What is an Expression ?

Expression is something which one evaluates for it's value

```
/// <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);
    }
```

What is a statement ?

.Statement is what one executes for it's effect. A statement mutates the STATE of variables.

```
 /// <summary>
///
///          Abstract base Class for Statement
///     ( one can use an interface as well !)
///
/// </summary>
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);

}
```

What is a Procedure/Function ?

Procedure is a collection of Statements which will be executed. Procedures are referred as a single entity

```
public abstract class PROC
  {
    //
    //public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont);
    // The above stuff is extended with Formal parameter list
    // addition in STEP 7
    public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont, ArrayList formals);

    public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);

  }
```

What is a Module ? (Program)

Module or a Compilation unit is a collection of procedures which are interrelated (most often in a single file) and Execution will start from a known entry point (MAIN in our case)

```
public abstract class CompilationUnit
  {
    //public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont);
    //Extended with Formal Parameter list given to Main
    //Addition in STEP 7
    public abstract SYMBOL_INFO Execute(RUNTIME_CONTEXT cont, ArrayList actuals);

    public abstract bool Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont);
  }
```

```
////////////////////////////////////////////////////
//
//  This software is released as per the clauses of MIT License
//
//
//  The MIT License
//
//  Copyright (c) 2010, Praseed Pai K.T.
//                  http://praseedp.blogspot.com
//                  praseedp@yahoo.com
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF
ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN
// THE SOFTWARE.
//
//
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; }
  }
}
```

```
////////////////////////////////////////////////////
//
//  This software is released as per the clauses of MIT License
//
//
//  The MIT License
//
//  Copyright (c) 2010, Praseed Pai K.T.
//                  http://praseedp.blogspot.com
//                  praseedp@yahoo.com
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF
ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN
// THE SOFTWARE.
//
//
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;

        }




    }
}
```

Calling Routine

```
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();
      }
   }
}
```

CODE GENERATION

For BooleanConstant

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

            return info;
}




public override bool
    Compile( DNET_COMPILATION_CONTEXT cont) {
     cont.CodeOutput.Emit(OpCodes.Ldc_I4,
    (   info.bol_val ) ? 1 : 0 );

 return true;
}
```

For String Literal

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

        return info;
}



public override bool
    Compile( DNET_COMPILATION_CONTEXT cont )
{

  cont.CodeOutput.Emit(OpCodes.Ldc_R8,info.dbl_val);

  return true;
}
```

For StringLiteral

```
public override bool Compile( DNET_COMPILATION_CONTEXT cont )
{
  cont.CodeOutput.Emit(OpCodes.Ldstr,info.str_val);
  return true;
}


public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
{

        return info;
}
```

For Variable Access

```
public override bool Compile( DNET_COMPILATION_CONTEXT cont )
{
    SYMBOL_INFO info =  cont.TABLE.Get(m_name);

       LocalBuilder lb = cont.GetLocal(info.loc_position);

    cont.CodeOutput.Emit( OpCodes.Ldloc ,lb);

    return true;
}
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 ;
    }
}
```

For BinaryPlus

```
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 UnaryMinus

```
 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;
      }
```

Compilation of Function Call

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

         if (m_proc == null)
         {
            // if it is  a recursive call..
            // resolve the address...
            m_proc = cont.GetProgram().Find(_procname);
         }

         string name = m_proc.Name;


         TModule str = cont.GetProgram();
         MethodBuilder bld = str._get_entry_point(name);

         foreach (Exp ex in m_actuals)
         {
            ex.Compile(cont);
         }
         cont.CodeOutput.Emit(OpCodes.Call, bld);
         return true;
      }
```

Interpretation of Function Call

```
public override SYMBOL_INFO Evaluate(RUNTIME_CONTEXT cont)
    {
      if (m_proc != null)
      {
        //
        // This is a Ordinary Function Call
        //
        //
        RUNTIME_CONTEXT ctx = new
RUNTIME_CONTEXT(cont.GetProgram());

        ArrayList lst = new ArrayList();

        foreach (Exp ex in m_actuals)
        {
          lst.Add(ex.Evaluate(cont));
        }

        return m_proc.Execute(ctx, lst);

      }
      else
      {
        // Recursive function call...by the time we
        // reach here..whole program has already been
        // parsed. Lookup the Function name table and
        // resolve the Address
        //
        //
        m_proc = cont.GetProgram().Find(_procname);
        RUNTIME_CONTEXT ctx = new
RUNTIME_CONTEXT(cont.GetProgram());
        ArrayList lst = new ArrayList();

        foreach (Exp ex in m_actuals)
        {
          lst.Add(ex.Evaluate(cont));
```

```
            }

            return m_proc.Execute(ctx, lst);


        }
    }
```

