Teaching Compiler Construction in Java and .NET

Master thesis

Rui Dong

May 25, 2010
Teaching Compiler Construction
In Java and .NET
Abstract

Most high-level program languages have their own compiler to interpret and compile source code, and maybe execute it in the special hardware environment. It is, however, important to understand the principles and functions of compiler so developers can write codes more effective and compact and avoiding obviously run-time error. At University of Oslo there is a course (INF2100) where students learn how to make a compiler for high-level program language RusC using Java. The Java implementation compiles the RusC source code to Rask code (Rasko) which is a machine language that can be executed in Rask machine. The main purpose of this thesis is to make .NET compiler for RusC using C# and compare with the Java implementation from INF2100 both in design and execution. The main feature of C# compiler is to generate not only machine codes which can be run on Rask machine, but also Microsoft Intermediate Language (MSIL) code, which can be assembled to Portable Executable (PE) file. Being an EXE file this PE file can then be executed in any machine installed with Microsoft .NET Framework environment. In addition experience from .NET implementation using C# might be useful for project development later in a bigger application in .NET Framework environment, especially where different programming languages are involved. On the other hand, Java is simpler to understand and easier to use as it is the standard program language currently taught at University of Oslo. It depends therefore on the goal and time assigned to Course INF2100 and the background of individual student to choose the proper program language for implementation.
Acknowledgements

First of all, I want to express my deepest gratitude to my supervisor Dag Langmyhr for constructive feedback and guidance of great value during the writing process of this thesis.

I am also grateful to Siri Lutnærs Enget and employees at Daldata AS for the support.

I also give my warmest thanks to my family for their love and support.

Folldal, May 2010
Table of Contents

TEACHING COMPILER CONSTRUCTION IN JAVA AND .NET ......................................................... 1

1 INTRODUCTION ....................................................................................................................... 9
  1.1 THE GOAL OF THIS THESIS .................................................................................................. 9
  1.2 PROBLEM AREA ................................................................................................................... 10
  1.3 THESIS OUTLINE ............................................................................................................... 10

2 BACKGROUND ........................................................................................................................... 12
  2.1 WHAT IS A COMPILER ........................................................................................................ 12
  2.2 INTRODUCTION OF RUSC AND RASK ............................................................................. 12
    2.2.1 Programming language RusC ....................................................................................... 13
    2.2.2 Computer Rask and its machine language ................................................................. 17
    2.2.3 Assembler Raskas ...................................................................................................... 18
  2.3 MICROSOFT .NET FRAMEWORK ......................................................................................... 19
    2.3.1 Common Language Infrastructure ................................................................................ 20
    2.3.2 Dynamic Language Runtime ..................................................................................... 21
    2.3.3 Common Language Runtime ..................................................................................... 21
    2.3.4 The CIL’s structure .................................................................................................... 22
    2.3.5 Assembly in .NET Framework .................................................................................... 23
    2.3.6 PE file format .............................................................................................................. 23
    2.3.7 Just-in-time compilation .............................................................................................. 24

3 JAVA IMPLEMENTATION .......................................................................................................... 25
  3.1 DESIGN ............................................................................................................................... 25
  3.2 IMPLEMENT IN JAVA ........................................................................................................... 26
    3.2.1 Scanner and Syntax ..................................................................................................... 26
    3.2.2 Code .......................................................................................................................... 26

4 .NET IMPLEMENTATION ............................................................................................................. 28
  4.1 DESIGN ............................................................................................................................... 28
  4.2 PROGRAM WITH MSIL ........................................................................................................ 29
    4.2.1 Basic understanding of MSIL ....................................................................................... 29
    4.2.2 Important directive in MSIL ....................................................................................... 30
    4.2.3 A Basic Collection of MSIL Function ......................................................................... 31
    4.2.4 Data type in MSIL ....................................................................................................... 33
    4.2.5 Others tips .................................................................................................................. 34
    4.2.6 More info ..................................................................................................................... 35
  4.3 PROGRAM MSIL FOR RUSC .............................................................................................. 35
    4.3.1 Class and Namespace ................................................................................................. 36
    4.3.2 Method with or without return value .......................................................................... 36
    4.3.3 While Loop ................................................................................................................ 37
    4.3.4 If-else statement ........................................................................................................ 39
    4.3.5 Array variable ............................................................................................................. 39

5 SUMMARY AND DISCUSSION ..................................................................................................... 41
5.1 .NET FRAMEWORK
5.2 SYNTAX TRANSLATION
5.3 COMPARISON OF RASK AND MSIL INSTRUCTIONS
  5.3.1 Data type
  5.3.2 Stack and Register
  5.3.3 Memory address
  5.3.4 RusC library function translation
  5.3.5 Loop and Branch translation
  5.3.6 Global and local variable translation
  5.3.7 Array translation
  5.3.8 Function call
  5.3.9 Concept in MSIL but not Rask
  5.3.10 Summary on comparison
5.4 PRACTICE ASPECT

6 CONCLUSIONS

7 APPENDIX
  7.1 APPENDIX B: SOURCE CODE IN THIS THESIS

8 BIBLIOGRAPHY
Figure

FIGURE 2-1: <PROGRAM> ................................................................. 13
FIGURE 2-2: <VARIABLE DECLARATION> ........................................ 13
FIGURE 2-3: <FUNCTION DECLARATION> ....................................... 13
FIGURE 2-4: <PARAMETER DECLARATION> .................................... 13
FIGURE 2-5: <FUNCTION BODY> ................................................... 13
FIGURE 2-6: <STATEMENT LIST> ................................................... 14
FIGURE 2-7: <STATEMENT> .......................................................... 14
FIGURE 2-8: <EMPTY STATEMENT> .............................................. 14
FIGURE 2-9: <FUNCTION CALL STATEMENT> ................................. 14
FIGURE 2-10: <ASSIGNMENT STATEMENT> ................................. 14
FIGURE 2-11: <ASSIGNMENT> ...................................................... 14
FIGURE 2-12: <RETURN STATEMENT> ......................................... 14
FIGURE 2-13: <WHILE STATEMENT> ............................................ 14
FIGURE 2-14: <IF STATEMENT> .................................................... 15
FIGURE 2-15: <ELSE PART> .......................................................... 15
FIGURE 2-16: <EXPRESSION> ....................................................... 15
FIGURE 2-17: <OPERATOR> ........................................................... 15
FIGURE 2-18: <VARIABLE> ............................................................ 15
FIGURE 2-19: <FUNCTION CALL> ................................................ 15
FIGURE 2-20: <SIMPLE EXPRESSION> .......................................... 16
FIGURE 2-21: <NAME EXPRESSION> ............................................ 16
FIGURE 2-22: <NUMBER EXPRESSION> ........................................ 16
FIGURE 2-24: AN EXAMPLE RASKAS CODE – NEXT.RASKAS .......... 18
FIGURE 2-26: VISUAL OVERVIEW OF CLI [3] ................................. 20
FIGURE 2-27: CLI STRUCTURE IN COMPILE TIME AND RUNTIME [10] 22
FIGURE 3-1: MODEL IN COMPILER – JAVA IMPLEMENTATION [2] .... 25
FIGURE 3-2: EXAMPLE – MINI.RUSC FILE ...................................... 26
FIGURE 3-3: EXAMPLE – MINI.RASK FILE ...................................... 27
FIGURE 3-4: EXAMPLE – MINI.LOG FILE ........................................ 27
FIGURE 4-1: MODEL IN COMPILER – C# IMPLEMENTATION ............ 28
FIGURE 4-2: A SIMPLE IL CODE – MINI.IL .................................... 30
FIGURE 4-3: MINIMUM ASSEMBLY INFORMATION IN IL FILE ......... 30
FIGURE 4-4: INVOKE A METHOD ................................................... 31
FIGURE 4-5: LOAD STRING TO STACK ........................................... 31
FIGURE 4-6: ADD – A = 3 + 5 ....................................................... 32
FIGURE 4-7: STORE VALUE – A .................................................... 34
FIGURE 4-8: STORE VALUE – B .................................................... 34
FIGURE 4-9: DECLARATION GLOBAL VARIABLE ............................. 35
FIGURE 4-10: SET VALUE TO GLOBAL VARIABLE ....................... 35
FIGURE 4-11: DECLARATION ARRAY IN C# .................................. 35
FIGURE 4-12: ARRAY DECLARATION IN IL .................................... 35
Table

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>RUSC LIBRARY FUNCTION [2]</td>
<td>16</td>
</tr>
<tr>
<td>2-2</td>
<td>RASK INSTRUCTS [2]</td>
<td>17</td>
</tr>
<tr>
<td>4-1</td>
<td>INTRODUCTION OF MSIL FUNCTION</td>
<td>32</td>
</tr>
<tr>
<td>4-2</td>
<td>DATA TYPE IN MSIL [4]</td>
<td>33</td>
</tr>
<tr>
<td>5-1</td>
<td>LIBRARY FUNCTION - ‘MAGIC’ MEMORY LOCATION IN RASK [2]</td>
<td>44</td>
</tr>
<tr>
<td>5-2</td>
<td>LIBRARY FUNCTION – MSIL TRANSLATION</td>
<td>44</td>
</tr>
<tr>
<td>5-3</td>
<td>COMPARISON BETWEEN RASK AND MSIL</td>
<td>46</td>
</tr>
</tbody>
</table>
Chapter 1

1 Introduction

Most high-level program languages have their own compiler to interpret and compile source codes, and maybe execute the codes in a special hardware environment. How compiler works is an interesting subject for some developers who want to get deeper understanding and improve their programs and application performance with more effective and compact codes. Sometimes even avoid obviously run-time error. In computer science it is important to understand the structure in high-level language and the execution process of low-level (machine-level) language. The best way to learn this is a ‘learn-by-doing’ process: making a compiler. In such project, we need to learn the specific syntax of source code language and exactly implementation language’s development specification. In addition, we have to learn assembler instructions for target executable environment as well.

