Write Your Own .NET Compiler

January 30,2010

Community Tech Days

Kochi

Praseed Pai K T

praseedp@yahoo.com

http://praseedp.blogspot.com

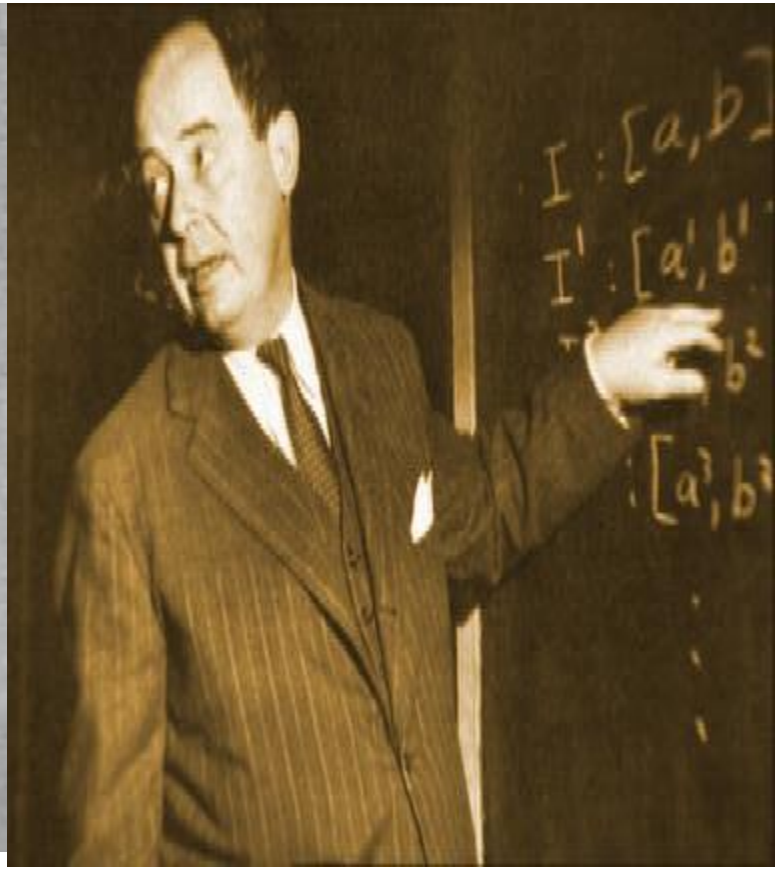
http://slangfordotnet.codeplex.com

A Point to Ponder

"A Science is any discipline in which a fool of this generation can go far beyond the point reached by the genius of earlier generation."

Max Gluckman , South African Anthropolgist

An interesting Tale



Compiler Construction

- Just an exercise in Software Engineering
- Now a days people can write compilers as fast as they can type !!!
- Better Tools and Better People around
- Compiler infrastructure software makes the task easier.

