SLANG4.NET

htttp://slangfordotnet.codeplex.com

The Art of Compiler Construction using C#

By

Praseed Pai K.T. (http://praseedp.blogspot.com)
praseedp@yahoo.com

CHAPTER 1

Version 0.1

Table of Contents

The Art of Compiler Construction using C#

Thanks to the availability of information and better tools writing a compiler has become just an excersise in software engineering. The Compilers are not difficult programs to write. The various phases of compilers are easy to understand in an independent manner. The relationship is not purely sequential. It takes some time to put phases in perspective in the job of compilation of programs.

The task of writing a compiler can be viewed in a top down fashion as follows

Parsing => Creation of Abstract Syntax Tree => Tree Traversal to generate the Object code or Recursive interpretation.

Abstract Syntax Tree

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract (simplified) syntactic structure of source code written in a certain programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses is implicit in the tree structure, and a syntactic construct such as if cond then expr may be denoted by a single node with two branches.Most of you might not be aware of the fact that , 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.

As a case in the point , let us look a simple expression evaluator . The expression evaluator will support double precision floating point value as the operands. The Operators supported are addition (+) , subtraction (-), multiplication (*) and division. The Object model support Unary operators (+ , -) as well. We are planning to use a composition model for modeling an expression.

In most imperative programming languages , an expression is something which u evaluate for it's value. Where as statements are something which you executes for it's effect.

let us define an abstract class for Exp

```
/// <summary>
///     Abstract for Expression evaluation
/// </summary>
abstract class Exp
 {
        public abstract double Evaluate(RUNTIME_CONTEXT cont);
 }
```

For the time being RUNTIME_CONTEXT is an empty class

```
/// <summary>
  ///      One can store the stack frame inside this class
  /// </summary>
  public class RUNTIME_CONTEXT
  {
     public RUNTIME_CONTEXT()
     {

     }
  }
```

Modeling Expression

Once u have declared the interface and it's parameters , we can create a hierarchy of classes to model an expression.

```
class Exp                       // Base class for Expression
   class  NumericConstant       //  Numeric Value
   class  BinaryExp             //  Binary Expression
   class  UnaryExp              //  Unary Expression
```

Take a look at the listing of NumericConstant class

```
/// <summary>
  ///      one can store number inside the class
  /// </summary>
 public class NumericConstant : Exp
  {
     private double _value;
     /// <summary>
     ///     Construction does not do much , just keeps the
     ///     value assigned to the private variable
     /// </summary>
     /// <param name="value"></param>

     public NumericConstant(double value)
     {
        _value = value;
     }
     /// <summary>
     ///     While evaluating a numeric constant , return the _value
     /// </summary>
     /// <param name="cont"></param>
     /// <returns></returns>
```

```
    public override double Evaluate(RUNTIME_CONTEXT cont)
    {
        return _value;
    }

}
```

Since the class is derived from Exp , it ought to implement the Evaluate method. In the Numeric Constant node , we will store a IEEE 754 double precision value. While evaluating the tree , the node will return the value stored inside the object.

Binary Expression

In a Binary Expression , one will have two Operands (Which are themselves expressions of arbitary complexity) and an Operator.

```
/// <summary>
/// This class supports Binary Operators like + , - , / , *
/// </summary>
public class BinaryExp : Exp
{
    private Exp _ex1, _ex2;
    private OPERATOR _op;
    /// <summary>
    ///
    /// </summary>
    /// <param name="a"></param>
    /// <param name="b"></param>
    /// <param name="op"></param>

    public BinaryExp(Exp a, Exp b, OPERATOR op)
    {
        _ex1 = a;
        _ex2 = b;
        _op = op;
    }

    /// <summary>
    /// While evaluating apply the operator after evaluating the left and right operands
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>
    public override double Evaluate(RUNTIME_CONTEXT cont)
    {


        switch (_op)
        {
            case OPERATOR.PLUS:
                return _ex1.Evaluate(cont) + _ex2.Evaluate(cont);
            case OPERATOR.MINUS:
                return _ex1.Evaluate(cont) - _ex2.Evaluate(cont);
```

```
            case OPERATOR.DIV:
                return _ex1.Evaluate(cont) / _ex2.Evaluate(cont);
            case OPERATOR.MUL:
                return _ex1.Evaluate(cont) * _ex2.Evaluate(cont);

        }

        return Double.NaN;

    }

}
```

Unary Expression

In an unary expression , one will have an Operand (which can be an expression of arbitary complexity) and an Operator which can be applied on the Operand.

```
/// <summary>
 ///     This class supports Unary Operators like + , - , / , *
 /// </summary>
 public class UnaryExp : Exp
 {
    private Exp _ex1;
    private OPERATOR _op;
    /// <summary>
    ///
    /// </summary>
    /// <param name="a"></param>
    /// <param name="b"></param>
    /// <param name="op"></param>

    public UnaryExp(Exp a, OPERATOR op)
    {
        _ex1 = a;
        _op = op;
    }

    /// <summary>
    ///     While evaluating apply the unary operator after evaluating the operand.
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>
    public override double Evaluate(RUNTIME_CONTEXT cont)
    {

        switch (_op)
        {
```

```
            case OPERATOR.PLUS:
                return _ex1.Evaluate(cont);
            case OPERATOR.MINUS:
                return -_ex1.Evaluate(cont);
        }

        return Double.NaN;

    }

}
```

In the CallSLANG project, we will include the SLANG_DOT_NET assembly before composing the expression.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SLANG_DOT_NET; // include SLANG_DOT_NET assembly

namespace CallSLANG
{
    class Program
    {
        static void Main(string[] args)
        {
            // Abstract Syntax Tree (AST) for 5*10
            Exp e = new BinaryExp(new NumericConstant(5),
                                  new NumericConstant(10),
                                  OPERATOR.MUL);

            //
            // Evaluate the Expression
            //
            //
            Console.WriteLine(e.Evaluate(null));

            // AST for  (10 + (30 + 50 ) )

            e = new UnaryExp(
                        new BinaryExp(new NumericConstant(10),
                            new BinaryExp(new NumericConstant(30),
                                          new NumericConstant(50),
                                    OPERATOR.PLUS),
                        OPERATOR.PLUS),
```

```
                OPERATOR.MINUS);

            //
            // Evaluate the Expression
            //
            Console.WriteLine(e.Evaluate(null));

            //
            // Pause for a key stroke
            //
            Console.Read();

        }
    }
}
```

SLANG4.NET

htttp://slangfordotnet.codeplex.com

The Art of Compiler Construction using C#

By

Praseed Pai K.T. (http://praseedp.blogspot.com)
praseedp@yahoo.com

CHAPTER 2

Version 0.1

INPUT Analysis

Compilers are programs which translate source language to a target language. The Source language can be a language like C,C++ or Lisp. The potential target languages are assembly languages , object code for the microprocessors like intel x86, itanium or power pc. There are programs which translate java to C++ and Lisp to C. In such case , target language is another programming language.

Any compiler has to understand the input. Once it has analyzed the input characters , it should convert the input into a form which is suitable for further processing.

Any input has to be parsed before the object code translation. To Parse means to understand.
The Parsing process works as follows

The characters are grouped together to find a token (or a word). Some examples of the tokens are '+','*',while , for , if etc. The module which reads character at a time and looks for legal token is called a lexical analyzer or Lexer. The input from the Lexer is passed into a module which identifies whether a group of tokens form a valid expression or a statement in the program. The module which determines the validity of expressions is called a parser. Rather than doing a lexical scan for the entire input , the parser requests the next token from the lexical analyzer. They act as if they are co-routines.

To put everything together let us write a small program which acts a four function calculator
The calculator is capable of evaluating mathematical expressions which contains four basic arithmetical operators , paranthesis to group the expression and unary operators.
Given below is the Lexical Specifications of the calculator.

TOK_PLUS – '+'
TOK_MUL - '*'
TOK_SUB - '-'
TOK_DIV - '/'
TOK_OPAREN – '('
TOK_CPAREN – ')'
TOK_DOUBLE – [0-9]+

The stuff can be converted into C# as follows

```
/// <summary>
///   Enumeration for Tokens
/// </summary>
 public enum TOKEN  {
      ILLEGAL_TOKEN = -1, // Not a Token
      TOK_PLUS = 1, // '+'
      TOK_MUL, // '*'
      TOK_DIV, // '/'
      TOK_SUB, // '-'
      TOK_OPAREN, // '('
      TOK_CPAREN, // ')'
      TOK_DOUBLE, // '('
      TOK_NULL // End of string
   }
```

The Lexical Analysis Algorithm scans through the input and returns the token associated with the operator. If it has found out a number , returns the token associated with the number. There should be another mechanism to retrieve the actual number identified.

Following pseudo code shows the schema of the lexical analyzer

```
while ( there is input ) {
    switch(currentchar) {
    case Operands:
        advance input pointer
        return TOK_XXXX;
   case Number:
        Extract the number( Advance the input )
        return TOK_DOUBLE;
   default:
       error
   }
}
```

The following C# code is a literal translation of the above algorithm.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SLANG_DOT_NET
{
   /// <summary>
   ///   Enumeration for Tokens
   /// </summary>
   public enum TOKEN
   {
      ILLEGAL_TOKEN = -1, // Not a Token
      TOK_PLUS = 1, // '+'
      TOK_MUL, // '*'
      TOK_DIV, // '/'
      TOK_SUB, // '-'
      TOK_OPAREN, // '('
      TOK_CPAREN, // ')'
      TOK_DOUBLE, // '('
      TOK_NULL // End of string
   }

   ////////////////////////////////////////////////////////
   //
   // A naive Lexical analyzer which looks for operators , Parenthesis
   // and number. All numbers are treated as IEEE doubles. Only numbers
   // without decimals can be entered. Feel free to modify the code
   // to accomodate LONG and Double values
```

```
public class Lexer
{
   String IExpr; // Expression string
   int index; // index into a character
   int length; // Length of the string
   double number; // Last grabbed number from the stream
   ////////////////////////////////////////////
   //
   // Ctor
   //
   //
   public Lexer(String Expr)
   {
      IExpr = Expr;
      length = IExpr.Length;
      index = 0;
   }
   ////////////////////////////////////////////////////
   // Grab the next token from the stream
   //
   //
   //
   //
   public TOKEN GetToken()
   {
      TOKEN tok = TOKEN.ILLEGAL_TOKEN;
      ////////////////////////////////////////////////////////
      //
      // Skip the white space
      //
      while (index < length &&
      (IExpr[index] == ' ' || IExpr[index] == '\t'))
         index++;
      ////////////////////////////////////////////
      //
      // End of string ? return NULL;
      //
      if (index == length)
         return TOKEN.TOK_NULL;
      ////////////////////////////////////////////////
      //
      //
      //
      switch (IExpr[index])
      {
         case '+':
            tok = TOKEN.TOK_PLUS;
            index++;
            break;
         case '-':
            tok = TOKEN.TOK_SUB;
            index++;
            break;
         case '/':
            tok = TOKEN.TOK_DIV;
            index++;
            break;
         case '*':
```

```
                tok = TOKEN.TOK_MUL;
                index++;
                break;
            case '(':
                tok = TOKEN.TOK_OPAREN;
                index++;
                break;
            case ')':
                tok = TOKEN.TOK_CPAREN;
                index++;
                break;
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                {
                    String str = "";
                    while (index < length &&
                    (IExpr[index] == '0' ||
                    IExpr[index] == '1' ||
                    IExpr[index] == '2' ||
                    IExpr[index] == '3' ||
                    IExpr[index] == '4' ||
                    IExpr[index] == '5' ||
                    IExpr[index] == '6' ||
                    IExpr[index] == '7' ||
                    IExpr[index] == '8' ||
                    IExpr[index] == '9'))
                    {
                        str += Convert.ToString(IExpr[index]);
                        index++;
                    }
                    number = Convert.ToDouble(str);
                    tok = TOKEN.TOK_DOUBLE;
                }
                break;
            default:
                Console.WriteLine("Error While Analyzing Tokens");
                throw new Exception();
        }
        return tok;
    }
    public double GetNumber() { return number; }
  }
}
```

The Grammar

In computer science, a formal grammar (or grammar) is a set of formation rules (grammar) that describe which strings formed from the alphabet of a formal language are syntactically valid within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics (i.e. what the strings mean).

A context-free grammar is a grammar in which the left-hand side of each production rule consists of only a single nonterminal symbol. This restriction is non-trivial; not all languages can be generated by context-free grammars. Those that can are called context-free languages.

The Backus Naur Form (BNF) notation is used to specify grammars for programming languages , commnd line tools , file formats to name a few . The semantics of BNF is beyond the scope of this book.

Grammar of the expression evaluator

```
<Expr>  ::=  <Term> | Term { + | - } <Expr>
<Term> ::=  <Factor> | <Factor>  {*|/} <Term>
<Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
```

There are two types of tokens in any grammar specifications. They are terminal tokens (terminals) or non terminals . In the above grammar , operators and <number> are the terminals.
<Expr>,<Term>,<Factor> are non terminals. Non terminals will have at least one entry on the left side.

