAssemblyCode(AssemblyOperator.jmp, null, null, jumpTarget);
    }
```

12.6.6. Load and Inspect Registers

The **LoadToRegister** method adds assembly code that loads the value of a symbol into a specified register. This method, as well as **InspectRegister**, **CarryExpression**, **JumpRegister**, **Interrupt**, and **SystemCall**, is only used by the standard library in internal system calls.

```
    private ISet<Track> m_syscallSet = new HashSet<Track>();

    public void LoadToRegister(MiddleCode middleCode) {
      Register register = (Register) middleCode[0];
      Symbol symbol = (Symbol) middleCode[1];
      Track track = LoadValueToRegister(symbol, register);
      m_syscallSet.Add(track);
    }
```

The **InspectRegister** method loads the value of a register to a symbol. More specifically, it adds a track holding the symbol and register to the track map.

```
    public void InspectRegister(MiddleCode middleCode) {
      Symbol symbol = (Symbol) middleCode[0];
      Register register = (Register) middleCode[1];
      Track track = new Track(symbol, register);
      m_trackMap.Add(symbol, track);
    }
```

The **CarryExpression** method adds assembly code that jumps to a middle code target if the carry flag is set. The target will later be changed by the **WindowsJumpInfo** method to a proper assembly code target.

```
    public void CarryExpression(MiddleCode middleCode) {
      AssemblyOperator objectOperator =
        m_middleToIntegralMap[middleCode.Operator];
      int jumpTarget = (int) middleCode[0];
      AddAssemblyCode(objectOperator, null, null, jumpTarget);
    }
```

The **JumpRegister** method adds assembly code that jumps to the address stored in a register.

```
public void JumpToRegister(MiddleCode middleCode) {
  Register jumpRegister = (Register) middleCode[0];
  AddAssemblyCode(AssemblyOperator.jmp, jumpRegister);
}
```

The **Interrupt** method adds code that performs an interrupt call. Before the call, we clear the system call set.

```
public void Interrupt(MiddleCode middleCode) {
  foreach (Track track in m_syscallSet) {
    AddAssemblyCode(AssemblyOperator.empty, track);
  }

  AddAssemblyCode(AssemblyOperator.interrupt,
                  (BigInteger) middleCode[0]);
  m_trackMap.Clear();
}
```

The **SystemCall** method adds code that performs an interrupt call. Like the interrupt case, we clear the system call set before the call.

```
public void SystemCall(MiddleCode middleCode) {
  foreach (Track track in m_syscallSet) {
    AddAssemblyCode(AssemblyOperator.empty, track);
  }

  AddAssemblyCode(AssemblyOperator.syscall);
  m_trackMap.Clear();
}
```

12.6.7. Initialization

The **Initializer** method adds code initializing a value. The value becomes static block, or a part of a static block, which

```
private void Initializer(MiddleCode middleCode) {
  Sort sort = (Sort) middleCode[0];
  object value = middleCode[1];
```

If the value is a static address, we add code for defining the address with its name and offset.

```
  if (value is StaticAddress) {
    StaticAddress staticAddress = (StaticAddress) value;
    string name = staticAddress.UniqueName;
    int offset = staticAddress.Offset; // dw name + offset
    AddAssemblyCode(AssemblyOperator.define_address, name, offset);
  }
```

Otherwise, we add code for defining the value. The initialization instruction will later be transformed to a sequence of instructions in the Linux case and memory block in the Windows case.

```
  else {
    AddAssemblyCode(AssemblyOperator.define_value, sort, value);
  }
}
```

The **InitializerZero** adds an instruction for a sequence of zero values, each holding one byte. Like **Initializer** above, the initialization instruction will later be transformed to a sequence of instructions in the Linux case and memory block in the Windows case.

```
private void InitializerZero(MiddleCode middleCode) {
  int size = (int) middleCode[0];
  Assert.ErrorXXX(size > 0);
  AddAssemblyCode(AssemblyOperator.define_zero_sequence, size);
}
```

12.6.8. Integral Multiplication, Division, and Modulo

The **IntegralMultiply** method adds assembly code for the integral operators multiplication, division, and module. These operations differ from the previous integral operations in that way that the value of the left operand is always stored in a specific register. However, the right symbol is given in the operation. The value of the right symbol is given in the operations. It can be stored in a register or on a memory address. However, unlike the previous integral operations, we cannot give an integer value directly. Instead, we have to store the value on an address, and give the address to the operation.

Moreover, there are in fact four registers involved in the operation. The specific registers depend on the type size.

The register where we store the value of the left symbol.

The register holding the upper part of the left symbol value. We ignore the value in this register, but we must set it to zero and mark in the track map we have in fact altered its value.

The result of multiplication or division, which is ignored by modulo.

The result of module, which is ignored by multiplication or division.

Even if one of the last two registers is ignored, we still need to mark in the track map that is has in fact been assigned a new value.

The **m_leftRegisterMap** map hold the registers where we store the value of the left symbol. The **m_zeroRegisterMap** map holds the registers that are to be set to zero before the operation. The **m_productQuintentRegisterMap** map holds the registers where the product or quintet is stored after a multiplication or division operations, while **m_remainderRegisterMap** holds the registers of the remainder of a modulo operations.

```
public static IDictionary<int,Register> m_leftRegisterMap =
  new Dictionary<int,Register>() {{1, Register.al}, {2, Register.ax},
                                  {4, Register.eax}, {8, Register.rax}};

public static IDictionary<int,Register> m_zeroRegisterMap =
  new Dictionary<int,Register>() {{1, Register.ah}, {2, Register.dx},
                                  {4, Register.edx}, {8, Register.rdx}};

public static IDictionary<int,Register> m_productQuintentRegisterMap =
  new Dictionary<int,Register>() {{1, Register.al}, {2, Register.ax},
                                  {4, Register.eax}, {8, Register.rax}};

public static IDictionary<int,Register> m_remainderRegisterMap =
  new Dictionary<int,Register>() {{1, Register.ah}, {2, Register.dx},
                                  {4, Register.edx}, {8, Register.rdx}};
```

The **IntegralMultiply** method adds code for multiplication operations. To begin with, we load the value of the left symbol into the register given by **m_leftRegisterMap**.

```
public void IntegralMultiply(MiddleCode middleCode) {
  Symbol leftSymbol = (Symbol) middleCode[1];
  int typeSize = leftSymbol.Type.SizeArray();
  Register leftRegister = m_leftRegisterMap[typeSize];
  Track leftTrack = LoadValueToRegister(leftSymbol, leftRegister);
```

Then we clear the zero register by perform the exclusive operation on itself, which is an effective way to clear a register, in order to provide current input to the operation.

```
  Register zeroRegister = m_zeroRegisterMap[typeSize];
  Track zeroTrack = new Track(leftSymbol, zeroRegister);
  AddAssemblyCode(AssemblyOperator.xor, zeroTrack, zeroTrack);
```

The we call **IntegralUnary** to load the right symbol into a suitable storage and perform the operation.

```
  Symbol rightSymbol = (Symbol) middleCode[2];
  IntegralUnary(middleCode.Operator, rightSymbol, rightSymbol);
```

When the operation is done, we need to find out which register to keep and which one to discard. We only use one of the registers, depending on the operation. But we also need to mark that the value of the discarded register has been modified, to prevent that another value is stored in the register during the operation.

```
  Register resultRegister, discardRegister;
```

In case of the modulo operation, the resulting register is picked from **m_remainderRegisterMap**, and the register to be discarded **m_productQuintentRegisterMap**. In case of multiplication or division, it is the other way around.

```
  if (middleCode.Operator == MiddleOperator.Modulo) {
    resultRegister = m_remainderRegisterMap[typeSize];
    discardRegister = m_productQuintentRegisterMap[typeSize];
  }
  else {
    resultRegister = m_productQuintentRegisterMap[typeSize];
    discardRegister = m_remainderRegisterMap[typeSize];
  }
```

We create a new track holding the result symbol and register. If the result symbol is a temporary variable, we store the track in the track map. We also add an empty instruction, to mark that the register has been altered by the operation.

```
  Symbol resultSymbol = (Symbol) middleCode[0];
  Track resultTrack = new Track(resultSymbol, resultRegister);

  if (resultSymbol.IsTemporary()) {
    Assert.ErrorXXX(resultSymbol.AddressSymbol == null);
    m_trackMap.Add(resultSymbol, resultTrack);
    AddAssemblyCode(AssemblyOperator.empty, resultTrack);
  }
```

If the result symbol is a regular variable, we store the value in the register at the address of the variable. In this case we do not need to add an empty instruction since the loading of the register marks that is has been in use.

```
  else {
    AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                    Offset(resultSymbol), resultTrack);
  }
```

However, we need to add an empty instruction for the discarded register, to mark that it has been altered.

```
      Track discaredTrack = new Track(resultSymbol, discardRegister);
      AddAssemblyCode(AssemblyOperator.empty, discaredTrack);
    }
```

12.6.1. Integral Assignment and Parameters

The **IntegralAssign** method generates code for assignment of an integral value. What makes it a bit complicated is that we must take into consideration of the assignment of a conditional expression. Both the values of the true and false expression shall be assigned to the same register.

```
    public void IntegralAssign(MiddleCode middleCode) {
      Symbol resultSymbol = (Symbol) middleCode[0],
             assignSymbol = (Symbol) middleCode[1];
      IntegralAssign(resultSymbol, assignSymbol);
    }
```

The **IntegralParameter** method is quite simple. We only need to create the symbol representing the parameter, and then call **IntegralAssign**.

```
    public void IntegralParameter(MiddleCode middleCode) {
      Type toType = (Type) middleCode[1];
      Symbol toSymbol = new Symbol(null, false, Storage.Auto, toType);
      Symbol fromSymbol = (Symbol) middleCode[2];
      int parameterOffset = (int) middleCode[0];
      toSymbol.Offset = m_totalExtraSize + parameterOffset;
      IntegralAssign(toSymbol, fromSymbol);
    }
```

The **IntegralAssign** method is called by **IntegralAssign** and **IntegralParameter**.

```
    public void IntegralAssign(Symbol resultSymbol, Symbol assignSymbol) {
      Track resultTrack = null, assignTrack = null;
      m_trackMap.TryGetValue(resultSymbol, out resultTrack);
      m_trackMap.TryGetValue(assignSymbol, out assignTrack);
      int typeSize = assignSymbol.Type.SizeArray();
```

If the result symbol is temporary and its address symbol is null, we are facing the assignment of a conditional expression.

```
      if (resultSymbol.IsTemporary()) {
        Assert.ErrorXXX(assignTrack == null);

        if (resultTrack == null) {
          resultTrack = new Track(resultSymbol);
          m_trackMap.Add(resultSymbol, resultTrack);
        }
```

If the assignment symbol is stored in the track map, it means that we assign the true expression of the conditional expression, we start by copying its track to the assignment track. Then we add the assignment track to the track map.

If the assignment symbol is not stored in the track map, we assign the false expression of the conditional expression, we begin by creating a track for the result symbol.

If the assign symbol is an integer value, we move it into the register.

```
        if (assignSymbol.Value is BigInteger) {
          AddAssemblyCode(AssemblyOperator.mov, resultTrack,
                          assignSymbol.Value);
        }
```

If the assign symbol is an array, function, string, or static address, we move its address rather than its value into the register.

```
        else if (assignSymbol.Type.IsArrayFunctionOrString() ||
                 (assignSymbol.Value is StaticAddress)) {
          AddAssemblyCode(AssemblyOperator.mov, resultTrack,
                          Base(assignSymbol));

          int offset = Offset(assignSymbol);
          if (offset != 0) {
            AddAssemblyCode(AssemblyOperator.add, resultTrack,
                            (BigInteger) offset);
          }
        }
```

Otherwise, we move the value of the symbol into the register.

```
        else {
          AddAssemblyCode(AssemblyOperator.mov, resultTrack,
                          Base(assignSymbol), Offset(assignSymbol));
        }
      }
      else {
        if (assignTrack != null) {
          AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                          Offset(resultSymbol), assignTrack);
          m_trackMap.Remove(assignSymbol);
        }
        else if (assignSymbol.Value is BigInteger) {
          BigInteger bigValue = (BigInteger) assignSymbol.Value;
          if (Start.Linux &&
              !((-2147483648 <= bigValue) && (bigValue <= 2147483647))) {
            assignTrack = new Track(assignSymbol);
            AddAssemblyCode(AssemblyOperator.mov, assignTrack,
                            assignSymbol.Value);
            AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                            Offset(resultSymbol), assignTrack);
          }
          else {
            AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                            Offset(resultSymbol), assignSymbol.Value,
                            typeSize);
          }
        }
        else if (assignSymbol.Type.IsArrayFunctionOrString() ||
                (assignSymbol.Value is StaticAddress)) {
          if (assignSymbol.AddressSymbol != null) {
            Track addressTrack =
              LoadValueToRegister(assignSymbol.AddressSymbol);
            addressTrack.Pointer = true;
            AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                            Offset(resultSymbol), addressTrack);
            if (assignSymbol.AddressOffset != 0) {
              AddAssemblyCode(AssemblyOperator.add, Base(resultSymbol),
                              Offset(resultSymbol), (BigInteger)
                              assignSymbol.AddressOffset, typeSize);
            }
          }
          else {
            AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                            Offset(resultSymbol), Base(assignSymbol),
                            TypeSize.PointerSize);

            int assignOffset = Offset(assignSymbol);
            if (assignOffset != 0) {
              AddAssemblyCode(AssemblyOperator.add, Base(resultSymbol),
                              Offset(resultSymbol), (BigInteger) assignOffset,
                              typeSize);
            }
          }
        }
        else {
          assignTrack = LoadValueToRegister(assignSymbol);
          AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                          Offset(resultSymbol), assignTrack);
        }
      }
    }
```

If the result symbol is not temporary, we move the value directly into the address of the result symbol.

In case of an integer value, there is a special case due to technical limitations if the size is eight bytes. In that case we have to load the value into a register and load the register into the address, since we cannot load an eight-byte value directly into an address.

If the value size is not eight-byte, we load the value directly into the address.

If the symbol is an array, function, string, or static address, we load the address rather than the value onto the address.

Otherwise, we need to load the value into a register, which we than load onto the address.

12.6.2. Unary Integral Operations

The **Unary** method is called to add code for unary subtraction and bitwise not. There is also unary addition, but it generates no code.

First, we need a map from middle code operators to assembly code operators for integral types. We will use the map in the **IntegralUnary** and **IntegralBinary** methods.

```
public static IDictionary<MiddleOperator,AssemblyOperator>
  m_middleToIntegralMap =
  new Dictionary<MiddleOperator, AssemblyOperator>() {
    {MiddleOperator.BitwiseNot, AssemblyOperator.not},
    {MiddleOperator.Minus, AssemblyOperator.neg},
    {MiddleOperator.Multiply, AssemblyOperator.imul},
    {MiddleOperator.Divide, AssemblyOperator.idiv},
    {MiddleOperator.Modulo, AssemblyOperator.idiv},
    {MiddleOperator.Assign, AssemblyOperator.mov},
    {MiddleOperator.Add, AssemblyOperator.add},
    {MiddleOperator.Subtract, AssemblyOperator.sub},
    {MiddleOperator.BitwiseAnd, AssemblyOperator.and},
    {MiddleOperator.BitwiseOr, AssemblyOperator.or},
    {MiddleOperator.BitwiseXOr, AssemblyOperator.xor},
    {MiddleOperator.ShiftLeft, AssemblyOperator.shl},
    {MiddleOperator.ShiftRight, AssemblyOperator.shr},
    {MiddleOperator.Equal, AssemblyOperator.je},
    {MiddleOperator.NotEqual, AssemblyOperator.jne},
    {MiddleOperator.Carry, AssemblyOperator.jc},
    {MiddleOperator.NotCarry, AssemblyOperator.jnc},
    {MiddleOperator.Compare, AssemblyOperator.cmp},
    {MiddleOperator.LessThan, AssemblyOperator.jl},
    {MiddleOperator.LessThanEqual,AssemblyOperator.jle},
    {MiddleOperator.GreaterThan, AssemblyOperator.jg},
    {MiddleOperator.GreaterThanEqual, AssemblyOperator.jge}},

  m_unsignedToIntegralMap =
  new Dictionary<MiddleOperator, AssemblyOperator>() {
    {MiddleOperator.Multiply, AssemblyOperator.mul},
    {MiddleOperator.Divide, AssemblyOperator.div},
    {MiddleOperator.Modulo, AssemblyOperator.div},
    {MiddleOperator.Equal, AssemblyOperator.je},
    {MiddleOperator.NotEqual, AssemblyOperator.jne},
    {MiddleOperator.LessThan, AssemblyOperator.jb},
    {MiddleOperator.LessThanEqual,AssemblyOperator.jbe},
    {MiddleOperator.GreaterThan, AssemblyOperator.ja},
    {MiddleOperator.GreaterThanEqual, AssemblyOperator.jae}};

public void IntegralUnary(MiddleCode middleCode) {
  Symbol resultSymbol = (Symbol) middleCode[0],
         unarySymbol = (Symbol) middleCode[1];
  IntegralUnary(middleCode.Operator, resultSymbol, unarySymbol);
}

public void IntegralUnary(MiddleOperator middleOperator,
                          Symbol resultSymbol, Symbol unarySymbol) {
  AssemblyOperator objectOperator;
  if (MiddleCode.IsMultiply(middleOperator) &&
      unarySymbol.Type.IsUnsigned()) {
    objectOperator = m_unsignedToIntegralMap[middleOperator];
  }
  else {
    objectOperator = m_middleToIntegralMap[middleOperator];
  }

  int typeSize = unarySymbol.Type.SizeArray();
  Track unaryTrack = null;
  if (unarySymbol.Value is BigInteger) {
    SymbolTable.StaticSet.Add(ConstantExpression.Value(unarySymbol));
    AddAssemblyCode(objectOperator, unarySymbol.UniqueName,
                    0, null, typeSize);
  }
  else if (m_trackMap.TryGetValue(unarySymbol, out unaryTrack)) {
    if (middleOperator != MiddleOperator.Plus) {
      AddAssemblyCode(objectOperator, unaryTrack);
    }
```

```
        m_trackMap.Remove(unarySymbol);
      }
```

We have to cases. If the result symbol equals the unary symbol, we do not need to use a register. We can just perform the operation directly on the address of the symbol. Unless the operator is unary addition, in which case we do nothing.

```
      else if (resultSymbol == unarySymbol) {
        Assert.ErrorXXX(unaryTrack == null);
        if (middleOperator != MiddleOperator.Plus) {
          AddAssemblyCode(objectOperator, Base(unarySymbol),
                          Offset(unarySymbol), null, typeSize);
        }
      }
```

If the result symbol does not equal the unary symbol, we cannot perform the operation directly. We need to store the value in a register, perform the operation on the register, and store its value in the result symbol. Again, unless the operator is unary addition, in which case we do not perform the operations. However, the value is stored in the same way as the other operators.

```
      else {
        unaryTrack = LoadValueToRegister(unarySymbol);

        if (middleOperator != MiddleOperator.Plus) {
          AddAssemblyCode(objectOperator, unaryTrack);
        }
```

If the result symbol is a temporary variable, we add the unary track to the result symbol in the track map.

```
        if (resultSymbol.IsTemporary()) {
          Assert.ErrorXXX(resultSymbol.AddressSymbol == null);
          m_trackMap.Add(resultSymbol, unaryTrack);
        }
```

If the result symbol is a regular variable, we store the resulting value of the register on its address.

```
        else {
          AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                          Offset(resultSymbol), unaryTrack);
        }

        m_trackMap.Remove(unarySymbol);
      }
    }
```

12.6.3. Integral Binary

The binary integral addition and subtraction as well as the bitwise **and**, **or**, and **xor** operations are rather straightforward. They take two values stored in registers, on memory addresses, or as integer values. However, the left and right **shift** operation demands that the right operands is stored in a specific register.

The **IntegralBinary** method calls the second **IntegralBinary** method with the operator and the operands.

```
    public void IntegralBinary(MiddleCode middleCode) {
      Symbol resultSymbol = (Symbol)middleCode[0],
             leftSymbol = (Symbol)middleCode[1],
             rightSymbol = (Symbol)middleCode[2];
      IntegralBinary(middleCode.Operator, resultSymbol,
                     leftSymbol, rightSymbol);
    }
```

The **IntegralRelation** method also calls the **IntegralBinary** with comparation operator and the operands. Note that the result symbol is null in this case. We do not expect a result value of the comparations. Instead, the result is placed in flags that the following jump instruction catches. The assembly code operator corresponding to the middle code operator is looked up in the **m_middleToIntegralMap** map.

```
    public void IntegralRelation(MiddleCode middleCode) {
      Symbol leftSymbol = (Symbol) middleCode[1],
             rightSymbol = (Symbol) middleCode[2];
      IntegralBinary(MiddleOperator.Compare, null, leftSymbol, rightSymbol);

      AssemblyOperator objectOperator;
      if (leftSymbol.Type.IsUnsigned()) {
        objectOperator = m_unsignedToIntegralMap[middleCode.Operator];
      }
      else {
        objectOperator = m_middleToIntegralMap[middleCode.Operator];
      }

      int target = (int) middleCode[0];
      AddAssemblyCode(objectOperator, null, null, target);
    }
```

The **IntegralBinary** method generates assembly code for binary integral, addition, bitwise, and shift expressions. Basically, we have two cases: we to load the left operand into a register, or we perform the operation on the address of the left operand. In both cases, the right operand may be stored in a register or on an address, it may also be an integer value or and address.

```
    public void IntegralBinary(MiddleOperator middleOperator,
                               Symbol resultSymbol,  Symbol leftSymbol,
                                Symbol rightSymbol) {
      Track leftTrack = null, rightTrack = null;
      m_trackMap.TryGetValue(leftSymbol, out leftTrack);
      m_trackMap.TryGetValue(rightSymbol, out rightTrack);
```

Generally speaking, we try to keep the values in their current form as long as possible in order to generate as effective code as possible. However, there are some cases where we have to load the value into a register. More specifically, if the result symbol is not null (which it is in comparations) and it does not equal the left symbol (which it does in compound assignment), we load its value into a register.

```
      if ((leftTrack == null) &&
          (resultSymbol != null) && (resultSymbol != leftSymbol)) {
        leftTrack = LoadValueToRegister(leftSymbol);
      }
```

Another condition is that the left symbol may be an array, a function, or a string, or a static address. If their offset is non-zero, we must load its value into a register.

```
      if ((leftTrack == null) &&
          ((leftSymbol.Type.IsArrayFunctionOrString() &&
           (leftSymbol.Offset != 0)) ||
           ((leftSymbol.Value is StaticAddress) &&
           (((StaticAddress) leftSymbol.Value).Offset != 0)))) {
        leftTrack = LoadValueToRegister(leftSymbol);
      }
```

If the same conditions apply on the right symbol, with the exception of the assignment, addition, or subtraction operator. For those operators we can handle the value with or without an offset.

```
if ((rightTrack == null) &&
    (middleOperator != MiddleOperator.Assign) &&
    (middleOperator != MiddleOperator.Add) &&
    (middleOperator != MiddleOperator.Subtract) &&
    ((rightSymbol.Type.IsArrayFunctionOrString() &&
     (rightSymbol.Offset != 0)) ||
     ((rightSymbol.Value is StaticAddress) &&
      (((StaticAddress) rightSymbol.Value).Offset != 0)))) {
  rightTrack = LoadValueToRegister(rightSymbol);
}
```

Another condition we need to check is whether the right symbol holds a large value. Due to technical limitation of the architecture, we must load the value into a register if it exceeds 31 bits. In that case, the value will be loaded to the register with the **mov** instruction, which is the only allowed instruction for values exceeding 31 bits. Except for the assignment operator, in which case we will use the **mov** instruction anyway. We do not need to check the left symbol in the same way, since the middle code optimizer has reduced the expression or swapped the operands if the left operand holds an integer value.

```
if ((rightTrack == null) && (rightSymbol.Value is BigInteger) &&
    ((middleOperator != MiddleOperator.Assign) ||
     (leftTrack == null))) {
  BigInteger bigValue = (BigInteger) rightSymbol.Value;
  if (Start.Linux &&
      !((-2147483648 <= bigValue) && (bigValue <= 2147483647))) {
    rightTrack = LoadValueToRegister(rightSymbol);
  }
}
```

In case of the shift operator, we need to load the right symbol into a specific register (which is always **cl**) due to technical limitations of the architecture.

```
if (MiddleCode.IsShift(middleOperator)) {
  rightTrack =
    LoadValueToRegister(rightSymbol, AssemblyCode.ShiftRegister);
}

if ((leftTrack == null) && (leftSymbol.Value is BigInteger)) {
  leftTrack = LoadValueToRegister(leftSymbol);
}

int typeSize = leftSymbol.Type.Size();
AssemblyOperator objectOperator = m_middleToIntegralMap[middleOperator];
```

In this section, we call the **Base** and **Offset** methods. The base is a name in case of an extern or static symbol or a static address, and a register in case of an auto or register symbol. The offset is zero for an extern or static symbol. It may be zero or non-zero for a static address. It is always non-zero for an auto or register symbol.

If the value of left symbol is stored in a register (**leftTrack** is not null), we have a sequence of cases to consider. If the value of right symbol is also stored in a register, we simply perform the operation on the registers.

```
if (leftTrack != null) {
  if (rightTrack != null) {
    AddAssemblyCode(objectOperator, leftTrack, rightTrack);
  }
```

If the right symbol is an integer value, we use that value directly. Note that the integer value does not exceed 31 bits. If it does, the value would have already been loaded int a register.

```
    else if (rightSymbol.Value is BigInteger) {
      AddAssemblyCode(objectOperator, leftTrack, rightSymbol.Value);
    }
```

If the right symbol is an array, function, string, or static value, we use its address rather than its value. We start by applying the operator, the call to **Base** gives the name of the symbol.

```
    else if (rightSymbol.Type.IsArrayFunctionOrString() ||
             (rightSymbol.Value is StaticAddress)) {
      AddAssemblyCode(objectOperator, leftTrack, Base(rightSymbol));
```

Then we need to look into the offset. Remember that there can only be addition, subtraction, or assignment if the offset is non-zero. Otherwise, the address would have been loaded into a register. In case of assignment, we add the offset to the address. Otherwise, we just apply the operator (**add** or **sub**).

```
      int rightOffset = Offset(rightSymbol);
      if (rightOffset != 0) {
        if (middleOperator == MiddleOperator.Assign) {
          AddAssemblyCode(AssemblyOperator.add, leftTrack,
                          (BigInteger) rightOffset);
        }
        else {
          AddAssemblyCode(objectOperator, leftTrack,
                          (BigInteger) rightOffset);
        }
      }
    }
```

If none of the above cases apply, we use the base and offset of the left symbol.

```
    else {
      if (middleOperator == MiddleOperator.Assign) {
        leftTrack = new Track(resultSymbol);
      }

      AddAssemblyCode(objectOperator, leftTrack,
                      Base(rightSymbol), Offset(rightSymbol));
    }
```

The only remaining issue is what to do with the resulting value, if there is one. If the result is a temporary variable, we add it to the track map to be used by succeeding instructions.

```
    if (resultSymbol != null) {
      if (resultSymbol.IsTemporary()) {
        Assert.ErrorXXX(resultSymbol.AddressSymbol == null);
        m_trackMap.Add(resultSymbol, leftTrack);
      }
```

If the result is not a temporary variable, we store the value on its address.

```
      else {
        AddAssemblyCode(AssemblyOperator.mov, Base(resultSymbol),
                        Offset(resultSymbol), leftTrack);
      }
```

```
    }
  }
```

If the left symbol is not stored in a register, we have a new sequence of cases to consider. If the right symbol is stored in a register, we simply use it.

```
  else {
    if (rightTrack != null) {
      AddAssemblyCode(objectOperator, Base(leftSymbol),
                      Offset(leftSymbol), rightTrack);
    }
```

If the right symbol is an integer value, we simply use it. Again, note that the value does not exceeds 31 bits. If it does, it would have been loaded into a register.

```
    else if (rightSymbol.Value is BigInteger) {
      AddAssemblyCode(objectOperator, Base(leftSymbol),
                      Offset(leftSymbol), rightSymbol.Value, typeSize);
    }
```

If the right symbol is an array, function, string, or static value, we use its address and apply the operators in the same way as above.

```
    else if (rightSymbol.Type.IsArrayFunctionOrString() ||
             (rightSymbol.Value is StaticAddress)) {
      AddAssemblyCode(objectOperator, Base(leftSymbol),
                      Offset(leftSymbol), Base(rightSymbol),
                      TypeSize.PointerSize);

      int rightOffset = Offset(rightSymbol);
      if (rightOffset != 0) {
        if (middleOperator == MiddleOperator.Assign) {
          AddAssemblyCode(AssemblyOperator.add, Base(leftSymbol),
                          Offset(leftSymbol), (BigInteger) rightOffset,
                          typeSize);
        }
        else {
          AddAssemblyCode(objectOperator, Base(leftSymbol),
                          Offset(leftSymbol), (BigInteger) rightOffset,
                          typeSize);
        }
      }
    }
```

If none of the above cases apply, we load the value of the right symbol into a register and use the base and offset of the left symbol.

```
    else {
      rightTrack = LoadValueToRegister(rightSymbol);
      AddAssemblyCode(objectOperator, Base(leftSymbol),
                      Offset(leftSymbol), rightTrack);
    }
  }
```

Finally, we need to remove the left and right symbol from the track map, since they shall not be used any more.

```
  m_trackMap.Remove(leftSymbol);
  m_trackMap.Remove(rightSymbol);
```

```
    }
```

12.6.1. Base and Offset

The **Base** method returns the base of a symbol, which may be a register, a track, or a name. If the symbol's value if a static address we return its name, which will later be replaced by a proper address by the linker.

```
    private object Base(Symbol symbol) {
      Assert.ErrorXXX(!(symbol.Value is BigInteger));

      if (symbol.Value is StaticAddress) {
        StaticAddress staticAddress = (StaticAddress) symbol.Value;
        return staticAddress.UniqueName;
      }
```

If the symbol's address symbol is not null, the symbol represents a dereferred symbol in a dereference, index, or arrow expression, and we return the register holding the address. We look up the address track (the track of the address symbol) by calling **LoadValueToRegister** and mark the track as pointer, which means that the register allocator will choose the register from a smaller set. Since the track is no longer needed, we delete it from the track map before we return it.

```
      else if (symbol.AddressSymbol != null) {
        Track addressTrack = LoadValueToRegister(symbol.AddressSymbol);
        Assert.ErrorXXX((addressTrack.Register == null) ||
                        RegisterAllocator.VariadicFunctionPointerRegisterSet.
                        Contains(addressTrack.Register.Value));
        addressTrack.Pointer = true;
        m_trackMap.Remove(symbol.AddressSymbol);
        return addressTrack;
      }
```

If the symbol is extern or static, we return its unique name, which will later be replaced by a proper address by the linker.

```
      else if (symbol.IsExternOrStatic()) {
        return symbol.UniqueName;
      }
```

Finally, if the symbol is auto or register, we call **BaseRegister** to obtain the regular or variadic frame pointer.

```
      else { //resultSymbol.IsAutoOrRegister()
        return BaseRegister(symbol);
      }
    }
```

The **Offset** method returns the offset of the symbol. If its value is a static address, we return the offset of the static address.

```
    private int Offset(Symbol symbol) {
      Assert.ErrorXXX(!(symbol.Value is BigInteger));

      if (symbol.Value is StaticAddress) {
        StaticAddress staticAddress = (StaticAddress) symbol.Value;
        return staticAddress.Offset;
      }
```

If the symbol's address symbol is not null, we return the offset of the address symbol.

```
      else if (symbol.AddressSymbol != null) {
        return symbol.AddressOffset;
      }
```

Finally, in all other case we return the offset of the symbol. The offset is significant in case of auto or register parameters and variables. In case of an extern or static symbol, the offset is always zero and we will not use the value.

```
      else {
        return symbol.Offset;
      }
    }
```

12.6.2. Case

The **Case** is method checks whether the switch symbol equals a value. Normally, we load the value of a symbol into a register, which we add to the track map. After the operation has been performed, we remove the register from the track map. However, when dealing with a sequence of case statements, we load the value of the switch symbol into a register, and then we keep the value in the register during all the case test instructions.

```
    public void Case(MiddleCode middleCode) {
      Symbol switchSymbol = (Symbol) middleCode[1];
      Track switchTrack = LoadValueToRegister(switchSymbol);
      Symbol caseSymbol = (Symbol) middleCode[2];
      BigInteger caseValue = (BigInteger) caseSymbol.Value; // cmp ax, 123
      AddAssemblyCode(AssemblyOperator.cmp, switchTrack, caseValue);
      int target = (int) middleCode[0];
      AddAssemblyCode(AssemblyOperator.je, null, null, target);
```

Note that we do add the symbol to the track map, in order to keep the value in the register thought out all the succeeding case instruction of the switch statement.

```
      m_trackMap.Add(switchSymbol, switchTrack);
    }
```

The **CaseEnd** method is called after the last case instruction of the switch statement. We only remove the switch symbol from the track map, since we shall not keep the value in the register after the last case instruction.

```
    public void CaseEnd(MiddleCode middleCode) {
      Symbol symbol = (Symbol) middleCode[0];
      m_trackMap.Remove(symbol);
    }
```

12.6.3. Address

The **Address** method adds assembly code instructions for the address operator. It is actually quite simple, since we only gave to call **LoadAddressToRegister** to load the address of the symbol into a register.

```
    public void Address(MiddleCode middleCode) {
      Symbol resultSymbol = (Symbol) middleCode[0],
             addressSymbol = (Symbol) middleCode[1];
      Track track = LoadAddressToRegister(addressSymbol);
      m_trackMap.Add(resultSymbol, track);
      m_trackMap.Remove(addressSymbol);
    }
```

The **IntegralDereference** method adds assembly code instructions for derefereeing a pointer to an integral value. We only have to call **LoadValueToRegister** for the address symbol.

```
public void IntegralDereference(MiddleCode middleCode) {
  Symbol resultSymbol = (Symbol) middleCode[0];
  Assert.ErrorXXX(resultSymbol.AddressSymbol != null);
  Track addressTrack = LoadValueToRegister(resultSymbol.AddressSymbol);
  m_trackMap.Add(resultSymbol.AddressSymbol, addressTrack);
}
```

The **FloatingDereference** method adds assembly code instructions for derefereing a pointer to a floating value.

```
public void FloatingDereference(MiddleCode middleCode) {
  Symbol resultSymbol = (Symbol) middleCode[0];
  Assert.ErrorXXX(resultSymbol.AddressSymbol != null);
  Track addressTrack = LoadValueToRegister(resultSymbol.AddressSymbol);
  m_trackMap.Add(resultSymbol.AddressSymbol, addressTrack);
}
```

12.6.1. Floating Binary

Similar to the integral operations, we need the **m_middleToFloatingMap** map to translate the middle code operators to their equivalent assembly code operators.

```
public static IDictionary<MiddleOperator, AssemblyOperator>
  m_middleToFloatingMap =
    new Dictionary<MiddleOperator, AssemblyOperator>() {
      {MiddleOperator.Minus, AssemblyOperator.fchs},
      {MiddleOperator.Add, AssemblyOperator.fadd},
      {MiddleOperator.Subtract, AssemblyOperator.fsub},
      {MiddleOperator.Multiply, AssemblyOperator.fmul},
      {MiddleOperator.Divide, AssemblyOperator.fdiv},
      {MiddleOperator.Equal, AssemblyOperator.je},
      {MiddleOperator.NotEqual, AssemblyOperator.jne},
      {MiddleOperator.LessThan, AssemblyOperator.ja},
      {MiddleOperator.LessThanEqual, AssemblyOperator.jae},
      {MiddleOperator.GreaterThan, AssemblyOperator.jb},
      {MiddleOperator.GreaterThanEqual, AssemblyOperator.jbe}
    };
```

The methods for the floating-point operations are in fact much more simple than the corresponding integral methods. The **FloatingUnary** method just adds the instruction to the assembly code. The operand of the operator has already been pushed to the floating-point stack, the operator pops that value, and pushes the result to the stack.

```
public void FloatingUnary(MiddleCode middleCode) {
  AddAssemblyCode(m_middleToFloatingMap[middleCode.Operator]);
}
```

The **FloatingBinary** works in the same way, the operator pops the two operands from the floating-point stack and pushes the result on the stack.

```
public void FloatingBinary(MiddleCode middleCode) {
  Assert.ErrorXXX((--m_floatStackSize) >= 0);
  AddAssemblyCode(m_middleToFloatingMap[middleCode.Operator]);
}
```

In the **FloatingParameter** we do not need to actual parameter symbol, since its value has already been pushed on the floating-point stack. We only need to type and offset of the parameter when we pop it on the floating-point stack by calling **TopPopSymbol**.

```
public void FloatingParameter(MiddleCode middleCode) {
  Type paramType = (Type) middleCode[1];
  Symbol paramSymbol = new Symbol(paramType);
  int paramOffset = (int) middleCode[0];
  paramSymbol.Offset = m_totalExtraSize + paramOffset;
  TopPopSymbol(paramSymbol, TopOrPop.Pop);
}
```

12.6.1. Floating Relation

The **FloatingRelation** method generate code for relation operators for floating types. The **fcompp** (float compare pop) assembly code instruction performs a comparation on the two values on top of the floating value stack, stores the result in the internal float status word, and pops the values from the stack. The **fstsw** (float store status word) assembly code instruction stores the floating status word in the **ax** register. The **sahf** (store ah into flags) stores the value of register **ah** (which hold the higher byte of register **ax**) into the integral flags. Finally, we add a jump instruction that matches the original middle code instruction and is looked up in the middle to floating map.

```
public void FloatingRelation(MiddleCode middleCode) {
  Assert.ErrorXXX((m_floatStackSize -= 2) >= 0);
  int target = (int) middleCode[0];
  AddAssemblyCode(AssemblyOperator.fcompp);
  AddAssemblyCode(AssemblyOperator.fstsw, Register.ax);
  AddAssemblyCode(AssemblyOperator.sahf);
  AssemblyOperator objectOperator =
    m_middleToFloatingMap[middleCode.Operator];
  AddAssemblyCode(objectOperator, null, null, target);
}
```

12.6.2. Floating Push and Pop

When pushing floating values to the floating-point stack, we need the **m_floatPushMap** map that maps the whether the type is logical and the type size to assembly code operators.

```
public static IDictionary<Pair<bool, int>, AssemblyOperator>
  m_floatPushMap = new Dictionary<Pair<bool, int>, AssemblyOperator>() {
    {new Pair<bool,int>(false, 2), AssemblyOperator.fild_word},
    {new Pair<bool,int>(false, 4), AssemblyOperator.fild_dword},
    {new Pair<bool,int>(false, 8), AssemblyOperator.fild_qword},
    {new Pair<bool,int>(true, 4), AssemblyOperator.fld_dword},
    {new Pair<bool,int>(true, 8), AssemblyOperator.fld_qword}
  };
```

The **PushSymbol** method generates code that pushes the value of a symbol to the floating-point stack.

```
public void PushSymbol(Symbol symbol) {
  Assert.ErrorXXX((++m_floatStackSize) <= FloatingStackMaxSize);
  Track track;
```

There are two special assembly code instructions for pushing the value zero or one to the floating-point stack: **fldz** (float load zero) **fld1** (float load one). If the value is (integral or floating) zero, we use **fldz** to load the value zero, and if the value is (integral or floating) one we use **fld1** to load the value zero.

```
      if (((symbol.Value is BigInteger) &&
          (((BigInteger) symbol.Value).IsZero)) ||
          ((symbol.Value is decimal) && (((decimal) symbol.Value) == 0))) {
        AddAssemblyCode(AssemblyOperator.fldz);
      }
      else if (((symbol.Value is BigInteger) &&
                (((BigInteger) symbol.Value).IsOne)) ||
               ((symbol.Value is decimal) &&
                (((decimal) symbol.Value) == 1))) {
        AddAssemblyCode(AssemblyOperator.fld1);
      }
```

For any other integral or floating value, we cannot load it directly. Instead, we need store the value on a memory address and load the value from the address. We add the symbol to the global static set for the linker to be able to find the value, and we add the push operation with the name of symbol, which will be changed to a proper address by the linker.

```
      else {
        Pair<bool,int> pair =
          new Pair<bool,int>(symbol.Type.IsFloating(), symbol.Type.Size());
        AssemblyOperator objectOperator = m_floatPushMap[pair];

        if ((symbol.Value is BigInteger) || (symbol.Value is decimal)) {
          SymbolTable.StaticSet.Add(ConstantExpression.Value(symbol));
          AddAssemblyCode(objectOperator, symbol.UniqueName, 0);
        }
```

If the symbol does not hold an integral or a floating value, we check if the value is stored in a register. In that case we cannot store the register directly, but like the value case above we need to store the value at the memory address. We use the special integral storage to load the value.

```
        else if (m_trackMap.TryGetValue(symbol, out track)) {
          m_trackMap.Remove(symbol);
          string containerName = AddStaticContainer(symbol.Type);
          AddAssemblyCode(AssemblyOperator.mov, containerName, 0, track);
          AddAssemblyCode(objectOperator, containerName, 0);
        }
```

If the symbol is an array, function, or string, we also use the integral storage name to store the value. However, in this case will shall load the address of the symbol, rather than its value. We start by loading the base address of the symbol to the integral storage. Then we add its offset, unless it is zero.

```
        else if (symbol.Type.IsArrayFunctionOrString()) {
          string containerName = AddStaticContainer(symbol.Type);
          AddAssemblyCode(AssemblyOperator.mov, containerName, 0,
                          Base(symbol), TypeSize.PointerSize);

          int offset = Offset(symbol);
          if (offset != 0) {
            AddAssemblyCode(AssemblyOperator.add, containerName, 0,
                            (BigInteger) offset, TypeSize.PointerSize);
          }

          AddAssemblyCode(objectOperator, containerName, 0);
        }
```

If none of the above apply, we perform the operation on the address of the symbol.

```
      else {
        AddAssemblyCode(objectOperator, Base(symbol), Offset(symbol));
      }
    }
  }

  private static string AddStaticContainer(Type type) {
    string containerName = "container" + type.Size() +
                           "bytes" + Symbol.NumberId;
    SymbolTable.StaticSet.Add(ConstantExpression.Value(containerName,
                                                     type, null));
    return containerName;
  }
```

When popping or topping a value from the floating-point stack, we also need the **m_floatPopMap** and **m_floatTopMap** maps that map whether the type is logical and the type size to **pop** and **top** assembly code instructions.