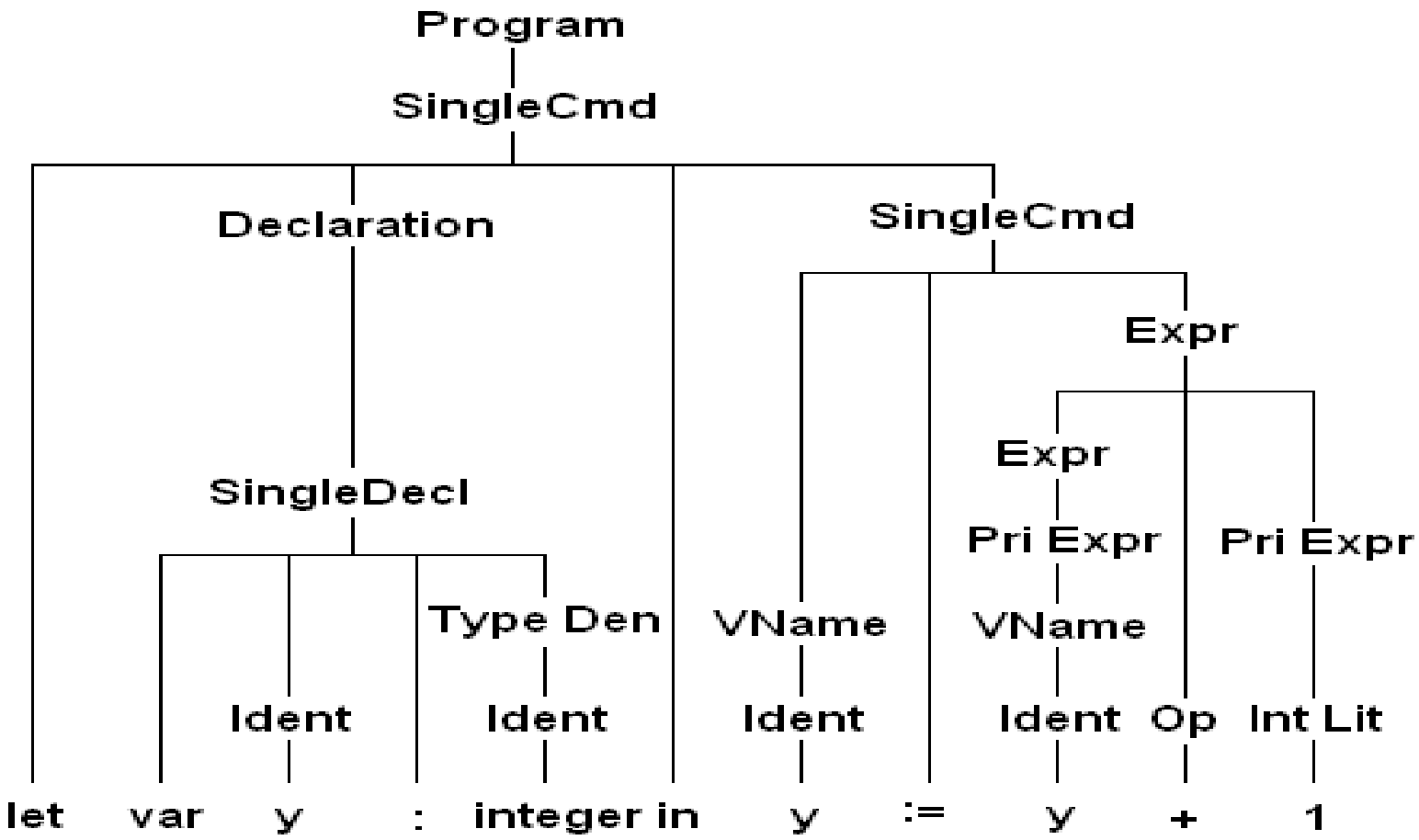
Why .net ?

- .NET Reflection API
- System.Reflection namespace
- System.Reflection.Emit namespace
- .NET contains facilities to compile user code into Assemblies (Assemblies can be DLL or EXE)
- C# contains 60 years of evolution of Programming models.

Computer Program – Can it be treated as data ?