Conversion of Expression to the psuedo code

```
//   <Expr> ::=   <Term>  { + | - }  <Expr>
Void Expr() {
        Term();

        if ( Token == TOK_PLUS ||  Token == TOK_SUB    )
        {
              // Emit instructions
              //  and perform semantic operations

              Expr();    // recurse
        }
}
```

Converstion of term to the psuedo code

```
//   <Term> ::=   <Factor>  { * | / }  <Term>
Void Term() {
        Factor();
```

```
        if ( Token == TOK_MUL ||  Token == TOK_DIV    )
        {
              // Emit instructions
             //  and perform semantic operations

             Term();    // recurse
        }
}
```

The following psuedo code demonstrates how to map <Factor> into code

```
//   <Factor> ::=  <TOK_DOUBLE> | ( <expr> ) |  { + |- }  <Factor>
//
Void Factor() {
        switch(Token)
          case TOK_DOUBLE:
          //   push token to IL operand stack return
         case TOK_OPAREN:
               Expr();    //recurse
               // check for closing parenthesis and return
          case UNARYOP:
               Factor();   //recurse
          default:
               //Error
}
```

The class RDParser is derived from the Lexer class. By using an algorithm by the name Recursive descent parsing , we will evaluate the expression.A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SLANG_DOT_NET
{
   /// <summary>
   ///
   /// </summary>
   public class RDParser : Lexer
```

```
{
    TOKEN Current_Token;


    public RDParser(String str)
        : base(str)
    {


    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public Exp CallExpr()
    {
        Current_Token = GetToken();
        return Expr();
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public Exp Expr()
    {
        TOKEN l_token;
        Exp RetValue = Term();
        while (Current_Token == TOKEN.TOK_PLUS || Current_Token == TOKEN.TOK_SUB)
        {
            l_token = Current_Token;
            Current_Token = GetToken();
            Exp e1 = Expr();
            RetValue = new BinaryExp(RetValue, e1,
                l_token == TOKEN.TOK_PLUS ? OPERATOR.PLUS : OPERATOR.MINUS);

        }

        return RetValue;

    }
    /// <summary>
    ///
    /// </summary>
    public Exp Term()
    {
        TOKEN l_token;
        Exp RetValue = Factor();

        while (Current_Token == TOKEN.TOK_MUL || Current_Token == TOKEN.TOK_DIV)
        {
            l_token = Current_Token;
            Current_Token = GetToken();


            Exp e1 = Term();
            RetValue = new BinaryExp(RetValue, e1,
```

```
                l_token == TOKEN.TOK_MUL ? OPERATOR.MUL : OPERATOR.DIV);

            }

            return RetValue;
        }

        /// <summary>
        ///
        /// </summary>
        public Exp Factor()
        {
            TOKEN l_token;
            Exp RetValue = null;
            if (Current_Token == TOKEN.TOK_DOUBLE)
            {

                RetValue = new NumericConstant(GetNumber());
                Current_Token = GetToken();

            }
            else if (Current_Token == TOKEN.TOK_OPAREN)
            {

                Current_Token = GetToken();

                RetValue = Expr();  // Recurse

                if (Current_Token != TOKEN.TOK_CPAREN)
                {
                    Console.WriteLine("Missing Closing Parenthesis\n");
                    throw new Exception();

                }
                Current_Token = GetToken();
            }

            else if (Current_Token == TOKEN.TOK_PLUS || Current_Token == TOKEN.TOK_SUB)
            {
                l_token = Current_Token;
                Current_Token = GetToken();
                RetValue = Factor();

                RetValue = new UnaryExp(RetValue,
                    l_token == TOKEN.TOK_PLUS ? OPERATOR.PLUS : OPERATOR.MINUS);
            }
            else
            {

                Console.WriteLine("Illegal Token");
                throw new Exception();
            }

            return RetValue;

        }
    }
}
```

Using the Builder Pattern , we will encapsulate the Parser , Lexer class activities

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SLANG_DOT_NET
{
    /// <summary>
    ///    Base class for all the Builders
    /// </summary>
    public class AbstractBuilder
    {

    }
    /// <summary>
    ///
    /// </summary>
    public class ExpressionBuilder : AbstractBuilder
    {
        public string _expr_string;
        public ExpressionBuilder(string expr)
        {
            _expr_string = expr;


        }
        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public Exp GetExpression()
        {
            try
            {
                RDParser p = new RDParser(_expr_string);
                return p.CallExpr();
            }
            catch (Exception)
            {

                return null;
            }
        }



    }

}
```

In the CallSLang Project , the expression compiler is invoked as follows

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SLANG_DOT_NET;

namespace CallSLANG
{
   class Program
   {
      static void Main(string[] args)
      {
         ExpressionBuilder b = new ExpressionBuilder("-2*(3+3)");
         Exp e = b.GetExpression();
         Console.WriteLine(e.Evaluate(null));

         Console.Read();
      }
   }
}
```

SLANG4.NET

htttp://slangfordotnet.codeplex.com

The Art of Compiler Construction using C#

By

Praseed Pai K.T. (http://praseedp.blogspot.com)
praseedp@yahoo.com

CHAPTER 3

Version 0.1

STATEMENTS

The crux of the SLANG4.net can be summed up in two sentences

Expression is what you evaluate for it's value

Statement is what you execute for it's effect (on variables)

The above two maxims can be converted into a computational structure as follows

A) Expression is what you evaluate for it's value

```
/// <summary>
///     Abstract for Expression evaluation
///     Expression is what you evaluate for it's value
/// </summary>
public abstract class Exp
{
    public abstract double Evaluate(RUNTIME_CONTEXT cont);
}
```

B) Statement is what you execute for it's effect (on variables or lack of it)

```
/// <summary>
///   Statement is what you Execute for it's Effect
/// </summary>
public abstract class Stmt
{
    public abstract bool Execute(RUNTIME_CONTEXT con);
}
```

Let us implement a Print statement for the SLANG4.net compiler. The basic idea is as follows
you add a class to model a statement and since the class has to inherit from the Stmt (abstract class) ,
it ought to implement Execute Method.

```
/// <summary>
/// Implementation of Print Statement
/// </summary>

public class PrintStatement : Stmt
{
    /// <summary>
    ///   At this point of time , Print will
    ///   spit the value of an Expression on the screen.
    /// </summary>
    private Exp _ex;
    /// <summary>
```

```
        ///    Ctor just stores the expression passed as parameter
        /// </summary>
        /// <param name="ex"></param>
        public PrintStatement(Exp ex)
        {
            _ex = ex;
        }

        /// <summary>
        ///    Execute method Evaluates the expression and
        ///    spits the value to the console using
        ///    Console.Write statement.
        /// </summary>
        /// <param name="con"></param>
        /// <returns></returns>
        public override bool Execute(RUNTIME_CONTEXT con)
        {
            double a = _ex.Evaluate(con);
            Console.Write(a.ToString());
            return true;
        }
    }
```

Let us add a PrintLine statement as well. PrintLine implementation is not different from Print statement. The only difference is it emits a new line after the expression value.

```
/// <summary>
    ///  Implementation of  PrintLine Statement
    /// </summary>
    public class PrintLineStatement : Stmt
    {
        private Exp _ex;

        public PrintLineStatement(Exp ex)
        {
            _ex = ex;
        }
        /// <summary>
        ///    Here we are calling Console.WriteLine to emit
        ///    an additional new line .
        /// </summary>
        /// <param name="con"></param>
        /// <returns></returns>
        public override bool Execute(RUNTIME_CONTEXT con)
        {
            double a = _ex.Evaluate(con);
            Console.WriteLine(a.ToString());
            return true;
        }
    }
```

Once we have created to classes to implement Print and PrintLine statement , we need to modify our parser (frontend) to support the statement in the language.

We are going to add few more tokens to support the Statements in the SLANG.

```
public enum TOKEN
   {
      ILLEGAL_TOKEN = -1, // Not a Token
      TOK_PLUS = 1, // '+'
      TOK_MUL, // '*'
      TOK_DIV, // '/'
      TOK_SUB, // '-'
      TOK_OPAREN, // '('
      TOK_CPAREN, // ')'
      TOK_DOUBLE, // 'number'
      TOK_NULL, // End of string
      TOK_PRINT, // Print Statement
      TOK_PRINTLN, // PrintLine
      TOK_UNQUOTED_STRING,
      TOK_SEMI // ;
   }
```

In the Lexer.cs module , we add a new data structure to be used for Keyword lookup.

```
/// <summary>
   ///     Keyword Table Entry
   /// </summary>
   ///
   public struct ValueTable
   {
      public TOKEN tok;         // Token id
      public String Value;      // Token string
      public ValueTable(TOKEN tok, String Value)
      {
         this.tok = tok;
         this.Value = Value;

      }
   }
```

In the ctor of Lexer.cs , we will populate an array of ValueTables with Token and it's textual representation as given below.

```
         _val = new ValueTable[2];

         _val[0] = new ValueTable(TOKEN.TOK_PRINT, "PRINT");

         _val[1] = new ValueTable(TOKEN.TOK_PRINTLN, "PRINTLINE");
```

We need to add a new entrypoint into the RDParser.cs class to support statements. The grammar for the SLANG at this point of time (to support statement) is as given below.

```
<stmtlist> :=  { statement }+

{statement}  :=  <printstmt> | <printlinestmt>

<printstmt> :=  print   <expr >;

<printlinestmt>:= printline <expr>;

<Expr>  ::=  <Term> | Term { + | - } <Expr>
<Term> ::=  <Factor> | <Factor>  {*|/} <Term>
<Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
```

The new entry point to the parser module is as follows...

```
/// <summary>
///   The new Parser entry point
/// </summary>
/// <returns></returns>
public ArrayList Parse()
{
   GetNext();  // Get the Next Token
   //
   // Parse all the statements
   //
   return StatementList();
}
```

The StatementList method implements the grammar given above. The BNF to source code translation is very easy and without much explanation it is given below

```
/// <summary>
///  The Grammar is
///
///  <stmtlist> :=  { <statement> }+
///
///  {<statement> :=  <printstmt> | <printlinestmt>
///  <printstmt> :=  print   <expr >;
///
/// <printlinestmt>:= printline <expr>;
///
/// <Expr>  ::=  <Term> | <Term> { + | - } <Expr>
/// <Term> ::=  <Factor> | <Factor>  {*|/} <Term>
/// <Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
///
///
/// </summary>
```

```
    /// <returns></returns>
    private ArrayList StatementList()
    {
      ArrayList arr = new ArrayList();
      while (Current_Token != TOKEN.TOK_NULL)
      {
        Stmt temp = Statement();
        if (temp != null)
        {
          arr.Add(temp);
        }
      }
      return arr;
    }
```

The method Statement queries the statement type and parses the rest of the statement.

```
/// <summary>
    ///   This Routine Queries Statement Type
    ///   to take the appropriate Branch...
    ///   Currently , only Print and PrintLine statement
    ///   are supported..
    ///   if a line does not start with Print or PrintLine ..
    ///   an exception is thrown
    /// </summary>
    /// <returns></returns>
    private Stmt Statement()
    {
      Stmt retval = null;
      switch (Current_Token)
      {
        case TOKEN.TOK_PRINT:
          retval = ParsePrintStatement();
          GetNext();
          break;
        case TOKEN.TOK_PRINTLN:
          retval = ParsePrintLNStatement();
          GetNext();
          break;
        default:
          throw new Exception("Invalid statement");
          break;
      }
      return retval;
    }
    /// <summary>
    ///   Parse the Print Staement .. The grammar is
    ///   PRINT <expr> ;
    ///   Once you are in this subroutine , we are expecting
    ///   a valid expression ( which will be compiled ) and a
    ///   semi collon to terminate the line..
    ///   Once Parse Process is successful , we create a PrintStatement
    ///   Object..
    /// </summary>
    /// <returns></returns>
    private Stmt ParsePrintStatement()
```

```
    {
        GetNext();
        Exp a = Expr();

        if (Current_Token != TOKEN.TOK_SEMI)
        {
            throw new Exception("; is expected");
        }
        return new PrintStatement(a);
    }
    /// <summary>
    ///    Parse the PrintLine Staement .. The grammar is
    ///    PRINTLINE <expr> ;
    ///    Once you are in this subroutine , we are expecting
    ///    a valid expression ( which will be compiled ) and a
    ///    semi collon to terminate the line..
    ///    Once Parse Process is successful , we create a PrintLineStatement
    ///    Object..
    /// </summary>
    /// <returns></returns>
    private Stmt ParsePrintLNStatement()
    {
        GetNext();
        Exp a = Expr();

        if (Current_Token != TOKEN.TOK_SEMI)
        {
            throw new Exception("; is expected");
        }
        return new PrintLineStatement(a);
    }
```

Finally in the callSlang Project , I have invoked these routines to demonstrate how everything is put together.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using SLANG_DOT_NET;

namespace CallSLANG
{
    class Program
    {
        static void TestFirstScript()
        {
            string a = "PRINTLINE 2*10;" + "\r\n" + "PRINTLINE 10;\r\n PRINT 2*10;\r\n";
            RDParser p = new RDParser(a);
            ArrayList arr = p.Parse();
            foreach (object obj in arr)
            {
```

```
            Stmt s = obj as Stmt;
            s.Execute(null);
        }
    }
    /// <summary>
    ///
    /// </summary>
    static void TestSecondScript()
    {
        string a = "PRINTLINE -2*10;" + "\r\n" + "PRINTLINE -10*-1;\r\n PRINT 2*10;\r\n";
        RDParser p = new RDParser(a);
        ArrayList arr = p.Parse();
        foreach (object obj in arr)
        {
            Stmt s = obj as Stmt;
            s.Execute(null);
        }
    }
    /// <summary>
    ///
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {
      // TestFirstScript();
        TestSecondScript();
        Console.Read();
    }
  }
}
```

SLANG4.NET

htttp://slangfordotnet.codeplex.com

The Art of Compiler Construction using C#

By

Praseed Pai K.T. (http://praseedp.blogspot.com)
praseedp@yahoo.com

CHAPTER 4

Version 0.1

Types, Variables and Assignment Statement

In this step , we will support data types and variables to SLANG. Assignment statement will also be implemented in this step.