1.1 The goal of this thesis

The goal of this thesis is to learn compiler construction in Rask and MSIL by implemented in Java and/or .NET.

We have already learned how to make a Java compiler for high-level program languages RusC in course INF2100 given at University of Oslo, where Java compiler translates RusC code (source code) to Rask code (machine code) and Rask code can be executed in Rask machine.

Here we are going to make a .NET compiler for RusC. C# is chosen as the implementation language, because it is a popular program language in the Microsoft .NET program language family and some of the program syntax and principles are similar to those in Java. It will be easier to start with than others in .NET program language family.

C# compiler will not only compile RusC code to Rask code, but also make that RusC code can be executed in Microsoft .NET Framework environment. We will then compare and discuss the differences in between these two alternatives, Java and C# implementation. At the end we will explore whether the C# compiler is better than the Java compiler and whether it is more suitable to use C# than Java in the project for course INF2100.

Why should RusC be executed in .NET Framework? As we know Microsoft Windows is dominating the world’s personal computer market. Most applications, softwares and programs are designed to run in the Microsoft windows operating system. Microsoft .NET Framework is intended to be used by most new applications created for the Windows platform. It has to be installed on computers with Microsoft windows operation systems before that applications designed for windows environment can be run. It is very important to understand what Microsoft .NET Framework is and how it works for developers who want to design and develop applications and softwares for Windows. .NET
Framework supports many different programming languages, like VB, C#, C++, Python, etc. How does .NET Framework execute programs written in different programming languages? To find the answer, we need to learn the .NET Framework structure, especially in the low level.

RusC is a simplified high-level language and contains basic programming principle. RusC codes cannot be executed in Microsoft .NET Framework directly and therefore needs to be compiled into the .NET Framework ‘code format’ first and then run in the Microsoft .NET environment. The basic idea of implementation is to read RusC source code and check the syntax of the source code. If the syntax is correct, then generate a Microsoft Intermediate Language (MSIL) which is a kind of binary code similar to the executable code in Java VM.

1.2 Problem area

As mentioned above, we are going to use RusC as an example program language to learn Microsoft .NET Framework low-level structure. In course INF2100 given by University of Oslo, Java is used to make a compiler for RusC. Here we are going to use C# to make a compiler and using .NET Framework as the execution environment. The following questions will be answered in this thesis.

- What is RusC and how to program with RusC?
- How to interpret, parsing RusC code?
- How does .NET Framework support for others program languages? How does .NET Framework support for RusC?
- How is Java compiler for RusC designed and implemented?
- How is C# compiler for RusC designed and implemented?
- What are the differences between Java implementation and C# implementation? What can we learn about compiler construction from both solutions?
- Does the C# implementation improve the Java implementation? Is it worth to modify the implementation in course INF2100 from using Java to C# in the future?

1.3 Thesis outline

This thesis is organized as following:

In Chapter 2, we take a look at RusC, the example program language for this project. Compiler, the programming project target, is designed to translate RusC to the machine language which can be run in Microsoft .NET Framework environment. We will also give an overview of what the Microsoft .NET Framework is and how it works.

Chapter 3 gives an overview of how RusC compiler can be implemented by Java. Java implemented compiler will translate RusC source code to Rask machine code.
Chapter 4 describes C# implementation of how to compile RusC in Microsoft .NET Framework environment. RusC is compiled to MSIL code and then could be executed in all Windows OS with .NET Framework. There is also an introduction to how to program with MSIL.

In chapter 5 we will look at the differences in Java implementation and C# implementation and some practical problems as well.

Final summary and conclusion will be given in Chapter 6.
Chapter 2

2 Background

2.1 What is a compiler

A compiler is a computer program (or set of programs) that transforms source code written in a computer language (the source language) into another computer language (the target language, often having a binary form known as object code). This definition is given from Wikipedia.

As we know, it is highly impractical to build computers so that they can directly perform a program written in a high-level programming language such as Java, .NET or Perl. Instead, computers are built so that they can only perform a limited repertoire of relatively simple instructions. However, computers can quickly perform long sequences of such instructions, roughly at a rate of one million 1000-3000 instructions per second. A high-level programming language may use natural language elements, which are easier to use, or more portable across platforms. It hides the details of CPU operations such as memory access models and managements of scope. Compiler is a means to translate high-level language which cannot directly execute on computer to machine language which can directly execute on computer.

2.2 Introduction of RusC and Rask

Programming language RusC is a kind of simplified version of program language C. The name, RusC, is an acronymic word from “Rudimentary simple C”. The RusC is invented by Professor Dag Langmyhr and Professor Stein Krogdahl from University of Oslo. This programming language is taught in course INF2100 at University of Oslo each autumn semester.

The main idea of this course, which is a large programming project, is to program a compiler. Source code is written in RusC and students will use Java to make a compiler of RusC. At the end of this course, compiler will generate an executable file Rask file which can be run at Rask computer. Here I will use C# to compiler RusC and generate an executable file with ‘EXE’ as the extension name. The result file (*.exe) can be run in any Microsoft windows operating system which has Microsoft .NET Framework.

The RusC language changes slightly from year to year. In this thesis, RusC language definition is based on the 2007 version. I give the original document link in the bibliography part. Introduction of RusC language gives here is a part of document written by Professor Dag Langmyhr and Professor Stein Krogdahl. I just quote all the diagrams and try to describe it in my words.
2.2.1 Programming language RusC

First of all, RusC-program is a collection of function/method (Figure 2-1). In between these functions, we can declare global variables. All functions are starting with keyword ‘func’. There will always be one function named ‘main’ and the execution of program will always start by calling ‘main’ function. There is only one data type in RusC – int (Figure 2-2). We can still use vector variable (one dimension array integer data type). However, all functions (Figure 2-3) are int - function, which means they can only return an int value. Pay attention to declaration of variables as there is no initial value for variables. We should always give the value before we use it. Same as other programming languages, variables have to be declared first.

![Diagram](image)

**Figure 2-1: <Program>**

![Diagram](image)

**Figure 2-2: <Variable declaration>**

![Diagram](image)

**Figure 2-3: <Function declaration>**

![Diagram](image)

**Figure 2-4: <Parameter declaration>**

Functions in RusC language may have parameters. The maximum parameter is four, and parameter cannot be a vector. The value returned by function can be a random value if there is not given an explicit value by using ‘return’ statement. RusC language does not support recursive function.

Others syntax diagram is given in figures from Error! Reference source not found. to Figure 2-22.

![Diagram](image)

**Figure 2-5: <Function body>**
Figure 2-6: <Statement list>

Figure 2-7: <Statement>

Figure 2-8: <Empty statement>

Figure 2-9: <Function call statement>

Figure 2-10: <Assignment statement>

Figure 2-11: <Assignment>

Figure 2-12: <Return statement>

Figure 2-13: <While statement>
Figure 2-14: <If statement>

Figure 2-15: <Else part>

Figure 2-16: <Expression>

Figure 2-17: <Operator>

Figure 2-18: <Variable>

Figure 2-19: <Function call>
In addition, there are five library functions for RusC as given in the Table 2-1. We can use them without special import.

Table 2-1: RusC library function [2]

<table>
<thead>
<tr>
<th>Function</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>int exit (int status)</td>
<td>Provides dump about the status ≠ 0; exit the program with the specified status value</td>
</tr>
<tr>
<td>int getchar ()</td>
<td>Read the next character from the keyboard</td>
</tr>
<tr>
<td>int getint ()</td>
<td>Read the next integer from the keyboard</td>
</tr>
<tr>
<td>int putchar (int c)</td>
<td>Write a character on the screen</td>
</tr>
<tr>
<td>int putint (int c)</td>
<td>Write a integer on the screen</td>
</tr>
</tbody>
</table>
2.2.2 Computer Rask and its machine language

A compiler translates usually machine language on an appropriate machine. For RusC, the machine language is Rasko and the machine is Rask. Rask computer is a single computer machine and it is a typical RISC (Reduced instruction set computer) computer.

Computer Rask has one processor, 32 general registers, one PC-register and one memory with 10 000 storage cell. Registers are named as R0 to R31. R0 will always store value 0, and R31 will automatically store return address. There are totally 17 assembly instructions in Rask, and are shown in Table 2-2. RA and RB are given register, and C is either a register 0 – 31 or a memory address 0 – 9999, or a positive number 0 – 9 999 999 999.

