

**ASSUMPTION UNIVERSITY**

**Faculty of Science and Technology**

Department of Computer Science

**SC 4299**

**SENIOR PROJECT**

**FINAL REPORT**

**Low Level Virtual Machine C# Compiler**

**Project Advisor: Dr. Kwankamol Nongpong**

**Project Committee: Dr. Songsak Channarukul**

**A. Se Won Kim**

**by**

Prabir Shrestha4915302

Myo Min Zin 4845411

Napaporn Wuthongcharernkun 4846824

**Semester 2/2009**

**ASSUMPTION UNIVERSITY**

**Faculty of Science and Technology**

Department of Computer Science

**Senior Project Approval**

**Title:** Low Level Virtual Machine C# Compiler

**Members:** Prabir Shrestha 4910552

Myo Min Zin 4845411

Napaporn Wuthongcharernkun 4846824

**Academic Year:** 2009

**Semester:** 2

The Department of Computer Science, Faculty of Science and Technology, Assumption University has approved of this final report, submitted in partial fulfillment for the three-credit course, SC 4299, under the requirements for the degree of Bachelor of Science in Computer Science.

Approving Committee:

|  |  |
| --- | --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Dr. Kwankamol Nongpong  (Advisor) |  |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Dr. Songsak Channarukul  (Committee Member) | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  A. Se Won Kim  (Committee Member) |

**Acknowledgement**

We would like to express our sincere appreciation and gratitude to the following individuals and groups for whom without their support, this project would not be possible:   
Our committee members and project advisor, Dr. Kwankamol Nongpong for her guidance and suggestions throughout the duration of the work on this project.   
JetBrains for providing the open source license for Resharper which was a great help during the development of this project.

**ABSTRACT**

In this report we present the framework and methodology for building a C# compiler that is independent from the .NET framework. Using the target code LLVM IR (Low Level Virtual Machine Intermediate Representation), a low-level language with high-level type information which supports further native code compilation.

The inner working of the compiler in each phase of the compilation process, from lexical scanning to code generation is elaborated on in a precise manner in this report.   
The overall aim in this project is to create a portable tool that serves as an alternative from the .NET framework for C# developers, at the same time enabling the usage of C# across multiple platforms.

Table of Contents

[Chapter 1 Introduction 1](#_Toc257211302)

[1.1 Rationale and Motivation 2](#_Toc257211303)

[1.2 Problem Statement 3](#_Toc257211304)

[1.3 Goals and Objectives 5](#_Toc257211305)

[1.4 Scope and Limitations 6](#_Toc257211306)

[Chapter 2 Literature Review 9](#_Toc257211307)

[2.1 Source Language Background 9](#_Toc257211308)

[2.2 LLVM Description 9](#_Toc257211309)

[2.3 Contributions to C# 10](#_Toc257211310)

[Chapter 3 Framework 12](#_Toc257211311)

[3.1 Scanner 13](#_Toc257211312)

[3.2 Parser 14](#_Toc257211313)

[3.3 Semantic Analyzer 14](#_Toc257211314)

[3.4 Code Generator 14](#_Toc257211315)

[Chapter 4 Methodology 15](#_Toc257211316)

[4.1 Scanner 15](#_Toc257211317)

[4.2 Parser 16](#_Toc257211318)

[4.2.1 Multi-symbol Look ahead 16](#_Toc257211319)

[4.2.2 Resolver Symbols 17](#_Toc257211320)

[4.3 Semantic Analysis 20](#_Toc257211321)

[4.3.1 Class Table Construction 20](#_Toc257211322)

[4.3.2 Class Hierarchy Construction 23](#_Toc257211323)

[4.3.3 Method Table Construction 25](#_Toc257211324)

[4.3.4 Object Layout 29](#_Toc257211325)

[4.3.5 Semantic Checking 29](#_Toc257211326)

[4.3.6 Semantic Errors 39](#_Toc257211327)

[4.4 Code Generator 39](#_Toc257211328)

[4.4.1 Integer and String Table Generation 41](#_Toc257211329)

[4.4.2 Floating Point Representation 42](#_Toc257211330)

[4.4.3 Object Layout Code Generation 42](#_Toc257211331)

[4.4.4 Basic Language Constructs Code Generation 43](#_Toc257211332)

[4.4.5 Method Code Generation 45](#_Toc257211333)

[4.4.6 Constructor Code Generation 47](#_Toc257211334)

[Chapter 5 Garbage Collector Internals 48](#_Toc257211335)

[5.1 Storage Structure 49](#_Toc257211336)

[5.2 Mark and Sweep Alogrithm 51](#_Toc257211337)

[5.2.1 Root Enumeration 51](#_Toc257211338)

[5.2.2 Mark 52](#_Toc257211339)

[5.2.3 Sweep 52](#_Toc257211340)

[Chapter 6 Conclusion 54](#_Toc257211341)

[References 55](#_Toc257211343)

[Appendix 56](#_Toc257211344)

[A.1 Project Dependency Graph 56](#_Toc257211345)

**List of Tables**

[Table 1‑1 Keywords 7](#_Toc257211353)

[Table 1‑2 Operators 7](#_Toc257211354)

[Table 4‑1 Method Table Data 25](#_Toc257211355)

[Table 4‑2 Method Table Data with Keys 27](#_Toc257211356)

[Table 4‑3 Member Modifiers 28](#_Toc257211357)

[Table 4‑4 Comparison Operators 29](#_Toc257211358)

**List of Figures**

[Figure 1.1 Compiler Phases 2](#_Toc257211360)

[Figure 3.1 Coco/R Visual Studio Plugin 12](#_Toc257211361)

[Figure 3.2 Overall Compiler Phases 13](#_Toc257211362)

[Figure 4.1 Binary Expressions AST Classes 18](#_Toc257211363)

[Figure 4.3 AST Nodes 20](#_Toc257211364)

[Figure 4.4 Class Inheritance 22](#_Toc257211365)

[Figure 4.5 Boolean Matirx 22](#_Toc257211366)

[Figure 4.6 The final Boolean Matrix 23](#_Toc257211367)

[Figure 4.7 Object Browser 24](#_Toc257211368)

[Figure 4.8 Symbol Table Structure I 31](#_Toc257211369)

[Figure 4.9 Symbol Table Structure II 32](#_Toc257211370)

[Figure 4.10 Symbol Table Structure III 33](#_Toc257211371)

[Figure 4.11 Symbol Table Strucutre IV 34](#_Toc257211372)

[Figure 4.12 ILLVMCodeGenerator 41](#_Toc257211373)

[Figure 4.13 String Layout 43](#_Toc257211374)

[Figure 5.1 Object Graph 49](#_Toc257211375)

[Figure 5.2 Garbage Collectable Object with Headers 50](#_Toc257211376)

# Chapter 1

**Introduction**

Modern programming languages today give us a means of expressivity for applications in a variety of ways, through varying means.

The developer’s choice of language in constructing an application, first and foremost, could almost instantly convey to us information about the purpose of the system design.

There are a myriad of classifications of styles of programming languages, from logical, imperative, and functional to object-oriented styles of programming. The wide mainstream use and popularity of object-oriented programming languages we believe is due to its ability to effectively and easily model the real world objects and their functionalities that we see around us in a way that machines can understand.  
Modern high-level languages such as the source language we have focused on, C#, more often than not contains a combination of all the above listed programming paradigms. In the newer versions that have been released, an increased ease of use in functionalities have been deployed in several areas such as generics, Language Integrated Queries (Linq), and anonymous functions to name a few.  
However the focus of our project is primarily on the basic object-oriented elements of the language which will capture the core-constructs of the syntax and semantics of our source language.

Diversity in alternative usage is another factor of importance when there are large communities of users for a particular language. To further this reason an alternative method of deploying and compiling a C# application is primarily our objective in this project. Large existing compiler frameworks are widely in use for the C# language such as Microsoft's .NET and Mono. These systems are sometimes however bulky due to the sets of features it provides even for those which developers would not be using. Therefore the practicality and usefulness of our project is seen as a small portable tool for developers of C# applications.

