Implementing Programming Languages Using C# 4.0

By **Christopher Diggins**, 12 Jul 2012

★★★★★ 4.92 (89 votes)

Download source code - 62.8 KB

Introduction

This article introduces the basic concepts of programming language implementation for C# programmers. It is meant to provide a quick overview of the concepts of implementing programming languages using a number of examples including an arithmetic evaluator and a simple JavaScript interpreter.

Accompanying this article is an Open-Source library written in C# 4.0 called Jigsaw. You can download the most up to date version of Jigsaw at code.google.com/p/jigsaw-library/.

Jigsaw comes with an efficient and robust parsing engine as well as a large number of sample grammars and evaluators. The Jigsaw parsing library is an evolution of the parsers used in the Cat language and in the later Heron language.

The Jigsaw library is licensed under the MIT Open-Source license. If you need another license, contact me at cdiggins@gmail.com. If you end up using or abusing Jigsaw, I would love to hear about it!

Built-in .NET Compilers

Before we start, I should point out that if you are looking for an off the shelf interpreter in C#, you should consider the `System.CodeDOM.Compiler` namespace and one or more of the following assemblies:

- Microsoft.CSharp
- Microsoft.Jscript
- Microsoft.VisualBasic

Each of these assemblies provides a class derived from `CodeDomProvider` (e.g., `CSharpCodeProvider`) that you can use to compile an assembly at run-time.

The following function shows how to use a `CodeDomProvider` to generate an assembly dynamically:

Collapse | Copy Code

```
public static Assembly CompileWithProvider(CodeDomProvider
provider, params string[] lines)
{
    var param = new
System.CodeDom.Compiler.CompilerParameters();
    var result = provider.CompileAssemblyFromSource(param,
lines);
    foreach (var e in result.Errors)
        Console.WriteLine("Error occured during
compilation {0}", e.ToString());
    if (result.Errors.Count > 0)
        return null;
    else
        return result.CompiledAssembly;
}
```

See the file *CodeDOMCompilers.cs* for more examples of how to use the built-in .NET compilers.

That said, I'm confident you are here because you want to learn the black arts of implementing programming languages, so continue on brave reader!

Anatomy of a Language Tool

Most language tools follow the same basic architecture:

1. **Tokenization** (optional) – This phase is also known as lexing, lexical analysis, and scanning. During this phase, a string of characters is converted into a string of tokens (also called *lexemes*). This phase is necessary for certain kinds of parsers (e.g., LALR) but not others (e.g., PEG). Parsers (like Jigsaw) without a tokenization phase are called *scanner-less* parsers.
2. **Parser** – Transforms a linear sequence of tokens or characters into a tree structure called a parse tree.
3. **Tree transformer** (optional) – Modifies the parse tree simplifying later steps.
4. **Tree visitor** – Visits each node in the tree and performs some action or creates a new data structure.

What happens during the node visit defines the type of language tool. For example:

- **Interpreter** – Transforms nodes into run-time values or executes a primitive action.
- **Compiler** – Transforms nodes into a machine-executable representation.
- **Pretty printer** – Transforms nodes into a human-readable form such as ASCII or HTML.
- **Translator** – Transforms nodes into a source code representation in a new programming language.

Top News

The One Tip That Will Help You Learn To Code 10x Faster

Get the Insider News free each morning.

Related Articles

Why I use explicit interface implementation as a default implementation technique

Concepts behind the C# 3.0 language

Allow users to select the user interface language in your ASP.NET Web application

DuckTyping: Runtime Dynamic Interface Implementation

Detect a written text's language

- **Type checker**– Nodes are transformed into a representation of the type of the expression. This is a form of *abstract interpretation*.
- **Partial evaluator** – An optimization phase where certain nodes in the tree are replaced by their evaluation (e.g., compile-time expressions).

A useful simplification is this: *language tools manipulate trees*.

Parsing

Parsing, or *syntactic analysis*, tells us whether an input string matches a specific syntactic form (*grammar*) and breaks the input up into a parse tree representing the syntactic components (*terms* or *syntactic phrases*).

The .NET Framework has no tools out of the box for doing parsing but there are a large number of third-party parsing tools to choose from. Some of the more popular tools are ANTLR, YACC, and BISON. These tools are *parser generators*, which means they generate the source code for a parser from a parser definition (the grammar).

One problem with using code generators is that the learning curve is quite steep. They each have their own syntax and rules. I had to implement my own hand-written parser before I could understand how to use these tools. By then it was too late, and I was hooked on writing parsers by hand.