```
  public static IDictionary<Pair<bool,int>,AssemblyOperator>
    m_floatPopMap = new Dictionary<Pair<bool,int>, AssemblyOperator>() {
      {new Pair<bool,int>(false, 2), AssemblyOperator.fistp_word},
      {new Pair<bool,int>(false, 4), AssemblyOperator.fistp_dword},
      {new Pair<bool,int>(false, 8), AssemblyOperator.fistp_qword},
      {new Pair<bool,int>(true, 4), AssemblyOperator.fstp_dword},
      {new Pair<bool,int>(true, 8), AssemblyOperator.fstp_qword}
    };

  public static IDictionary<Pair<bool,int>,AssemblyOperator>
    m_floatTopMap = new Dictionary<Pair<bool,int>, AssemblyOperator>() {
      {new Pair<bool,int>(false, 2), AssemblyOperator.fist_word},
      {new Pair<bool,int>(false, 4), AssemblyOperator.fist_dword},
      {new Pair<bool,int>(false, 8), AssemblyOperator.fist_qword},
      {new Pair<bool,int>(true, 4), AssemblyOperator.fst_dword},
      {new Pair<bool,int>(true, 8), AssemblyOperator.fst_qword}
    };

  public enum TopOrPop {Top, Pop};
```

The **PopEmpty** method pops the floating-point stack without storing the value anywhere. However, there is no matching assembly code instruction. The instructions always store the value somewhere. Therefore, we use the integral storage, simply because we have to use some storage.

```
  public void PopEmpty() {
    string containerName = AddStaticContainer(Type.LongDoubleType);
    AddAssemblyCode(AssemblyOperator.fistp_word, containerName, 0);
  }
```

The **TopPopSymbol** method tops or pops the top-most value of the floating-point stack.

```
  public void TopPopSymbol(Symbol symbol, TopOrPop topOrPop) {
    Assert.ErrorXXX(symbol != null);
    Pair<bool,int> pair =
      new Pair<bool,int>(symbol.Type.IsFloating(), symbol.Type.Size());
    AssemblyOperator objectOperator;
```

Depending on the value of the **topOrPop** parameter, we extract the assembly code instruction from the **m_floatPopMap** and **m_floatTopMap** maps.

```
    if (topOrPop == TopOrPop.Pop) {
      objectOperator = m_floatPopMap[pair];
      Assert.ErrorXXX((--m_floatStackSize) >= 0);
    }
    else {
      objectOperator = m_floatTopMap[pair];
    }
```

If the symbol is a temporary variable, we store its value in a register. However, we cannot store the value directly in a register. Therefore, we pop or top the value to the integral storage and loads the value from there into the register.

```
      if (symbol.Type.IsIntegralPointerOrArray() && symbol.IsTemporary()) {
        string containerName = AddStaticContainer(symbol.Type);
        AddAssemblyCode(objectOperator, containerName, 0);
        Track track = new Track(symbol);
        AddAssemblyCode(AssemblyOperator.mov, track, containerName, 0);
        m_trackMap.Add(symbol, track);
      }
```

If none of the above cases apply, we pop or top the value to the address of the symbol.

```
      else {
        AddAssemblyCode(objectOperator, Base(symbol), Offset(symbol));
      }
    }
```

12.6.3. Type Conversion

The **IntegralToIntegral** method adds assembly code instructions that converts an integral value from one size to another size.

```
    public void IntegralToIntegral(MiddleCode middleCode, int index) {
      Symbol targetSymbol = (Symbol) middleCode[0],
             sourceSymbol = (Symbol) middleCode[1];
```

The **sizeArray** method returns pointer size for arrays, functions, and strings.

```
      Type targetType = targetSymbol.Type, sourceType = sourceSymbol.Type;
      int targetSize = targetType.SizeArray(),
          sourceSize = sourceType.SizeArray();
```

We need target add a set-track-size instruction for

```
      Track sourceTrack = LoadValueToRegister(sourceSymbol);
      AddAssemblyCode(AssemblyOperatargetr.set_track_size,
                      sourceTrack, targetSize);
```

If the source size is smaller than the target size, we add code to mask the upper bits; that is, we set to zero the bits of the target value that do not fit in the source value.

```
      if (sourceSize != targetSize) {
        if (sourceSize < targetSize) {
```

If the target size is eight bytes, we have a special case. We need to load the mask to a register, and use the **and** instruction to do the actual masking.

```
          if (targetSize == 8) {
            Track targetTrack = new Track(targetSymbol);
            AddAssemblyCode(AssemblyOperatargetr.mov, targetTrack,
                            TypeSize.GetMask(sourceType.Sort));
            AddAssemblyCode(AssemblyOperatargetr.and,
                                 sourceTrack, targetTrack);
          }
```

If the target size is not eight bytes, we can use the **and** instruction directly.

```
          else {
            AddAssemblyCode(AssemblyOperatargetr.and, sourceTrack,
                            TypeSize.GetMask(sourceType.Sort));
          }
        }
```

If both the source type and target type is signed, we need to preserve the signed status throught the conversion process. If the value is negative, the top-most bit is set to one. Since the source size does not equals the target size, the top-most bits differ between the values. Therefore, if the value is negative, we need to change it to a positive value with the source size and change it back to a negative value with the target size. In this way, the signed status will be preserved. However, if the value is non-negative (zero or positive), we do nothing. We just add to the track map that the values holds a new size.

```
        if (sourceType.IsSigned() && targetType.IsSigned()) {
          AddAssemblyCode(AssemblyOperatargetr.set_track_size,
                          sourceTrack, sourceSize);
          AddAssemblyCode(AssemblyOperatargetr.cmp, sourceTrack,
                          BigInteger.Zero);
          AddAssemblyCode(AssemblyOperatargetr.jge, null, null, index + 1);
          AddAssemblyCode(AssemblyOperatargetr.neg, sourceTrack);
          AddAssemblyCode(AssemblyOperatargetr.set_track_size,
                          sourceTrack, targetSize);
          AddAssemblyCode(AssemblyOperatargetr.neg, sourceTrack);
        }
      }
```

If the target type and source type have the same size, we only add the target symbol with the source track in the track map.

```
      m_trackMap.Add(targetSymbol, sourceTrack);
    }
```

The **IntegralToFloating** and **FloatingToIntegral** are quite simple. In the integral-to-floating-case we push the integral value at the floating-point stack, and in the floating-to-integral case, we pop the value from the stack. Note that there is no floating-to-floating case. In that case, we just keep the value on the stack, the type conversion occurs when the value is pushed, popped, or topped.

```
    public void IntegralToFloating(MiddleCode middleCode) {
      Symbol fromSymbol = (Symbol) middleCode[1];
      PushSymbol(fromSymbol);
    }

    public void FloatingToIntegral(MiddleCode middleCode) {
      Symbol toSymbol = (Symbol) middleCode[0];
      TopPopSymbol(toSymbol, TopOrPop.Pop);
    }
```

12.6.4. Struct and Union

This section describes the handling of struct and unions, assignment, parameter, and function return value. Note that these methods make no difference between struct and unions, in all cases the task is to move the data of a memory block between two addresses.

The **StructUnionAssign** method copies the data from the address of the target symbol to address of the source symbol by calling **MemoryCopy**.

```
public void StructUnionAssignInit(MiddleCode middleCode) {
  Symbol targetSymbol = (Symbol)middleCode[0],
         sourceSymbol = (Symbol)middleCode[1];
  MemoryCopyInit(targetSymbol, sourceSymbol);
}

public void StructUnionAssign(MiddleCode middleCode, int middleIndex) {
  MemoryCopyLoop(middleIndex);
}
```

The **StructUnionParameter** copies the data from the address of the parameter symbol to the address on the activation record by calling **MemoryCopy**.

```
public void StructUnionParameterInit(MiddleCode middleCode) {
  int paramOffset = (int) middleCode[0];
  Symbol sourceSymbol = (Symbol) middleCode[2];
  Symbol targetSymbol = new Symbol(Type.IntegerPointerType);
  targetSymbol.Offset = m_totalExtraSize + paramOffset;
  MemoryCopyInit(targetSymbol, sourceSymbol);
}

public void StructUnionParameter(MiddleCode middleCode, int middleIndex) {
  MemoryCopyLoop(middleIndex);
}
```

The **StructUnionGetReturnValue** loads the address symbol of the symbol into the return pointer register, which is then added to the track map. In this way, is the value of the address symbol of the struct or union stored in the track and will be used in later operations.

```
public void StructUnionGetReturnValue(MiddleCode middleCode) {
  Symbol targetSymbol = (Symbol) middleCode[0];
  CheckRegister(targetSymbol, AssemblyCode.ReturnAddressRegister);
  Track targetAddressTrack =
    new Track(targetSymbol.AddressSymbol,
              AssemblyCode.ReturnAddressRegister);
  m_trackMap.Add(targetSymbol.AddressSymbol, targetAddressTrack);
}
```

The **StructUnionSetReturnValue** method

```
public void StructUnionSetReturnValue(MiddleCode middleCode) {
  Symbol returnSymbol = (Symbol) middleCode[1];
  LoadAddressToRegister(returnSymbol, AssemblyCode.ReturnAddressRegister);
}
```

The **m_labelCount** field is used by **MemoryCopy** to generate a new label for each copy process.

```
private Track m_targetAddressTrack = null, m_sourceAddressTrack = null;
private static Track m_countTrack = null;
```

The **MemoryCopy** method adds code to copy the data of a memory block between two memory addresses.

```
private void MemoryCopyInit(Symbol targetSymbol, Symbol sourceSymbol) {
  m_targetAddressTrack = LoadAddressToRegister(targetSymbol);
  m_sourceAddressTrack = LoadAddressToRegister(sourceSymbol);
```

We create tracks for the register to count of the data block and the value being copied. If the size of the data block is less than 256, we use the character types (one byte) for the count register, otherwise we use the integer type (four bytes for Linux, two bytes for Windows).

```
      int size = sourceSymbol.Type.Size();
      Type countType = (size < 256) ? Type.UnsignedCharType
                                    : Type.UnsignedIntegerType;
```

We start by loading the size of the data block into the count register, the loop continues until it reaches zero.

```
      m_countTrack = new Track(countType);
      AddAssemblyCode(AssemblyOperator.mov, m_countTrack, (BigInteger) size);
    }
```

Then the loop begins, we use a label that is unique in the assembly file with the **m_labelCount** field.

```
    private void MemoryCopyLoop(int middleIndex) {
      Track valueTrack = new Track(Type.UnsignedCharType);
```

We move a value from the source address to the target address.

```
      AddAssemblyCode(AssemblyOperator.mov, valueTrack,
                      m_sourceAddressTrack, 0);
      AddAssemblyCode(AssemblyOperator.mov, m_targetAddressTrack,
                      0, valueTrack);
```

We increment the source and target address.

```
      AddAssemblyCode(AssemblyOperator.inc, m_sourceAddressTrack);
      AddAssemblyCode(AssemblyOperator.inc, m_targetAddressTrack);
```

We decrement the count register.

```
      AddAssemblyCode(AssemblyOperator.dec, m_countTrack);
```

If the count register does not equal zero, we jump to the label. If it does equal zero, the copy process is done.

```
      AddAssemblyCode(AssemblyOperator.cmp, m_countTrack, BigInteger.Zero);
      AddAssemblyCode(AssemblyOperator.jne, null, null, middleIndex);
    }
```

12.6.5. Initialization Code

The initialization of data blocks differs between the Linux and the Windows environment. In this section we look into the Linux environment, the Windows environment is described in Chapter 13.

AssemblyCodeGenerator.cs

```
    public static void InitializationCodeList() {
      List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();
      AddAssemblyCode(assemblyCodeList, AssemblyOperator.label, "_start");
      AddAssemblyCode(assemblyCodeList, AssemblyOperator.comment,
                      "Initializerialize Stack Pointer");
```

We begin by initialize the frame pointer. It set to the address of the stack, which is the address directly after the code and static data. We do not need to set the variadic pointer, since the main function is not variadic.

```
      AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                      AssemblyCode.RegularFrameRegister,
Linker.CallStackStart);
```

We initialize the heap to 64 kilobytes (65536 bytes) and set the heap pointer at address 65534 to zero, since the heap is empty at the beginning of the execution.

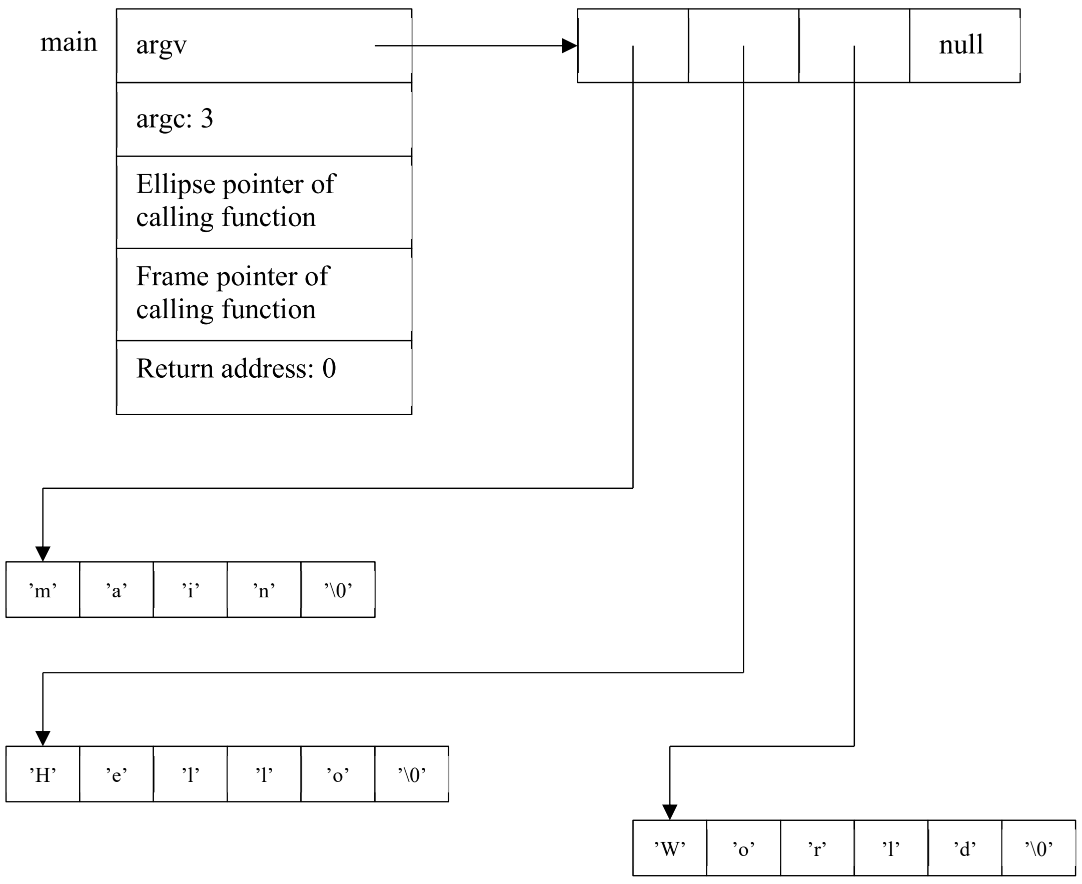
```
      if (Start.Linux) {
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.comment,
                        "Initializerialize Heap Pointer");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_dword,
                        Linker.CallStackStart, 65534,
                        Linker.CallStackStart);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.add_dword,
                        Linker.CallStackStart, 65534,
                        (BigInteger) 65534);
```

We also need to initialize the floating-point unit control word so that floating point operations truncates.

```
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.comment,
                        "Initializerialize FPU Control Word, truncate mode " +
                        "=> set bit 10 and 11.");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.fstcw,
                        AssemblyCode.RegularFrameRegister, 0);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.or_word,
                        AssemblyCode.RegularFrameRegister, 0,
                        (BigInteger) 0x0C00);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.fldcw,
                        AssemblyCode.RegularFrameRegister, 0);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Linker.CallStackStart, 0, BigInteger.Zero,
                        TypeSize.PointerSize);

        List<string> textList = new List<string>();
        textList.Add("section .text");
        ISet<string> externSet = new HashSet<string>();
        AssemblyCodeGenerator.LinuxTextList(assemblyCodeList, textList,
                                            externSet);
        SymbolTable.InitSymbol =
          new StaticSymbolLinux(AssemblyCodeGenerator.InitializerName,
                                textList, externSet);
      }

      if (Start.Windows) {
        // ...
      }
    }
```

12.6.6. Command Line Arguments

Assuming that the **main** function is declared as **void main(int argc, char* argv[]);** and that we start the execution with the command line **main Hello World** the following structure becomes generated:



The initialization code looks as follows. We begin by popping the system stack to **rbx** to obtain the number of arguments, which is three in the example above. We also copy the value to **rax**. We will use **rbx** to count down the number or arguments, and **rax** to store the number on the activation record. Moreover, we use **rbp** as the frame pointer and copy its value in **rdx**. We will put the pointer to the argument directly above the code and data part of the code, and directly below the activation record for the first **main** call.

```
        pop rbx
        mov rax, rbx
        mov rdx, rbp
```

We count down the number of arguments in **rbx**. When it has reached zero, we quit the loop. Each pointer to e new argument is popped from the system stack, added to memory with **rbp**. The **rbp** is increased by eight bytes, and **rbx** is decreased by one.

```
 $args$loop:
        cmp rbx, 0
        je $args$exit
        pop rsi
        mov [rbp], rsi
        add rbp, 8
        dec rbx
        jmp $args$loop
```

When all argument pointers have been added, we add the final null pointer and increase **rbp**. By then the argument list has been properly added to the memory.

```
$args$exit:
     mov qword [rbp], 0
     add rbp, 8
```

The **rbp** register has by now been given its correct value, it now points at the beginning of the activation record. We set the return address (at offset zero) to zero, indicating the returning from the function shall result in an exit from the execution. We also set the **argc** parameter (at offset 24) to the value **eax**, holding the number of arguments, and the **argv** parameter (at offset 28) the value of **rdx**, holding the value of the first argument pointer.

```
     mov qword [rbp], 0
     mov [rbp + 24], eax
     mov [rbp + 28], rdx
```

The code for obtaining the command line arguments differs between the Linux and the Windows environment.

```
    public static void ArgumentCodeList() {
      List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();
```

When to execution starts, the system stack is loaded with values corresponding to the command line arguments. We pop the stack to the **rbx** register to obtain the number of arguments.

```
      if (Start.Linux) {
        /*  pop rbx
            mov rax, rbx
            mov rdx, rbp */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.comment,
                        "Initialize Command Line Arguments");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.pop, Register.rbx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.rax, Register.rbx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.rdx, Register.rbp);

        /* $args$loop:
             cmp rbx, 0
             je $args$exit
             pop rsi
             mov [rbp], rsi
             add rbp, 8
             dec rbx
             jmp $args$loop */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.label,
                        "$args$loop");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.cmp,
                        Register.rbx, BigInteger.Zero);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.je,
                        null, null, "$args$exit");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.pop, Register.rsi);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.rbp, 0, Register.rsi);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.add, Register.rbp,
                        (BigInteger) TypeSize.PointerSize);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.dec, Register.rbx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.jmp,
                    null, null, "$args$loop");

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.label,
                        "$args$exit");
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_qword,
                        Register.rbp, 0, BigInteger.Zero);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.add, Register.rbp,
                        (BigInteger) TypeSize.PointerSize);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.rbp,0, BigInteger.Zero,TypeSize.PointerSize);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov, Register.rbp,
                        SymbolTable.FunctionHeaderSize, Register.eax);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov, Register.rbp,
                        SymbolTable.FunctionHeaderSize +
                        TypeSize.SignedIntegerSize, Register.rdx);

        /* $args$exit:
             mov qword [rbp], 0
             add rbp, 8 */

        List<string> textList = new List<string>();
        ISet<string> externSet = new HashSet<string>();
        AssemblyCodeGenerator.LinuxTextList(assemblyCodeList, textList,
                                            externSet);
        SymbolTable.ArgsSymbol =
          new StaticSymbolLinux(AssemblyCodeGenerator.ArgsName,
                                textList, externSet);
      }

      if (Start.Windows) {
        // ...
      }
    }
```

12.6.1. Text List

When the assembly code has been generated, we need to be written as a text file, which is the task of **LinuxTextList**. It iterates through the assembly code list and writes the assembly code instruction at one line each by calling **ToString** in **AssemblyCode**. It also adds names add the **externSet** parameter for each name in the instructions.

```
public static void LinuxTextList(IList<AssemblyCode> assemblyCodeList,
                                 IList<string> textList,
                                 ISet<string> externSet) {
  foreach (AssemblyCode assemblyCode in assemblyCodeList) {
    AssemblyOperator assemblyOperator = assemblyCode.Operator;
    object operand0 = assemblyCode[0],
           operand1 = assemblyCode[1],
           operand2 = assemblyCode[2];

    if (assemblyOperator == AssemblyOperator.define_value) {
      Sort sort = (Sort) operand0;

      if ((sort != Sort.String) && (operand1 is string)) {
        string name1 = (string) operand1;

        if (!name1.Contains(Symbol.SeparatorId)) {
          externSet.Add(name1);
        }
      }
    }
    else if ((assemblyOperator != AssemblyOperator.label) &&
             (assemblyOperator != AssemblyOperator.comment)) {
      if (operand0 is string) {
        string name0 = (string) operand0;

        if (!name0.Contains(Symbol.SeparatorId)) {
          externSet.Add(name0);
        }
      }

      if (operand1 is string) {
        string name1 = (string) operand1;

        if (!name1.Contains(Symbol.SeparatorId)) {
          externSet.Add(name1);
        }
      }

      if (operand2 is string) {
        string name2 = (string) operand2;

        if (!name2.Contains(Symbol.SeparatorId)) {
          externSet.Add(name2);
        }
      }
    }

    string text = assemblyCode.ToString();
    if (text != null) {
      textList.Add(text);
    }
  }
}
```

13. Executable Code Generation

So far in this book, the compiler has generated assembly code files and a make file. In this chapter, the compiler instead generates an executable file. I have chosen the 16-bit .com format, which is a simple file format. Unfortunately, the format is no longer supported by Windows. Therefore, we need to use a simulator the supports the format. I use **DosBox**, which is a simple simulator capable of executing files in the **.com** format.

13.1. The Windows Environment

In the book so far, we have regarded the **Start.Linux** condition on several occasions, indicating code specific for the Linux target machine. In this chapter we go the outer way around and regard the **Start.Windows** condition, which indicates code specific for the Windows target machine.

The difference between the Windows environment, compared to the Linux environment, is:

- The types hold different sizes.
- A static value is stored as a byte list rather than a text list, with an access map that keeps track of the accesses of other static values.
- A function is stored as a byte list, with an access map. A function also holds a call map to keep track of functions calls and a return set to keep track of the return addresses of the calls.
- To exit the execution, we perform an interrupt call rather than a system call.
- The initialization code is different.
- The command line arguments code is also different. The command line is stored in the first 256 bytes of the segment.
- The total code, including code, static values, call stack and heap, is stored in 64 kilobytes.
- In the standard library, we use interrupts instead of system calls to access the surrounding operating system.

The accesses and calls are stored with their unique names, and replaced by proper address by the linker in the next step. The return addresses are stored as the address relative the beginning of the function and is replaced by the corresponding global address by the linker. The reason we keep the accesses and calls in separate maps is that accesses are replaced by absolute global addresses while calls are replaced by addresses relative to the call.

13.1.1. Main

Main.cs
```
using System;
using System.IO;
using System.Text;
using System.Globalization;
using System.Collections.Generic;

namespace CCompiler {
```

```
public class Start {
  // ...

  public static void Main(string[] args){
    // ...

    if (Start.Windows) {
      ObjectCodeTable.Initializer();
    }

    // ...

    try {
      // ...

      if (Start.Linux) {
        // ...
      }
```

We iterate through the command line arguments in the same way as in Chapter 12. However, we add the **doLink** variable that is set to true if at least one source file becomes recompiled and we thereby shall link the files into an executable file. We also state the file suffix as ".obj" rather than ".asm".

```
      if (Start.Windows) {
        bool doLink = false;

        foreach (string arg in argList) {
          FileInfo file = new FileInfo(SourcePath + arg);

          if (rebuild || !IsGeneratedFileFresh(file, ".obj")) {
            if (print) {
              Console.Out.WriteLine("Compiling \"" +
                                    file.FullName + ".c\".");
            }

            CompileSourceFile(file);
            doLink = true;
          }
        }

        if (doLink) {
          FileInfo targetFile =
            new FileInfo(TargetPath + argList[0] + ".com");
          Linker linker = new Linker();

          CCompiler_Main.Scanner.Path = null;
          foreach (string arg in argList) {
            FileInfo file = new FileInfo(SourcePath + arg);

            if (print) {
              Console.Out.WriteLine("Loading \"" + file.FullName +
                                    ".obj\".");
            }

            ReadObjectFile(file, linker);
          }
```

```
            linker.Generate(targetFile);
          }
          else if (print) {
            Console.Out.WriteLine(SourcePath + argList[0] +
                                  ".com is up-to-date.");
          }
        }

        // ...
      }
    }

    // ...

    public static void CompileSourceFile(FileInfo file) {
      // ...

      if (Start.Linux) {
        // ...
      }

      // ...

      if (Start.Windows) {
        FileInfo objectFile = new FileInfo(file.FullName + ".obj");
        BinaryWriter binaryWriter =
          new BinaryWriter(File.Open(objectFile.FullName, FileMode.Create));

        binaryWriter.Write(SymbolTable.StaticSet.Count);
        foreach (StaticSymbol staticSymbol in SymbolTable.GlobalStaticSet) {
          staticSymbol.Write(binaryWriter);
        }

        binaryWriter.Close();
      }
    }
  }
}
```

The **ReadObjectFile** method is called for reading an object file when it is not necessary to recompile the source file. An object file is made up by static object, and we read the object and add them to the linker.

```
    public static void ReadObjectFile(FileInfo file, Linker linker) {
      FileInfo objectFile = new FileInfo(file.FullName + ".obj");

      try {
        BinaryReader dataInputStream =
          new BinaryReader(File.OpenRead(objectFile.FullName));
```

The first value of the file is the number of static objects. For each object we call the **Load** method of the **StaticSymbolWindows** class and the **Add** method of the **Linker** class.

```
        int linkerSetSize = dataInputStream.ReadInt32();
        for (int count = 0; count < linkerSetSize; ++count) {
          StaticSymbolWindows staticSymbol = new StaticSymbolWindows();
          staticSymbol.Read(dataInputStream);
          linker.Add(staticSymbol);
        }
        dataInputStream.Close();
      }
      catch (Exception exception) {
        Console.Out.WriteLine(exception.StackTrace);
        Assert.Error(exception.Message);
      }
    }
```

13.1.2. Type Size

To begin with, we need new type sizes for the Windows environment. A character is, as always, one byte. A short is one byte, an integer or a pointer is two bytes, and a long is four bytes. Note that an integer and a pointer have equal sizes in the Windows environment, as opposed to the Linux environment.

TypeSize.cs

```
    static TypeSize() {
      // ...

      if (Start.Linux) {
        // ...
      }

      if (Start.Windows) {
        PointerSize = 2;
        SignedIntegerSize = 2;

        m_sizeMap.Add(Sort.Void, 0);
        m_sizeMap.Add(Sort.Function, 0);
        m_sizeMap.Add(Sort.Logical, 1);
        m_sizeMap.Add(Sort.Array, 2);
        m_sizeMap.Add(Sort.Pointer, 2);
        m_sizeMap.Add(Sort.String, 2);
        m_sizeMap.Add(Sort.SignedChar, 1);
        m_sizeMap.Add(Sort.UnsignedChar, 1);
        m_sizeMap.Add(Sort.SignedShortInt, 1);
        m_sizeMap.Add(Sort.UnsignedShortInt, 1);
        m_sizeMap.Add(Sort.SignedInt, 2);
        m_sizeMap.Add(Sort.UnsignedInt, 2);
        m_sizeMap.Add(Sort.SignedLongInt, 4);
        m_sizeMap.Add(Sort.UnsignedLongInt, 4);
        m_sizeMap.Add(Sort.Float, 4);
        m_sizeMap.Add(Sort.Double, 8);
        m_sizeMap.Add(Sort.LongDouble, 8);

        m_signedMap.Add(1, Sort.SignedChar);
        m_signedMap.Add(2, Sort.SignedInt);
        m_signedMap.Add(4, Sort.SignedLongInt);

        m_unsignedMap.Add(1, Sort.UnsignedChar);
        m_unsignedMap.Add(2, Sort.UnsignedInt);
        m_unsignedMap.Add(4, Sort.UnsignedLongInt);

        m_minValueMap.Add(Sort.Logical, 0);
        m_minValueMap.Add(Sort.SignedChar, -128);
        m_minValueMap.Add(Sort.UnsignedChar, 0);
        m_minValueMap.Add(Sort.SignedShortInt, -128);
        m_minValueMap.Add(Sort.UnsignedShortInt, 0);
        m_minValueMap.Add(Sort.SignedInt, -32768);
        m_minValueMap.Add(Sort.UnsignedInt, 0);
        m_minValueMap.Add(Sort.Array, 0);
        m_minValueMap.Add(Sort.Pointer, 0);
        m_minValueMap.Add(Sort.SignedLongInt, -2147483648);
        m_minValueMap.Add(Sort.UnsignedLongInt, 0);

        m_maxValueMap.Add(Sort.Logical, 1);
        m_maxValueMap.Add(Sort.SignedChar, 127);
        m_maxValueMap.Add(Sort.UnsignedChar, 255);
        m_maxValueMap.Add(Sort.SignedShortInt, 127);
        m_maxValueMap.Add(Sort.UnsignedShortInt, 255);
        m_maxValueMap.Add(Sort.SignedInt, 32767);
        m_maxValueMap.Add(Sort.UnsignedInt, 65535);
        m_maxValueMap.Add(Sort.Array, 65535);
        m_maxValueMap.Add(Sort.Pointer, 65535);
        m_maxValueMap.Add(Sort.SignedLongInt, 2147483647);
        m_maxValueMap.Add(Sort.UnsignedLongInt, 4294967295);
      }
    }
```

13.1.1. Static Symbol

The **StaticSymbolWindows** class is a sub class of **StaticSymbol** in Section 6.2.

StaticSymbolWindows.cs

```
using System;
using System.IO;
using System.Text;
using System.Globalization;
using System.Collections.Generic;

namespace CCompiler {
  public class StaticSymbolWindows : StaticSymbol {
```

The resulting code of each function definition and static variable and constant are stored in **m_byteList**.

```
    private List<byte> m_byteList;
```

A static variable or constant may access other static variables or constant, we store the addresses of the accesses in **m_accessMap**. Moreover, a function may call other functions, we store the addresses of the calls in **m_callMap**. The addresses are relative the beginning of the function or static object. They are later replaced by proper addresses by the linker. The difference between the addresses of the final accesses and calls is that the access addresses are absolute while the call addresses are relative the address of the call.

```
    private IDictionary<int,string> m_accessMap, m_callMap;
```

When a called function returns to the calling is, we store the return address in **m_returnSet**. The addresses are relative the address of the function call in the calling function. It is later replaced by a proper address by the linker.

```
    private ISet<int> m_returnSet;
```

We need the default constructor when loading object of the class from a file.

```
    public StaticSymbolWindows() {
      // Empty.
    }
```

The second constructor takes a unique name, a byte list, and an access map. This constructor is intended for static variables and constants.

```
    public StaticSymbolWindows(string uniqueName, List<byte> byteList = null,
                               IDictionary<int,string> accessMap = null)
     :base(uniqueName) {
      m_byteList = (byteList != null) ? byteList : (new List<byte>());
      m_accessMap = (accessMap != null) ? accessMap
                                        : (new Dictionary<int,string>());
      m_callMap = new Dictionary<int,string>();
      m_returnSet = new HashSet<int>();
    }
```

The third constructor takes a unique name, a byte list, and an access map, a call map, and a return set. This constructor is intended for function definitions.

```
    public StaticSymbolWindows(string uniqueName, List<byte> byteList,
                               IDictionary<int,string> accessMap,
                       IDictionary<int,string> callMap, ISet<int> returnSet)
     :base(uniqueName) {
      m_byteList = byteList;
      m_accessMap = accessMap;
      m_callMap = callMap;
      m_returnSet = returnSet;
    }

    public List<byte> ByteList {
      get { return m_byteList; }
    }

    public IDictionary<int,string> AccessMap {
      get { return m_accessMap; }
    }

    public IDictionary<int,string> CallMap {
      get { return m_callMap; }
    }

    public ISet<int> ReturnSet {
      get { return m_returnSet; }
    }
```

The **Write** method writes the byte list, access map, call map, and return set to the file. We call **base** to write the unique name of the object.

```
    public override void Write(BinaryWriter outStream) {
      base.Write(outStream);
```

We first write the size (number of bytes) of the byte list, then we write its bytes.

```
      outStream.Write(m_byteList.Count);
      foreach (sbyte b in m_byteList) {
        outStream.Write(b);
      }
```

In the same way, we write the size of the access map, and the then its key-value pairs.

```
      outStream.Write(m_accessMap.Count);
```

```
        foreach (KeyValuePair<int,string> entry in m_accessMap) {
          outStream.Write(entry.Key);
          outStream.Write(entry.Value);
        }

        outStream.Write(m_callMap.Count);
        foreach (KeyValuePair<int,string> entry in m_callMap) {
          outStream.Write(entry.Key);
          outStream.Write(entry.Value);
        }

        outStream.Write(m_returnSet.Count);
        foreach (int address in m_returnSet) {
          outStream.Write(address);
        }
      }
```

The Read method mirrors the Write method above. We read the unique name, byte list, access map, call map, and return set of the object.

```
      public override void Read(BinaryReader inStream) {
        base.Read(inStream);
```

First, we read the size of the byte list, and then its bytes.

```
        { m_byteList = new List<byte>();
          int byteListSize = inStream.ReadInt32();

          for (int index = 0; index < byteListSize; ++index) {
            byte b = inStream.ReadByte();
            m_byteList.Add(b);
          }
        }
```

In the same way, we read the size of the access map, and then its key-value pairs.

```
        { m_accessMap = new Dictionary<int,string>();
          int accessMapSize = inStream.ReadInt32();
          for (int index = 0; index < accessMapSize; ++index) {
            int address = inStream.ReadInt32();
            string name = inStream.ReadString();
            m_accessMap.Add(address, name);
          }
        }

        { m_callMap = new Dictionary<int,string>();
          int callMapSize = inStream.ReadInt32();
          for (int index = 0; index < callMapSize; ++index) {
            int address = inStream.ReadInt32();
            string name = inStream.ReadString();
            m_callMap.Add(address, name);
          }
        }

        { m_returnSet = new HashSet<int>();
          int returnSetSize = inStream.ReadInt32();
          for (int index = 0; index < returnSetSize; ++index) {
            int address = inStream.ReadInt32();
            m_returnSet.Add(address);
        }
      }
    }
  }
}
```

13.1.2. Static Value

When creating a static value in the Windows environment, the executable code is stored in a list of bytes. Moreover, we also need an access map to keep track of the accesses of the value. For instance, a pointer may point to another static value.

ConstantExpression.cs

```
    public static StaticSymbol Value(string uniqueName, Type type,
                                     object value) {
      // ...

      if (Start.Linux) {
        // ...
      }

      if (Start.Windows) {
        List<byte> byteList = new List<byte>();
        IDictionary<int,string> accessMap = new Dictionary<int,string>();
        AssemblyCodeGenerator.
          GenerateTargetWindows(assemblyCodeList, byteList,
                                accessMap, null, null);
        return (new StaticSymbolWindows(uniqueName, byteList, accessMap));
      }

      // ...
    }
  }
}
```

13.1.3. Function End

When creating a function in the Windows environment, we need the byte list and access map as in the static value case above. However, we also need a call map and a return set to keep track of the function calls and the return addresses of those calls.

MiddleCodeGenerator.cs

```
    public static void FunctionEnd(Statement statement) {

      // ...

      if (Start.Linux) {
        // ...
      }

      if (Start.Windows) {
        List<byte> byteList = new List<byte>();
        IDictionary<int,string> accessMap = new Dictionary<int,string>();
        IDictionary<int,string> callMap = new Dictionary<int,string>();
        ISet<int> returnSet = new HashSet<int>();
        AssemblyCodeGenerator.GenerateTargetWindows
        (assemblyCodeList, byteList, accessMap, callMap, returnSet);
      StaticSymbol staticSymbol =
        new StaticSymbolWindows(SymbolTable.CurrentFunction.UniqueName,
                                byteList, accessMap, callMap, returnSet);
      SymbolTable.StaticSet.Add(staticSymbol);
    }

    // ...
  }
```

13.1.4. Target Code Generation

```
  public static void GenerateTargetWindows
    (List<AssemblyCode> assemblyCodeList, List<byte> byteList,
     IDictionary<int,string> accessMap, IDictionary<int,string> callMap,
     ISet<int> returnSet) {
    AssemblyCodeGenerator objectCodeGenerator =
      new AssemblyCodeGenerator(assemblyCodeList);
    objectCodeGenerator.WindowsJumpInfo();
    objectCodeGenerator.WindowsByteList
                      (byteList, accessMap, callMap, returnSet);
  }
```

13.1.5. Exit

To exit the execution and return control to the surrounding operating system in the Windows environment, we perform an interrupt 33 system call, with the value 76 in the **ah** register. We store the return value in the **al** register. If there is no return value, we store zero in **al**.

AssemblyCodeGenerator.cs

```
  public void Exit(MiddleCode middleCode) {
    Symbol exitSymbol = (Symbol) middleCode[0];

    if (Start.Linux) {
      // ...
    }

    if (Start.Windows) {
      if (exitSymbol != null) {
        LoadValueToRegister(exitSymbol, Register.al);
      }
      else {
        AddAssemblyCode(AssemblyOperator.mov, Register.al,
                        BigInteger.Zero);
      }

      AddAssemblyCode(AssemblyOperator.mov, Register.ah,
                      (BigInteger) 76); // 0x4C
      AddAssemblyCode(AssemblyOperator.interrupt, (BigInteger) 33); // 0x21
    }
  }
```

13.1.6. Initialization Code

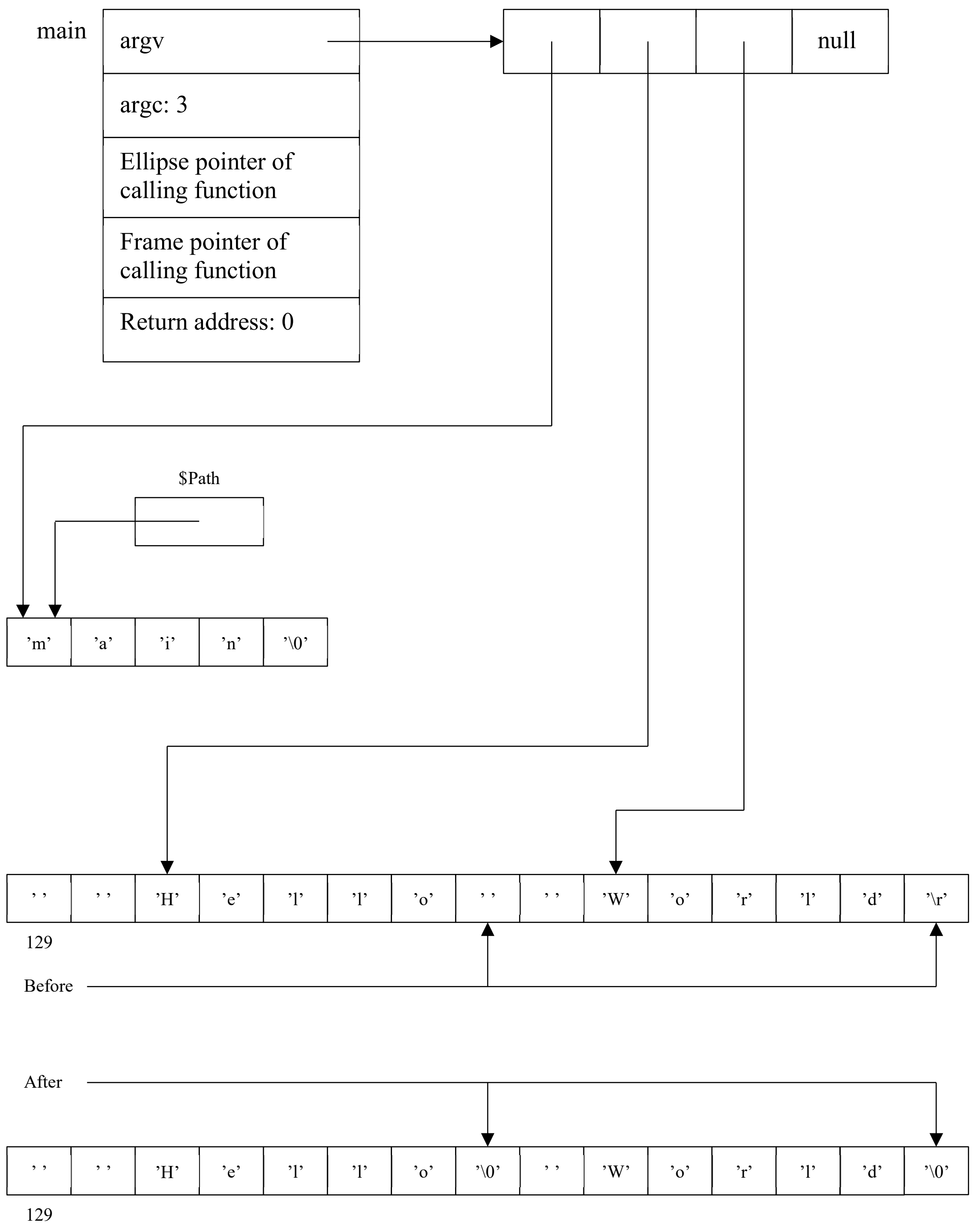
The initialization of data blocks differs between the Linux and the Windows environment.

AssemblyCodeGenerator.cs

```
    public static void InitializationCodeList() {
      List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();
      AddAssemblyCode(assemblyCodeList, AssemblyOperator.comment,
                      "Initializerialize Stack Pointer");

      AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                      AssemblyCode.RegularFrameRegister,
Linker.CallStackStart);

      if (Start.Linux) {
        // ...
      }

      if (Start.Windows) {
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_word,
                        null, 65534, (BigInteger)65534);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.fstcw,
                        AssemblyCode.RegularFrameRegister, 0);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.or_word,
                        AssemblyCode.RegularFrameRegister, 0,
                        (BigInteger) 0x0C00);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.fldcw,
                        AssemblyCode.RegularFrameRegister, 0);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Linker.CallStackStart, 0, BigInteger.Zero,
                        TypeSize.PointerSize);

        List<byte> byteList = new List<byte>();
        IDictionary<int, string> accessMap = new Dictionary<int, string>();
        IDictionary<int, string> callMap = new Dictionary<int, string>();
        ISet<int> returnSet = new HashSet<int>();
        AssemblyCodeGenerator.GenerateTargetWindows(assemblyCodeList,
                              byteList, accessMap, callMap, returnSet);
        StaticSymbol staticSymbol =
          new StaticSymbolWindows(AssemblyCodeGenerator.InitializerName,
                                  byteList, accessMap, callMap, returnSet);
        SymbolTable.StaticSet.Add(staticSymbol);
      }
    }
```

13.1.7. Command Line Arguments

The code for obtaining the command line arguments differs between the Linux code in Chapter XXX. The main difference is the arguments (excluding the file name) is stored at address 129, each word may begin with one or several spaces, and that the list is finished by a return character. The file name is unfortunately not stored, why we instead let the linker stored the name of the final executable file with the **$Path** name


main
argv
argc: 3
Ellipse pointer of calling function
Frame pointer of calling function
Return address: 0
null
$Path
'm' 'a' 'i' 'n' '\0'
' ' ' ' 'H' 'e' 'l' 'l' 'o' ' ' ' ' 'W' 'o' 'r' 'l' 'd' '\r'
129
Before
After
' ' ' ' 'H' 'e' 'l' 'l' 'o' '\0' ' ' 'W' 'o' 'r' 'l' 'd' '\0'
129

We start by saving the current value of **bp** in **si**, we will need it when we set the **main** parameter list at the end. We then move to **$Path** address to the current address of **bp** and increase it value by two bytes (a pointer in the Windows environment is bytes long). By now we have the name of the execution file stored at the beginning of the array.