The core objective of this project is to create a compiler for the C# language that generates a portable intermediate representation of low level code, which can then be used across a wide variety of architectures and operating systems with minimal or no code modification to the original source. In order to accomplish the task, Low Level Virtual Machine Intermediate Representation (LLVM IR) has been chosen as the target code output generated by the compiler due to its nature of independence.

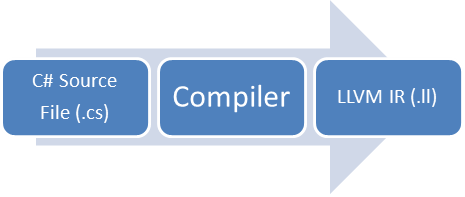


Figure 1.1 Compiler Phases

## Rationale and Motivation

From different contributions and evolutions to the compiler technologies and programming paradigms we had motivations to pursue in the creation of a new C# compiler.

Distributing the binaries created by the C# compilers requires us to install the bulky .NET Framework. Even a traditional “helloworld” program would require all the features of .NET Framework to be installed. To solve this problem we have taken the approach of C and C++ which link the appropriate libraries required to the program successfully.

D Language has also been one of the major inspirations, providing the programmers with features of modern languages such as automatic memory management by garbage collection, interfaces and yet producing high performance codes to enable system programming[1] such as system drivers and even operating systems.

Writing of operating system has been evolving throughout the past decades from assembly codes to high level languages such as C and C++. There have been many other projects such as SharpOS[2], Comos and even Microsoft’s research operating system – Singularity[3], which have taken a different approach by writing the kernel, device drivers and application in managed code. The compilers of these operating systems have been the motivation to create a C# compiler that produces native codes.

“Write Once, Run Anywhere” (WORA) slogan from Sun Microsystems has made us think to generate a portable code which could be used over a wide variety of operating system and computer architectures.

## Problem Statement

The way we write programs have been evolving ever since the beginning of the stored program concept and continue to evolve even at the present due to the advances in hardware and software technologies. From the introduction of Java and now the .NET framework, the concept of virtual stack machine and Just in Time Compilation (JIT) has been coming to popularity. One of the notable compilers which use this concept is C#. It has been allowing the programmers to write compiled machine-independent codes which could virtually be executed in any architecture.

Even though Java byte-code and Common Language Infrastructure (CLI) consists of highly machine independent code, it has not been a candidate for system programming due to performance issues such as lack of speed as compared to other languages such as C and C++ and due to the JIT. LLVM has a similar concept of JIT by converting the code to a compiled LLVM bit code which could then be executed in other architecture and operating system. In order to gain better performance for a particular architecture or operating system, it could further be compiled to a native code. As of writing, LLVM’s retargettable code generator currently supports most of the popular architectures such as x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, PIC16 MIPS, MSP430, SystemZ and XCore [4].

While languages such as C and C++ provide better execution speed than compared to C# and Java, programmers do have to face with unsafe codes such as manual memory management which could lead to memory leak or dangling pointers. This memory problem is usually solved by the use of garbage collection as seen in C# and Java. It also introduces the concepts of delegates by avoiding the use of unsafe function pointers.

As developers have been writing their codes, a set of common principles on the way they write code have been evolving. Uses of accessors and mutators have been a common way of accessing variable in the object oriented world rather than the use of public variables. Many of these features have been addressed by C# language. Because of features such as the memory management and the adherence to the principles of writing a program, C# was chosen as an input language for our compiler.

Migration to different platforms causes the programmers to write architecture specific code to each of those platforms. Languages such as C and C++ do not have a straight forward way to know the length of integer – 32 bit or 64 bit. But C# provides an easier way to access it by using the inbuilt Int32 object.

## Goals and Objectives

The objective of our project is to create a compiler for the C# language in which the target language is in a form of low level independent language similar to assembly code called Low Level Virtual Machine Intermediate Representation (LLVM IR).

The focus of our project is primarily on each phase of the compilation process, from scanning the source language until target code generation. These phases include Lexical Analysis, Syntax Analysis, Semantic Analysis and Intermediate Code Generator. Other phases such as assembling and linking are handled by LLVM tools.

The finalization and expected outcome of the project is a compiler that is set to be functional for the C# language specifications according to the designated scope of the language that we determined.

The basic requirements for the compiler include the following:

The compiler will properly recognize the lexical structures of the C# language.

Check the syntax taking into account the correct grammar according to the language specifications as well as the semantics of the program otherwise generating errors accordingly.

The objective of our project is to create a compiler for the C# language in which the target language is in a form of low level independent language similar to assembly code called Low Level Virtual Machine Intermediate Representation. (LLVM IR).

The focus of our project is primarily on each phase of the compilation process, from scanning the source language until target code generation. These phases include Lexical Analysis, Syntax Analysis, Semantic Analysis and Intermediate Code Generator. Other phases such as assembling and linking are handled by LLVM tools.

The finalization and expected outcome of the project is a compiler that is set to be functional for the C# language specifications according to the designated scope of the language that we determined.

The basic requirements for the compiler include the following:

* The compiler will properly recognize the lexical structures of the C# language.
* Check the syntax taking into account the correct grammar according to the language specifications as well as the semantics of the program otherwise generating errors accordingly.

## Scope and Limitations

The scope from the language specifications has been determined for our project according to the following listed keywords and operators, which is a subset of C# version 1.0. We have chosen version 1 rather than the newer versions of C# because we will not be supporting most of those new additional features such as Generics, Language Integrated Query (Linq).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| base  bool  break  char  class  const  continue  do  else | enum  explicit  extern  false  float  for  get  if  implicit | int  namespace  new  null  operator  object  public  override | private  protected  return  sealed  set  static  string  struct | this  true  typeof  using  virtual  void  while  value |

Table 1‑1 Keywords

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primary** x.y  f(x)  a[x]  x++  x--  new | **Unary** + -  !  ++x  --x (T)x | **Relational and type testing**  <  >  <=  >=  ==  != | **Assignment**  =  +=  -=  \*=  /= | **Multiplicative & Conditional** \*  /  &&  || |

Table 1‑2 Operators

Based on the ECMA-334 C# Language Specification [7], the value of char in C# is a Unicode Character. Microsoft’s implementation of .NET framework implements it as 16-bit characters that can be used to represent most of the known written languages in the world. For our C# compiler the implementation is not the same as the original version but rather, char takes on the size of 8-bit which is the same as the standard C and C++. This holds the same for string type.

C# Language Specification based on the ECMA-344 allows a distinct type for enumeration type such as byte, sbyte, short, ushort and int. The compiler will only be supporting 32-bit integer (int) as the enumeration type.

Microsoft .NET has provided base class libraries, which are the classes, structures, enumerations and delegates, for C# programmers to deal I/O, accessing Database. In our complier, a subset of these libraries is provided.

System.Char System.Int32 System.Single System.Object System.Void

System.Console System.Environment System.Boolean System.String

Our own libraries are provided for the end user to assemble and link with the output LLVM IR.

# Chapter 2

**Literature Review**

## 2.1 Source Language Background

C# is a high-level object-oriented programming language that is part of the .NET language family developed by Microsoft. Although the language is considered to be primarily object-oriented a closer look reveals that it is in fact a multi-paradigm language with aspects of functional and imperative programming styles included in it as well.

It is currently designed to function within the Common Language Infrastructure (CLI) which provides a CTS (Common Type System) and CLS (Common Language Specification) so that when it is compiled it generates the CIL (Common Intermediate Language).

## 2.2 LLVM Description

Low Level Virtual Machine (LLVM) is a compiler infrastructure that consists of two primary components, an optimizer and a code generator. It is designed so that optimizations of programs can occur at different phases of the program life such as compile-time, link-time and run-time [5].

LLVM IR (Intermediate Representation) is a low-level language similar to assembly language containing RISC like instruction set that effectively captures the operations of the processor whilst avoiding machine-specific constraints such as pipelines, physical registers and other low-level calling conventions. By increasing the layer of abstraction apart from the hardware specifics in the code, the LLVM IR is in a sense, platform independent and can be used on a variety of machines with different hardware specifications.