For this article, I am using a C# parsing library I wrote called *Jigsaw*. I chose to use a hand-written parser because it is easier to understand and debug. Jigsaw is a memoizing recursive descent backtracking PEG parser. This is is a mouthful so let's break down what it means:

- A *recursive descent* parser uses a set of mutually recursive procedures to recognize syntactic elements in the language.
- A *back-tracking* parser will try parsing a set of rules in order, when facing a choice, until one succeeds.
- A *memoizing* parser is a parser that caches (*memoizes*) intermediate results of the parsing rule using a lookup table which improves the time complexity of the algorithm. Memoizing PEG parsers are also known as *Packrat parsers*.
- A PEG (*Parsing Expression Grammar*) parser recognizes grammars where each rule corresponds directly to a pattern matching algorithm. A more common alternative to PEG are predictive LR parsers (of which there are many) but they are more complicated.

None of this is particularly important when you first learn about parsing. My experience has been that these kinds of parsers most closely correspond to my intuition of how a parser should work.

Grammars

Abstract Class versus Interface

HowTo: Export C++ classes from a DLL

Cat - A Statically Typed Programming Language Interpreter in C#

Basics of Dataflow Programming in F# and C#

Adding Multilanguage Support to Your Objects

Implement Phonetic ("Sounds-like") Name Searches with Double Metaphone Part V: .NET Implementation

Word stemming for German on .NET Framework

Multilingual Support for Web Applications

The Future of Software Development: CodeDOMs (Part 1)

Polymorphism in C

Irony - .NET Compiler Construction Kit

OrderedDictionary: A generic implementation of IOrderedDictionary

Creating Your Own Freaking Awesome Programming Language

Understanding and Implementing the Iterator Pattern in C# and C++

Convenient wrapper of VBScript.RegExp for VC++

COM Interface Basics

A grammar is a formal definition of the syntax of a language. Writing a grammar is a bit like writing a Regular Expression. The grammar consists of one or more rules defined in terms of other rules using rule operators (also called combinators). Each rule describes a particular syntactic element in the language known as a phrase.

In the Jigsaw library, grammars are expressed as a PEG (Parsing Expression Grammar). In a PEG grammar, each rule defines a parser for a particular syntactic element.

> *Time for some unlearning*: If you have previously learned about Context Free Grammars (CFG), a PEG is distinct in that each rule describes how to match strings, not how to generate them (as is the case with a CFG). Some implications are that PEGs can have zero-width rules and are unambiguous. The difference between a PEG and a CFG is subtle but important.

Each rule is an instance of a class derived from `Rule`. Its role is to recognize a syntactic element in the language (called a *term* or *phrase*). This recognition is done in the function `Match()` which returns a `bool` value indicating whether matching was successful. The `Match` function accepts an instance of a `ParserState` class which holds the input string, a position in the input string, and the parse tree.

You can create instances of rules from static member functions of the `Grammar` class.

Collapse | Copy Code

```
Grammar.MatchString("fox").Match("fox")); //  true
Grammar.MatchString("fox").Match("foxy")); //  true
Grammar.MatchString("fox").Match("flying fox")); //  false
```

You can build compound rules that succeed only if a sequence of child rules match successfully using the plus ("+") operator.

Collapse | Copy Code

```
var rule = Grammar.MatchString("cat") +
Grammar.MatchString("fish")
rule.Match("catnip"); // false
rule.Match("dogfish"); // false
rule.Match("catfish"); // true
```

You can build compound rules that attempt to match any of a sequence of rules using the pipe ("|") operator.

Collapse | Copy Code

```
var rule = Grammar.MatchString("cat") |
Grammar.MatchString("dog")
rule.Match("catfish"); // true
rule.Match("doggedly"); // true
```

Rules operators can be combined to create even more sophisticated rules.

Collapse | Copy Code

```
var rule = (Grammar.MatchString("cat") |
```

```
Grammar.MatchString("dog")) + Grammar.MatchString("fish")
rule.Match("dogfish"); // true
rule.Match("catfish"); // true
rule.Match("swordfish"); // false
rule.Match("cat"); // false
```

Rules can be defined as optional using the `Grammar.Opt()` function:

Collapse | Copy Code

```
var rule = Grammar.MatchString("cat") +
    Grammar.Opt(Grammar.MatchString("fish") |
    Grammar.MatchString("nap"));
rule.Match("cat"); // true
rule.Match("catfish"); // true
```

Repeated rules can also be created using `Grammar.ZeroOrMore()` and `Grammar.OneOrMore()`. For example:

Collapse | Copy Code

```
var rule =
Grammar.OneOrMore(Grammar.MatchString("badger")) +
Grammar.MatchString("snake");
rule.Match("badger badger badger badger snake!"); // true
```

Creating a Parse Tree

Simply recognizing whether or not a string belongs to some grammar is not particularly useful when writing a programming language tool. What we really want is a data structure that represents the structure of the input.

Jigsaw allows this to be done by embedding `Grammar.Node` rules in the grammar. These rules add a new instance of a `Node` to a parse tree if the associated rule is matched successfully. Calling the `Rule.Parse()` function will return a list of parse nodes, each one the root of a parse tree.

