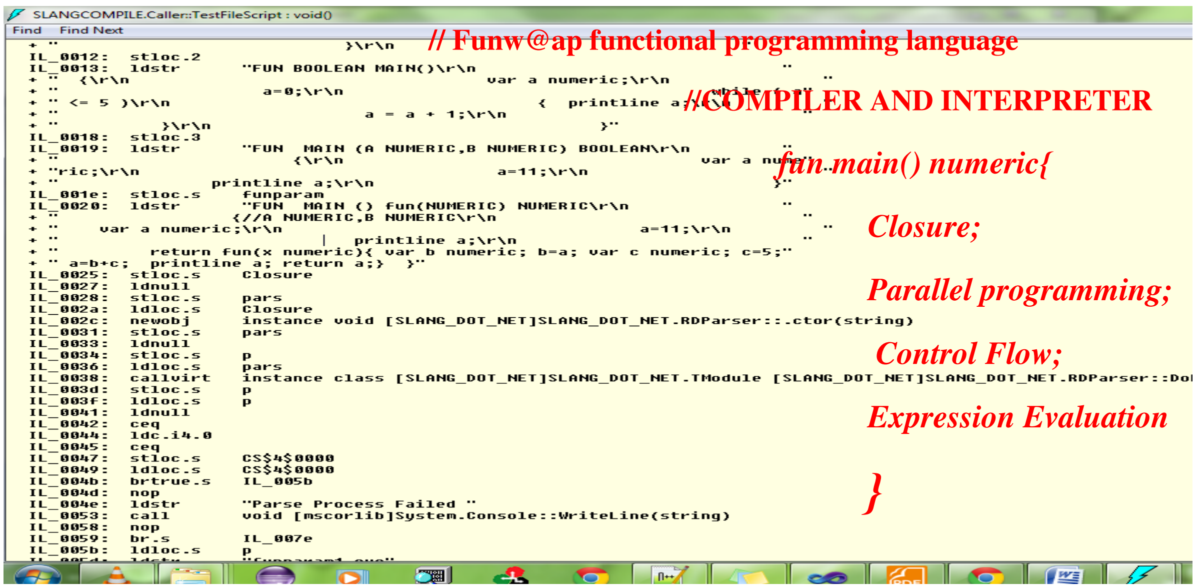
FUNW@AP

Master Program in Computer Science and Networking

Advanced Programming Course Final Project AY 2014/15



Gezahgn, Tsinu Assefa

Date of Submission Jan-14-2015

Table of Contents

Introduction

General Overview

FUNWAP is a Domain specific language which is functional programming and has futures such as control flow statements which enabling your program to *conditionally* execute particular blocks of code. (if else statement, while loop), high order function and parallel programming to execute code in asynchronous manner.

Language basics

***Variables*:-** have name and type, and used to store specific values.

- Variables declared as var keyword followed by identifier name and type

Syntax **var** IDENTIFIER <type>;

- **Data types**: - Fun@ap support 3 data types i.e. *numeric* to store decimal number (IEEE 754 floting point), *string* to store string values, and *boolean* to store Boolean values.

***Operators*:** FUNap has arithmetic, logical and relational operators which are similar to C.

- **Arithmetic operators**: - (+,-,*, /) can be either unary or binary operator
- **Relational operators** :-(==,!=,>,>=,<,<=)
- **Conditional operators**: - (&&, ||)
- **Logical operators**:- (&&,!,||)

Expressions: - is a construct made up variables, operators and method invocation, which are made according to the syntax of the language. An expression **evaluate** to a single value.

Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution.

Some concepts

- **Asynchronous programming (Async)** is a means of parallel programming in which a unit of work *runs separately from the main application thread and notifies the calling thread of its completion, failure or progress*.
- **Closure** is a function or reference to a function together with a *referencing environment*—a *table storing a reference to each of the non-local variables* (also called free variables or up values) of that function.

- ❖ **Conditional**: - perform different computations or actions depending on whether a programmer-specified Boolean *condition* evaluates to true or false.
- ❖ **Iteration**:- Executing a block of statements repeatedly based on the given condition.

Design

Grammar

Grammar is a set of rule / syntax for the funw@ap that describes which expression and statement are valid for the language. The semantics of BNF is used to express the grammar of this language

Considered issues before writing the grammar

- Ambiguity
- Left recursion
- Left factor

<Module>::= {<Procudure>}$^{+}$
<Procudure>::= **fun identifier** “(“ <arglist>“)” <retType> <Statments>
<retType>::= <type> | **fun** “(“ <typelist>“)” <retType>
<typelist>::= <type> | [<typelist>]
<type>::= **numeric | boolean |string**
<arglist>::= “(“ “) “ | “ (“ **identifier** <type> [, <arglist>] “) “