```
    mov si, bp
    mov word [bp], $Path
    add bp, 2
```

We let ax hold the number of arguments, which is one to begin with since we have already included the execution file name. We let **bx** hold the current address of the characters in the command line, which starts at address 129. We begin by checking the special case where we have no arguments at all. In that case, address 129 hold the return character (with ASCII value 13) and we jump to **ListDone**.

```
    mov ax, 1
    mov bx, 129
    cmp byte [bx], 13
    je ListDone
```

If the argument list is not empty, we start by iterating through the looping through the potential initial space characters. They shall not be included in the argument word. We continue to increase **bx** and jump back to **SpaceLoop** an long as we encounter space characters. Note that we cannot encounter a return character since they only occur after the last argument word. The case without any arguments words, where the list only holds one single return character, we have dealt with above.

```
SpaceLoop:
    cmp byte [bx], 32
    jne WordStart
    inc bx
    jmp SpaceLoop
```

When we encounter a word different from space, we have found the beginning a new argument word. We increment **ax**, holding the number of arguments, with one. We also store the address of the first character in the word and the array pointed at by **bp** and increase bp with two bytes.

```
WordStart:
    inc ax
    mov word [bp], bx
    add bp, 2
```

Then we iterate the argument word, until we encounter a space (ASCII value 32) of a return (ASCII value 13). In case of a space character, the current word is done and we jump to **WordDone** to address the next word. In case of a return character, the list is done and we jump to **ListDone**. If none of those cases apply, we increase **bx** and jump back to **WordLoop** to handle the next character of the word.

```
WordLoop:
    cmp byte [bx], 32
    je WordDone
    cmp byte [bx], 13
    je ListDone
    inc bx
    jmp WordLoop
```

When we are finish with a word, we replace its finishing space with a zero character. We also increase **bx** to let it point at the character following the word, and jump back to **SpaceLoop** to find the next word in the list.

```
WordDone:
    mov byte [bx], 0
    inc bx
    jmp SpaceLoop
```

When we have iterated through the whole list, there are som finishing up to do. Just as when we encountered a space character at the end of a word, we need to change the last character to a zero character also when we encounter a return character. We also add a zero to the array, since the last entry in the argument array shall be a null pointer. Finally, we increase **bx** with two.

```
ListDone:
    mov byte [bx], 0
    mov word [bp], 0
    add bp, 2
```

By now, **bx** points at the beginning of the first activation record, for the initial call to **main**. We need to set its return address (offset zero) to zero, for the return statement to exit the execution, and the **argc** and **argv** parameters. We mov the number of arguments (**ax**) to **argc** (offset six), and the beginning of the argument list (**si**) to **argv** (offset eight).

```
    mov word [bp], 0
    mov [bp + 6], ax
    mov [bp + 8], si
```

Below follows the code that generates the argument list code.

AssemblyCodeGenerator.cs

```
    public static void ArgumentCodeList() {
      List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();

      if (Start.Linux) {
        // ...
      }

      if (Start.Windows) {
        /*    mov si, bp
              mov word [bp], $Path
              add bp, 2
              mov ax, 1
              mov bx, 129
              cmp byte [bx], 13
              je ListDone */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.si, Register.bp);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_word,
                        Register.bp, 0, AssemblyCodeGenerator.PathName);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.add,
                        Register.bp, (BigInteger) 2);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.ax, BigInteger.One);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.bx, (BigInteger) 129);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.cmp_byte,
                        Register.bx, 0, (BigInteger) 13);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.je,
                        null, assemblyCodeList.Count + 17);

        /* SpaceLoop:
             cmp byte [bx], 32
             jne WordStart
             inc bx
             jmp SpaceLoop */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.cmp_byte,
                        Register.bx, 0, (BigInteger) 32);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.jne,
                        null, assemblyCodeList.Count + 3);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.inc, Register.bx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.jmp,
                        null, assemblyCodeList.Count - 3);

        /* WordStart:
             inc ax
             mov word [bp], bx
             add bp, 2 */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.inc, Register.ax);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                        Register.bp, 0, Register.bx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.add,
                        Register.bp, (BigInteger) 2);

        /* WordLoop:
             cmp byte [bx], 32
             je WordDone
             cmp byte [bx], 13
             je ListDone
             inc bx
             jmp WordLoop */

        AddAssemblyCode(assemblyCodeList, AssemblyOperator.cmp_byte,
                        Register.bx, 0, (BigInteger) 32);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.je,
                        null, assemblyCodeList.Count + 5);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.cmp_byte,
                        Register.bx, 0, (BigInteger) 13);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.je,
                        null, assemblyCodeList.Count + 6);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.inc, Register.bx);
        AddAssemblyCode(assemblyCodeList, AssemblyOperator.jmp,
                        null, assemblyCodeList.Count - 5);

        /* WordDone:
             mov byte [bx], 0
```

```
      inc bx
      jmp SpaceLoop */

  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_byte,
                  Register.bx, 0, BigInteger.Zero);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.inc, Register.bx);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.jmp,
                  null, assemblyCodeList.Count - 15);

  /* ListDone:
      mov byte [bx], 0
      mov word [bp], 0
      add bp, 2
      mov word [bp], 0
      mov [bp + 6], ax
      mov [bp + 8], si */

  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_byte,
                  Register.bx, 0, BigInteger.Zero);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_word,
                  Register.bp, 0, BigInteger.Zero);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.add,
                  Register.bp, (BigInteger) 2);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov_word,
                  Register.bp, 0, BigInteger.Zero);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                  Register.bp, 6, Register.ax);
  AddAssemblyCode(assemblyCodeList, AssemblyOperator.mov,
                  Register.bp, 8, Register.si);

  List<byte> byteList = new List<byte>();
  IDictionary<int, string> accessMap = new Dictionary<int, string>();
  IDictionary<int, string> callMap = new Dictionary<int, string>();
  ISet<int> returnSet = new HashSet<int>();
  AssemblyCodeGenerator.
    GenerateTargetWindows(assemblyCodeList, byteList,
                          accessMap, callMap, returnSet);
  StaticSymbol staticSymbol =
    new StaticSymbolWindows(AssemblyCodeGenerator.ArgsName, byteList,
                            accessMap, callMap, returnSet);
  SymbolTable.StaticSet.Add(staticSymbol);
}
```
}

13.1.1. Windows Jump Info

The **WindowsJumpInfo** method first changes the jump addresses from middle code line numbers to assembly code line numbers, and then to relative byte size addresses. To begin with we need the **middleToAssemblyMap** map to keep track of the assembly code addresses.

AssemblyCodeGenerator.cs

```
private void WindowsJumpInfo() {
  IDictionary<int,int> middleToAssemblyMap = new Dictionary<int,int>();
```

We iterate through the assembly code list and add the assembly code line number each time we encounter a new-middle-code instruction. We really need just the line numbers of first of the assembly line that corresponds to a middle code instruction, since all jumps are to those addresses.

```
for (int assemblyIndex = 0;
     assemblyIndex < m_assemblyCodeList.Count; ++assemblyIndex) {
  AssemblyCode assemblyCode = m_assemblyCodeList[assemblyIndex];

  if (assemblyCode.Operator == AssemblyOperator.new_middle_code) {
    int middleIndex = (int) assemblyCode[0];
    middleToAssemblyMap.Add(middleIndex, assemblyIndex);
    assemblyCode.Operator = AssemblyOperator.empty;
  }
}
```

Then we iterate through the assembly code list and change all jump targets from middle code line numbers to assembly code line numbers. Note that the middle code line number was stored in the third position (index 2) of the instruction, and that we set the assembly line number to the second position (index 1).

```
for (int line = 0; line < m_assemblyCodeList.Count; ++line) {
  AssemblyCode assemblyCode = m_assemblyCodeList[line];

  if (/*(assemblyCode[0] == null) &&*/ !(assemblyCode[1] is int) &&
      (assemblyCode.IsRelationNotRegister() ||
       assemblyCode.IsJumpNotRegister())) {
    int middleTarget = (int) assemblyCode[2];
    assemblyCode[1] = middleToAssemblyMap[middleTarget];
  }
}
```

When we have changed the jump targets from middle code line numbers to assembly code line numbers, the second step is to change the targets into the number of bytes between the next assembly instruction (the instruction following the current instruction) and the target instruction, which is a negative value in case of backward jumps.

First, we need the **assemblyToByteMap** map to keep track of the address of each assembly code instruction, counted in bytes from the beginning of the code. Then we iterate through the assembly code list and add the byte address of each instruction. At the moment, all jump instructions are equally large (in bytes), but that may change later on.

```
IDictionary<int,int> assemblyToByteMap = new Dictionary<int,int>();

{ int byteSize = 0, line = 0;
  foreach (AssemblyCode assemblyCode in m_assemblyCodeList) {
    assemblyToByteMap.Add(line++, byteSize);

    if (!(assemblyCode.IsRelationNotRegister() ||
          assemblyCode.IsJumpNotRegister())) {
      byteSize += assemblyCode.ByteList().Count;
    }
  }
  assemblyToByteMap.Add(m_assemblyCodeList.Count, byteSize);
}
```

When we change the assembly code targets to relative byte targets, we may have to iterate several times because short jumps (between -128 and 127, inclusive) instructions are smaller than long jumps instructions. At the beginning, all jumps are long jumps, but some of them may be changed to short jump during the iteration.

```
while (true) {
  for (int line = 0; line < (m_assemblyCodeList.Count - 1); ++line) {
    AssemblyCode thisCode = m_assemblyCodeList[line],
                 nextCode = m_assemblyCodeList[line + 1];

    if (thisCode.IsRelationNotRegister() ||
        thisCode.IsJumpNotRegister()) {
      int assemblyTarget = (int) thisCode[1];
```

The **byteSource** variable is the byte address of the target assembly instruction following the current instruction, **byteTarget** is the byte address of the target instruction, and **byteDistance** is difference between **byteTarget** and **byteSource**, which becomes the final target address.

```
            int byteSource = assemblyToByteMap[line + 1],
                byteTarget = assemblyToByteMap[assemblyTarget];
            int byteDistance = byteTarget - byteSource;
            thisCode[0] = byteDistance;
          }
        }
```

When the jump instruction has been given proper target address, we iterate through the assembly list and update the **assemblyToByteMap**. Some byte address may have been changed because some long jump instructions have been changed to short jump instructions. If any address has been changed, **update** is set to true and the overall loop will perform another iteration.

```
        bool update = false;
        { int byteSize = 0, line = 0;
          foreach (AssemblyCode objectCode in m_assemblyCodeList) {
            if (assemblyToByteMap[line] != byteSize) {
              assemblyToByteMap[line] = byteSize;
              update = true;
            }

            byteSize += objectCode.ByteList().Count;
            ++line;
          }
        }
```

If nothing has been changed from the previous iteration, every assembly jump instruction holds the correct target address, and we break the loop.

```
        if (!update) {
          break;
        }
      }
```

When the addresses have been set, we need to erase any potential zero jumps; that is, jumps to the next instruction.

```
      foreach (AssemblyCode objectCode in m_assemblyCodeList) {
        if (objectCode.IsRelationNotRegister() ||
            objectCode.IsJumpNotRegister()) {
          int byteDistance = (int) objectCode[0];

          if (byteDistance == 0) {
            objectCode.Operator = AssemblyOperator.empty;
          }
        }
      }
```

Finally, we need to iterate through the assembly code list, and change the return addresses in the same way as the byte target addresses above.

```
      for (int line = 0; line < m_assemblyCodeList.Count; ++line) {
        AssemblyCode assemblyCode = m_assemblyCodeList[line];

        if (assemblyCode.Operator == AssemblyOperator.return_address) {
          int middleAddress = (int) ((BigInteger) assemblyCode[2]);
          int assemblyAddress = middleToAssemblyMap[middleAddress];
          int byteAddress = assemblyToByteMap[assemblyAddress];
          assemblyCode[2] = (BigInteger) byteAddress;
        }
      }
    }
```

13.1.2. Windows Byte List

The **WindowsByteList** method iterates through the assembly code list and generates the byte list for each instruction by calling **ByteList** in **AssemblyCode**. It also adds names to the **accessMap**, **callMap**, and **returnSet** parameters.

AssemblyCodeGenerator.cs

```
    private void WindowsByteList(List<byte> byteList,
                                 IDictionary<int,string> accessMap,
                                 IDictionary<int,string> callMap,
                                 ISet<int> returnSet) {
      foreach (AssemblyCode assemblyCode in m_assemblyCodeList) {
```

We add the byte list of the instruction to the byte list parameter.

```
        byteList.AddRange(assemblyCode.ByteList());
```

Unless the instruction is a label, comment, or zero sequence, we check whether there are any names in the code that shall be added to the access map or the call map, or the return set.

```
        if ((assemblyCode.Operator != AssemblyOperator.label) &&
            (assemblyCode.Operator != AssemblyOperator.comment) &&
            (assemblyCode.Operator != AssemblyOperator.define_zero_sequence)){
```

If the instruction is an address definition, we add its name to the access map.

```
          if (assemblyCode.Operator == AssemblyOperator.define_address) {
            string name = (string) assemblyCode[0];
            accessMap.Add(byteList.Count - TypeSize.PointerSize, name);
          }
```

If the instruction is a value definition, we need to look into its sort. If it is a pointer or a static address, we add its name.

```
          else if (assemblyCode.Operator == AssemblyOperator.define_value) {
            Sort sort = (Sort) assemblyCode[0];
            object value = assemblyCode[1];

            if (sort == Sort.Pointer) {
              if (value is string) {
                accessMap.Add(byteList.Count - TypeSize.PointerSize,
                              (string) value);
              }
              else if (value is StaticAddress) {
                StaticAddress staticAddress = (StaticAddress) value;
                accessMap.Add(byteList.Count - TypeSize.PointerSize,
                              staticAddress.UniqueName);
              }
            }
          }
```

In case of a function call, add the name of the function to be called to the call map.

```
          else if ((assemblyCode.Operator == AssemblyOperator.call) &&
                   (assemblyCode[0] is string)) {
            string calleeName = (string) assemblyCode[0];
            int address = byteList.Count - TypeSize.PointerSize;
            callMap.Add(address, calleeName);
          }
```

In case of a function return, we add the address the return set.

```
          else if (assemblyCode.Operator == AssemblyOperator.return_address) {
            int address = byteList.Count - TypeSize.PointerSize;
            returnSet.Add(address);
          }
```

If the first operand is a string, we have two cases: if the third operand is a value of **BigInteger**, we need decrease the address with its size and with the size of a pointer. Otherwise, we just decrease the address with the pointer size.

```
          else if (assemblyCode[0] is string) { // Add [g + 1], 2
            if (assemblyCode[2] is BigInteger) {
              int size = AssemblyCode.SizeOfValue((BigInteger)assemblyCode[2],
                                                  assemblyCode.Operator);
              int address = byteList.Count - TypeSize.PointerSize - size;
              accessMap.Add(address, (string) assemblyCode[0]);
            }
            else {
              int address = byteList.Count - TypeSize.PointerSize;
              accessMap.Add(address, (string) assemblyCode[0]);
            }
          }
```

If the second or third operand is a string, we add its name the address, which is the size of the instruction minus the pointer size.

```
          else if (assemblyCode[1] is string) { // mov ax, [g + 1]; mov ax, g
            int address = byteList.Count - TypeSize.PointerSize;
            accessMap.Add(address, (string) assemblyCode[1]);
          }
          else if (assemblyCode[2] is string) { // Add [bp + 2], g
            int address = byteList.Count - TypeSize.PointerSize;
            accessMap.Add(address, (string) assemblyCode[2]);
          }
        }
      }
    }
  }
}
```

13.1.3. Byte List

The **ByteList** method return the assembly code instruction converted to a list of bytes.

AssemblyCode.cs

```
    public List<byte> ByteList() {
      object operand0 = m_operandArray[0],
             operand1 = m_operandArray[1],
             operand2 = m_operandArray[2];
```

For empty instruction, or for a label or comment instruction we just return an empty list.

```
      if ((Operator == AssemblyOperator.empty) ||
          (Operator == AssemblyOperator.label) ||
          (Operator == AssemblyOperator.comment)) {
        return (new List<byte>());
      }
```

For the definition of an address, we add the offset to the byte list.

```
      else if (Operator == AssemblyOperator.define_address) {
        int offset = (int) operand1;
        List<byte> byteList = new List<byte>(new byte[TypeSize.PointerSize]);
        LoadByteList(byteList, 0, TypeSize.PointerSize, (BigInteger) offset);
        return byteList;
      }
```

For a zero value, we just return a list of zeros.

```
      else if (Operator == AssemblyOperator.define_zero_sequence) {
        int size = (int) operand0;
        return (new List<byte>(new byte[size]));
      }
```

A value can be a floating value, a string, a static address, a pointer, or an integral value.

```
      else if (Operator == AssemblyOperator.define_value) {
        Sort sort = (Sort) operand0;
        object value = operand1;
        List<byte> byteList;
```

In case of float, double, or long double, we call **GetBytes** in the standard library **BitConverter** class.

```
        if (sort == Sort.Float) {
          float floatValue = (float) ((decimal) operand0);
          byteList =  new List<byte>(BitConverter.GetBytes(floatValue));
        }
        else if ((sort == Sort.Double) || (sort == Sort.LongDouble)) {
          double doubleValue = (double) ((decimal) value);
          byteList = new List<byte>(BitConverter.GetBytes(doubleValue));
        }
```

In case of string we fill the byte list with the characters of the string and add the terminating zero character.

```
        else if (sort == Sort.String) {
          string text = (string) value;
          byteList = new List<byte>();

          foreach (char c in text) {
            byteList.Add((byte) c);
          }
          byteList.Add((byte) 0);
        }
```

Otherwise, we have an integral of pointer value. If the value is a static address, we load its offset into the byte list. The address itself will be added by the linker later on. If the value is not a static address, we just load it into the byte list.

```
        else {
          int size = TypeSize.Size(sort);
          byteList = new List<byte>(new byte[size]);

          if (value is StaticAddress) {
            StaticAddress staticAddress = (StaticAddress) value;
            LoadByteList(byteList, 0, size,
                         (BigInteger) staticAddress.Offset);
          }
          else {
            LoadByteList(byteList, 0, size, (BigInteger) value);
          }
        }

        return byteList;
      }
```

Next, we have a jump or a call to a register; that is, a jump of call to an address stored in a register.

```
      else if (IsJumpRegister() || IsCallRegister()) {
        Register register = (Register) operand0;
        return LookupByteArray(AssemblyOperator.jmp, register);
      }
```

A regular function call is equivalent with a long jump. The actual address of the called function will be set by the linker later on.

```
      else if (IsCallNotRegister()) {
        return LookupByteArray(AssemblyOperator.jmp, TypeSize.PointerSize);
      }

      else if (Operator == AssemblyOperator.return_address) {
        Register register = (Register) operand0;
        int offset = (int) operand1;
        int size = SizeOfValue(offset);
        int address = (int)((BigInteger)operand2);
        List<byte> byteList =
          LookupByteArray(AssemblyOperator.mov_word, register,
                          size, TypeSize.PointerSize);
        LoadByteList(byteList, byteList.Count - (size + TypeSize.PointerSize),
                     size, offset);
        LoadByteList(byteList, byteList.Count - TypeSize.PointerSize,
                     TypeSize.PointerSize, address);
        return byteList;
      }
```

The size of the address is either one (short jump) or two (long jump), which we load into the byte list.

```
      else if (IsRelationNotRegister() || IsJumpNotRegister()) {
        int address = (int) operand0;
```

If the address is zero, a jump to the next instruction, we just return an empty list, since zero jumps shall not generate any code.

```
        if (address == 0) {
          return (new List<byte>());
        }
        else {
          int size = SizeOfValue(address);

          if (address == 127) { // XXX
            size = 2;
          }

          List<byte> byteList = LookupByteArray(Operator, size);
          LoadByteList(byteList, byteList.Count - size, size, address);
          return byteList;
        }
      }
```

Let us continue with the operations that perform computations and let us start with the nullary operators; that is, operators that take no operands. For instance: **lahf**. We call **LookupByteArray** with the operator as argument to obtain and return the assay of bytes corresponding to the assembly instruction.

```
      else if ((operand0 == null) && (operand1 == null) &&
               (operand2 == null)) {
        return LookupByteArray(Operator);
      }
```

Then we continue with the unary operators; that is, operators that take one operand. We have three different cases:

Category	Description	Examples
1	unary operator register	inc ax
2	unary operator integer value	int 33
3	unary operator [register + offset] unary operator [static name + offset]	neg word [bp + 2] not word [stdin + 4]

The operators of the first category take a register as operand. We call **LookupByteArray** with the operator and register.

```
      // inc ax
      else if ((operand0 is Register) && (operand1 == null) &&
               (operand2 == null)) {
        Register register = (Register) operand0;
        return LookupByteArray(Operator, register);
      }
```

The operators of the second category take an integer value as operand. For instance: **int 33**. We call **LookupByteArray** just like in the previous cases. But we must also load the array with the integer value of the operation, which we do by calling **LoadByteList** with the value and its size. We determine the size of the value by calling **SizeOfValue**. Note that we have to subtract the size of the value from the size of the byte list to obtain the location to insert the value in the byte list. Integer values are always **BigInteger** objects.

```
      // int 33
      else if ((operand0 is BigInteger) && (operand1 == null) &&
               (operand2 == null)) {
        BigInteger value = (BigInteger) operand0;
        int size = SizeOfValue(value);
        List<byte> byteList = LookupByteArray(Operator, size);
        LoadByteList(byteList, byteList.Count - size, size, value);
        return byteList;
      }
```

In the third category of the unary operations the operand is an address, made up by a register and an offset or a static variable and an offset. For instance: **inc [bp + 2]** or **inc [global + 4]**. Note that offsets are always **int** values. If the address includes a register, the size of the offset is determined by calling **SizeOfValue**. If the address instead includes a static variable, the offset size is always the size of a pointer.

```
        // inc [bp + 2]; inc [global + 4]
        else if (((operand0 is Register) || (operand0 is string)) &&
                 (operand1 is int) && (operand2 == null)) {
          int offset = (int) operand1;
          int size = (operand0 is Register) ? SizeOfValue(offset)
                                            : TypeSize.PointerSize;
          List<byte> byteList = LookupByteArray(Operator, operand0, size);
          LoadByteList(byteList, byteList.Count - size, size, offset);
          return byteList;
        }
```

For binary operators, we have seven categories. Note that we in category six have a special case, where we load a value into an address given only by an offset. This case is used in initialization at the beginning of the code.

Category	Description	Examples
1	binary operator register, register	mov ax, bx
2	binary operator register, static name	add ax, stdin
3	binary operator register, integer value	sub ax, 123
4	binary operator [register + offset], register binary operator [static name + offset], register	mov [bp + 2], bx mov [stdin + 4], bx
5	binary operator [register + offset], static name binary operator [static name + offset], static name	add word [bp + 2], stdin add word [stdin + 4], stdin
6	binary operator [register + offset], integer value binary operator [static name + offset], integer value binary operator [offset], integer value	sub word [bp + 2], 123 sub word [stdin + 4], 123 mov word [65534], 65534
7	binary operator register, [register + offset] binary operator register, [static name + offset]	mov ax, [bp + 2] add ax, [stdin +4]

In category one, we assign a register the value of another register.

```
        // 1: mov ax, bx
        else if ((operand0 is Register) && (operand1 is Register) &&
                 (operand2 == null)) {
          Register toRegister = (Register) operand0,
                   fromRegister = (Register) operand1;
          return LookupByteArray(Operator, toRegister, fromRegister);
        }
```

In category two, we assign a register the address a static variable. We have to load the zero value into the byte list, to mark that there is not offset. The linker will add the actual address of the variable later on.

```
// 2: mov ax, global
else if ((operand0 is Register) && (operand1 is string) &&
         (operand2 == null)) {
  Register register = (Register) operand0;
  List<byte> byteList =
    LookupByteArray(Operator, register, TypeSize.PointerSize);
  LoadByteList(byteList, byteList.Count - TypeSize.PointerSize,
               TypeSize.PointerSize, 0);
  return byteList;
}
```

In category three, we load an integral value into a register. We must check the operator, in case of **mov** or **and**, the size of the value is given by the register. Otherwise, the size is given by the value.

```
// 3: mov ax, 123
else if ((operand0 is Register) && (operand1 is BigInteger) &&
         (operand2 == null)) {
  Register register = (Register) operand0;
  BigInteger value = (BigInteger) operand1;
  int size = ((Operator == AssemblyOperator.mov) ||
              (Operator == AssemblyOperator.and))
             ? SizeOfRegister(register) : SizeOfValue(value);
  List<byte> byteList = LookupByteArray(Operator, register, size);
  LoadByteList(byteList, byteList.Count - size, size, value);
  return byteList;
}
```

In category four, we assign the value of a register to an address, defined by a register and an offset or a static variable and an offset. Also in this case, we need to obtain the size of the offset. Its is given by **SizeOfValue** in case of a register address. In case of a static variable address, the size is always the size of a pointer.

```
// 4: mov [bp + 2], ax; mov [global + 4], ax
else if (((operand0 is Register) || (operand0 is string)) &&
         (operand1 is int) && (operand2 is Register)) {
  Register register = (Register) operand2;
  int offset = (int) operand1;
  int size = (operand0 is Register) ? SizeOfValue(offset)
                                    : TypeSize.PointerSize;
  List<byte> byteList =
    LookupByteArray(Operator, operand0, size, register);
  LoadByteList(byteList, byteList.Count - size, size, offset);
  return byteList;
}
```

In category five, we assign the address of a static variable to an address, given by a register and an offset or a static variable and an offset. We load the offset into the byte list. However, we also need to load the zero value to the byte list, to mark that the address is zero at the moment. The linker will add the actual address of the variable later on.

```
// 5: mov [bp + 2], global; mov [global + 4], global
else if (((operand0 is Register) || (operand0 is string)) &&
         (operand1 is int) && (operand2 is string)) {
  int offset = (int) operand1;
  int size = (operand0 is Register) ? SizeOfValue(offset)
                                    : TypeSize.PointerSize;
  List<byte> byteList = LookupByteArray(Operator, operand0,
                                    size, TypeSize.PointerSize);
  LoadByteList(byteList, byteList.Count -
              (size + TypeSize.PointerSize), size, offset);
  LoadByteList(byteList, byteList.Count - TypeSize.PointerSize,
               TypeSize.PointerSize, 0);
  return byteList;
}
```

In category six, we assign an integral value to an address. We need to determine the size of both the offset and the value, and then load them both into the byte list. We also have the special case where the static variable is null, which occurs in the initialization in the beginning of the code, such as **mov word [65534], 65534**.

```
        // 6: mov [bp + 2], 123; mov [global + 4], 123
        // mov [null + 4], 123; Special case
        else if (((operand0 is Register) || (operand0 is string) ||
                  (operand0 == null)) && (operand1 is int) &&
                 (operand2 is BigInteger)) {
          int offset = (int) operand1;
          BigInteger value = (BigInteger) operand2;
          int offsetSize = (operand0 is Register) ? SizeOfValue(offset)
                                                  : TypeSize.PointerSize,
              valueSize = SizeOfValue(value, Operator);
          List<byte> byteList =
            LookupByteArray(Operator, operand0, offsetSize, valueSize);
          LoadByteList(byteList, byteList.Count - (offsetSize + valueSize),
                       offsetSize, offset);
          LoadByteList(byteList, byteList.Count - valueSize,
                       valueSize, value);
          return byteList;
        }
```

In category seven, we assign a register the value stored on an address, defined by a register and an offset or a static variable and an offset.

```
        // 7: mov ax, [bp + 2]; mov ax, [global + 4]
        else if ((operand0 is Register) && ((operand1 is Register) ||
                  (operand1 is string)) && (operand2 is int)) {
          Register register = (Register) operand0;
          int offset = (int) operand2;
          int size = (operand1 is Register) ? SizeOfValue(offset)
                                            : TypeSize.PointerSize;
          List<byte> byteList =
            LookupByteArray(Operator, register, operand1, size);
          LoadByteList(byteList, byteList.Count - size, size, offset);
          return byteList;
        }
      }
```

The **LoadByteList** method load the byte list with a value.

```
      public static void LoadByteList(IList<byte> byteList, int index,
                                      int size, BigInteger value) {
        switch (size) {
          case 1: {
              if (value < 0) {
                byteList[index] = (byte) ((sbyte) value);
              }
```

```
            else {
              byteList[index] = (byte) value;
            }
          }
          break;

        case 2: {
            if (value < 0) {
              short shortValue = (short) value;
              byteList[index] = (byte) ((sbyte) shortValue);
              byteList[index + 1] = (byte) ((sbyte) (shortValue >> 8));
            }
            else {
              ushort ushortValue = (ushort) value;
              byteList[index] = (byte) ushortValue;
              byteList[index + 1] = (byte) (ushortValue >> 8);
            }
          }
          break;

        case 4: {
            if (value < 0) {
              int intValue = (int) value;
              byteList[index] = (byte) ((sbyte) intValue);
              byteList[index + 1] = (byte) ((sbyte) (intValue >> 8));
              byteList[index + 2] = (byte) ((sbyte) (intValue >> 16));
              byteList[index + 3] = (byte) ((sbyte) (intValue >> 24));
            }
            else {
              uint uintValue = (uint) value;
              byteList[index] = (byte) uintValue;
              byteList[index + 1] = (byte) (uintValue >> 8);
              byteList[index + 2] = (byte) (uintValue >> 16);
              byteList[index + 3] = (byte) (uintValue >> 24);
            }
          }
          break;
      }
    }
```

The **LookupByteArray** method looks up the assembly code instruction in the **MainArrayMap** map in **ObjectCodeTable**.

```
    public static List<byte> LookupByteArray(AssemblyOperator objectOp,
                      object operand1 = null, object operand2 = null,
                      object operand3 = null) {
      if ((objectOp == AssemblyOperator.shl) ||
          (objectOp == AssemblyOperator.shr)) {
        operand1 = (operand1 is BigInteger) ? 0L : operand1;
        operand2 = (operand2 is BigInteger) ? 0L : operand2;
        operand3 = (operand3 is BigInteger) ? 0L : operand3;
      }
```

If any of the operands is a string, it shall be changed to null.

```
      operand0 = (operand0 is string) ? null : operand0;
      operand1 = (operand1 is string) ? null : operand1;
      operand2 = (operand2 is string) ? null : operand2;
      ObjectCodeInfo info =
        new ObjectCodeInfo(objectOp, operand1, operand2, operand3);
      byte[] byteArray = ObjectCodeTable.MainArrayMap[info];
      Assert.ErrorXXX(byteArray != null);
      List<byte> byteList = new List<byte>();

      foreach (byte b in byteArray) {
        byteList.Add(b);
      }

      return byteList;
    }
```

13.2. The Linker

The linker is the final part of the compilation process. Its merges together the compiled files and generates an executable file.

13.2.1. The Linker Class

The **Linker** class merge code of the object files and modifies the global accesses, function calls, and return assignments, and saves the final code in a .com file, which is executable in 16-bits Windows. More specific, the linker has the following tasks:

- First, we load all function and static variables from all the object files and to make sure that two elements do not share the same name.
- Then we identify the **main** function and trace all functions and global variables reachable from **main**. All elements not reachable from **main** are removed.
- For each function or global variable, we modify all accesses, and for each function we modify all calls and return assignments.
- Finally, we save the code and data for all elements reachable from **main** in the .com file.

Linker.cs
```
using System;
using System.IO;
using System.Collections.Generic;

namespace CCompiler {
  public class Linker {
```

There are several maps and lists. The **m_globalMap** map holds all the static symbols of the source code, while **m_globalList** holds only the symbols that are reachable (directly or indirectly) from the **main** function. The final code is generated from the symbols of **m_globalList**, the other symbols are omitted from the final code. The **m_addressMap** holds the address of each symbol in **m_globalList**. Finally, **m_totalSize** holds the current total size of the symbols of **m_globalList** and is used to add the positions of the symbols to **m_addressMap**. The **StackStart** field is used to identify the first address after the code and data.

```
    private IDictionary<string,StaticSymbolWindows> m_globalMap =
      new Dictionary<string,StaticSymbolWindows>();
    private List<StaticSymbolWindows> m_globalList =
      new List<StaticSymbolWindows>();
    private IDictionary<string,int> m_addressMap =
      new Dictionary<string,int>();
    private int m_totalSize = 256;
    public static string StackStart = Symbol.SeparatorId + "StackTop";
```

The **Add** method is called for each static symbol of the source code. If two symbols have the same unique name not ending in the number identification character ('#'), we have two symbols with external linkage, which is not allowed. However, if the name ends with the identification character, it only means that two numerical constants with the same value is present in two different files, with is allowed. In that case we simply refrain from adding the second symbol to the map.

```
    public void Add(StaticSymbol staticSymbol) {
      StaticSymbolWindows staticSymbolWindows =
        (StaticSymbolWindows) staticSymbol;
      string uniqueName = staticSymbolWindows.UniqueName;

      if (!m_globalMap.ContainsKey(uniqueName)) {
        m_globalMap.Add(uniqueName, staticSymbolWindows);
      }
      else {
        Assert.Error(uniqueName.EndsWith(Symbol.NumberId),
                     SimpleName(uniqueName), Message.Duplicate_global_name);
      }
    }
```

The **Generate** method writes the final executable code to the target file. Its task is to identify the symbols reachable from the **main** function, and to replace the names of accessed symbols with called functions with proper addresses as well as to replace the function returns addresses with proper addresses. However, we need to start by including the code for the initialization execution code and, optionally, the code for handling the command line arguments.

```
    public void Generate(FileInfo targetFile) {
```

The code for initialization shall be added to the code at the beginning, before the **main** function

```
      StaticSymbolWindows initializerInfo =
        m_globalMap[AssemblyCodeGenerator.InitializerName];
      m_globalList.Add(initializerInfo);
      m_totalSize += initializerInfo.ByteList.Count;
      m_addressMap.Add(AssemblyCodeGenerator.InitializerName, 0);
```

In case of command line arguments, we add its symbol after the initialization code symbol and before the **main** function code.

```
      StaticSymbolWindows pathNameSymbol = null;
      if (m_globalMap.ContainsKey(AssemblyCodeGenerator.ArgsName)) {
        StaticSymbolWindows argsInfo =
          m_globalMap[AssemblyCodeGenerator.ArgsName];
        m_globalList.Add(argsInfo);
        Console.Out.WriteLine(argsInfo.UniqueName);
        m_totalSize += argsInfo.ByteList.Count;
        m_addressMap.Add(AssemblyCodeGenerator.ArgsName, 0);
```

We also need to add the path name of final executable file.

```
        List<byte> byteList = new List<byte>();
        IDictionary<int, string> accessMap = new Dictionary<int, string>();
        pathNameSymbol = (StaticSymbolWindows)
          ConstantExpression.Value(AssemblyCodeGenerator.PathName,
                                   Type.StringType, @"C:\D\Main.com");
```

```
      m_globalMap.Add(AssemblyCodeGenerator.PathName, pathNameSymbol);
    }
```

We look up the main function symbol and report an error if it is not present. If it is present, we add it after the initiation symbol and potential command line symbol. Then we call **GenerateTrace**, which traces all symbols reachable from the main function, and adds them to **m_globalList** and **m_addressMap**.

```
    StaticSymbolWindows mainInfo;
    Assert.Error(m_globalMap.TryGetValue("main", out mainInfo),
                 "non-static main", Message.Function_missing);
    GenerateTrace(mainInfo);
```

In case of command line arguments, we add the symbol for the executable file path name to **m_globalList** and **m_addressMap**.

```
    if (pathNameSymbol != null) {
      Assert.ErrorXXX(!m_globalList.Contains(pathNameSymbol));
      m_globalList.Add(pathNameSymbol);
      m_addressMap.Add(pathNameSymbol.UniqueName, m_totalSize);
      m_totalSize += (int) pathNameSymbol.ByteList.Count;
    }
```

Finally, we add the stack top name, which will be the address of the activation record of the **main** function.

```
    m_addressMap.Add(StackStart, m_totalSize);
```

We iterate through the global list and, for each of its symbols, we call **GenerateAccess**, **GenerateCall**, **GenerateReturn**, which replace the names of accessed static symbols and called functions as well as return addresses with proper addresses.

```
    foreach (StaticSymbolWindows staticSymbol in m_globalList) {
      List<byte> byteList = staticSymbol.ByteList;
      int startAddress = m_addressMap[staticSymbol.UniqueName];
      GenerateAccess(staticSymbol.AccessMap, byteList);
      GenerateCall(startAddress, staticSymbol.CallMap, byteList);
      GenerateReturn(startAddress, staticSymbol.ReturnSet, byteList);
    }
```

Finally, we write the byte list of each symbol in the global list to the target file.

```
    { Console.Out.WriteLine("Generating \"" + targetFile.FullName + "\".");
      targetFile.Delete();
      BinaryWriter targetStream =
        new BinaryWriter(File.OpenWrite(targetFile.FullName));

      foreach (StaticSymbolWindows staticSymbol in m_globalList) {
        foreach (sbyte b in staticSymbol.ByteList) {
          targetStream.Write(b);
        }
      }

      targetStream.Close();
    }
  }
```

The **GenerateTrace** method adds the symbol to the **m_globalList** iterates through the names of access name set and function call name set and calls **GenerateTrace** recursively for each symbol not yet present in the **m_globalList**.

```
  private void GenerateTrace(StaticSymbolWindows staticSymbol) {
      if (!m_globalList.Contains(staticSymbol)) {
        m_globalList.Add(staticSymbol);
        m_addressMap.Add(staticSymbol.UniqueName, m_totalSize);
        m_totalSize += (int) staticSymbol.ByteList.Count;

        ISet<string> accessNameSet =
          new HashSet<string>(staticSymbol.AccessMap.Values);
        foreach (string accessName in accessNameSet) {
          StaticSymbolWindows accessSymbol;
          Assert.Error(m_globalMap.TryGetValue(accessName, out accessSymbol),
                       accessName, Message.Object_missing_in_linking);
          Assert.ErrorXXX(accessSymbol != null);
          GenerateTrace(accessSymbol);
        }

        ISet<string> callNameSet =
          new HashSet<string>(staticSymbol.CallMap.Values);
        foreach (string callName in callNameSet) {
          StaticSymbolWindows funcSymbol;
          Assert.Error(m_globalMap.TryGetValue(callName, out funcSymbol),
                       callName, Message.Function_missing_in_linking);
          Assert.Error(funcSymbol != null, SimpleName(callName),
                         Message.Missing_external_function);
          GenerateTrace(funcSymbol);
        }
      }
    }
```

The **GenerateAccess** methods iterators through the access map and modifies the addresses. For each element, reachable from **main**, we need to modify its static accesses, function calls and return assignments (for a global variable the call map and return set are empty) by calling **GenerateAccess**, **GenerateCall**, and **GenerateReturn**.

```
    private void GenerateAccess(IDictionary<int,string> accessMap,
                                List<byte> byteList) {
      foreach (KeyValuePair<int,string> entry in accessMap) {
        int sourceAddress = entry.Key;
        string name = entry.Value;
```

We obtain the target address from the low and high byte of the source address. Note that the target address does not need to be zero, it may hold an offset. For instance, the target address may be a pointer to a value in an array of a non-zero index, in which case its offset is non-zero.

```
        byte lowByte = byteList[sourceAddress],
             highByte = byteList[sourceAddress + 1];
        int targetAddress = ((int) highByte << 8) + lowByte;
```

We modify the target address by adding the address of the name to target address, which we restore to the source address.

```
        targetAddress += m_addressMap[name];
        byteList[sourceAddress] = (byte) targetAddress;
        byteList[sourceAddress + 1] = (byte) (targetAddress >> 8);
      }
    }
```

The **GenerateCall** method iterates through the names of the functions to called. There are two differences between this method and **GenerateAccess** above. First, we do not need to take inspect the original address of the function. It is always zero since it is not possible to add an offset to a function call. Second, the address of the call is not absolute, it is relative the next assembly code instruction.

```
private void GenerateCall(int startAddress,
                          IDictionary<int, string> callMap,
                          List<byte> byteList) {
  const byte NopOperator = -112 + 256;
  const byte ShortJumpOperator = -21 + 256;

  foreach (KeyValuePair<int,string> entry in callMap) {
    int address = entry.Key;
```

The caller address is the last two bytes of the function assembly code instruction, why the address of the next instruction is the source address plus two. The callee address we simply look up in the **m_addressMap** map. The relative address of the call is the difference between the addresses of the callee and caller function.

```
    int callerAddress = startAddress + address + 2;
    int calleeAddress = m_addressMap[entry.Value];
    int relativeAddress = calleeAddress - callerAddress;
```

If the relative address is more or equals to -128, we change the call to a short jump.

```
    if (relativeAddress >= -128) {
      byteList[address - 1] = (byte) NopOperator;
      byteList[address] = (byte) ShortJumpOperator;
      byteList[address + 1] = (byte) relativeAddress;
    }
```

If the relative address is exactly -129, we have a special case. We change the call to a short jump, but move the call one byte backwards.

```
    else if (relativeAddress == -129) {
      byteList[address - 1] = (byte) ShortJumpOperator;
      byteList[address] = (byte) (-128 + 256);
      byteList[address + 1] = (byte) NopOperator;
    }
```

Otherwise, we just set the address of the call.

```
    else {
      byteList[address] = (byte) ((sbyte) relativeAddress);
      byteList[address + 1] = (byte) ((sbyte) (relativeAddress >> 8));
    }
  }
}
```

The **GenerateReturn** method changes the return address from the address relative the beginning of the function to the address relative the beginning of the executable code. Similar to the **GenerateAccess** case above, we need to inspect the original address, and add the address of the beginning function.