Node rules have to be named using the `Rule.SetName()` function (unlike other rules where the name is optional). This name is used as the label of the associated node in the node tree.

The following code defines a simple grammar for parsing words:

Collapse | Copy Code

```
// Define the rules
Rule word =
Grammar.Node(Grammar.Pattern(@"\w+")).SetName("word");
Rule ws = Grammar. Pattern(@"\s+");
Rule eos = Grammar.CharSet("!.?");
Rule sentence = Grammar.Node(Grammar.ZeroOrMore(word | ws)
+
  eos).SetName("sentence");
Rule sentences = Grammar.OneOrMore(sentence +
Grammar.Opt(ws));

var nodes = sentences.Parse("Hey! You stole my pen. Hey
you stole my pen!");
foreach (var n in nodes) {
  Console.WriteLine(n);
  foreach (var n2 in n.Nodes)
```

```
    Console.WriteLine("  " + n2);
}
```

The output of the above program is:

Collapse | Copy Code

```
sentence:Hey!
  word:Hey
sentence:You stole my pen.
  word:You
  word:stole
  word:my
  word:pen
sentence:Hey you stole my pen!
  word:Hey
  word:you
  word:stole
  word:my
  word:pen
```

Memoizing (Caching) Intermediate Results

Some recursive-descent parsers can take extremely long to parse certain inputs. A solution to this is to cache intermediate parse results in a look-up table. This technique is called *memoization*.

In the Jigsaw library, all `NodeRule` match results are cached in a dictionary stored in the `ParserState` object.

If you set the `NodeRule.UseCache` constant to false, you can see how long it takes to parse certain grammars by running the tests in the `JavaScriptTests` class.

Simplifying Grammars: Deriving from the Grammar Class

When defining a grammar, the following points are particularly annoying:

1. Needing to add the prefix `Grammar` in front of the rule creation functions.
2. Needing to explicitly set the names of node rules.

You can work around these issues by defining all rules in a class derived from `Grammar` with each rule declared as a public static field. You can then use the `InitGrammar()` function in the static initializer to assign names automatically to each rule associated with a field.

You could simplify the grammar used in the previous example as follows:

Collapse | Copy Code

```
public class SentenceGrammar : Grammar
{
    public static Rule word = Node(Pattern(@"\w+"));
    public static Rule ws = Pattern(@"\s+");
    public static Rule eos = CharSet("!.?");
    public static Rule sentence = Node(ZeroOrMore(word |
ws) + eos);
    public static Rule sentences = OneOrMore(sentence +
```

```
Opt(ws));

    static SentenceGrammar() {
        Grammar.InitGrammar(typeof(SentenceGrammar));
    }
}
```

Recursive Rules

Rules defined as variables or fields can't refer to themselves or to a rule that hasn't been defined yet. This would result in a null reference in the rule definition. For example, the following grammar will generate an error during type initialization:

Collapse | Copy Code

```
static Rule Number = Pattern("\d+");
static Rule Operator = Node(MatchCharSet("+-*/"));
static Rule Expr = (MatchString("(") + Expr + ")")| (Expr
+ Operator + Expr) | Number;
```

One solution is to use `Recursive()` rules which take a lambda expression that returns an expression at run time.

Collapse | Copy Code

```
static Rule RecExpr = Recursive(() => Expr);
static Rule Number = Pattern("\d+");
static Rule Operator = Node(MatchCharSet("+-*/"));
static Rule Expr = ("(" + RecExpr + ")")| (RecExpr +
Operator + RecExpr) | Number;
```

Avoiding Left-Recursive Rules

In a PEG grammar, recursive rules are not allowed in the left most position of a sequence. These are called "left-recursive" rules and will cause the parser to enter into an infinite loop.

For example, consider this grammar for recognizing function calls (e.g., f(x) or f(x)(y)(z)):

Collapse | Copy Code

```
static Rule UnaryExpr = Node(Number | Identifier);
static Rule ArgList =
Node(Parenthesize(CommaList(RecExpr())));
static Rule PostInc = Node(MatchString("++"));
static Rule PostDec = Node(MatchString("--"));
static Rule PostfixOp = Node(ArgList | PostInc | PostDec);
static Rule PostfixExpr = Node((Recursive(() => PostfixOp)
+ ArgList) | UnaryExpr);
```

This enters into an infinite loop and will cause a stack overflow exception. The workaround is to rewrite the recursive rule as an iterative rule.

Collapse | Copy Code

```
static Rule PostfixExpr = Node(UnaryExpr +
ZeroOrMore(PostfixOp));
```

This rule recognizes the desired expressions but unfortunately produces a parse-tree which can introduce undesired complexity into an evaluator or compiler. You can address this by using a post-parse transformation pass. This is discussed in the "Writing

a Tree Transformer" section.