<Statments>::= {<stmt>}$^{+}$
<stmt>::= <varDeclarestmt>|<Printstmt>|<assigmentstmt> | <callstmt> | <ifstmt>
| < whilestmt > | <retstmt>
<varDeclarestmt>::= **var identifier** <type> “ ; “ | **var identifier** <type>=<BExpr> “;”
| **var identifier fun =**<callexpr> “;”
<Printstmt>::=**println** <BExpr>“;”
<assigmentstmt>::= **identifier** “=“ <BExpr> “;” | **identifier** “=“ <Async>
<ifstmt>::=**if** <BExpr> “{“<Statments> } [**else** “{“<Statments>“}”]
<whilestmt>::= **while “(“** <Bexpr> ”)” “{“ <Statments> “}”
<retstmt>=**return** <Bexpr> “;”
<rexpr>=<BExpr> | **fun** “(“ <arglist>“)” <retType> <Statments> “;”
<Async>=“**async**” **return** “{“ <callexpr>“}”

<BExpr> ::= <LExpr> <LOGIC_OP> <LExpr>
<LExpr> ::= <Expr> <REL_OP ><Expr>
<Expr> ::= <Term> <ADD_OP ><Expr>
<Term>::= <Factor> <MUL_OP> <Term>
<Factor> ::= **<numeric>** | **<string>** | **true | false** | **<identifier>** | <callexpr> | “(“ <expr> “)” | {+|-|!}
<callexpr> ::= **identifier** “(“ “) “ | **identifier** “(“ <callpar> “)” “;”
<callpar>::=<BExpr> | [',< callpar >']
<LOGIC_OP> := “&&” | “||”
<REL_OP> := “>“ |” < “|” >=“ |” <=“ |” <>“ |” ==“

<MUL_OP> := "*" |" /"
<ADD_OP> := "+" |" -"
<**identifier**>::= letter { letter | digit }
<**numeric**>::=digit

Lexer

The Laxer determines which tokens are ultimately sent to the parser and, throw out things that are not defined in the grammar, like comments. As for Funw@ap the Lexer cares about characters (A-Z and the usual symbols), numbers (0-9), characters that define operations (such as +, -, *, and /), quotation marks for string encapsulation, open and closed curly brace ({,}), comma, open and closed paternities, semicolons and keywords.

The lexer should read the source code character by character, determines which tokens are ultimately sent to the parser.

After each token, it should use the next character c to decide what kind of token to read.

- if c is a digit, collect an numeric as long as you read digits
- if c is a letter, collect an identifier or keyword as long as you read identifier characters (digit, letter, ")
- if c is a double quote, collect a string literal as long as you read characters other than a double quote
- if c is space character (i.e. space, newline, or tab), ignore it and read next character.

  The lexer here needs a one-character **lookahead** - it must read one more than the next character. For example during lexing, the lexer knows if it is reading a comment symbol and does not start reading a comment in the middle of this.

  Regular expression for the lexer
  - Tokens = Space (Token Space)*
  - Token = TInt | TId | TKey | TSpec
  - Numeric = Digit Digit*
  - Digit = "0" | "1" | "2" |"3" | "4" |"5" | "6" |"7" | "8" | "9"
  - TId = Letter IdChar*
  - Letter = "A" | ... | "Z" | "a" | ... | "z"
  - IdChar = Letter | Digit
  - TKey = "i" "f" | "e" "l" "s" "e" |"f""u""n"| "a"s"y"n"c"|"r"e"t"u"r"n"|"w"h"i"l"e", |"t"r"u"e"|"f"a"l"s"|"e"
  - Tart = "+""-" | "*" |"/" ...
  - TokRelational="<"|">"|"="|"<="|">="|"!="
  - TLogical="&&"|"||"| "!"
  - Space = (" " | "\n" | "\t")*
  - Comment= //

Parser

Before Interpretation or generating .Net IL code all input have to be parsed, the parser accept a token from a lexer (e.g. fun, if, return ,async ,identifier ,number etc) and parsed to form AST and *handles error output, if there is a failure* .The parser class derived from the laxer class, It is recursive descent parser which means top down parser built from a set of mutually recursive procedures, where for each production rule of the grammar there is one procedure implement.