Table 2-2: Rask instructs [2]

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD Ra, Rb, C</td>
<td>Ra ← Mem[Rb+C]</td>
</tr>
<tr>
<td>2</td>
<td>SET Ra, Rb, C</td>
<td>Ra = Rb+C</td>
</tr>
<tr>
<td>3</td>
<td>STORE Ra, Rb, C</td>
<td>Mem[Rb+C] ← Ra</td>
</tr>
<tr>
<td>4</td>
<td>ADD Ra, Rb, Rc</td>
<td>Ra = Rb+Rc</td>
</tr>
<tr>
<td>5</td>
<td>SUB Ra, Rb, Rc</td>
<td>Ra = Rb-Rc</td>
</tr>
<tr>
<td>6</td>
<td>MUL Ra, Rb, Rc</td>
<td>Ra = RbX Rc</td>
</tr>
<tr>
<td>7</td>
<td>DIV Ra, Rb, Rc</td>
<td>Ra = Rb/Rc</td>
</tr>
<tr>
<td>8</td>
<td>EQ Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb=Rc, OR 0</td>
</tr>
<tr>
<td>9</td>
<td>NEQ Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb≠Rc, OR 0</td>
</tr>
<tr>
<td>10</td>
<td>LESS Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb&lt;Rc, OR 0</td>
</tr>
<tr>
<td>11</td>
<td>LESSEQ Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb≤Rc, OR 0</td>
</tr>
<tr>
<td>12</td>
<td>GTR Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb&gt;Rc, OR 0</td>
</tr>
<tr>
<td>13</td>
<td>GTREQ Ra, Rb, Rc</td>
<td>Ra ← 1 IF Rb≥Rc, OR 0</td>
</tr>
<tr>
<td>14</td>
<td>JUMPEQ Ra, Rb, C</td>
<td>IF Ra=Rb; PC ← C</td>
</tr>
<tr>
<td>15</td>
<td>JUMPNEQ Ra, Rb, C</td>
<td>IF Ra≠Rb; PC ← C</td>
</tr>
<tr>
<td>16</td>
<td>CALL Ra, Rb, C</td>
<td>R31 ← PC AND PC ← C</td>
</tr>
<tr>
<td>17</td>
<td>RET</td>
<td>PC ← R31</td>
</tr>
</tbody>
</table>

When Rask machine starts, it reads the file that tells what should be in the memory cells in the machine when execution begins. This file is in a particularly format called Rasko.
2.2.3 Assembler Raskas

Assembler for Rask is called Raskas. Raskas assembler takes Raskas code as input file and translates it to Rask instructions in Rasko format. The Figure 2-22 shows the relationship between RusC, Raskas, Rasko and Rask. A small Raskas code example is given in Figure 2-23.

![Figure 2-23: Relationship between RusC, Raskas, Rasko and Rask](image)

```
1  # A minimal testprogram:
2  # Ask user give a number and gives v+1 on screen
3
4  main:  SET          R11,'?'    #   '?'
5     CALL     putchar   # putchar(  )
6     SET          R11,' '   #   '
7     CALL     putchar   # putchar(  )
8
9     CALL     getint    # R1 = getint();
10    SET          r3,1     #   1
11    ADD          R11,R1,R3  #R11 = R1+ ;
12    CALL     putint    # putint(R11);
13    SET          R11,10   #   LF
14    CALL     putchar   # putchar(  );
15
16    SET          R11,0    #   0
17    CALL     exit       # exit();
```

*Figure 2-24: An example Raskas code – next.raskas*
2.3 Microsoft .NET Framework

.NET is Microsoft’s strategy for developing large distributed software system. A core component of .NET is the .NET framework, a component model for the Internet. [1] Microsoft .NET Framework is a software framework and includes a large library of coded solution to common programming problems and a virtual machine that manages the execution of programs written specifically for the framework. The framework's Base Class Library provides a large range of features including user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. The class library is used by programmers to produce their own applications. [3]

.NET Framework defines the data transaction rules between all program languages, and how to compile the program codes to the executable codes. In addition, .NET Framework manages application’s excitations created by any Visual Studio .NET’s program languages. When the applications run, .NET Framework provides a common programming/execution environment for all .NET program languages, not individual environment for each .NET program language. Programs written for the .NET Framework execute in a software environment that manages the program’s runtime requirements.

![The .NET Framework stack](image)

*Figure 2-25: The .NET Framework stack [3]*

The .NET Framework could be contrasted with Microsoft’s Component Object Model (COM), and with Object Management Group’s (OMG’s), and with Java. Figure 2-25 shows an extremely simplified view of the architecture and relationship between elements of the .NET Framework. In the first layer of the figure is Common Language Runtime (CLR). The CLR is the base on which all of the .NET
Framework’s other facilities are built. The CLR is a core component of the .NET Framework. It consists of the three main components: [1]

- A type system that supports many of the types and operations found in modern programming languages
- A metadata system that allows metadata to be persisted along with types at compile time and then be interrogated by other CLR compilers or the execution system at runtime
- An execution system that executes CLR programs, utilizing the metadata to perform such services as memory management.

Base Framework is above of the CLR, and it is a set of class libraries that can be used and shared by any .NET-aware language. The CLR and fundamental parts of the Base Framework are collectively known as the Common Language Infrastructure (CLI).

### 2.3.1 Common Language Infrastructure

The Common Language Infrastructure (CLI) has been standardized by ECMA. CLI is an open specification (published under ECMA-335 and ISO/IEC 23271) developed by Microsoft that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework and the free and open source implementations Mono and Portable .NET. The specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.

![Figure 2-26: Visual overview of CLI](image)
CLI Languages are computer programming languages that are used to produce libraries and programs that conform to the Common Language Infrastructure specifications. With some notable exceptions, most CLI languages compile entirely to the Common Intermediate Language (CIL), an intermediate language that can be executed using an implementation of CLI such as the Common Language Runtime (CLR), a part of the Microsoft .NET Framework, Mono, or Portable .NET. As the program is being executed by the CLR, the CIL code is compiled and cached, just in time, to the machine code appropriate for the architecture on which the program is running. Figure 2-26 gives an overview about how CLI works at .NET Framework. [8]

2.3.2 Dynamic Language Runtime

The Dynamic Language Runtime (DLR) from Microsoft is an ongoing effort to bring a set of services that run on top of Common Language Runtime (CLR) and provides language services for several different dynamic languages. These services include: A dynamic type system, to be shared by all languages utilizing the DLR services; dynamic method dispatch; dynamic code generation; hosting API. The DLR is used to implement dynamic languages like Python and Ruby on the .NET Framework. The DLR services are currently used in the development version of IronRuby, a .NET implementation of the Ruby language, and for IronPython. [31]

By having several dynamic language implementations share a common underlying system, it should be easier to let these implementations interact with one another. For example, it should be possible to use libraries from any dynamic language in any other dynamic language. In addition, the hosting API allows interoperability with statically typed CLI languages like C#.

2.3.3 Common Language Runtime

The public runtime environment is called Common Language Runtime (CLR). CLR is a core component of .NET Framework. It is Microsoft’s implementation of the Common Language Infrastructure (CLI) standard, which defines an execution environment for program code. The CLR provides the appearance of an application virtual machine so that programmers need not consider the capabilities of the specific CPU that will execute the program. The CLR also provides other important services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework. [1]

Type system – supports many of the types and operations found in modern programming languages. The type system is logical divided into two subsystems, value types and reference types. A value type consists of a sequence of bits in memory, such as a 32-bit integer. Values types include built-in values types such as int and float, and user-defined values types. Reference types combine the address of a value and the value’s sequence of bits. There are three categories of reference types exist: object types, interface types, and pointer types.

Metadata system – allows metadata to be persisted along with types at compile time and then be interrogated by other CLR compilers or the execution system at runtime. The metadata system is the essential link that bridges the gap between the type system and the execution engine. Metadata permits types defined in one language to be used in another language. Metadata and the common
type system are the facilities that help ensure language interoperability in the CLR. The execution engine requires metadata to manage objects. Managing objects includes services such as the layout of objects in memory, memory management, and security.

**Execution system** – executes CLR programs, utilizing the metadata to perform such services as memory management. When the .NET application is compiled at the first time, it will generate original code to Common Intermediate Language/Intermediate Language (CIL/IL, previously known as MSIL – Microsoft Intermediate Language) could be understood by all .NET languages. Because IL is machine independent, it can be moved from machine to machine. The CLR allows programmers to ignore many details of the specific CPU that will execute the program. It also provides other important service, including the following: Memory management, Thread management, Exception handling, Garbage collection, Security. At the execution time the Just-In-Time (JIT) compilation of (CIL/IL) will translate it to native code at current execution platform. In addition, it ensures that executing code meets the system’s security requirements. The safe execution of cod within the .NET Framework is the concept of type safety. Execution un-type-safe CIL can produce erroneous or destructive behavior within the execution system. So the security system, which is a component of the execution system, is responsible for preventing CIL from calling methods that it does not have permission to call. Alternatively, the CIL code can be compiled to native code in a separate step prior to runtime by using the Native Image Generator (NGEN). This speeds up all later runs of the software as the CIL-to-native compilation is no longer necessary. Although some other implementations of the Common Language Infrastructure run on non-Windows operating system, Microsoft’s implementation runs only on Microsoft Windows OS. Figure 2-27 shows CLI structure in details with compile time and runtime.

![Diagram of CLI structure in compile time and runtime](image)

**Figure 2-27: CLI structure in compile time and runtime [10]**

### 2.3.4 The CIL’s structure

Common Intermediate Language (Microsoft Intermediate Language or MSIL, CIL) is the lowest-level human-readable programming language defined by the Common Language Infrastructure (CLI)
specification. Languages which target a CLI-compatible runtime environment compile to CIL, which is assembled into byte code. CIL is an object-oriented assembly language and is entirely stack-based. It is executed by a virtual machine. [12]

CIL was originally known as MSIL during the beta releases of the .NET languages. Due to standardization of C# and the CLI, the byte code is now officially known as CIL. Because of the legacy, CIL is still frequently referred to as MSIL, especially by long-standing users of the .NET languages.