A Simple Arithmetic Grammar

Putting together everything we have learned so far, here is a simple grammar for parsing arithmetic expressions:

Collapse | Copy Code

```
class ArithmeticGrammar : SharedGrammar
{
    new public static Rule Integer =
Node(SharedGrammar.Integer);
    new public static Rule Float =
Node(SharedGrammar.Float);
    public static Rule RecExpr = Recursive(() =>
Expression);
    public static Rule ParanExpr = Node(CharToken('(') +
RecExpr + WS + CharToken(')'));
    public static Rule Number = (Integer | Float) + WS;
    public static Rule PrefixOp = Node(MatchStringSet("! -
~"));
    public static Rule PrefixExpr = Node(PrefixOp +
Recursive(() => SimpleExpr));
    public static Rule SimpleExpr = PrefixExpr | Number |
ParanExpr;
    public static Rule BinaryOp =
      Node(MatchStringSet("<= >= == != << >> && || < > & |
+ - * % / ^"));
    public static Rule Expression = Node(SimpleExpr +
ZeroOrMore(BinaryOp + WS + SimpleExpr));
    static ArithmeticGrammar() {
InitGrammar(typeof(ArithmeticGrammar)); }
}
```

You may have noticed that I've cheated here by using a `SharedGrammar` base class. This class defines some additional common rules and rule operations like WS, Integer, Float, and `CharToken()`. I'll leave it to you to poke around in the code to see what is available.

Evaluating the Arithmetic Grammar Parse Tree

Once a parse tree is generated by parsing a string, we will want to convert it into a value. This process is called *evaluation*.

To do this, we create a new class called `ArithmeticEvaluator` that has two functions: `Eval(string s)` and `Eval(node n)`.

For the sake of convenience, both functions return a `dynamic` value. When we declare a value as having the `dynamic` type, it tells the compiler to skip type-checking. Instead all operations are resolved at run-time.

Collapse | Copy Code

```
class ArithmeticEvaluator
{
    public static dynamic Eval(string s)
    {
        return
Eval(Grammars.ArithmeticGrammar.Expression.Parse(s)[0]);
    }
```

```
    public static dynamic Eval(Node n)
    {
        switch (n.Label)
        {
            case "Number":      return Eval(n[0]);
            case "Integer":     return Int64.Parse(n.Text);
            case "Float":       return
Double.Parse(n.Text);
            case "PrefixExpr":
                switch (n[0].Text)
                {
                    case "-": return -Eval(n[1]);
                    case "!": return !Eval(n[1]);
                    case "~": return ~Eval(n[1]);
                    default: throw new
Exception(n[0].Text);
                }
            case "ParanExpr":   return Eval(n[0]);
            case "Expression":
                switch (n.Count)
                {
                    case 1:
                        return Eval(n[0]);
                    case 3:
                        switch (n[1].Text)
                        {
                            case "+": return Eval(n[0]) +
Eval(n[2]);
                            case "-": return Eval(n[0]) -
Eval(n[2]);
                            case "*": return Eval(n[0]) *
Eval(n[2]);
                            case "/": return Eval(n[0]) /
Eval(n[2]);
                            case "%": return Eval(n[0]) %
Eval(n[2]);
                            case "<<": return Eval(n[0])
<< Eval(n[2]);
                            case ">>": return Eval(n[0])
>> Eval(n[2]);
                            case "==": return Eval(n[0])
== Eval(n[2]);
                            case "!=": return Eval(n[0])
!= Eval(n[2]);
                            case "<=": return Eval(n[0])
<= Eval(n[2]);
                            case ">=": return Eval(n[0])
>= Eval(n[2]);
                            case "<": return Eval(n[0]) <
Eval(n[2]);
                            case ">": return Eval(n[0]) >
Eval(n[2]);
                            case "&&": return Eval(n[0])
&& Eval(n[2]);
                            case "||": return Eval(n[0])
|| Eval(n[2]);
                            case "&": return Eval(n[0]) &
Eval(n[2]);
                            case "|": return Eval(n[0]) |
Eval(n[2]);
                            default: throw new
Exception("Unrecognized operator " + n[1].Text);
                        }
                    default:
                        throw new Exception(String.Format(
                          "Unexpected number of nodes {0}
in expression", n.Count));
                }
            default:
                throw new Exception("Unexpected type of
node " + n.Label);
        }
```

```
    }
}
```

Writing a JSON Parser

JSON stands for *JavaScript Object Notation*. It is a subset of the JavaScript language that is frequently used as a textual data representation language. It has a similar structure to XML.

In the Jigsaw library, there is a class `JsonObject` derived from `DynamicObject` that can be created dynamically from a string. This means that we can write something like:

Collapse | Copy Code

```
dynamic d = JsonObject.Parse("{ \"answer\" : 42 }");
Console.WriteLine(d.answer);
```

For this article, the most interesting part of the implementation of `JsonObject` is the `Parse()` function.