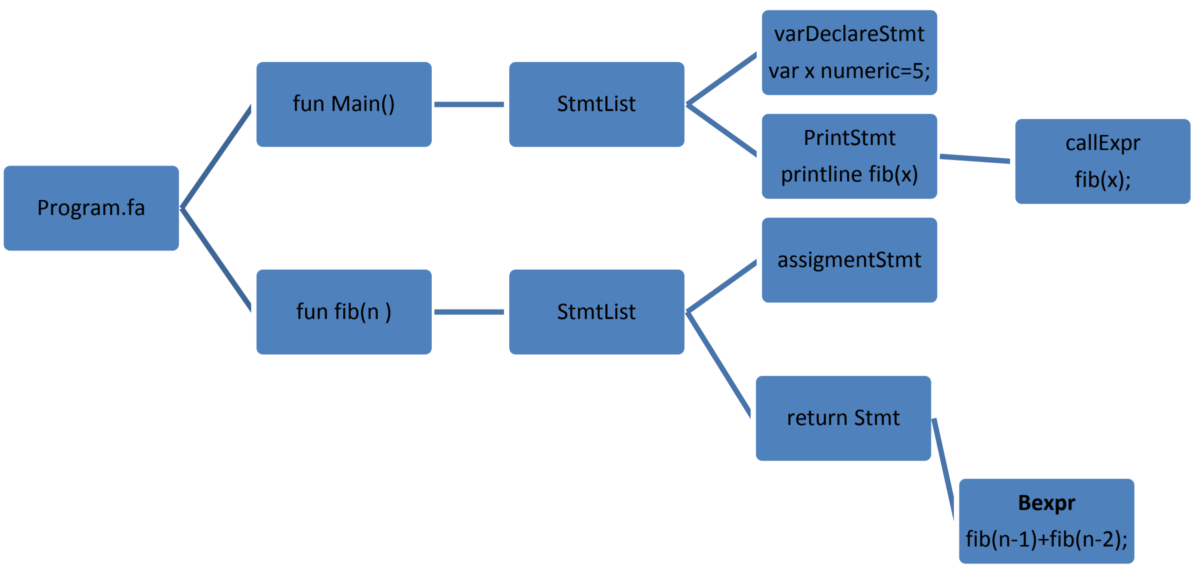


Fig 1 AST for the program.fa

Variables

Each variable have name, type and value .A variable store in the SYMBOL_INFO class .During variable declaration a new SYMBOL_INFO object is create and variable information (name, type and value) will add to the SymbolTable class in the current context(the context class of a Procedure) and a variable node is added to the AST.

Any variable encountered *during the Parse Process* will be put into the symbol table associated with *Compilation Context*

SYMBOL TABLE =- store in a Hashtable SYMBOL_INFO class (name value pair)

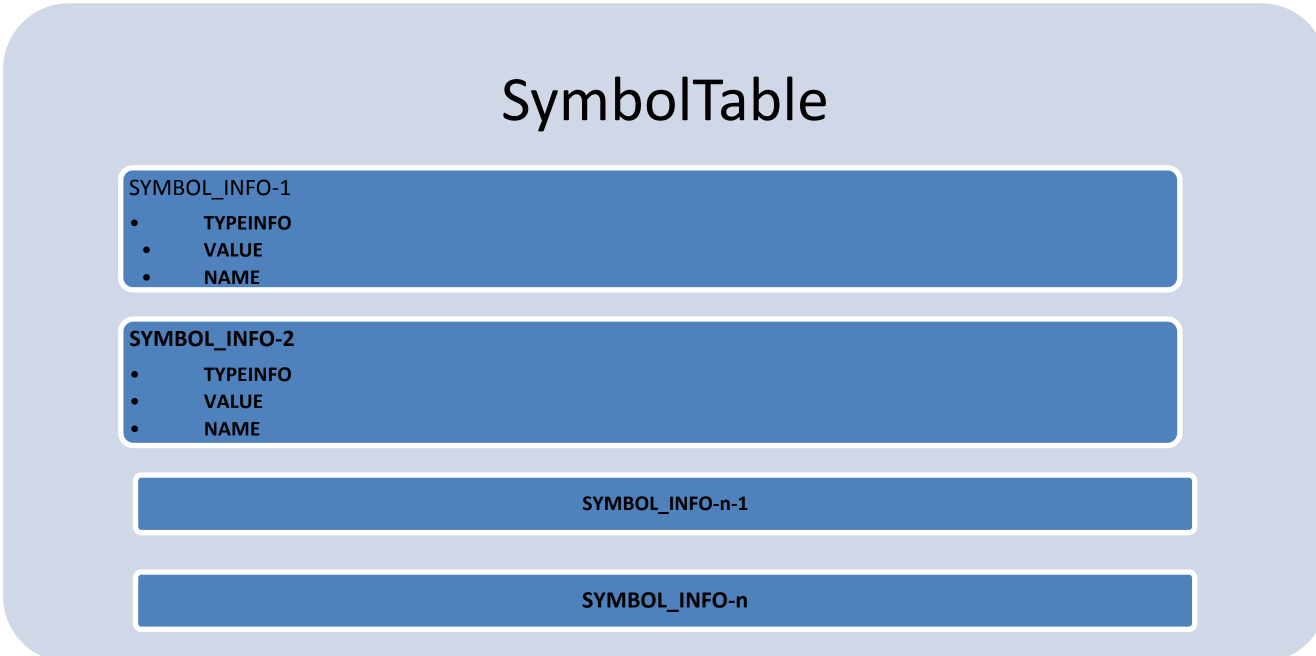


Fig2 diagram representation of symbol table

Context classes:-

- ✓ COMPILATION_CONTEXT :-hold SymbolTable class and use to store variables , type checking during parsing .
- ✓ RUNTIME_CONTEXT:-During interpretation and hold SymbolTable class and store variables, use to managing some issues like scope .
- ✓ DNET_EXECUTABLE_GENERATION_CONTEXT : during compilation(IL Code Generation) I will discuss in the implementation section.