As we mentioned above, the source code is translated into CIL code rather than platform or processor-specific object code. It improves the portability of the software/application. CIL is a CPU and platform-independent instruction set that can be executed in any environment supporting the CLI (either the .NET runtime on Microsoft Windows operating system, or the independently derived Mono, which also works on Linux or Unix-based operating systems). CIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries. [12]

CIL is object-oriented and stack-based, which means that data is pushed on a stack instead of pulled from registers like in most CPU architectures. Therefore, when we compile a program, .NET Framework will first check the syntax of source code and then compiled to an executable file containing IL.

There is a very useful program named ILDASM.exe. It can decompile the assembly code to code in CIL. It is located at \..\Program Files\Microsoft SDKs\Windows\v7.0A\bin\ILDASM after the .NET Framework is installed in Microsoft Window 7 OS.

### 2.3.5 Assembly in .NET Framework

Assemblies are self-describing installation units, consisting of one or more files. One assembly could be a single DLL or EXE that includes metadata, or it can be made of different files, for example, resource files, metadata, DLLs, and an EXE. Installation of an assembly can be as simple as copying all of its files. Another feature of assemblies is that they can be private or shared. With COM this doesn’t exist, as all COM components are shared. After the source code generate the assembly code, .NET Framework will use JIT compilation translate the assembly code to byte code. [3]

### 2.3.6 PE file format

The Portable Executable (PE) format is a file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of window operating systems. It is the executable file generated by the compiler of any product in the .NET world. Therefore all DLL files and EXE files are PE format file. In another word, PE file is the result file generated by assembly program in .NET Framework. The term “portable” refers to the versatility in numerous environments of operating system software architecture. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, OBJ, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments. [6]
For example, when we compile a C# program without Microsoft Visual Studio embedded function ‘build’, we use command ‘csc’ compiler. In .NET Framework, PE file is the result file generated by .NET assembly, whatever the application or program developed by which .NET language. PE file format is an executable file which is written in binary code and designed for Windows NT, Windows 95 and Win32.

2.3.7 Just-in-time compilation

In computing, Just-In-Time compilation (JIT), also known as dynamic translation, is a technique for improving the runtime performance of a computer program. JIT builds upon two earlier ideas in runtime environments: bytecode compilation and dynamic compilation. It converts code at runtime prior to executing it natively, for example bytecode into native machine code. The performance improvement over interpreters originates from caching the results of translating blocks of code, and not simply reevaluating each line or operand each time it is met. It also has advantages over statically compiling the code at development time, as it can recompile the code if this is found to be advantageous, and may be able to enforce security guarantees. Thus JIT can combine some of the advantages of interpretation and static (ahead-of-time) compilation. [14]

Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution. In computer science, compile time refers to either the operations performed by a compiler (the "compile-time operations"), programming language requirements that must be met by source code to be successfully compiled (the "compile-time requirements"), or properties of the program that can be reasoned about at compile time. [14]
Chapter 3

3 Java implementation

As we mentioned in chapter 2, in course INF2100 we used Java make a RusC compiler. I will give a look on compiler design idea and how we implement it.

3.1 Design

Compiler design is from course INF2100. The model diagram is given in Figure 3-1. Course INF2100 gives a project package which includes models classes and some finished functions. There are totally six models in the package. Compiler project has been divided into three parts by Professor Dag Langmyhr and Stein Krogdahl. First part is about reading in RusC source code correctly. We should work on models CharGenerator, Scanner, Log and Error. Second part is about parsing source code. We should work on both Syntax and Log models. Third part is about generating machine code. We should work on Code, Syntax and Log models. Here comes a short description of each part.

Figure 3-1: Model in compiler – Java implementation [2]
3.2 Implement in Java

3.2.1 Scanner and Syntax

Scanner model and CharGenerator model is the first part to be finished. It is the fundamental part of the project. Scanner works with CharGenerator model to read in the source code line by line and character by character, and then translate each word of source code to correct RusC token. There are total twenty-nine tokens. We use log file for all the source code line and tokens, so we can check the Scanner part runs correctly.

Syntax model verifies each token in syntax structure, and checks whether there exist illegal syntax in source code. The ‘subway’ diagram description for syntactic RusC is given during the course, and a quota description can be found in section 2.2. Similar to other compilers, RusC compiler can also report the syntactic errors. Compiler stops parsing when error occurs and will be shown in log file. This means that only one error can be discovered at a time. Core work in Syntax phase is to understand ‘subway’ diagram and relationship with each diagram. For example, SyntaxUnit is a super class whereas ProgramUnit, DeclarationUnit, StatementUnit and ExpreElementUnit implement SyntaxUnit. FuncDeclUnit and VarDeclUnit are subclass of DeclarationUnit. DeclListUnit holds all the type declarations both for functions and variables. There are six type statement classes which implement StatementUnit: AssignmentUnit, WhileUnit, FuncCallStatementUnit, EmptyStateUnit, IfUnit, and ReturnUnit. StatmListUnit holds all statements for code block. Class ExpreElementUnit is the super class for SimpleUnit, FuncCallUnit and VarUnit. ExprementUnit holds relationship for each ExpreElementUnit object.

Syntax model and Scanner model are the precondition to generate correct assembly code for RusC.

3.2.2 Code

The last part is Code model, which generates a file with ‘.rask’ as extension and this file is the executable file in Rask machine or any machine which has Rask simulator. In Log model, Code will also generate a log file which can show how the Code works. In order to generate correct Rask code, Code depends on the source code structure built by Syntax model. Here is an example where source code is given in Error! Reference source not found., and assembly code is in Figure 3-3, and log file is in Figure 3-4.
```c
1  func main ()
2  {
3      int a, b;
4      int sum;
5      a = 17;
6      b = 28;
7      sum = a + b;
8      putint(sum);
9  }
```

**Figure 3-2: Example – mini.rusc file**

```bash
#!/usr/bin/rusc

1000000000000000 2110000000000000 1800000000000000 0 0
0 0 0 1100000000000000 5100000000000000 20
0000000000000000 3010000000000000 3010000000000000 3010000000000000 3010000000000000
2110000000000000 1000000000000000 1210000000000000 1700000000000000
```

**Figure 3-3: Example – mini.rask file**

Log file's instruction line is in the following structure.

<Label> <instruct> <Param1> <Param2> <Param3> #Comment

<table>
<thead>
<tr>
<th>Code</th>
<th>&lt;Label&gt;</th>
<th>&lt;instruct&gt;</th>
<th>&lt;Param1&gt;</th>
<th>&lt;Param2&gt;</th>
<th>&lt;Param3&gt;</th>
<th>#Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CALL</td>
<td>0 0</td>
<td>00</td>
<td></td>
<td></td>
<td># dummy main program</td>
</tr>
<tr>
<td>11</td>
<td>SET</td>
<td>11 0</td>
<td>00</td>
<td></td>
<td></td>
<td># (i)</td>
</tr>
<tr>
<td>21</td>
<td>CALL</td>
<td>0 0</td>
<td>00</td>
<td></td>
<td></td>
<td># exit</td>
</tr>
<tr>
<td>31</td>
<td>RES</td>
<td>1</td>
<td># return address in main</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>RES</td>
<td>1</td>
<td># int a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>RES</td>
<td>1</td>
<td># int b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>RES</td>
<td>1</td>
<td># int sum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>STORE</td>
<td>31 0</td>
<td>03</td>
<td></td>
<td></td>
<td># save return address</td>
</tr>
<tr>
<td>81</td>
<td>STORE</td>
<td>20 0</td>
<td>04</td>
<td></td>
<td></td>
<td># save R0</td>
</tr>
<tr>
<td>91</td>
<td>SET</td>
<td>3 1</td>
<td>00</td>
<td></td>
<td></td>
<td># R1 = R0</td>
</tr>
<tr>
<td>10</td>
<td>LOAD</td>
<td>7</td>
<td>00</td>
<td></td>
<td></td>
<td># R1 = R0</td>
</tr>
<tr>
<td>11</td>
<td>ADD</td>
<td>7</td>
<td>00</td>
<td></td>
<td></td>
<td># get sum</td>
</tr>
<tr>
<td>12</td>
<td>STORE</td>
<td>1 0</td>
<td>00</td>
<td></td>
<td></td>
<td># save R1</td>
</tr>
<tr>
<td>13</td>
<td>STORE</td>
<td>21 1</td>
<td>00</td>
<td></td>
<td></td>
<td># (address of 'main')</td>
</tr>
</tbody>
</table>

**Figure 3-4: Example – mini.log file**

From the above figure, you can see that the instructions in Rask file is the same as the instructions in log file on <instruct> part. Therefore the log file will help us to check the Rask instructions whether are correct.

All the Rasko assembler instructions (totally seventeen instructions) have been given in Table 2-2. We can find the regulation on how to translate RusC statement or expression to Rask. For example, assignment expression ‘<v> = <e>’ can be translated as ‘STORE R1, R0, Address (<v>)’. Here we have stored the calculation result of expression <e> in register R0. From the example, we got to know that it is important to store the memory address for each instruction of value.
The more details for Code model programming with Java I will give in Chapter 5. We are going to compare and discuss the compiler construction between MSIL and Rask for each instruction there.

Now we know the RusC compiler structure and how it works. Based on the design from course INF2100, the compiler is implemented using Java and executed in Java virtual machine.
Chapter 4

4 .NET implementation

In Chapter 2 there is a short introduction of the .NET Framework and how an application or program runs in .NET Framework environment. In this chapter we will look into more details about how to generate the Microsoft Intermediate Language (MSIL) code and how to get the executable file code for RusC.