The common code representation used throughout all phases of the LLVM compilation strategy is a Single Static Assignment (SSA) based representation which provides type safety, low-level operations and is flexible and capable of representing high-level languages in a clear and efficient manner.

A key important factor contributing to the productivity of the LLVM system is its virtual instruction set. The LLVM code is a low level representation while being able to contain high-level information due to its designed structure.

## 2.3 Contributions to C#

Other C# compiler projects that are available apart from Microsoft's .NET framework are discussed briefly here to give an overview of the relevant developments that have surfaced in this particular field, these include Mono, Cosmos (IL2CPU) [6], Bartok and Ensemble.

Mono is an open source implementation of the .NET framework, it contains a Mono C# compiler that is written in C# and can be run on several different operating systems such as Linux, UNIX, Mac OS X and Solaris. The concept of how it works is first the C# code gets compiled into MSIL then the Mono JIT translates the MSIL into native code at run time which is similar to as the original implementation of the .NET framework by Microsoft.

Cosmos(C# Open Source Managed Operating System) is an OS that is written entirely in C#, the OS makes use of IL2CPU which is an AOT (ahead-of-time) compiler that translates the CIL into machine code by outputting raw assembly files which then get processed through NASM (Netwide Assembler).

Bartok was originally made for the use of the OS Singularity developed by Microsoft Research. It works by translating CIL into native code by using three intermediate representations, HIR (High-level IR), MIR (Medium-level IR) and LIR (Low-level IR). At each of these representations starting from high-level it works its way down to low-level IR and gradually changes the code representation at each phase until it reaches the lowest level which is basically assembly, and then a standard linker puts the objects together to create the native x86 executable.

# Chapter 3

**Framework**

The compiler has been implemented in C# language using Microsoft Visual Studio and .NET Framework. The core command line compiler (lsc.exe) requires minimal .NET version 2 while the compiler visualize (lsc-visual.exe) requires .NET 3.5 Service Pack 1.

Implementation of scanner and parser was done using the automatic scanner and parser generator called Coco/R (compiler compiler generating recursive descent parser) which is also written in C#. To ease our development we have developed a Visual Studio plugin for Coco/R. The Coco/R plugin is responsible for auto generating the scanner and parser files when the attributed grammar file is saved.

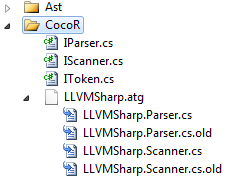


Figure 3.1 Coco/R Visual Studio Plugin

In order to prevent accidental mistakes, the parser and scanner files are also backed up by replacing the extensions with .cs.old.

The overall compilation process can be summarized from the following diagram.

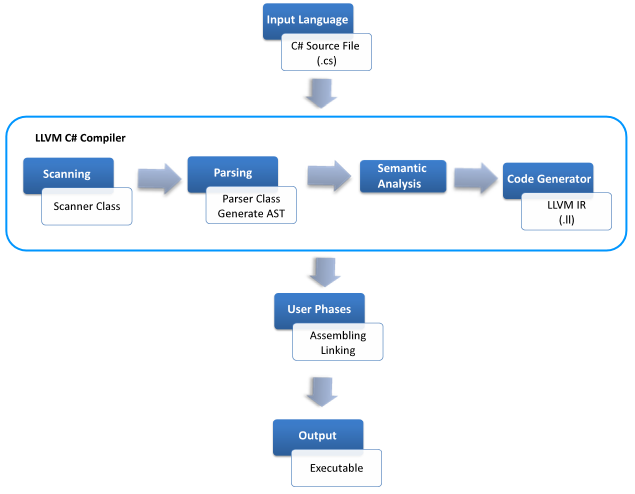


Figure 3.2 Overall Compiler Phases

## 3.1 Scanner

Basically Coco/R takes the attributed grammar of source language and generates a scanner and recursive descent parser for this particular language. The scanner generated by Coco/R is responsible for reading the input stream and returning the stream of tokens to the parser.

In a traditional overview of the compilation, scanning and parsing process are seen as two distinct separate processes occurring one after the other. However, using Coco/R tool, the scanner and parser generation occurs at the same time where the codes for scanner and parser are written in the same attributed grammar file usually ending with .atg as extension.

## 3.2 Parser

The parser handles the syntax analysis for the source language. During the syntax analysis phase the focus of concern is checking for the source input program’s adherence to the grammatical rules of the source language. Two major techniques can be applied for parsing – table driven or recursive descent. The Coco/R tool deploys the recursive descent parsing technique.

Recursive descent parsing is well-known as top-down parsing technique that is simple, convenient and accomplishes the task efficiently for the next sequenced phase, i.e., semantic analysis to begin.

## 3.3 Semantic Analyzer

After the creation of AST, semantic analysis involves the following steps that needed to be performed one after another.

* Class Table Construction
* Class Hierarchy Construction
* Method Table Construction
* Object Layout Construction
* Scope Checking and Symbol Table
* Type Checking

- Semantic Errors

## 3.4 Code Generator

Once the source code has successfully undergone semantic analysis and is proven to be semantically correct according to the language specifications of C#, LLVM IR code is generated.

# Chapter 4

**Methodology**

## 4.1 Scanner

The scanner is responsible for performing lexical analysis on the source language by taking the syntax input of the program, tokenizing it and checking for lexical errors. Tokenization refers to the process of categorizing the syntax of the program into its basic building blocks which are tokens. Tokens usually comprise of identifiers, keywords, number and symbols which are the fundamental blocks of a program.

Following scanner code fragment in Coco/R attributed grammar file defines the characters that can be used in the source language input.

letter = ‘A’.. ‘Z’+ ‘a’.. ‘z’ + ‘\_’ .

digit = ‘0’ .. ‘9’ .

.. represents the range. These are defined under the CHARACTERS section in the attributed grammar file. We could then use these to define tokens as follows under the TOKENS section.

ident = [‘@’] letter { letter | digit } .

integerConstant = digit { digit } .

Words enclosed in the square brackets [ ] represents optional tokens or characters, while { } represents one or more occurrences. | represents OR.

## 4.2 Parser

The Coco/R tools implements the top-down parsing technique. This parsing technique as the name suggests, starts constructing the parse tree from the top of the tree, the root and works its way downwards, making predictions for each next token input as to which production rule may be used, and adding them on to the parse tree. The controls flow of recursive descent parsing is strictly linear, no jumps, loops or conditional statements are used. However recursive subroutines are in effect as that is a primary characteristic of recursive descent parsing.

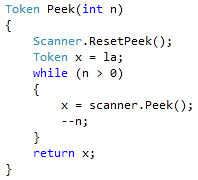
However in general, for this parsing technique a basic requirement of the grammar is that it should be in LL(1) form. LL(1) is an abbreviation for left to right with left canonical derivations using only one look-ahead symbol. The grammar of the source language which we have written however is not in LL(1) form, this then presents another factor into the equation. There are a number of solutions that Coco/R uses for grammars that are not in LL(1) form. They are typically termed as ‘Conflict Resolvers’ and include the following.

1. Multi-symbol Look ahead
2. Symbol Resolvers

### 4.2.1 Multi-symbol Look ahead

In this technique the parser generated by the Coco/R uses two global variables that store the last recognized terminal and the current look ahead symbol. When the need arises to look ahead more than one symbol, the generated scanner does this by using the methods ResetPeek() and Peek(). The ResetPeek method initializes the peeking to begin from the symbol after the current look ahead symbol. The Peek method returns the next symbol as a Token but does not remove it from the input stream, so these symbols will be sent again by the scanner when the parsing resumes.

There are cases when one look ahead is not enough. To solve this problem we have created a custom function called Peek(int) utilizing ResetPeek() and Peek() function of Coco/R which returns the n-th token after the current look ahead token.



### 4.2.2 Resolver Symbols

These are artificial tokens that are added into a separate section in the grammar to help direct the parser in the correct way. They are inserted on-the-fly during parse time as seen necessary by the resolution routine that is used by Coco/R. These resolution routines are automatically put into the generated parser by Coco/R.

During the parsing phase, Abstract Syntax Tree (AST) is generated. All the AST nodes inherit from a comman class called AstNode. Some AstNodes implement IAstExpression indicating it is an expression.