From the grammar we learnt that a program (module) is a collection of procedures (functions) and a statement is a collection (list) of expression.

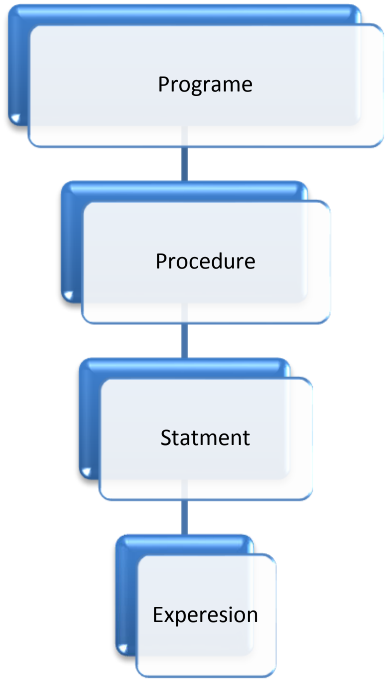


Fig 3 structure of the Program/module

Implementation

Language selection

There are plenty of programming language, before going to implementation I have to choice the one which is powerful, easy to learn and to write lexer, parser, interpreter and Code generation. then my inclination was toward C# .The reasons are String class in C# *have powerful methods so string manipulation is simple , Parallel programming via Asynchronous method , the get and set method allow me to write effective code with explicit property method, collection frame work class to manipulate objects as array, and to generate .Net IL code and others*.

In short ,there is no doubt about C# is the best language for compiler development in .net Framework .I can illustrate this by observing the most power full High-level language compiler (C# 5.0) is also developed by itself.

Expression and Statement Implementation

Programming languages are hierarchical in nature. We can model programming language constructs as classes. Trees are a natural data structure to represent most things hierarchical. Modeling Expression and statement .Once you have declared the abstract class, we can create a hierarchy of classes to model an expression

Expression: - evaluate for the value. The expression abstract classes have four methods;

- Evaluate: - these methods perform some operation and return the result during Interpretation.
- Type check: - use to check the data type of expression, return the expression/s data type if they are compatible otherwise return null.
- GetType:- return expression data type.
- Compile: - Generate IL opcodes.

All different expression classes inherit this class. Some Expression class and their methods Description.

- Binary add/minus: - class for manipulating binary add and subtract.
- BooleanConstant:- class for manipulating Boolean values
- StringLiteral= class for manipulating Boolean values .
- CallExper, class for calling a procedure accepts a procedure name, argumentList and call the Procedure.Excute Method.
- Unary add/minus:- class for manipulating binary add and subtract
- Logical: - class for manipulating logical operation (&&, ||,!) and only applicable for Boolean value
- Relational: - class for manipulating relational operations (==, <=,>=, <) Evaluate the Left Expression (Exp1) and Right Expression (Exp2) ,the Operand Type (must be of same type).

Statement: - execute for its effect, have 2 methods

- Execute :- Execute the statement and return value.
- Compile:- Generate IL opcodes.

Some Statement class and their methods Description

- If statement:- accept condition, true part (statementlist)and false part(statementlist)evaluate the condition and if the condition is true execute the true stetmentList ,else execute the false StamentList.
- WhileStatment:-accept condition and statmentList evaluate it and excute the statement until the condtion is false.

- ReturnStatment= accept an expression evaluate it and return null/SymbolInfo

Lexer and parser Implementation

Hard written parser and laxer vs. parser and Lexer generator Tools

We can implement both via tools (such as coco/R,yacc,ANTLR,fslex and fsyac,..) or writing code by hand. Before diving to the implementation let's discuss about both techniques.

- hand written lexer / parser : the programmer has a full control on the program (lookahead) ,since they are smaller program, but it require much work to include the position, manage the next ,previous token ,etc.
- Tool generated Lexer/parser: - the lexer /parser is generated by the tool they doesn't require much effort and time, but they are not too flexible as hand written lexer/parser, Generated code is hard to understand and debug.

  Hence, the grammar is not complex and one of the objectives of this aim of this Project is to excel programming skill I prefer to write lexer and parser by hand.

Tools use for Implementation

- Visual studio 2010 Ultimate Version
- .Net Framework 4.0
- ILDSAM

  ILDASM takes a portable executable (PE) file that contains intermediate language (IL) code and creates a text file suitable as input to Ilasm.exe.
  - ✓ This tool is automatically installed with Visual Studio.

Lexing

The lexer class initializes keywords, read the porgrame.fa and tokenizes each character via GetNext() method then returns a token to the parser.

Parsing

The parser class derived from the laxer class. Since it is recursive descent parser for each production rule of the grammar there is one procedure implement. The parser accept a token from the lexical analyzer class via GetNext() method.