The language supports only three data types viz

NUMERIC
STRING
BOOLEAN

Let us add an Enum for data types

```
  /// <summary>
/// Type info enumerations
/// </summary>
public enum TYPE_INFO
{
   TYPE_ILLEGAL = -1, // NOT A TYPE
   TYPE_NUMERIC,     // IEEE Double precision floating point
   TYPE_BOOL,        // Boolean Data type
   TYPE_STRING,      // String data type
}
```

Every variable will have a name , type and a slot for storing it's value in the symbol table. Moreover , Functions return SYMBOL_INFO.

```
 /// <summary>
///    Symbol Table entry for variable
///    using Attributes , one can optimize the
///    storage by simulating C/C++ union.
/// </summary>
public class SYMBOL_INFO
{
   public String SymbolName;   // Symbol Name
   public TYPE_INFO Type;      // Data type
   public String str_val;      // memory to hold string
   public double dbl_val;      // memory to hold double
   public bool bol_val;      // memory to hold boolean
}
```

The next step is to modify the Exp and Stmt interface to reflect the variable support.

```
/// <summary>
///     In  this Step , we add two more methods to the Exp class
///     TypeCheck => To do Type analysis
///     get_type  => Type of this node
/// </summary>
abstract class Exp
{
    public abstract SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont);
    public abstract TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont);
    public abstract TYPE_INFO get_type();

}
```

The class RunTime context contains the Symbol Table during interpretation.

```
/// <summary>
///   A Context is necessary for Variable scope...will be used later
/// </summary>
public class RUNTIME_CONTEXT
 {
    /// <summary>
    ///    Symbol Table for this context
    /// </summary>
    private SymbolTable m_dt;


    /// <summary>
    ///    Create an instance of Symbol Table
    /// </summary>
    public RUNTIME_CONTEXT()
    {
        m_dt = new SymbolTable();

    }



    /// <summary>
    ///    Property to retrieve Table
    /// </summary>
    public SymbolTable TABLE
    {

        get
        {
            return m_dt;
        }

        set
        {
            m_dt = value;
        }
    }



 }
```

```
/// <summary>
  ///   A Context is necessary for Variable scope...will be used later
  /// </summary>
  public class COMPILATION_CONTEXT
  {
     /// <summary>
     ///    Symbol Table for this context
     /// </summary>
     private SymbolTable m_dt;

     /// <summary>
     ///    Create an instance of Symbol Table
     /// </summary>
     public COMPILATION_CONTEXT()
     {
        m_dt = new SymbolTable();
     }

     /// <summary>
     ///    Property to retrieve Table
     /// </summary>
     public SymbolTable TABLE
     {

        get
        {
           return m_dt;
        }

        set
        {
           m_dt = value;
        }
     }



  }
```

Let us write the code for BooleanConstant node. This will store TRUE or FALSE value.

```
/// <summary>
 /// <summary>
  ///    Node for Boolean Constant ( TRUE, FALSE }
  ///    Value
  /// </summary>
  public class BooleanConstant : Exp
  {
     /// <summary>
     ///     Info Field
```

```
    /// </summary>
    private SYMBOL_INFO info;
    /// <summary>
    ///    Ctor
    /// </summary>
    /// <param name="pvalue"></param>

    public BooleanConstant(bool pvalue)
    {
       info = new SYMBOL_INFO();
       info.SymbolName = null;
       info.bol_val = pvalue;
       info.Type = TYPE_INFO.TYPE_BOOL;
    }
    /// <summary>
    ///    Evaluation of boolean will given the value
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
       return info;
    }
    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
       return info.Type;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override TYPE_INFO get_type()
    {
       return info.Type;
    }
  }
```

The next thing which we should support is NumericConstant. This will store a IEEE 754 double precision floating point value.

```
/// <summary>
  ///
  /// </summary>
  public class NumericConstant : Exp
  {
     /// <summary>
     ///    Info field
     /// </summary>
     private SYMBOL_INFO info;
```

```
    /// <summary>
    ///
    /// </summary>
    /// <param name="pvalue"></param>
    public NumericConstant(double pvalue)
    {
        info = new SYMBOL_INFO();
        info.SymbolName = null;
        info.dbl_val = pvalue;
        info.Type = TYPE_INFO.TYPE_NUMERIC;

    }

    /// <summary>
    ///    Evaluation of boolean will given the value
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
        return info;
    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        return info.Type;
    }
    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override TYPE_INFO get_type()
    {
        return info.Type;
    }


}
```

The AST node for storing String Literal is as given below

```
/// <summary>
  ///   To Store Literal string enclosed
  ///   in Quotes
  /// </summary>
 public class StringLiteral : Exp
```

```
{
    /// <summary>
    ///  info field
    /// </summary>
    private SYMBOL_INFO info;
    /// <summary>
    ///   Ctor
    /// </summary>
    /// <param name="pvalue"></param>
    public StringLiteral(string pvalue)
    {
        info = new SYMBOL_INFO();
        info.SymbolName = null;
        info.str_val = pvalue;
        info.Type = TYPE_INFO.TYPE_STRING;
    }

    /// <summary>
    ///    Evaluation of boolean will given the value
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
        return info;
    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        return info.Type;
    }
    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override TYPE_INFO get_type()
    {
        return info.Type;
    }

}
```

Addding support to variable is an involved activity. The code has been extensively commented to explain the rationale.

```
/// <summary>
  ///    Node to store Variables
  ///    The data types supported are
  ///      NUMERIC
```

```
///     STRING
///     BOOLEAN
///   The node store only the variable name , the
///   associated data will be found in the
///   Symbol Table attached to the
///     COMPILATION_CONTEXT
///
/// </summary>

public class Variable : Exp
{
   private string m_name;  // Var name
   TYPE_INFO _type;       // Type
   /// <summary>
   ///     this Ctor just stores the variable name
   /// </summary>
   /// <param name="inf"></param>
   public Variable(SYMBOL_INFO inf)
   {
      m_name = inf.SymbolName;

   }
   /// <summary>
   ///     Creates a new symbol and puts into the symbol table
   ///     and stores the key ( variable name )
   /// </summary>
   /// <param name="st"></param>
   /// <param name="name"></param>
   /// <param name="_value"></param>
   public Variable(COMPILATION_CONTEXT st, String name, double _value)
   {
      SYMBOL_INFO s = new SYMBOL_INFO();
      s.SymbolName = name;
      s.Type = TYPE_INFO.TYPE_NUMERIC;
      s.dbl_val = _value;
      st.TABLE.Add(s);
      m_name = name;
   }
   /// <summary>
   ///     Creates a new symbol and puts into the symbol table
   ///     and stores the key ( variable name )
   /// </summary>
   /// <param name="st"></param>
   /// <param name="name"></param>
   /// <param name="_value"></param>
   public Variable(COMPILATION_CONTEXT st, String name, bool _value)
   {
      SYMBOL_INFO s = new SYMBOL_INFO();
      s.SymbolName = name;
      s.Type = TYPE_INFO.TYPE_BOOL;
      s.bol_val = _value;
      st.TABLE.Add(s);
      m_name = name;
   }
   /// <summary>
   ///     Creates a new symbol and puts into the symbol table
   ///     and stores the key ( variable name )
   /// </summary>
```

```
/// <param name="st"></param>
/// <param name="name"></param>
/// <param name="_value"></param>
public Variable(COMPILATION_CONTEXT st, String name, string _value)
{
   SYMBOL_INFO s = new SYMBOL_INFO();
   s.SymbolName = name;
   s.Type = TYPE_INFO.TYPE_STRING;
   s.str_val = _value;
   st.TABLE.Add(s);
   m_name = name;
}

/// <summary>
///    Retrieves the name of the Variable ( method version )
/// </summary>
/// <returns></returns>

public string GetName()
{
   return m_name;
}

/// <summary>
///   Retrieves the name of the Variable ( property version )
/// </summary>
/// <returns></returns>
public string Name
{
   get
   {
      return m_name;
   }

   set
   {
      m_name = value;
   }
}

/// <summary>
///    To Evaluate a variable , we just need to do a lookup
///    in the Symbol table ( of RUNTIME_CONTEXT )
/// </summary>
/// <param name="st"></param>
/// <param name="glb"></param>
/// <returns></returns>
public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
{

   if (cont.TABLE == null)
   {
      return null;
   }
   else
   {
      SYMBOL_INFO a = cont.TABLE.Get(m_name);
      return a;
```

```
        }

    }

    /// <summary>
    ///     Look it up in the Symbol Table and
    ///     return the type
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {

        if (cont.TABLE == null)
        {
            return TYPE_INFO.TYPE_ILLEGAL;
        }
        else
        {
            SYMBOL_INFO a = cont.TABLE.Get(m_name);
            if (a != null)
            {
                _type = a.Type;
                return _type;

            }


            return TYPE_INFO.TYPE_ILLEGAL;

        }

    }

    /// <summary>
    ///     this should only be called after the TypeCheck method
    ///     has been invoked on AST
    /// </summary>
    /// <returns></returns>
    public override TYPE_INFO get_type()
    {
        return _type;
    }


}
```

At this point of time Expression hierarchy (without operators) looks like as follows

class Exp
    class BooleanConstant
    class NumericConstant
    class StringLiteral
    class Variable

Once we have created nodes to represents constants of the type which we are planning to support , we created a variable node. The next challenge is to add support for the operators. Till now , I had UnaryExp and BinaryExp. For clarity , I will have classes like Plus (+) , Minus (-) , Div (/) and Mul (*) for BinaryExp and will have classes UnaryPlus (+) , UnaryMinus (-) for Unary operators

The first operator to be supported is Binary +

```
/// <summary>
  ///    the node to represent Binary +
  /// </summary>

  public class BinaryPlus : Exp
  {
     /// <summary>
     ///  Plus has got a left expression (exp1 )
     ///  and a right expression...
     ///  and a Associated type information
     /// </summary>
     private Exp exp1, exp2;
     TYPE_INFO _type;
     /// <summary>
     ///
     /// </summary>
     /// <param name="e1"></param>
     /// <param name="e2"></param>
     public BinaryPlus(Exp e1, Exp e2)
     {
        exp1 = e1; exp2 = e2;
     }
     /// <summary>
     ///
     /// </summary>
     /// <param name="cont"></param>
     /// <returns></returns>

     public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
     {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        SYMBOL_INFO eval_right = exp2.Evaluate(cont);

        if (eval_left.Type == TYPE_INFO.TYPE_STRING &&
           eval_right.Type == TYPE_INFO.TYPE_STRING)
```

```
        {
            SYMBOL_INFO ret_val = new SYMBOL_INFO();
            ret_val.str_val = eval_left.str_val + eval_right.str_val;
            ret_val.Type = TYPE_INFO.TYPE_STRING;
            ret_val.SymbolName = "";
            return ret_val;
        }
        else if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC &&
            eval_right.Type == TYPE_INFO.TYPE_NUMERIC)
        {
            SYMBOL_INFO ret_val = new SYMBOL_INFO();
            ret_val.dbl_val = eval_left.dbl_val + eval_right.dbl_val;
            ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
            ret_val.SymbolName = "";
            return ret_val;

        }
        else
        {
            throw new Exception("Type mismatch");
        }

    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        TYPE_INFO eval_left = exp1.TypeCheck(cont);
        TYPE_INFO eval_right = exp2.TypeCheck(cont);

        if (eval_left == eval_right && eval_left != TYPE_INFO.TYPE_BOOL)
        {
            _type = eval_left;
            return _type;
        }
        else
        {
            throw new Exception("Type mismatch failure");

        }
    }




    public override TYPE_INFO get_type()
    {
        return _type;
    }

}
```

Where as Evaluate routine for StringLiteral , NumericConstant , BooleanConstant and Variable just involves returning the SYMBOL_INO , in the case of Operators things are bit evolved...

```
/// <summary>
        ///
        /// </summary>
        /// <param name="cont"></param>
        /// <returns></returns>

        public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
        {
            SYMBOL_INFO eval_left = exp1.Evaluate(cont);
            SYMBOL_INFO eval_right = exp2.Evaluate(cont);

            if (eval_left.Type == TYPE_INFO.TYPE_STRING &&
                eval_right.Type == TYPE_INFO.TYPE_STRING)
            {
                SYMBOL_INFO ret_val = new SYMBOL_INFO();
                ret_val.str_val = eval_left.str_val + eval_right.str_val;
                ret_val.Type = TYPE_INFO.TYPE_STRING;
                ret_val.SymbolName = "";
                return ret_val;
            }
            else if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC &&
                eval_right.Type == TYPE_INFO.TYPE_NUMERIC)
            {
                SYMBOL_INFO ret_val = new SYMBOL_INFO();
                ret_val.dbl_val = eval_left.dbl_val + eval_right.dbl_val;
                ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
                ret_val.SymbolName = "";
                return ret_val;

            }
            else
            {
                throw new Exception("Type mismatch");
            }

        }
```

In the above code snippet , Left and Right expressions are evaluated and the types are queried. In our compiler , operations involving numerics and strings are successful only if all the operands are of the same type.

The routine TypeCheck is similar to Evaluate. Only difference is that TypeCheck updates the type information of the nodes in a Recursive manner. The routine get_type is only valid once you have called TypeCheck routine.