Compilation of WhileStatement

```
 public override bool
Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
        {
           Label true_label, false_label;
           true_label = cont.CodeOutput.DefineLabel();
           false_label = cont.CodeOutput.DefineLabel();
           cont.CodeOutput.MarkLabel(true_label);
           cond.Compile(cont);
           cont.CodeOutput.Emit(OpCodes.Ldc_I4, 1);
           cont.CodeOutput.Emit(OpCodes.Ceq);
           cont.CodeOutput.Emit(OpCodes.Brfalse, false_label);

           foreach (Stmt rst in stmnts)
           {
              rst.Compile(cont);
           }

           cont.CodeOutput.Emit(OpCodes.Br, true_label);
           cont.CodeOutput.MarkLabel(false_label);
           return true;

        }
```

Interpretation of WhileStatement

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

      Test:

         SYMBOL_INFO m_cond = cond.Evaluate(cont);


         if (m_cond == null || m_cond.Type != TYPE_INFO.TYPE_BOOL)
            return null;

         if (m_cond.bol_val != true)
            return null;

         SYMBOL_INFO tsp = null;
         foreach (Stmt rst in stmnts)
         {
            tsp = rst.Execute(cont);
            if (tsp != null)
            {
               return tsp;
            }
         }

         goto Test;
      }
```

Comilation of Ifstatement

```
 public override bool
Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
      {
         Label true_label, false_label;
         //
         // Generate Label for True
         true_label = cont.CodeOutput.DefineLabel();
         // Generate Label for False
         false_label = cont.CodeOutput.DefineLabel();
         //
         // Compile the expression
         //
         cond.Compile(cont);
         //
         // Check whether the top of the stack contain
         // 1 ( TRUE)
         cont.CodeOutput.Emit(OpCodes.Ldc_I4, 1);
         cont.CodeOutput.Emit(OpCodes.Ceq);
         //
         //  if False , jump to false_label ...
         //  ie to else part
         cont.CodeOutput.Emit(OpCodes.Brfalse, false_label);

         foreach (Stmt rst in stmnts)
         {
            rst.Compile(cont);
         }
         // Once we have reached here...go
         // to True label...
         cont.CodeOutput.Emit(OpCodes.Br, true_label);
         //
         // Place a Label here...if the condition evaluates
         // to false , jump to this place..
         cont.CodeOutput.MarkLabel(false_label);

         if (else_part != null)
         {
```

```
            foreach (Stmt rst in else_part)
            {
                rst.Compile(cont);

            }
        }
        //
        // Place a label here...to mark the end of the
        // IF statement
        cont.CodeOutput.MarkLabel(true_label);
        return true;
    }
```

Interpretation of IFStatement

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


        //
        //  Evaluate the Condition
        //
        SYMBOL_INFO m_cond = cond.Evaluate(cont);

        //
        // if cond is not boolean..or evaluation failed
        //
        if (m_cond == null ||
            m_cond.Type != TYPE_INFO.TYPE_BOOL)
            return null;


        SYMBOL_INFO ret = null;
        if (m_cond.bol_val == true)
        {
            //
            // if cond is true
            foreach (Stmt rst in stmnts)
            {
                ret = rst.Execute(cont);
                if (ret != null)
                    return ret;
            }
        }
        else if (else_part != null)
        {
            // if cond is false and there is
            // else statement ..!
            foreach (Stmt rst in else_part)
            {
                    ret = rst.Execute(cont);
                    if ( ret != null )
```

```
                return ret;
            }


        }

        return null;


    }
```

Compilation of Procedure Defenition

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

                                if ( m_formals != null )
                                {


            int i=0;

                                                foreach( SYMBOL_INFO b in m_formals )
                                                {

                                                        System.Type type = (b.Type ==
TYPE_INFO.TYPE_BOOL ) ?

                                                                typeof(bool) : (b.Type
== TYPE_INFO.TYPE_NUMERIC ) ?

                                                                typeof(double) :
typeof(string);
                                                        int s = cont.DeclareLocal(type);
                                                        b.loc_position = s;
                                                        cont.TABLE.Add(b);

            cont.CodeOutput.Emit(OpCodes.Ldarg,i);

            cont.CodeOutput.Emit(OpCodes.Stloc,cont.GetLocal(s));
                i++;
                                                }

                                }


          foreach (Stmt e1 in m_statements)
```

```
                {
                        e1.Compile(cont);
                }

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

        }
```

Compilation of LogicalExp

```
public override bool
Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
      {
         ex1.Compile(cont);
         ex2.Compile(cont);
         if (m_op == TOKEN.TOK_AND)
            cont.CodeOutput.Emit(OpCodes.And);
         else if (m_op == TOKEN.TOK_OR)
            cont.CodeOutput.Emit(OpCodes.Or);


         return true;
      }
```

Compilation of RelationalExp

```
/// <summary>
        ///
        /// </summary>
        /// <param name="cont"></param>
        /// <returns></returns>
        private bool
CompileStringRelOp(DNET_EXECUTABLE_GENERATION_CONTEXT
cont)
        {
            //
            // Compile the Left Expression
            ex1.Compile(cont);
            //
            // Compile the Right Expression
            ex2.Compile(cont);

            // This is a string type..we need to call
            // Compare method..