The process of parsing start from parsing parseFunction() these function parse function name, return type and argument list , then Parse statements after that add to the ProcedureBuilder class(which is used to build a procedure) to build a Procedure then its add to the TModuleBuilder(which is used to build a Module or programe.fa) .The parsing is repeatedly doing until all function parsed .finally Tmodule(Programe) or in another word AST is constructed. In between there is , data type and scope checking (inside a block) .e.g. before assignment values to a variable check if the variable is exits in the symbol table and data type.

Some fundamental Classes

TModule (program) is a collection of Procedures and interpretation / translation to .net IL start from this class by executing the main method.

Since we create Procedure and `TModule` after parsing process, we need to accumulate the requisite Objects before we create Procedure *building pattern* is an elegant way to do such thing.

Some basic class to build a program

- `TModuleBuilder`:-use to build a module which is a collection of procedures
- `ProcedureBuilder`:-use to build a procedure
- `Procedure`: - Function which has name, return type, argument/s and statement list

Interpretation

Now after successfully parsing the Program Interpreting start .It's a recursive descent interpretation (top down). Interpreter start traverse the AST from the main Procedure (fun main () numeric) and execute each statement, each statement evaluate expressions. From the TModule class (which contain list of Procedures) the method Find("main") is responsible to find the main function/entry point of the program; since the program execution start from main() function, other functions must be called (from main or other method) .

To manage scope the interpreter use `RUNTIME_CONTEXT class` which has a *symbol table class variable* . When a function called a new instance of `RUNTIME_CONTEXT created and` its parameter will added to the symbol table of the Program. During Interpreting there is also , data type and scope checking (inside a function/closure function)

Generating .Net IL code and .Net Executable

Now I am going to generate IL code for funw@ap, DotNet IL use stack based machine code and CLR (responsible for executing machine code) has got the evaluation stack for processing expressions, statement and Procedures.

CLR takes care of a number of low-level executions such as application hosting, thread handling, memory management, security checks and application performance. Its primary role is to locate, load, and manage the .NET types (class, array, object etc.). The beauty of CLR is that all .NET-supported languages can be executed under this single defined runtime layer.

To generate IL code import System.reflection namespace and System.reflection.Emit which are used to retrieve information about assemblies, modules, members, parameters, and other to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. In this namespace there is some hierarchy in order to generate .exe the hierarchies are (the written a program based on this hierarchy)

- AssemblyBuilder
    - ModuleBuilder
        - TypeBuilder
            - MethodBuilder

Some Important class for IL Code Generation and to create PE

`ExeGenerator`- Take the program/module and generate IL Code and write into exe file
`DNET_EXECUTABLE_GENERATION_CONTEXT`:- use for compile each Procedure Which has a gadget for compilation and pass as argument for each compile methods.