For simplification some of the AST nodes have been aggregated to from a clearer root AST node such as AstBinaryExpression which contains LeftOperand and RightOperand. These LeftOperand and RightOperand are IAstExpression. Other binary expressions such as addition, subtraction, multiplication and division derive from AstBinaryExpression.

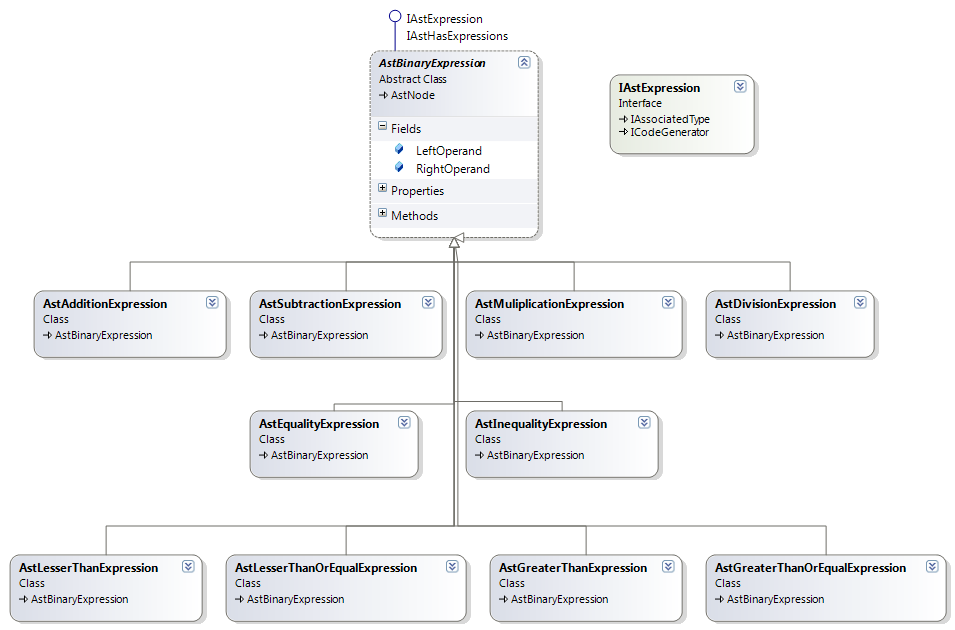
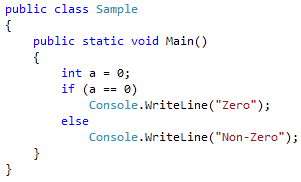


Figure 4.1 Binary Expressions AST Classes

According to the scope of this project the following constructs are syntactically recognized by the parser. Below is a comprehensive list of the supported syntax.

1. Declaratives
2. Namespace
3. Primitive & Constructed Types
4. Expressions
   1. Assignment
   2. Function calls
   3. Type Casting
5. Statements
   1. Initialization of variables/members
   2. Constructors
   3. Post/Pre Increment
   4. Post/Pre Decrement
   5. Indexers
6. Operators
   1. Basic Operators
   2. Operator Overloading
7. Encapsulation
   1. Member Modifiers
   2. Accessors / Mutators

AST nodes could then be visualized using the compiler visualizer (lsc.visual.exe). Below is a sample code fragment including the AstNode visualized by the compiler.



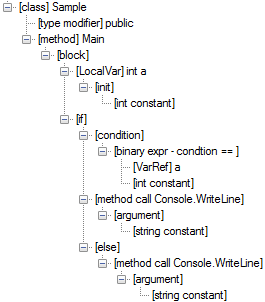


Figure 4.3 AST Nodes

## 4.3 Semantic Analysis

### 4.3.1 Class Table Construction

This is the first phase of Semantic Analysis out of the six mentioned in the framework. This is where the all the classes in the program are listed in one global table. The information is later used to construct Class Hierarchy, Class Table, Method Table and Object Layout. This part is also responsible for checking class name duplication and circular inheritance. On failure of this construction the compiler cannot go into next steps.

The Class Table contains the fully qualified class name as Key and Value which is referenced to AST Class Node. This information is collected from the AST Tree by recursively calling each node.

The AST generated by the parser has only the information of the class name as inputted by the user, but not the fully qualified class name. First, the fully qualified class name is constructed and added to the AST node information. Once we have all the classes in the Class Table, the circular reference checking is started.

Circular reference checking was done by arranging the inheritance relationship within a square Boolean matrix.

The dimensions of the matrix are equivalent to the number of classes that are in a class inheritance relationship. For each class one row and the same corresponding column represent one class. The inheritance relationships are represented by setting the boolean value to true in the cell of the boolean matrix where the row class inherits from the column class. The other column classes from which the row class does not inherit, the values are set to false.

For example, given the class hierarchy where Class A inherits from Class D, Class B inherits from Class A, Class C inherits from Class B, and Class D inherits from Class C as depicted in the figure 4.4.

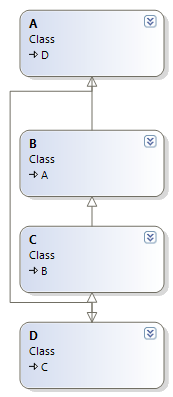


Figure 4.4 Class Inheritance

The Class inheritance hierarchy would be represented in the Boolean Matrix in this manner.



Figure 4.5 Boolean Matirx

After the matrix has been generated, the following formula is applied to the matrix where C represents the matrix and Rank(C) denotes the maximum number of linearly independent rows in the matrix.

The result of applying this formula on the matrix is depicted Figure 4.6.

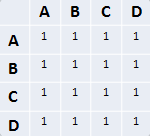


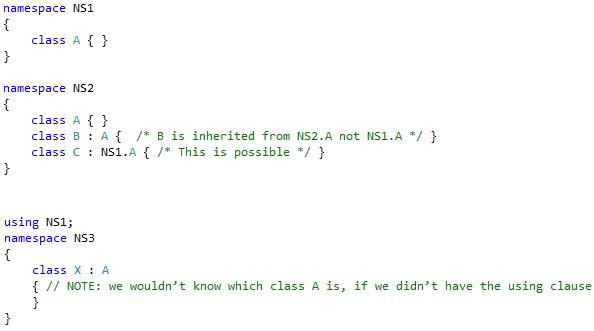
Figure 4.6 The final Boolean Matrix

Classes involved in circular inheritance would be marked in the new matrix as having a Boolean value of true on their corresponding diagonal element. From the results of the Boolean operation the diagonal rows corresponding to classes A, B, C and D have the value of true which shows that these classes are involved in circular inheritance therefore a semantic error has been detected and an error message is added to the error list accordingly.

### 4.3.2 Class Hierarchy Construction

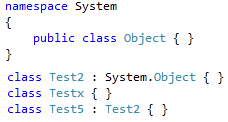
In this stage, the class objects are collected and made a tree hierarchy based on the parent and child information. It constructs a tree that shows the parent and child relationship of Classes.

AST Node contains the name of the parent class but not the fully qualified name so we would not know which class we are referring to if the two classes use the same name under different namespaces. For example, the following C# code is correct and possible.



AST only gave the information of class name at this point and we have to go through all the using to find the parent class fully qualified name.

This is an example of the object hierarchy visualization created from the code fragment provided.