```
private void GenerateReturn(int functionStartAddress, ISet<int> returnSet,
                            List<byte> byteList) {
  foreach (int sourceAddress in returnSet) {
    int lowByte = byteList[sourceAddress],
        highByte = byteList[sourceAddress + 1];
    int targetAddress = (highByte << 8) + lowByte;
    targetAddress += functionStartAddress;
    byteList[sourceAddress] = (byte) targetAddress;
    byteList[sourceAddress + 1] = (byte) (targetAddress >> 8);
```

```
      }
    }
```

The **SimpleName** method returns the part to the left of the leftmost separator identity characters (‘$’), if present. Otherwise, the whole string is returned.

```
    public static string SimpleName(string name) {
      int index = name.LastIndexOf(Symbol.SeparatorId);
      return (index != -1) ? name.Substring(0, index) : name;
    }
  }
}
```

14. The Standard Library

In the **limits**, **setjmp**, **stdio**, **stdlib**, and **time** standard libraries there is different C code for the Linux and Windows environment. We use the **__LINUX__** and **__WINDOWS__** macros to distinguish between them.

14.1. Integral and Floating Limits

The file **limits.h** holds limits for integral values for the Linux and the Windows environment.

limits.h

```
#ifndef __LIMITS_H__
#define __LIMITS_H__

#ifdef __WINDOWS__
  #define CHAR_BIT 8

  #define CHAR_MIN -128S
  #define CHAR_MAX 127S
  #define UCHAR_MAX 255US

  #define SHRT_MIN -128S
  #define SHRT_MAX 127S
  #define USHRT_MAX 255US

  #define INT_MIN -32768
  #define INT_MAX 32767
  #define UINT_MAX 65535U

  #define LONG_MIN -2147483648L
  #define LONG_MAX 2147483647L
  #define ULONG_MAX 4294967295UL
#endif

#ifdef __LINUX__
  #define CHAR_BIT 8

  #define CHAR_MIN -128S
  #define CHAR_MAX 127S
  #define UCHAR_MAX 255US

  #define SHRT_MIN -32768S
  #define SHRT_MAX 32767S
  #define USHRT_MAX 65535US

  #define INT_MIN -2147483648
  #define INT_MAX 2147483647
  #define UINT_MAX 4294967295U

  #define LONG_MIN -9223372036854775808L
  #define LONG_MAX 9223372036854775807L
  #define ULONG_MAX 0xFFFFFFFFFFFFFFFFUL
  #define ULONG_MAX 18446744073709551615UL
```

```
#endif

#endif
```

The file **float.h** holds limits for floating-point values. Note that there is no difference between the Linux and the Windows environment.

float.h
```
#ifndef __FLOAT_H__
#define __FLOAT_H__

#define FLT_ROUNDS 1
#define FLT_RADIX  2

#define FLT_MANT_DIG 2
#define DBL_MANT_DIG 2
#define LDBL_MANT_DIG 2

#define FLT_DIG      6
#define FLT_EPSILON  1e-6
#define FLT_MIN_EXP  -38
#define FLT_MIN      1.2E-38
#define FLT_MAX_EXP  38
#define FLT_MAX      3.4E+38

#define DBL_DIG      6
#define DBL_EPSILON  1e-6
#define DBL_MANT_DIG 2
#define DBL_MIN_EXP  -308
#define DBL_MIN      2.3E-308
#define DBL_MAX_EXP  308
#define DBL_MAX      1.7E+308

#endif
```

14.2. The Assert Macro

The assert macro aborts the execution with an error message if the expression is false. We use the **__FILE__** and **__LINE__** predefined macros to obtain the current file name and line number. The user can explicitly set the **NDEBUG** macro to shorten the definition of the **assert** macro.

assert.h
```
#ifndef __ASSERT_H__
#define __ASSERT_H__

#ifndef NDEBUG
  #include <stdio.h>
  #include <stdlib.h>
  #define assert(expression) if (!(expression)) { \
    fprintf(stderr, "Assertion failed: \"%s\" in file %s at line %i\n", \
    #expression, __FILE__, __LINE__); abort(); }
#else
  #define assert(expression)
#endif

#endif
```

14.3. Locale Data

The file locale library holds functions for locale data.

locale.h
```
#ifndef __LOCALE_H__
#define __LOCALE_H__

#define LC_COLLATE  0x01
#define LC_CTYPE    0x02
#define LC_MONETARY 0x04
#define LC_NUMERIC  0x08
#define LC_TIME     0x10
#define LC_ALL      0x1F
```

The **lconv** struct holds the information of the locale environment. The **summerTimeZone** and **winter-TimeZone** holds the number of hours beyond the Greenwich time for the local summer and winter time zones. To decide whether there actually is winter or summer time at the current time of the year, we need the functions of the **time** standard library.

```
struct lconv {
  int summerTimeZone, winterTimeZone;
```

The **shortDayList**, **longDaysList**, **shortMonthList**, and **longMonthList** fields hold the names of the days of the week and the names of the months of the year, abbreviated and in full text.

```
  char **shortDayList;
  char **longDayList;
  char **shortMonthList;
  char **longMonthList;
```

The **lowerCase** and **upperCase** fields hold the alphabet of the current language.

```
  char *lowerCase;
  char *upperCase;
```

The **messageList** field holds the error message on the current language.

```
  char **messageList;
};
```

The error message enumeration constants are assigned the values from zero and upwards, which corresponds to the indexes of the **enMessageList** array in the **locale.c** file.

```
extern enum {NO_ERROR, NO_FUNCTION, NO_FILE, NO_PATH, NO_HANDLE, NO_ACCESS,
             EDOM, ERANGE, EILSEQ, FOPEN, FFLUSH, FCLOSE, NO_MODE, FWRITE,
             FREAD, FSEEK, FTELL, FSIZE, FREMOVE,FRENAME, FTEMPNAME,FTEMPFILE};

extern char* setlocale(int flag, char* name);
extern struct lconv *localeconv(void);

#endif
```

locale.c
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
```

We initialize the **enShortDayList**, **enLongDayList**, **enShortMonthList**, and **enLongMonthList** with the English names of the days of the week and the months of the year.

```
static char* enShortDayList[] = {"Sun", "Mon", "Tue", "Wed",
                                 "Thu", "Fri", "Sat"};
static char* enLongDayList[] ={"Sunday", "Monday", "Tuesday", "Wednesday",
                               "Thursday", "Friday", "Saturday"};

static char* enShortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
static char* enLongMonthList[] = {"January", "February", "March", "April",
                               "May", "June", "July", "August",
                             "September", "October", "November", "December"};
```

The indexes of the error messages of the **enMessageList** array correspond to the enumeration values of the **locale.h** file.

```
char* enMessageList[] = {"no error", "function not found",
                         "file not found", "path not found",
                         "no handle available", "access denied",
                         "out of domain", "out of range",
                         "invalid multibyte sequence", "error while opening",
                         "error while flushing", "error while closing",
                         "open mode invalid", "error while writing",
                         "error while reading", "error while seeking",
                         "error while telling", "error while sizing",
                         "error while removing file",
                         "error while renaming file" };
```

We initialize the **en_US_utf8** structure with the information of the English language and the time setting of the American east coast.

```
static struct lconv en_US_utf8 = {-5, -4, enShortDayList, enLongDayList,
                                  enShortMonthList, enLongMonthList,
                                  "abcdefghijklmnopqrstuvwxyz",
                                  "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                                  enMessageList};
static const struct _s {
  char* name;
  struct lconv* localePtr;
} g_sArray[] = {{"", &en_US_utf8}, {"C", &en_US_utf8}, {"US", &en_US_utf8}};
```

The **g_sSize** constant holds the size of the **g_sArray** array, which we calculate by dividing the total size of the array by the size of its first value.

```
static const int g_sSize = (sizeof g_sArray) / (sizeof g_sArray[0]);
```

We let the static pointer **g_currStructPtr** point at the first value of **g_sArray**, we assume that there is always at least one value in the array.

```
static struct _s* g_currStructPtr = &g_sArray[0];
```

The **setLocale** function look up the locale settings with the given name, which is later returned by **localeconv** below. The function returns the name of the previous name settings, or null if there is no settings.

```
char* setlocale(int /*flag*/, char* newName) {
  int index;
  char *oldName = (g_currStructPtr != NULL) ? g_currStructPtr->name : NULL;
  g_currStructPtr = NULL;

  if (newName != NULL) {
    for (index = 0; index < g_sSize; ++index) {
      if (strcmp(newName, g_sArray[index].name) == 0) {
        g_currStructPtr = &g_sArray[index];
        break;
      }
    }
  }

  return oldName;
}
```

The **localeconv** function returns the current locale settings.

```
struct lconv* localeconv(void) {
  return (g_currStructPtr != NULL) ? g_currStructPtr->localePtr : NULL;
}
```

14.4. Character Types

The **ctype.h** and **ctype.c** files hold a set functions that test different character conditions.

ctype.h

```
#ifndef __CTYPE_H__
#define __CTYPE_H__

extern int islower(int c);
extern int isupper(int c);
extern int isalpha(int c);
extern int isdigit(int c);
extern int isalnum(int c);
extern int isxdigit(int c);
extern int isgraph(int c);
extern int isprint(int c);
extern int ispunct(int c);
extern int iscntrl(int c);
extern int isspace(int c);
extern int tolower(int c);
extern int toupper(int c);

#endif
```

ctype.c

```
#include <ctype.h>
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stddef.h>
```

The **islower** function tests whether the character is a lowercase character. If there are locale settings defined (see Section 14.3) available, we use it. Otherwise, we regard a character to be lowercase if its ASCII value is between the ASCII values for the ‘a’ and ‘z’ characters, inclusive.

```
int islower(int c) {
  struct lconv* localeConvPtr = localeconv();
  if (localeConvPtr != NULL) {
    return (strchr(localeConvPtr->lowerCase, c) != NULL);
  }
  else {
    return ((c >= 'a') && (c <= 'z'));
  }
}
```

The **isupper** function tests whether the character is an uppercase character in the same way as **islower** above. If there are locale settings available, we use it. Otherwise, we regard a character to be lowercase if its ASCII value is between the ASCII values for the 'A' and 'Z' character, inclusive.

```
int isupper(int c) {
  struct lconv* localeConvPtr = localeconv();

  if (localeConvPtr != NULL) {
    return (strchr(localeConvPtr->upperCase, c) != NULL);
  }
  else {
    return ((c >= 'A') && (c <= 'Z'));
  }
}
```

The **isalpha** function tests whether the character is a letter; that is, a lowercase or an uppercase character.

```
int isalpha(int c) {
  return islower(c) || isupper(c);
}
```

The **isdigit** function tests whether the character is a digit. In this case, we do not use a potential locale convention; we just test whether the ASCII value for the character is between the ASCII values of the '0' and '9' characters, inclusive. Note that we test against the ASCII value of the '0' character, which is not the same as the ASCIII value of the zero-character ('\0'), which is zero.

```
int isdigit(int c) {
  return (c >= '0') && (c <= '9');
}
```

The **isalnum** function tests whether the character is an alphanumerical character; that is, a lowercase or an uppercase character, or a digit.

```
int isalnum(int c) {
  return isalpha(c) || isdigit(c);
}
```

The **isxdigit** function tests whether the character is a hexadecimal digit. Both lowercase and uppercase letters are allowed.

```
int isxdigit(int c) {
  return isdigit(c) || ((c >= 'a') && (c <= 'f'))
                    || ((c >= 'A') && (c <= 'F'));
}
```

The **isgraph** function tests whether the character is a graphical character; that is, a character visible when printed (including space).

```
int isgraph(int c) {
  return (c >= 32) && (c <= 126);
}
```

The **isprint** function tests whether the character is a printable character; that is, a character visible when printed (excluding space).

```
int isprint(int c) {
  return isgraph(c) && (c != ' ');
}
```

The **ispunct** function tests whether the character is a punctation character; that is, a character that is graphical but no alphanumerical, resulting is all visible characters that are not letters or digits.

```
int ispunct(int c) {
  return isgraph(c) && !isalnum(c);
}
```

The **iscntrl** function tests whether the character is a control character, which is all non-printable characters. Note that space is a control character.

```
int iscntrl(int c) {
  return !isprint(c);
}
```

The **isspace** function tests whether the character is a white-space character; that is, a space, form feed, new line, return, horizontal tabulator, or vertical tabulator.

```
int isspace(int c) {
  return (c == ' ') || (c == '\f') || (c == '\n') ||
         (c == '\r') || (c == '\t') || (c == '\v');
}
```

The **tolower** function returns the character converted to uppercase if it is a lowercase character. We use a locale conversion, is available, to locate the lowercase and uppercase character strings.

```
int tolower(int c) {
  if (isupper(c)) {
    struct lconv* localeConvPtr = localeconv();
```

We obtain the strings of lowercase and uppercase characters from the locale conversion, look up the index of the character by in the uppercase string by calling **strchr**, and return the lowercase character at the same index in the lowercase string.

```
    if (localeConvPtr != NULL) {
      char *lowerCase = localeConvPtr->lowerCase,
           *upperCase = localeConvPtr->upperCase;
      int index = (strchr(upperCase, c) - upperCase);
      return ((int) lowerCase[index]);
    }
```

If there is no locale convention available, we return the character with ASCII value plus 32, since the lowercase value are located on values 32 above the uppercase values in the ASCII table.

```
    else {
      return (c + 32);
    }
  }
```

If the character is not an uppercase character, we simply return it.

```
  else {
    return c;
  }
```

```
}
```

The **toupper** function returns the character converted to lowercase if it is an uppercase character in the same way as **tolower** above. Again, we use a locale conversion, is available, to locate the lowercase and uppercase character string, look up the index of character in the lowercase string and return the character at the same index in the uppercase string.

```
int toupper(int c) {
  if (islower(c)) {
    struct lconv* localeConvPtr = localeconv();

    if (localeConvPtr != NULL) {
      char *lowerCase = localeConvPtr->lowerCase,
           *upperCase = localeConvPtr->upperCase;
      int index = (strchr(lowerCase, c) - lowerCase);
      return ((int) upperCase[index]);
    }
```

If there is no locale convention available, we return the character with ASCII value minus 32, since the uppercase value are located on values 32 below the lowercase values in the ASCII table.

```
    else {
      return (c - 32);
    }
  }
```

If the character is not a lowercase character, we simply return it.

```
  else {
    return c;
  }
}
```

14.5. Strings

The **string** standard library holds function for searching and modifying strings.

string.h

```
#ifndef __STRING_H__
#define __STRING_H__

#define size_t int

extern char* strcpy(char* target, const char* source);
extern char* strncpy(char* target, const char* source, size_t size);
extern char* strcat(char* target, const char* source);
extern char* strncat(char* target, const char* source, size_t size);
extern int strcmp(const char* left, const char* right);
extern int strncmp(const char* left, const char* right, size_t size);
extern char* strchr(const char* text, int i);
extern char* strrchr(const char* text, int i);
extern size_t strspn(const char* mainString, const char* charSet);
extern size_t strcspn(const char* mainString, const char* charSet);
extern char* strpbrk(const char* mainString, const char* charSet);
extern char* strstr(const char* mainString, const char* subString);
extern size_t strlen(const char* string);
extern char* strerror(int error);
extern char* strtok(char* string, const char* charSet);
extern char* memcpy(char* target, const char* source, size_t size);
extern char* memmove(char* target, const char* source, size_t size);
extern int memcmp(const char* left, const char* right, size_t size);
extern char* memchr(const char* block, int i, size_t size);
extern char* memset(char* block, int i, size_t size);

#endif
```

string.c

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <locale.h>
```

The **strlen** function returns the number of characters of the string, not counting the terminating zero-character.

```
size_t strlen(const char* string) {
  int index;
```

We simply iterate through the string until we reach the terminating zero-character.

```
  for (index = 0; string[index] != '\0'; ++index) {
    // Empty.
  }

  return index;
}
```

14.5.1. String Copying

The **strcpy** function copies the text of the **source** parameter to the **target** parameter.

```
char* strcpy(char* target, const char* source) {
  int index;
```

We iterate through the source text and copies the characters, until we reach the terminating zero character ('\0').

```
  for (index = 0; source[index] != '\0'; ++index) {
    target[index] = source[index];
  }
```

We add the terminating zero-character to the target string, and return it.

```
  target[index] = '\0';
  return target;
}
```

The **strcpy** function copies the text of the **source** parameter to the **target** parameter, with **size** as the maximum number of copied characters.

```
char* strncpy(char* target, const char* source, size_t size) {
  int index;

  for (index = 0; (index < size) && (source[index] != '\0'); ++index) {
    target[index] = source[index];
  }
```

We add the terminating zero character only if there is room in the target string; that is, if we have not already copied **size** characters.

```
  if (index < size) {
    target[index] = '\0';
  }

  return target;
}
```

14.5.2. String Concatenation

The **strcat** function adds the source string to the target string.

```
char* strcat(char* target, const char* source) {
  int index, targetLength = strlen(target);

  for (index = 0; source[index] != '\0'; ++index) {
    target[targetLength + index] = source[index];
  }
```

We add the terminating zero-character at the end of the target string.

```
  target[targetLength + index] = '\0';
  return target;
}
```

The **strncat** function copies the text of the **source** parameter to the **target** parameter, with **size** as the maximum number of copied characters.

```
char* strncat(char* target, const char* source, size_t size) {
  int index;
  const int targetLength = strlen(target);

  for (index = 0; (index < size) && (source[index] != '\0'); ++index) {
    target[targetLength + index] = source[index];
  }

  if ((targetLength + index) < size) {
    target[targetLength + index] = '\0';
  }

  return target;
}
```

14.5.3. String Comparation

The **strcmp** function compares the left and right string. It returns minus oner if the left string is smaller than the right string, zero if they are equal, and one if the right string is smaller than the left string.

```
int strcmp(const char* left, const char* right) {
  int index;
```

The only way the regard the two string as equal is if they both reach the terminating zero-character at the same time.

```
  for (index = 0; TRUE; ++index) {
    if ((left[index] == '\0') && (right[index] == '\0')) {
      return 0;
    }
```

If one of the strings reaches the zero-character first, it is the smaller string. Technically, we could omit these two **if** statements, since the zero character has ASCII value zero and thereby less than all other characters.

```
    else if (left[index] == '\0') {
      return -1;
    }
    else if (right[index] == '\0') {
      return 1;
    }
```

Until one or both strings has reached the zero-character, we compare the current characters. If the left character is smaller the right character, the left string is smaller. If the right character is smaller the left character, the right string is smaller. When comparing two characters, their ASCII values are compared.

```
    else if (left[index] < right[index]) {
      return -1;
    }
    else if (left[index] > right[index]) {
      return 1;
    }
```

When we have reached this point, where we have not yet reached the end of either of the string, and they have so far been equal, we just iterate afresh and compare the next characters of the strings.

```
  }
}
```

The **strncmp** function compares the left and right string as **strcmp** above. The difference is that it compares up until size number of characters. If the first **size** number of characters are equal, the strings are regarded equal.

```
int strncmp(const char* left, const char* right, size_t size) {
  int index;

  for (index = 0; index < size; ++index) {
    if ((left[index] == '\0') && (right[index] == '\0')) {
      return 0;
    }
    else if (left[index] == '\0') {
      return -1;
    }
    else if (right[index] == '\0') {
      return 1;
    }
    else if (left[index] < right[index]) {
      return -1;
    }
    else if (left[index] > right[index]) {
      return 1;
    }
  }
```

If we have iterated size number of times without finding any difference between the characters, the strings are regarded equal, and we return zero.

```
  return 0;
}
```

14.5.4. String Searching

The **strchr** function looks up the character and returns the address its first occurrence in the string if it is present, and null if it is not present.

```
char* strchr(const char* text, int i) {
  int index;
  char c = (char) i;

  for (index = 0; text[index] != '\0'; ++index) {
    if (text[index] == c) {
      return &text[index];
    }
  }

  return NULL;
}
```

The **strrchr** function looks up the character and returns the address its last occurrence in the string if it is present, and null if it is not present.

```
char* strrchr(const char* text, int i) {
  int index;
  char* result = NULL;
  char c = (char) i;
```

Note that we do not return any value inside the loop in this function. Instead we assign **result** the address of each occurrence. In this way, result will eventually be given the address of the last occurrence of the character, if it is present in the string.

```
  for (index = 0; text[index] != '\0'; ++index)  {
    if (text[index] == c) {
      result = &text[index];
    }
  }
```

As result was initialized by null, null is return if there was no occurrence of the character in the string.

```
  return result;
}
```

The **strspn** function returns number of initial characters of the main string made up of the characters of the character set.

```
size_t strspn(const char* mainString, const char* charSet) {
  int index;

  for (index = 0; mainString[index] != '\0'; ++index) {
    if (strchr(charSet, mainString[index]) == NULL) {
      return index;
    }
  }

  return -1;
}
```

The **strcspn** function return the index in the string of the first occurrence of one of the characters not present in the character set, or minus one if there is no such occurrence.

```
size_t strcspn(const char* mainString, const char* charSet) {
  int index;

  for (index = 0; mainString[index] != '\0'; ++index) {
    if (strchr(charSet, mainString[index]) != NULL) {
      return index;
    }
  }

  return -1;
}
```

The **strpbrk** function

```
char* strpbrk(const char* mainString, const char* charSet) {
  int index;

  for (index = 0; mainString[index] != '\0'; ++index) {
    if (strchr(charSet, mainString[index]) != NULL) {
      return &mainString[index];
    }
  }

  return NULL;
}
```

The **strstr** function returns the index of the first occurrence of the substring in the main string, or null if there is no occurrence.

```
char* strstr(const char* mainString, const char* subString) {
  int index;
  const int subStringSize = strlen(subString);
```

We iterate through the main string and call **strncmp** for each index to test whether the sub stirng is indeed a sub string of the main string.

```
  for (index = 0; mainString[index] != '\0'; ++index) {
    if (strncmp(mainString + index, subString, subStringSize) == 0) {
      return &mainString[index];
    }
  }

  return NULL;
}
```

14.5.5. Error Messages

The **strerror** function return the error message corresponding to the error value in the local settings if there are locale settings with an error message list, otherwise null.

```
char* strerror(int errno) {
  struct lconv* localeConvPtr = localeconv();

  if (localeConvPtr != NULL) {
    char** messageList = localeConvPtr->messageList;

    if (messageList != NULL) {
      return messageList[errno];
    }
  }

  return NULL;
}
```

14.5.6. Tokenization

The **strtok** function divide the string in tokens, separated by any of the characters in character set. The static global variable token holds the address of the first token of the string.

```
static char* token = NULL;

char* strtok(char* string, const char* charSet) {
  int index;
  char* tokenStart;

  if (string != NULL) {
    if (string[0] == '\0') {
      return NULL;
    }

    for (index = 0; string[index] != '\0'; ++index) {
      if (strchr(charSet, string[index]) != NULL) {
        string[index] = '\0';
        token = &string[index + 1];
        return string;
      }
    }

    token = &string[index];
    return string;
  }
  else if (token == NULL) {
    return NULL;
  }
  else {
    if (token[0] == '\0') {
      return NULL;
    }

    for (index = 0; token[index] != '\0'; ++index) {
      if (strchr(charSet, token[index]) != NULL) {
        char* tokenStart2 = token;
        token[index] = '\0';
        token = &token[index + 1];
        return tokenStart2;
      }
    }

    tokenStart = token;
    token = &token[index];
    return tokenStart;
  }
}
```

14.5.7. Memory Functions

Finally, we have the three memory functions **memcpy**, **memmove**, **memcmp**, **memchr**, and **memset**. Basically, the do the same thing as their string function counterparts. However, there is no terminating zero-character. Instead, there is always a size in bytes. The **memcpy** function copies a memory block of a certain size (in bytes) from a source address to a target address. However, it takes no consideration in whether the memory blocks overlaps.

```
void* memcpy(void* target, const void* source, size_t size) {
  char* charTarget = (char*) target;
  const char* charSource = (const char*) source;

  int index;
  for (index = 0; index < size; ++index) {
    charTarget[index] = charSource[index];
  }

  return ((void*) target);
}
```

Similar to the **memcpy** function above, **memmove** copies a memory block of a certain size (in bytes) from a source address to a target address. The difference is that **memmove** takes consideration in whether the memory blocks overlaps.

```
void* memmove(void* target, const void* source, size_t size) {
  char* charTarget = (char*) target;
  const char* charSource = (const char*) source;
```

If the source address is lower than the target address, we copy the bytes in reversed order to make sure the bytes are copied in valid order in case of overlapping.

```
  int index;
  if (source < target) {
    for (index = (size - 1); index >= 0; --index) {
      charTarget[index] = charSource[index];
    }
  }
  else {
    for (index = 0; index < size; ++index) {
      charTarget[index] = charSource[index];
    }
  }

  return ((void*) target);
}
```

The **memcmp** function compares the two memory block, and return minus one if the left block is the smaller one, zero if they are equal, and plus one if the right block is the smaller one.

```
int memcmp(const void* left, const void* right, size_t size) {
  const char* charLeft = (const char*) left;
  const char* charRight = (const char*) right;

  int index;
  for (index = 0; index < size; ++index) {
    if (charLeft[index] < charRight[index]) {
      return -1;
    }
    else if (charLeft[index] > charRight[index]) {
      return 1;
    }
  }

  return 0;
}
```

The **memchr** function searches the memory block after a byte and returns the address of the first occurrence if it is present in the block, otherwise null.

```
void* memchr(const void* block, int i, size_t size) {
  int index;
  const char* charBlock = (const char*) block;
  char c = (char) i;

  for (index = 0; index < size; ++index) {
    if (charBlock[index] == c) {
      return (void*) &charBlock[index];
    }
  }

  return NULL;
}
```

The **memset** function sets the memory block with the value.

```
void* memset(void* block, int i, size_t size) {
  char* charBlock = (char*) block;
  char c = (char) i;

  int index;
  for (index = 0; index < size; ++index) {
    charBlock[index] = c;
  }

  return block;
}
```

14.6. Long Jumps

The **setjmp** standard library holds functions for preparing and performing long jumps; that is, jumps through function call chains.

setjmp.h

```
typedef void* jmp_buf[3];
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int value);
```

setjmp.c

```
#include <setjmp.h>
```

The **setjmp** function stores the return address, regular frame pointer, and variadic frame pointer in the **buf** static variable.

```
#ifdef __LINUX__
  int setjmp(jmp_buf buf) {
    void** rbp_pointer = register_rbp;
    buf[0] = rbp_pointer[0];
    buf[1] = rbp_pointer[1];
    buf[2] = rbp_pointer[2];
    return 0;
  }
```

The **longjmp** function resets the return address, regular frame pointer, and variadic frame pointer by looking up their values from the **buf** static variable by **setjmp** above. The last line jumps back to the return address of the previous **setjmp** call, with the return value given as a parameter to **longjmp**. In this way the previous **setjmp** call return zero and the following **longjmp** call returns the (presumably non-zero) value of its parameter.

```
  void longjmp(jmp_buf buf, int return_value) {
    register_ebx = return_value;
    register_rcx = buf[0];
    register_rdi = buf[2];
    register_rbp = buf[1];
    jump_register(register_rcx);
  }
#endif
```

The code for the Windows environment is similar to the Linux environment. The difference is that we use 16-bit registers rather than 32-bit and 64-bit registers.

```
#ifdef __WINDOWS__
  int setjmp(jmp_buf buf) {
    void** bp_pointer = register_bp;
    buf[0] = bp_pointer[0];
    buf[1] = bp_pointer[1];
    buf[2] = bp_pointer[2];
    return 0;
  }

  void longjmp(jmp_buf buf, int return_value) {
    register_bx = return_value;
    register_cx = buf[0];
    register_di = buf[2];
    register_bp = buf[1];
    jump_register(register_cx);
  }
#endif
```

The following code demonstrates how **setjmp** and **longjmp** can be used by jumping back through a function call chain.

setjmptest.c

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf buffer;

double inverse(double x);
double divide(double x, double y);

void main() {
  char* message;
  double x;

  printf("Please input a value: ");
  scanf("%lf", &x);

  if ((message = setjmp(buffer)) == 0) {
    printf("1.0 / %f = %f\n", x, inverse(x));
  }
  else {
    printf("%s\n", message);
  }
}

double inverse(double x) {
  return divide(1, x);
}
```

In case of a non-zero denominator, we return the quintet in a normal way and **setjmp** returns zero. In case of a zero denominator, we call **longjmp** with an error message. The address of the error message is then be returned by the call to **setjmp**, instead of zero.

```
double divide(double x, double y) {
  if (y != 0) {
    return x / y;
  }
  else {
    longjmp(buffer, "Division by Zero.");
    return 0;
  }
}
```

14.7. Mathematical Functions

The mathematical functions are implemented completely in C, they do not make system calls.

math.h

```
#ifndef __MATH_H__
#define __MATH_H__

#define PI 3.14159265358979323846264433
#define E  2.71828182845904523536028747

extern double exp(double x);
extern double log(double x);
extern double log10(double x);

extern double pow(double x, double y);
extern double ldexp(double x, int exponent);
extern double frexp(double x, int* exponent);

extern double sqrt(double x);
extern double modf(double x, double* integral);
extern double fmod(double x, double y);

extern double sin(double x);
extern double cos(double x);
extern double tan(double x);
extern double sinh(double x);
extern double cosh(double x);
extern double tanh(double x);

extern double asin(double x);
extern double acos(double x);
extern double atan(double x);
extern double atan2(double x, double y);

extern double floor(double x);
extern double ceil(double x);
extern double round(double x);
extern double fabs(double x);

#endif
```

14.7.1. Exponent and Logarithm Functions

The Sine, Cosine, Exponent, and Logarithm functions are calculated by iterative methods, and the rest of the functions call these functions.

math.c

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
```

The exponent function can be calculated be the following iterative formula for any values of x.

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \sum_{i=1}^{\infty} \frac{x^i}{i!}$$

Naturally, we cannot iterate indefinitely in the code. Instead, we break the iteration when the term is smaller than the EPSILON constant.

```
#define EPSILON   1e-9

double exp(double x) {
  double index = 1, term, sum = 1, faculty = 1, power = x;

  do {
    term = power / faculty;
    sum += term;
    power *= x;
    faculty *= ++index;
  } while (fabs(term) >= EPSILON);

  return sum;
}
```

The logarithm function can be calculated be the following formula for any values $0 < x < 2$, which gives $-1 < (x - 1) < 1$.

$$\ln x = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}(x-1)^i}{i}$$

To begin with, we must make sure that the input value is more than zero and less than two. Moreover, in order to keep the number of iterations down, the value shall not be too close to either zero or two. Therefore, we make sure the value holds the interval $\frac{1}{e} < x < 1$. We can do that by multiply or divide by e, and use the facts that

$$\ln xe^n = \ln x + \ln e^n = \ln x + n \ln e = \ln x + n \cdot 1 = \ln x + n$$

and

$$\ln x \sqrt[n]{e} = \ln x\, e^{-n} = \ln x + \ln e^{-n} = \ln x - \ln e^n = \ln x - n \ln e = \ln x - n \cdot 1 = \ln x - n$$

This gives that we can divide x with e until $x < 1$, and we can multiply x with e until $x > \frac{1}{e}$, as long as we keep track of the number of divisions and multiplications.

```
#define E_INVERSE (1 / E)

double log(double x) {
  if (x > 0) {
    int n = 0;
```

If x is more than zero and not exact one, we test whether x is more than one. If it is, we divide x with e until x is less than one.

```
    if (x > 1) {
      while (x > 1) {
        x /= E;
        ++n;
      }
    }
```

If x is less than $\frac{1}{e}$, we multiple e with e until it is more than more $\frac{1}{e}$. In both cases we keep track of the number of divisions and multiplications by updating n.

```
    else if (x < E_INVERSE) {
      while (x < E_INVERSE) {
        x *= E;
        --n;
      }
    }
```

When we have made sure that $\frac{1}{e} < x < 1$, we iterate with the formula $\ln x = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}(x-1)^i}{i}$.

```
    { double index = 1, term, sum = 0, sign = 1,
             x_minus_1 = x - 1, power = x_minus_1;

      do {
        term = sign * power / index++;
        sum += term;
        power *= x_minus_1;
        sign *= -1.0;
      } while (fabs(term) >= EPSILON);
```

The result is the sum of the iteration plus the number of division and multiplication (n) before the iteration.

```
      return sum + n;
    }
  }
```

If x is less than zero, we report an error by setting **errno** domain error and return zero.

```
  else {
    errno = EDOM;
    return 0;
  }
}
```

The common logarithm, with base 10, is defined as follows:

$$\log_{10} x = \frac{\ln x}{\ln 10}$$

We define a constant value for $\ln 10$, which we divide $\ln x$ with.

```
#define LN_10 2.302585092994045684017991 4

double log10(double x) {
  return log(x) / LN_10;
}
```

14.7.2. Power Functions

The power function x^y is defined for all real values $x > 0$ as follows:

$$x^y = \begin{cases} e^{y \ln x} & if\ x > 0 \\ 1 & if\ x = 0\ and\ y = 0 \\ 0 & if\ x = 0\ and\ y > 0 \end{cases}$$

If $x < 0$ and y is an integer value, then

$$x^y = \begin{cases} e^{y \ln(-x)} & if\ y \text{ is even} \\ -e^{y \ln(-x)} & if\ y \text{ is odd} \end{cases}$$

```
double pow(double x, double y) {
  if (x > 0)  {
    return exp(y * log(x));
  }
  else if ((x == 0) && (y == 0)) {
    return 1;
  }
  else if ((x == 0) && (y > 0)) {
    return 0;
  }
```

We test if y is an integer value by comparing its floor and ceiling values. If they are equal, the value is an integer.

```
  else if ((x < 0) && (floor(y) == ceil(y))) {
    long long_y = (long) y;
```

If the integer value of y modulo two is zero, it is even. Otherwise, it is odd.

```
    if ((long_y % 2) == 0) {
      return exp(y * log(-x));
    }
    else {
      return -exp(y * log(-x));
    }
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The function **ldexp** is defined as ldexp $(x, n) = x \cdot 2^n$.

```
double ldexp(double x, int n) {
  return x * pow(2, n);
}
```

The function **frexp** splits **x** into normalized fraction of **x**. The return value is within the interval from 0.5, inclusive, to 1, exclusive.

```
#define LN_2 0.693147180559945309417232121

static log2(double x) {
  return log(x) / LN_2;
}

double frexp(double x, int* p) {
  if (x != 0)  {
    int exponent = (int) log2(fabs(x));

    if (pow(2, exponent) < x) {
      ++exponent;
    }

    if (p != NULL) {
      *p = exponent;
    }

    return (x / pow(2, exponent));
  }
  else {
    if (p != NULL) {
      *p = 0;
    }

    return 0;
  }
}
```

14.7.1. Square Root

The square root r of x is for all $x > 0$ is iterative defined as:

$$\begin{cases} r_0 = 1 \\ r_{i+1} = \dfrac{r_i + \dfrac{x}{r_i}}{2} \end{cases}$$

```
double sqrt(double x) {
  if (x >= 0) {
    double root_i, root_i_plus_1 = 1;

    do {
      root_i = root_i_plus_1;
      root_i_plus_1 = (root_i + (x / root_i)) / 2;
    } while (fabs(root_i_plus_1 - root_i) >= EPSILON);

    return root_i_plus_1;
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

14.7.1. Modulo Functions

The **modf** function split **x** into a integral and fractional part. The integral is returned, and the fractional part is assigned the value that **p** points at, if it is not null. The integral and fractional values hold the same sign as the original value.

```
double modf(double x, double* p) {
  double abs_x = fabs(x),
  integral = (double) ((long) abs_x),
  fractional = abs_x - integral;

  if (p != NULL)  {
    *p = (x > 0) ? integral : -integral;
  }

  return (x > 0) ? fractional : -fractional;
}
```

The **fmod** function returns the floating-point remainder of **x** / **y**, with the same sign as **x**.

```
double fmod(double x, double y) {
  if (y != 0) {
    double remainder = fabs(x - (y * ((int) (x / y))));
    return (x > 0) ? remainder : -remainder;
  }
```

If **y** is zero, we have division by zero, we report a domain error and return zero.

```
  else {
    errno = EDOM;
    return 0;
  }
}
```

14.7.2. Trigonometric Functions

The sine function is defined for all x as follows:

$$\sin(x) = \sin(x + 2\pi n) \quad n \in \mathbb{Z}$$

$$\sin x = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$$

```
double sin(double x) {
  if (fabs(x) > (2 * PI)) {
    x = fmod(x, 2 * PI);
  }

  { double index = 1, term, sum = 0, sign = 1, power = x, faculty = 1;

    do {
      term = sign * power / faculty;
      sum += term;
      sign *= -1;
      power *= x * x;
      faculty *= ++index * ++index;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The cosine function is defined for all x as follows:

$$\cos(x) = \cos\,(x + 2\pi n) \quad n \in \mathbb{Z}$$

$$\cos(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i}}{(2i)!}$$

```
double cos(double x) {
  if (fabs(x) > (2 * PI)) {
    x = fmod(x, 2 * PI);
  }

  { double index = 0, term, sum = 0, sign = 1, power = 1, faculty = 1;

    do {
      term = sign * power / faculty;
      sum += term;
      sign *= -1;
      power *= x * x;
      faculty *= ++index * ++index;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The tangent function is defined as $\tan x = \frac{\sin x}{\cos x}$ for all x such as $\cos x \neq 0$.

```
double tan(double x) {
  double cos_of_x = cos(x);
  printf("cos(%f) = %f\n", x, cos(x));

  if (cos_of_x != 0) {
    return (sin(x) / cos_of_x);
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

14.7.3. Inverted Trigonometric Functions

The arcsine function is defined for all x such as $|x| \leq 1$ as follows:

$$\operatorname{asin}(1) = \frac{\pi}{2}$$
$$\operatorname{asin}(-x) = -\operatorname{asin}(x)$$
$$\operatorname{asin}(x) = \operatorname{atan}\left(\frac{x}{\sqrt{1-x^2}}\right) \quad if\ |x| < 1$$

```
double asin(double x) {
  if (x == 1) {
    return PI / 2;
  }
  else if (x < 0) {
    return -asin(-x);
  }
  else if (x < 1) {
    return atan(x / sqrt(1 - (x * x)));
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The arccosine function is defined for all x such as $|x| \leq 1$ as follows:

$$\operatorname{acos}(0) = \frac{\pi}{2}$$
$$\operatorname{acos}(-x) = \pi - acos(x)$$
$$\operatorname{acos}(x) = \operatorname{atan}\left(\frac{\sqrt{1-x^2}}{x}\right) \quad if\ 0 < |x| \leq 1$$

```
double acos(double x) {
  if (x == 0) {
    return PI / 2;
  }
  else if (x < 0) {
    return PI - acos(-x);
  }
  else if (x <= 1) {
    return atan(sqrt(1 - (x * x)) / x);
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The arctangent function is defined for all $-1 \leq x \leq 1$ as follows:

$$\operatorname{atan}(x) = -\operatorname{atan}(-x) \quad for\ all\ x$$

$$\operatorname{atan}(x) = \frac{\pi}{2} - \operatorname{atan}\left(\frac{1}{x}\right) \quad if\ x > 0$$

$$\operatorname{atan}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1} \quad if\ 0 \leq x < 1$$

```
double atan(double x) {
  if (x < 0) {
    return -atan(-x);
  }
  else if (x > 1) {
    return PI / 2 - atan(1 / x);
  }
```

The number of iterations of the formula above is very high for values close to one. Therefore, we use the following formula for $\frac{1}{2} < x \leq 1$. The function argument of the right-hand call is always less than or equal to $\frac{1}{2}$, since the numerator is at most one and the denominator is at least two.

$$\operatorname{atan}(x) = 2\operatorname{atan}\left(\frac{x}{1+\sqrt{1+x^2}}\right)$$

```
  else if (x > 0.5) {
    return 2 * atan(x / (1 + sqrt(1 + (x * x))));
  }
```

We have finally reached the iterative formula. There is at most 30 iterations for values close to $1/2$, and less iteration for values closer to zero.

$$\operatorname{atan}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1} \quad if\ 0 \leq x < 1$$

```
  else {
    double term, sum = 0, sign = 1, denominator = 1, product = x;

    do {
      term = sign * product / denominator;
      sum += term;
      sign = -sign;
      product *= x * x;
      denominator += 2;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The second arctangent function return arctangent of $\frac{x}{y}$, if $y \neq 0$.