4.1 Design

![Diagram](image)

*Figure 4-1: Model in compiler – C# implementation*

C# version compiler will have the same compiler principle as Java version compiler, so the basic design for C# implementation is same as Java implementation. The source code needs to be scanned...
first. Compiler will then verify the syntax of the source code, and find the illegal syntax in statement
or delection. If the syntax is all correct in compiler time, program will translate the source code to
MSIL code. Otherwise, errors will be given in log file.

Comparing the compiler design in Java and C# implementation, we can see that the first step,
programming Scanner model, and the second step, programming Syntax model, are the same. For
Code model, instead of generating a Rask file in Java implementation, C# implementation will
generate a MSIL file. This MSIL file can be assembled with program ILAsm.exe, and generate a file
with EXE as file extension name which can then be executed in any operation platform installed .NET
Framework. Except for the translation of Rasko code in Java implementation, I use Assembly model
to generate the MSIL file in C# implementation. Figure 4-1 shows models in compiler implemented in
C#.

Now let’s look at the focus of C# implementation: how to generate a legal, executable MSIL file.

4.2 Program with MSIL

The source code is converted to Intermediate Language (IL) in .NET, regardless of programming
language. IL is also known as Microsoft Intermediate Language (MSIL) or Common Intermediate
Language (CIL). It is functionally similar to the Byte Code generated by Java. It is important to know
how .NET deals with data types and how the code is converted to IL code etc in order to understand
the code emitted by .NET compiler. It is thus possible to examine the code emitted by the complier
and make necessary changes (though not needed in most cases). Such changes might not be allowed
by the high-level language but can increase the performance of source code. This also may help
debug code at low level. It is essential step for writing a compiler for .NET.

4.2.1 Basic understanding of MSIL

MSIL itself is in binary format and has, similar to other binary (executable) code, an assembly
language known as IL Assembly (ILAsm). IL Assembly is located at directory of .NET Framework 2.0
and .NET Framework 4.0 (i.e. for .NET Framework 2.0, you can find the ILAsm at \..\Windows\Microsoft.NET\Framework\v2). The instructions in IL Assembly are also similar to the
other native assembly languages. For example, ‘add’ to add two numbers and ‘sub’ to subtract two
numbers. It is obvious that .NET runtime (JIT) cannot execute the ILAsm directly. It has to be
compiled to IL code first.

IL and IL Assembly are two different things. IL means the binary code emitted by the .NET compiler
whereas ILAsm refers to the IL assembly language which is not in binary form.

ILAsm language has the same instruction set as other native assembly languages, for example Rask
instructions, but it is more complex. ILAsm language is object oriented language as well. The code for
ILAsm language can be written in any text editor such as notepad or TextPad and then use the
command line compiler (ILAsm.exe) provided by the .NET framework to compile. ILAsm language is
difficult for those programmers who have only been working with high level languages but relatively
easy for programmers using C or C++. In IL Assembly, everything is done manually, such as pushing
values to stack, managing memory etc. ILAsm language is the assembly language that deals with native Windows executables and .NET executables. [4]

So let’s see what basic instructions in MSIL are and learn the important directive in MSIL first.

4.2.2 Important directive in MSIL

A simple mini program is given in Error! Reference source not found. and I will describe MSIL directive line by line. Mini program will print a single phrase on the screen (console).

```csharp
1 // mini IL
2 .assembly extern mscorlib {}
3 .assembly Mini
4 {
5 .ver 1:0:1:0
6 }
7 .module mini.exe
8 .namespace Rusc
9 {
10 .class private mini
11 {
12 .method static void main() cil managed
13 {
14 .entrypoint
15 .maxstack 1
16 ldstr "Hello, IL Assembly Language!"
17 call void [mscorlib]System.Console::WriteLine (string)
18 ret
19 }
20 }
21 }
```

*Figure 4-2: A simple IL code – mini.il*

First, ‘.assembly extern mscorlib{}’ instructs ILAsm to use an external library (which is not written in this code, but pre compiled). In ILAsm, every statement started with ‘.’ to indicate that the statement is a special instruction or directive.

Code block shown in Figure 4-3 is used to define the assembly information, including the name of assembly and within brackets supplied with the information about the output assembly, which is the version information in this case. Of course more information about the assembly can also be provided in this block, like public key etc.

```csharp
3 .assembly Mini
4 {
5 .ver 1:0:1:0
6 }
```

*Figure 4-3: Minimum Assembly information in IL file*

After the information block, ‘.module mini.exe’ tells the module name of this assembly. As we know there must be at least one module in each assembly. ‘.method’ directive flag is used to define a method and return value type. Accessibility flag are ‘public’, ‘static’, ‘private’. Return value type can be any data type. Instruction ‘void’ means return nothing. The name of method and parameters written in one par parentheses should also be behind return value type. Instruction ‘cil
managed’ should be followed as well. It instructs the compiler to compile this as the managed code. Similar to the high-level program language, there will always be one function named ‘main’. Inside main function, there must be the instruction ‘.entrypoint’ which tells the compiler to mark this method as the Entry Point of the application, that is, the first function of the program where the execution will start with. ‘.maxstack’ is the instruction that must be given in any function, which announces the maximum number of items that will be loaded in the memory. It can also be called as evaluation stack.

The evaluation stack can be considered as a normal machine stack, which is used to store information just before the execution of a statement. After ‘.maxstack’ a number will be given to the stack. For example, for ‘.maxstack 3’ runtime will create a room of three values in the stack which can be used at any time. It doesn’t mean that we can load only three values in the stack, but means that we can move maximum three values at one time. Values are removed from the stack when processing finishes. It should also be noted that whenever the function is called or invoked, the value used in function are removed from the stack and stack space is available. This is how the Garbage Collector works in .NET Framework. Also, there is no limitation of data type to the stack. We can move any kind of data, such like string, integer, objects, etc., to stack at any time.

Statement (Figure 4-4) invokes a method which resides in the mscorlib library. Note that we have given the full signature of this method including the return type, types and also that in which library resides. We have passed the string as parameter, which is a data type but not a variable. The previous statement (Figure 4-5) loaded the string to stack and this method is using the same string to print.

```
19          call    void [mscorlib]System.Console::WriteLine (string)
Figure 4-4: Invoke a method
```

```
18          ldstr "Hello, IL Assembly Language!"
Figure 4-5: Load string to stack
```

Comments for single line in IL file is started with ‘//’, and we can also use ‘/* ... */’ block as well for multiple lines comments.

IL file is case sensitive. ‘myadd’ and ‘myAdd’ are two different functions.

‘.namespace’ and ‘.class’ are used in mini.il and will be explained in section 4.3.

### 4.2.3 A Basic Collection of MSIL Function

In Table 4-1 the basic functions of MSIL instructions are shown. It is the minimum collection of MSIL instructions or functions, and all will be used in our C# implementation project. Function name is on the left hand side and on the right hand side explanations of the function and how to use. The small example is also shown in the right hand side.
Table 4-1: Introduction of MSIL Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>add/sub/multi/div</td>
<td>Pops two values off the stack and calculate as add, subtract, multiply, divided.</td>
</tr>
<tr>
<td></td>
<td><strong>Format</strong>: add</td>
</tr>
<tr>
<td></td>
<td><strong>Example for add</strong>:</td>
</tr>
<tr>
<td></td>
<td>2                            ldc.i4 3</td>
</tr>
<tr>
<td></td>
<td>3                            ldc.i4 5</td>
</tr>
<tr>
<td></td>
<td>4                            add</td>
</tr>
<tr>
<td></td>
<td>5                            stloc.1</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Figure 4-6: add – a = 3 + 5" /></td>
</tr>
<tr>
<td></td>
<td>Constant 3 and constant 5 pops off the stack and sum will store in local variable with sequence number 1. That is (a = 3 + 5). Local variable sequence number starts with 0.</td>
</tr>
<tr>
<td>ldc/ldloc/ldstr/ldarg</td>
<td>Push a value onto the stack. The value could be a local variable (ldloc), or a string (ldstr), or a constant (ldc), or a parameter variable (ldarg).</td>
</tr>
<tr>
<td></td>
<td><strong>Format</strong>: ldloc number</td>
</tr>
<tr>
<td></td>
<td>In Figure 4-6, it gives ‘ldc.i4 3’. It means integer data type with four byte.</td>
</tr>
<tr>
<td>stloc/ststr/starg</td>
<td>Pops value off the stack, and stores it in the local variable (stloc), or a string (ststr), or a parameter variable (starg).</td>
</tr>
<tr>
<td></td>
<td><strong>Format</strong>: stloc number</td>
</tr>
<tr>
<td>br/brtrue/brfalse</td>
<td><strong>br</strong> means unconditional jump to label followed. <strong>brtrue</strong> means jump to label followed when the value in top stack is 1. <strong>brfalse</strong> means jump to label followed when the value in top stack is 0.</td>
</tr>
<tr>
<td></td>
<td><strong>Format</strong>: brtrue Start</td>
</tr>
<tr>
<td>call</td>
<td>Invoke a method</td>
</tr>
<tr>
<td></td>
<td><strong>Format</strong>: call full_signature_of_method</td>
</tr>
<tr>
<td>ret</td>
<td>Return from the method.</td>
</tr>
</tbody>
</table>
4.2.4 Data type in MSIL

Now let’s look at the data types used in MSIL. Data types of IL Assembly are shown in Table 4-2. There is no consistency in .NET data type definition in different languages. For example an integer (32 bit) in VB .NET is defined by `Integer` but in C# and VC++, it is `int`. It also has to keep in mind whether it is Common Language Specification (CLS) Compliant or not. Data type not recognized by VB .NET is not CLS Compliant. There are so many data types in MSIL and this might be one of the reasons why it works for all program languages in the .NET world.