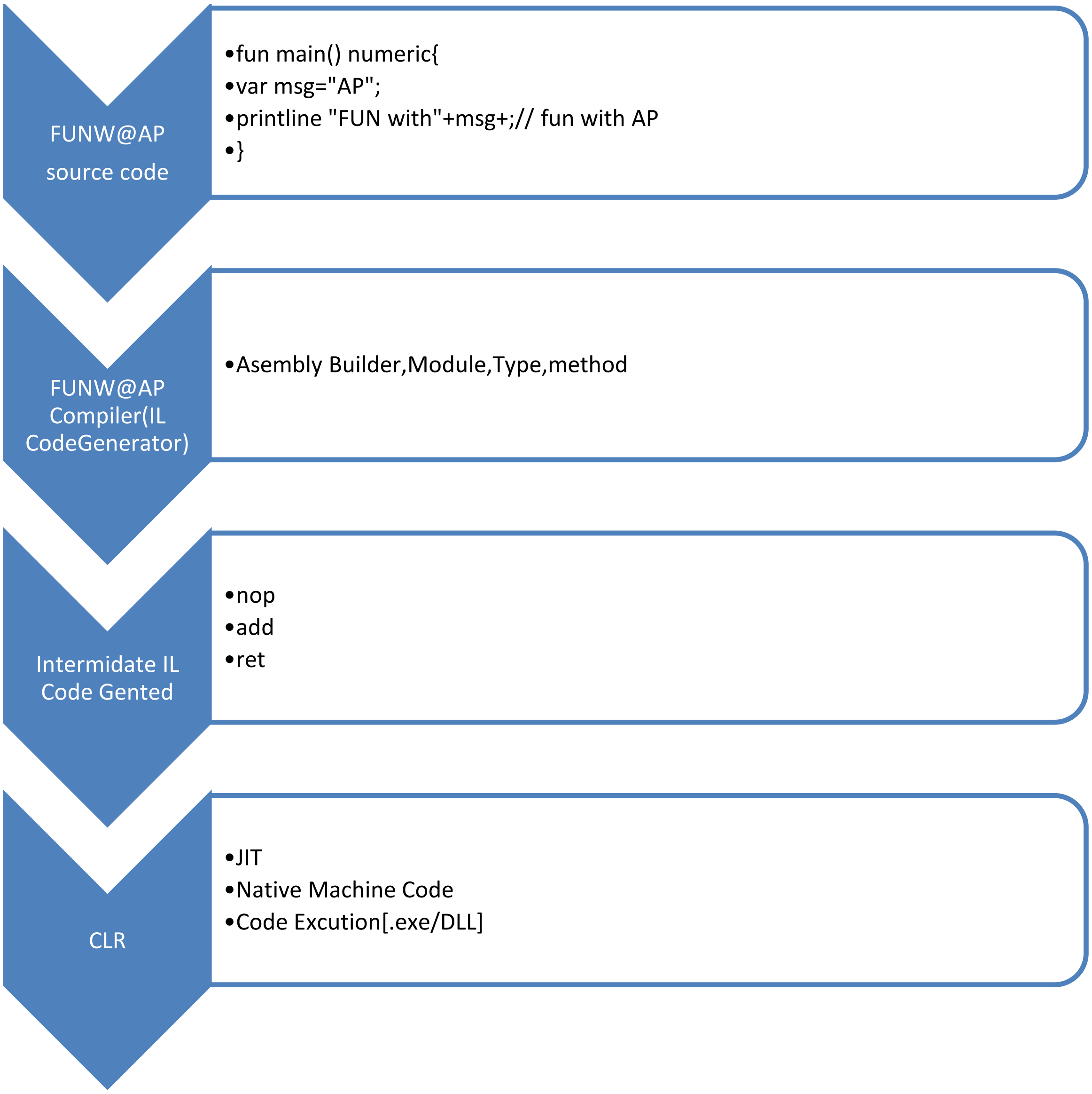
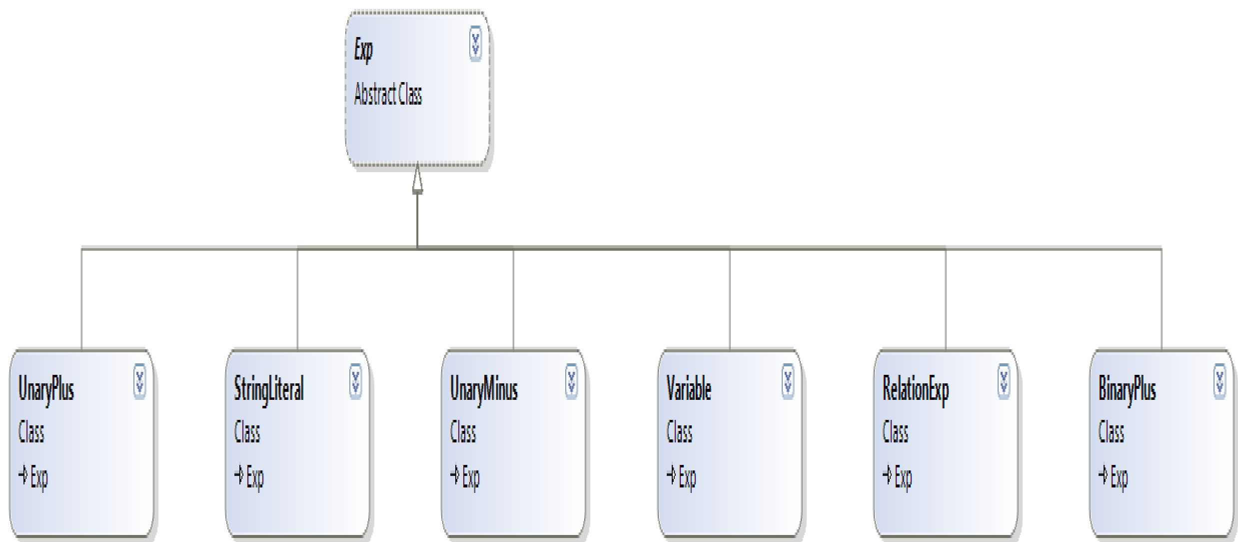
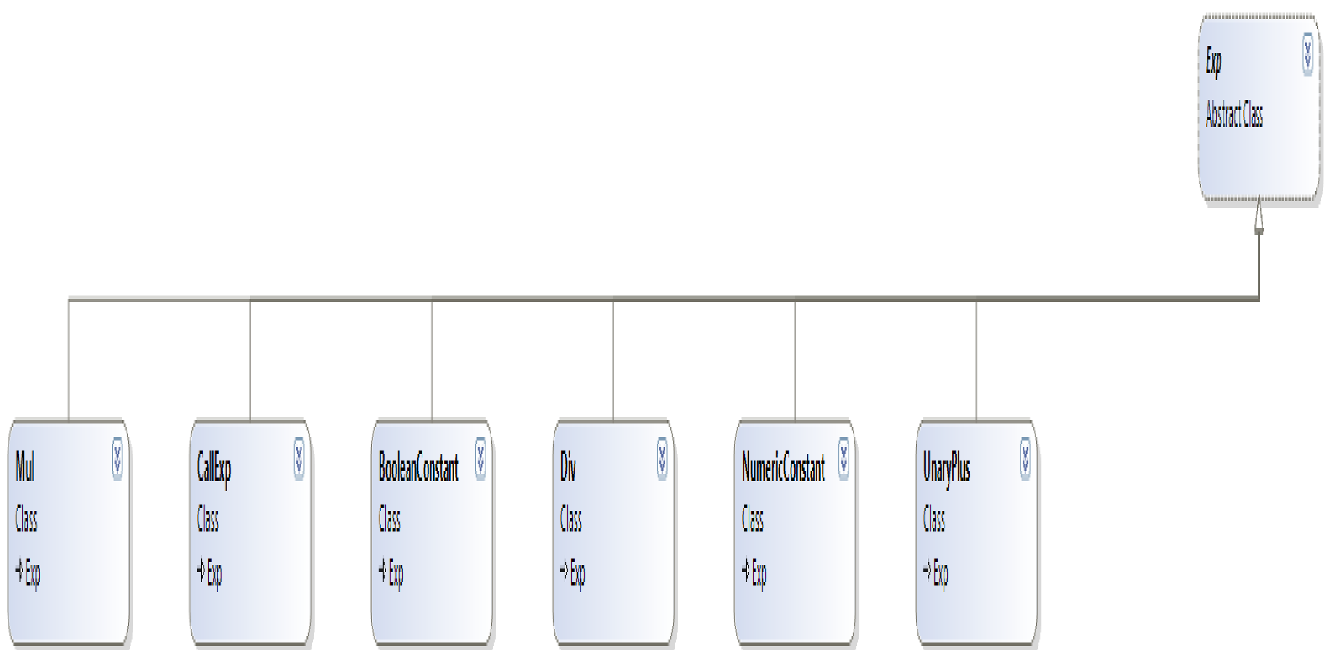