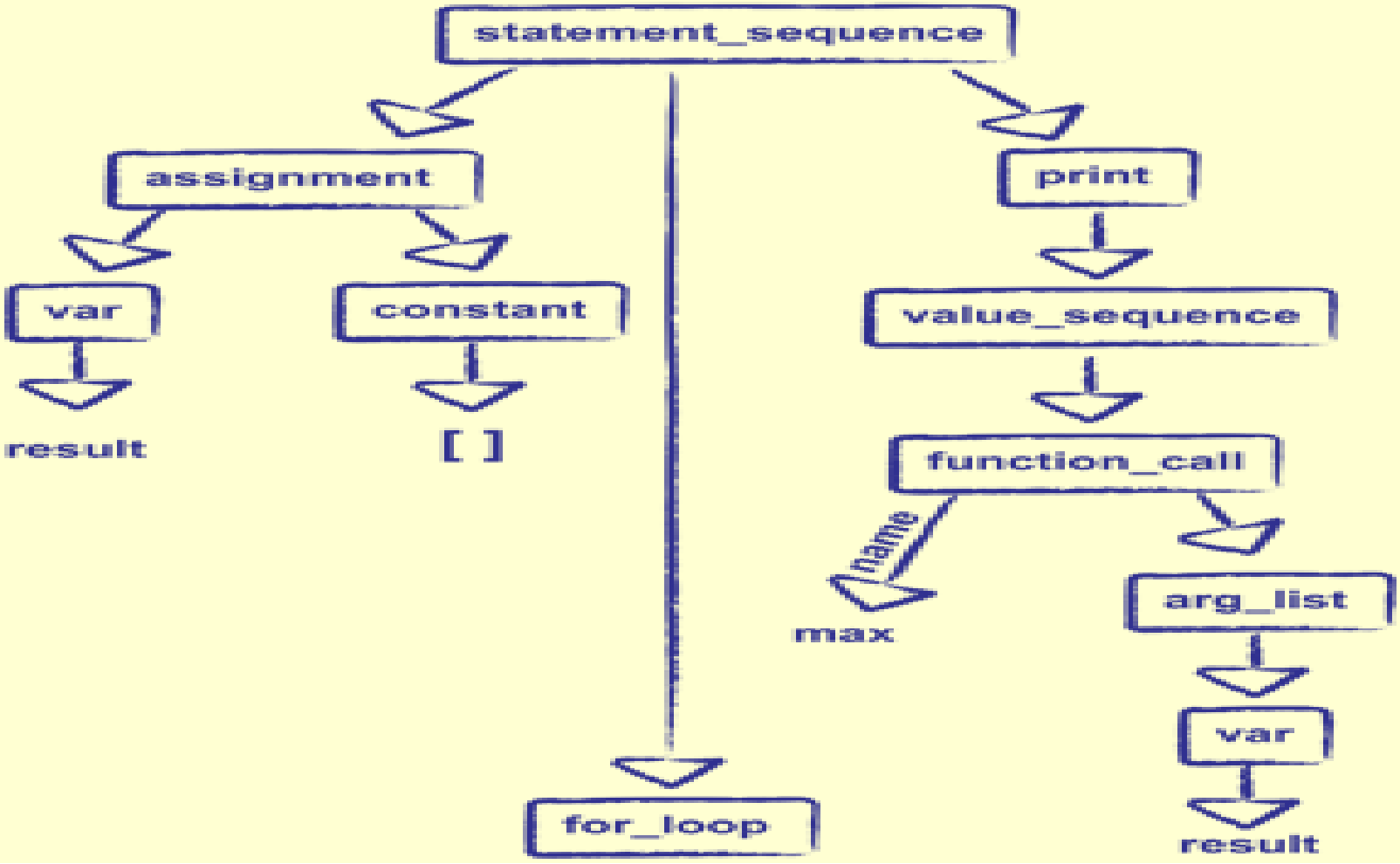
- Interpreter/compiler of a programming language is just another program !!!
- One can create an Object Model for Programming Constructs
- Programming constructs can be Organized as hierarchies
- The Concept of Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST)



AST (contd…)

```
result = [ ]
for val in seq:
        result.append (analyze(vol))
print max (result)
```



ACTORS

FOUR ACTORS IN THE PROCESS

What is an Expression ?

- Expression is something which one evaluates for it’s value

What is a Statement ?

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

What is a Procedure/Function ?

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

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)

Object Model

Exp //---------- base class for expressions

NumericConstant //--- double constant

BinaryExp // + , / , - , *

UnaryExp // + , -

```
abstract class Exp
{
        public abstract double Evaluate(RUNTIME_CONTEXT cont);
 }
```

Object Specification

```
class BinaryExp : Exp    {
        private Exp _ex1, _ex2;
        private OPERATOR _op;
}

class NumericConstant {
        private double _value;
}

class UnaryExp : Exp  {
        private Exp _ex1;
        private OPERATOR _op;
}
```

Operators

```
enum OPERATOR
    {
        ILLEGAL = -1,
        PLUS,
        MINUS,
        DIV,
        MUL
    }
```

Interpretation of NumericConstant

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

Interpretation of BinaryExpr

```
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;

    }
```