<table>
<thead>
<tr>
<th>IL Name</th>
<th>.NET Base Type</th>
<th>Meaning</th>
<th>CLS Compliant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void</td>
<td></td>
<td>No data, only used as return type</td>
<td>No</td>
</tr>
<tr>
<td>Bool</td>
<td>System.Boolean</td>
<td>Boolean value</td>
<td>No</td>
</tr>
<tr>
<td>Char</td>
<td>System.Char</td>
<td>Character value (16 bit unicode)</td>
<td>No</td>
</tr>
<tr>
<td>Int8</td>
<td>System.SByte</td>
<td>Single byte integer (signed)</td>
<td>No</td>
</tr>
<tr>
<td>Int16</td>
<td>System.Int16</td>
<td>Two byte integer (signed)</td>
<td>No</td>
</tr>
<tr>
<td>Int32</td>
<td>System.Int32</td>
<td>Four byte integer (signed)</td>
<td>Yes</td>
</tr>
<tr>
<td>Int64</td>
<td>System.64</td>
<td>Eight byte integer (signed)</td>
<td>Yes</td>
</tr>
<tr>
<td>Native int</td>
<td>System.Intptr</td>
<td>Signed integer</td>
<td>Yes</td>
</tr>
<tr>
<td>Unsigned int8</td>
<td>System.Byte</td>
<td>One byte integer (unsigned)</td>
<td>No</td>
</tr>
<tr>
<td>Unsigned int16</td>
<td>System.UInt16</td>
<td>Two byte integer (unsigned)</td>
<td>No</td>
</tr>
<tr>
<td>Unsigned int32</td>
<td>System.UInt32</td>
<td>Four byte integer (unsigned)</td>
<td>No</td>
</tr>
<tr>
<td>Unsigned int64</td>
<td>System.UInt64</td>
<td>Eight byte integer (unsigned)</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Data Type</td>
<td>Description</td>
<td>Implemented</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>native unsigned int</td>
<td>System.UIntPtr</td>
<td>Unsigned integer</td>
<td>Yes</td>
</tr>
<tr>
<td>Float32</td>
<td>System.Single</td>
<td>Four byte floating point</td>
<td>No</td>
</tr>
<tr>
<td>Float64</td>
<td>System.Double</td>
<td>Eight byte floating point</td>
<td>No</td>
</tr>
<tr>
<td>Object</td>
<td>System.Object</td>
<td>Object type value</td>
<td>Yes</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>Managed Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>*</td>
<td>System.IntPtr</td>
<td>Unmanaged pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>Typedef</td>
<td>System.Typed</td>
<td>Special type that holds data and explicitly indicates the type of data</td>
<td>Yes</td>
</tr>
<tr>
<td>Array</td>
<td>System.Array</td>
<td>Array</td>
<td>Yes</td>
</tr>
<tr>
<td>String</td>
<td>System.String</td>
<td>String type</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Not all data types above are implemented in our project. Since there is only one type `int` in RusC, `int32` is the only data type used here.

### 4.2.5 Others tips

#### 4.2.5.1 Using Local Variables

*Instruction* `.'locals init (…)'` *is used for local variables* and has to be defined for each local variable. The instruction `.'locals init (int32, int32, int32)'` means there are three local variables and all have the type `System.Int32`. We can also declare variables by name, for example, `.'locals init (int32 a, int32 b, int32 c)'`.

![Figure 4-7: Store value – A](image)

![Figure 4-8: Store value – B](image)

The code in Figure 4-7 means to push constant 25 to the stack, and pop off the value to the first local variable (local variable sequence number 0). Then we can access them like Figure 4-8.

In C# implementation, two extra local variables are declared, one for comparison result value with data type `bool`, and one for function return value with data type `int32`. These two variables have no variable name and are not declared by source code.
4.2.5.2 Using Global Variables

Normally in a program project, we will use global variables to store data and access them in different class or scope. In MSIL, to declare global variables use ‘.field’ directive. We can also give accessibility flags such like private, public and static as well.

Accessing global variables is more complex than local ones, since it should give full signature when variable is decelerated. The following code snippet (Figure 4-9 and Figure 4-10) are declaration and accessing examples.

```
.field private static int32 a

// declaration global variable
```

```
// assignment value to global variable a
ldc.i4 10
stsfd int32 Rusc.mini::a

// Set value to global variable
```

**Figure 4-9: declaration global variable**  **Figure 4-10: Set value to global variable**

4.2.5.3 Using Array

Array variables can be either global or local variables. For example ‘int32[]’ for int32 array. With ‘newarr [mscorlib] System.Int32’, a reference link can be made from an array variable to a new array object. ‘[mscorlib] System.Int32’ gives directive to call extern library. Example here is reference to a System.Int32 data type.

For a given source code in Figure 4-11 MSIL code will be written as in Figure 4-12.

```
int[] a = new Array[8];
```

```
ldc.i4 8
newarr [mscorlib] System.Int32
stloc a

// Declaration array in C#
// Array declaration in IL
```

**Figure 4-11: Declaration array in C#**  **Figure 4-12: Array declaration in IL**

4.2.6 More info

There are also other important and useful MSIL instructions, such as creating user defined classes and namespaces, creating and using class objects, and scope of the objects, etc. These will not be mentioned here as the purpose of this thesis is to make MSIL code for RusC. We will focus on how to make RusC MSIL file depending on the RusC programming parsing in the next section.

4.3 Program MSIL for RusC

After we understand the basic functions and how to program with MSIL, we can begin with translation of RusC as the original programming language to MSIL. I will emphasize those MSIL directives and instructions that need to be implemented. Others IL data types, i.e. Object, Float, Char, Bool, String, Typeof, etc. and instructions, i.e. creating properties, creating windows form, using class object etc. will not be mentioned here.
4.3.1 Class and Namespace

There is no definition of Class and Namespace in RusC. So we have to implement it in IL code of RusC. It is allowed to declare global variables in RusC, and global variables can be accessed in any functions. We therefore need to use Namespace concept to create an accessibility scope for source code, and Class concept for the functions scope. In mini.il file (Figure 4-2) namespace and class are used, although there is no global variable in code. As an example the following code snippet in Figure 4-13 gives more details for another function named function1.

In C# implementation project, I use a fixed namespace name ‘Rusc’ to avoid name problem and signature reference problem for namespace.

```csharp
1 .namespace Rusc
2 {
3   .class private primes_test
4   {
5     .field private static int32[] prime
6     .field private static int32 LF
7     .method static int32 function1() cil managed{
8         ...
9     }
10    }
11   .method static void main() cil managed {
12       .entrypoint
13       .maxstack 4
14       ...
15     }
16 }
17 }
```

*Figure 4-13: Namespace and class in IL*

4.3.2 Method with or without return value

The method of declaring methods in ILAsm language is almost the same as that in high-level program language i.e. C#. We know how to declare a method in the previous section. However the return value isn’t mentioned. Methods in ILAsm language will always finish with instruction ‘ret’ regardless of return value. The return value from function should be stored in the stack which was returned to the calling method, and then the ‘ret’ instruction is issued immediately. We can use an extra local variable to store the return value, as described in section 4.2.5.1, or write MSIL code shown in Figure 4-4. An example code in Figure 4-14 shows the method with return date type int32 and value of the sum calculating by two parameters. Both work correctly. An extra local variable is chosen to store the return value in C# implementation, since it makes the MSIL code more readable.
4.3.3 While Loop

Loop is nothing but the repetition of the same block of code again and again [4]. (I hope that this is correct ref) It involves the branching which depends on the value of a variable called loop index. There is only one type loop in RusC, while-loop (based on RusC language version 2008). In ILAsm ‘bgt’ (>), ‘bge’ (≥), ‘blt’ (<), ‘ble’ (≤), ‘bne’ (#), ‘beg’ (==) are used as comparison instruction and hereby named ‘comparison version 1’, whereas ‘cgt’ (>), ‘cge’ (≥), ‘clt’ (<), ‘cle’ (≤), ‘cne’ (#), ‘ceg’ (==) ‘comparison version 2’.