BinaryMinus is similar to BinaryPlus. The only difference is only Numerics can be subtracted.

```
/// <summary>
  ///
  /// </summary>

  class BinaryMinus : Exp
  {
     private Exp exp1, exp2;
     TYPE_INFO _type;

     public BinaryMinus(Exp e1, Exp e2)
     {
        exp1 = e1; exp2 = e2;
     }


     public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
     {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        SYMBOL_INFO eval_right = exp2.Evaluate(cont);

        if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC &&
           eval_right.Type == TYPE_INFO.TYPE_NUMERIC)
        {
           SYMBOL_INFO ret_val = new SYMBOL_INFO();
           ret_val.dbl_val = eval_left.dbl_val - eval_right.dbl_val;
           ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
           ret_val.SymbolName = "";
           return ret_val;

        }
        else
        {
           throw new Exception("Type mismatch");
        }

     }


     /// <summary>
     ///
     /// </summary>
     /// <param name="local"></param>
     /// <param name="global"></param>
     /// <returns></returns>
     public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
     {
        TYPE_INFO eval_left = exp1.TypeCheck(cont);
        TYPE_INFO eval_right = exp2.TypeCheck(cont);

        if (eval_left == eval_right && eval_left == TYPE_INFO.TYPE_NUMERIC)
        {
           _type = eval_left;
           return _type;
        }
        else
```

```
        {
            throw new Exception("Type mismatch failure");

        }
    }

    public override TYPE_INFO get_type()
    {
        return _type;
    }



}
```

Multiplication and Division operators are only valid for Numeric Types. If you have understood the implementation of BinaryPlus , the Mul and Div operators are easy to follow

```
///
  /// <summary>
  ///
  /// </summary>

  class Mul : Exp
  {
    private Exp exp1, exp2;
    TYPE_INFO _type;

    public Mul(Exp e1, Exp e2)
    {
        exp1 = e1; exp2 = e2;
    }




    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        SYMBOL_INFO eval_right = exp2.Evaluate(cont);

        if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC &&
            eval_right.Type == TYPE_INFO.TYPE_NUMERIC)
        {
            SYMBOL_INFO ret_val = new SYMBOL_INFO();
            ret_val.dbl_val = eval_left.dbl_val * eval_right.dbl_val;
            ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
            ret_val.SymbolName = "";
            return ret_val;

        }
        else
        {
            throw new Exception("Type mismatch");
```

```
        }

    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        TYPE_INFO eval_left = exp1.TypeCheck(cont);
        TYPE_INFO eval_right = exp2.TypeCheck(cont);

        if (eval_left == eval_right && eval_left == TYPE_INFO.TYPE_NUMERIC)
        {
            _type = eval_left;
            return _type;
        }
        else
        {
            throw new Exception("Type mismatch failure");

        }
    }

    public override TYPE_INFO get_type()
    {
        return _type;
    }




}
/// <summary>
///
/// </summary>

class Div : Exp
{
    private Exp exp1, exp2;
    TYPE_INFO _type;

    public Div(Exp e1, Exp e2)
    {
        exp1 = e1; exp2 = e2;
    }


    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        SYMBOL_INFO eval_right = exp2.Evaluate(cont);

        if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC &&
```

```
            eval_right.Type == TYPE_INFO.TYPE_NUMERIC)
        {
            SYMBOL_INFO ret_val = new SYMBOL_INFO();
            ret_val.dbl_val = eval_left.dbl_val / eval_right.dbl_val;
            ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
            ret_val.SymbolName = "";
            return ret_val;

        }
        else
        {
            throw new Exception("Type mismatch");
        }

    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        TYPE_INFO eval_left = exp1.TypeCheck(cont);
        TYPE_INFO eval_right = exp2.TypeCheck(cont);

        if (eval_left == eval_right && eval_left == TYPE_INFO.TYPE_NUMERIC)
        {
            _type = eval_left;
            return _type;
        }
        else
        {
            throw new Exception("Type mismatch failure");

        }
    }

    public override TYPE_INFO get_type()
    {
        return _type;
    }



}
```

UnaryPlus and UnaryMinus is also similar to the implementation of other operators. Both these operators are only applicable for Numeric data type.

```
/// <summary>
  ///    the node to represent Unary +
  /// </summary>

  class UnaryPlus : Exp
  {
     /// <summary>
     ///  Plus has got a right expression (exp1 )
     ///  and a Associated type information
     /// </summary>
     private Exp exp1;
     TYPE_INFO _type;
     /// <summary>
     ///
     /// </summary>
     /// <param name="e1"></param>
     /// <param name="e2"></param>
     public UnaryPlus(Exp e1)
     {
        exp1 = e1;
     }
     /// <summary>
     ///
     /// </summary>
     /// <param name="cont"></param>
     /// <returns></returns>

     public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
     {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC)
        {
           SYMBOL_INFO ret_val = new SYMBOL_INFO();
           ret_val.dbl_val = eval_left.dbl_val;
           ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
           ret_val.SymbolName = "";
           return ret_val;

        }
        else
        {
           throw new Exception("Type mismatch");
        }

     }


     /// <summary>
     ///
     /// </summary>
     /// <param name="local"></param>
     /// <param name="global"></param>
     /// <returns></returns>
     public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
     {
```

```
        TYPE_INFO eval_left = exp1.TypeCheck(cont);


        if (eval_left == TYPE_INFO.TYPE_NUMERIC)
        {
            _type = eval_left;
            return _type;
        }
        else
        {
            throw new Exception("Type mismatch failure");

        }
    }




    public override TYPE_INFO get_type()
    {
        return _type;
    }

}

/// <summary>
///    the node to represent Unary -
/// </summary>

class UnaryMinus : Exp
{
    /// <summary>
    ///  Plus has got a right expression (exp1 )
    ///  and a Associated type information
    /// </summary>
    private Exp exp1;
    TYPE_INFO _type;
    /// <summary>
    ///
    /// </summary>
    /// <param name="e1"></param>
    /// <param name="e2"></param>
    public UnaryMinus(Exp e1)
    {
        exp1 = e1;
    }
    /// <summary>
    ///
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>

    public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
        SYMBOL_INFO eval_left = exp1.Evaluate(cont);
        if (eval_left.Type == TYPE_INFO.TYPE_NUMERIC)
        {
            SYMBOL_INFO ret_val = new SYMBOL_INFO();
```

```
            ret_val.dbl_val = -eval_left.dbl_val;
            ret_val.Type = TYPE_INFO.TYPE_NUMERIC;
            ret_val.SymbolName = "";
            return ret_val;

        }
        else
        {
            throw new Exception("Type mismatch");
        }

    }


    /// <summary>
    ///
    /// </summary>
    /// <param name="local"></param>
    /// <param name="global"></param>
    /// <returns></returns>
    public override TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont)
    {
        TYPE_INFO eval_left = exp1.TypeCheck(cont);


        if (eval_left == TYPE_INFO.TYPE_NUMERIC)
        {
            _type = eval_left;
            return _type;
        }
        else
        {
            throw new Exception("Type mismatch failure");

        }
    }




    public override TYPE_INFO get_type()
    {
        return _type;
    }

}
```

The statement related nodes are moved to a seperate module by the name AstForStatements.

In this step , we have added support for Variable Declaration and Assignment statement.

The AST for Variable declaration is given below

```
/// <summary>
    ///    Compile the Variable Declaration statements
    /// </summary>
    public class VariableDeclStatement : Stmt
    {
        /// <summary>
        ///
        /// </summary>
        SYMBOL_INFO m_inf = null;
        Variable var = null;
        /// <summary>
        ///
        /// </summary>
        /// <param name="inf"></param>
        public VariableDeclStatement(SYMBOL_INFO inf)
        {
            m_inf = inf;

        }
        /// <summary>
        ///
        /// </summary>
        /// <param name="cont"></param>
        /// <returns></returns>
        public override SYMBOL_INFO Execute(RUNTIME_CONTEXT cont)
        {
            cont.TABLE.Add(m_inf);
            var = new Variable(m_inf);
            return null;

        }



    }
```

In the parser , before we create VariableDeclStatement node , we need to insert the variable's SYMBOL_INFO into the SymbolTable of the COMPILATION_CONTEXT. The VariableDeclStatement node just stores the variable name in the Variable AST.

While Executing the VariableDeclStatement , a Variable is created in the Symbol table of RUNTIME_CONTEXT.

Both Compilation Context (COMPILATION_CONTEXT) and Run time Context (RUNTIME_CONTEXT) just contains references to respective symbol tables.

The AST for Assignment statement is given below

```
/// <summary>
  ///   Assignment Statement
  /// </summary>
  public class AssignmentStatement : Stmt
  {
     private Variable variable;
     private Exp exp1;

     public AssignmentStatement(Variable var, Exp e)
     {
        variable = var;
        exp1 = e;

     }

     public AssignmentStatement(SYMBOL_INFO var, Exp e)
     {
        variable = new Variable(var);
        exp1 = e;

     }
     public override SYMBOL_INFO Execute(RUNTIME_CONTEXT cont)
     {
        SYMBOL_INFO val = exp1.Evaluate(cont);
        cont.TABLE.Assign(variable, val);
        return null;
     }



  }
```

At this point of time , AST for Statements is as shown below

class Stmt
  class VariableDeclStatement
  class AssignmentStatement
  class PrintStatement
  class PrintLineStatement

The class SymbolTable is just a vector of name/value pair . The source code of the SymbolTable is given below.

```
/// <summary>
  ///   Symbol Table for Parsing and Type Analysis
  /// </summary>
  public class SymbolTable
  {
     /// <summary>
```

```
    ///    private data structure
    /// </summary>
    private System.Collections.Hashtable dt = new Hashtable();
    /// <summary>
    ///    Add a symbol to Symbol Table
    /// </summary>
    /// <param name="s"></param>
    /// <returns></returns>
    public bool Add(SYMBOL_INFO s)
    {
       dt[s.SymbolName] = s;
       return true;
    }

    /// <summary>
    ///    Retrieve the Symbol
    /// </summary>
    /// <param name="name"></param>
    /// <returns></returns>
    public SYMBOL_INFO Get(string name)
    {
       return dt[name] as SYMBOL_INFO;
    }

    /// <summary>
    ///    Assign to the Symbol Table
    /// </summary>
    /// <param name="var"></param>
    /// <param name="value"></param>
    public void Assign(Variable var, SYMBOL_INFO value)
    {
       value.SymbolName = var.GetName();
       dt[var.GetName()] = value;

    }
    /// <summary>
    ///   Assign to a variable
    /// </summary>
    /// <param name="var"></param>
    /// <param name="value"></param>
    public void Assign(string var, SYMBOL_INFO value)
    {
       dt[var] = value;
    }

}
```

The class CsyntaxErrorLog and CsemanticErrorLog (in SupportClasses.cs) is meant for error logging while the compilation process is going on.....

Let us go back to Lexical Analysis stage once again. This time we have added lot of new keywords to the language and Token set has become bit larger than the previous step.

```
public enum TOKEN
   {
      ILLEGAL_TOKEN = -1, // Not a Token
      TOK_PLUS = 1, // '+'
      TOK_MUL, // '*'
      TOK_DIV, // '/'
      TOK_SUB, // '-'
      TOK_OPAREN, // '('
      TOK_CPAREN, // ')'

      TOK_NULL, // End of string
      TOK_PRINT, // Print Statement
      TOK_PRINTLN, // PrintLine
      TOK_UNQUOTED_STRING, // Variable name , Function name etc
      TOK_SEMI , // ;

      //---------- Addition in Step 4

      TOK_VAR_NUMBER,       // NUMBER data type
      TOK_VAR_STRING,       // STRING data type
      TOK_VAR_BOOL,         // Bool data type
      TOK_NUMERIC,          // [0-9]+
      TOK_COMMENT ,     // Comment Token ( presently not used )
      TOK_BOOL_TRUE,        // Boolean TRUE
      TOK_BOOL_FALSE  ,   // Boolean FALSE
      TOK_STRING,        // String Literal
      TOK_ASSIGN         // Assignment Symbol =


   }
```

We have also moved couple of routines and state variables to Lexer class .

