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();
        }
    }
}
```