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();

        }
    }
}
```