Interpretation of UnaryExpr

```
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;

    }
```

Some Examples

A) 10*5

```
Exp e = new BinaryExp(new NumericConstant(5),new
 NumericConstant(10), OPERATOR.MUL);
```

B) - (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);
```

Compiler Construction - stages

- Lexical Analysis
- Parsing
- Creation of Abstract Syntax Tree (AST)
- AST can be interpreted using a Recursive Walk
- AST can be compiled into an instruction set to create executables

Representing Programs as Object is tedious

- From the texual representation of the expression (code) , we need to generate the requisite objects to represent it as a Tree. (AST)
- The idea of Recursive Descent Parsing
- Parser and Lexical Analysis routine work side by side. (Parser on the fly demands the next Token from the Lexical routine)
- As the Parse Progresses we will create AST

Lexical Specifications

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

Lexical Analysis

```
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
   }
}
```

BNF

<Expr> ::= <Term> | Term { + | - } <Expr>

<Term> ::= <Factor> | <Factor> {*|/} <Term>

<Factor>::= <number> | (<expr>) | {+|-} <factor>

Grammar (BNF to psuedo code)

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

     Exp RetValue =  Term();

     if ( Token == TOK_PLUS ||  Token == TOK_SUB    )
     {
        // -- Advance the  input pointer and get the next token
           Exp temp = Expr();    // recurse
           RetValue = new BinaryExpr( RetValue,Temp );
      }
}
```

Grammar (BNF to psuedo code)

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

        if ( Token == TOK_MUL ||  Token == TOK_DIV    )
        {
              // Advance the input pointer
              Exp temp = Term();    // recurse
              RetValue = new BinaryExpr(RetValue,Temp);
        }
        return RetValue;
}
```

Grammar (BNF to pseudo code)

```
//   <Factor> ::=  <TOK_DOUBLE> | ( <expr> ) |
//    { + |- }  <Factor>
Exp Factor() {
        switch(Token)
          case TOK_DOUBLE:
              return new NumericConstant(#);
          case TOK_OPAREN:
               Exp p = Expr();    //recurse
               // check for closing parenthesis and return
               return p;
          case UNARYOP:
               return Factor();   //recurse
           default:
               //Error
        }
```

Generation of Executable Code

- .NET IL
- Recursive Walk of the AST tree created
- Refection.Emit package
- Assembly , AppDomain , TypeBuilder , etc
- APIs for creating Executables

Modelling Programming constructs using OOP

Class Exp

class NumericConstant

class StringLiteral,BooleanConstant

class Variable, BinaryPlus,BinaryMinus,
class Div, Mul,UnaryPlus,UnaryMinus

class RelationExp,LogicalExp,LogicalNot

class CallExp

Modelling Statements…

Class Stmt

- class VariableDeclarationStatement
- class PrintStatement,PrintLineStatement
- class AssignmentStatement
- class WhileStatement
- class IfStatement
- class ReturnStatement

Modeling Procedure

```
abstract class PROC    {
    public abstract SYMBOL_INFO
    Execute(RUNTIME_CONTEXT cont,ArrayList actuals ) ;

    public abstract bool
    Compile(DNET_COMPILATION_CONTEXT cont );
}
class Procedure : PROC          {
            public string m_name;
            public ArrayList m_formals=null;
            public SymbolTable m_locals=null;
            public ArrayList statements=null;
            public SYMBOL_INFO return_value = null;
            public TYPE_INFO _type;
   }
```

Modeling Modules

```
abstract class CompilationUnit {
public abstract SYMBOL_INFO
Execute(RUNTIME_CONTEXT cont ,ArrayList actuals) ;

public abstract bool
Compile(DNET_EXECUTABLE_GENERATION_CONTEXT
cont );
}

class TModule : CompilationUnit     {
private ArrayList m_procs;
     private ArrayList compiled_procs = null;
private ExeGenerator _exe = null;