Instruction ‘bgt label_name’ compares two operands. If first operand is greater than second, jump to the target instruction with given label name; otherwise executes next instruction. Instruction ‘cgt’ will also compare two operands, but store the result value. That is, if the first operand is greater than second, push 1 (int32) to evaluation stack; otherwise, push 0 (int32) to evaluation stack. Instruction ‘cgt’ must work with instruction ‘brtrue’. Instruction ‘cgt’ cannot direct to branch directive as ‘bgt’. Therefore, there need an additional instruction to check the value in stack which pushed by instruction ‘bgt’. Then use instruction ‘brtrue’ or ‘brfalse’ to decide to jump to which label marked instructions. The while-loop C# in Figure 4-16 will be translated into IL code in Figure 4-17 using comparison version 1 and in Figure 4-18 using comparison version 2.
```csharp
int i = 0;
while (i < 5)
{
    System.Console.WriteLine(i);
    i++;
}
```

**Figure 4-16: C# while loop example**

```csharp
.method static void WhileLoopExample() cil managed {
    .local init(int32 start, int32 stop)
    .maxstack 2
    ldc.i4 4   //init stop
    stloc stop
    ldc.i4 0   //init start
    stloc start
    Begin:
    //check loop index
    ldloc start
    ldloc stop
    bgt Exit
    ldloc start
    call void [mscorlib]System.Console::WriteLine(int32)
    //increase the loop index(Counter)
    ldc.i4 1
    ldloc start
    add
    stloc start
    br Begin
    Exit:
    ret
}
```

**Figure 4-17: Comparison version 1 IL loop**

```csharp
.method static int32 WhileLoopExample() cil managed {
    .maxstack 4
    .locals init (bool int32, int32 loopNumber, int32 counter)
    ldc.i4 0
    stloc counter
    ldc.i4 5
    stloc loopNumber
    br WhileTest1
    WhileStart1:
    ldloc counter
    call void [mscorlib]System.Console::Write(int32)
    ldloc counter
    ldc.i4 1
    add
    stloc counter
    WhileTest1:
    ldloc counter
    ldc.i4 5
    cli
    stloc 0
    ldloc 0
    brtrue WhileStart1
    ret
}
```

**Figure 4-18: Comparison version 2 IL loop**
ILDAsm is a .NET program which translate PE file to assembly language. ILDAsm will always translate condition expression statement into comparison version 2. Therefore, comparison version 2 is used in the implementation using C# here.

### 4.3.4 If-else statement

Translation of If-else statement depends on the value of condition expression of if-test. The translation of condition expression will also use comparison instructions. Comparison version 2 is used to decide which branch code block should be executed in next step. The RusC code in Figure 4-19 is an example for If-else statement example, and the corresponding MSIL code is in Figure 4-20.

```csharp
35   if (var1 > var2){
36       putint(var1);
37   }else{
38       putint(var2);
39   }
```

*Figure 4-19: if-else statement in RusC*

```csharp
44   ldc var1
45   ldc var2
46   cg
47   ldc.i4.0
48   ceq
49   stloc.0
50   ldc.0
51   brtrue ElseStart2
52   ldc var1
53   call void [mscorlib]System.Console::Write(int32)
54   br IfFinish2
55   ElseStart2:
56   ldc var2
57   call void [mscorlib]System.Console::Write(int32)
58   IfFinish2:
59   ret
```

*Figure 4-20: If-else instruction in MSIL*

### 4.3.5 Array variable

Since there is no `new` keyword to declare array variable in RusC, array length is given directly in source code, such as `int prime[1001]`. For local array variable, an array object with function is created directly, showed as Figure 4-12. However what about array object for global variables? My suggestion is to execute `new` at `main` method, because only main method has the program entry point. Others methods then could access global array variables any time at runtime.

Compared to normal variables it is different to set value to array object with given array index. IL code has to state first which array object is going to be accessed and what array index is, and then
execute add or others calculation instruction. Finally set the result to the array object with given array index, as shown in example in Figure 4-21 and Figure 4-22.

Figure 4-21: RusC code—assign value to array  

```c
9  a[7] = 3 + 5;
```

Figure 4-22: IL code – assign value to array  

```
13  ld sidew int32[] Rusc.mini::a
14  ldc.i4 7 // array index: a[7]
15  ldc.i4 3
16  ldc.i4 5
17  add
18  stelem.i4
```
Chapter 5

5 Summary and discussion

As stated in earlier section, RusC is a simplified high-level program language. Thus there are not many instructions, date type and MSIL concept involved in the translation to MSIL code in this project using C# implementation. The discussion is divided into three parts: first, whether it is worth to learn .NET Framework environment; second, what are the differences in syntax implementation using Java and C#; and at last, what are the differences in MSIL and Rask translation. The first part focuses on the theoretical value of C# implementation, and the rest parts focus on the practical experience.

5.1 .NET Framework

.NET Framework structure and component have been introduced in chapter 2. The focus of this project is on the low level component CLR and how it works for various applications developed in different program languages. As we all know, .NET Framework is the most important part in Microsoft concept since all the technologies developed by Microsoft, whether the older ones as VB or newer ones as SilverLight, RIA, LinQ, are based on .NET Framework. It is therefore crucial to learn and understand .NET Framework for software developers using Microsoft technology. As Microsoft system is widely used .NET Framework should definitely be introduced to the students at Institute of Informatics at University of Oslo.

5.2 Syntax translation

The syntax translation of RusC is implemented in the same way by Java or C#. The main idea comes from course INF2100. Syntax structure is shown in Chapter 2 and relationship among classes for Syntax model is described in chapter 3 as well. Converting figure structure to class structure is then quite easy.

From programming respect Java and C# implementation are quite similar in syntactic part. There are small differences in between, such as declaration for array in Java ‘int[] new_arr;’ and ‘int new_arr[]’ are both legal while in C# array definition only ‘int[] new_arr;’ is accepted. Not all the syntactic difference between Java and C# are listed here, since it is easy to find explanation from programming books. It is however interesting to look at the definition of enum data type. In Java, functions can be defined in enum data type but it is not possible in C#. In Figure 5-1 and in Figure 5-2 shows the difference in enum data type for Java and C#.
5.3 Comparison of Rask and MSIL instructions

Rask machine has only 32 registers. In Rask instructions, the number of register number is needed for correct storing or calculating. The precise memory address in instructions is also necessary. However, this is not the case in MSIL, where there is no register in instruction but stack concept is used. Stack can be accessed with variable name or stack sequence number. Instruction '.maxstack' gives maximum room number which can be accessed at one time in the Java implementation. In .NET Framework there is an algorithm to evaluate the possible value of maximum stack number. It is important to have a proper maximum room in runtime, especially for large program and application. Such algorithm was however not implemented here. In my implementation, the maximum stack number was chosen to be the result of 2 plus the number of local variable.

Source code is translated into MSIL using C# implementation. In case of Java implementation, it translates the source to the Rask code which can directly be run in Rask machine. It is not so easy to translate RusC to MSIL, even though RusC is a simplifed high-level language. MSIL is designed for all Microsoft high level languages, and it can handle large program and application.

Now we come to the details on differences between Rask and MSIL, and these differences make the C# implementation more difficult.

5.3.1 Data type

In Rask, we need not to give data type in translation, because there is no other data type in Rask definition. It is different for MSIL which has many data types. Since RusC language has only one data type, int, all variables and constants translated in MSIL are int32 data type. We should point out
which data type variable is in translation on set values, invoke functions, and deal with return value of functions call, etc.

### 5.3.2 Stack and Register

In Rask, register is used to store value and operate the calculation between registers, while in MSIL, stack is used. As mentioned above, ‘.maxstack’ instruction is a new concept for Rask. In Rask there are maximum three registers that work at same time. In MSIL, calculation instructions, i.e. add, sub, multi and div, are working with two stacks at same time. In my implementation, I did not deal with it well. The proper maximum number to access stack at same time might be 4 for RusC compiler, because RusC has not parenthesis priority calculation and multi-dimension vector.

Following is an example code to show the instructions on translating Rask code and MSIL code. For RusC code ‘a =17; b =28; sum = a + b;’ Rask translation result is shown in Figure 5-3 and MSIL translation in Figure 5-4.

#### Figure 5-3: Rask translation

```plaintext
Code 10: 201000000000017    SET 1 0    17 # R11 = 17
Code 11: 301000000000003    STORE 1 0    5 # a =
Code 12: 201000000000018    SET 1 0    20 # R11 = 20
Code 13: 301000000000004    STORE 1 0    6 # b =
Code 14: 101000000000005    LOAD 1 0    5 # R11 = a
Code 15: 201010000000000    SET 3 1    0 # R0 = R1
Code 16: 101000000000006    LOAD 1 0    6 # R11 = b
Code 17: 401010000000003    ADD 1 1    3 # R1 = R1 + R3
Code 18: 301000000000007    STORE 1 0    7 # sum =
Code 19: 101000000000007    LOAD 1 0    7 # get sum
Code 20: 211010000000000    SET 11 1    0 # R11 = sum
Code 21: 16000000000009994    CALL 0 0    # print(R11)
```

#### Figure 5-4: MSIL translation

```plaintext
21     ldc.i4.17
22     stloc a
23     ldc.i4.28
24     stloc b
25     ldloc a
26     ldloc b
27     add
28     stloc sum
29     ldloc sum
30     call void [mscorlib]System.Console::Write(int32)
```

We can see that the translation regulation is same. LOAD or STORE constant first, then calculate two register or stack. Store or push the result to register or stack. Thus it is not big difference on using register in Rask and stack in MSIL.

### 5.3.3 Memory address

It should be pointed out that a value is not assigned to a variable directly in Rask. It uses memory address to load and store variable’s value. It is a kind of address reference to access variables. This is not obviously used in MSIL. MSIL does not care where variables are located in memory. It accesses variables through the name or signature of variables.
5.3.4 RusC library function translation

It is not necessary to translate library functions in Rask, since it is a kind of embedded function in Rask instruction constructor. Rask has reserved memory address for these functions (which is called ‘magic’ memory location in Rask shown in Table 5-1).

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>9990</td>
<td>Exit(R_{11})</td>
</tr>
<tr>
<td>9991</td>
<td>Getchar()</td>
</tr>
<tr>
<td>9992</td>
<td>Getint()</td>
</tr>
<tr>
<td>9993</td>
<td>Putchar(R_{11})</td>
</tr>
<tr>
<td>9994</td>
<td>Putint(R_{11})</td>
</tr>
</tbody>
</table>

The library functions in MSIL, on the other hand, need to be translated using .NET Framework external library. Table 5-2 shows how to translate RusC library functions. These MSIL translation need to work with value in stack together, since some of the library functions have parameters to carry.