The two notable addition are

```
      /// <summary>
      ///    Current Token and Last Grabbed Token
      /// </summary>
      protected TOKEN Current_Token;  // Current Token
      protected TOKEN Last_Token;     // Penultimate token
```

Since we have added support for string type... , we need to support string literals (or the last grabbed string) in the lexical analyzer...

```
/// <summary>
    ///       last_str := Token assoicated with
    /// </summary>

    public String last_str; // Last grabbed String
```

We need to update the keyword table with additional key words supported by the compiler

```
        ////////////////////////////////////////////////
        // Fill the Keywords
        //
        //
        keyword = new ValueTable[7];

        keyword[0] = new ValueTable(TOKEN.TOK_BOOL_FALSE, "FALSE");
        keyword[1] = new ValueTable(TOKEN.TOK_BOOL_TRUE, "TRUE");
        keyword[2] = new ValueTable(TOKEN.TOK_VAR_STRING, "STRING");
        keyword[3] = new ValueTable(TOKEN.TOK_VAR_BOOL, "BOOLEAN");
        keyword[4] = new ValueTable(TOKEN.TOK_VAR_NUMBER, "NUMERIC");
        keyword[5] = new ValueTable(TOKEN.TOK_PRINT, "PRINT");
        keyword[6] = new ValueTable(TOKEN.TOK_PRINTLN, "PRINTLINE");
```

The Parsing of the statements starts from Parse Routine of RDParser.cs

```
    / // <summary>
    ///   The new Parser entry point
    /// </summary>
    /// <returns></returns>
    public ArrayList Parse(COMPILATION_CONTEXT ctx)
    {
        GetNext();  // Get the Next Token
        //
        // Parse all the statements
        //
        return StatementList(ctx);
    }
```

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

The Logic of the StatementList is as follows,

while there is more statements
    Parse and Add Statements to the ArrayList

The Grammar supported is given as comments

```
/// <summary>
      ///  The Grammar is
      ///
      ///  <stmtlist> :=  { <statement> }+
      ///
      ///  {<statement> :=  <printstmt> | <printlinestmt>
      ///  <printstmt> :=   print   <expr >;
      ///  <vardeclstmt> := STRING <varname>;  |
      ///                   NUMERIC <varname>; |
      ///                   BOOLEAN <varname>;
      ///
      /// <printlinestmt>:= printline <expr>;
      ///
      /// <Expr>  ::=  <Term> | <Term> { + | - } <Expr>
      /// <Term> ::=  <Factor> | <Factor>  {*|/} <Term>
      /// <Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
      ///              <variable> | TRUE | FALSE
      ///
      ///
      ///
      /// </summary>
      /// <returns></returns>
      private ArrayList StatementList(COMPILATION_CONTEXT ctx)
      {
         ArrayList arr = new ArrayList();
         while (Current_Token != TOKEN.TOK_NULL)
         {
            Stmt temp = Statement(ctx);
            if (temp != null)
            {
               arr.Add(temp);
            }
         }
         return arr;
      }
```

The Statement method just queries the statement type and calls the appropriate Parse Routines

```
/// <summary>
      ///    This Routine Queries Statement Type
      ///    to take the appropriate Branch...
      ///    Currently , only Print and PrintLine statement
      ///    are supported..
      ///    if a line does not start with Print or PrintLine ..
      ///    an exception is thrown
      /// </summary>
      /// <returns></returns>
      private Stmt Statement(COMPILATION_CONTEXT ctx)
```

```
    {
        Stmt retval = null;
        switch (Current_Token)
        {
            case TOKEN.TOK_VAR_STRING:
            case TOKEN.TOK_VAR_NUMBER:
            case TOKEN.TOK_VAR_BOOL:
                retval = ParseVariableStatement(ctx);
                GetNext();
                return retval;
            case TOKEN.TOK_PRINT:
                retval = ParsePrintStatement(ctx);
                GetNext();
                break;
            case TOKEN.TOK_PRINTLN:
                retval = ParsePrintLNStatement(ctx);
                GetNext();
                break;
            case TOKEN.TOK_UNQUOTED_STRING:
                retval = ParseAssignmentStatement(ctx);
                GetNext();
                return retval;

            default:
                throw new Exception("Invalid statement");

        }
        return retval;
    }
```

The Source code of the ParseVariableDeclStatement is as given below

```
/// <summary>
    ///    Parse Variable declaration statement
    /// </summary>
    /// <param name="type"></param>

    public Stmt ParseVariableDeclStatement(COMPILATION_CONTEXT ctx)
    {

        //--- Save the Data type
        TOKEN tok = Current_Token;

        // --- Skip to the next token , the token ought
        // to be a Variable name ( UnQouted String )
        GetNext();

        if (Current_Token == TOKEN.TOK_UNQUOTED_STRING)
        {
            SYMBOL_INFO symb = new SYMBOL_INFO();
            symb.SymbolName = base.last_str;
            symb.Type = (tok == TOKEN.TOK_VAR_BOOL) ?
            TYPE_INFO.TYPE_BOOL : (tok == TOKEN.TOK_VAR_NUMBER) ?
            TYPE_INFO.TYPE_NUMERIC : TYPE_INFO.TYPE_STRING;

            //---------- Skip to Expect the SemiColon
```

```
            GetNext();



            if (Current_Token == TOKEN.TOK_SEMI)
            {
               // ---------- Add to the Symbol Table
               // for type analysis
               ctx.TABLE.Add(symb);

               // --------- return the Object of type
               // --------- VariableDeclStatement
               // This will just store the Variable name
               // to be looked up in the above table
               return new VariableDeclStatement(symb);
            }
            else
            {
               CSyntaxErrorLog.AddLine("; expected");
               CSyntaxErrorLog.AddLine(GetCurrentLine(SaveIndex()));
               throw new CParserException(-100, ", or ; expected", SaveIndex());
            }
         }
         else
         {

            CSyntaxErrorLog.AddLine("invalid variable declaration");
            CSyntaxErrorLog.AddLine(GetCurrentLine(SaveIndex()));
            throw new CParserException(-100, ", or ; expected", SaveIndex());
         }
      }
```

Assignment statement is easy to parse as the required ground work has already been done... !

```
/// <summary>
      ///    Parse the Assignment Statement
      ///    <variable> = <expr>
      /// </summary>
      /// <param name="pb"></param>
      /// <returns></returns>
      public Stmt ParseAssignmentStatement(COMPILATION_CONTEXT ctx)
      {

         //
         // Retrieve the variable and look it up in
         // the symbol table ..if not found throw exception
         //
         string variable = base.last_str;
         SYMBOL_INFO s = ctx.TABLE.Get(variable);
         if (s == null)
         {
            CSyntaxErrorLog.AddLine("Variable not found  " + last_str);
            CSyntaxErrorLog.AddLine(GetCurrentLine(SaveIndex()));
            throw new CParserException(-100, "Variable not found", SaveIndex());

         }
```

```
    //------------ The next token ought to be an assignment
    // expression....

    GetNext();

    if (Current_Token != TOKEN.TOK_ASSIGN)
    {

       CSyntaxErrorLog.AddLine("= expected");
       CSyntaxErrorLog.AddLine(GetCurrentLine(SaveIndex()));
       throw new CParserException(-100, "= expected", SaveIndex());

    }

    //-------- Skip the token to start the expression
    // parsing on the RHS
    GetNext();
    Exp exp = Expr(ctx);

    //------------ Do the type analysis ...

    if (exp.TypeCheck(ctx) != s.Type)
    {
       throw new Exception("Type mismatch in assignment");

    }

    // -------------- End of statement ( ; ) is expected
    if (Current_Token != TOKEN.TOK_SEMI)
    {
       CSyntaxErrorLog.AddLine("; expected");
       CSyntaxErrorLog.AddLine(GetCurrentLine(SaveIndex()));
       throw new CParserException(-100, " ; expected", -1);

    }
    // return an instance of AssignmentStatement node..
    //   s => Symbol info associated with variable
    //   exp => to evaluated and assigned to symbol_info
    return new AssignmentStatement(s, exp);

  }
```

Parsing Expressions

The grammar for expression is given below

```
<Expr>  ::=  <Term> | <Term> { + | - } <Expr>
<Term> ::=  <Factor> | <Factor>  {*|/} <Term>
<Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
                <variable> | TRUE | FALSE
```

Let us take a look at the Expression routine , ie the top most expression parsing routine at this point of time... (In future , when logical expressions and relational expressions are added , we modify the

grammar)

```
/// <summary>
///    <Expr>  ::=  <Term> | <Term> { + | - } <Expr>
///
/// </summary>
/// <returns></returns>
public Exp Expr(COMPILATION_CONTEXT ctx)
{
   TOKEN l_token;
   Exp RetValue = Term(ctx);
   while (Current_Token == TOKEN.TOK_PLUS || Current_Token == TOKEN.TOK_SUB)
   {
      l_token = Current_Token;
      Current_Token = GetToken();
      Exp e1 = Expr(ctx);

      if (l_token == TOKEN.TOK_PLUS)
         RetValue = new BinaryPlus(RetValue, e1);
      else
         RetValue = new BinaryMinus(RetValue, e1);
   }

   return RetValue;

}
```

The Term routine handles the mul and the div operators

```
/// <summary>
/// <Term> ::=  <Factor> | <Factor>  {*|/} <Term>
/// </summary>
public Exp Term(COMPILATION_CONTEXT ctx)
{
   TOKEN l_token;
   Exp RetValue = Factor(ctx);

   while (Current_Token == TOKEN.TOK_MUL || Current_Token == TOKEN.TOK_DIV)
   {
      l_token = Current_Token;
      Current_Token = GetToken();


      Exp e1 = Term(ctx);
      if (l_token == TOKEN.TOK_MUL)
         RetValue = new Mul(RetValue, e1);
      else
         RetValue = new Div(RetValue, e1);

   }

   return RetValue;
}
```

The factor routine is where we handle Variables , unary Operations , Constants etc....

```
/// <summary>
    ///     <Factor>::=  <number> | ( <expr> ) | {+|-} <factor>
    ///          <variable> | TRUE | FALSE
    /// </summary>
    public Exp Factor(COMPILATION_CONTEXT ctx)
    {
       TOKEN l_token;
       Exp RetValue = null;



       if (Current_Token == TOKEN.TOK_NUMERIC)
       {

          RetValue = new NumericConstant(GetNumber());
          Current_Token = GetToken();

       }
       else if (Current_Token == TOKEN.TOK_STRING)
       {
          RetValue = new StringLiteral(last_str);
          Current_Token = GetToken();
       }
       else if (Current_Token == TOKEN.TOK_BOOL_FALSE ||
             Current_Token == TOKEN.TOK_BOOL_TRUE)
       {
          RetValue = new BooleanConstant(
             Current_Token == TOKEN.TOK_BOOL_TRUE ? true : false);
          Current_Token = GetToken();
       }
       else if (Current_Token == TOKEN.TOK_OPAREN)
       {

          Current_Token = GetToken();

          RetValue = Expr(ctx);  // Recurse

          if (Current_Token != TOKEN.TOK_CPAREN)
          {
             Console.WriteLine("Missing Closing Parenthesis\n");
             throw new Exception();

          }
          Current_Token = GetToken();
       }

       else if (Current_Token == TOKEN.TOK_PLUS || Current_Token == TOKEN.TOK_SUB)
       {
          l_token = Current_Token;
          Current_Token = GetToken();
          RetValue = Factor(ctx);
          if (l_token == TOKEN.TOK_PLUS)
             RetValue = new UnaryPlus(RetValue);
          else
             RetValue = new UnaryMinus(RetValue);
```

```
        }
        else if (Current_Token == TOKEN.TOK_UNQUOTED_STRING)
        {
            ///
            ///  Variables
            ///
            String str = base.last_str;
            SYMBOL_INFO inf = ctx.TABLE.Get(str);

            if (inf == null)
                throw new Exception("Undefined symbol");

            GetNext();
            RetValue = new Variable(inf);
        }

       else
        {

            Console.WriteLine("Illegal Token");
            throw new Exception();
        }


        return RetValue;

    }
```

In the CallSlang Project , this is how we need to invoke the Script....

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using System.IO;
using SLANG_DOT_NET;

namespace CallSLANG
{
   class Program
   {

      /// <summary>
      ///    Driver routine to call the program script
      /// </summary>
      static void TestFileScript(string filename)
      {

         if (filename == null)
               return;


         // ---------- Read the contents from the file
```

```
        StreamReader sr = new StreamReader(filename);
        string programs2 = sr.ReadToEnd();


        //---------------- Creates the Parser Object
        // With Program text as argument
        RDParser pars = null;
        pars = new RDParser(programs2);

        // Create a Compilation Context
        //
        //
        COMPILATION_CONTEXT ctx = new COMPILATION_CONTEXT();

        //
        // Call the top level Parsing Routine with
        // Compilation Context as the Argument
        //
        ArrayList stmts = pars.Parse(ctx);

        //
        // if we have reached here , the parse process
        // is successful... Create a Run time context and
        // Call Execute statements of each statement...
        //

        RUNTIME_CONTEXT f = new RUNTIME_CONTEXT();
        foreach(Object obj in stmts )
        {
           Stmt s = obj as Stmt;
           s.Execute(f);
        }

    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {


        if (args == null ||
            args.Length != 1)
        {
           Console.WriteLine("CallSlang <scriptname>\n");
           return;

        }
        TestFileScript(args[0]);
        //------------- Wait for the Key Press
        Console.Read();

    }
  }
}
```

First.sl (Slang script)

```
////////////////////////////////
//
// A simple SLANG script
//
//    Sample #1
//
NUMERIC a;   // Declare a Numeric variable

a = 2*3+5* 30 + -(4*5+3);  // Assign

PRINTLINE a;  // Dump a

//----- String concatenation

PRINT "Hello " + "World";
//-------------- Write a new line

PRINTLINE "";
//-------------- string data type
STRING c;

c = "Hello ";  // assignment to string

//---------------- assignment and concatenation

C = C + "World";

PRINTLINE c;

//-------------- boolean variable

BOOLEAN d;

d= TRUE;

PRINTLINE d;

d= FALSE;