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

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

               if (m_op == RELATION_OPERATOR.TOK_EQ)
               {
                  cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
                  cont.CodeOutput.Emit(OpCodes.Ceq);
               }
               else
               {
                  //
```

```
            // This logic is bit convoluted...
            // String.Compare will give 0 , 1 or -1
            // First we will check whether the stack value
            // is zero..
            // This will put 1 on stack ..if value was zero
            // after string.Compare
            // Once again check against zero ...it is equivalent
            // to negation

            cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
            cont.CodeOutput.Emit(OpCodes.Ceq);
            cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
            cont.CodeOutput.Emit(OpCodes.Ceq);


        }


        return true;
    }


    /// <summary>
    ///     Compile the Relational Expression...
    /// </summary>
    /// <param name="cont"></param>
    /// <returns></returns>
    public override bool
Compile(DNET_EXECUTABLE_GENERATION_CONTEXT cont)
    {
        if (_optype == TYPE_INFO.TYPE_STRING)
        {
            return CompileStringRelOp(cont);
        }

        //
        // Compile the Left Expression
        ex1.Compile(cont);
        //
```

```
// Compile the Right Expression
ex2.Compile(cont);


if (m_op == RELATION_OPERATOR.TOK_EQ)
   cont.CodeOutput.Emit(OpCodes.Ceq);
else if (m_op == RELATION_OPERATOR.TOK_GT)
   cont.CodeOutput.Emit(OpCodes.Cgt);
else if (m_op == RELATION_OPERATOR.TOK_LT)
   cont.CodeOutput.Emit(OpCodes.Clt);
else if (m_op == RELATION_OPERATOR.TOK_NEQ)
{
   // There is no IL instruction for !=
   // We check for the equivality of the
   // top two values on the stack ...
   // This will put 0 ( FALSE ) or 1 (TRUE)
   // on the top of stack...
   // Load zero and check once again
   // Check == once again...

   cont.CodeOutput.Emit(OpCodes.Ceq);
   cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
   cont.CodeOutput.Emit(OpCodes.Ceq);

}
else if (m_op == RELATION_OPERATOR.TOK_GTE)
{

   // There is no IL instruction for >=
   // We check for the <  of the
   // top two values on the stack ...
   // This will put 0 ( FALSE ) or 1 (TRUE)
   // on the top of stack...
   // Load Zero and
   // Check == once again...

   cont.CodeOutput.Emit(OpCodes.Clt);
   cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
   cont.CodeOutput.Emit(OpCodes.Ceq);
```

```
        }
        else if (m_op == RELATION_OPERATOR.TOK_LTE)
        {
            // There is no IL instruction for <=
            // We check for the >  of the
            // top two values on the stack ...
            // This will put 0 ( FALSE ) or 1 (TRUE)
            // on the top of stack...
            // Load Zero and
            // Check == once again...

            cont.CodeOutput.Emit(OpCodes.Cgt);
            cont.CodeOutput.Emit(OpCodes.Ldc_I4, 0);
            cont.CodeOutput.Emit(OpCodes.Ceq);

        }



        return true;

    }

    public override TYPE_INFO get_type()
    {
        return _type;
    }
}
```

Interpretation of Procedure Defenition

```
public override SYMBOL_INFO  Execute(RUNTIME_CONTEXT
cont,ArrayList actuals  )
                        {
                                ArrayList vars = new ArrayList();
                                int i=0;

                                FRAME ft = new FRAME();

                                if ( m_formals != null && actuals !=null )
                                {

                                        i=0;
                                        foreach( SYMBOL_INFO b in m_formals )
                                        {

                                                SYMBOL_INFO inf = actuals[i]
as SYMBOL_INFO;
                                                inf.SymbolName =
b.SymbolName;
                        cont.TABLE.Add(inf);
                                                i++;
                                        }

                                }

        foreach (Stmt e1 in m_statements)
                                {
```

```
                                        return_value = e1.Execute(cont);

                                        if ( return_value != null )
                                                return return_value;

                                }

                                return null;

        }
```