<table>
<thead>
<tr>
<th>Function</th>
<th>MSIL instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit(R_{11})</td>
<td>ret</td>
</tr>
<tr>
<td>Getchar()</td>
<td>Call int32 [mscorlib]System.Console::Read()</td>
</tr>
<tr>
<td>Getint()</td>
<td>call int32 [mscorlib]System.Console::Read()</td>
</tr>
<tr>
<td>Putchar(R_{11})</td>
<td>call void [mscorlib]System.Console::Write(char)</td>
</tr>
<tr>
<td>Putint(R_{11})</td>
<td>call void [mscorlib]System.Console::Write(char)</td>
</tr>
</tbody>
</table>

5.3.5 Loop and Branch translation

There is only one type loop in RusC: While-loop. The condition test expressions can only have operators like ‘equal’, ‘notEqual’, ‘greater’, ‘greaterEqual’, ‘less’, ‘lessEqual’. There are two different ways to implement comparison IL instructions, named as comparison version 1 and comparison version 2 as mentioned in section 4.3.3.

About the loop and branch (if-else) translation, there is not big difference in between, especially on comparison version 2. Rask stores also the result after compared two values in register. Compared with MSIL, Rask loop and branch translation is relatively easier. Comparison version 2 is more similar to Rask instructions, but need more instructions to implement for comparison. MSIL will use another instruct ‘ceq’ to test the value after comparison of greater or less. It is important to set the right jump label. Label names like ‘Start’, ‘Exit’ are used in this project to mark the jump label.
For example, there is an instruction ‘GTR Rₐ, Rₐ₊, Rₐ’ to deal with comparison ‘greater than’. In MSIL, instruction \texttt{cgt, ceq and brtrue} are needed to work together for getting the correct jumping label. MSIL example about this is in Figure 4-20, and RusC code is in Figure 4-19. The relative Rask code for same RusC code is given in Figure 5-5.

Even though the instructions are different, translation logic is the same: compare the first and second value in register or stack, jump depended on the comparison result.

In MSIL, the comparison value is stored in a stack. Therefore a local variable with data type \texttt{bool} is declared to store the comparison value. This was done at the beginning of the declaration list in my implementation.

### 5.3.6 Global and local variable translation

There is no difference for Rask to access a global or local variable, since it uses memory address to reference the right variable. For MSIL, it has concept on namespace and class and different instruction is used to access global or local variable. Examples are given in Figure 4-8 and Figure 4-9.

### 5.3.7 Array translation

There are limited uses of array in RusC. For example, it cannot be used in function’s parameters. There is also no ‘new’ word for array declaration. In MSIL, it can however be solved easily in local variables declaration by creating array object directly after declaration. But for global variables declaration, it is difficult to know where the first time accessing array reference in program occurs. In this project all the global array variables were created inside main class and program search the list of all the global variables first. If an array variable is found, array object will be created directly before the function body translation. In Rask, array variable translation needs to allocate memory room at once when the array variable is declared.

Besides the declaration array, it is also different to assign value to array in Rask and MSIL. In Rask, we use the address of array variable index 0 add array index to access the correct element in array. In MSIL, we need not to calculate the array element memory address self, just give the correct array index number, then MSIL will get the reference self.
5.3.8 Function call

In Rask, instruction register R31 is used to store the return value. MSIL is almost the same as Rask code. MSIL will return the value on the evaluation stack.

In Rask, function call is translated by memory address, whereas MSIL translation is more complex. In MSIL, a function with full function signature is invoked. The full function signature includes the function return value type. This means if function has return value with type integer, signature should begin with ‘int32’; otherwise, ‘void’ should be given. Function signature given in function call should be the same as function declaration. If there is a function with return value, MSIL will create a local variable to store return value. I learned using ILDAsm.exe. I tried also without a special local variable to store return value. It works as well. But with the special local variable, it will be more readable for MSIL code. This is how it is implemented here. For definition correct for function with or without return value, RusC compiler should check ‘return’ statement. It is not implemented here. If a function is called from another function with return value, program should also verify whether it has the return type for function definition and return statement inside function code block or not. This verification is not implemented here, either. A return statement translation is used instead, where return value will load to the stack if there is a return statement inside a function; otherwise, do nothing. This created some problems during test but kept unchanged due to the limited time. This part definitely could be improved.

5.3.9 Concept in MSIL but not Rask

There are some instructions that were implemented in C# version compiler but do not exist in Rask, such as flag instruction, ‘static’, ‘public’, ‘private’, etc.

5.3.10 Summary on comparison

The biggest differences between these two implementations are how to translate RusC code to MSIL code and Rask code, respectively. Implementation differences between Rask and MSIL are shown in Table 5-3.

Table 5-3: Comparison between Rask and MSIL

<table>
<thead>
<tr>
<th>Compare item</th>
<th>Rask translation</th>
<th>MSIL translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global and local</td>
<td>Access by memory address</td>
<td>Different instruction</td>
</tr>
<tr>
<td>variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>Access by memory address with</td>
<td>Different instruction with</td>
</tr>
<tr>
<td></td>
<td>array index</td>
<td>array index</td>
</tr>
<tr>
<td>Library function</td>
<td>Give memory address</td>
<td>External library from .NET</td>
</tr>
<tr>
<td>Loop</td>
<td>Simple instructions with correct jump label</td>
<td>Complex instructions with correct jump label</td>
</tr>
<tr>
<td>If-else branch</td>
<td>Simple instructions with correct jump label</td>
<td>Complex instructions with correct jump label</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Data type</td>
<td>No data type definition</td>
<td>Many</td>
</tr>
<tr>
<td>Computer hardware</td>
<td>Register</td>
<td>stack</td>
</tr>
<tr>
<td>Memory</td>
<td>Use memory address to access variable and function call</td>
<td>No memory use. Has special instructions</td>
</tr>
<tr>
<td>Flag instruction</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 5.4 Practice aspect

MSIL is very complex, and there is not much information in literature, books and articles. Most of the information is also written for older version of .NET Framework. Help function given by .NET Framework itself, i.e. ILDAsm.exe and Peverify.exe is the best support I got for this project.

ILDAsm.exe is useful to understand MSIL structure. Entering ‘ILDAsm.exe SomeProject.exe /out: SomeProject.il’ in command line and .NET SDK will generate IL code of any of .NET EXE file. The output file from ILDAsm was used to compare my MSIL solution to find the mistakes translation.

There is another useful tool in .NET SDK: peverify.exe. Pre Verify tool do not use source code to verify the assembly, but takes an exe to examine whether the compiler has emitted the invalid code. This was used in testing of this project.
Chapter 6

6 Conclusions

The main purpose of this project is to implement a complier for RusC using C# and then compare with the earlier implementation using Java. We will see whether the .NET implementation improves the Java implementation, and whether it is worth to modify the implementation in course INF2100 from using Java to .NET.

The advantage of the Java implementation is: Java implementation was an ‘honest’ compiler. It compiled RusC source code to Rask code, which can be executed on Rask machine. It is clear how a compiler works in original routine: compiler translates RusC source code to Rasko code, and Rasko code is exactly the machine code (native code) for Raskas assembled by Raskas.

The .NET implementation using C# compiler, on the other hand, was more flexible. .NET implementation can not only translate the RusC code to Rask code, but also translate to MSIL code. Thus the RusC source code not only could be executed in Rask machine, but also in any Microsoft Windows OS with Microsoft .NET Framework. It therefore enlarges the executable environment for RusC, and improves the RusC language’s portability. However, understanding of CLR and the core component of .NET Framework is more time consuming and might be overwhelming for new beginner and/or small projects where the main focus is not .NET framework.

The differences between Rask instruction translations and MSIL are discussed in chapter 5. Clearly MSIL is more complex and difficult than Rask during the implementation. However all .NET program languages must be translated to MSIL first, and then be assembled to executable file. This project with .NET implementation helps us to understand .NET Framework. Thus it is easier to manage and development project for big application, especially application involved with different programming languages in .NET Framework environment. .NET implementation gives also understanding on how to execute a program or application cross platform. This is not the case in Java implementation. On the other hand it takes more time for .NET implementation, especially for students who have no previous experience in .NET program environment.

If the purpose of course INF2100 is focused on understanding how compiler works, using Java as an implementation tool is probably a better choice. If the focus is on how to develop application in big project with several development languages, and understanding the .NET Framework program environment, how to improve execution environment for high-level program languages, .NET implementation is of course a better choice and more challenging, but more time-consuming.
7 Appendix

7.1 Appendix B: Source Code in this thesis

Source code of C# implementation and RusC code examples can be found online at the following address: http://heim.ifi.uio.no/~ruid/master/
8 Bibliography


http://www.uio.no/studier/emner/matnat/ifi/INF2100/h07/undervisningsmateriale/kompendium.pdf


[4] Introduction to IL Assembly Language


[6] C# Classes—Threads, WinForms and XML.


[17] Interpreted language – Wikipedia, the free encyclopedia. 


[26] .NET assembly – Wikipedia, the free encyclopedia.

[27] .NET assembly – Wikipedia, the free encyclopedia.


   accessed 7-February-2010].

   accessed 7-February-2010].

   February-2010].