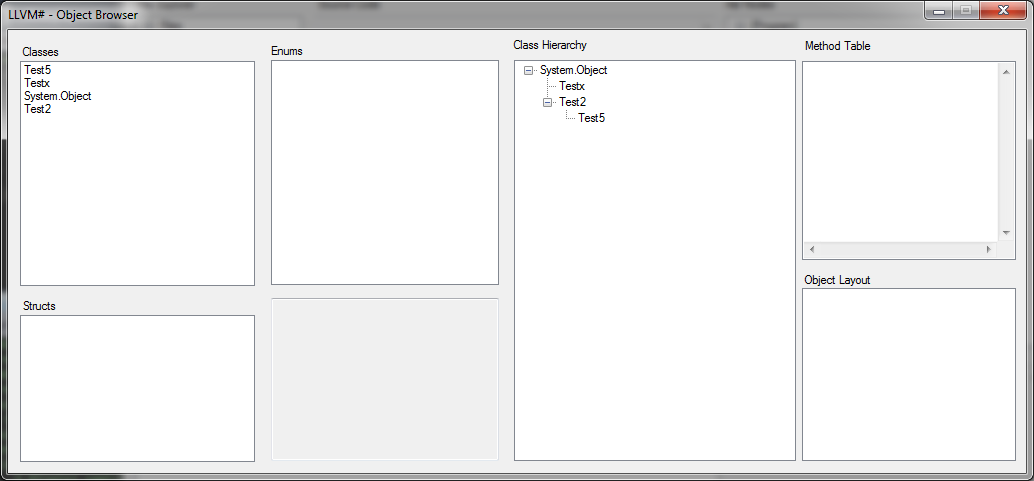


Figure 4.7 Object Browser

### 4.3.3 Method Table Construction

In the phase of compilation where the semantic checks occur, a method table is generated for each class contained in the user’s source code. This method table contains a collection of 5 different types of nodes pertaining to the class which are constructors, methods, operator overloads, type conversion functions and accessors. At the implementation level this method table is stored as a hash table, where the key is given a unique signature in order to avoid duplicity.

|  |
| --- |
| **Method Table Data** |
| Constructors |
| Functions |
| Operator Overloads |
| Accessors |
| Type Converters |

Table 4‑1 Method Table Data

The way that the key is generated for all these 5 members is similar to each other.   
If it is a constructor the key is generated by first checking if the constructor is derived from inheritance via the base class or if it’s a constructor generated in its own class. Hence, if it is from an inherited class, the key which is represented as a string will append “\_cb\_” where ‘c ‘ means constructor and ‘b’ implies it’s from the base class. Otherwise “\_ct\_” is appended to the string, where‘t’ means this current type.   
Next the name of the constructor which is basically the type that the constructor belongs to is added on to the string, followed by the fully qualified type of each parameter. The whole AstConstructor node is then stored as the value that is mapped to the key.

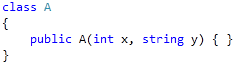
In the case that no constructor has been defined for a given class, an automatic constructor is generated taking no parameters. The remaining members of the constructed type are stored in the method table in a similar manner.

For accessors the same concept is reinforced. When there is a class inheritance relationship, the collection of accessors in the parent class is first iterated through. They are denoted as “\_ab\_” on the key whereas accessors contained in the current class are represented by “\_at\_” where ‘a’ represents that it’s an accessor, ‘b’ from the base class and ‘t’ from the current class. Following this the name of the accessor is then appended to the key. The return type is not mangled into the key as C# does not allow accessors with the same name but differing return types in a given class or struct. Therefore the name of the accessor and the reference of inheritance are sufficient enough to ensure there’s no duplicity of an accessor in the constructed type.

The operator overloaded functions are also added into each class’s method table. The key for the operator overloaded functions in the method table are represented with the string “\_op\_”, then depending on which operator is being overloaded in that function a few letter abbreviations are used to signify that. For example, the binary operators ‘+’ and ‘-‘ are represented as “p” and “m” respectively on the key. Then the fully qualified type of each parameter is appended. Once that is complete, if the method table does not contain an overloaded function with the same signature, the AstOperatorOverload object is stored in the hashtable.

Explicit and implicit type conversions are represented as methods for each type. However, only one overloaded type of conversion is allowed between 2 different types, implicit or explicit. Therefore when adding the type conversion functions into the method table a check is performed to determine if it is an implicit or explicit type conversion then the string “\_imp\_” or “\_exp\_” is appended accordingly. Followed by the return type and the fully qualified type of the single parameter it takes in. The code fragments below gives a concrete example for each generalized case.

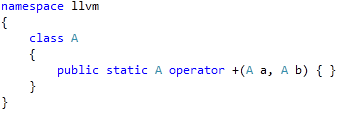
**Constructor**



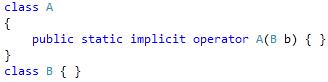
**Accessor**



**Operator Overload**



**Type Converter**



From the above source code fragments the following keys are generated.

|  |  |
| --- | --- |
| **Method Data** | **Key Generated** |
| Constructor | \_ct\_1A\_13\_6System5Int32\_27\_6System6String |
| Function | \_mt\_8Function\_13\_6System5Int32\_25\_6System4Char |
| Accessor | \_at\_MyNum |
| Operator Overload | \_op\_p\_6llvm.A\_6llvm.A |
| Type Conversion | \_op\_imp\_A\_llvm.B |

Table 4‑2 Method Table Data with Keys

When adding members from the base class, the protection modifier is also taken into account for each member to see if access is allowed from the child class. Should the class member in the parent class be marked ‘private’ this denotes that this member can be used only within its current class so it’s not allowed to be used by classes that derive from it. Therefore the constructor, accessor or method will not be added into the child class’s method table for this case. However the ‘protected’ and ‘public’ access modifiers allow derived classes to use these members. So in the case that the member uses those modifiers they are passed down accordingly in an inheritance relationship.

Member modifiers give semantic meaning to each of the methods as it is a determining factor in the capabilities and constraints of that member in particular. According to the proposed scope of this project the following table shows the modifiers that are supported for class members in the compiler.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Member Modifiers | | | | |
| private | protected | public | static | sealed |

Table 4‑3 Member Modifiers

There are two distinct cases for the member modifiers as some of the above modifiers are valid for struct members while others are not. This validation check also takes place to ensure the correctness of the program, as modifiers such as virtual, override, and sealed do not make any sense for structure types as they have no inheritance relationship with other types and are sealed by default.

Operator overloaded methods have a set of rules that must be adhered to, these include that the operator function must be marked as public and static and at least one of the parameters of the overloaded method must be of the enclosing type.

Comparison operators that are in the language are required to be overloaded in pairs this implies that if the operator ‘<’ is overloaded the matching operator ‘>’ must be overloaded as well. Failure to do so results in a semantic error being thrown. This holds true for all comparison operators listed below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Comparison Operators | | | | | | | |
| > | < | >= | <= | == | != | true | false |

Table 4‑4 Comparison Operators

Type Conversion functions must also be marked as public and static and declared as either an implicit or explicit type conversion. The return type or parameter type is required to be of the enclosing type.

### 4.3.4 Object Layout

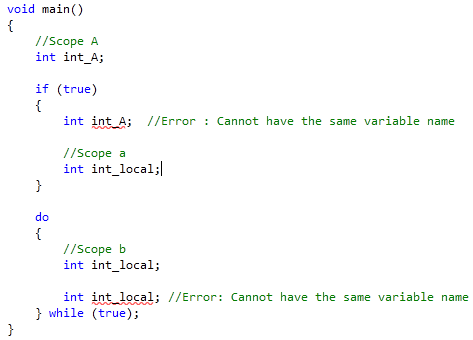
The object layout construction is done at the same time while the Method Table is constructing.

The object layout table contains filed information and a link to Method table for each class. The child class table must have the parent’s information. Like in Method Table Construction, it is done from roots class to child classes.

### 4.3.5 Semantic Checking

#### Scope Checking

One of the most important parts in semantic checking is identifying the program scope along the program and maintaining the variable information within the scope. For instance, two variables cannot have a same name in the same scope as it would lead to ambiguity. But variables are allowed to have the same name as long as they are in different scope. The same scope refers to the area in which a particular block or blocks that are nested share the same variable information. The different scope means blocks that are in the same level and information inside the blocks are independent to each other. For example:



In the code fraction above, Scope ”A” contains main block, “if” condition block, which is scope ‘a’, and “do” loop block, which is a scope ‘b’. The integer int\_A is declared in scope “A” and trough out the scope no other variable can be named int\_A, which mean we cannot have int\_A in both scope ‘a’ and scope ‘b’. But for scope ‘a’ and ‘b’, they are in the different scope and thus they can have integer int\_local inside the scope with the same name. But, they cannot have two variable name int\_local inside their scope.

The semantic checking ensures that no variable with the same name is declared twice and the error message is given if found. The actually implementation is done by using Symbol Table.