PRINTLINE d;
```

Second.sl

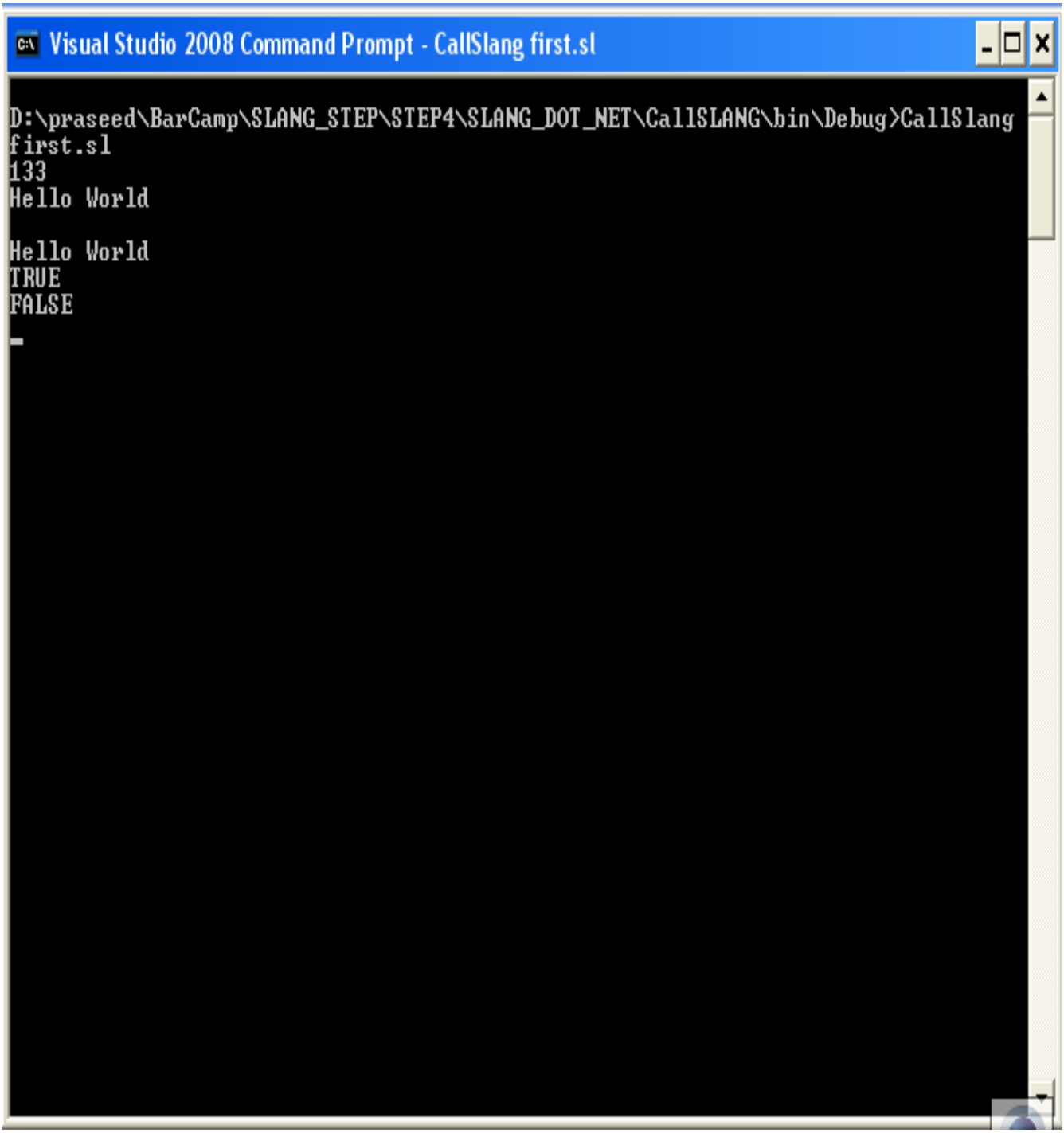
```
////////////////////////////////////
//
// A slang script to test unary expression
//
//
//


Numeric a ;

a = -1 ;

a = -a;

Print a;
```

Third.sl

```
//////////////////////
//
//
//
//
//


Numeric a;

String b;


a = ---1;

PrintLine a*4 + 10;
```

You can go to the command line and invoke the CallSlang executable as follows....a screen snapshot is given in the next page

CallSlang first.sl
CallSlang second.sl
CallSlang third.sl


Visual Studio 2008 Command Prompt - CallSlang first.sl
D:\praseed\BarCamp\SLANG_STEP\STEP4\SLANG_DOT_NET\CallSLANG\bin\Debug>CallSlang first.sl
133
Hello World
Hello World
TRUE
FALSE

SLANG4.NET

htttp://slangfordotnet.codeplex.com

The Art of Compiler Construction using C#

By

Praseed Pai K.T. (http://praseedp.blogspot.com)
praseedp@yahoo.com

CHAPTER 5

Version 0.1

Compilation To a .NET executable !

Till now , the Slang Compiler was interpreting the statements and expressions. In this step we will try to compile the source into .net IL (.net intermediate Language)

In this step , we are not going to make any change into the SLANG grammar. With Future (support for Logical Expression, Relational Expression, Control structures and Procedures) in mind , we will re factor the code.

.NET IL uses a Stack based machine model. Your Intel x86 machines are register based. Stack based machine was touted as the future of machines in the early 1980s. What this basically means , the .NET CLR has got a evaluation stack for processing expressions and functions....

Read more about Reflection.Emit , CLR and IL to understand this.

To give a quick feel of what is involed , let us think about the expression 2*(3+4)

This expression can be decomposed into

```
Acc = ADD 3 and 4  ( Acc stands for accumulator )
Acc =  Acc MUL 2 ;
```

if we are using a stack as an accumulator (which .net does) this can be written as

```
PUSH 3      // now stack contains    [ 3 ]
PUSH 4      // now stack constains   [ 3 4 ]
ADD         //  pop 4 , pop 3 , add  , push result
            //   Now stack contains  [ 7 ]
PUSH 2      //   Now stack conains   [  7 2 ]
MUL         //    pop 2 , pop 7 , multiply , push the result
            //    now stack contains the result [ 14 ]
```

when the same stuff was compiled into .net ...this is what we got .....

```
// ---------------- IL output of 2*(3+4)
 IL_0000:  ldc.r8    2.    //   push 2                  [ 2 ]
 IL_0009:  ldc.r8    3.    //   push  3                 [ 2 , 3 ]
 IL_0012:  ldc.r8    4.    //   push  4                 [ 2,3,4]
 IL_001b:  add             //   pop 4 , pop 3 , add , push result   [ 2,7 ]
 IL_001c:  mul             //    pop 7 , pop 2 , mul , push result  [ 14 ]
```

This out put was generated using ILDASM.exe utility. This will help to disassemble .net executables.

To generate .NET executable , in the System.Reflection namespace there are some classes which we need to be familiar.

```
AssemblyBuilder
 ModuleBuilder
 TypeBuilder
 MethodBuilder
```

From the name we can infer that these classes are based on GOF Builder pattern. The .NET StringBuilder class is another notable class which is based on Builder pattern. Consult Wikipedia or a decent book on design patterns to learn more about Builder Pattern. Later , we will be using Builder Pattern to Build (!) Procedure and Module from Program text.

An Assembly (.NET exe or .net DLL) is composed of Collection of Modules. In most cases , we will have one module per assembly. Module will have collection of Types (classes and structs). Each Type will have collection of method. The hierarchical relationonship between these entities are given below.

```
Hierarchy is as follows..
    Assembly
        Module
           Type
              Method
```

For each entity there is a corresponding builder available. Please include System.Reflection and System.Reflection.Emit to get access to these classes. How this has been used is given below

```
        //
        // Get The App Domain
        //
        //
        AppDomain _app_domain = Thread.GetDomain();
        AssemblyName _asm_name = new AssemblyName();
        //
        //  One can give a strong name , if we want
        //
        _asm_name.Name = "MyAssembly";
        //
        // Save the Exe Name
        //
        _name = name;
        //
        // Create an instance of Assembly Builder
        //
        //
        _asm_builder = AppDomain.CurrentDomain.DefineDynamicAssembly(
            _asm_name,
           AssemblyBuilderAccess.RunAndSave);
        //
        // Create a module builder , from AssemblyBuilder
        //
        _module_builder = _asm_builder.DefineDynamicModule("DynamicModule1", _name, false);
```

```
            //
            // Create a class by the name MainClass..
            // We compile the statements into a static method
            // of the type MainClass .. the entry point will
            // be called Main
            //
            _type_builder = _module_builder.DefineType("MainClass");
```

Now one can use Typebuilder instance to add methods to the type.

All the Executable creation logic has been included in a class by the name ExeGenerator.....

```
/// <summary>
    ///    ExeGenerator - Takes care of the creation of
    ///    .NET executable...
    ///
    /// </summary>
    public class ExeGenerator
    {
        /// <summary>
        ///  Hierarchy is as follows..
        ///    Assembly
        ///       Module
        ///          Type
        ///             Method
        ///   Refer to Reflection.Emit documentation for
        ///   more details on creation of .NET executable
        /// </summary>
        AssemblyBuilder _asm_builder = null;
        ModuleBuilder _module_builder = null;
        TypeBuilder _type_builder = null;

        /// <summary>
        ///     Name of the Executable
        /// </summary>
        string _name = "";

        /// <summary>
        ///     Program to be Compiled...
        /// </summary>
        TModule _p = null;



        /// <summary>
        ///     Ctor which takes Executable name
        ///     as parameter
        /// </summary>
        /// <param name="name"></param>
        public ExeGenerator(TModule  p, string name)
        {
            //
            // The Program to be compiled...
            //
```

```
        _p = p;
        //
        // Get The App Domain
        //
        //
        AppDomain _app_domain = Thread.GetDomain();
        AssemblyName _asm_name = new AssemblyName();
        //
        //  One can give a strong name , if we want
        //
        _asm_name.Name = "MyAssembly";
        //
        // Save the Exe Name
        //
        _name = name;
        //
        // Create an instance of Assembly Builder
        //
        //
        _asm_builder = AppDomain.CurrentDomain.DefineDynamicAssembly(
            _asm_name,
            AssemblyBuilderAccess.RunAndSave);
        //
        // Create a module builder , from AssemblyBuilder
        //
        _module_builder = _asm_builder.DefineDynamicModule("DynamicModule1", _name, false);
        //
        // Create a class by the name MainClass..
        // We compile the statements into a static method
        // of the type MainClass .. the entry point will
        // be called Main
        // ExeGenerator will be called from TModule.Compile method
        // We will add methods to the type MainClass as static method
        //
        _type_builder = _module_builder.DefineType("MainClass");

    }
    /// <summary>
    ///   return the type builder....
    /// </summary>
    public TypeBuilder type_bulder
    {

        get
        {
            return _type_builder;
        }
    }




    /// <summary>
    ///
    /// </summary>

    public void Save()
    {
```

```
        //
        //  Note :- Call this (Save ) method only after
        //  Compilation of All statements....
        _type_builder.CreateType();

        //
        // Retrieve the Entry Point from TModule....
        //
        MethodBuilder mb = _p._get_entry_point("MAIN");

       if (mb != null)
        {
          //
          // Here we will set the Assembly as a Console Application...
          // We will also set the Entry Point....
          //
           _asm_builder.SetEntryPoint(mb, PEFileKinds.ConsoleApplication);
        }
        //
        // Write the Resulting Executable...
        //
        _asm_builder.Save(_name);
    }
}
```

The Above class will be called from Tmodule.CreateExecutable (more about Tmodule later)

```
public bool CreateExecutable(string name)
    {
        //
        // Create an instance of Exe Generator
        // ExeGenerator takes a TModule and
        // exe name as the Parameter...
        _exe = new ExeGenerator(this,name);
        // Compile The module...
        Compile(null);
        // Save the Executable...
        _exe.Save();
        return true;
    }
```

To encapsulate the activities to be undertaken to Compile a Procedure to a static method , we have created a new class by the name , DNET_EXECUTABLE_GENERATION_CONTEXT

For Compiling each Subroutine , an instance of this class will be created. This class contains

a) ArrayList to store local variables...

```
/// <summary>
    ///    Auto (Local) Variable support
    ///    Stores the index return by DefineLocal
    ///    method of MethodBuilder
    /// </summary>
```

```
private ArrayList variables = new ArrayList();
```

b) Reference to an ILGenerator Object.

```
// <summary>
///     ILGenerator Object
/// </summary>
private ILGenerator ILout;
```

c) SymbolTable for doing type analysis

```
/// <summary>
///    Symbol Table for storing Types and
///    doing the type analysis
/// </summary>
SymbolTable m_tab = new SymbolTable();
```

At this point of time , we are not supplying Procedure name to this class. We do not support user defined subroutines in this step. This class only generates a method by the name MAIN.

By inspecting the Ctor of DNET_EXECTUABLE_GENERATION_CONTEXT , we can get a fair idea of the usage of this class...

```
public DNET_EXECUTABLE_GENERATION_CONTEXT(TModule  program ,
                                          Procedure proc,
                                          TypeBuilder bld)
    {
      // All the code in the Source module is compiled
      // into this procedure
      _proc = proc;
      //
      //  TModule Object
      //
      _program = program;
      //
      // Handle to the type (MainClass )
      //
      _bld = bld;
      //
      //  The method does not take any procedure
      System.Type[] s = null;
      // Return type is void
      System.Type ret_type = null;
      //
      //  public static void Main()
      //
      _methinfo = _bld.DefineMethod("Main",
         MethodAttributes.Public | MethodAttributes.Static,
         ret_type, s);
      //
      // We have created the Method Prologue
      // Get the handle to the code generator
      //
      //
      ILout = _methinfo.GetILGenerator();
```

```
    }
```

Creataion of Local Variables is not a difficult task. The ILGenerator class has got a method called DeclareLocal for that. The following code snippet will explain the logic involved..