Collapse | Copy Code

```
public static JsonObject Parse(string s)
{
    var nodes = JsonGrammar.Object.Parse(s);
    return Eval(nodes[0]);
}
```

To implement the parse function, we first need to define a JSON grammar.

Collapse | Copy Code

```
public class JsonGrammar : SharedGrammar
{
    new public static Rule Integer =
Node(SharedGrammar.Integer);
    new public static Rule Float =
Node(SharedGrammar.Float);
    public static Rule Number = Node(Integer | Float);
    public static Rule True = Node(MatchString("true"));
    public static Rule False = Node(MatchString("false"));
    public static Rule Null = Node(MatchString("null"));
    public static Rule UnicodeControlChar =
      Node(MatchString("\\u") + HexDigit + HexDigit +
HexDigit + HexDigit);
    public static Rule ControlChar = Node(MatchChar('\\')
+ CharSet("\"\\/bfnt"));
    public static Rule PlainChar = Node(ExceptCharSet("\"
\\")); // "
    public static Rule Char = Node(UnicodeControlChar |
ControlChar | PlainChar);
    public static Rule StringChars =
Node(ZeroOrMore(Char));
    public static Rule String = Node(MatchChar('"') +
StringChars + MatchChar('"'));
    public static Rule Value =
      Node(Recursive(() => String | Number | Object |
Array | True | False | Null));
    public static Rule Name = Node(String);
    public static Rule Pair = Node(Name + WS +
CharToken(':') + Value + WS);
    public static Rule Members =
Node(CommaDelimited(Pair));
    public static Rule Elements =
Node(CommaDelimited(Value));
    public static Rule Array = Node(CharToken('[') +
```

```
Elements + WS + CharToken(']'));
    public static Rule Object = Node(CharToken('{') +
Members + WS + CharToken('}'));
    static JsonGrammar() {
InitGrammar(typeof(JsonGrammar)); }
}
```

Next we need to write a function that converts from a parse tree to a `JsonObject`. We will follow the same basic form as the ArithmeticEvaluator example. We will write only one `Eval()` function that can return any valid JSON type (e.g., number, string, array, etc.) depending on the label of the argument.

Collapse | Copy Code

```
public static dynamic Eval(Node n)
{
    switch (n.Label)
    {
        case "Name": return Eval(n[0]);
        case "Value": return Eval(n[0]);
        case "Number": return Eval(n[0]);
        case "Integer": return Int32.Parse(n.Text);
        case "Float": return Double.Parse(n.Text);
        case "String": return n.Text.Substring(1,
n.Text.Length - 2);
        case "True": return true;
        case "False": return false;
        case "Null": return new JsonObject();
        case "Array": return n.Nodes.Select(Eval).ToList();
        case "Object":
            {
                var r = new JsonObject();
                foreach (var pair in n.Nodes)
                {
                    var name = pair[0].Text;
                    var value = Eval(pair[1]);
                    r[name] = value;
                }
                return r;
            }
        default:
            throw new Exception("Unexpected node type " +
n.Label);
    }
}
```

Writing a Simple JavaScript Interpreter

The simplest kind of interpreter is little more than a wrapper around an evaluation function. Unlike the JSON and arithmetic evaluators, a programming language evaluator has to manage variable and function names.

Writing a JavaScript Grammar

It makes sense to derive a JavaScript grammar from the JSON grammar. Some rules will have to be rewritten, such as rules for defining object and array literals so that arbitrary expressions can be placed in objects and arrays, not just literals.

Here is the JavaScript grammar:

Collapse | Copy Code

```
public class JavaScriptGrammar : JsonGrammar
```

```
{
    // Recursive rules defined at the top
    public static Rule RecExpr = Recursive(() => Expr);
    public static Rule RecStatement = Recursive(() =>
Statement);
    public static Rule Literal =
      Recursive(() => String | Integer | Float | Object |
Array | True | False | Null);

    // Redefine Identifier so that it creates nodes in the
parse tree
    public new static Rule Identifier =
Node(SharedGrammar.Identifier);

    // The following rules are redefined from JsonGrmmar
because
    // arbitrary expressions are allowed, not just literals
    public static Rule PairName = Identifier |
DoubleQuotedString | SingleQuotedString;
    public new static Rule Pair = Node(PairName + WS +
CharToken(':') + RecExpr + WS);
    public new static Rule Array =
      Node(CharToken('[') + CommaDelimited(RecExpr) + WS +
CharToken(']'));
    public new static Rule Object =
      Node(CharToken('{') + CommaDelimited(Pair) + WS +
CharToken('}'));

    // Function expressions
    public static Rule ParamList =
Node(Parenthesize(CommaDelimited(Identifier + WS)));
    public static Rule NamedFunc =
Node(Keyword("function") +
                                    Identifier + WS +
ParamList + RecStatement);
    public static Rule AnonFunc = Node(Keyword("function")
+
                                   ParamList +
RecStatement);
    public static Rule Function = NamedFunc | AnonFunc;

    // Expression rules
    public static Rule ArgList = Node(CharToken('(') +
CommaDelimited(RecExpr) + CharToken(')'));
    public static Rule Index = Node(CharToken('[') +
RecExpr + CharToken(']'));
    public static Rule Field = Node(CharToken('.') +
Identifier);
    public static Rule PrefixOp = Node(MatchStringSet("! -
~"));
    public static Rule ParenExpr = Node(CharToken('(') +
RecExpr + WS + CharToken(')'));
    public static Rule NewExpr = Node(Keyword("new") +
Recursive(() => PostfixExpr));
    public static Rule LeafExpr = ParenExpr | NewExpr |
Function | Literal | Identifier;
    public static Rule PrefixExpr = Node(PrefixOp +
Recursive(() => PrefixOrLeafExpr));
    public static Rule PrefixOrLeafExpr = PrefixExpr |
LeafExpr;
    public static Rule PostfixOp = Field | Index | ArgList;
    public static Rule PostfixExpr = Node(PrefixOrLeafExpr
+ WS + OneOrMore(PostfixOp + WS));
    public static Rule UnaryExpr = PostfixExpr |
PrefixOrLeafExpr;
    public static Rule BinaryOp =
      Node(MatchStringSet("<= >= == != << >> && || < > & |
+ - * % /"));
    public static Rule BinaryExpr = Node(UnaryExpr + WS +
OneOrMore(BinaryOp + WS + UnaryExpr));
    public static Rule AssignOp =
      Node(MatchStringSet("&&= ||= >>= <<= += -= *= %= /=
```

```
&s= |= ^= ="));
    public static Rule AssignExpr = Node((Identifier |
PostfixExpr) + WS + AssignOp + WS + RecExpr);
    public static Rule TertiaryExpr = Node((AssignExpr |
BinaryExpr | UnaryExpr) +
           WS + CharToken('?') + RecExpr + CharToken(':')
+ RecExpr + WS);
    public static Rule Expr = Node((TertiaryExpr |
AssignExpr | BinaryExpr | UnaryExpr) + WS);

    // Statement rules
    public static Rule Block = Node(CharToken('{') +
ZeroOrMore(RecStatement) + CharToken('}'));
    public static Rule VarDecl = Node(Keyword("var") +
Identifier + WS + Opt(Eq + Expr) + Eos);
    public static Rule While = Node(Keyword("while") +
Parenthesize(Expr) + RecStatement);
    public static Rule For = Node(Keyword("for") +
           Parenthesize(VarDecl + Expr + WS + Eos + Expr +
WS) + RecStatement);
    public static Rule Else = Node(Keyword("else") +
RecStatement);
    public static Rule If = Node(Keyword("if") +
Parenthesize(Expr) + RecStatement + Opt(Else));
    public static Rule ExprStatement = Node(Expr + WS +
Eos);
    public static Rule Return = Node(Keyword("return") +
Opt(Expr) + WS + Eos);
    public static Rule Empty = Node(WS + Eos);
    public static Rule Statement =
      Block | For | While | If | Return | VarDecl |
ExprStatement | Empty;

    // The top-level rule
    public static Rule Script = Node(ZeroOrMore(Statement)
+ WS + End);

    // Grammar initialization
    static JavaScriptGrammar()
    {
        InitGrammar(typeof(JavaScriptGrammar));
    }
}
```

Writing a Source Code Printer

Given a parse tree generated from a JavaScript parser, one of the simplest tools we can build is a source code printer. This is a useful intermediate step when you are developing a language for validating that the parser is working as expected.

A source code printer (or *pretty printer*) prints a formatted representation of the input AST. This can be useful in text editors, or when doing source-to-source translation.

The `Printer` class in the Jigsaw library facilitates writing custom source translators. By deriving from `Printer`, you only need to override the `Print()` function and can then easily generate output depending on the kind of node received.

The following snippet of code is taken from the `JavaScriptSourcePrinter` class.

Collapse | Copy Code

```
switch (n.Label)
{
```

```
    case "Script":
        return Print(n.Nodes);
    case "Statement":
        return Print(n[0]);
    case "Empty":
        return Print(";");
    case "Return":
        return (n.Count > 0)
            ? Print("return ").Print(n[0]).Print(";")
            : Print("return;");
    case "ExprStatement":
        return Print(n[0]).Print(";");
    case "If":
        Print("if
(").Print(n[0]).Print(")").Indent().Print(n[1]).Unindent();
        return n.Count > 2
            ? Print(n[2])
            : this;
    case "Else":
        return
Print("else").Indent().Print(n[0]).Unindent();
    //...
}
```

Writing a Tree Transformer

The JavaScript parser works well, although certain parts of the parse tree will be hard to work with during evaluation:

- Chained binary expressions are not separated. For example, when the `BinaryExpr` rule encounters `3 + 4 * 5`, it will treat one node with 5 children (3, +, 4, *, 5) instead of (3, +, (4, * ,5)) which would be easier to work with.
- Chained postfix operators are not separated. For example, `f(1)["hello"].field` will be parsed by the `PostfixExpr` rule as (f, (1), `["hello"]`, `.field`)" whereas it would be easier if it was (((f, (1)), `["hello"]`), `.field`).