#### Symbol Table

Symbol Table is a data structure that stores all the information about names, type and location of a declared variable in current scope. In this project, we use symbol table mainly for scope checking purpose and type checking of a local variable.

Generally, the symbol table has two main functions called “Insert” and “Lookup”. The “Insert” function will receive name and type of a local variable to store in symbol table. “Lookup” function will receive name to look up in the symbol table and will return SymbolObject if found. SymbolObject contains the name and type.

The way we use with symbol table for scope checking is simple. Whenever a variable declaration is found in the program, we insert it into the symbol table with its name and type. The insert function check existing symbol records and tell us if the same variable name exist in the symbol table or not. The insert function use lookup function to perform the checking. The symbol table maintains the current scope. If the two or more variable is declared with the same name in current scope the error event is fired and gives the user the clear error message.

The following figure shows the data structure when the program enters the first scope and encounters the local variable integer int\_A and then adds it to the symbol table.

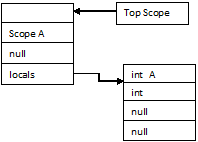


Figure 4.8 Symbol Table Structure I

The figure shows when the program is entered into scope ‘a’. When integer int\_A is inserted the ‘insert’ method will check the existing record and give an error message.

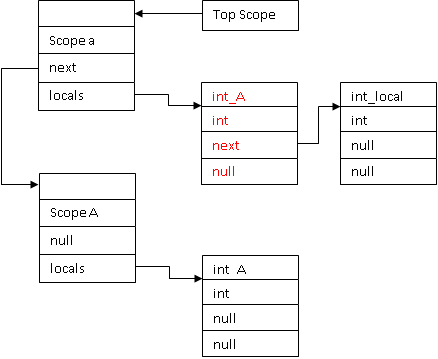


Figure 4.9 Symbol Table Structure II

The following figures show the state when the program enters into the scope ‘b’. Before it enters into the scope ‘b’, it exits scope ‘a’. Once scope ‘a’ exits, the top scope pointer will point to scope “A” and the scope ‘a’ objects are deleted.

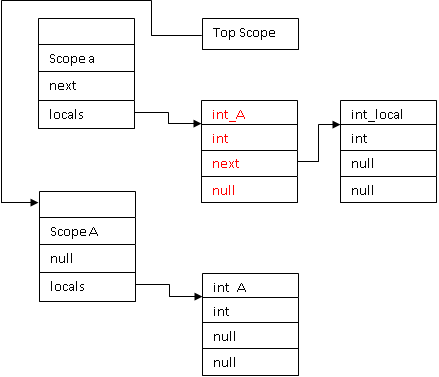


Figure 4.10 Symbol Table Structure III

When the new scope ‘b’ is entered the top scope will be changed to ‘b’ and ‘b’ will point to scope “A”.

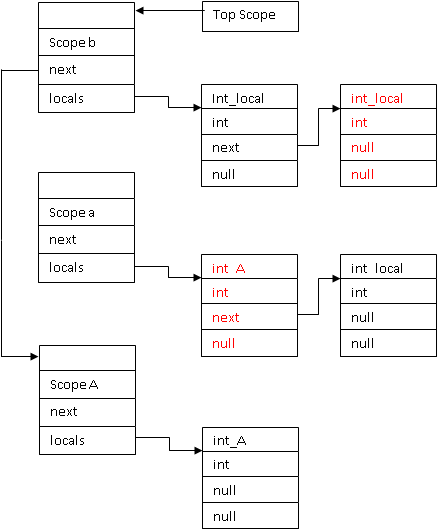
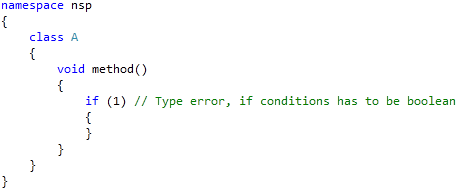
****

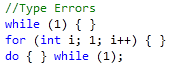
Figure 4.11 Symbol Table Strucutre IV

The job of symbol table does not end with scope checking. It also helps with the process of semantic type checking, which is a big issue for a strong-typed language like C#. The SymbolObject in symbol table stores the type information of the local variable so that when the lookup is called, we have both the name and type of the particular variable reference that we are looking for. By using this information we compare the type in each expression and statement to report the semantic error if there is a mismatch. The following code fragment shows the type check needed for the “if” condition.

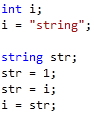


In C++, the above code fragment will be correct. But, C# is strong-typed language and it will not allow 1 to pass into the “if” condition. Our semantic checker will report the error on this since ‘1’ will have the type information as ‘System.Int32’ and ‘if’ conditional will have the type information as ‘System.Boolean'.

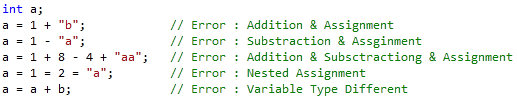
Not only the ‘if’ statement but also the following statements are enforced by semantic type checking.



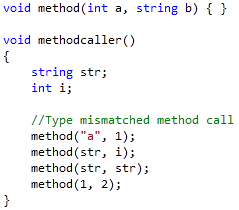
Variable type checking is the most basic part of semantic type checking. The following code fragment will show the most common semantic error found in programming.



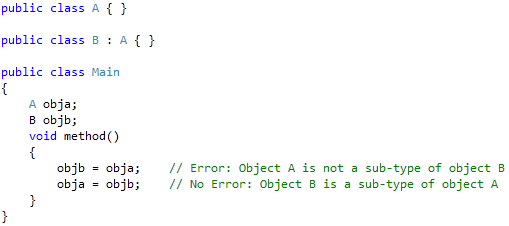
Our semantic checker will report error message with the type information of the variables that are used incorrectly.



Type checking on method call is also done in the semantic type checker. The actual and formal parameters have to have the same type information and the semantic error will show if there is a mismatch.

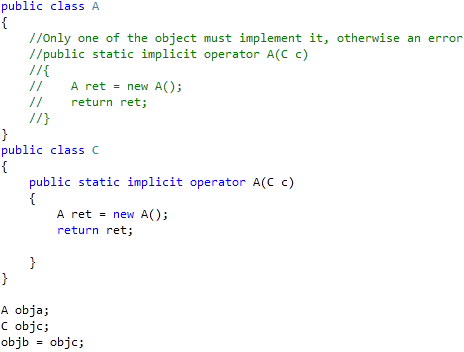


The more complicated part of type checking involved the sub-type checking. If an object B is a sub-type of an object A, the type of an object B should be able to assign to the type of object A. For example,

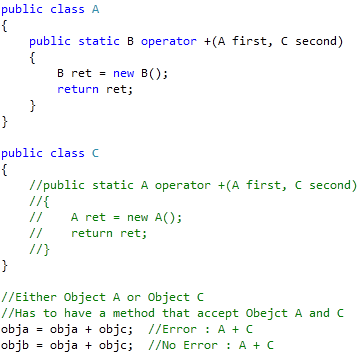


The semantic analyzer uses the typechecker object to do the sub-type checking. The typechecker object will check the object will travel thorough the object hierarchy tree to determine the sub-type of a particular type.

C# supports the implicit type converting. If one of an object A and object C have implicit type converter that convert C to A, then the type of object C can be assigned to the type of object A.



The same type checking is done for operator overloading that implement in one of the object A and object C.



The class A or class C has an add operator overloading method that accepts Object A as first operand and Object C as second operand. But, they cannot have the operator overloading method with the same parameters. If they have, then it would be ambiguous and the semantic error will be shown. The return type of the operator overloading can be any type and it will represent the type information of the result of add expression of object A and object C in above example. So that the result should be assigned to correct type, which is object B in this case.

### 4.3.6 Semantic Errors

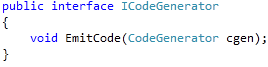
The importance of having the appropriate and sufficient information in the error messages is also an important key factor. The semantic analysis phase was designed in such a way so that we were able to extract important information for a particular item in the program that is causing an error so as to provide the user with clarity on the exact source of the problem.