```
double atan2(double x, double y) {
  if (y > 0) {
    return atan(x / y);
  }
  else if ((x >= 0) && (y < 0))  {
    return PI + atan(x / y);
  }
  else if ((x < 0) && (y < 0)) {
    return (-PI) + atan(x / y);
  }
  else if ((x > 0) && (y == 0))  {
    return PI / 2;
  }
  else if ((x < 0) && (y == 0))  {
    return (-PI) / 2;
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

14.7.4. Hyperbolic Trigonometric Functions

The hyperbolic sine function is defined for all x as follows:

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

```
double sinh(double x) {
  return (exp(x) - exp(-x)) / 2;
}
```

The hyperbolic cosine function is defined as follows:

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

```
double cosh(double x) {
  return (exp(x) + exp(-x)) / 2;
}
```

The hyperbolic tangent function is defined for all x as follows:

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Not that we do not have to check for division by zero in this case, since $\cosh x$ cannot be zero.

```
double tanh(double x) {
  return sinh(x) / cosh(x);
}
```

14.7.5. Floor, Ceiling, Absolute, and Rounding Functions

The **floor** function returns the integer value closer to zero.

```
double floor(double x) {
  if (x < 0) {
    return -ceil(-x);
  }

  return (double) ((long) x);
}

double ceil(double x) {
  if (x < 0) {
    return -floor(-x);
  }
  return (double) ((long) (x + 0.999999999999));
}

double round(double x) {
  return (double) ((long) ((x < 0) ? (x - 0.5) : (x + 0.5)));
}
```

$$|x| = \begin{cases} -x & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

```
double fabs(double x) {
  return (x < 0) ? -x : x;
}
```

14.8. Standard Output

The standard output library is rather large, due to the flexibility of the **printf** function.

stdio.h

```
#ifndef __STDIO_H__
#define __STDIO_H__

#include <math.h>
#include <ctype.h>
#include <stdarg.h>
#include <stddef.h>
#include <file.h>
#include <temp.h>
#include <scanf.h>
#include <printf.h>

#endif
```

printf.h

```
#ifndef __PRINTF_H__
#define __PRINTF_H__
```

The output can be directed to a device (**stdout** or a file) or a string, **g_outStatus** keep track of the case. The **g_charCount** field count the number of characters written, and **g_outDevice** holds the device (**stdout** or a file) to be written to.

```
#define DEVICE 0
#define STRING 1

extern int g_outStatus, g_charCount;
extern void* g_outDevice;

int putc(int c, FILE* stream);
int fputc(int c, FILE* stream);
int putchar(int c);

int printf(const char* format, ...);
int vprintf(const char* format, va_list arg_list);
int fprintf(FILE* outStream, const char* format, ...);
int vfprintf(FILE* outStream, const char* format, va_list arg_list);
int sprintf(char* outString, const char* format, ...);
int vsprintf(char* outString, const char* format, va_list arg_list);
```

```
#endif
```

printf.c

```
#include <math.h>
#include <ctype.h>
#include <stdio.h>
#include <stddef.h>
#include <stdarg.h>
#include <stdlib.h>
#include <scanf.h>
#include <printf.h>

#define DEVICE 0
#define STRING 1
#define BLANK  2

int g_outStatus, g_outChars;
void* g_outDevice;

#define DEFAULT_PRECISION 6
```

14.8.1. Print Character and String

The **putc, fputc**, and **putchar** functions write a character to the stream. They all call **printChar** below.

```
int putc(int i, FILE* stream) {
  g_outStatus = DEVICE;
  g_outDevice = (void*) stream;
  printChar((char) i);
  return 1;
}

int fputc(int i, FILE* stream) {
  g_outStatus = DEVICE;
  g_outDevice = (void*) stream;
  printChar((char) i);
  return 1;
}

int putchar(int i) {
  g_outStatus = DEVICE;
  g_outDevice = (void*) stdout;
  printChar((char) i);
  return 1;
}
```

The **printChar** function write a character to a stream.

```
void printChar(char c) {
  int handle;
  char* outString;

  switch (g_outStatus) {
    case DEVICE: {
        FILE* stream = (FILE*) g_outDevice;
```

In the Linux environment we perform a system call, while we in the Windows enviorment performs an interrupt call.

```
#ifdef __LINUX__
        register_rax = 0x01;
        register_rdi = (unsigned long) stream->handle;
        register_rsi = (unsigned long) &c;
        register_rdx = 1L;
        syscall();
#endif

#ifdef __WINDOWS__
        register_ah = 0x40;
        register_bx = stream->handle;
        register_cx = 1;
        register_dx = &c;
        interrupt(0x21s);
#endif

        ++g_outChars;
        break;
      }
```

If the output device is a string, we add the character to the output string.

```
    case STRING: {
        outString = (char*) g_outDevice;
        outString[g_outChars++] = c;
      }
      break;
```

The status can also be blank in which case we only increase the character count. This option is used when calculating the number of right-hand paddings.

```
    case BLANK:
      g_outChars++;
      break;
  }
}
```

The **printString** function prints a string.

```
void printString(const char* s, int precision) {
  if (s != NULL) {
    int index;
```

If the precision is zero, we continue to print until we reach the zero character.

```
    if (precision == 0) {
      for (index = 0; s[index] != '\0'; ++index) {
        printChar(s[index]);
      }
    }
```

If the precision is zero, we continue to print the number of characters indicated by the precision.

```
    else {
      for (index = 0; (precision-- > 0) && (s[index] != '\0'); ++index) {
        printChar(s[index]);
      }
    }
  }
```

If the string is a null pointer, we write the string “<NULL>”.

```
  else {
    printChar('<');
    printChar('N');
    printChar('U');
    printChar('L');
    printChar('L');
    printChar('>');
  }
}
```

14.8.1. Print Values

The **printLongIntRec** function prints a positive long integer value by calling itself recursively until the value is zero.

```
void printLongIntRec(long longValue) {
  if (longValue != 0) {
```

We extract the digit to be printed by modulo ten, and call **printLongRec** recursively with the value divided by ten.

```
    int digit = (int) (longValue % 10L);
    printLongIntRec(longValue / 10L);
    printChar((char)(digit + '0'));
  }
}
```

The printLongInt print a positive or negative long integer value. If the value is less than zero, a minus is written before the value. If the **space** parameter is true, a space is written before the value. If the **plus** parameter is true, the plus sign is written if the value is non-negative.

```
void printLongInt(long longValue, BOOL plus, BOOL space) {
  if (longValue < 0L) {
    longValue = -longValue;
    printChar('-');
  }
  else if (space) {
    printChar(' ');
  }
  else if (plus) {
    printChar('+');
  }
```

If the value is zero, we simply write zero. Otherwise, we call printLongIntRec to write the actual value. Remember that the value in this case is always positive.

```
  if (longValue == 0L) {
    printChar('0');
  }
  else {
    printLongIntRec(longValue);
  }
}
```

The **digitToChar** function writes an octal, decimal, or hexadecimal digit. A hexadecimal digit is written in uppercase if the **capital** parameter is true.

```
char digitToChar(int digit, BOOL capital) {
  if (digit < 10) {
    return ((char) ('0' + digit));
  }
  else if (capital) {
    return ((char) ('A' + (digit - 10)));
  }
  else {
    return ((char) ('a' + (digit - 10)));
  }
}
```

The **printUnsignedLongRec** function writes an unsigned long integer value where the **base** parameter indicates its base. Potential hexadecimals digits are written in uppercase if **capital** is true.

```
void printUnsignedLongRec(unsigned long unsignedValue,
                          unsigned long base, BOOL capital) {
  if (unsignedValue > 0ul) {
    int digit = (int) (unsignedValue % base);
    printUnsignedLongRec(unsignedValue / base, base, capital);

    { char c = digitToChar(digit, capital);
      printChar(c);
    }
  }
}
```

The **printUnsignedLong** function write the unsigned long integer value by calling **printUnsignedLong-Rec** in the same way as **printSignedLongRec** above. As the value is unsigned we not need to take any potential negative values in consideration.

```
void printUnsignedLong(unsigned long unsignedValue, BOOL plus, BOOL space,
                       BOOL grid, unsigned long base, BOOL capital) {
  if (plus) {
    printChar('+');
  }

  if (space) {
    printChar(' ');
  }

  if (grid) {
    if (base == 8ul) {
      printChar('0');
    }

    if (base == 16ul) {
      printChar('0');
      printChar(capital ? 'X' : 'x');
    }
  }

  if (unsignedValue == 0ul) {
    printChar('0');
  }
  else {
    printUnsignedLongRec(unsignedValue, base, capital);
  }
}
```

The **printLongDoubleFraction** function print the fraction (decimal) part of a double value.

```
void printLongDoubleFraction(long double longDoubleValue,
                             BOOL grid, int precision) {
```

We start by subtracting the value by its integral part.

```
  longDoubleValue -= (long) longDoubleValue;

  if (precision == 0) {
    precision = DEFAULT_PRECISION;
  }
```

If the grid parameter is ture, we also write a dot ('.'). Otherwise, we write a dot if precision is greater than zero; that is, if decimals are to be written.

```
  if (grid || (precision > 0)) {
    printChar('.');
  }
```

We continue to write decimal as long as the precision is greater than zero. We extract the digit to be written by multiplying the fraction with then and then cast it to an integer.

```
  while (precision-- > 0) {
    long double longDoubleValue10 = 10.0L * longDoubleValue;
    int digitValue = (int) longDoubleValue10;
    printChar((char) (digitValue + '0'));
    longDoubleValue = longDoubleValue10 - ((long double) digitValue);
  }
}
```

The **printLongDoublePlain** function writes a double value in plain (non-exponential) form.

```
void printLongDoublePlain(long double longDoubleValue, BOOL plus,
                          BOOL space, BOOL grid, int precision) {
  if (longDoubleValue < 0.0L) {
    printChar('-');
    longDoubleValue = -longDoubleValue;
    plus = FALSE;
    space = FALSE;
  }

  { long longValue = (long) longDoubleValue;
    printLongInt(longValue, plus, space);
    longDoubleValue -= (long double) longValue;
    printLongDoubleFraction(longDoubleValue, grid, precision);
  }
}
```

The **printLongDoubleExpo** function write a double value at exponent form.

```
void printLongDoubleExpo(long double value, BOOL plus, BOOL space,
                         BOOL grid, int precision, BOOL capital) {
```

If the value is zero, we just write zero.

```
  if (value == 0.0L) {
    printChar('0');
    printLongDoubleFraction(0.0L, precision, grid);
    printChar(capital ? 'E' : 'e');
    printChar('0');
  }
  else {
    if (value < 0.0L) {
      printChar('-');
      value = -value;
    }
```

We divide the value by its exponent and write it by calling **printLongDoublePlain**.

```
    { int expo = (int) log10(value);
      value /= pow(10.0, expo);

      printLongDoublePlain(value, plus, space, grid, precision);
      printChar(capital ? 'E' : 'e');
      printLongInt(expo, TRUE, FALSE);
    }
  }
}
```

14.8.2. Print Argument

The **printArgument** function writes an argument of the **printf** functions. The **format** parameter points to the next per cent code in the format string sent to **printf**.

```
va_list printArgument(const char* format, va_list arg_list, BOOL plus,
                      BOOL space, BOOL grid, int* widthPtr, int precision,
                      BOOL shortInt, BOOL longInt, BOOL longDouble, BOOL sign,
                      BOOL* negativePtr) {
  char c = format[0], charValue;
  int *intPtr;
  long double longDoubleValue;
  void* ptrValue;

  switch (c) {
    case 'i':
    case 'd': {
        long longValue;
```

The value to be written may be short integer, integer, or long integer.

```
        if (shortInt) {
          longValue = (long) (short) va_arg(arg_list, int);
        }
        else if (longInt) {
          longValue = va_arg(arg_list, long);
        }
        else {
          longValue = (long) va_arg(arg_list, int);
        }
```

The negative pointer, if not null, is set if the value is less than zero.

```
        if (negativePtr != NULL) {
          *negativePtr = (longValue < 0);
        }

        if (!sign) {
          longValue = labs(longValue);
        }
```

The **checkWidthAndPrecision** calculates the potential characters written before the value.

```
        arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
        printLongInt(longValue, plus, space);
      }
      break;

    case 'c':
      charValue = (char) va_arg(arg_list, int);
      arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
      printChar(charValue);
      break;

    case 's': {
        char* stringValue = va_arg(arg_list, char*);
        arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
        printString(stringValue, precision);
      }
      break;

    case 'u':
    case 'o':
    case 'b':
    case 'x':
    case 'X': {
```

The base of a decimal value may be 2, 8, 10, or 16.

```
        unsigned long base = ((c == 'u') ? 10ul : ((c == 'o') ? 8ul :
                                                  ((c == 'b') ? 2ul : 16ul)));
        unsigned long value;
```

The decimal value to be written may hold the type short integer, integer, or long integer.

```
        if (shortInt) {
          value = (unsigned long) (unsigned short)
                  va_arg(arg_list, unsigned int);
        }
        else if (longInt) {
          value = va_arg(arg_list, unsigned long);
        }
        else {
          value = (unsigned long) va_arg(arg_list, unsigned int);
        }

        arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
        printUnsignedLong(value, plus, space, grid, base, isupper(c));
      }
      break;

    case 'f':
    case 'e':
    case 'E':
```

```
    case 'g':
    case 'G':
```

The type of a floating value may be double or long double. Note that all arguments of type float are casted to double in variadic function calls.

```
      if (longDouble) {
        longDoubleValue = va_arg(arg_list, long double);
        printLongDoublePlain(longDoubleValue, FALSE, FALSE, FALSE, 3);
      }
      else {
        longDoubleValue = (long double) va_arg(arg_list, double);
      }

      if (negativePtr != NULL) {
        *negativePtr = (longDoubleValue < 0);
      }

      if (!sign) {
        longDoubleValue = fabs(longDoubleValue);
      }

      arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);

      if (c == 'f') {
        printLongDoublePlain(longDoubleValue, plus, space, grid, precision);
      }
      else if (tolower(c) == 'e') {
        printLongDoubleExpo(longDoubleValue, plus, space,
                            grid, precision, isupper(c));
      }
      else {
        int expo = (int) log10(fabs(longDoubleValue));

        if ((expo >= -3) && (expo <= 2)) {
          printLongDoublePlain(longDoubleValue, plus, space, grid, precision);
        }
        else {
          printLongDoubleExpo(longDoubleValue, plus, space,
                              grid, precision, isupper(c));
        }
      }
      break;

    case 'p':
      ptrValue = va_arg(arg_list, void*);
      arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
      printUnsignedLong((void*) ptrValue, FALSE, FALSE, FALSE, 10u, FALSE);
      break;
```

The **n** code returns the number of characters written.

```
    case 'n':
      ptrValue = va_arg(arg_list, void*);
      intPtr = va_arg(arg_list, int*);
      arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
      *intPtr = g_outChars;
      break;
    case '%':
      arg_list = checkWidthAndPrecision(arg_list, widthPtr, &precision);
      printChar('%');
      break;
  }

  return arg_list;
}
```

The **checkWidthAndPrecision** reads the width and precision pointers from the argument list.

```
va_list checkWidthAndPrecision(va_list arg_list, int* widthPtr,
                               int* precisionPtr) {
  if ((widthPtr != NULL) && (*widthPtr == -1)) {
    *widthPtr = va_arg(arg_list, int);
  }

  if ((precisionPtr != NULL) && (*precisionPtr == -1)) {
    *precisionPtr = va_arg(arg_list, int);
  }

  return arg_list;
}
```

14.8.3. Print Format

The **printFormat** function is (directly or indirectly) called by **printf**, **vprintf**, **fprintf**, **vfprintf**, **sprintf**, and **vsprintf**.

```
int printFormat(const char* format, va_list arg_list) {
  int index, width = 0, precision = 0;
  BOOL percent = FALSE, plus = FALSE, minus = FALSE, space = FALSE,
       zero = FALSE, grid = FALSE, widthStar = FALSE,
       period = FALSE, precisionStar = FALSE,
       shortInt = FALSE, longInt = FALSE, longDouble = FALSE;
  g_outChars = 0;

  for (index = 0; format[index] != '\0'; ++index) {
    char c = format[index];

    if (percent) {
      switch (c) {
```

A plus sign causes the plus sign be preceded a positive value.

```
        case '+':
          plus = TRUE;
          break;

        case '-':
          minus = TRUE;
          break;

        case ' ':
          space = TRUE;
          break;

        case '0':
          zero = TRUE;
          break;

        case '#':
          grid = TRUE;
          break;

        case '.':
          period = TRUE;
          break;

        case '*':
          if (!period) {
            width = -1;
          }
          else {
            precision = -1;
          }
          break;

        case 'h':
          shortInt = TRUE;
          break;

        case 'l':
          longInt = TRUE;
          break;

        case 'L':
          longDouble = TRUE;
          break;

        case 'i':
        case 'd':
        case 'u':
        case 'b':
        case 'o':
        case 'x':
        case 'X':
        case 'c':
        case 's':
        case 'f':
        case 'e':
        case 'E':
        case 'g':
        case 'G':
        case 'p':
        case 'n':
        case '%': {
```

For all kinds of values, we call **printArgument** to write the value. For non-zero codes, we write the value left justified.

```
            if (minus) {
              int startChars = g_outChars;
              arg_list = printArgument(&format[index], arg_list, plus, space,
                                       grid, &width, precision, shortInt,
                                       longInt, longDouble, TRUE, NULL);

              { int field = g_outChars - startChars;

                while (field++ < width) {
                  printChar(' ');
                }
              }
            }
```

For zero codes, we write the value right justified.

```
        else if (zero) {
          int startChars = g_outChars, oldOutStatus = g_outStatus;
          BOOL negative = FALSE;

          g_outStatus = BLANK;
          printArgument(&format[index], arg_list, FALSE, FALSE, grid,
                        &width, precision, shortInt, longInt,
                        longDouble, FALSE, &negative);
          g_outStatus = oldOutStatus;

          { int field = g_outChars - startChars;
            g_outChars = startChars;

            if (negative) {
              printChar('X');
              printChar('-');
              ++field;
            }
            else if (plus) {
              printChar('+');
              ++field;
            }
            else if (space) {
              printChar(' ');
              ++field;
            }

            while (field++ < width) {
              printChar('0');
            }

            arg_list = printArgument(&format[index], arg_list, FALSE,
                                     FALSE, grid, NULL, precision,
                                     shortInt, longInt, longDouble,
                                     FALSE, NULL);
          }
        }
        else {
          int startChars = g_outChars, oldOutStatus = g_outStatus;

          g_outStatus = BLANK;
          printArgument(&format[index], arg_list, plus, space, grid,
                        &width, precision, shortInt, longInt,
                        longDouble, TRUE, NULL);
          g_outStatus = oldOutStatus;
          { int field = g_outChars - startChars;
            g_outChars = startChars;

            while (field++ < width) {
              printChar(' ');
            }

            arg_list = printArgument(&format[index], arg_list, plus, space,
                                     grid, NULL, precision, shortInt,
                                     longInt, longDouble, TRUE, NULL);
          }
        }

        percent = FALSE;
      }
      break;

    default: {
        int value = 0;
        while (isdigit(c)) {
          value = (10 * value) + (c - '0');
          c = format[++index];
        }
        --index;

        if (!period) {
          width = value;
        }
        else {
          precision = value;
       }
      }
      break;
  }
}
else {
  if (c == '%') {
    percent = TRUE;
    plus = FALSE;
    minus = FALSE;
    space = FALSE;
    zero = FALSE;
    grid = FALSE;
    widthStar = FALSE;
    period = FALSE;
    precisionStar = FALSE;
    shortInt = FALSE;
    longInt = FALSE;
    longDouble = FALSE;
    width = 0;
    precision = 0;
  }
  else {
    printChar(c);
  }
}
```

```
  }

  if (g_outStatus == STRING) {
    char* outString = (char*) g_outDevice;
    outString[g_outChars] = '\0';
  }

  return g_outChars;
}
```

14.8.4. printf

The **printf** function is a variadic function that write text to **stdout**.

```
int printf(char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vprintf(format, arg_list);
}
```

The **vprintf** function is a non-variadic function that takes a variadic list and writes text to **stdout**.

```
int vprintf(char* format, va_list arg_list) {
  return vfprintf(stdout, format, arg_list);
}
```

The **fprintf** function is a variadic function that writes text to a stream.

```
int fprintf(FILE* outStream, char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vfprintf(outStream, format, arg_list);
}
```

The **vprintf** function is a non-variadic function that takes a variadic list and writes text to a stream.

```
int vfprintf(FILE* outStream, char* format, va_list arg_list) {
  g_outStatus = DEVICE;
  g_outDevice = (void*) outStream;
  return printFormat(format, arg_list);
}
```

The **sprintf** function is a variadic function that writes text to a string.

```
int sprintf(char* outString, char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vsprintf(outString, format, arg_list);
}
```

The **vsprintf** function is a non-variadic function that takes a variadic list and writes text to a string.

```
int vsprintf(char* outString, char* format, va_list arg_list) {
  g_outStatus = STRING;
  g_outDevice = (void*) outString;
  return printFormat(format, arg_list);
}
```

14.9. Standard Input

The standard input library reads values from **stdin** of a file.

scanf.h
```
#ifndef __SCANF_H__
#define __SCANF_H__

#define DEVICE 0
#define STRING 1

#define EOF -1

char scanChar(void);
void unscanChar(char c);
void scanString(char* string, int precision);
long scanLongInt(int base);
unsigned long scanUnsignedLongInt(int base);
long double scanLongDouble(void);

int scanf(const char* format, ...);
int vscanf(const char* format, va_list arg_list);
int fscanf(FILE* inStream, const char* format, ...);
int vfscanf(FILE* inStream, const char* format, va_list arg_list);
int sscanf(char* inString, const char* format, ...);
int vsscanf(char* inString, const char* format, va_list arg_list);

#endif
```

scanf.c
```
#include <math.h>
#include <ctype.h>
#include <stdio.h>
#include <stddef.h>
#include <stdarg.h>
#include <String.h>
#include <scanf.h>
#include <printf.h>
```

The **g_inStatus** field decides whether the **g_inDevice** is a stream (**stdin** or a file) or string. The **g_device** field hold the actual device (a file or a string) while **g_inChars** counts the characters read and **g_inCount** counts the arguments read.

```
int g_inStatus, g_inChars;
void* g_inDevice;
int g_inCount;
```

14.9.1. Scan Character and String

The **scanChar** function reads a character from a string or file.

```
char scanChar(void) {
  char c = '\0';
  FILE* stream;
  int handle;
  char* inString;

  switch (g_inStatus) {
    case DEVICE:
      stream = (FILE*) g_inDevice;

      handle = stream->handle;
```

In the Linux environment, we perform a system call, while we in the Windows environment perform a interrupt call.

```
#ifdef __LINUX__
      register_rax = 0x00L;
      register_rdi = (unsigned long) stream->handle;
      register_rsi = (unsigned long) &c;
      register_rdx = 1L;
      syscall();
#endif

#ifdef __WINDOWS__
      register_ah = 0x3Fs;
      register_bx = handle;
      register_cx = 1;
      register_dx = &c;
      interrupt(0x21s);
#endif

      ++g_inChars;
      return c;
```

In case of string reading, we index the input string, and increase the character counts.

```
    case STRING:
      inString = (char*) g_inDevice;
      return inString[g_inChars++];
```

In case of neither device nor string, we do nothing.

```
    default:
      return '\0';
  }
}
```

The **unscanChar** function puts back a character to a file or string.

```
void unscanChar(char /* c */) {
  switch (g_inStatus) {
    case DEVICE:
      --g_inChars;
      break;

    case STRING:
      --g_inChars;
      break;
  }
}
```

14.9.2. Scan Pattern

The **scanPattern** function read the **string** as long as its characters are stored in **pattern**.

```
void scanPattern(char* string, char* pattern, int size, BOOL not) {
  int index = 0;
  char input = scanChar();

  while (isspace(input)) {
    input = scanChar();
  }
  if (string != NULL) {
    while ((!not && strnchr(pattern, size, input)) ||
           (not && !strnchr(pattern, size, input))) {
      string[index++] = input;
      input = scanChar();
    }

    string[index] = '\0';
  }
  else {
    while ((!not && strnchr(pattern, size, input)) ||
           (not && !strnchr(pattern, size, input))) {
      input = scanChar();
    }
  }
}
```

The **strnchr** function searches the string **text** for the character **c** for at most **size** characters.

```
static char* strnchr(const char* text, int size, int i) {
  int index;
  char c = (char) i;

  for (index = 0; index < size; ++index) {
    if (text[index] == c) {
      return &text[index];
    }
  }

  return NULL;
}
```

The **scanString** function reads a string.

```
void scanString(char* string, int precision) {
  int index = 0;
  char input = scanChar();
  BOOL found = FALSE;
```

We begin by reading potential trailing white-spaces.

```
  while (isspace(input)) {
    input = scanChar();
  }
```

If the precision is zero, we read until we reach the terminating white space, end-of-file, or newline. // XXX

```
  if (string != NULL) {
    if (precision == 0) {
      while (!isspace(input) && (input != EOF) && (input != '\n')) {
        string[index++] = input;
        input = scanChar();
        found = TRUE;
        ++g_inChars;
      }

      string[index] = '\0';
      ++g_inChars;
    }
```

If the precision is not zero, we read until the precision is zero or we reach a terminating character.

```
    else {
      while ((precision-- > 0) && (!isspace(input) &&
             (input != EOF) && (input != '\n'))) {
        string[index++] = input;
        input = scanChar();
        found = TRUE;
        ++g_inChars;
      }

      if (precision > 0) {
        string[index] = '\0';
        ++g_inChars;
      }
    }
  }
  else {
    if (precision == 0) {
      while (!isspace(input) && (input != EOF) &&
             (input != '\n')) {
        input = scanChar();
        found = TRUE;
        ++g_inChars;
      }

      ++g_inChars;
    }
```

If there is no string to read, we instead read from the device.

```
    else {
      while ((precision-- > 0) && (!isspace(input) &&
             (input != EOF) && (input != '\n'))) {
        input = scanChar();
        found = TRUE;
        ++g_inChars;
      }

      if (precision > 0) {
        ++g_inChars;
      }
    }
  }

  if (found) {
    ++g_inCount;
  }
}
```

The static **isDigitInBase** function return true if the character is a digit within the base.

```
static BOOL isDigitInBase(char c, int base) {
  if (isdigit(c)) {
    int value = c - '0';
    return ((value >= 0) && (value < base));
  }
  else if (islower(c)) {
    int value = (c - 'a') + 10;
    return ((value >= 0) && (value < base));
  }
  else if (isupper(c)) {
    int value = (c - 'A') + 10;
    return ((value >= 0) && (value < base));
  }
  else {
    return FALSE;
  }
}
```

The static **digitToValue** function return the integer stored in the character.

```
static int digitToValue(char c) {
  if (isdigit(c)) {
    return (c - '0');
  }
  else if (islower(c)) {
    return ((c - 'a') + 10);
  }
  else if (isupper(c)) {
    return ((c - 'A') + 10);
  }
  else {
    return 0;
  }
}
```

14.9.3. Scanning Values

The **scanLongInt** function reads a signed long integer value.

```
long scanLongInt(int base) {
  long longValue = 0l;
  BOOL minus = FALSE, found = FALSE;
  char input = scanChar();

  while (isspace(input)) {
    input = scanChar();
  }

  if (input == '+') {
    input = scanChar();
  }
  else if (input == '-') {
    minus = TRUE;
    input = scanChar();
  }

  if (base == 0) {
    if (input == '0') {
      input = scanChar();

      if (tolower(input) == 'x') {
        base = 16;
        input = scanChar();
      }
      else {
        base = 8;
      }
    }
    else {
      base = 10;
    }
  }

  while (isDigitInBase(input, base)) {
    longValue *= base;
    longValue += digitToValue(input);
    input = scanChar();
    found = TRUE;
  }

  if (minus) {
    longValue = -longValue;
  }

  if (found) {
    ++g_inCount;
  }

  unscanChar(input);
  return longValue;
}
```

The **scanUnsignedLongInt** function reads an unsigned long integer value.

```
unsigned long scanUnsignedLongInt(int base) {
  unsigned long unsignedLongValue = 0, digit;
  char input = scanChar();
  BOOL found = TRUE;

  while (isspace(input)) {
    input = scanChar();
  }

  if (input == '+') {
    input = scanChar();
  }

  if (base == 0) {
    if (input == '0') {
      input = scanChar();

      if (tolower(input) == 'x') {
        base = 16;
        input = scanChar();
      }
      else {
        base = 8;
      }
    }
    else {
      base = 10;
    }
  }

  while (isDigitInBase(input, base)) {
    unsignedLongValue *= base;
    unsignedLongValue += digitToValue(input);
    found = TRUE;
    input = scanChar();
  }

  if (found) {
    ++g_inCount;
  }

  unscanChar(input);
  return unsignedLongValue;
}
```

The **scanLongDouble** function reads a double value.

```
long double scanLongDouble(void) {
  BOOL minus = FALSE, found = FALSE;
  long double value = 0.0L, factor = 1.0L;
  char input = scanChar();

  while (isspace(input)) {
    input = scanChar();
  }

  if (input == '+') {
    input = scanChar();
  }
  else if (input == '-') {
    minus = TRUE;
    input = scanChar();
  }

  while (isdigit(input)) {
    value = (10.0L * value) + ((long double) (input - '0'));
    input = scanChar();
    found = TRUE;
  }

  if (input == '.') {
    input = scanChar();

    while (isdigit(input)) {
      factor /= 10.0L;
      value += factor * ((long double) (input - '0'));
      input = scanChar();
      found = TRUE;
    }
  }

  if (tolower(input) == 'e') {
    double exponent = (double) scanLongInt(10);
    value *= pow(10.0, exponent);
  }
  else {
    unscanChar(input);
  }

  if (minus) {
    value = -value;
  }

  if (found) {
    ++g_inCount;
  }

  return value;
}
```

14.9.4. Scan Format

The **scanFormat** function is called by **scanf**, **vscanf**, **fscanf**, **vfscanf**, **sscanf**, and **vsscanf**.

```
int scanFormat(char* format, va_list arg_list) {
  char c, *charPtr;
  BOOL percent = FALSE, shortInt = FALSE, longIntOrDouble = FALSE,
       longDouble = FALSE, star = FALSE;

  long longValue, *longPtr;
  short* shortPtr;
  int index, *intPtr, *charsPtr;

  unsigned long unsignedLongValue, *unsignedLongPtr;
  unsigned short* unsignedShortPtr;
  unsigned int* unsignedIntPtr;
  long double longDoubleValue;

  g_inCount = 0;
  g_inChars = 0;

  for (index = 0; format[index] != '\0'; ++index) {
    c = format[index];

    { int d = c + 1;
      if (percent) {
        switch (d - 1) {
          case 'h':
            shortInt = TRUE;
            break;

          case 'l':
            longIntOrDouble = TRUE;
            break;

          case 'L':
            longDouble = TRUE;
            break;

          case '*':
            star = TRUE;
            break;

          case 'c': {
              char charValue = scanChar();

              if (!star) {
                charPtr = va_arg(arg_list, char*);
                *charPtr = charValue;
              }

              percent = FALSE;

              if (charValue != EOF) {
                ++g_inCount;
              }
            }
            break;

          case 's':
            if (!star) {
              charPtr = va_arg(arg_list, char*);
              scanString(charPtr, 0);
            }
            else {
              scanString(NULL, 0);
            }

            percent = FALSE;
            break;

          case 'i':
          case 'd':
            longValue = scanLongInt(10);

            if (!star) {
              if (shortInt) {
                shortPtr = va_arg(arg_list, short*);
                *shortPtr = (short) longValue;
              }
              else if (!longIntOrDouble) {
                intPtr = va_arg(arg_list, int*);
                *intPtr = (int) longValue;
              }
              else {
                longPtr = va_arg(arg_list, long*);
                *longPtr = longValue;
              }
            }

            percent = FALSE;
            break;

          case 'o':
            unsignedLongValue = scanUnsignedLongInt(8);

            if (!star) {
              if (shortInt) {
                unsignedShortPtr = va_arg(arg_list, unsigned short*);
                *unsignedShortPtr = (short) unsignedLongValue;
              }
              else if (!longIntOrDouble) {
                unsignedIntPtr = va_arg(arg_list, unsigned int*);
                *unsignedIntPtr = (int) unsignedLongValue;
              }
              else {
                unsignedLongPtr = va_arg(arg_list, unsigned long*);
                *unsignedLongPtr = unsignedLongValue;
              }
            }

            percent = FALSE;
            break;

          case 'x':
            unsignedLongValue = scanUnsignedLongInt(16);

            if (!star) {
              if (shortInt) {
                unsignedShortPtr = va_arg(arg_list, unsigned short*);
                *unsignedShortPtr = (short) unsignedLongValue;
              }
              else if (!longIntOrDouble) {
                unsignedIntPtr = va_arg(arg_list, unsigned int*);
                *unsignedIntPtr = (int) unsignedLongValue;
              }
              else {
                unsignedLongPtr = va_arg(arg_list, unsigned long*);
                *unsignedLongPtr = unsignedLongValue;
              }
            }

            percent = FALSE;
            break;

          case 'u':
            unsignedLongValue = scanUnsignedLongInt(0);

            if (!star) {
              if (shortInt) {
                unsignedShortPtr = va_arg(arg_list, unsigned short*);
                *unsignedShortPtr = (short) unsignedLongValue;
              }
              else if (!longIntOrDouble) {
                unsignedIntPtr = va_arg(arg_list, unsigned int*);
                *unsignedIntPtr = (int) unsignedLongValue;
              }
              else {
                unsignedLongPtr = va_arg(arg_list, unsigned long*);
                *unsignedLongPtr = unsignedLongValue;
              }
            }

            percent = FALSE;
            break;
```

```
case 'e':
case 'f':
case 'g':
  longDoubleValue = scanLongDouble();

  if (!star) {
    if (longIntOrDouble) {
      double* doublePtr = va_arg(arg_list, double*);
      *doublePtr = (double) longDoubleValue;
    }
    else if (longDouble) {
      long double* longDoublePtr = va_arg(arg_list, long double*);
      *longDoublePtr = longDoubleValue;
    }
    else {
      float* floatPtr = va_arg(arg_list, float*);
      *floatPtr = (float) longDoubleValue;
    }
  }

  percent = FALSE;
  break;

case '[': {
    BOOL not = FALSE;
    ++index;

    if (format[index] == '^') {
      not = TRUE;
      ++index;
    }

    { int startIndex = index;

      while (format[index] != ']') {
        ++index;
      }

      { int size = index - startIndex;
        char c = format[index];
        format[index] = '\0';

        if (!star) {
          char* string = va_arg(arg_list, char*);
          scanPattern(string, &format[startIndex], size, not);
        }
        else {
          scanPattern(NULL, &format[startIndex], size, not);
        }

        format[index] = c;
      }
    }
  }
  break;
```

```
          case 'n':
            charsPtr = va_arg(arg_list, int*);
            *charsPtr = g_inChars;
            percent = FALSE;
            break;

          default:
            printf("scanFormat c = '%c'\n", c);
            break;
        }
      }
      else {
        if (c == '%') {
          percent = TRUE;
          shortInt = FALSE;
          longIntOrDouble = FALSE;
          longDouble = FALSE;
          star = FALSE;
        }
      }
    }
  }

  return g_inCount;
}
```

14.9.5. scanf

The **scanf** function is a variadic function that reads text from **stdin**.

```
int scanf(const char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vscanf(format, arg_list);
}
```

The **vscanf** function is a non-variadic function that takes a variadic list and reads text from **stdin**.

```
int vscanf(const char* format, va_list arg_list) {
  return vfscanf(stdin, format, arg_list);
}
```

The **fscanf** function is a variadic function that reads text from a stream.

```
int fscanf(FILE* inStream, const char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vfscanf(inStream, format, arg_list);
}
```

The **vfscanf** function is a non-variadic function that takes a variadic list and reads text from a stream.

```
int vfscanf(FILE* inStream, const char* format, va_list arg_list) {
  g_inStatus = DEVICE;
  g_inDevice = (void*) inStream;
  return scanFormat(format, arg_list);
}
```

The **sscanf** function is a variadic function that reads text from a string.

```
int sscanf(char* inString, const char* format, ...) {
  va_list arg_list;
  va_start(arg_list, format);
  return vsscanf(inString, format, arg_list);
}
```

The **vsscanf** function is a non-variadic function that takes a variadic list and reads text from a string.

```
int vsscanf(char* inString, const char* format, va_list arg_list) {
  g_inStatus = STRING;
  g_inDevice = (void*) inString;
  return scanFormat(format, arg_list);
}
```

14.10. File Management

The file management standard library holds function for creating, removed, and renaming files as well as reading values from files and writing values to files.

file.h

```
#ifndef __FILE_H__
#define __FILE_H__

#define FOPEN_MAX 20
#define FILENAME_MAX 16

#define fpos_t int
#define EOF -1
```

The **File** type holds the information about a file. The **handle** field holds the connection to the file system.

```
typedef struct {
  BOOL open;
  unsigned int handle;
  char name[FILENAME_MAX], ungetc;
  int errno;
  unsigned int position, size;
  BOOL temporary;
} FILE;

extern FILE *stdin, *stdout, *stderr;

extern enum {EEXIST, ENOENT, EACCES};
extern enum {SEEK_SET, SEEK_CUR, SEEK_END};
extern enum {READ, WRITE, READ_WRITE};

#ifdef __LINUX__
  #define O_RDONLY    0x0000L      /* open for reading only */
  #define O_WRONLY    0x0001L      /* open for writing only */
  #define O_CREAT     0x0200L      /* create if nonexistant */
  #define O_TRUNC     0x0400L      /* truncate to zero length */

  #define FILE_DESC_STDOUT 1

  #define SYS_EXIT 1
  #define SYS_READ 3
  #define SYS_WRITE 4
  #define SYS_OPEN 5
  #define SYS_CLOSE 6
#endif

#define getc(stream) fgetc(stream)

BOOL fileexists(const char* name);
FILE* fopen(const char* filename, const char* mode);
FILE* freopen(const char* filename, const char* mode, FILE* stream);
int fflush(FILE* stream);
int fclose(FILE* stream);
int remove(const char* name);
int rename(const char* oldName, const char* newName);
int setvbuf(FILE* stream, char* buffer, int mode, size_t size);
void setbuf(FILE* stream, char* buffer);
int fgetc(FILE* stream);
char* fgets(char* s, int n, FILE* stream);
int fputc(int i, FILE* stream);
int fputs(const char* s, FILE* stream);
int getchar(void);
char* gets(char* s);
int putchar(int c);
int puts(const char* s);
int ungetc(int c, FILE* stream);
size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t nobj, FILE* stream);
int fseek(FILE* stream, int offset, int origin);
int ftell(FILE* stream);
void rewind(FILE* stream);
int fgetpos(FILE* stream, fpos_t* ptr);
int fsetpos(FILE* stream, const fpos_t* ptr);
void clearerr(FILE* stream);
BOOL feof(FILE* stream);
int ferror(FILE* stream);
void perror(const char* s);

#endif
```

file.c

```
#include <stdio.h>
#include <errno.h>
#include <locale.h>
#include <string.h>

FILE g_fileArray[FOPEN_MAX] = {{TRUE, 0}, {TRUE, 1}, {TRUE, 2}};
FILE *stdin = &g_fileArray[0], *stdout = &g_fileArray[1],
     *stderr = &g_fileArray[2];

enum {EEXIST, ENOENT, EACCES};
enum {SEEK_SET = 0, SEEK_CUR = 1, SEEK_END = 2};
enum {READ = 0x40, WRITE = 0x41, READ_WRITE = 0x42};

#define MAX(a,b) (((a) > (b)) ? (a) : (b))

#define FILE_NOT_FOUND 0x02

#define PRINT(x,y) { printf(#x " = <%" #y ">\n", (x)); }
```

14.10.1. File Open and Close

The static **filecreate** function create a file with the given name.

```
static int filecreate(const char* name) {
```

In the Linux environment perform a system call. The handle of the file is stored in the **eax** register.

```
#ifdef __LINUX__
  register_rax = 85;
  register_rdi = (unsigned long) name;
  register_rsi = 0777L; // octal
  syscall();
  return register_eax;
#endif
```

In the Windows environment perform a interrupt call. The handle of the file is stored in the **ax** register.

```
#ifdef __WINDOWS__
  register_ah = 0x3Cs;
  register_cx = 0x00;
  register_dx = name;
  interrupt(0x21s);

  { int handle = register_ax;

    if (carry_flag) {
      errno = FOPEN;
      return -1;
    }

    return handle;
  }
#endif
}
```

The **fileexists** function check whether the file exists.

```
BOOL fileexists(const char* name) {
#ifdef __LINUX__
  register_rax = 5;
  register_rdi = (unsigned long) name;
  register_rsi = NULL;
  syscall();
  return register_rbx;
#endif

#ifdef __WINDOWS__
  register_ah = 0x43s;
  register_al = 0x00s;
  register_dx = name;
  interrupt(0x21s);
  return !carry_flag;
#endif
}
```

The **fileopen** function opens a file.

```
static int fileopen(const char* name, unsigned short mode) {
```

```
#ifdef __LINUX__
  register_rax = 2;
  register_rdi = (unsigned long) name;
  register_rsi = (unsigned long) mode;
  syscall();
  return register_rax;
#endif

#ifdef __WINDOWS__
  register_ah = 0x3Ds;
  register_al = mode;
  register_dx = name;
  interrupt(0x21s);

  if (carry_flag) {
    errno = FOPEN;
    return -1;
  }
  else {
    return register_ax;
  }
#endif
}
```

The fopen function opens a file and places its information in the **fileArray** array.

```
FILE* fopen(const char* name, const char* mode) {
  int index;
  for (index = 0; index < FOPEN_MAX; ++index) {

    if (!g_fileArray[index].open) {
      return freopen(name, mode, &g_fileArray[index]);
    }
  }

  return NULL;
}
```

The freopen function opens the file. The mode may be “r” for reading, “w” for writing, and “a” for append.

```
FILE* freopen(const char* name, const char* mode, FILE* stream) {
  int handle = -1;

  if (strcmp(mode, "r") == 0) {
    handle = fileopen(name, (unsigned short) READ);
  }
  else if (strcmp(mode, "w") == 0) {
    handle = filecreate(name);
  }
  else if (strcmp(mode, "a") == 0) {
    handle = fileopen(name, (unsigned short) WRITE);

    if (handle != -1) {
      fseek(stream, 0L, (int) SEEK_END);
    }
    else {
      handle = filecreate(name);
    }
  }
  else if (strcmp(mode, "r+") == 0) {
    handle = fileopen(name, (unsigned short) READ_WRITE);
  }
  else if (strcmp(mode, "w+") == 0) {
    if (fileexists(name)) {
      handle = fileopen(name, (unsigned short) READ_WRITE);
    }
    else {
      handle = filecreate(name);
    }
  }
  else if (strcmp(mode, "a+") == 0) {
    handle = fileopen(name, (unsigned short) READ_WRITE);

    if (handle != -1) {
      fseek(stream, 0L, (int) SEEK_END);
    }
    else {
      handle = filecreate(name);
    }
  }
```

If the file was successfully opened, we assign the values to the **FILE** struct.

```
  if (handle != -1) {
    stream->open = TRUE;
    stream->handle = handle;
    stream->size = 0l; // filesize(handle);
    strcpy(stream->name, name);
    stream->temporary = FALSE;
    return stream;
  }
```

If the file was not successfully opened, we just set the **open** field to false.

```
  else {
    stream->open = FALSE;
    return NULL;
  }
}
```

The **fflush** function flushes all open files.

```
int fflush(FILE* stream) {
  if (stream == NULL) {
    int index;

    for (index = 0; index < FOPEN_MAX; ++index) {
      if (g_fileArray[index].open) {
        if (fflush(&g_fileArray[index]) == EOF) {
          return EOF;
        }
      }
    }
  }

  // ...
  return 0;
}
```

The fclose function closes the file if the stream is not null. If the stream is null, it closes all opened files.

```
int fclose(FILE* stream) {
  if (stream != NULL) {
#ifdef __LINUX__
  register_rax = 3L;
  register_rdi = (unsigned long) stream->handle;
  syscall();
  return 0;
#endif

#ifdef __WINDOWS__
    register_ah = 0x3Es;
    register_bx = stream->handle;
    interrupt(0x21s);

    if (carry_flag) {
      errno = FCLOSE;
      return -1;
    }

    if (stream->temporary) {
      remove(stream->name);
    }

    stream->open = FALSE;
    return 0;
#endif
  }
  else {
    int index;

    for (index = 0; index < FOPEN_MAX; ++index) {
      if (g_fileArray[index].open) {
        if (fclose(&g_fileArray[index]) == -1) {
          return -1;
        }
      }
    }

    return 0;
  }
}
```

14.10.2. File Remove and Rename

The **remove** function removes a file and returns zero if it succeeds.

```
int remove(const char* name) {
#ifdef __LINUX__
  register_rax = 88L;
  register_rdi = (unsigned long) name;
  syscall();

  if (register_ebx == 0) {
    return 0;
  }
#endif

#ifdef __WINDOWS__
  register_ah = 0x41s;
  register_cl = 0s;
  register_dx = name;
  interrupt(0x21s);

  if (!carry_flag) {
    return 0;
  }
#endif

  errno = FREMOVE;
  return -1;
}
```

The **rename** function renames a file and returns zero if it succeeds.

```
int rename(const char* oldName, const char* newName) {
#ifdef __LINUX__
  register_rax = 82L;
  register_rdi = (unsigned long) oldName;
  register_rsi = (unsigned long) newName;
  syscall();

  if (register_eax == 0) {
    return 0;
  }
#endif

#ifdef __WINDOWS__
  register_ah = 0x56s;
  register_cl = 0s;
  register_dx = oldName;
  register_di = newName;
  interrupt(0x21s);

  if (!carry_flag) {
    return 0;
  }
#endif

  errno = FRENAME;
  return -1;
}
```