}
```

Programming Language Specification

get your
GRAMMAR right !

Grammar for the Module and Function

<Module> ::= {<Procedure>}+;

<Procedure>::= FUNCTION <type> func_name ‘(‘ arglist ‘)’
<stmts>
END

<type> := NUMERIC | STRING | BOOLEAN

arglist ::= '(' {} ')' | '(' <type> arg_name [, arglist] ')'

Grammar for Statements…

<stmts> := { stmt }+

{stmt} :=
<vardeclstmt>|<printstmt>|<printlnstmt>|<assignmentstmt>|<callstmt>|<ifstmt>|<whilestmt> |<returnstmt>

<vardeclstmt> ::= <type> var_name;

<printstmt> := PRINT <expr>;

<assignmentstmt>:= <variable> = value;

<ifstmt>::= IF <expr> THEN <stmts> [ELSE <stmts>] ENDIF

<whilestmt>::= WHILE <expr> <stmts> WEND

<returnstmt>:= Return <expr>

Grammar for Expression

<expr> ::= <BExpr>

<BExpr> ::= <LExpr> LOGIC_OP <BExpr>

<LExpr> ::= <RExpr> REL_OP <LExpr>

<RExpr> ::= <Term> ADD_OP <RExpr>

<Term>::= <Factor> MUL_OP <Term>

<Factor> ::= <Numeric> | <String> | TRUE | FALSE | <variable> | ‘(‘ <expr> ‘)’ | {+|-|!}

<Factor> | <callexpr>

<callexpr> ::= funcname ‘(‘ actuals ‘)’

<LOGIC_OP> := '&&' | ‘||’

<REL_OP> := '>' |' < '|' >=' |' <=' |' <>' |' =='

<MUL_OP> := '*' |' /'

<ADD_OP> := '+' |' -'

Parsing Algorithm

- Recursive Descent Parsing
- Alternative is to use ANTLR or some other tool

Design Patterns

- GOF book and the pattern movement
- Use of Builder Pattern to create Procedures and Programs out of textual representation of program (Code).
- ProcedureBuilder and ModuleBuilder co-ordinates with the parser to keep track of the state. At the end of the parse , we create Procedure and Program Objects.

Interpret or Compile ?

- One can interpret the AST by recursive walk
- One can compile the stuff into IL
- Execute method is for Interpretation
- Compile method will compile into IL

Execution/Compilation

BooleanConstant

NumericConstant

StringLiteral

Variable

BinaryPlus

UnaryMinus

CallExp (Function Call)

WhileStatement (While Loop)

IFStatement (IF)

Procedure (Function Defenition)

LogicalExp

RelationalExp

Recursive Procedures

- Being a One Pass Compiler , one has to work around to support recursion
- If a call to a function , which is not available in the list of parsed procedures , we make an assumption that call is a recursive one.
- A strategy with holes at this point of time.
- The call is resolved at the compilation/interpretation time.

Samples

Hello World Program

```
//////////////////////
// Helloworld.sl
// Hello World Program
//
FUNCTION BOOLEAN MAIN()
PRINT "Hello World";
END
```

Loop

```
////////////////////////////
// onetohundred.sl
// Program to Print One To Hundred
// STEP 7 and above
//
FUNCTION BOOLEAN MAIN()
NUMERIC d;
d=0;
While ( d <= 100 )
    PRINTLINE d;
   d = d+1;
 Wend
END
```

Recursive Fibonacci routine

```
////////////////////////////////
//
// Recursive Fibonacci series routine
//
//
FUNCTION NUMERIC FIB( NUMERIC n )
IF ( n <= 1 ) then
  return 1;
ELSE
 RETURN FIB(n-1) + FIB(n-2);
ENDIF
END
```

Caller for Fibnacci series

```
////////////////////////////////////
//
//
// Main routine to call
//
FUNCTION BOOLEAN MAIN()
NUMERIC d;
d=0;
 While ( d <= 10 )
    PRINTLINE FIB(d);
    d = d+1;
  Wend
END
```

Conclusion

- Extend the language to add call by reference , multiple modules
- Adding Object Oriented Programming Constructs
- Using a Compiler infrastructure like LLVM or MS common compiler infrastructure, to target native code and gain good optimization

Thank You

- Q& A
- VISIT http://slangfordotnet.codeplex.com