Our error messages created are at a competitive standard as the error messages contain precise information to lead the users to easily find and correct the error within a small or large program. In some cases we were actually able to improve the error messages by adding in more information about the method and the class containing it thereby making it easier for users to locate the source of the problem and rectify it.

We were able to improve on the error messages provided for type conversion methods by giving further indication to the user of the exact class or struct the error is located in along with more information on the type of conversion the method is, then followed by the standard error.

## 4.4 Code Generator

The code generator has been implemented in a loosely coupled fashion, allowing users to write their own output code. Following interface is implemented by AstNodes.



CodeGenerator is an abstract class, containing methods that are required to generate appropriate output codes for the AST nodes. Below code fragment contains methods that must be implemented by the CodeGenerators.



When the compiler is executed, the code generator’s dynamic link library is automatically loaded (lsc.llvmsharp.dll) and appropriate CodeGenerator class (LLVMSharp.Compiler.CodeGenerators.LLVMSharpCodeGenerator) in instantiated using reflection. This makes it easy for users to use our project’s codebase and target different output with ease.

The compiler also loads the lsc.llvm.dll which contains helper functions needed to generate the LLVM IR codes. Each instruction derives from ILLVMCodeGenerator which contains EmitCode function.

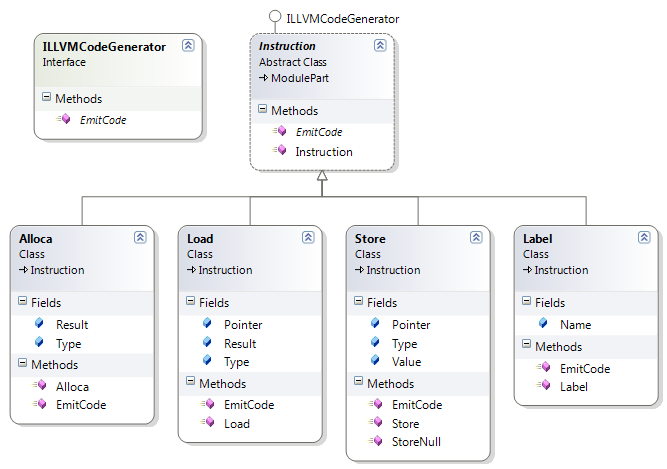
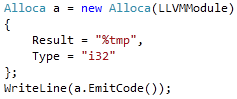


Figure 4.12 ILLVMCodeGenerator



The above code would result in the following output.



Code generation is divided into different phases, which involves generating integer table, string table, object layout, methods, constructors and entry point.

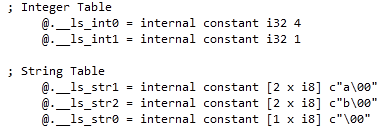
### 4.4.1 Integer and String Table Generation

While searching for full qualified types during the compilation phases, integer table and string table are created. These tables are responsible for storing unique integer and string constants respectively which are present in the source code. This allows minor optimizations to the generated code as it does not store multiple copies of the same values. Unlike integer and string, floating point constants doesn’t contain a table.

Integer table and string table is maintained internally in terms of hash table. The key contains the integer or string constant itself while the value contains the index.



The above C# code fragment would generate the following LLVM IR code.



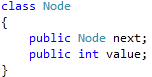
Since the values in ‘i' and ‘j’ are the same, there exist only one constant variable holding the value 1. Same holds true for string tables.

### 4.4.2 Floating Point Representation

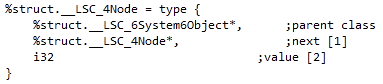
LLVM representation of floats is in a 16-bit Hexadecimal Format which matches the IEEE 754 double precision representation. In order to meet this requirement a conversion to a hexadecimal representation was done during code generation to represent floats in a compatible manner.

### 4.4.3 Object Layout Code Generation

The next phase involves object layout code generation. The object layout contains the information about the parent class along with the fields it holds.



The above class would generate the following LLVM IR code.



Classes are created as a c-style structure in which the first field contains the pointer to the parent class. If a field is a reference object, a pointer to it is created. If it is a value type, it stores the actually value.

System.String objects being primitive types are represented differently than other reference type objects. Header contains the length of the string while the actual string content is a char pointer holding the actual string value.

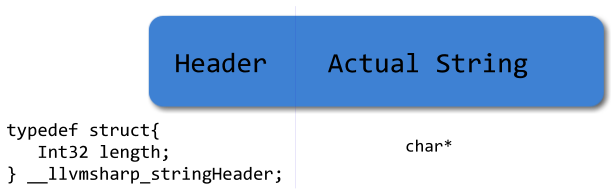


Figure 4.13 String Layout

### 4.4.4 Basic Language Constructs Code Generation

**If –Else Condition Statement**

The if-else statement construct makes use of the branching instruction “br” in LLVM, it is implemented into usage of a conditional branch module and an unconditional branch module instruction. When first encountering the condition within the “if” statement braces, the expression contained within the braces is first evaluated and the code generation for that expression is emitted.

The result of this expression is stored in a temporary register, that is then used in the branching instruction. The result of the expression at this point must always be a Boolean value of true or false. Two labels are then created accordingly, the first label created for the conditional branch is the go to point for the case that the condition expression is evaluated to true, this label marks the starting point of code execution within the if block.

However should the Boolean value that the temporary register holds as a result of the condition expression be false, the control of the program flows to the second label, which is the point of continued execution of the program after that particular if or else-if block. For each else-if construct this process is repeated.

**Loop Constructs**

There are three types of loops supported: For Loop, While Loop and Do-While Loop.  
The while loop first checks a condition to see if it should proceed to execute the code within the loop block, this is implemented by placing a unconditional branch instruction at the very beginning that will always jump to the condition expression evaluation first.

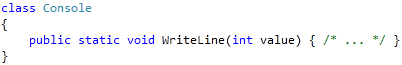
The expression that returns a Boolean value is first evaluated, the result of that expression is then stored in a temporary register. This temporary register is then loaded to be used in the branching instruction. Should the value be true, the control flow is transferred to the first label which denotes the starting point of code execution within the loop block.

At the end of the loop block another unconditional branch instruction is placed to jump to the condition check starting point. This process is repeated continuously until the condition check evaluates to false. At that point the control flow of the program will jump to the second label which is the point of code execution outside the loop block.

The code generation process for the do-while loop is somewhat similar to the while loop with the exception that the control flow is different. In the do-while loop the code inside the body of the loop is executed first. Then after that it will check the condition that determines whether or not to jump to the first label at the beginning of the code in the loop body.

### 4.4.5 Method Code Generation

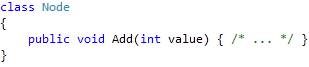
There are three possible types of methods that contain different signatures when generating the code – static methods, non-static methods and extern methods.



The above static method generates a LLVM IR function as shown below.



The parameters for the function are passed as it is defined in the C# language. In case of non-static method a special parameter holding the pointer to the current object is passed as the first parameter.



The non-static method would add this pointer as the first parameter.



Extern methods in this compiler are treated differently when generating LLVM IR. These methods are meant to be used only by the C# runtime library (mscorlib) which acts as a bridge between the actual runtime library (llvmsrtl) written in C language.

llvmsrtl (LLVMSharp runtime library) contains methods that interacts with the Operating System such as initializing memory, writing to the console output. This library is written in C language.



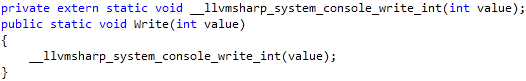
Then the C# runtime library (referred as mscorlib), defines an extern function as below.



This extern method does not contain a method body and instructs the compiler that the method with particular declaration exist in a different location. As extern methods are treated differently with the compiler, the function name does not get mangled. This allows codes written in C# to directly call a code written in C-style functions.



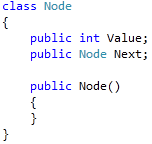
The C# runtime library, mscorlib then uses the standard C# function to create a wrapper between the extern method.