14.10.3. Buffer

The **setvbuf** and **setbuf** functions set buffers. // XXX

```
int setvbuf(FILE* /* stream */, char* /* buffer */,
            int /* mode */, size_t /* size */) {
  return 0;
}
```

```
void setbuf(FILE* /* stream */, char* /* buffer */) {
  // Empty.
}
```

14.10.4. Character and String

The **fgetc** function reads a character from a stream.

```
int fgetc(FILE* stream) {
  char c = '\0';

  if (fread(&c, sizeof (char), 1, stream) > 0) {
    return (int) c;
  }

  return -1;
}
```

The **fgets** function reads a string from a stream.

```
char* fgets(char* text, int size, FILE* stream) {
  int count = 0;
  char prevChar = '\0';

  while ((count < (size - 1))) {
    char currChar = '\0';
    fscanf(stream, "%c", &currChar);

    if ((prevChar == '\r') && (currChar == '\n')) {
      text[count] = '\0';
      break;
    }

    if (currChar == -1) {
      text[count] = '\0';
      break;
    }

    if ((currChar != '\r') && (currChar != '\n')) {
      text[count++] = currChar;
    }

    prevChar = currChar;
  }

  return text;
}
```

The **fputs** function writes a string to a stream.

```
int fputs(const char* s, FILE* stream) {
  int size = (strlen(s) + 1) * sizeof (char);
  return (fwrite(s, size, 1, stream) == size) ? 0 : EOF;
}
```

The getchar function reads a character from **stdin**.

```
int getchar(void) {
  return fgetc(stdin);
}
```

The **gets** function reads a string from **stdin**.

```
char* gets(char* s) {
  if (fgets(s, -1, stdin) != NULL) {
    int size = strlen(s);

    if (size > 0) {
      s[size - 1] = '\0';
    }

    return s;
  }
  else {
    return NULL;
  }
}
```

The **puts** function writes a string to **stdout**.

```
int puts(const char* s) {
  if (fputs(s, stdout) != 0) {
    return fputc('\n', stdout);
  }

  return EOF;
}
```

The **ungetc** function write back a character to a stream.

```
int ungetc(int c, FILE* stream) {
  if (stream->ungetc != EOF) {
    stream->ungetc = (char) c;
  }

  return c;
}
```

14.10.5. Reading and Writing

The **fread** function reads a block of data from the stream.

```
size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream) {
#ifdef __LINUX__
  register_rax = 0L;
  register_rdi = (unsigned long) stream->handle;
  register_rsi = (unsigned long) ptr;
  register_rdx = (unsigned long) (size * nobj);
  syscall();
  return register_eax;
#endif

#ifdef __WINDOWS__
  register_bx = stream->handle;
  register_cx = size * nobj;
  register_ah = 0x3Fs;
  register_dx = ptr;
  interrupt(0x21s);
  if (carry_flag) {
    stream->errno = errno = FREAD;
    return 0;
  }
  else {
    return register_ax;
  }
#endif
}
```

The **fwrite** function writes a block of data to the stream.

```
size_t fwrite(const void* ptr, size_t size, size_t nobj, FILE* stream) {
#ifdef __LINUX__
  register_rax = 1;
  register_rdi = (unsigned long) stream->handle;
  register_rsi = (unsigned long) ptr;
  register_rdx = (unsigned long) (size * nobj);
  syscall();
  return register_eax;
#endif

#ifdef __WINDOWS__
  register_bx = stream->handle;
  register_cx = size * nobj;
  register_ah = 0x40s;
  register_dx = ptr;
  interrupt(0x21s);

  if (carry_flag) {
    stream->errno = errno = FWRITE;
    return 0;
  }
  else {
    return register_ax;
  }
#endif
}
```

The **fseek** function sets and returns the file position. The origin parameter may be **SEEK_SET** for the beginning of the file, **SEEK_CUR** for the current position, and **SEEK_END** for en end of the file.

```
int fseek(FILE* stream, int offset, int origin) {
#ifdef __WINDOWS__
  register_al = (short) origin;
  register_ah = 0x42s;
  register_bx = stream->handle;
  register_cx = 0;
  register_dx = (int) offset;
  interrupt(0x21s);

  if (!carry_flag) {
    stream->position = register_ax;
    return stream->position;
  }
  else {
    stream->errno = FSEEK;
    return -1;
  }
#endif

#ifdef __LINUX__
  register_rax = 8;
  register_rdi = (unsigned long) stream->handle;
  register_rsi = (unsigned long) offset;
  register_rdx = (unsigned long) origin;
  syscall();
  return register_eax;
#endif
}
```

14.10.6. File Positioning

The **ftell** function returns the current position.

```
int ftell(FILE* stream) {
  return fseek(stream, 0, SEEK_CUR);
}
```

The **rewind** function sets the file position to the beginning of the file.

```
void rewind(FILE* stream) {
  (void) fseek(stream, 0, SEEK_SET);
}
```

The **fgetpos** function sets the file position the **ptr** pointer.

```
int fgetpos(FILE* stream, fpos_t* ptr) {
  *ptr = (fpos_t) ftell(stream);
  return 0;
}
```

The **fgetpos** function sets the file position of the **ptr** pointer.

```
int fsetpos(FILE* stream, const fpos_t* ptr) {
  return ((int) fseek(stream, *ptr, (int) SEEK_SET));
}
```

The **feof** function return true if the file position is at the end of the file.

```
BOOL feof(FILE* stream) {
  long unsigned currPosition = fseek(stream, 0L, (int) SEEK_CUR);
  long unsigned lastPosition = fseek(stream, 0L, (int) SEEK_END);
  fseek(stream, currPosition, (int) SEEK_SET);

  { BOOL endOfFile = (currPosition == lastPosition);
    return endOfFile;
  }
}
```

14.10.7. Error Messages

The **clearerr** function sets the **errno** field of the stream as well as the global **errno** variable to zero.

```
void clearerr(FILE* stream) {
  stream->errno = errno = 0;
}
```

The **ferror** function return the **errno** field of the stream.

```
int ferror(FILE* stream) {
  return stream->errno;
}
```

The **perror** function writes the value and message of the global **errno** variable.

```
void perror(const char* s) {
  printf("%s: %s.\n", s, strerror(errno));
}
```

14.11. The Standard Library

The standard library holds functions for type casting, random number generating, absolute value, division and remainder as well as searching and sorting.

stdlib.h

```
#ifndef __STDLIB_H__
#define __STDLIB_H__

#define NULL ((void*) 0)

double atof(const char* s);
int atoi(const char* s);
long atol(const char* s);

double strtod(const char* s, char** endp);
long strtol(const char* s, char** endp, int base);
unsigned long strtoul(const char* s, char** endp, int base);

int rand(void);
void srand(unsigned int seed);

char* getenv(const char* name);
int system(const char* command);

void abort(void);
void exit(int status);

typedef void (*FUNC_PTR)(void);
int atexit(FUNC_PTR fcn);

#define FUNC_MAX 256
#define OPEN_MAX 16

int abs(int value);
long labs(long value);

void* malloc(size_t size);
void* realloc(void* ptr, size_t newSize);
void* calloc(size_t num, size_t size);
void free(void* ptr);

void qsort(void* valueList, size_t listSize, size_t valueSize,
           int (*compare)(const void*, const void*));
void* bsearch(const void* key, const void* valueList,
              size_t listSize, size_t valueSize,
              int (*compare)(const void*, const void*));

int abs(int value);
long labs(long value);

typedef struct {
  int quot, rem;
} div_t;

div_t div(int num, int denum);

typedef struct {
  long quot, rem;
} ldiv_t;

ldiv_t ldiv(long num, long denum);
#endif
```

stdlib.c

```
#include <math.h>
#include <ctype.h>
#include <errno.h>
#include <stdarg.h>
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

extern FILE g_fileArray[];
```

14.11.1. Type Casting

The **atoi** (ASCII-to-integer) and **atol** (ASCII-to-long) functions cast a string to an integer value by calling **strtol**.

```
int atoi(const char* s) {
  return (int) strtol(s, (const char**) NULL, 10);
}

long atol(const char* s) {
  return strtol(s, (const char**) NULL, 10);
}
```

The **strtol** (string-to-long) function casts a string to a long integer with the given base.

```
extern int g_inStatus, g_inChars;
extern void* g_inDevice;

long strtol(const char* s, char** endp, int base) {
  g_inStatus = STRING;
  g_inDevice = s;
  g_inChars = 0;

  { long value = scanLongInt(base);

    if (endp != NULL) {
      *endp = s + g_inChars;
    }

    return value;
  }
}
```

The **strtoul** (string-to-unsigned-long) function casts a string to a long integer with the given base.

```
unsigned long strtoul(const char* s, char** endp, int base) {
  g_inStatus = STRING;
  g_inDevice = s;
  g_inChars = 0;

  { unsigned long unsignedLongValue = scanUnsignedLongInt(base);

    if (endp != NULL) {
      *endp = s + g_inChars;
    }

    return unsignedLongValue;
  }
}

double atof(const char* s) {
  return strtod(s, (char**) NULL);
}
```

The **strtod** (string-to-double)

```
double strtod(const char* s, char** endp) {
  int chars = '\0';
  double value = 0;
  sscanf(s, "%lf%n", &value, &chars);

  if (endp != NULL) {
    *endp = s + chars;
  }

  return value;
}
```

14.11.2. Environment Variables

```
char* getenv(const char* /* name */) {
  return NULL;
}

int system(const char* /* command */) {
  return -1;
}
```

14.11.3. Searching

```
void* bsearch(const void* keyPtr, const void* valueList,
              size_t listSize, size_t valueSize,
              int (*compare)(const void*, const void*)) {
  int firstIndex = 0, lastIndex = listSize - 1;

  if (listSize == 0) {
    return NULL;
  }

  while (TRUE) {
    { char* firstValuePtr = ((char*) valueList) + (firstIndex * valueSize);
      int firstCompare = compare(keyPtr, firstValuePtr);

      if (firstCompare < 0) {
        return NULL;
      }
      else if (firstCompare == 0) {
        return firstValuePtr;
      }
    }

    { char* lastValuePtr = ((char*) valueList) + (lastIndex * valueSize);
      int lastCompare = compare(keyPtr, lastValuePtr);

      if (lastCompare > 0) {
        return NULL;
      }
      else if (lastCompare == 0) {
        return lastValuePtr;
      }
    }

    { int middleIndex = (firstIndex + lastIndex) / 2;
      char* middleValuePtr = ((char*)valueList) + (middleIndex * valueSize);
      int middleCompare = compare(keyPtr, middleValuePtr);

      if (middleCompare < 0) {
        lastIndex = middleIndex;
      }
      else if (middleCompare > 0) {
        firstIndex = middleIndex;
      }
      else {
        return middleValuePtr;
      }
    }
  }
}
```

14.11.4. Random Number Generation

```
static long g_randValue;

#define A 1664525l
#define C 1013904223l
#define RAND_MAX 127

int rand(void) {
  g_randValue = ((A * g_randValue) + C) % RAND_MAX;
  return (int) g_randValue;
}
```

```
void srand(unsigned int seed) {
  g_randValue = (long) seed;
}

#define STACK_TOP_ADDRESS   32766
#define HEAP_BOTTOM_ADDRESS 32764
#define HEAP_TOP_ADDRESS    32762
#define HEADER_SIZE (2 * sizeof (unsigned int))
```

14.11.5. Abortion and Exit

```
void abort(void) {
#ifdef __WINDOWS__
  register_ah = 0x4Cs;
  register_al = -1s;
  interrupt(0x21s);
#endif

#ifdef __LINUX__
  register_rax = 60L;
  register_rdi = -1L;
  syscall();
#endif
}

FUNC_PTR g_funcArray[FUNC_MAX] = { NULL };

int atexit(FUNC_PTR fcn) {
  int index;

  for (index = 0; index < FUNC_MAX; ++index) {
    if (g_funcArray[index] == NULL) {
      g_funcArray[index] = fcn;
      return 0;
    }
  }

  return -1;
}

void exit(int status) {
  int index;

  for (index = (FUNC_MAX - 1); index >= 0; --index) {
    if (g_funcArray[index] != NULL) {
      g_funcArray[index]();
    }
  }

#ifdef __WINDOWS__
  register_al = (short) status;
  register_ah = 0x4Cs;
  interrupt(0x21s);
#endif

#ifdef __LINUX__
  register_rax = 60L;
  register_rdi = (unsigned long) status;
  syscall();
#endif
}
```

14.11.6. Sorting

```
void swap(char* leftValuePtr, char* rightValuePtr, int valueSize) {
  int index;
  for (index = 0; index < valueSize; ++index) {
    char tempValue = leftValuePtr[index];
    leftValuePtr[index] = rightValuePtr[index];
    rightValuePtr[index] = tempValue;
  }
}

void qsort(void* valueList, size_t listSize, size_t valueSize,
           int (*compare) (const void*, const void*)) {
  char* charList = (char*) valueList;
  int size;
  for (size = (listSize - 1); size > 0; --size)  {
    int index;
    BOOL update = FALSE;
    for (index = 0; index < size; ++index)  {
      char* valuePtr1 = charList + (index * valueSize);
      char* valuePtr2 = charList + ((index + 1) * valueSize);

      if (compare(valuePtr1, valuePtr2) > 0) {
        memswap(valuePtr1, valuePtr2, valueSize);
        update = TRUE;
      }
    }

    if (!update) {
      break;
    }
  }
}
```

14.11.7. Absolute Values

```
int abs(int value) {
  return (value < 0) ? -value : value;
}

long labs(long value) {
  return (value < 0l) ? -value : value;
}
```

14.11.8. Division and Modulo

```
div_t div(int num, int denum) {
  div_t result = {0, 0};

  if (denum == 0) {
    errno = EDOM;
    return result;
  }
  result.quot = num / denum;
  result.rem = num % denum;
  return result;
}

ldiv_t ldiv(long num, long denum) {
  ldiv_t result = {0, 0};

  if (denum == 0) {
    errno = EDOM;
    return result;
  }

  result.quot = num / denum;
  result.rem = num % denum;
  return result;
}
```

14.11.9. Dynamic Memory Management

14.12. Time

time.h

```
#ifndef __TIME_H__
#define __TIME_H__

#define size_t int
#define time_t unsigned long
#define clock_t long

struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};

extern clock_t clock(void);
extern time_t time(time_t* time);
extern double difftime(time_t time2, time_t time1);
extern time_t mktime(struct tm* timeStruct);

extern char* asctime(const struct tm* timeStruct);
extern char* ctime(const time_t* time);
extern struct tm* gmtime(const time_t* time);
extern struct tm* localtime(const time_t* time);

extern size_t strftime(char* buffer, size_t size, const char* format,
                              const struct tm* timeStruct);

#endif
```

time.c
```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <assert.h>

static int getWeekNumber(const struct tm*);

clock_t clock(void) {
  return -1;
}

static BOOL isLeapYear(int year) {
  return (((year % 4) == 0) && ((year % 100) != 0)) || ((year % 400) == 0);
}
```

14.12.1. Obtaining Time

The **time** function returns a value representing the number of seconds since January 1, 1970, Greenwich time.

```
time_t time(time_t* timePtr) {
  time_t time;
```

There is a large difference between the Linux and Windows environments. In the Linux environment we can use a system call to obtain the value directly. We add the address of the value to the call, and it loads the value.

```
#ifdef __LINUX__
  register_rax = 201L;
  register_rdi = (unsigned long) &time;
  syscall();
#endif
```

In the Windows environment, on the other hand, we must obtain the current data and time, and manually calculate the value.

```
#ifdef __WINDOWS__
  int year;
  short month, dayOfMonth;
  short hour, min, sec;
```

We perform interrupt calls to obtain the current data and time.

```
  register_ah = 0x2As;
  interrupt(0x21s);
  year = register_cx - 1900;
  month = register_dh - 1s;
  dayOfMonth = register_dl;

  register_ah = 0x2Cs;
  interrupt(0x21s);
  hour = register_ch;
  min = register_cl;
  sec = register_dh;
```

If there is locale settings, we decrease the hour with the winter time zone to When we have obtained the year, month, day of month, hour, minute, and second, we still need to calculate the day of the year.

```
  { struct lconv* localeConvPtr = localeconv();

    if (localeConvPtr != NULL) {
      hour -= localeConvPtr->winterTimeZone;
    }
  }
```

When we have obtained the year, month, day of month, hour, minute, and second, we need to calculate the day of the year.

```
  { const int daysOfMonths[] = {31, isLeapYear(year) ? 29 : 28, 31,
                                30, 31, 30, 31, 31, 30, 31, 30, 31};
    int dayOfYear = dayOfMonth - 1, monthIndex;

    for (monthIndex = 0; monthIndex < month; ++monthIndex) {
      dayOfYear += daysOfMonths[monthIndex];
    }
```

When we have the time, date, and day of year, we call **mktime** to obtain the number of seconds from January 1, 1970.

```
    { struct tm s = {sec, min, hour, dayOfMonth, month, year, 0, dayOfYear};
      time = mktime(&s);
    }
  }
#endif

  if (timePtr != NULL) {
    *timePtr = time;
  }

  return time;
}
```

The **mktime** function calculates the number of seconds from January 1, 1970, given the date and time in the **tm** structure.

```
time_t mktime(struct tm* tp) {
  if (tp != NULL) {
    const long leapDays = (tp->tm_year - 69) / 4;
```

Remember that the tm_year field holds the year from the year 1900.

```
    const long totalDays = 365 * (tp->tm_year - 70) + leapDays + tp->tm_yday;
    return (86400L * totalDays) + (3600L * tp->tm_hour) +
           (60L * tp->tm_min) + tp->tm_sec;
  }

  return 0;
}
```

The **gmtime** function calculates the date and time, given the number of seconds from January 1, 1970, in the Greenwich Mean Time zone. The result is stored in the static global **g_timeStruct** structure, and a pointer to the structure is returned.

```
static struct tm g_timeStruct;

struct tm* gmtime(const time_t* timePtr) {
  if (timePtr != NULL) {
    time_t totalSeconds = *timePtr;
    const long secondsOfDay = totalSeconds % 86400L,
               secondsOfHour = secondsOfDay % 3600;
```

It easy to calculate the time, we simply divide the total number of seconds with 3,600 for the hour, and with 60 for the minute. The second if the remainder after we have divided by 60, so we perform modulo 60.

```
    g_timeStruct.tm_hour = secondsOfDay / 3600;
    g_timeStruct.tm_min = secondsOfHour / 60;
    g_timeStruct.tm_sec = secondsOfHour % 60;
```

The number of total days is the total number of seconds divided by 86,400.

```
    { long totalDays = totalSeconds / 86400L;
```

The weekday is also easy to calculate. January 1, 1970 was on a Thursday. Therefore, if the total days less than three, we add four to obtain the weekday.

```
      if (totalDays < 3) {
        g_timeStruct.tm_wday = totalDays + 4;
      }
```

If the total days are at least three, we subtract three and perform module seven to obtain the weekday.

```
      else {
        g_timeStruct.tm_wday = (totalDays - 3) % 7;
      }
```

However, we begin by calculating the year day. When doing so, we need to take into consideration the number of leap days since January 1, 1970.

```
      { int year = 1970 + (totalDays / 365);
        const int leapDays = (year - 1969) / 4;
        int daysOfYear = (totalDays % 365) - leapDays;
```

When subtracting the leap days from the year days, it may be less than zero. In that case, we decrease the year by one and add 366 to the days in case of a leap year, 365 otherwise.

```
        if (daysOfYear < 0) {
          --year;

          if (isLeapYear(year)) {
            daysOfYear += 366;
          }
          else {
            daysOfYear += 365;
          }

          g_timeStruct.tm_year = year - 1900;
          g_timeStruct.tm_yday = daysOfYear;
        }
```

If is a little more complicated to calculate the month and month day since the months have different length. We increase the month and subtract its number of days from the year day as long as the month exceeds the remaining year days.

```
        { const int daysOfMonths[] = {31, isLeapYear(year) ? 29 : 28, 31,
                                  30, 31, 30, 31, 31, 30, 31, 30, 31};
          int month = 0;
          while (totalDays >= daysOfMonths[month]) {
            totalDays -= daysOfMonths[month++];
          }

          g_timeStruct.tm_mon = month;
          g_timeStruct.tm_mday = totalDays + 1;
          g_timeStruct.tm_isdst = -1;
          return &g_timeStruct;
        }
      }
    }
  }

  return NULL;
}
```

The **localtime** function returns a pointer to the tm structure in the same way as **gmtime** above. The difference is that **localtime** returns the local time; that is, it takes the time difference into consideration.

```
struct tm* localtime(const time_t* timePtr) {
  struct tm* tmPtr = gmtime(timePtr);
  time_t t = *timePtr;
```

If there are locale settings, we add the time difference to the time.

```
  struct lconv* localeConvPtr = localeconv();

  if (localeConvPtr != NULL) {
    int timeZone = (tmPtr->tm_isdst == 1) ? localeConvPtr->summerTimeZone
                                          : localeConvPtr->winterTimeZone;
    t += (3600l * timeZone);
  }
```

When the time is modified with regard to anpotential time difference, weFinally, we call **gmtime** to do the actually calculate the time.

```
  return gmtime(&t);
}
```

The **difftime** function return the number of seconds between two times.

```
double difftime(time_t time1, time_t time2) {
  return (double) (time2 - time1);
}
```

14.12.2. Time Formatting

The **asctime** and **ctime** function below returns a string holding a time, we need the static global string **g_timeString** to hold the text.

```
static char g_timeString[256];
```

For the **asctime**, **ctime**, and **strftime** function we need the abbreviated and full names of the weekdays and months.

```
static char* g_shortDayList[] = {"Sun", "Mon", "Tue", "Wed",
                                 "Thu", "Fri", "Sat"};
static char* g_longDayList[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                                "Thursday", "Friday", "Saturday"};
static char* g_shortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
static char* g_longMonthList[] = {"January", "February", "March", "April",
                                  "May", "June", "July", "August",
                                  "September", "October",
                                  "November", "December"};
```

The **asctime** function returns a string with the given time in the format "Mon Jan 1 01:02:03 1970", given the time as a **tm** structure.

```
char* asctime(const struct tm* tp) {
  struct lconv* localeConvPtr = NULL;
```

If there are locale settings, we use its weekdays and months, otherwise we use the global static weekdays and months above.

```
  char **shortDayList, **shortMonthList;

  if ((localeConvPtr != NULL) && (localeConvPtr->shortDayList != NULL)) {
    shortDayList = localeConvPtr->shortDayList;
  }
  else {
    shortDayList = g_shortDayList;
  }

  if ((localeConvPtr != NULL) && (localeConvPtr->shortMonthList != NULL)) {
    shortMonthList = localeConvPtr->shortMonthList;
  }
  else {
    shortMonthList = g_shortMonthList;
  }
```

We call **sprintf** to format the text holding the time and date.

```
  sprintf(g_timeString, "%s %s %i %02i:%02i:%02i %i",
          shortDayList[tp->tm_wday], shortMonthList[tp->tm_mon],
          tp->tm_mday, tp->tm_hour, tp->tm_min,
          tp->tm_sec, tp->tm_year + 1900);
  return g_timeString;
}
```

The **ctime** function does also returns a string with the given time in the format "Mon Jan 1 01:02:03 1970". The difference between **asctime** above is that the time is given with the **time_t** type, the number of seconds from January 1, 1970, instead of the **tm** structure. We simply call **localtime** to receive the time on the struct **tm** form, and **asctime** to receive the text holding the time and date.

```
char* ctime(const time_t* time) {
  return asctime(localtime(time));
}
```

The **strftime** function formats the text of date and time with codes in accordance with the following table.

Code	**Meaning**	**Example**
a	Abbreviated weekday name	Mon
A	Full weekday name	Monday
b	Abbreviated month name	Jan
B	Full month name	January

c	Local date and time	01-02-03 04:05:06
d	Day of month	30
H	Hour, 24-hour clock (00-23)	23
I	Hour, 12-hour clock (01-12)	11
j	Day of year (001-366)	364
m	Month (01-12)	12
M	Minute (00-59)	59
p	AM or PM	PM
S	Second (00-59)	59
U	Week number (00-53)	53
w	Weekday (0-6, Sunday is 0)	6
W	Week number (00-53)	
x	Local date	01-02-03
X	Local time	04:05:06
y	Year without century (00-99)	20
Y	Year with century	2020
Z	Time zone name	
%	The per cent sign	%

The **s** parameter holds the final text, **smax** is the allowed maximal length of **s**, **fmt** holds the format text, and **tp** the date and time. The function returns the length of s, exclusive the terminating zero-character.

```
size_t strftime(char* result, size_t maxSize,
                const char* format, const struct tm* tp) {
```

The abbreviated and full names of the weekdays and months are obtained from the locale settings, if present. Otherwise, they are obtained from the global static arrays above.

```
  struct lconv* localeConvPtr = localeconv();
  char **shortDayList, **shortMonthList, **longDayList, **longMonthList;

  if ((localeConvPtr != NULL) && (localeConvPtr->shortDayList != NULL)) {
    shortDayList = localeConvPtr->shortDayList;
  }
  else {
    shortDayList = g_shortDayList;
  }

  if ((localeConvPtr != NULL) && (localeConvPtr->longDayList != NULL)) {
    longDayList = localeConvPtr->longDayList;
  }
  else {
    longDayList = g_longDayList;
  }

  if ((localeConvPtr != NULL) && (localeConvPtr->shortMonthList != NULL)) {
    shortMonthList = localeConvPtr->shortMonthList;
  }
  else {
    shortMonthList = g_shortMonthList;
  }

  if ((localeConvPtr != NULL) && (localeConvPtr->longMonthList != NULL)) {
    longMonthList = localeConvPtr->longMonthList;
  }
  else {
    longMonthList = g_longMonthList;
  }
```

We begin by clearing the **result** parameters.

```
  strcpy(result, "");
  { int index;
```

Remember that the **tm_wday** field starts with index zero on Sunday. In this function we also have the **weekDayStartMonday** value, that start with index zero on Monday. We initialize it to be one less than the **tm_wday** field.

```
    const int weekNumberStartSunday = getWeekNumber(tp);
    int weekNumberStartMonday = weekNumberStartSunday;
```

If the current day is a Sunday, we decrease the week number for the weeks that starts with Monday by one, since that week have not yet started.

```
    if (tp->tm_mday == 0) {
      --weekNumberStartMonday;
    }
```

The main loop of this function iterates through the format text, and replace every per cent code with the proper value.

```
    for (index = 0; format[index] != '\0'; ++index) {
      char add[20];

      if (format[index] == '%') {
        switch (format[++index]) {
          case 'a':
            strcpy(add, shortDayList[tp->tm_wday]);
            break;

          case 'A':
            strcpy(add, longDayList[tp->tm_wday]);
            break;

          case 'b':
            strcpy(add, shortMonthList[tp->tm_mon]);
            break;

          case 'B':
            strcpy(add, longMonthList[tp->tm_mon]);
            break;

          case 'c':
            sprintf(add, "%02i-%02i-%02i %02i:%02i:%02i",
                    1900 + tp->tm_year, tp->tm_mon + 1, tp->tm_mday,
                    tp->tm_hour, tp->tm_min, tp->tm_sec);
            break;

          case 'd':
            sprintf(add, "%02i", tp->tm_mday);
            break;

          case 'H':
            sprintf(add, "%02i", tp->tm_hour);
```

```
  break;

case 'I':
  sprintf(add, "%02i", (tp->tm_hour % 12));
  break;

case 'j':
  sprintf(add, "%03i", tp->tm_yday);
  break;

case 'm':
  sprintf(add, "%02i", tp->tm_mon + 1);
  break;

case 'M':
  sprintf(add, "%02i", tp->tm_min);
  break;

case 'p':
  sprintf(add, "%s", (tp->tm_hour < 12) ? "AM" : "PM");
  break;

case 'S':
  sprintf(add, "%02i", tp->tm_sec);
  break;

case 'U':
  sprintf(add, "%02i", weekNumberStartSunday);
  break;

case 'w':
  sprintf(add, "%02i", tp->tm_wday);
  break;

case 'W':
  sprintf(add, "%02i", weekNumberStartMonday);
  break;

case 'x':
  sprintf(add, "%02i:%02i:%02i", tp->tm_hour,
          tp->tm_min, tp->tm_sec);
  break;

case 'X':
  sprintf(add, "%02i:%02i:%02i", tp->tm_hour,
          tp->tm_min, tp->tm_sec);
  break;

case 'y':
  sprintf(add, "%02i", (tp->tm_year % 100));
  break;

case 'Y':
  sprintf(add, "%02i", 1900 + tp->tm_year);
  break;

case 'Z':
```

```
            strcpy(add, tp->tm_isdst ? "summer" : "winter");
            break;

          case '%':
            strcpy(add, "%");

          default:
            strcpy(add, "");
            break;
        }
      }
      else {
        add[0] = format[index];
        add[1] = '\0';
      }

      { int x = strlen(result), y = strlen(add);
        if ((x + y) < maxSize) {
          strcat(result, add);
          //printf("");
        }
        else {
          break;
        }
      }
    }
  }

  return strlen(result);
}

static int getWeekNumber(const struct tm* tp) {
  const long leapDays = (tp->tm_year - 69) / 4;
  const int totalDays = 365 * (tp->tm_year - 70) + leapDays;
  int weekDayJanuaryFirst;

  if (totalDays < 3) {
    weekDayJanuaryFirst = totalDays + 4;
  }
  else {
    weekDayJanuaryFirst = (totalDays - 3) % 7;
  }

  { int firstWeekSize = 7 - weekDayJanuaryFirst;

    if (tp->tm_yday < firstWeekSize) {
      return 0;
    }
    else {
      return ((tp->tm_yday - firstWeekSize) / 7) + 1;
    }
  }
}
```

A. The Preprocessor

The preprocessor processes the source code before the compiler. This chapter is divided into two parts, where the first part is a scanner and a parser, and the second part takes care of comments, string, and characters as well as handling macros and conditional programming.

A.1. The Expression Scanner and Parser

Before we consider the preprocessor itself, we need to find a way to decide the whether the expressions given in the #**if** or #**elif** directives equals zero. For that we need the expression parser, which can be viewed as a smaller version of the main parser of Chapter 2.3. We look into the grammar, scanner, and parser.

A.1.1. The Grammar

The grammar of an expression parser follows below. Similar to the expression part of the main parser of Chapter 2.3, we add a new grammar rule for each precedence level.

```
expression ::=
    logical_or_expression
  | logical_or_expression ? expression : expression

logical_or_expression ::=
    logical_and_expression
  | logical_or_expression || logical_and_expression

logical_and_expression ::=
    Bitwise_or_expression
  | bitwise_and_expression && bitwise_or_expression

bitwise_or_expression ::=
    bitwise_xor_expression
  | bitwise_or_expression | bitwise_xor_expression

bitwise_xor_expression ::=
    bitwise_and_expression
  | bitwise_xor_expression ^ bitwise_and_expression

bitwise_and_expression ::=
    equality_expression
  | bitwise_and_expression & equality_expression

equality_expression ::=
    relation_expression
  | equality_expression == relation_expression
  | equality_expression != relation_expression

relation_expression ::=
    shift_expression
  | relation_expression < shift_expression
  | relation_expression <= shift_expression
  | relation_expression > shift_expression
  | relation_expression >= shift_expression

shift_expression ::=
    add_expression
  | shift_expression << add_expression
  | shift_expression >> add_expression

add_expression ::=
    multiply_expression
  | add_expression + multiply_expression
  | add_expression - multiply_expression

multiply_expression ::=
    prefix_expression
  | multiply_expression * prefix_expression
  | multiply_expression / prefix_expression
  | multiply_expression % prefix_expression

prefix_expression ::=
    primary_expression
  | + prefix_expression
  | - prefix_expression
  | ~ prefix_expression
  | ! prefix_expression

primary_expression ::=
    identifier
  | value
  | defined identifier
  | defined ( identifier )
  | ( expression )
```

A.1.2. The Parser

The result of the parser is an integer value, floating values are not used by the preprocessor.

ExpressionParser.gppg

```
%namespace CCompiler_Exp
%partial

%using CCompiler;

%{
  // Empty.
%}

%token DEFINED ADD SUBTRACT MULTIPLY DIVIDE MODULO EQUAL NOT_EQUAL LESS_THAN
       LESS_THAN_EQUAL GREATER_THAN GREATER_THAN_EQUAL LEFT_SHIFT RIGHT_SHIFT
       QUESTION_MARK COLON LEFT_PARENTHESIS RIGHT_PARENTHESIS LOGICAL_OR
       LOGICAL_AND LOGICAL_NOT BITWISE_XOR BITWISE_OR BITWISE_AND BITWISE_NOT
       EOL

%union {
  public string name;
  public int integer_value;
}
```

```
%token <name> NAME
%token <integer_value> VALUE

%type <integer_value> expression logical_or_expression
                      logical_and_expression bitwise_or_expression
                      bitwise_xor_expression bitwise_and_expression
                      equality_expression relation_expression
                      shift_expression add_expression
                      multiply_expression prefix_expression
                      primary_expression

%start expression

%%
```

The result of the parsing is placed in the **PreProcessorResult** of the **Preprocessor** class. The return value of the parser is a Boolean value representing whether the expression complied with the grammar.

```
expression:
    logical_or_expression  {
      $$ = $1;
      Preprocessor.PreProcessorResult = $$;
    }
  | logical_or_expression QUESTION_MARK expression
                          COLON expression {
      $$ = ($1 != 0) ? $3 : $5;
      Preprocessor.PreProcessorResult = $$;
    };
```

The **logical_or_expression** and **logical_and_expression** return one or zero

```
logical_or_expression:
    logical_and_expression  {
      $$ = $1;
    }
  | logical_or_expression LOGICAL_OR logical_and_expression {
      $$ = (($1 != 0) || ($3 != 0)) ? 1 : 0;
    };

logical_and_expression:
    bitwise_or_expression  {
      $$ = $1;
    }
  | bitwise_and_expression LOGICAL_AND bitwise_or_expression {
      $$ = (($1 != 0) && ($3 != 0)) ? 1 : 0;
    };

bitwise_or_expression:
    bitwise_xor_expression  {
      $$ = $1;
    }
  | bitwise_or_expression BITWISE_OR bitwise_xor_expression {
      $$ = $1 | $3;
    };

bitwise_xor_expression:
    bitwise_and_expression  {
      $$ = $1;
    }
  | bitwise_xor_expression BITWISE_XOR bitwise_and_expression {
      $$ = $1 ^ $3;
    };

bitwise_and_expression:
    equality_expression {
      $$ = $1;
    }
  | bitwise_and_expression BITWISE_AND equality_expression {
      $$ = $1 & $3;
    };

equality_expression:
    relation_expression {
      $$ = $1;
    }
  | equality_expression EQUAL relation_expression {
      $$ = ($1 == $3) ? 1 : 0;
    }
  | equality_expression NOT_EQUAL relation_expression {
      $$ = ($1 != $3) ? 1 : 0;
    };

relation_expression:
    shift_expression  { $$ = $1; }
  | relation_expression LESS_THAN shift_expression {
      $$ = ($1 < $3) ? 1 : 0;
    }
  | relation_expression LESS_THAN_EQUAL shift_expression {
      $$ = ($1 <= $3) ? 1 : 0;
    }
  | relation_expression GREATER_THAN shift_expression {
      $$ = ($1 > $3) ? 1 : 0;
    }
  | relation_expression GREATER_THAN_EQUAL shift_expression {
      $$ = ($1 >=$3) ? 1 : 0;
    };

shift_expression:
    add_expression {
      $$ = $1;
    }
  | shift_expression LEFT_SHIFT add_expression {
      $$ = $1 << $3;
    }
  | shift_expression RIGHT_SHIFT add_expression {
      $$ = $1 >> $3;
    };

add_expression:
    multiply_expression {
      $$ = $1;
    }
  | add_expression ADD multiply_expression {
      $$ = $1 + $3;
    }
  | add_expression SUBTRACT multiply_expression {
      $$ = $1 - $3;
    };

multiply_expression:
    prefix_expression {
      $$ = $1;
    }
  | multiply_expression MULTIPLY prefix_expression {
      $$ = $1 * $3;
    }
  | multiply_expression DIVIDE prefix_expression {
      $$ = $1 / $3;
    }
  | multiply_expression MODULO prefix_expression {
      $$ = $1 % $3;
    };

prefix_expression:
    primary_expression {
      $$ = $1;
    }
  | ADD prefix_expression {
      $$ = $2;
    }
  | SUBTRACT prefix_expression {
      $$ = -$2;
    }
  | BITWISE_NOT prefix_expression {
      $$ = ~$2;
    }
```

An expression is considered to be true if it does not equal zero.

```
  | LOGICAL_NOT prefix_expression {
      $$ = ($2 == 0) ? 1 : 0;
    };
```

In the preprocessor directive, every identifier is treated as the value zero.

```
primary_expression:
    NAME {
      $$ = 1;
    }
  | VALUE {
      $$ = $1;
    }
```

The **defined** directive works both with and without parenthesis.

`#if defined (x)`	`#if defined x`
(a) With parenthesis	(b) Without parenthesis

ExpressionParser.gppg

```
  | DEFINED NAME {
      $$ = Preprocessor.MacroMap.ContainsKey($2) ? 1 : 0;
    }
  | DEFINED LEFT_PARENTHESIS NAME RIGHT_PARENTHESIS {
      $$ = Preprocessor.MacroMap.ContainsKey($3) ? 1 : 0;
    }
  | LEFT_PARENTHESIS expression RIGHT_PARENTHESIS {
      $$ = $2;
    };
```

A.1.3. The Scanner

As mention in Section 0, tokens are the smallest parts of the language. However, the parser cannot know that, for instance, a sequence of letters is an identifier, or that a sequence of digits is an integer value. Instead, that is the task of a scanner.

ExpressionScanner.gplex

```
%namespace CCompiler_Exp

%using CCompiler;

%{
  // Empty.
%}

OCTAL_VALUE (\+|\-)?[0][0-7]*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?
DECIMAL_VALUE (\+|\-)?[1-9][0-9]*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?
HEXADECIMAL_VALUE                                     (\+|\-)?[0][xX][0-9a-fA-
F]+([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?

NAME [a-zA-Z_][a-zA-Z0-9_]*
WHITE_SPACE [ \t\r\n\f]

%%
"defined" { return ((int) Tokens.DEFINED); }

"?" { return ((int) Tokens.QUESTION_MARK); }
":" { return ((int) Tokens.COLON); }

"||" { return ((int) Tokens.LOGICAL_OR); }
"&&" { return ((int) Tokens.LOGICAL_AND); }
"!" { return ((int) Tokens.LOGICAL_NOT); }

"&" { return ((int) Tokens.BITWISE_AND); }
"^" { return ((int) Tokens.BITWISE_XOR); }
"|" { return ((int) Tokens.BITWISE_OR); }
"~" { return ((int) Tokens.BITWISE_NOT); }

"==" { return ((int) Tokens.EQUAL); }
"!=" { return ((int) Tokens.NOT_EQUAL); }

"<"  { return ((int) Tokens.LESS_THAN); }
"<=" { return ((int) Tokens.LESS_THAN_EQUAL); }
">"  { return ((int) Tokens.GREATER_THAN); }
">=" { return ((int) Tokens.GREATER_THAN_EQUAL); }

"<<" { return ((int) Tokens.LEFT_SHIFT); }
">>" { return ((int) Tokens.RIGHT_SHIFT); }

"+" { return ((int) Tokens.ADD); }
"-" { return ((int) Tokens.SUBTRACT); }
```

```
"*" { return ((int) Tokens.MULTIPLY); }
"/" { return ((int) Tokens.DIVIDE); }
"%" { return ((int) Tokens.MODULO); }

"(" { return ((int) Tokens.LEFT_PARENTHESIS); }
")" { return ((int) Tokens.RIGHT_PARENTHESIS); }

"\0" { return ((int) Tokens.EOL); }
```

In case of an octal, decimal, or hexadecimal value, we just remove the letters u (unsigned), s (short), l (long), and x (hexadecimal value marker) from the text and let the Integer class interpret the value.

```
{NAME} {
  yylval.name = yytext;
  return ((int) Tokens.NAME);
}

{OCTAL_VALUE} {
  { string text = yytext.ToLower().Replace("u", "").Replace("s", "")
                                  .Replace("l", "");
    yylval.integer_value = Convert.ToInt32(text, 8);
    return ((int) Tokens.VALUE);
  }
}

{DECIMAL_VALUE} {
  { string text = yytext.ToLower().Replace("u", "").Replace("s", "")
                                  .Replace("l", "");
    yylval.integer_value = Convert.ToInt32(text, 10);
    return ((int) Tokens.VALUE);
  }
}

{HEXADECIMAL_VALUE} {
  { string text = yytext.ToLower().Replace("x", "").Replace("u", "").
                                   Replace("s", "").Replace("l", "");
    yylval.integer_value = Convert.ToInt32(text, 16);
    return ((int) Tokens.VALUE);
  }
}
```

It is important to detect white spaces, otherwise the user would not be allowed to add any spaces to the code.

```
{WHITE_SPACE} {
  if (yytext.Equals("\n")) {
    ++CCompiler_Main.Scanner.Line;
  }
}
```

If we detect a character we do not recognize, we just stop the execution with an error message.

```
. { Assert.Error(yytext, Message.Unknown_character); }
```

A.2. The Preprocessor Scanner and Parser

There is actually a second parser and scanner, that parsers the source code. The parser is very simple. Its only task is to define a set of terminals.