```
public int DeclareLocal(System.Type type)
    {
      // It is possible to create Local ( auto )
      // Variables by Calling DeclareLocl method
      // of ILGenerator... this returns an integer
      // We store this integer value in the variables
      // collection...
      LocalBuilder lb = ILout.DeclareLocal(type);
      // Now add the integer value associated with the
      // variable to variables collection...
      return variables.Add(lb);
    }
```

The entire source code of DNET_EXECUTABLE_GENERATION_CONTEXT class is given below

```
/// <summary>
 ///     DNET_EXECUTABLE_GENERATION_CONTEXT is for generating
 ///     CLR executable
 /// </summary>
 public class DNET_EXECUTABLE_GENERATION_CONTEXT
 {
    /// <summary>
    ///     ILGenerator Object
    /// </summary>
    private ILGenerator ILout;
    /// <summary>
    ///    Auto (Local) Variable support
    ///    Stores the index return by DefineLocal
    ///    method of MethodBuilder
    /// </summary>
    private ArrayList variables = new ArrayList();
    /// <summary>
    ///    Symbol Table for storing Types and
    ///    doing the type analysis
    /// </summary>
    SymbolTable m_tab = new SymbolTable();
    /// <summary>
    ///    CLR Reflection.Emit.MethodBuilder
    /// </summary>
    MethodBuilder _methinfo = null;
    /// <summary>
    ///     CLR Type Builder ( useful for creating
    ///     classes in the run time
    /// </summary>
    TypeBuilder _bld=null;

    /// <summary>
    ///    Procedure to compiled
    /// </summary>
```

```
Procedure _proc = null;
/// <summary>
///    Program to be compiled...
///
/// </summary>
TModule _program;

/// <summary>
///
/// </summary>
/// <param name="prog"></param>
/// <param name="p"></param>
/// <param name="bld"></param>
public DNET_EXECUTABLE_GENERATION_CONTEXT(TModule  program ,
                                Procedure proc,
                                TypeBuilder bld)
{
   // All the code in the Source module is compiled
   // into this procedure
   _proc = proc;
   //
   //  TModule Object
   //
   _program = program;
   //
   // Handle to the type (MainClass )
   //
   _bld = bld;
   //
   //  The method does not take any procedure
   System.Type[] s = null;
   // Return type is void
   System.Type ret_type = null;
   //
   //  public static void Main()
   //
   _methinfo = _bld.DefineMethod("Main",
      MethodAttributes.Public | MethodAttributes.Static,
      ret_type, s);
   //
   // We have created the Method Prologue
   // Get the handle to the code generator
   //
   //
   ILout = _methinfo.GetILGenerator();

}

/// <summary>
///
/// </summary>
public string MethodName
{
   get
   {
      return _proc.Name ;
   }
}
```

```
    /// <summary>
    ///
    /// </summary>
    public MethodBuilder MethodHandle
    {

       get
       {
          return _methinfo;
       }


    }

    /// <summary>
    ///
    /// </summary>
    public TypeBuilder TYPEBUILDER
    {

       get
       {

          return _bld;
       }

    }

    /// <summary>
    ///
    /// </summary>
    public SymbolTable TABLE
    {

       get
       {
          return m_tab;
       }

    }
    /// <summary>
    ///
    /// </summary>
    public ILGenerator CodeOutput
    {
       get
       {
          return ILout;
       }

    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="type"></param>
    /// <returns></returns>
    public int DeclareLocal(System.Type type)
```

```
    {
        // It is possible to create Local ( auto )
        // Variables by Calling DeclareLocl method
        // of ILGenerator... this returns an integer
        // We store this integer value in the variables
        // collection...
        LocalBuilder lb = ILout.DeclareLocal(type);
        // Now add the integer value associated with the
        // variable to variables collection...
        return variables.Add(lb);
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="s"></param>
    /// <returns></returns>
    public LocalBuilder GetLocal(int s)
    {

        return variables[s] as LocalBuilder;

    }



}
```

COMPILING EXPRESSIONS

To support compilation into .NET IL , we have an addional method appropriately named Compile to the Exp base class.

```
/// <summary>
  ///     In  this Step , we add two more methods to the Exp class
  ///     TypeCheck => To do Type analysis
  ///     get_type  => Type of this node
  /// </summary>
  public abstract class Exp
  {
    public abstract SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont);
    public abstract TYPE_INFO TypeCheck(COMPILATION_CONTEXT cont);
    public abstract TYPE_INFO get_type();
    /// <summary>
    ///   Added in the STEP 5 for .NET IL code generation
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>
    public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);
  }
```

This additional method takes a DNET_EXECUTABLE_GENERATION_CONTEXT as the parameter.

To generate Code for Loading a numeric ....this what we one has to do ....

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
       //
       // Emit LDC_R8 => Load Constant Real 8
       // IEEE 754 floating Point
       //
       // cont.CodeOutput will return ILGenerator of the
       // current method...
       cont.CodeOutput.Emit(OpCodes.Ldc_R8, info.dbl_val);
       return true;
    }
```

The code emitted for a BooleanConstant

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
       //
       // Retrieve the IL Code generator and Emit
       //    LDC_I4 => Load Constant Integer 4
       // We are planning to use a 32 bit long for Boolean
       // True or False
       cont.CodeOutput.Emit(OpCodes.Ldc_I4, (info.bol_val) ? 1 : 0);
       return true;
    }
```

The code emitted for a StringLiteral

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
       //
       //
       // For string emit
       //     LDSTR => Load String
       //
       cont.CodeOutput.Emit(OpCodes.Ldstr , info.str_val);
       return true;

    }
```

The Code emitted for loading a variable reference on to the stack

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
       //
       // Retrieve the Symbol information from the
       // Symbol Table. Symbol name is the key here..
       //
```

```
        SYMBOL_INFO info = cont.TABLE.Get(m_name);
        //
        // Give the Position to retrieve the Local Variable
        // Builder.
        //
        LocalBuilder lb = cont.GetLocal(info.loc_position);
        //
        // LDLOC => Load Local... we need to give
        // a Local Builder as parameter
        //
        cont.CodeOutput.Emit(OpCodes.Ldloc, lb);
        return true;
    }
```

The Code emitted for BinaryPlus (+) needs to query the operand to generate the appropriate code. If the Left and the right expression is numeric... it needs to Generate Add opcode. There is no appropriate operator for String Concatenation. One needs to use String.Concat method to do that....

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {

        // Compile the Left Expression
        exp1.Compile(cont);
        //
        // Compile the Right Expression
        exp2.Compile(cont);
        //
        // Emit Add instruction
        //
        if (_type == TYPE_INFO.TYPE_NUMERIC)
        {
            cont.CodeOutput.Emit(OpCodes.Add);
        }
        else
        {
            // This is a string type..we need to call
            // Concat method..

            Type[] str2 = {
                            typeof(string),
                            typeof(string)
                          };

            cont.CodeOutput.Emit(OpCodes.Call,
               typeof(String).GetMethod("Concat", str2));

         }
        return true;
    }

  }
```

For this routine to work correctly , one needs to call TypeCheck every time to update the type information on all nodes.. This polymorphic behavior forced us to call TypeCheck every time we finished parsing an expression.

```
private Stmt ParsePrintStatement(ProcedureBuilder ctx)
      {
         GetNext();
         Exp a = Expr(ctx);
         //
         // Do the type analysis ...
         //
         a.TypeCheck(ctx.Context);
         if (Current_Token != TOKEN.TOK_SEMI)
         {
            throw new Exception("; is expected");
         }
         return new PrintStatement(a);
      }
```

The code generation of Subtract Operation is as follows

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
     {
        exp1.Compile(cont);
        exp2.Compile(cont);
        cont.CodeOutput.Emit(OpCodes.Sub);
        return true;
     }
```

The code emitted for UnaryMinus is as given below

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
     {
        // Compile the expression
        exp1.Compile(cont);
        //
        // Negate the value on the top of the
        // stack
        //
        cont.CodeOutput.Emit(OpCodes.Neg);
        return true;
     }
```

COMPILING STATEMENTS

To support the Compilation of statement into IL an additional method has been added to the Stmt base class

```
public abstract class Stmt
{
      public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont);
      //
      // Added in the Step 5 for .net IL compilation
       public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);
}
```

The Code emitted for PrintStatement is given below

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
      //
      //  Compile the Expression
      //  The Output will be on the top of stack
      exp1.Compile(cont);
      //
      // Generate Code to Call Console.Write
      //
      System.Type typ = Type.GetType("System.Console");
      Type[] Parameters = new Type[1];

      TYPE_INFO tdata = exp1.get_type();

      if (tdata == TYPE_INFO.TYPE_STRING)
         Parameters[0] = typeof(string);
      else if (tdata == TYPE_INFO.TYPE_NUMERIC)
         Parameters[0] = typeof(double);
      else
         Parameters[0] = typeof(bool);
      cont.CodeOutput.Emit(OpCodes.Call, typ.GetMethod("Write", Parameters));
      return true;
    }
```

The Code emitted for VariableDeclStatement is as follows

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
      //
      // Retrieve the type from the SYMBOL_INFO
      //
      System.Type type = (m_inf.Type == TYPE_INFO.TYPE_BOOL) ?
         typeof(bool) : (m_inf.Type == TYPE_INFO.TYPE_NUMERIC) ?
         typeof(double) : typeof(string);
      //
      //  Get the offset of the variable
      //
      int s = cont.DeclareLocal(type);
      // Store the offset in the SYMBOL_INFO
      //
      m_inf.loc_position = s;
      //
      // Add the variable into Symbol Table..
      //
      cont.TABLE.Add(m_inf);

      return true;
    }
```

The code for the Assignment statement is as given below

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
      if (!exp1.Compile(cont))
      {
        throw new Exception("Compilation in error string");
      }
      SYMBOL_INFO info = cont.TABLE.Get(variable.Name);
      LocalBuilder lb = cont.GetLocal(info.loc_position);
      cont.CodeOutput.Emit(OpCodes.Stloc, lb);
      return true;
    }
```

To support additional language constructs , we had two additional base classes in this project.

The class Procedure is to model a FUNCTION. At this point we compile all the statement into a Function. In future , we will support User defined Function.

```
/// <summary>
  ///   Abstract base class for Procedure
  ///   All the statements in a Program ( Compilation unit )
  ///   will be compiled into a PROC
  /// </summary>
  public abstract class PROC
  {
    public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont);
    public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);


  }
```

The Concrete class Procedure implements in this interface.

```
public class Procedure : PROC
  {
    /// <summary>
    ///   Procedure name ..which defaults to Main
    ///   in the type MainClass
    /// </summary>
    public string m_name;
    /// <summary>
    ///   Formal parameters...
    /// </summary>
    public ArrayList m_formals = null;
    /// <summary>
    ///    List of statements which comprises the Procedure
    /// </summary>
    public ArrayList m_statements = null;
    /// <summary>
    ///    Local variables
    /// </summary>
    public SymbolTable m_locals = null;
    /// <summary>
```

```
        ///         return_value.... a hard coded zero at this
        ///         point of time..
        /// </summary>
        public SYMBOL_INFO return_value = null;
        /// <summary>
        ///       TYPE_INFO => TYPE_NUMERIC
        /// </summary>
        public TYPE_INFO _type = TYPE_INFO.TYPE_ILLEGAL;

    // ------------------ Rest of the Code...
}
```

Let us look at the Compile method of the Procedure class. Procedure is nothing but a collection of staements.

```
public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
        {


            foreach (Stmt e1 in m_statements)
            {
                e1.Compile(cont);
            }

            cont.CodeOutput.Emit(OpCodes.Ret);
            return true;

        }
```

The Execute method of the Procedure is very similar to Compile method. Instead of Calling Compile method , Execute will delegate the call to Execute mathod of statement.

```
public override SYMBOL_INFO Execute(RUNTIME_CONTEXT cont)
        {

            foreach (Stmt stmt in m_statements)
                    stmt.Execute(cont);

            return null;

        }
```

Another interface defined is to support the concept of a module. Module (at this point of time) is just collection of statements. The collection of statements will be compiled into a Procedure. And the Procedure will embedded in a module.

When we support user defined Functions , the meaning of module will be changed a bit. Then the Module will become collection of Functions. All the Functions will be compiled as static methods of a single type. One can easily extend the compiler to support classes.

```
/// <summary>
///    A bunch of statement is called a Compilation
///    unit at this point of time... STEP 5
///    In future , a Collection of Procedures will be
///    called a Compilation unit
///
///    Added in the STEP 5
/// </summary>
public abstract class CompilationUnit
{
   public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont);
   public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);
}
```

Since there is a preexisting type called Module available in the .net class hierarchy, we will use the name Tmodule instead.

```
/// <summary>
          ///     A CodeModule is a Compilation Unit ..
///     At this point of time ..it is just a bunch
///     of statements...
          /// </summary>
public class TModule : CompilationUnit
{
   /// <summary>
   ///    A Program is a collection of Procedures...
   ///    Now , we support only global function...
   /// </summary>
   private ArrayList m_procs=null;
   /// <summary>
   ///    List of Compiled Procedures....
   ///    At this point of time..only one procedure
   ///    will be there....
   /// </summary>
   private ArrayList compiled_procs = null;
   /// <summary>
   ///    class to generate IL executable...
   /// </summary>

   private ExeGenerator _exe = null;

   /// <summary>
   ///    Ctor for the Program ...
   /// </summary>
   /// <param name="procedures"></param>

   public TModule(ArrayList procs)
   {
      m_procs = procs;

   }

   /// <summary>
   ///
   /// </summary>
   /// <param name="name"></param>
   /// <returns></returns>
```

```
    public bool CreateExecutable(string name)
    {
        //
        // Create an instance of Exe Generator
        // ExeGenerator takes a TModule and
        // exe name as the Parameter...
        _exe = new ExeGenerator(this,name);
        // Compile The module...
        Compile(null);
        // Save the Executable...
        _exe.Save();
        return true;
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>
    public override bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
        compiled_procs = new ArrayList();
        foreach (Procedure p in m_procs)
        {
            DNET_EXECUTABLE_GENERATION_CONTEXT con = new
DNET_EXECUTABLE_GENERATION_CONTEXT(this,p, _exe.type_bulder);
            compiled_procs.Add(con);
            p.Compile(con);

        }
        return true;

    }

    public override SYMBOL_INFO Execute(RUNTIME_CONTEXT cont)
    {
        Procedure p = Find("Main");

        if (p != null)
        {

            return p.Execute(cont);
        }

        return null;

    }

    public MethodBuilder _get_entry_point(string _funcname)
    {
        foreach (DNET_EXECUTABLE_GENERATION_CONTEXT u in compiled_procs)
        {
            if (u.MethodName.Equals(_funcname))
            {
                return u.MethodHandle;
            }

        }
```

```
        return null;


    }

    public Procedure Find(string str)
    {
        foreach (Procedure p in m_procs)
        {
            string pname = p.Name;

            if (pname.ToUpper().CompareTo(str.ToUpper()) == 0)
                return p;

        }

        return null;

    }


}
```

RETURN OF THE BUILDERS

To create Tmodule and Procedure Object , we need to Parse the statemeents. We can create the Objects only when we have finished the Parsing. We need to accumulate the requisite Objects before we create Procedure. Builder Pattern is a nice way to organize the code to do these kind of stuff.