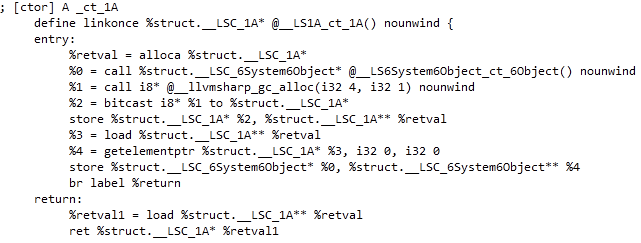
Creating extern methods as private prevents other classes from calling the function. From user’s perspective the C# runtime library - mscorlib used by the compiler seems the same as the ones used by the Microsoft C# compiler.

### 4.4.6 Constructor Code Generation

Constructors are responsible for allocating the memory and initializing default values.



If the constructor does not exist in the source code, a default constructor is automatically created. For the above example, a method for the constructor is created which takes in zero parameters and returns itself – which is the newly created Node. If it contains a parent class, appropriate constructor of parent class is also called. Once the memory has been allocated, default values are initialized to the fields. For instance, Integer values are initialized to 0, boolean values are initialized to false and reference types are initialized to null.



# Chapter 5

**Garbage Collector Internals**

In 1959 LISP started a new technique called Automatic Memory Management known as Garbage Collection (GC). This tries to address issues concerning memory leaks by requiring only explicit memory allocation but removing the need to manually de-allocate memory. The de-allocation is achieved by hunting down the un-used memory and automatically freeing them without user-code intervention.

The garbage collector implemented in this project makes use of mark and sweep algorithm which is written in C and is linked as a part of the llvmsrtl. It is only implemented on reference objects.

In mark and sweep algorithm the memory is not freed as soon as it becomes garbage, rather it frees when the system starts running out of memory or manual collect – System.GC.Collect() is fired.

Objects created by the Garbage Collector are stored on the heap. As more objects are created a directional graph of objects in memory is formed which is known as object graph or reference graph. Sample of the object graph is shown below. The nodes are the objects in memory and the edges are references one object holds to another. There can also be circular references between two nodes (Object3 and Object5) or nodes referencing themselves (Object6).

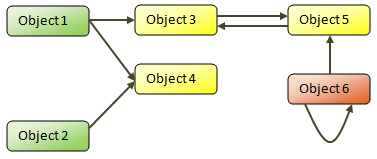


Figure 5.1 Object Graph

There are also a set of nodes in the object graph from which the references start referred to as root nodes. These are typically references of local variable on the stack or global variables. The green nodes in the above diagram are roots.

Certain nodes in the graph which have no edge referencing them are known as unreachable object. The orange node (Object6) in the diagram above is an example of unreachable object. This is a node that gets freed when GC needs to clean or free because it is not reachable from any root nodes and hence is garbage.

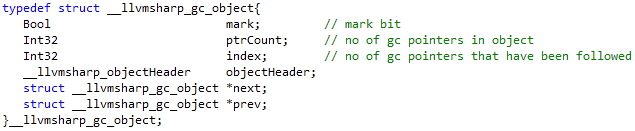
## 5.1 Storage Structure

As garbage collector requires additional information about the object, a header is also created when a garbage collectable object is created. Since the garbage collector header will be accessed frequently, to preserve the spatial locality of reference it is appended before the actual object data. So every time an allocation is asked for, the size of allocation is sizeof(header) + sizeof(type of object to be allocated). Below diagram shows the actual representation of the garbage collectable object.



Figure 5.2 Garbage Collectable Object with Headers

Internally the header is represented as a struct in C and is divided into two parts – GC header and object header. GC header contains information required for GC while object header contains information such as class information.



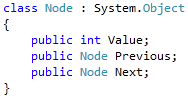
A garbage collectable object can be allocated by calling the \_\_llvmsharp\_gc\_alloc method which takes in the total size of the object and number of referenced object it holds (pointer count).



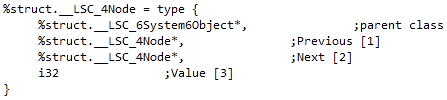
When allocating memory, if the GC undergoes certain threshold it fires the garbage collection automatically and tries to allocate the memory again and finally adds it to \_\_llvmsharp\_gc\_objects double linked list on successful allocation and returns the pointer to the actual allocated data.



Pointer count for the below class Node would be the total number of referenced objects along with the parent class which is 3.



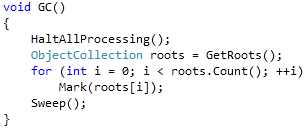
The internal structure of the object Node would be as follows.



All the referenced fields would be added to the top. This allows the garbage collector to iterate easily across other referenced object in the class Node by the help of pointer count in the GC header.

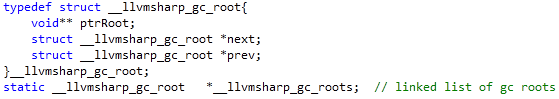
## 5.2 Mark and Sweep Alogrithm

The mark and sweep algorithm has three phases – enumeration of roots, marking all object references starting from the root and finally sweeping them. Below is the pseudo code for collecting garbage written in C.



### 5.2.1 Root Enumeration

The compiler needs to provide ways for the GC to collect the list of roots. List of these roots are usually all local variables. These local variables are marked as root when a new instance is created and unmarked as root when goes out of the method.

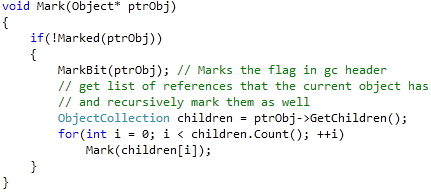


Roots are also maintained in a double linked list. Each GC root contains ptrRoot which is a pointer to the actual \_\_llvmsharp\_gc\_object. The function llvmsharp\_gc\_markRoot allows a GC object to be marked as root.



### 5.2.2 Mark

Ever garbage collectable object contains a header containing a flag used to mark that object.

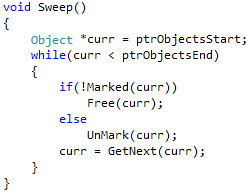


This is a recursive algorithm and its termination criteria are

1. Either a node doesn’t have any children
2. If it is already marked which ensures that we do not encounter an infinite recursion for circular references.

### 5.2.3 Sweep

Sweep starts by iterating through the entire memory and frees memory block that are not marked. It also clears the mark bit so that subsequent GC passes can correctly mark or unmark them.



# Chapter 6

**Conclusion**

The compiler we built in this project takes C# as the source language and emits the target code LLVM. In this project we have developed each phase of the compilation process from scanning to code generation. At the same time ensuring that the source code is syntactically and semantically correct according to the standard C# language specifications.

The scope of the features in the language that are currently supported are a subset of C# v1.0 that contains the main basic language constructs.

Since C# is a mainstream widely used popular language, the development of this project aids programmers by providing an alternative compilation platform that is practical, as the compiler that we have developed is small and portable.

The source code for the compiler has been available to the open source community for further usage, learning, research or further development.

# References

1. Intro - D Programming Language - Digital Mars. *D Programming Lauage - Digital Mars.* [Online] [Cited: August 5, 2009.] http://www.digitalmars.com/d/.

2. *SharpOS Wiki.* [Online] [Cited: August 14, 2009.] http://www.sharpos.org.

3. Singularity - Microsoft Research. [Online] [Cited: August 14, 2009.] http://research.microsoft.com/en-us/projects/singularity/.

4. LLVM Compiler Infrastructure Project. *LLVM Compiler Infrastructure Project.* [Online] [Cited: August 5, 2009.] http://llvm.org/Features.html.

5. Lattner, Chris and Adve, Vikram. The LLVM Compiler Infrastructure Project. *The LLVM Compiler Infrastructure Project.* [Online] March 2004. [Cited: August 8, 2009.] http://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

6. *Cosmos.* [Online] [Cited: August 14, 2009.] http://www.gocosmos.org.

7. [Online] [Cited: August 14, 2009.] http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-334,%201st%20edition,%20December%202001.pdf.

8. LLVM Assembly Language Reference Manual. *The LLVM Compiler Infrastructure Project.* [Online] [Cited: August 10, 2009.] http://llvm.org/docs/LangRef.html.

9. Standard ECMA-334 - C# Language Specification. [Online] [Cited: August 12, 2009.] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

# Appendix

## A.1 Project Dependency Graph