PreParser.gppg
```
%namespace CCompiler_Pre
%partial

%{
  // Empty.
%}

%union {
  public string name;
}

%token <name> NAME NAME_WITH_PARENTHESES STRING LEFT_PARENTHESIS
              RIGHT_PARENTHESIS COMMA SHARP DOUBLE_SHARP TOKEN END_OF_LINE

%start source_code_file

%%

source_code_file:
  /* Empty. */;

%%
```

The scanner is a little bit more complicated. Its task is to define a set of tokens. Similar to the scanner of Chapter 2, it defines a set of values tokens are to be used of the preprocessor. The idea is that we divide the

PreScanner.gplex
```
%namespace CCompiler_Pre

%{
  public static int NewlineCount = 0;
  public static bool Whitespace = false;
%}

OCTAL_VALUE (\+|\-)?[0][0-7]*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?
DECIMAL_VALUE (\+|\-)?[1-9][0-9]*([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])?
HEXADECIMAL_VALUE (\+|\-)?[0][xX][0-9a-fA-F]+
                  ([uU]|[sSlL]|[uU][sSlL]|[sSlL][uU])? // XXX
FLOATING_VALUE    (\+|\-)?(([0-9]+"."[0-9]*|"."[0-9]+)([eE][\+\-]?[0-9]+)?|[0-
9]+[eE][\+\-]?[0-9]+)([fF]|[lL])?
CHAR_VALUE \'("\n"|"\\\'"|[^\'])*\'
STRING_VALUE \"("\n"|"\\\""|[^\"])*\"
NAME [a-zA-Z_][a-zA-Z0-9_]*
WHITE_SPACE [ \t\r\n\f]
TOKEN [^a-zA-Z0-9()#,\"\' \t\r\n\f\0]+
%%

"(" { yylval.name = "("; return ((int) Tokens.LEFT_PARENTHESIS); }
")" { yylval.name = ")"; return ((int) Tokens.RIGHT_PARENTHESIS); }
"##" { yylval.name = "##"; return ((int) Tokens.DOUBLE_SHARP); }
```

```
"#"  { yylval.name = "#"; return ((int) Tokens.SHARP); }
"," { yylval.name = ","; return ((int) Tokens.COMMA); }
//"\0" { /*CCompiler_Pre.Scanner.NewlineCount = 0;*/ yylval.name = ""; return
((int) Tokens.EOL); }

{NAME} {
  yylval.name = yytext;
  return ((int) Tokens.NAME);
}

{OCTAL_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{DECIMAL_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{HEXADECIMAL_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{FLOATING_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{CHAR_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{STRING_VALUE} {
  yylval.name = yytext;
  return ((int) Tokens.STRING);
}

{TOKEN} {
  yylval.name = yytext;
  return ((int) Tokens.TOKEN);
}

{WHITE_SPACE} {
  if (yytext.Equals("\n")) {
    ++CCompiler_Pre.Scanner.NewlineCount;
  }

  CCompiler_Pre.Scanner.Whitespace = true;
}

. { CCompiler.Assert.Error("<" + yytext + ">",
    CCompiler.Message.Unknown_character); }
```

A.2.4. If-Else-Chain

An If-Else-Chain is a sequence of **if**, **elif**, **else**, and **endif** preprocessor directives. We use the **IfElseChain** class to keep track of the sequence. We need to keep track of three statues:

- **Former Status**. Has an earlier **if** or **elif** directive been evaluated to true? In that case, all succeeding **if**, **elif**, or **else** directive shall be evaluated to false.
- **Current Status**. Has the current **if**, **elif**, or **else** directive been evaluated to true? In that case the code of this directive is visible; that is, the code shall be included. If the directive has been evaluated to false, the code is invisible and shall be excluded.
- **Else Status**. A chain can only hold one **else** directive. If we encounter a second else directive, we report an error.

The **IfElseChain** class holds the former, current, and else status of an if-else-chain.

IfElseChain.cs

```
namespace CCompiler {
  class IfElseChain {
    bool m_formerStatus, m_currentStatus, m_elseStatus;

    public IfElseChain(bool formerStatus, bool currentStatus, bool elseStatus){
      m_formerStatus = formerStatus;
      m_currentStatus = currentStatus;
      m_elseStatus = elseStatus;
    }

    public bool FormerStatus {
      get { return m_formerStatus; }
    }

    public bool CurrentStatus {
      get { return m_currentStatus; }
    }

    public bool ElseStatus {
      get { return m_elseStatus; }
    }
  }
}
```

A.3. The Preprocessor

The preprocessor has several tasks:

- **Tri Graphs**. When C was originally introduced, some keyboards had a limited set of keys. Therefore, a special set of double question mark character sequence was introduced, which are replaced by modern equivalents.
- **Comments**. The line comments are removed, and each block comment is replaced by a blank character.
- **Slashes in strings and characters**. Each backslash is processed and transformed into a regular character. Thereafter, to ease the macro expansion later we transform each character into the format of a backslash followed by three octal digits.

- **Include files**. The system include files (encapsulated by ‘<’ and ‘>’) and internal include files (encapsulated by quotes) are read and included in the final code.
- **Conditional programming**. The #**if**, #**ifdef**, #**ifndef**, #**elif**, and #**endif** directives are processed.
- **Macro expansion**. The source code is traversed, and each macro is expanded. Since the contents of each character and string constant has been translated into slash codes, we do not need to check whether the macro to be expanded is part of a character of string.
- **String sequences**. Finally, every sequence of string is concatenated into one string. For instance, the sequence "ab" "cd" "ef" is concatenated into "abcded".

The first phase of the preprocessor is to read the source file and place in a text buffer (a **StringBuilder** object), and then replace the tri graph sequences, remove the comments and replace every character in string and characters with its equivalent backslash code. Then the buffer is divided into lines and each line starting with a preprocessor directive is evaluated and every line is search for macros to expand.

Preprocessor.cs
```
using System;
using System.IO;
using System.Text;
using System.Collections.Generic;

namespace CCompiler {
  public class Preprocessor {
    private IDictionary<string,Macro> m_macroMap =
              new Dictionary<string,Macro>();
    private Stack<FileInfo> m_includeStack = new Stack<FileInfo>();
    private ISet<FileInfo> m_includeSet = new HashSet<FileInfo>();
    private Stack<IfElseChain> m_ifElseChainStack = new Stack<IfElseChain>();
    private StringBuilder m_outputBuffer = new StringBuilder();

    public string PreprocessedCode {
      get { return m_outputBuffer.ToString(); }
    }

    public ISet<FileInfo> IncludeSet {
      get { return m_includeSet; }
    }
```
We add the **__LINUX__** macro for the compiler to compile the source code in accordance with the Linux environment, and the **__WINDOWS__** macro for the Windows environment. There are several parts of the standard library source code that are different depending on whether the macro is present, see Chapter 14.

```
    public Preprocessor(FileInfo file) {
      if (Start.Linux) {
        m_macroMap.Add("__LINUX__", new Macro(0, new List<Token>(), null));
      }

      if (Start.Windows) {
        m_macroMap.Add("__WINDOWS__", new Macro(0, new List<Token>(), null));
      }

      DoProcess(file);
      Assert.Error(m_ifElseChainStack.Count == 0, Message.
                   If___ifdef____or_ifndef_directive_without_matching_endif);
    }
```

```
public void DoProcess(FileInfo file) {
  StreamReader streamReader = new StreamReader(file.FullName);
  StringBuilder inputBuffer =
    new StringBuilder(streamReader.ReadToEnd());
  streamReader.Close();

  CCompiler_Main.Scanner.Path = file;
  CCompiler_Main.Scanner.Line = 2;
  GenerateTriGraphs(inputBuffer);
  TraverseBuffer(inputBuffer);
  List<string> lineList =
    GenerateLineList(inputBuffer.ToString());

  CCompiler_Main.Scanner.Line = 1;
  int stackSize = m_ifElseChainStack.Count;
  TraverseLineList(lineList);
  Assert.Error(m_ifElseChainStack.Count ==
    stackSize, Message.Unbalanced_if_and_endif_directive_structure);
}
```

A.3.5. Tri Graphs

A tri graph character sequence is three-character sequence that starts with two question marks and represent another character. It is a remain from old times, when the keyboards hold a reduced set of characters. The following table shows the tri graphs character sequences and their modern equivalences:

Tri-Graph Character Sequence	Modern Equivalence
??=	#
??/	\
??′	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

We need to replace the tri graphs sequences with their modern equivalences, which is easy to do since we do not need to take in consideration whether the sequences are placed inside strings, characters, or comments.

```
private static IDictionary<string,string> m_triGraphMap =
  new Dictionary<string,string>()
    {{"??=", "#"}, {"??/", "\\"}, {"??\'", "^"},
     {"??(", "["}, {"??)", "]"}, {"??!", "|"},
     {"??<", "{"}, {"??>", "}"}, {"??-", "~"}};

public void GenerateTriGraphs(StringBuilder buffer) {
  foreach (KeyValuePair<string,string> pair in m_triGraphMap) {
    buffer.Replace(pair.Key, pair.Value);
  }
```

We also replace all newline and return pairs with single newlines, and we replace all white spaces (form feed, return, horizontal tabulator, and vertical tabulator) with single spaces.

```
  buffer.Replace("\r\n", "\n");
```

```
      buffer.Replace("\f", " ");
      buffer.Replace("\r", " ");
      buffer.Replace("\t", " ");
      buffer.Replace("\v", " ");
```

Finally, we also need to merge lines that ends with two backslashes. For instance:

```
#define min(x,y) \\
  (((x) < (y)) ? \\
   (x) : (y))
```

(a) Before

```
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

(b) After

We replace all backslashes and newlines at the end of a line with returns. Remember that we have removed all returns above, so the only returns left are the ones marking a removed newline. We keep the returns in order to add blank lines after the merged lines, which we in turn do to keep the correct line numbers in macros and error messages.

```
      buffer.Replace("\\\n", "\r");
    }
```

A.3.6. Comments, Strings, and Characters

The next step is to take care of block comments ('/*' to '*/') and line comments ('//' to the end of the line), strings, and characters. Since they may be nested, they need to be evaluated in the same phase. The idea is that we go through the buffer, and when we find the beginning of a line comment, a block comment, a string, or a character we iterate through the buffer until we find its end. To make the inspection of the source code buffer easier we start by adding three zero characters, which are removed at the end of the method.

```
    public void TraverseBuffer(StringBuilder buffer) {
      buffer.Append("\0");
      for (int index = 0; index < buffer.Length; ++index) {
```

When we encounter the beginning of a block comment ('/*'), we continue until we find the end of the comment ('*/'). If we instead find the end of the buffer ('\0') the comment is unterminated, which result in an error message. Each character is replaced by a space character. However, newlines ('\n') are not replaced, in order for the line count to work properly and for future error messages to report the correct line.

```
        if ((buffer[index] == '/') && (buffer[index + 1] == '*')) {
          buffer[index++] = ' ';
          buffer[index++] = ' ';

          for (; true; ++index) {
            if (buffer[index] == '\0') {
              Assert.Error(Message.Unfinished_block_comment);
            }
            else if ((buffer[index] == '*') && (buffer[index + 1] == '/')) {
              buffer[index++] = ' ';
              buffer[index] = ' ';
              break;
            }
            else if (buffer[index] == '\n') {
              ++CCompiler_Main.Scanner.Line;
            }
            else {
              buffer[index] = ' ';
```

```
      }
    }
  }
```

When we encounter a line comment ('//'), we continue until we find the end of the line ('\n') or the end of the buffer ('\0'), and replace every preceding character up until then with a blank.

```
  else if ((buffer[index] == '/') && (buffer[index + 1] == '/')) {
    buffer[index++] = ' ';
    buffer[index++] = ' ';

    for (; true; ++index) {
      if ((buffer[index] == '\n') || (buffer[index] == '\0')) {
        break;
      }
      else {
        buffer[index] = ' ';
      }
    }
  }
```

When we encounter a single-quote ('\''), which marks the beginning of a character, we iterate through the buffer until we find another finishing single-quote. We also need to look out for the end-of-buffer ('\0') or newline ('\n') characters, which in both cases result in error messages.

```
  else if ((buffer[index] == '\'')) {
    ++index;

    for (; true; ++index) {
      if (index == buffer.Length) {
        Assert.Error(Message.Unfinished_character);
      }
      else if (buffer[index] == '\n') {
        Assert.Error(Message.Newline_in_character);
      }
```

If we encounter a backslash ('\') followed by a single-quote, we simply ignore it since that combination holds a single-quote rather than the end of the string.

```
      else if ((buffer[index] == '\\') && (buffer[index] == '\'')) {
        ++index;
      }
      else if (buffer[index] == '\'') {
        break;
      }
    }
  }
```

In the same way, when we encounter a double-quote, which marks the beginning of a string ('\"'), we iterate through the buffer until we find another finishing double-quote. We also need to look out for the end-of-buffer ('\0') or newline ('\n') characters.

```
  else if ((buffer[index] == '\"')) {
    ++index;

    for (; true; ++index) {
      if (buffer[index] == '\0') {
        Assert.Error(Message.Unfinished_string);
```

```
          }
          else if (buffer[index] == '\n') {
            Assert.Error(Message.Newline_in_string);
          }
```

If we encounter a backslash ('\') followed by a double quote, we simply ignore it since that combination holds a double-quote rather than the end of the string.

```
          else if ((buffer[index] == '\\') && (buffer[index] == '\"')) {
            ++index;
          }
          else if (buffer[index] == '\"') {
            break;
          }
        }
      }
```

Finally, when we encounter a newline in the buffer, outside of a comment, string, or character, we increase the line count in order for future error messages to report the correct line.

```
      else if (buffer[index] == '\n') {
        ++CCompiler_Main.Scanner.Line;
      }
    }

    buffer.Remove(buffer.Length - 1, 1);
  }
```

A.3.7. The Line List

When the buffer has been modified in accordance with the methods above, the next step is to transform it into a list of lines to enable the processing of the preprocessor directives. The transformation is simple, we just use the **Split** method in the **String** C# standard class and call **Trim** on each line to remove trailing and ending whitespaces.

```
  private List<string> GenerateLineList(string text) {
    List<string> lineList = new List<string>(text.Split('\n'));
```

In order to keep the newlines of the text, we iterate through the line list, add the lines without the returns and add blank line for each return.

```
    List<string> mergeList = new List<string>();

    foreach (string line in lineList) {
      int returnCount = line.Split('\r').Length - 1;
      mergeList.Add(line.Replace('\r', ' ').Trim());

      for (int count = 1; count < returnCount; ++count) {
        mergeList.Add("");
      }
    }
```

The next step is to merge all lines between preprocessor directives, because a macro call may cover several lines.

```
    { int index = 0;
      List<string> resultList = new List<string>();

      while (index < mergeList.Count) {
        if (mergeList[index].StartsWith("#")) {
          resultList.Add(mergeList[index++]);
        }
        else {
          bool first = true;
          StringBuilder buffer = new StringBuilder();
```

When we encounter a line that is not preprocessor directive, we iterate and merge the following lines until we reach the end of the line list or a preprocessor directive.

```
          for (; (index < mergeList.Count) &&
                 !mergeList[index].StartsWith("#"); ++index) {
            buffer.Append((first ? "" : "\n") + mergeList[index]);
            first = false;
          }

          resultList.Add(buffer.ToString());
        }
      }

      return resultList;
    }
  }
```

The **Scan** method scans a line and convert it into a list of tokens.

```
  private List<Token> Scan(string text) {
    byte[] byteArray = Encoding.ASCII.GetBytes(text);
    MemoryStream memoryStream = new MemoryStream(byteArray);
    CCompiler_Pre.Scanner scanner = new CCompiler_Pre.Scanner(memoryStream);
    List<Token> tokenList = new List<Token>();

    while (true) {
      CCompiler_Pre.Tokens tokenId = (CCompiler_Pre.Tokens) scanner.yylex();
```

When we encounter the end of the line, we add an end-of-line token to the list, and break the iteration. However, the end-of-line token does not generate any text at the end.

```
      if (tokenId == CCompiler_Pre.Tokens.EOF) {
        tokenList.Add(new Token(CCompiler_Pre.Tokens.END_OF_LINE, ""));
        break;
      }

      tokenList.Add(new Token(tokenId, scanner.yylval.name));
    }

    memoryStream.Close();
    return tokenList;
  }
```

The **TokenListToString** method converts a token list to a string, with a space between the tokens. It is called when errors are reported.

```
  private string TokenListToString(List<Token> tokenList) {
    StringBuilder buffer = new StringBuilder();

    foreach (Token token in tokenList) {
      buffer.Append(((buffer.Length > 0) ? " " : "") + token.ToString());
    }
```

```
      return buffer.ToString();
    }
```

The **CloneList** method return a copy of the token list.

```
    private List<Token> CloneList(List<Token> tokenList) {
      List<Token> resultList = new List<Token>();

      foreach (Token token in tokenList) {
        resultList.Add((Token) token.Clone());
      }

      return resultList;
    }
```

The **TraverseLineList** method traverses the line list and calls appropriate methods for each preprocessor directive.

```
    public void TraverseLineList(List<string> lineList) {
      foreach (string line in lineList) {
        List<Token> tokenList = Scan(line);

        if (tokenList[0].Id == CCompiler_Pre.Tokens.SHARP) {
          Token secondToken = tokenList[1];

          if (secondToken.Id == CCompiler_Pre.Tokens.NAME) {
            string secondTokenName = (string) secondToken.Value;

            if (secondTokenName.Equals("ifdef")) {
              DoIfDefined(tokenList);
            }
            else if (secondTokenName.Equals("ifndef")) {
              DoIfNotDefined(tokenList);
            }
            else if (secondTokenName.Equals("if")) {
              DoIf(tokenList);
            }
            else if (secondTokenName.Equals("elif")) {
              DoElseIf(tokenList);
            }
            else if (secondTokenName.Equals("else")) {
              DoElse(tokenList);
            }
            else if (secondTokenName.Equals("endif")) {
              DoEndIf(tokenList);
            }
```

The preprocessor directives that do not concern conditional programming are inspected if the code is visible only.

```
            else if (IsVisible()) {
              if (secondTokenName.Equals("include")) {
                DoInclude(tokenList);
              }
              else if (secondTokenName.Equals("define")) {
                DoDefine(tokenList);
```

```
          }
          else if (secondTokenName.Equals("undef")) {
            DoUndef(tokenList);
          }
          else if (secondTokenName.Equals("line")) {
            DoLine(tokenList);
          }
          else if (secondTokenName.Equals("error")) {
            Assert.Error(TokenListToString
                         (tokenList.GetRange(1, tokenList.Count - 1)));
          }
        }
      }

      AddNewlinesToBuffer(tokenList);
    }
```

If the line is not a preproeccor directive, we check if the line is visible; that is, not enclosed in a **#if**, **#ifdef**, **#ifndef**, or **#ifelse** directive that has (directly or indirectly) been evaluated to false. If it is visible, we search for macros and replace them with their proper values, concatenates tokens, merge string, and add the tokens of the line to the buffer.

```
    else {
      if (IsVisible()) {
        SearchForMacros(tokenList, new Stack<string>());
        ConcatTokens(tokenList);
        MergeStrings(tokenList);
        AddTokenListToBuffer(tokenList);
      }
```

If the line is not visible, we just add its newlines to the buffer.

```
      else {
        AddNewlinesToBuffer(tokenList);
      }
    }
  }
}
```

The **AddTokenListToBuffer** method iterates through the token list and adds their text to the output buffer.

```
private void AddTokenListToBuffer(List<Token> tokenList) {
  foreach (Token token in tokenList) {
    m_outputBuffer.Append(token.ToNewlineString() + token.ToString());
  }

  m_outputBuffer.Append("\n");
  ++CCompiler_Main.Scanner.Line;
}
```

The **AddNewlinesToBuffer** method iterates through the token list and adds their newlines the output buffer.

```
private void AddNewlinesToBuffer(List<Token> tokenList) {
  foreach (Token token in tokenList) {
    m_outputBuffer.Append(token.ToNewlineString());
    CCompiler_Main.Scanner.Line += token.GetNewlineCount();
  }
```

```
    m_outputBuffer.Append("\n");
    ++CCompiler_Main.Scanner.Line;
  }
```

The **IsVisible** method checks whether we are in a visible part of the source code; that is, a part that has not been earlier omitted by a **#if**, **#ifdef**, **#ifndef**, **#elif**, or **#else** preprocessor directive. We use the **m_ifElseChainStack** stack since conditional programming can be nested. If the source code is not visible at any nesting level (that is, anywhere in the stack), the area is not visible, and we return false.

```
  private bool IsVisible() {
    foreach (IfElseChain ifElseChain in m_ifElseChainStack) {
      if (!ifElseChain.CurrentStatus) {
        return false;
      }
    }
```

The area is visible if we have not found any level where it is invisible.

```
    return true;
  }
```

A.3.8. Lines

The **DoLine** method handles the **#line** directive by setting the **Compiler_Main**.**Path** and **Compiler_Main**.**Line** fields and returns a text with the path and line that starts and ends with dollar signs ('$'). It is allowed to omit the path, the path and line, but not only the line. If the path or line is omitted, they are not set. For instance:

```
#line 100 C:\Temp\Test.c  $C:\Temp\Test.c,100$
#line 200                 $previous path,200$
#line                     $previous path,previous line$
```

(a) C Code (b) Preprocessed code

```
  private void DoLine(List<Token> tokenList) {
    int listSize = tokenList.Count;
```

The size of the token list may wary between two and four, inclusively. If the size is two

```
    Assert.Error((listSize >= 2) && (listSize <= 4),
                 TokenListToString(tokenList),
                 Message.Invalid_preprocessor_directive);
```

In case of at least three tokens, we have a line number, which we try to parse and assign the **CCompiler_Main.Scanner**.**Line** field.

```
    if (listSize >= 3) {
      string lineText = (string) tokenList[2].Value;
      Assert.Error(int.TryParse(lineText, out CCompiler_Main.Scanner.Line),
                                lineText, Message.Invalid_line_number);
    }
```

In case of at four tokens, we have a path, which we assign the **CCompiler_Main.Scanner**.**Path** field.

```
    if (listSize == 4) {
      CCompiler_Main.Scanner.Path =
        new FileInfo((string) tokenList[3].Value);
    }
```

When we have obtained the path and line, we add the marker to the output buffer.

```
      m_outputBuffer.Append(Symbol.SeparatorId + CCompiler_Main.Scanner.Path +
                            "," + CCompiler_Main.Scanner.Line +
                            Symbol.SeparatorId + "\n");
    }
```

A.3.9. Includes

There are two kinds of include files: system files ('<' and '>') and internal files ('\"'). The systems files are included from the path given by **Main.IncludePath**, and the internal files are included locally. The **Main.IncludeStack** is used to make prevent circular inclusion.

The **CCompiler_Main.Scanner**.**Path** and **CCompiler_Main.Scanner.Line** fields are temporary replaced by the path and line of the included file. After the include line has been read, the original path and line is added to the included source code, in the dollar sign format (for example: **$C:\Temp\Test.c,100$**). The method returns the source code of the included file, which replaces the original include preprocessor.

```
    private void DoInclude(List<Token> tokenList) {
      FileInfo includeFile = null;

      if ((tokenList[2].Id == CCompiler_Pre.Tokens.STRING) &&
          (tokenList[3].Id == CCompiler_Pre.Tokens.END_OF_LINE)) {
        string text = tokenList[2].ToString();
        string file = text.ToString().Substring(1, text.Length - 1);
        includeFile = new FileInfo(Start.SourcePath + file);
      }
      else {
        StringBuilder buffer = new StringBuilder();

        foreach (Token token in tokenList.GetRange(2, tokenList.Count - 2)) {
          buffer.Append(token.ToString());
        }

        string text = buffer.ToString();

        if (text.StartsWith("<") && text.EndsWith(">")) {
          string file = text.ToString().Substring(1, text.Length - 2);
          includeFile = new FileInfo(Start.SourcePath + file);
        }
        else {
          Assert.Error(TokenListToString(tokenList),
                       Message.Invalid_preprocessor_directive);
        }
      }

      Assert.Error(!m_includeStack.Contains(includeFile),
                   includeFile.FullName, Message.Repeted_include_statement);
      m_includeStack.Push(includeFile);
      m_includeSet.Add(includeFile);
      FileInfo oldPath = CCompiler_Main.Scanner.Path;
      int oldLine = CCompiler_Main.Scanner.Line;
      CCompiler_Main.Scanner.Path = includeFile;
      CCompiler_Main.Scanner.Line = 1;
      m_outputBuffer.Append(Symbol.SeparatorId + CCompiler_Main.Scanner.Path +
                            ",0" + Symbol.SeparatorId + "\n");
      DoProcess(includeFile);
      CCompiler_Main.Scanner.Line = oldLine;
      CCompiler_Main.Scanner.Path = oldPath;
      m_outputBuffer.Append(Symbol.SeparatorId + CCompiler_Main.Scanner.Path +
                            "," + (CCompiler_Main.Scanner.Line - 1) +
                            Symbol.SeparatorId + "\n");
      m_includeStack.Pop();
    }
```

A.3.10. Macros

The **Macro** class keep track of a macro, a macro has a possible empty list of parameters.

Macro.cs

```
using System.Linq;
using System.Collections.Generic;

namespace CCompiler {
  public class Macro {
    private int m_parameters;
    private List<Token> m_tokenList;
```

A macro holds a number of parameters. Unlike function definitions, we are not interested of the types or names of the parameters, only the number of parameters.

```
    public Macro(int parameters, List<Token> tokenList) {
      m_parameters = parameters;
      m_tokenList = new List<Token>(tokenList);
    }

    public int Parameters {
      get {return m_parameters;}
    }

    public List<Token> TokenList {
      get {return m_tokenList;}
    }

    public IDictionary<int,int> IndexToParamMap {
      get {return m_indexToParamMap;}
    }

    public override int GetHashCode() {
      return base.GetHashCode();
    }
```

Two macros are considered equal if the hold the same parameters.

```
    public override bool Equals(object obj) {
      if (obj is Macro) {
        Macro macro = (Macro) obj;
        return (m_parameters == macro.m_parameters) &&
               (m_tokenList.SequenceEqual(macro.m_tokenList));
      }

      return false;
    }
  }
}
```

A.3.11. Tokens

The preprocessor scanner returns tokens of the **Token** class. The token holds an identity, a value and a count of newlines.

Token.cs
```
using System.Text;

namespace CCompiler {
  public class Token {
    private CCompiler_Pre.Tokens m_id;
    private object m_value;
    private int m_newlineCount;

    public Token(CCompiler_Pre.Tokens id, object value) {
      m_id = id;
      m_value = value;
      m_newlineCount = CCompiler_Pre.Scanner.NewlineCount;
      CCompiler_Pre.Scanner.NewlineCount = 0;
    }

    public Token(CCompiler_Pre.Tokens id, object value, int newlineCount) {
      m_id = id;
      m_value = value;
      m_newlineCount = newlineCount;
    }

    public void AddNewlineCount(int newlineCount) {
      m_newlineCount += newlineCount;
    }

    public object Clone() {
      Token token = new Token(m_id, m_value);
      token.m_newlineCount = 0;
      return token;
    }

    public CCompiler_Pre.Tokens Id {
      get { return m_id; }
      set { m_id = value; }
    }

    public object Value {
      get { return m_value; }
      set { m_value = value; }
    }

    public int GetNewlineCount() {
      return m_newlineCount;
    }

    public string ToNewlineString() {
      StringBuilder buffer = new StringBuilder();

      if (m_newlineCount > 0) {
        for (int count = 0; count < m_newlineCount; ++count) {
          buffer.Append('\n');
        }
      }
      else {
        buffer.Append(' ');
      }

      return buffer.ToString();
    }

    public void ClearNewlineCount() {
      m_newlineCount = 0;
    }

    public override int GetHashCode() {
      return base.GetHashCode();
    }

    public override bool Equals(object obj) {
      if (obj is Token) {
        Token token = (Token) obj;
        return (m_id == token.m_id) &&
                m_value.Equals(token.m_value);
      }

      return false;
    }

    public override string ToString() {
      switch (m_id) {
        case CCompiler_Pre.Tokens.NAME_WITH_PARENTHESES:
          return m_value.ToString() + " (";

        default:
          return m_value.ToString();
      }
    }
  }
}
```

A.3.12. Define

The **DoDefine** method splits the line into the name of the macro, a potential parameter list, and the macro list, made up by the part of the token list following the name and potential parameter list.

Preprocessor.cs

```
    public void DoDefine(List<Token> tokenList) {
      Assert.Error((tokenList[2].Id == CCompiler_Pre.Tokens.NAME) ||
                   (tokenList[2].Id ==
                    CCompiler_Pre.Tokens.NAME_WITH_PARENTHESES),
                   TokenListToString(tokenList),
                   Message.Invalid_define_directive);
```

We have two cases: macro definition with and without parameters. In the case without parameters, we just create a macro without parameters and with the rest of the token list as its token list.

```
      Macro macro;
      if (tokenList[2].Id == CCompiler_Pre.Tokens.NAME) {
        macro = new Macro(0, tokenList.GetRange(3, tokenList.Count - 3),null);
      }
```

In case of a macro with parameters, there is a little bit more complicated. First, we need to identify the parameters, then we need to replace the parameter names in the token list with parameter markers.

```
      else {
        int tokenIndex = 3, paramIndex = 0;
        IDictionary<string,int> paramMap = new Dictionary<string,int>();
```

We iterate through the token list until we find the finishing right parenthesis, and stores the parameter names in **paramMap**.

```
        while (true) {
          Token nextToken = tokenList[tokenIndex++];
```

The next token after the left parantheses must ba a name.

```
          Assert.Error(nextToken.Id == CCompiler_Pre.Tokens.NAME,
                       nextToken.ToString(),
                       Message.Invalid_macro_definition);
```

If the name already occurs in the parameter list, we report an error.

```
          string paramName = (string) nextToken.Value;
          Assert.Error(!paramMap.ContainsKey(paramName),
                       paramName, Message.Repeated_macro_parameter);
```

We add each parameter name with its index in the list to **paramMap**.

```
          paramMap.Add(paramName, paramIndex++);
```

The next token after the name must be a comma or a left parenthesis.

```
          nextToken = tokenList[tokenIndex++];
          if (nextToken.Id == CCompiler_Pre.Tokens.COMMA) {
            // Empty.
          }
          else if (nextToken.Id == CCompiler_Pre.Tokens.RIGHT_PARENTHESIS) {
            break;
          }
          else {
            Assert.Error(nextToken.ToString(),
                         Message.Invalid_macro_definition);
          }
        }
```

When the parameter list has been stored, we extract the list of tokens following the parameters.

```
        List<Token> macroList =
          tokenList.GetRange(tokenIndex, tokenList.Count - tokenIndex);
```

We iterate through the macro list and replace each name-with-parameter token with a name and a parameter token.

```
        for (int index = macroList.Count - 1; index >= 0; --index) {
          if (macroList[index].Id ==
              CCompiler_Pre.Tokens.NAME_WITH_PARENTHESES) {
            macroList[index].Id = CCompiler_Pre.Tokens.NAME;
            Token newToken =
              new Token(CCompiler_Pre.Tokens.LEFT_PARENTHESIS, "(");
            macroList.Insert(index + 1, newToken);
          }
        }
```

We iterate through the macro list one more time and for each name we check if it is present in the parameter map. If it is present in the parameter list, we add its indexes in the macro list and in the parameter list to **indexToParamMap**.

```
      IDictionary<int,int> indexToParamMap = new Dictionary<int,int>();
      for (int index = 0; index < macroList.Count; ++index) {
        Token macroToken = macroList[index];

        if (macroToken.Id == CCompiler_Pre.Tokens.NAME) {
          string macroName = (string) macroToken.Value;

          if (paramMap.ContainsKey(macroName)) {
            indexToParamMap[index] = paramMap[macroName];
          }
        }
      }
```

Finally, we create a macro with the number of parameters, the macro list and the **indexToParam** map.

```
      macro = new Macro(paramMap.Count, macroList, indexToParamMap);
    }
```

If the macro is not yet added to the macro map, we simply add it.

```
    string name = (string) tokenList[2].Value;
    if (!m_macroMap.ContainsKey(name)) {
      m_macroMap.Add(name, macro);
    }
```

It is allowed to add a new identical macro with the same name. Therefore, if the macro is already added to the macro map, we report an error it they are not equal.

```
    else {
      Assert.Error(m_macroMap[name].Equals(macro), name,
                   Message.Invalid_macro_redefinition);
    }
  }
```

The **DoUndef** method removes a macro from the **MacroMap** map. If the macro does not exists a warning is given.

```
  public void DoUndef(List<Token> tokenList) {
    Assert.Error((tokenList[2].Id == CCompiler_Pre.Tokens.NAME) &&
                 (tokenList[3].Id == CCompiler_Pre.Tokens.END_OF_LINE),
                 TokenListToString(tokenList),
                 Message.Invalid_undef_directive);
    string name = tokenList[2].ToString();
    Assert.Error(m_macroMap.Remove(name),
                 name, Message.Macro_not_defined);
  }
```

A.3.13. Conditional Programming

With conditional programming, it is possible to exclude part of the source code. The **doIf** method uses the parser of Section A.1. . It calls **parseExpression**, which creates a scanner and parser, parse the line, and returns a Boolean value that is true in case of a non-zero value.

```
private void DoIf(List<Token> tokenList) {
  bool result = ParseExpression(TokenListToString
                   (tokenList.GetRange(2, tokenList.Count - 2)));
  m_ifElseChainStack.Push(new IfElseChain(result, result, false));
}

public static object PreProcessorResult;

private bool ParseExpression(string line) {
  int result = 0;

  try {
    byte[] byteArray = Encoding.ASCII.GetBytes(line);
    MemoryStream memoryStream = new MemoryStream(byteArray);
    CCompiler_Exp.Scanner expressionScanner =
      new CCompiler_Exp.Scanner(memoryStream);
    CCompiler_Exp.Parser expressionParser =
      new CCompiler_Exp.Parser(expressionScanner, m_macroMap);
    Assert.Error(expressionParser.Parse(), Message.Preprocessor_parser);
    result = (int) PreProcessorResult;
    memoryStream.Close();
  }
  catch (Exception exception) {
    Console.Out.WriteLine(exception.StackTrace);
    Assert.Error(line, Message.Invalid_expression);
  }

  return (result != 0);
}
```

The **DoIfDefined** and **DoIfNotDefined** checks if the macro is defined or not, respectively, and add the result to the **m_ifElseChainStack** stack.

```
private void DoIfDefined(List<Token> tokenList) {
  Assert.Error((tokenList[2].Id == CCompiler_Pre.Tokens.NAME) &&
               (tokenList[3].Id == CCompiler_Pre.Tokens.END_OF_LINE),
               TokenListToString(tokenList),
               Message.Invalid_preprocessor_directive);
  bool result = m_macroMap.ContainsKey((string) tokenList[2].Value);
  m_ifElseChainStack.Push(new IfElseChain(result, result, false));
}

private void DoIfNotDefined(List<Token> tokenList) {
  Assert.Error((tokenList[2].Id == CCompiler_Pre.Tokens.NAME) &&
               (tokenList[3].Id == CCompiler_Pre.Tokens.END_OF_LINE),
               TokenListToString(tokenList),
               Message.Invalid_preprocessor_directive);
  bool result = !m_macroMap.ContainsKey((string)tokenList[2].Value);
  m_ifElseChainStack.Push(new IfElseChain(result, result, false));
}
```

The **DoIfElseIf** method is a little bit more complicated. It checks whether there is a preceding **#if** directive (**m_ifElseChainStack** is not empty). Thereafter, it looks up the status of the if-stack. The stack value is a if-else-chain, the first value indicates whether the current if-status is true, and the second value gives whether there has been an earlier true if-status. The third value is of type **IfStatus**, that can hold the value **If** and **Else**. If the value is **Else**, and #**elseif** has already occurred, and another on is not allowed. Finally, if this if-chain has ever been true (**lastTrue** is true) is does not matter whether the condition of directive is true, the new status is false. If not, we consider if this value is true. In both cases, we push the result on the stack.

```
private void DoElseIf(List<Token> tokenList) {
  Assert.Error(m_ifElseChainStack.Count > 0, Message.
   Elif_directive_without_preceeding_if____ifdef____or_ifndef_directive);
  IfElseChain ifElseChain = m_ifElseChainStack.Pop();

  Assert.Error(!ifElseChain.ElseStatus,
               Message.Elif_directive_following_else_directive);

  if (ifElseChain.FormerStatus) {
    m_ifElseChainStack.Push(new IfElseChain(true, false, false));
  }
  else {
    bool result =
      ParseExpression(TokenListToString
                     (tokenList.GetRange(2, tokenList.Count - 2)));
    m_ifElseChainStack.Push(new IfElseChain(result, result, false));
  }
}
```

The **DoElse** method is a little bit easier. It checks whether there has ever been a true part of the if-chain. If not, it pushes true on the stack.

```
private void DoElse(List<Token> tokenList) {
  Assert.Error(m_ifElseChainStack.Count > 0, Message.
   Else_directive_without_preceeding_if____ifdef____or_ifndef_directive);
  Assert.Error(tokenList[2].Id == CCompiler_Pre.Tokens.END_OF_LINE,
               TokenListToString(tokenList),
               Message.Invalid_preprocessor_directive);

  IfElseChain ifElseChain = m_ifElseChainStack.Pop();
  Assert.Error(!ifElseChain.ElseStatus,
               Message.Else_directive_after_else_directive);

  bool formerStatus = ifElseChain.FormerStatus;
  m_ifElseChainStack.Push(new IfElseChain(!formerStatus,
                                          !formerStatus, true));
}
```

Finally, the **DoEndIf** just pops the if-stack, unless it is not already empty. If it is empty, we report an error.

```
private void DoEndIf(List<Token> tokenList) {
  Assert.Error(m_ifElseChainStack.Count > 0, Message.
  Endif_directive_without_preceeding_if____ifdef____or_ifndef_directive);
  Assert.Error(tokenList[2].Id == CCompiler_Pre.Tokens.END_OF_LINE,
               tokenList[2].ToString(),
               Message.Invalid_preprocessor_directive);
  m_ifElseChainStack.Pop();
}
```

A.3.14. Macro Expansion

In the regular source code (lines not starting with '#'), we need to replace macro calls with their resulting bodies. We also need to recursively replace macro calls in the bodies. However, unlike function, recursive macro calls are not allowed.

We go through the text, and look for identifiers, we do not have to worry about identifier in strings or characters, since every letter or underline (an identifier must begin with a letter or underline) in strings or characters has been changed to a three-octal-digit backslash code by **SlashToChar** in Section 2.3.

When we find and identifier that is a stored macro name (either in m**_specialMacroSet** or **MacroMap**), we scan the parameters if the name is followed by a left parenthesis, each parameter is recursively searched for macros. We then look up the macro body and replace its marked parameter locations with the parameters. Finally, we replace the macro call with its body. The **nameStack** stack is used to prevent recursive macro calls.

```
    private void SearchForMacros(List<Token> tokenList,
                                 Stack<string> nameStack) {
      for (int index = 0; index < tokenList.Count; ++index) {
        Token thisToken = tokenList[index];
        CCompiler_Main.Scanner.Line += thisToken.GetNewlineCount();
```

In case of a macro name without parameters.

```
        if (thisToken.Id == CCompiler_Pre.Tokens.NAME) {
          string name = (string) thisToken.Value;
          int beginNewlineCount = thisToken.GetNewlineCount();

          if (!nameStack.Contains(name) && m_macroMap.ContainsKey(name)) {
            Macro macro = m_macroMap[name];
            Assert.Error(macro.Parameters == 0, name, Message.
                         Invalid_number_of_parameters_in_macro_call);
            List<Token> cloneListX = CloneList(macro.TokenList);
```

We must search for macros recursivly in its macro list.

```
            nameStack.Push(name);
            SearchForMacros(cloneListX, nameStack);
            nameStack.Pop();

            tokenList.RemoveAt(index);
            tokenList.InsertRange(index, cloneListX);
            tokenList[index].AddNewlineCount(beginNewlineCount);
            index += cloneListX.Count - 1;
          }
```

If the name does not occours in the macro map, we check for special macros.

```
          else {
            switch (name) {
```

The **__STDC__** macro is replaced with 1, since this compilar complies with the C standard.

```
              case "__STDC__":
                tokenList[index] =
                  new Token(CCompiler_Pre.Tokens.TOKEN, 1, beginNewlineCount);
                break;
```

The **__FILE__** macro is replaced with the current file name.

```
              case "__FILE__": {
                   string text = "\"" + CCompiler_Main.Scanner.Path
                                .FullName.Replace("\\", "\\\\") + "\"";
                   tokenList[index] = new Token(CCompiler_Pre.Tokens.TOKEN,
                                                  text, beginNewlineCount);
                }
                break;
```

The **__LINE__** macro is replaced with the current line number.

```
              case "__LINE__":
                tokenList[index] =
                  new Token(CCompiler_Pre.Tokens.TOKEN,
                           CCompiler_Main.Scanner.Line, beginNewlineCount);
                break;
```

The **__DATE__** macro is replaced with the current date.

```
              case "__DATE__": {
                  string text = "\"" + DateTime.Now.ToString("MMMM dd yyyy") +
                                "\"";
                  tokenList[index] = new Token(CCompiler_Pre.Tokens.TOKEN,
                                                 text, beginNewlineCount);
                }
                break;
```

Finally, the **__TIME__** macro is replaced with the current time.

```
              case "__TIME__": {
                  string text = "\"" + DateTime.Now.ToString("HH:mm:ss") +
                                "\"";
                  tokenList[index] = new Token(CCompiler_Pre.Tokens.TOKEN,
                                                 text, beginNewlineCount);
                }
                break;
            }
          }
        }
```

In case of a name followed by a parameter list, it is a little bit more complicated. First, we need to identify the parameters, and then we replace the parameters in macro list with the actual parameters.

```
        else if (thisToken.Id == CCompiler_Pre.Tokens.NAME_WITH_PARENTHESES) {
          string name = (string) thisToken.Value;
          int beginNewlineCount = thisToken.GetNewlineCount();
```

Each actual parameter may be made up of several tokens, or no tokens at all. Therefore, we store each actual parameter in a token list.

```
          if (!nameStack.Contains(name) && m_macroMap.ContainsKey(name)) {
            int countIndex = index + 1, level = 1, totalNewlineCount = 0;
            List<Token> subList = new List<Token>();
            List<List<Token>> mainList = new List<List<Token>>();
```

We iterate trought the token list and stores the actual parameter sub lists in the **mainList** list.

```
            while (true) {
              Token nextToken = tokenList[countIndex];
              int newlineCount = nextToken.GetNewlineCount();
              totalNewlineCount += newlineCount;
              CCompiler_Main.Scanner.Line += newlineCount;
              nextToken.ClearNewlineCount();

              Token token = tokenList[countIndex];
              Assert.Error(token.Id != CCompiler_Pre.Tokens.END_OF_LINE,
                           Message.Invalid_end_of_macro_call);

              switch (token.Id) {
                case CCompiler_Pre.Tokens.LEFT_PARENTHESIS:
                  ++level;
                  subList.Add(token);
                  break;

                case CCompiler_Pre.Tokens.RIGHT_PARENTHESIS:
                  if ((--level) > 0) {
                    subList.Add(token);
                  }
                  break;

                default:
                  if ((level == 1) &&
                      (token.Id == CCompiler_Pre.Tokens.COMMA)) {
                    Assert.Error(subList.Count > 0, name,
                                  Message.Empty_macro_parameter);
                    SearchForMacros(subList, nameStack); // XXX
                    mainList.Add(subList);
                    subList = new List<Token>();
                  }
                  else {
                    subList.Add(token);
                  }
                  break;
              }

              if (level == 0) {
                Assert.Error(subList.Count > 0, name,
                              Message.Empty_macro_parameter_list);
                mainList.Add(subList);
                break;
              }

              ++countIndex;
            }
```

When we have identified the parameter lists.