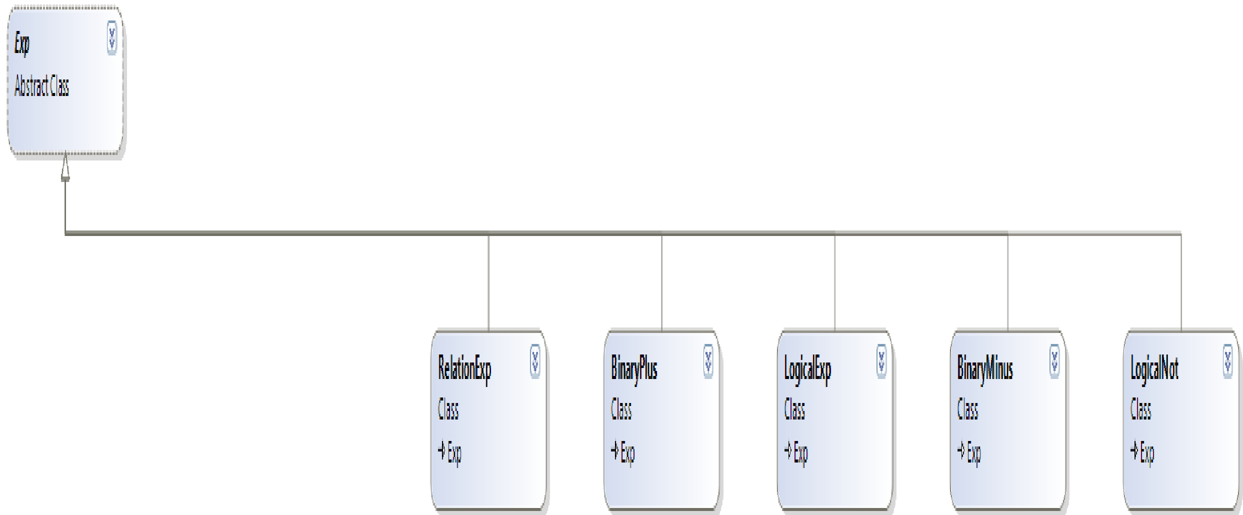
Figure 4 IL Code generation/ PE creation process

Thank you for your Attention!