To simplify the evaluator, and possibly other tools, a class derived from `TreeTransformer` called `JSTransformer` can be found in the Jigsaw library. This transformer does a number of tasks:

- Converts postfix expressions into one of the following types of expression:
    - FieldExpr – An expression followed by a "." and an identifier. For example, "myobject.myfield".
    - IndexExpr – An expression followed by an index operator. For example, "myarray[index]".
    - CallExpr – An expression followed by an argument list. For example, "myfunc(arg0, arg1)".
    - MethodCallExpr – A `FieldExpr` followed by an argument list. For example, "myobject. myfunc(arg0, arg1)".
- Separates long binary expressions according to precedence rules.
- Converts named functions (e.g., function f(x) { }) into variable declarations of the form: var f = function(x) { }.

- Converts `while` loops into `for` loops.
- Rewrites special assignment operators, so that the evaluator only has to consider the basic assignment operator. For example, "a += b" becomes "a = a + b".
- Replaces nodes which only ever have one child by the child node.

Managing the Environment

When you write an evaluator for a programming language, you have to track values associated with names (e.g., function names, variable names, argument names). The binding of values to names is collectively called the *environment*. In Jigsaw, the environment is managed by a class called `VarBindings`. You can see an example of its usage in the `Evaluator` class.

The `VarBindings` class is a recursively defined associative list class. It contains a name and its associated value, along with a pointer to another `VarBindings` class. This representation of an environment is not particularly efficient but very convenient to use.

In JavaScript, when a variable is declared, it is added to the list of variable bindings. When a "block" goes out of scope, all names declared in that scope are unbound. In Jigsaw, this is done in a function called "`EvalScoped`". The `EvalScoped` function takes another function as an argument. It takes a snapshot of the environment, executes the function, and then restores the environment.

Collapse | Copy Code

```
public dynamic EvalScoped(Func<dynamic> f)
{
    var e = env;
    var r = f();
    env = e;
    return r;
}
```

In JavaScript, when a new name is first used, a binding is created automatically. This might not have been a great language design decision, since it increases the chances of programmer error, but we respect it in our implementation. This logic is handled in a function called `AddOrCreateBinding()`.

JavaScript Functions

Unlike JSON, when implementing a JavaScript interpreter, we have to consider what data structure to use to represent functions. In JavaScript, functions are *closures*. This means the function can refer to variables declared outside of the function. Such variables are called *free variables*.

One of the simplest methods to implement a closure is to use a copy of the environment from the moment the function value is declared. When the function is applied to its arguments (i.e., called), the current evaluator's environment is temporarily

replaced with the version stored with the function.

The implementation of the JavaScript function used in Jigsaw is shown here:

Collapse | Copy Code

```
public class JSFunction
{
    public static int FuncCount = 0;
    public Node node;
    public VarBindings capture;
    public Node parms;
    public string name = String.Format("_anonymous_{0}",
FuncCount++);
    public Node body;

    public JSFunction(VarBindings c, Node n) {
        capture = c;
        node = n;
        if (n.Count == 3)
        {
            name = n[0].Text;
            parms = n[1];
            body = n[2];
        }
        else
        {
            parms = n[0];
            body = n[1];
        }
    }

    public dynamic Apply(JavaScriptEvaluator e, params
dynamic[] args)
    {
        var restore = e.env;
        var originalReturningState = e.isReturning;
        dynamic result = null;

        try
        {
            e.env = capture;
            e.isReturning = false;
            int i = 0;
            foreach (var p in parms.Nodes)
                e.AddBinding(p.Text,
args[i++]);
            e.Eval(body);
            result = e.result;
        }
        finally
        {
            e.result = null;
            e.env = restore;
            e.isReturning = originalReturningState;
        }
        return result;
    }
}
```

Return Statements

When evaluating a series of statements and a `return` statement is encountered, you need to store the return value and exit the enclosing function.

In the JavaScript evaluators, we set a flag (`isReturning`). This flag is checked whenever a compound statement is executed

(e.g., for loops, while loops, etc.), to decide whether the execution of later statements should be skipped. We chose this approach because it is simple to understand and easy to implement.

An even simpler approach would have been to throw an exception. However, this would have made debugging harder and would have had a significant negative impact on performance.

A more efficient approach to this problem is to rewrite the parse tree so that there is only one "`return`" statement at the end of a function. This is a bit complicated since it requires adding additional variables and rewriting any loop conditions, but the complexity would be moved out of the evaluator function and into the transformer.