```
          Macro macro = m_macroMap[name];
          Assert.Error(macro.Parameters == mainList.Count, name,
                       Message.Invalid_number_of_parameters_in_macro_call);

          List<Token> cloneListX = CloneList(macro.TokenList);
          IDictionary<int,int> indexToParamMap = macro.IndexToParamMap;

          for (int macroIndex = (cloneList.Count - 1);
                macroIndex >= 0; --macroIndex) {
            Token macroToken = cloneList[macroIndex];
```

For each index in the token list, we check it it is reprsented in the **indexToParamMap** map. In that case, we look up the parameter list and replace the token with the list.

```
              int paramIndex;
              if (indexToParamMap.TryGetValue(macroIndex, out paramIndex)) {
                cloneList.RemoveAt(macroIndex);
                List<Token> replaceList = CloneList(mainList[paramIndex]);
```

If a parameter is preceded by a sharp ('#'), the parameter list shall be enclosed in double-qoutes.

```
                if ((macroIndex > 0) && (cloneList[macroIndex - 1].Id ==
                                         CCompiler_Pre.Tokens.SHARP)) {
                  string text = "\"" + TokenListToString(replaceList) + "\"";
                  cloneList.Insert(macroIndex,
                              new Token(CCompiler_Pre.Tokens.STRING, text));
                  cloneList.RemoveAt(--macroIndex);
                }
                else {
                  cloneList.InsertRange(macroIndex, replaceList);
                }
              }
            }
```

We must search for macros recursively in the replaced list.

```
            nameStack.Push(name);
            SearchForMacros(cloneListX, nameStack);
            nameStack.Pop();

            tokenList.RemoveRange(index, countIndex - index + 1);
            tokenList.InsertRange(index, cloneListX);
            tokenList[index].AddNewlineCount(beginNewlineCount);
            tokenList[index +
                      cloneListX.Count].AddNewlineCount(totalNewlineCount);
            index += cloneListX.Count - 1;
          }
        }
      }
    }
```

A.3.15. Concatenate Tokens

The **ConcatTokens** method concatenates tokens with a double sharp token ('##') in between them.

`abc ## 123`	`abc123`
(a) Before	(b) After

```
    private void ConcatTokens(List<Token> tokenList) {
      for (int index = 1; index < (tokenList.Count - 1); ++index) {
        Token thisToken = tokenList[index];

        if (thisToken.Id == CCompiler_Pre.Tokens.DOUBLE_SHARP) {
          Token prevToken = tokenList[index - 1],
                nextToken = tokenList[index + 1];
```

If at least one of the tokens is a string, we just add the newline count. If both tokens are string, they will be merged in a later phase. If just one of them is a string, there is no meaning in merging them.

```
          if ((prevToken.Id == CCompiler_Pre.Tokens.STRING) ||
              (nextToken.Id == CCompiler_Pre.Tokens.STRING)) {
            nextToken.AddNewlineCount(thisToken.GetNewlineCount());
            tokenList.RemoveAt(index);
          }
          else {
            prevToken.Value = prevToken.ToString() + nextToken.ToString();
            prevToken.AddNewlineCount(thisToken.GetNewlineCount() +
                                      nextToken.GetNewlineCount());
            tokenList.RemoveAt(index);
            tokenList.RemoveAt(index);
          }
        }
      }
    }
```

A.3.16. String Merging

When the all the macros have been expanded, it is finally time to check whether there are pairs of string constants that need to be merged:

(a) Before	(b) After
`"Hello" "World"`	`"HelloWorld"`

```
    private void MergeStrings(List<Token> tokenList) {
      for (int index = (tokenList.Count - 2); index >= 0; --index) {
        Token thisToken = tokenList[index], nextToken = tokenList[index + 1];

        if ((thisToken.Id == CCompiler_Pre.Tokens.STRING) &&
            (nextToken.Id == CCompiler_Pre.Tokens.STRING)) {
          string thisText = thisToken.ToString(),
                 nextText = nextToken.ToString();
          thisToken.Value =
            thisText.ToString().Substring(0, thisText.Length - 1) +
            nextText.ToString().Substring(1, nextText.Length - 1);
          thisToken.AddNewlineCount(nextToken.GetNewlineCount());
          tokenList.RemoveAt(index + 1);
        }
      }
    }
  }
}
```

B. The Register Set

The architecture holds a set of overlapping registers.

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
al	ah		
ax			
eax			
rax			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
bl	bh		
bx			
ebx			
rbx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
cl	ch		
cx			
ecx			
rcx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
dl	ah		
dx			
edx			
rdx			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
si			
esi			
rsi			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
di			
edi			
rdi			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
bp			
ebp			
rbp			

Bits 0-7	Bits 8-15	Bits 16-31	Bits 32-63
sp			
esp			
rsp			

C. The C Grammar

This grammar of the C compiler of this book is based on (and to a large extent identical with) the grammar defined in the *American National Standard for Information systems – Programming Language C, X3.159-1989* standard, which is described in the second edition of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

C.1. The Preprocessor Grammar

The following grammar is implemented in GPPG in Chapter 1.3.

```
control_line ::=
    # define identifier token-sequence
  | # define identifier( identifier, … , identifier ) token-sequence
  | # undef identifier
  | # include <filename>
  | # include "filename"
  | # line constant
  | # line constant "filename"
  | # error optional_token_sequence
  | # pragma optional_token_sequence
  | #
  | preprocessor_conditional

optional_token_sequence ::=
    /* empty */
  | token_sequence

preprocessor_conditional ::=
    if-line text optional_elif-parts optional_else_line # endif

if-line ::=
    # if constant_expression
  | # ifdef identifier
  | # ifndef identifier

optional_elif-parts ::=
    /* empty */
  | elif-part optional_elif-parts

elif-part ::=
    # elif constant_expression text

optional_else_part ::=
    /* empty */
  | #else text
```

C.1.1. The Language Grammar

The following grammar is implemented in GPPG in Chapter 0. It is the same grammar, written in one block.

```
source_code_file ::=
    external_declaration
  | source_code_file external_declaration

external_declaration ::=
    function_definition
 |  declaration

function_definition ::=
    direct_declarator optional_declaration_list ( optional_statement_list )
  | declaration_specifier_list direct_declarator optional_declaration_list
    ( optional_statement_list )

optional_declaration_list ::=
    /* empty */
  | optional_declaration_list declaration

declaration ::=
    declaration_specifier_list ;
  | declaration_specifier_list declarator_list ;

declaration_specifier_list ::=
    declaration_specifier
  | declaration_specifier_list declaration_specifier

declaration_specifier ::=
    const | volatile | auto | register | static | extern | typedef | void
  | char | wchar_t | short | int | long | float | double | signed | unsigned
  | struct_union_specifier | enum_specifier | typedef_name

struct_union_specifier ::=
    struct_union optional_identifier ( declaration_list )
  | struct_union identifier

struct_union ::=
    struct | union

optional_identifier ::=
    /* empty */
  | identifier

declaration_list ::=
    declaration
  | declaration_list declaration

enum_specifier ::=
    enum optional_identifier ( enum_list )
  | enum identifier

enum_list ::=
    enum
  | enum_list , enum

enum ::=
    identifier
  | identifier = const_expression
```

```
declarator_list ::=
    initialization_bitfield_direct_declarator
  | declarator_list , initialization_bitfield_direct_declarator

initialization_bitfield_direct_declarator ::=
    direct_declarator
  | direct_declarator = initializer
  | direct_declarator : const_expression
  | : const_expression

direct_declarator ::=
    optional_pointer_list direct_direct_declarator

direct_direct_declarator ::=
    identifier
  | ( direct_declarator )
  | direct_direct_declarator [ optional_const_expression ]
  | direct_direct_declarator ( parameter_variadic_list )
  | direct_direct_declarator ( optional_identifier_list )

optional_pointer_list ::=
    /* empty */
  | pointer_list

pointer_list ::=
    pointer
  | pointer_list pointer

pointer ::=
    * | * declaration_specifier_list

parameter_variadic_list ::=
    parameter_list
  | parameter_list , ...

parameter_list ::=
    parameter_declaration
  | parameter_list , parameter_declaration

parameter_declaration ::=
    declaration_specifier_list
  | declaration_specifier_list direct_declarator
  | declaration_specifier_list abstract_direct_declarator

optional_identifier_list ::=
    /* empty */
  | identifier_list

identifier_list ::=
    identifier
  | identifier_list , identifier

initializer_list ::=
    initializer
  | initializer_list , initializer

initializer ::=
    assignment_expression
  | ( initializer_list )
  | ( initializer_list , )

type_name ::=
    declaration_specifier_list
  | declaration_specifier_list abstract_direct_declarator

abstract_direct_declarator ::=
    pointer_list
  | optional_pointer_list direct_abstract_direct_declarator

direct_abstract_direct_declarator ::=
    ( abstract_direct_declarator )
  | [ optional_const_expression ]
  | direct_abstract_direct_declarator [ optional_const_expression ]
  | ( optional_parameter_variadic_list )
  | direct_abstract_direct_declarator ( optional_parameter_variadic_list )

optional_statement_list ::=
    /* empty */
  | statement_list

statement_list ::=
    statement
  | statement_list statement

statement ::=
    open_statement
  | closed_statement

open_statement ::=
    if ( expression ) statement
  | if ( expression ) closed_statement else open_statement
  | switch ( expression ) open_statement
  | case const_expression : open_statement
  | while ( expression ) open_statement
  | for ( optional_expression ; optional_expression ;
          optional_expression ) open_statement
  | identifier : open_statement

closed_statement ::=
    if ( expression ) closed_statement else closed_statement
  | switch ( expression    ) closed_statement
  | while ( expression ) closed_statement
  | for ( optional_expression ; optional_expression ;
          optional_expression ) closed_statement
  | declaration
  | case const_expression : closed_statement
  | default : closed_statement
  | optional_expression ;
  | ( optional_statement_list )
  | do statement while ( expression ) ;
  | goto identifier ;
  | continue ;
  | break ;
  | return optional_expression ;
  | load_register ( identifier , expression ) ;
  | store_register ( identifier , expression ) ;
  | store_flagbyte ( expression ) ;
  | jump_register ( identifier ) ;
  | interupt ( const_expression ) ;

optional_expression ::=
    /* empty */
  | expression

expression ::=
    assignment_expression
  | expression , assignment_expression

assignment_expression ::=
    conditional_expression
  | prefix_expression assignment_operator assignment_expression

assignment_operator ::=
    = | *= | /= | %=  | += | -= | <<= | <<= | &= | |= | ^=

conditional_expression ::=
    logical_or_expression
  | logical_or_expression
    ? expression : conditional_expression

optional_const_expression ::=
    /* empty */
  | const_expression

const_expression ::=
    conditional_expression

logical_or_expression ::=
    logical_and_expression
  | logical_or_expression || logical_and_expression

logical_and_expression ::=
    bitwise_or_expression
  | bitwise_and_expression && bitwise_or_expression

bitwise_or_expression ::=
    bitwise_xor_expression
  | bitwise_or_expression | bitwise_xor_expression

bitwise_xor_expression ::=
    bitwise_and_expression
  | bitwise_xor_expression ^ bitwise_and_expression

bitwise_and_expression ::=
    equality_expression
  | bitwise_and_expression & equality_expression

equality_expression ::=
    relation_expression
  | equality_expression equality_operator relation_expression
```

```
equality_operator ::=
    == | !=

relation_expression ::=
    shift_expression
  | relation_expression relation_operator shift_expression

relation_operator ::=
    < | <= | > | >=

shift_expression ::=
    add_expression
  | shift_expression shift_operator add_expression

shift_operator ::=
    << | >>

add_expression ::=
    multiply_expression
  | add_expression binary_add_operator multiply_expression

binary_add_operator ::=
    + | -

multiply_expression ::=
    type_cast_expression
  | multiply_expression multiply_operator type_cast_expression

multiply_operator ::=
    * | / | %

type_cast_expression ::=
    prefix_expression
  | ( type_name ) type_cast_expression

prefix_expression ::=
    postfix_expression
  | increment_operator prefix_expression
  | prefix_add_operator type_cast_expression
  | ! type_cast_expression
  | ~ type_cast_expression
  | & type_cast_expression
  | * type_cast_expression
  | sizeof prefix_expression
  | sizeof ( type_name )

prefix_add_operator ::=
    + | -

increment_operator ::=
    ++ | --

postfix_expression ::=
    primary_expression
  | postfix_expression [ expression ]
  | postfix_expression ( optional_argument_expression_list )
  | postfix_expression . identifier
  | postfix_expression -> identifier
  | postfix_expression increment_operator

primary_expression ::=
    identifier
  | value
  | ( expression )

optional_argument_expression_list ::=
    /* empty */
  | argument_expression_list

argument_expression_list ::=
    assignment_expression
  | argument_expression_list , assignment_expression
```

D. The Gardens Point Tools

In this chapter, we use the Gardens Point LEX (GPLEX) scanner generator and the Gardens Point Parser Generator (GPPG) to construct a parser and scanner for a simple demonstration programming language.

D.1.1. The Language

A program is made up by a non-empty sequence of **statements**, each terminated by a semicolon. There are five kinds of statements:

- **Read**. A non-empty sequence of variables is assigned values read from the input stream, with an optional prompt.
- **Write**. The non-empty sequence of values is written to the output stream, with an optional text.
- **Assign**. A variable is assigned the value of an expression.

The mathematical constants **pi** and **e** are stored in the variable map when the execution starts. An expression can be made up of the four rules of arithmetic, parentheses as well as the functions **sin**, **cos**, **tan**, **log**, **exp**, **log10**, **exp10**, and **sqrt** (square root). Division by zero, invalid function arguments, or invalid input generate error messages.

Below is an example:

```
// The area and circumference of a circle
read "Input the radius of a circle: ", radius;
assign area = pi * radius * radius;
assign circumference = 2 * pi * radius;
write "The area and circumference are: ", area, circumference;
newline;

// Fahrenheit to Celsius
read "Input a temperature in degrees Fahrenheit: ", fahrenheit;
assign celsius = (fahrenheit - 32) / 1.8;
write "In Celsius degrees: ", celsius;
newline;

// Pythagorean Theorem
read "Input the two cathetuses of a right-angled triangle: ", a, b;
assign c = sqrt(a * a + b * b);
write "The hypothenuse is: ", c;
```

When executed (the underlined text represent input values):

```
Input the radius of a circle: 10
The area and circumference are: 314.159265358979, 62.8318530717959

Input a temperature in degrees Fahrenheit: 212
In Celsius degrees: 100

Input the two cathetuses of a right-angled triangle: 3,4
The hypothenuse is: 5
```

D.1.2. The Grammar

Every programming language has a syntax, which may be defined by a **grammar**. The grammar of our language is given below. A grammar is made up by a set of *rules*, among which one is the start rule (in this case **statement_list**). The grammar rules for our language follows below, the vertical bar at the left of the rules can be read as "or". The first rule **statement_list** can be read as "a statement list is a statement or a statement followed by a statement list."

```
statement_list ::=
    statement
  | statement statement_list

statement ::=
    assign identifier EQUAL expression ;
  | read optional_output identifier_list ;
  | write optional_output expression_list ;
  | newline ;

optional_output ::=
    /* Empty. */
  | string ,

identifier_list ::=
    identifier
  | identifier , identifier_list

expression_list ::=
    expression
  | expression , expression_list

expression ::=
    binary_expression
  | expression + binary_expression
  | expression - binary_expression

binary_expression ::=
    unary_expression
  | binary_expression * unary_expression
  | binary_expression / unary_expression

unary_expression ::=
    primary_expression
  | + unary_expression
  | - unary_expression
  | sin unary_expression
  | cos unary_expression
  | tan unary_expression
  | log unary_expression
  | exp unary_expression
  | log10 unary_expression
  | exp10 unary_expression
  | sqrt unary_expression

primary_expression ::=
    ( expression )
  | identifier
  | value
```

The words and character in boldface is the end products of the grammar. The rules shall be applied in such a way that finally only tokens remain. The first line of the example in the previous section are made up of the following tokens:

read string , identifier ;

D.1.3. GPPG

GPPG is a tool based on the classic parser generator Yacc. The name Yacc is an abbreviation for Yet Another Compiler-Compiler. However, it is pronounced as the animal yak. Even though it is possible to write a parser in pure C#, it is easier to use a tool like GPPG. Simply put, GPPG takes a grammar as input and generators C# code as output. However, it does not only check whether the input string comply with the grammar, each rule is also allowed to perform **actions**, defined by C# code.

The parser is divided into two parts: one part stating the rules and tokens of the grammar, and one part defining the grammar rules. The tokens written in boldface in the previous section are written in capital letters in GPPG.

Parser.gppg

```
%namespace Calculator
%partial

%{
  // Empty.
%}
```

The following tokens do not hold attributes.

```
%token ASSIGN READ WRITE NEWLINE EQUAL PLUS MINUS TIMES DIVIDE SIN COS TAN
       LOG EXP LOG10 EXP10 SQRT LEFT_PAREN RIGHT_PAREN COMMA SEMICOLON
```

The STRING, IDENTIFIER, and VALUE **tokens have attributes.**

```
%union {
  public string name;
  public string text;
  public double value;
  public List<string> nameList;
  public List<double> valueList;
}

%token <text> TEXT
%token <name> NAME
%token <value> VALUE

%type <nameList> name_list
%type <valueList> expression_list
%type <value> expression binary_expression unary_expression primary_expression
```

There are also the rules statement_list, statement, and optional_output **in the rules section below. However, they do not return values and do not need to be defined in this section.**

```
%%

statement_list:
    statement
  | statement statement_list;
```

The right-hand side are numbered from one, and the attribute values are accessible with the dollar-notation. In the rule below, the value of the identifier is stored in $2 and the value of the expression is stored in $4.

```
statement:
    ASSIGN NAME EQUAL expression SEMICOLON {
      MainX.VariableMap[$2] = $4;
    }

  | READ optional_output name_list SEMICOLON {
      try {
        string buffer = Console.In.ReadLine();
        string[] textArray = buffer.Split(',');

        if ($3.Count != textArray.Length) {
          Console.Error.WriteLine("Invalid number of values.");
          Environment.Exit(-1);
        }

        for (int index = 0; index < $3.Count; ++index) {
          string name = $3[index], text = textArray[index];
          double value = double.Parse(text);
          MainX.VariableMap[name] = value;
        }
      }
      catch (Exception exception) {
        Console.Error.WriteLine("Invalid input: " + exception.ToString());
        Environment.Exit(-1);
      }
    }

  | WRITE optional_output expression_list SEMICOLON {
      bool first = true;
      foreach (double value in $3) {
        Console.Out.Write((first ? "" : ", ") + value);
        first = false;
      }

      Console.Out.WriteLine();
    }

  | NEWLINE SEMICOLON {
      Console.Out.WriteLine();
    };

optional_output:
    /* Empty. */
  | TEXT COMMA {
      Console.Out.Write($1);
    };
```

The value of the rule is stored in $$. However, the $$ value is always a reference to an Object. Therefore, we need to transform it into a list before we can call the add method.

```
name_list:
    NAME {
      $$ = new List<string>();
```

```
        $$.Add($1);
      }
    | name_list COMMA NAME {
        $$ = $1;
        $$.Add($3);
      };

expression_list:
      expression {
        $$ = new List<double>();
        $$.Add($1);
      }
    | expression_list COMMA expression {
        $$ = $1;
        $$.Add($3);
      };

expression:
      binary_expression {
        $$ = $1;
      }
    | expression PLUS binary_expression {
        $$ = $1 + $3;
      }
    | expression MINUS binary_expression {
        $$ = $1 - $3;
      };

binary_expression:
      unary_expression {
        $$ = $1;
      }
    | binary_expression TIMES unary_expression {
        $$ = $1 * $3;
      }
    | binary_expression DIVIDE unary_expression {
        if ($3 == 0) {
          Console.Error.WriteLine("Division by Zero.");
          Environment.Exit(-1);
        }

        $$ = $1 / $3;
      };

unary_expression:
      primary_expression {
        $$ = $1;
      }
    | PLUS unary_expression {
        $$ = $2;
      }
    | MINUS unary_expression {
        $$ = -$2;
      }
    | SIN unary_expression {
        $$ = Math.Sin($2);
      }
```

```
  | COS unary_expression {
      $$ = Math.Cos($2);
    }
  | TAN unary_expression {
      $$ = Math.Tan($2);
    }
  | LOG unary_expression {
      if ($2 <= 0) {
        Console.Error.WriteLine("Logarithm of Non-Positive Value.");
        Environment.Exit(-1);
      }

      $$ = Math.Log($2);
    }
  | EXP unary_expression {
      $$ = Math.Exp($2);
    }
  | LOG10 unary_expression {
      if ($2 <= 0) {
        Console.Error.WriteLine("Logarithm of Non-Positive Value.");
        Environment.Exit(-1);
      }

      $$ = Math.Log10($2);
    }
  | EXP10 unary_expression {
      $$ = Math.Pow(10, $2);
    }
  | SQRT unary_expression {
      if ($2 < 0) {
        Console.Error.WriteLine("Square Root of Negativ Value.");
        Environment.Exit(-1);
      }

      $$ = Math.Sqrt($2);
    };

primary_expression:
    LEFT_PAREN expression RIGHT_PAREN {
      $$ = $2;
    }
  | NAME {
      if (MainX.VariableMap.ContainsKey($1)) {
        $$ = MainX.VariableMap[$1];
      }
      else {
        Console.Error.WriteLine("Unknown Name: \"" + $1 + "\".");
        Environment.Exit(-1);
      }
    }
  | VALUE {
      $$ = $1;
    };

%%
```

D.1.4. JPlex

For the parser of the previous section to work properly it also needs a scanner that identifies the tokens. JPlex is a scanner-generator, which task is to identify and transform groups of characters into tokens.

When it comes to complicated tokens, JFlex applies regular expressions. Some characters have special meanings in accordance with the following table. However, their special status can be canceled by preceeding it with a backslash.

Character	**Meaning**
(and)	Grouping subexpressions
[and]	Any character within the brackets
[^ and]	Any character **except** the characters within the brackets
*	Zero or more characters
+	One or moree characters
.	Any character except new line

Scanner.jplex

```
%namespace Calculator

%{
  // Empty.
%}
```

In the scanner code below, a line comment is two slashes followed by zero or more occurrences of any character except newline. Since both the slash has special meaning we must insert a backslash before it for it to mean just a slash.

```
LINE_COMMENT \/\/.*
```

A string is a double quotation mark, followed by zero or more occurrences of any character (except double quotation mark) followed by a double quotation mark.

```
TEXT \"[^\"]*\"
```

An identifier is a capital or small letter, or a underline, followed by zero or more capital or small letters, or digits, or underlines.

```
NAME [a-zA-Z_][a-zA-Z0-9_]*
```

A value is as least one digit, potentially followed by more digits, a dot, and even more digits.

```
VALUE [0-9]+|([0-9]+\.[0-9]+)
```

A white space is either a space, a tabulator, a return, a newline, or a form-feed.

```
WHITE_SPACE [ \t\r\n\f]

%%

"assign"  { return ((int) Tokens.ASSIGN);  }
```

```
"read"    { return ((int) Tokens.READ);    }
"write"   { return ((int) Tokens.WRITE);   }
"newline" { return ((int) Tokens.NEWLINE); }
"sin"     { return ((int) Tokens.SIN);     }
"cos"     { return ((int) Tokens.COS);     }
"tan"     { return ((int) Tokens.TAN);     }
"log"     { return ((int) Tokens.LOG);     }
"exp"     { return ((int) Tokens.EXP);     }
"log10"   { return ((int) Tokens.LOG10);   }
"exp10"   { return ((int) Tokens.EXP10);   }
"sqrt"    { return ((int) Tokens.SQRT);    }

"=" { return ((int) Tokens.EQUAL);       }
"+" { return ((int) Tokens.PLUS);        }
"-" { return ((int) Tokens.MINUS);       }
"*" { return ((int) Tokens.TIMES);       }
"/" { return ((int) Tokens.DIVIDE);      }
"(" { return ((int) Tokens.LEFT_PAREN);  }
")" { return ((int) Tokens.RIGHT_PAREN); }
"," { return ((int) Tokens.COMMA);       }
";" { return ((int) Tokens.SEMICOLON);   }
```

The **yytext** method return the scanned text, which is useful with identifiers and strings.

```
{NAME} {
  yylval.name = yytext;
  return ((int) Tokens.NAME);
}

{TEXT} {
  yylval.text = yytext.Substring(1, yytext.Length - 2);
  return ((int) Tokens.TEXT);
}

{VALUE} {
  yylval.value = Double.Parse(yytext);
  return ((int) Tokens.VALUE);
}

{LINE_COMMENT} {
  // Empty.
}

{WHITE_SPACE} {
  // Empty.
}

<<EOF>> {
  return ((int) Tokens.END_OF_LINE);
}
```

Finally, the last part of the scanner is the dot rule, which catch the case not caught by any of the previous tokens (equivalent to **default** in a **switch** statement). In that case, we just exit the execution with an error message.

```
. {
  Console.Error.WriteLine("Unknown character: \'" + yytext + "\'.");
  Environment.Exit(-1);
}
```

D.1.5. Main

GPPG and JPLEX generates the **Parser** and **Scanner** classes. The main class creates a scanner object which is used by the parser object. The **parse** method is called with invoke the parsing. The input is read from **BufferedReader**, the output is written to **OutputStream**, error messages are written to **ErrorStream**, and **VariableMap** is used to keep track of the variables read and assigned by the program.

Main.cs

```
using System;
using System.IO;
using System.Collections.Generic;

namespace Calculator {
  class MainX {
    public static Dictionary<string,double> VariableMap =
      new Dictionary<string,double>();

    static void Main(string[] args) {
      if (args.Length != 1) {
        Console.Out.WriteLine("Usage: calculator inputfile");
      }

      { System.Globalization.CultureInfo customCulture =
         (System.Globalization.CultureInfo)
         System.Threading.Thread.CurrentThread.CurrentCulture.Clone();
        customCulture.NumberFormat.NumberDecimalSeparator = ".";
        System.Threading.Thread.CurrentThread.CurrentCulture = customCulture;
      }

      VariableMap["pi"] = Math.PI;
      VariableMap["e"] = Math.E;

      string s = "abcd", t = "\"\"";
      string u = s.Substring(1, s.Length - 2);
      string v = s.Substring(1, t.Length - 2);

      try {
        FileStream file = new FileStream(args[0], FileMode.Open);
        Scanner scanner = new Scanner(file);
        Parser parser = new Parser(scanner);
        parser.Parse();
      }
      catch (Exception exception) {
        Console.Out.WriteLine(exception.ToString());
      }
    }
  }
```

The Parser **class is necessary to make the parser work with the scanner. It works as link between the scanner and the parser**

```
  public partial class Parser :
         QUT.Gppg.ShiftReduceParser<ValueType, QUT.Gppg.LexLocation> {
    public Parser(Scanner scanner)
     :base(scanner) {
      // Empty.
    }
  }
}
```

Finally, below follows a small script file that generates the C# source code for the parser and scanner.

Parser.bat

```
@C:
@cd C:\Users\Stefan\Documents\Calculator_CS
@"C:\gppg-distro-1_5_2\binaries\Gppg" /gplex Parser.gppg > Parser.cs
@"C:\gppg-distro-1_5_2\binaries\Gplex" Scanner.gplex
```

E. Auxiliary Classes

C# comes with a large class library, with functionality for almost all requirements. However, there are some classes that are missing, which we need to write ourselves: a class for error handling, container classes, and a graph class.

E.1. Error Handling

To begin with, we need error handling, a way to notify the programmer of errors and warnings. In the C Standard Library there is a macro called assert, and in this application, we imitate that macro by defining a class with the corresponding behavior. It has the two method sets error and warning, that writes a message. The difference between them is that error stops the execution.

Assert.cs

```
using System;

namespace CCompiler {
  public class Assert {
    public static void ErrorXXX(bool test) {
      if (!test) {
        Error(null, null);
      }
    }

    public static void Error(string message) {
      Error(message, null);
    }

    public static void Error(bool test, Message message) {
      if (!test) {
        Error(message, null);
      }
    }

    public static void Error(object value, Message message) {
      Error(false, value, message);
    }

    public static void Error(bool test, object value, Message message) {
      if (!test) {
        Error(message, value.ToString());
      }
    }

    public static void Error(Message message) {
      Error(message, null);
    }

    private static void Error(Message message, string text) {
      Error(Enum.GetName(typeof(Message), message).
            Replace("___", ",").Replace("__", "-").
            Replace("_", " "), text);
    }

    private static void Error(string message, string text) {
      Message("Error", message, text);
      Console.In.ReadLine();
      System.Environment.Exit(-1);
    }


    private static void Message(string type, string message,
                                string text) {
      string funcText;

      if (SymbolTable.CurrentFunction != null) {
        funcText = " in function " + SymbolTable.CurrentFunction.UniqueName;
      }
      else {
        funcText = " in global space";
      }

      string extraText = (text != null) ? (": " + text) : "";

      if ((message != null) &&
          (CCompiler_Main.Scanner.Path != null)) {
        Console.Error.WriteLine(type + " at line " +
                CCompiler_Main.Scanner.Line + funcText +
                " in file " + CCompiler_Main.Scanner.Path.Name +
                ". " + message + extraText + ".");
      }
      else if ((message == null) &&
               (CCompiler_Main.Scanner.Path != null)) {
        Console.Error.WriteLine(type + " at line " +
                CCompiler_Main.Scanner.Line + funcText +
                " in file " + CCompiler_Main.Scanner.Path.Name +
                extraText + ".");
      }
      else if ((message != null) &&
               (CCompiler_Main.Scanner.Path == null)) {
        Console.Error.WriteLine(type + ". " + message +
                                extraText + ".");
      }
      else if ((message == null) &&
               (CCompiler_Main.Scanner.Path == null)) {
        Console.Error.WriteLine(type + extraText + ".");
      }
    }
  }
}
```

Message.cs

```
namespace CCompiler {
  public enum Message {
    Keyword_defined_twice,
    Invalid_specifier_sequence,
    Invalid_specifier_sequence_together_with_type,
    Out_of_registers,
    Unfinished_block_comment,
    Invalid_type_cast,
    Newline_in_string,
    Unfinished_string,
    Parse_error,
    Invalid_type,
    Operator_size,
    Newline_in_character,
    Unfinished_character,
    Invalid_hexadecimal_code,
    Invalid_slash_sequence,
    Double_sharps_at_beginning_of_line,
    Double_sharps_at_end_of_line,
    Two_consecutive_double_sharps,
    Only_extern_or_static_storage_allowed_for_functions,,
    Only_auto_or_register_storage_allowed_for_struct_or_union_member,
    Function_missing,
    Not_a_function,
    Function_main_must_return_void_or_integer,
    Undefined_parameter_in_old__style_function_definition,
    Unmatched_number_of_parameters_in_old__style_function_definition,
    A_function_must_be_static_or_extern,
    Invalid_parameter_list,
    Undefined_label,
    Unreferenced_label,
    Missing_goto_address,
    Tag_already_defined,
    Tag_not_found,
    Duplicate_symbol,
    Unknown_enumeration,
    Not_an_enumeration,
    Different_return_type_in_function_redeclaration,
    Mixing_variadic_function_parameter_in_redeclaration,
    Different_parameter_lists_in_function_redeclaration,
    Functions_cannot_be_initialized,
    Typedef_cannot_be_initialized,
    Extern_cannot_be_initialized,
    Struct_or_union_field_cannot_be_initialized,
    Auto_or_register_storage_in_global_scope,
    Name_already_defined,
    Different_types_in_redeclaration,
    Invalid_tag_redeclaration,
    Case_without_switch,
    Non__constant_case_value,
    Repeated_case_value,
    Default_without_switch,
    Repeted_default,
    Break_without_switch____while____do____or____for,
    Continue_without_while____do____or____for,
    Unnamed_function_definition,
    New_and_old_style_mixed_function_definition,
    Non__integral_enum_value,
    Non__constant_enum_value,
    Bitfields_only_allowed_in_structs_or_unions,
    Only_auto_or_register_storage_allowed_in_struct_or_union,
    Non__integral_bits_expression,
    Bits_value_out_of_range,
    Non__positive_array_size,
    Non__constant_expression,
    Not_constant_or_static_expression,
    An_variadic_function_must_have_at_least_one_parameter,
    A_void_parameter_cannot_be_named,
    An_variadic_function_cannot_have_a_void_parameter,
    Invalid_void_parameter,
    Parameters_must_have_auto_or_register_storage,
    If___ifdef____or_ifndef_directive_without_matching_endif,
    Array_of_incomplete_type_not_allowed,
    Array_of_function_not_allowed,
    Function_cannot_return_array,
    Function_cannot_return_function,
    Duplicate_name_in_parameter_list,
    Syntax_error,
    Invalid_octal_sequence,
    Invalid_char_sequence,
    Non__integral_expression,
    Not_assignable,
    Invalid_type_in_bitwise_expression,
    Invalid_type_in_shift_expression,
    Pointer_to_void,
    Invalid_type_in_expression,
    Invalid_types_in_addition_expression,
    Invalid_types_in_subtraction_expression,
    Non__arithmetic_expression,
    Register_storage_not_allowed_in_sizof_expression,
    Sizeof_applied_to_function_not_allowed,
    Sizeof_applied_to_bitfield_not_allowed,
    Not_addressable,
    Invalid_address_of_register_storage,
    Invalid_dereference_of_non__pointer,
    Not_a_pointer_in_arrow_expression,
    Not_a_pointer_to_a_struct_or_union_in_arrow_expression,
    Unknown_member_in_arrow_expression,
    Invalid_type_in_increment_expression,
    Not_a_struct_or_union_in_dot_expression,
    Member_access_of_uncomplete_struct_or_union,
    Unknown_member_in_dot_expression,
    Invalid_type_in_index_expression,
    Too_few_actual_parameters_in_function_call,
    Too_many_parameters_in_function_call,
    Unknown_name,
    Unknown_name____assuming_function_returning_int,
    Too_many_initializers,
    Duplicate_global_name,
    Missing_external_function,
    Reached_the_end_of_a_non__void_function,
    Floating_stack_overflow,
    Unbalanced_if_and_endif_directive_structure,
    Invalid_line_number,
    Invalid_preprocessor_directive,
    Repeted_include_statement,
    Defined_twice,
    Non__void_return_from_void_function,
    Void_returned_from_non__void_function,
    Invalid_define_directive,
    Invalid_macro_definition,
    Repeated_macro_parameter,
    Mixing_signed_and_unsigned_types,
    Invalid_macro_redefinition,
    Invalid_undef_directive,
    Macro_not_defined,
    Preprocessor_parser,
    Elif_directive_without_preceeding_if____ifdef____or_ifndef_directive,
    Elif_directive_following_else_directive,
    Else_directive_without_preceeding_if____ifdef____or_ifndef_directive,
    Else_directive_after_else_directive,
    Endif_directive_without_preceeding_if____ifdef____or_ifndef_directive,
    Invalid_end_of_macro_call,
    Empty_macro_parameter,
    Empty_macro_parameter_list,
    Invalid_number_of_parameters_in_macro_call,
    Unknown_character,
    Function_missing_in_linking,
    Object_missing_in_linking,
    Non__static_initializer,
    Unnamed_parameter,
    Unknown_register,
    Extern_enumeration_item_cannot_be_initialized,
    Only_array_struct_or_union_can_be_initialized_by_a_list,
    Unmatched_register_size,
    Value_overflow,
    Invalid_expression,
    String_does_not_fit_in_array,
    Only_auto_or_register_storage_allowed_in_parameter_declaration,
    Only_auto_or_register_storage_allowed_for_struct_or_union_scope,
    Only_extern____static____or_typedef_storage_allowed_in_global_scope
  };
}
```

E.2. Container Classes

C# has a large class library holding many container classes. However, there are no classes for pairs or ifElseChains or.

E.2.1. Ordered Pair

In the preprocessor (and in several other classes in the following chapters) we also need the auxiliary class **Pair**.

Pair.cs

```
namespace CCompiler {
  public class Pair<FirstType,SecondType> {
    private FirstType m_first;
    private SecondType m_second;

    public Pair(FirstType first, SecondType second) {
      m_first = first;
      m_second = second;
    }

    public FirstType First {
      get { return m_first; }
      set { m_first = value; }
    }

    public SecondType Second {
      get { return m_second; }
      set { m_second = value; }
    }

    public override int GetHashCode() {
      return m_first.GetHashCode() + m_second.GetHashCode();
    }

    public override bool Equals(object obj) {
      if (obj is Pair<FirstType,SecondType>) {
        Pair<FirstType, SecondType> pair = (Pair<FirstType, SecondType>) obj;
        return m_first.Equals(pair.m_first) && m_second.Equals(m_second);
      }

      return false;
    }
  }
}
```

E.3. Graph

Graph implements a general unordered graph. It holds of set of vertices and a set of edges.

Graph.cs

```
using System.Collections.Generic;

namespace CCompiler {
  public class Graph<VertexType> {
    private ISet<VertexType> m_vertexSet;
    private ISet<Pair<VertexType,VertexType>> m_edgeSet;

    public Graph() {
      m_vertexSet = new HashSet<VertexType>();
      m_edgeSet = new HashSet<Pair<VertexType,VertexType>>();
    }

    public Graph(ISet<VertexType> vertexSet) {
      m_vertexSet = vertexSet;
      m_edgeSet = new HashSet<Pair<VertexType,VertexType>>();
    }

    public Graph(ISet<VertexType> vertexSet,
                 ISet<Pair<VertexType,VertexType>> edgeSet) {
      m_vertexSet = vertexSet;
      m_edgeSet = edgeSet;
    }

    public ISet<VertexType> VertexSet {
      get { return m_vertexSet; }
    }
```

```
public ISet<Pair<VertexType,VertexType>> EdgeSet {
  get { return m_edgeSet; }
}
```

The **neighbourSet** method goes through all edges and add each found neighbor to the vertex.

```
public ISet<VertexType> GetNeighbourSet(VertexType vertex) {
  ISet<VertexType> neighbourSet = new HashSet<VertexType>();

  foreach (Pair<VertexType,VertexType> edge in m_edgeSet) {
    if (edge.First.Equals(vertex)) {
      neighbourSet.Add(edge.Second);
    }

    if (edge.Second.Equals(vertex)) {
      neighbourSet.Add(edge.First);
    }
  }

  return neighbourSet;
}
```

E.3.2. Addition and Removal of Vertices and Edges

```
public void AddVertex(VertexType vertex) {
  m_vertexSet.Add(vertex);
}

public void EraseVertex(VertexType vertex) {
  ISet<Pair<VertexType,VertexType>> edgeSetCopy =
    new HashSet<Pair<VertexType,VertexType>>(m_edgeSet);

  foreach (Pair<VertexType,VertexType> edge in edgeSetCopy) {
    if ((vertex.Equals(edge.First)) || (vertex.Equals(edge.Second))) {
      m_edgeSet.Remove(edge);
    }
  }

  m_vertexSet.Remove(vertex);
}

public void AddEdge(VertexType vertex1, VertexType vertex2) {
  Pair<VertexType,VertexType> edge =
    new Pair<VertexType,VertexType>(vertex1, vertex2);
  m_edgeSet.Add(edge);
}

public void EraseEdge(VertexType vertex1, VertexType vertex2) {
  Pair<VertexType,VertexType> edge =
    new Pair<VertexType,VertexType>(vertex1, vertex2);
  m_edgeSet.Remove(edge);
}
```

E.3.3. Graph Partition

The method **partitionate** divides the graph into free subgraphs; that is, subgraphs which vertices have no neighbors in any of the other free subgraphs. First we go through the vertices and perform a deep search to find all vertices reachable from the vertex. Then we generate a subgraph for each such vertex set.

```
    public ISet<Graph<VertexType>> Split() {
      ISet<ISet<VertexType>> subgraphSet = new HashSet<ISet<VertexType>>();

      foreach (VertexType vertex in m_vertexSet) {
        ISet<VertexType> vertexSet = new HashSet<VertexType>();
        DeepSearch(vertex, vertexSet);
        subgraphSet.Add(vertexSet);
      }

      ISet<Graph<VertexType>> graphSet = new HashSet<Graph<VertexType>>();
      foreach (ISet<VertexType> vertexSet in subgraphSet) {
        Graph<VertexType> subGraph = InducedSubGraph(vertexSet);
        graphSet.Add(subGraph);
      }

      return graphSet;
    }
```

The **DeepFirstSearch** method search recursively through the graph in order to find all vertices reachable from the given vertex. To avoid cyclic search, the search is terminated if the vertex is already a member of the result set.

```
    private void DeepSearch(VertexType vertex, ISet<VertexType> resultSet) {
      if (!resultSet.Contains(vertex)) {
        resultSet.Add(vertex);
        ISet<VertexType> neighbourSet = GetNeighbourSet(vertex);

        foreach (VertexType neighbour in neighbourSet) {
          DeepSearch(neighbour, resultSet);
        }
      }
    }
```

The **generateSubGraph** method goes through the edge set and add all edges which both end vertices are members of the vertex set.

```
    private Graph<VertexType> InducedSubGraph(ISet<VertexType> vertexSet) {
      ISet<Pair<VertexType,VertexType>> resultEdgeSet =
        new HashSet<Pair<VertexType,VertexType>>();

      foreach (Pair<VertexType,VertexType> edge in m_edgeSet) {
        if (vertexSet.Contains(edge.First) &&
            vertexSet.Contains(edge.Second)) {
          resultEdgeSet.Add(edge);
        }
      }

      return (new Graph<VertexType>(vertexSet, resultEdgeSet));
    }
  }
}
```

F. The ASCII Table

0	*nul \0*	32	*blank*	64	@ É	96	` é
1	*soh*	33	!	65	A	97	a
2	*stx*	34	"	66	B	98	b
3	*etx*	35	#	67	C	99	c
4	*eot*	36	$	68	D	100	d
5	*enq*	37	%	69	E	101	e
6	*ack*	38	&	70	F	102	f
7	*bel \a*	39	'	71	G	103	g
8	*bs \b*	40	(	72	H	104	h
9	*ht \t*	41	)	73	I	105	i
10	*lf \n*	42	*	74	J	106	j
11	*vt \vt*	43	+	75	K	107	k
12	*ff \f*	44	,	76	L	108	l
13	*cr \r*	45	-	77	M	109	m
14	*soh*	46	.	78	N	110	n
15	*si*	47	/	79	O	111	o
16	*dle*	48	0	80	P	112	p
17	*dc1*	49	1	81	Q	113	q
18	*dc2*	50	2	82	R	114	r
19	*dc3*	51	3	83	S	115	s
20	*dc4*	52	4	84	T	116	t
21	*nak*	53	5	85	U	117	u
22	*syn*	54	6	86	V	118	v
23	*etb*	55	7	87	W	119	w
24	*can*	56	8	88	X	120	x
25	*em*	57	9	89	Y	121	y
26	*sub*	58	:	90	Z	122	z
27	*esc*	59	;	91	[Ä	123	{ ä
28	*fs*	60	<	92	\ Ö	124	| ö
29	*gs*	61	=	93	] Å	125	} å
30	*rs*	62	>	94	^ Ü	126	~ ü
31	*us*	63	?	95	_	127	*delete*