```
/// <summary>
    ///    Base class for all the Builder
    ///
    ///
    ///
    ///          AbstractBuilder
    ///                  TModuleBuilder
    ///                  ProcedureBuilder
    /// </summary>
    public class AbstractBuilder
    {

    }
```

The Source code of the Module Builder is given below

```
/// <summary>
    ///      A Builder for Creating a Module
    /// </summary>
    class TModuleBuilder : AbstractBuilder
    {
        /// <summary>
        ///     Array of Procs
        /// </summary>
        private ArrayList procs;
```

```
    /// <summary>
    ///    Array of Function Prototypes
    ///    not much use as of now...
    /// </summary>
    private ArrayList protos=null;

    /// <summary>
    ///     Ctor does not do much
    /// </summary>
    public TModuleBuilder()
    {
       procs = new ArrayList();
       protos = null;
    }

    /// <summary>
    ///     Add Procedure
    /// </summary>
    /// <param name="p"></param>
    /// <returns></returns>
    public bool Add(Procedure p)
    {
       procs.Add(p);
       return true;
    }

    /// <summary>
    ///      Create Program
    /// </summary>
    /// <returns></returns>
    public TModule GetProgram()
    {
       return new TModule(procs);
    }

    ///
    ///
    ///
    public Procedure GetProc(string name)
    {
       foreach (Procedure p in procs)
       {
          if (p.Name.Equals(name))
          {
             return p;
          }

       }

       return null;

    }

}
```

The source code of the ProcedureBuilder is given below...

```
/// <summary>
  ///
  /// </summary>
  public class ProcedureBuilder : AbstractBuilder
  {
     /// <summary>
     ///    Procedure name ..now it is hard coded
     ///    to MAIN
     /// </summary>
     private string proc_name = "";
     /// <summary>
     ///    Compilation context for type analysis
     /// </summary>
     COMPILATION_CONTEXT ctx = null;
     /// <summary>
     ///    Procedure does not take any argument..
     /// </summary>
     ArrayList m_formals = null;
     /// <summary>
     ///    Array of Statements
     /// </summary>
     ArrayList m_stmts = new ArrayList();
     /// <summary>
     ///    Return Type of the procedure
     /// </summary>
     TYPE_INFO inf = TYPE_INFO.TYPE_ILLEGAL;
     /// <summary>
     ///
     /// </summary>
     /// <param name="name"></param>
     /// <param name="_ctx"></param>

     public ProcedureBuilder(string name, COMPILATION_CONTEXT _ctx)
     {
        ctx = _ctx;
        proc_name = name;
     }


     /// <summary>
     ///
     /// </summary>
     /// <param name="info"></param>
     /// <returns></returns>
     public bool AddLocal(SYMBOL_INFO info)
     {
        ctx.TABLE.Add(info);
        return true;
     }
     /// <summary>
     ///
     /// </summary>
     /// <param name="e"></param>
     /// <returns></returns>
     public TYPE_INFO TypeCheck(Exp e)
```

```
        {
            return e.TypeCheck(ctx);
        }
        /// <summary>
        ///
        /// </summary>
        /// <param name="st"></param>
        public void AddStatement(Stmt st)
        {
            m_stmts.Add(st);
        }
        /// <summary>
        ///
        /// </summary>
        /// <param name="strname"></param>
        /// <returns></returns>
        public SYMBOL_INFO GetSymbol(string strname)
        {

            return ctx.TABLE.Get(strname);

        }

        /// <summary>
        ///   Check the function Prototype
        /// </summary>
        /// <param name="name"></param>
        /// <returns></returns>
        public bool CheckProto(string name)
        {
            return true;

        }
        /// <summary>
        ///
        /// </summary>
        public TYPE_INFO TYPE
        {
            get
            {
                return inf;
            }

            set
            {
                inf = value;

            }


        }
        /// <summary>
        ///
        /// </summary>
        public SymbolTable TABLE
        {
            get
            {
```

```
            return ctx.TABLE;
        }
    }

    public COMPILATION_CONTEXT Context
    {
        get
        {
            return ctx;
        }
    }
    /// <summary>
    ///
    /// </summary>
    public string Name
    {
        get
        {
            return proc_name;
        }

        set
        {
            proc_name = value;

        }

    }
    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public Procedure GetProcedure()
    {
        Procedure ret = new Procedure(proc_name,
              m_stmts, ctx.TABLE, inf);

        return ret;
    }




}
```

The Parse Process in the RDParser modules starts from the DoParse Method

```
/// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public TModule DoParse()
    {
        ProcedureBuilder p = new ProcedureBuilder("MAIN", new COMPILATION_CONTEXT());
        ArrayList stmts = Parse(p);
```

```
        foreach (Stmt s in stmts)
        {
           p.AddStatement(s);
        }

        Procedure pc =  p.GetProcedure();

        prog.Add(pc);
        return prog.GetProgram();

    }
```

In this version all the methods in the Parser takes a ProcedureBuilder object as parameter...

```
  private ArrayList StatementList(ProcedureBuilder ctx)
    {
        ArrayList arr = new ArrayList();
        while (Current_Token != TOKEN.TOK_NULL)
        {
           Stmt temp = Statement(ctx);
           if (temp != null)
           {
              arr.Add(temp);
           }
        }
        return arr;
    }
```

In the previous step , these routines were taking a COMPILATION_CONTEXT as paramter. In this step , this object is encapsulated in ProcedureBuilder object.

Now onwards , we will have three projects in the solution

SLANG_DOT_NET => Compilation Engine
SLANGCOMPILE   => Calls the Compilation Engine to generate .NET executable
SLANGINTERPRET => Calls the Compilation Engine to interpret the statements

Let us take a look at the source code of SLANGCOMPILE

```
using System;
using System.Collections;
using System.Linq;
using System.Text;
using System.IO;
using SLANG_DOT_NET;
namespace SLANGCOMPILE
{
   class Caller
   {

      /// <summary>
      ///    Driver routine to call the program script
```

```
        /// </summary>
        static void TestFileScript(string filename)
        {

            if (filename == null)
                return;


            // ---------- Read the contents from the file

            StreamReader sr = new StreamReader(filename);
            string programs2 = sr.ReadToEnd();
            sr.Close();
            sr.Dispose();

            //---------- Creates the Parser Object
            // With Program text as argument
            RDParser pars = null;
            pars = new RDParser(programs2);
            TModule p = null;
            p = pars.DoParse();

            //
            //  Now that Parse is Successul...
            //  Create an Executable...!
            //
            if (p.CreateExecutable("First.exe"))
            {
                Console.WriteLine("Creation of Executable is successul");
                return;
            }



        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {


            if (args == null ||
                 args.Length != 1)
            {
                Console.WriteLine("SLANGCOMPILE <scriptname>\n");
                return;

            }
            TestFileScript(args[0]);
            //---------- Wait for the Key Press
            Console.Read();

        }
    }
}
```

The source code of the SLANGINTERPRET is as given below

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
using System.IO;
using SLANG_DOT_NET;

namespace CallSLANG
{
   class Caller
   {

      /// <summary>
      ///    Driver routine to call the program script
      /// </summary>
      static void TestFileScript(string filename)
      {

         if (filename == null)
               return;


         // ---------- Read the contents from the file

         StreamReader sr = new StreamReader(filename);
         string programs2 = sr.ReadToEnd();


         //---------- Creates the Parser Object
         // With Program text as argument
         RDParser pars = null;
         pars = new RDParser(programs2);
         TModule p = null;
         p = pars.DoParse();

         //
         //  Now that Parse is Successul...
         //  Do a recursive interpretation...!
         //
         RUNTIME_CONTEXT f = new RUNTIME_CONTEXT(p);
         SYMBOL_INFO fp = p.Execute(f);

      }

      /// <summary>
      ///
      /// </summary>
      /// <param name="args"></param>
      static void Main(string[] args)
      {


         if (args == null ||
             args.Length != 1)
```

```
        {
            Console.WriteLine("SLANGINTERPRET <scriptname>\n");
            return;

        }
        TestFileScript(args[0]);
        //------------- Wait for the Key Press
        Console.Read();

    }
  }
}
```

Let us how our compiler has generated code for some script....

```
////////////////////////////
//  Mul.sl
//
//
//

PRINTLINE  2*(3+4);
```

Using the ILDASM utility , here is what we get from our compiler...

```
.method public static void  Main() cil managed
{
  .entrypoint
  // Code size       35 (0x23)
  .maxstack  3
  IL_0000:  ldc.r8     2.
  IL_0009:  ldc.r8     3.
  IL_0012:  ldc.r8     4.
  IL_001b:  add
  IL_001c:  mul
  IL_001d:  call       void [mscorlib]System.Console::WriteLine(float64)
  IL_0022:  ret
} // end of method MainClass::Main
```

The next script will check string Concatentaion

```
PRINT "Hello World" + " SL ";
```

The output generated by our compiler was as follows

```
.method public static void  Main() cil managed
```

```
{
  .entrypoint
  // Code size       21 (0x15)
  .maxstack  2
  IL_0000:  ldstr      "Hello World"
  IL_0005:  ldstr      " SL "
  IL_000a:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_000f:  call       void [mscorlib]System.Console::Write(string)
  IL_0014:  ret
} // end of method MainClass::Main
```

The next script tests the working of unary expressions.

```
//////////////////////
//
//
//  Third.sl
//
//


Numeric a;

String b;


a = ---1;

PrintLine a*4 + 10;
```

The code generated by the compiler is as follows

```
.method public static void  Main() cil managed
{
  .entrypoint
  // Code size       40 (0x28)
  .maxstack  2
  .locals init (float64 V_0,
           string V_1)
  IL_0000:  ldc.r8     1.
  IL_0009:  neg
  IL_000a:  neg
  IL_000b:  neg
  IL_000c:  stloc.0
  IL_000d:  ldloc.0
  IL_000e:  ldc.r8     4.
  IL_0017:  mul
  IL_0018:  ldc.r8     10.
  IL_0021:  add
  IL_0022:  call       void [mscorlib]System.Console::WriteLine(float64)
```

```
  IL_0027:  ret
} // end of method MainClass::Main
```

This script test allmost all the features of the compiler (implemented at this point of time)

```
////////////////////////////////////
//
// A simple SLANG script
//
//    Sample #1
//

NUMERIC a;   // Declare a Numeric variable

a = 2*3+5* 30 + -(4*5+3);  // Assign

PRINTLINE a;  // Dump a



//----- String concatenation

PRINT "Hello " + "World";


//--------------- Write a new line

PRINTLINE "";


//--------------- string data type
STRING c;

c = "Hello ";  // assignment to string

//----------------- assignment and concatenation

C = C + "World";

PRINTLINE c;

//--------------- boolean variable

BOOLEAN d;

d= TRUE;

PRINTLINE d;

d= FALSE;

PRINTLINE d;
```

The output is

```
.method public static void  Main() cil managed
{
  .entrypoint
  // Code size       156 (0x9c)
  .maxstack  4
  .locals init (float64 V_0,
           string V_1,
           bool V_2)
  IL_0000:  ldc.r8     2.
  IL_0009:  ldc.r8     3.
  IL_0012:  mul
  IL_0013:  ldc.r8     5.
  IL_001c:  ldc.r8     30.
  IL_0025:  mul
  IL_0026:  ldc.r8     4.
  IL_002f:  ldc.r8     5.
  IL_0038:  mul
  IL_0039:  ldc.r8     3.
  IL_0042:  add
  IL_0043:  neg
  IL_0044:  add
  IL_0045:  add
  IL_0046:  stloc.0
  IL_0047:  ldloc.0
  IL_0048:  call       void [mscorlib]System.Console::WriteLine(float64)
  IL_004d:  ldstr      "Hello "
  IL_0052:  ldstr      "world"
  IL_0057:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_005c:  call       void [mscorlib]System.Console::Write(string)
  IL_0061:  ldstr      ""
  IL_0066:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_006b:  ldstr      "Hello "
  IL_0070:  stloc.1
  IL_0071:  ldloc.1
  IL_0072:  ldstr      "world"
  IL_0077:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_007c:  stloc.1
  IL_007d:  ldloc.1
  IL_007e:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0083:  ldc.i4     0x1
  IL_0088:  stloc.2
```

```
  IL_0089:  ldloc.2
  IL_008a:  call       void [mscorlib]System.Console::WriteLine(bool)
  IL_008f:  ldc.i4     0x0
  IL_0094:  stloc.2
  IL_0095:  ldloc.2
  IL_0096:  call       void [mscorlib]System.Console::WriteLine(bool)
  IL_009b:  ret
} // end of method MainClass::Main
```