The Evaluation Function

The JavaScript evaluation function is similar in structure to the `JsonObject.Eval()` function with the following differences:

- The node tree is transformed into a form that is a bit easier to evaluate
- Variables are managed in a `VarBindings` object
- A new data type (`JSFunction`) is introduced

The evaluation function is too long to list here, but here is a representative snippet:

Collapse | Copy Code

```
case "AnonFunc":
    // Creates an unnamed function
    return new JSFunction(env, n);
case "Block":
    // Execute a sequence of instructions
    return EvalScoped(() => EvalNodes(n.Nodes));
case "If":
    // Check if the condition is false (or NULL)
    if (Eval(n[0]) ?? false)
        // Execute first statement
        return Eval(n[1]);
    else if (n.Count > 2)
        // Execute else statement if the condition is
false, and it exists
        return Eval(n[2]);
    else
        // By default reutrn the result
        return null;
case "VarDecl":
    // Variable declaration
    // It may or may not be initialized
    return AddBinding(n[0].Text, n.Count > 1 ? Eval(n[1])
: null);
case "Empty":
    // An empty statement means we do nothing
    return null;
case "ExprStatement":
    return Eval(n[0]);
```

Extending the Primitive Set

The evaluator included implements only basic operators, and has a single built-in function "`alert`". Built-in functions are implemented using the `JSPrimitive` class and are added to the global environment when an evaluator is initialized.

Collapse | Copy Code

```
public JavaScriptEvaluator()
{
    // This is where you could add all sorts
    // of primitive objects and functions. Or don't. Fine.
    AddBinding("alert", new JSPrimitive(args => {
Console.WriteLine(args[0]); }));
}
```

A JavaScript to Expression Tree Compiler

The `Expression` class in the `System.Linq.Expressions` namespace (also known as an *expression tree*) is a convenient way to dynamically create new functions at run-time. By converting a parse tree into an expression tree, we can then use the `Expression.Compile()` function to create delegates at run-time that can be executed using `DynamicInvoke()`.

Generating expression trees is similar to evaluation, except that instead of returning a dynamic value for each node, we return an instance of the `Expression` class. An expression compiler can be derived from the `ExpressionCompiler` utility class.

The sample JavaScript to expression compiler in the Jigsaw library, `JavaScriptExpressionCompiler`, provides two compilation functions, one that takes a `string`, and the other that takes a `Node`.

Collapse | Copy Code

```
public static Delegate CompileLambda(string s)
{
    var nodes = JavaScriptGrammar.AnonFunc.Parse(s);
    return CompileLambda(nodes[0]);
}

public static Delegate CompileLambda(Node n)
{
    n = JavaScriptTransformer.Transform(n);
    var compiler = new JavaScriptExpressionCompiler();
    var expr = (LambdaExpression)compiler.ToExpr(n);
    if (expr == null) return null;
    return expr.Compile();
}
```

Note that like the evaluation function, `Transform` is used to simplify the parse tree before entering into the "`ToExpr`" function which converts each Node into an Expression.

Next Steps

There are a number of JavaScript features that are not implemented in the JavaScript interpreter and expression compiler. For further study, you can extend the samples provided to implement more features or add programming language features that you may find interesting.

Hopefully you have learned enough that you can start experimenting with creating your own languages.

There are several other samples included with the Jigsaw library that you may find interesting:

- *ILCompiler.cs* – An implementation of an IL assembly code.
- *SchemeExpressionCompiler.cs* – A simple expression compiler that works for a small subset of the Scheme language. Scheme is a dialect of LISP.
- *CatEvaluator.cs* – Contains an evaluator for the Cat language, a simple functional stack-based language.

Final Words

This article only scratches the surface of implementing programming languages. If you liked this article and want to learn more about programming language implementation, you may be interested to know that I am working on a book with the working title "Implementing Programming Languages in C#". Please email me at cdiggins@gmail.com if you would like to learn more, become a beta reviewer, or just to show your support. Thanks!

History

- Oct. 22, 2011 - First submission.
- Oct. 23, 2011 - Removed superfluous files from download package, and added missing link to download.
- Oct. 28, 2011 - Made a large number of edits thanks to a wonderfully detailed review by Tracey Houston. Thanks Tracey!
- Dec. 27, 2011 - Made a number of edits thanks to a detailed review by David Haguenauer. Thanks David, sorry it took so long for me to add them!

License

About the Author

Christopher Diggins
Software Developer Autodesk

Canada
Member

Follow on Twitter
Google

This article was written by Christopher Diggins, a computer

science nerd who currently works at Autodesk as an SDK specialist.

Comments and Discussions

[modified]

Last Visit: 31 Dec '99 - 18:00
Last Update: 1 Apr '13 - 17:57

Refresh 1 2 3 4 5 6 7 8 Next »

General News Suggestion Question Bug Answer Joke Rant Admin