Packrat Parser in C#

This parser is an implementation of a [Packrat Parser](http://en.wikipedia.org/wiki/Parsing_expression_grammar) with support for left-recursion.  The algorithm for left recursion is a modified version of [Packrat parsers can support left recursion](http://portal.acm.org/citation.cfm?id=1328408.1328424).

A normal PackRat parser utilizes a Parsing Expression Grammar (PEG) and parses input in linear time.  This parser does the same, but allows for PEGs that contain left-recursion, a staple for anyone familiar with BNF while still maintaining near linear-time.  A PEG is far more expressive than BNF, allowing for *conditional*, *not*, *zero/one or more*, and other useful operators.  The syntax of a standard PEG has been modified to allow the grammars to be expressed naturally using C# syntax.

### Standard PEG Syntax

Formally, a parsing expression grammar consists of:

* A finite set N of nonterminal symbols
* A finite set Σ of terminal symbols that is disjoint from N.
* A finite set P of parsing rules.
* An expression eS termed the starting expression.

Each parsing rule in P has the form A ← e, where A is a nonterminal symbol and e is a parsing expression. A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An atomic parsing expression consists of:
   * any terminal symbol,
   * any nonterminal symbol, or
   * the empty string ε.
2. Given any existing parsing expressions e, e1, and e2, a new parsing expression can be constructed using the following operators:
   * Sequence: e1 e2
   * Ordered choice: e1 / e2
   * Zero-or-more: e\*
   * One-or-more: e+
   * Optional: e?
   * And-predicate: &e
   * Not-predicate: !e

*From Wikipedia*

This parser is intended to be just as convenient to use for building a full-scale grammar as for simple ad hoc search/replace in strings.  It allows for grammars as sophisticated as languages such as C# itself, to patterns that are simple and nearly as concise as regular expressions (but far more scrutable).

Before going into depth on what each operator is about, let’s look at some small examples.

## Decimals

Let's start with a simple pattern that will match a decimal number such as **1**, **42**, **3.14**, **-9**, etc. First let's create a pattern to handle **1**:

var pattern = '0'.To('9');

This creates a pattern that accepts any character between 0 and 9, in other words, a digit. To test this pattern for 1 we simply call the Match(...) method:

bool result = pattern.Match("1");

Since **1** is a valid character, it would be accepted by this pattern. What about **42**? **4** is a valid character, and so is **2**, so will it accept this input? The answer is no, because we have designed the pattern to only account for a single digit. Since **42** has two digits, it fails. We'll have to modify our pattern to support multiple digits. Fortunately, that is easy -- we can just use the '+' operator to indicate that one or more digits in a row are allowed:

var pattern = +'0'.To('9');

The plus (+) unary prefix operator requires at least one instance of its operand to be in the input, but allows any number of valid subsequent instances. In this case, '0'.To('9') is the operand so this allows any number of digits. Thus, **42** is now allowed.

Clearly **3.14** will fail as we've done nothing to account for the decimal point. Let's modify the pattern:

var pattern = +'0'.To('9') + '.' + +'0'.To('9');

While this works, it's probably a bit vexing that '0'.To('9') is repeated twice. We can easily fix that by simply declaring a variable:

var digit = +'0'.To('9');  
var pattern = digit + '.' + digit;

Unfortunately, with our change, it now requires the decimal point, so our first two examples now fail. This is easy to fix using the optional operator, expressed as a tilde (~):

var pattern = digit + ~('.' + digit);

Finally, we need to handle the minus sign (-):

var pattern = ~'-'.\_() + digit + ~('.' + digit);

And there you have it, a pattern that can handle any integer or decimal number.

You may be asking yourself about '.\_()'. The problem is that all the operators you've seen so far work on the class Expression. '.' is a **char**, and thus the ~ operator won't work on it. One solution is to create an extension method that takes a **char** and returns a CharacterTerminal (a type of Expression) based off it. That is what the .\_() extension method on **char** is, and hence the syntax. It was chosen to be short and minimize distraction.