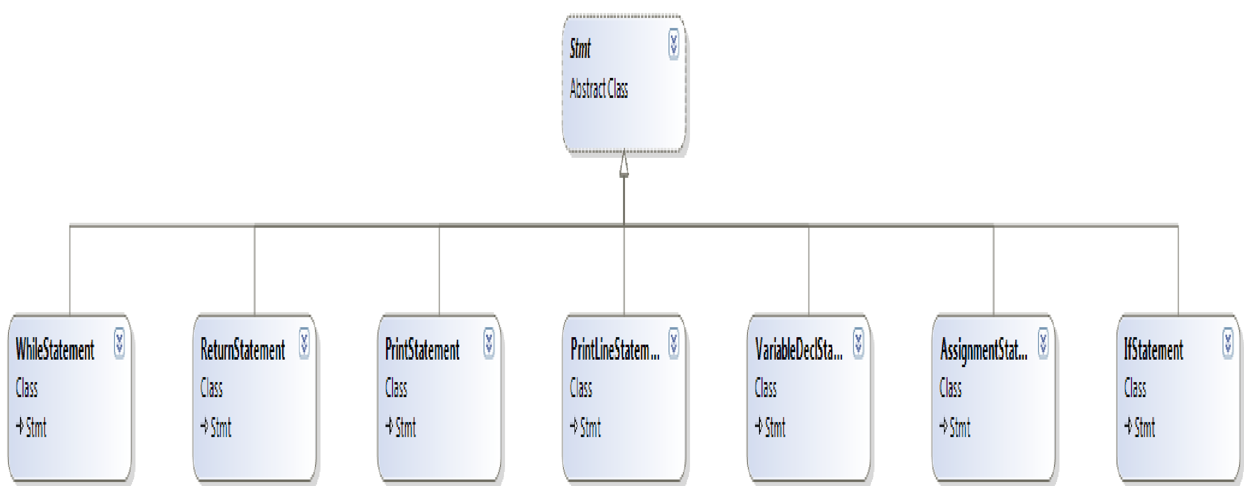
Appendex 1 Class diagram of Expression







Appendix 2 Class Diagram of Statement class



Appendix 3 sample program and IL Code generation output

```
//basic.fa

FUN  MAIN()BOOLEAN

{

        var a numeric;

        a=0;

        var b numeric;

        b=0;

        //a_eq_b  is TRUE


        var a_eq_b boolean;

        a_eq_b=a==b;

        printline a_eq_b;

        var i numeric;

        i=0;

        while ( i <=32 )

        {

                        var a_eq_zero boolean;

                          a_eq_zero=a==0;

                            if(a_eq_zero)

                                    {

                                    printline "a is zero so b is";

                                    }

                                    else

                                    { printline "at least one of a b is zero" ;
```

```
}
i = i + 1;
a=a+i;
}
}
```

C:\ FUNWAPCOMPILE\>FUNWAPCOMPILE.exe basic.fa

Output.txt file contains the following

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 3.5.30729.1
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 4:0:0:0
}
.assembly MyAssembly
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
```

```
}
.module DynamicModule1
// MVID: {54276915-023D-4895-90F3-8EFD8EAE30DE}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00210000


// =============== CLASS MEMBERS DECLARATION ===================
```

```
.class private auto ansi MainClass
       extends [mscorlib]System.Object
{
  .method public static void  MAIN() cil managed
  {
    .entrypoint
    // Code size       147 (0x93)
    .maxstack  4
    .locals init (float64 V_0,
             float64 V_1,
             bool V_2,
             float64 V_3,
             bool V_4)
    IL_0000:  ldc.r8     0.0
    IL_0009:  stloc.0
    IL_000a:  ldc.r8     0.0
    IL_0013:  stloc.1
    IL_0014:  ldloc.0
    IL_0015:  ldloc.1
    IL_0016:  ceq
    IL_0018:  stloc.2
    IL_0019:  ldloc.2
```

```
IL_001a:  call       void [mscorlib]System.Console::WriteLine(bool)
IL_001f:  ldc.r8     0.0
IL_0028:  stloc.3
IL_0029:  ldloc.3
IL_002a:  ldc.r8     32.
IL_0033:  cgt
IL_0035:  ldc.i4     0x0
IL_003a:  ceq
IL_003c:  ldc.i4     0x1
IL_0041:  ceq
IL_0043:  brfalse    IL_0092

IL_0048:  ldloc.0
IL_0049:  ldc.r8     0.0
IL_0052:  ceq
IL_0054:  stloc.s    V_4
IL_0056:  ldloc.s    V_4
IL_0058:  ldc.i4     0x1
IL_005d:  ceq
IL_005f:  brfalse    IL_0073

IL_0064:  ldstr      "a is zero so b is"
IL_0069:  call       void [mscorlib]System.Console::WriteLine(string)
IL_006e:  br         IL_007d
```

```
  IL_0073:  ldstr      "at least one of a b is zero"
  IL_0078:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_007d:  ldloc.3
  IL_007e:  ldc.r8     1.
  IL_0087:  add
  IL_0088:  stloc.3
  IL_0089:  ldloc.0
  IL_008a:  ldloc.3
  IL_008b:  add
  IL_008c:  stloc.0
  IL_008d:  br         IL_0029

  IL_0092:  ret
} // end of method MainClass::MAIN

.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method MainClass::.ctor
```

```
} // end of class MainClass



// ==============================================================


// *********** DISASSEMBLY COMPLETE ***********************
```