## Quoted Strings

To illustrate the use of some other operators, let's create a pattern to match quoted string literals as you see in many languages (such as C#). Some valid values: "value", "", "a \"quote\"". We'll start by creating a pattern that can match "value":

### Modifications to PEG

This parser uses C# operator overloading in order to represent these patterns as naturally as possible. There are only so many operators available, and as such the symbols as defined by PEG have been adjusted as follows:

**sequence**: No operator, just expressions followed in succession. This would be illegal in C# so the + operator is used to separate expressions in a sequence.

**ordered choice**: '/'

The precedence of this operator in C# would be far too high so has been changed to the traditional pipe character | which has very low precedence.

**one or more**: '+'

In C# it's the same, but the unary operator must precede the expression, rather than as a suffix. i.e. +a instead of a+.

**zero or more**: '\*'

In C#, '\*' is not a valid unary operator, so has been changed to -. Use as a prefix.

**optional**: '?'

In C#, ? is not a valid unary operator, so has been changed to ~. Again, as a prefix.

**and predicate**: '&'

We have run out of operators, so we will use an extension method named And.

**not predicate**: '!'

Exactly the same. ! should precede the expression.

var pattern = '"' + +Peg.Any + '"';

That is, a double-quote, the special operator Any, and a final double-quote. The Any operator will accept any possible character. But wait, surly this includes the double-quote! How then do you terminate the string? The answer is to also use the *not* operator (!):

var pattern = '"' + +(!'"'.\_() + Peg.Any) + '"';

So the *one-or-more* operator + was used on the expression (!'"'.\_() + Peg.Any). The *not* operator never consumes input, but will cause the Sequence to fail. In this case, that failure will mean that the *one-or-more* operator will stop consuming further characters. At that point, we arrive at the final double-quote and accept it.

Next, will this pattern successfully match the empty string, ""? Since we are using a *one*-or-more operator, it will not! Since having no characters between the double-quotes is an acceptable match, we need to replace the operator with the *zero*-or-more operator:

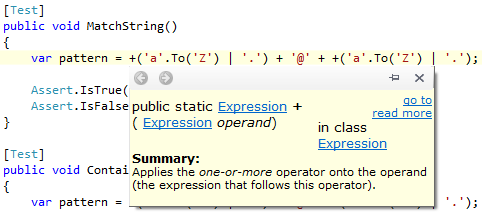
var pattern = '"' + -(!'"'.\_() + Peg.Any) + '"';

As you can see, we switched the + for the - ; the minus sign indicates *zero-or-more*. Since zero is less than one (and vice versa), hopefully the symbols should be easy to remember. For the finishing touch, we want to allow double-quote characters so long as they are properly escaped with the backslash:

var pattern = '"' + -(!('"'.\_() | @"\") + Peg.Any | @"\\" | @"\""") + '"';

Where before it was a *one-or-more* of "not the double-quote and any character" now it is a *one-or-more* of "neither the double-quote nor the backslash plus any character **or** *two* backslashes in a row (an escaped backslash) **or** a backslash followed by a double-quote (an escaped double-quote)". With just a short pattern like this, we have a powerful way to parse any double-quoted string, including the ability to escape the double-quote.

It's true, some of this syntax may look pretty foreign. But one thing to remember is that *this is all C#*. This means that you get all the help that you're used to from C#:



Now let's take a look at the complete API available to you in your grammars and patterns.

# API Reference

A PEG is built by composing complex **expressions** together based on simpler ones. When they are composed inline as C# expressions and statements, they are referred to as **patterns**. (These are what you saw in the two samples above) Patterns are concise and easy to whip up when the need arises. Some disadvantages to them are that they are anonymous and they cannot support recursion. For example, we earlier declared digit this way:

var digit = +'0'.To('9');  
var pattern = digit + '.' + digit;

pattern.ToString() would yield "? + '.' + ?". If you want more readable strings, it *is* possible to name them explicitly:

var digit = +'0'.To('9').Name("digit");

Feels a little dirty to type "digit" twice but now, pattern.ToString() would yield "digit + '.' + digit" -- just as you typed it.

Patterns also cannot support recursion. For example, this simple domain name pattern is not possible:

var word = +'a'.To('Z');  
var domain = domain + '.' + word | word;

This fails because we are trying to use domain before we have finished declaring it. An alternative approach is to define a class that represents your syntax, and this is called a **grammar**. A grammar is composed of a series of instance methods that return an Expression. Each method represents a **nonterminal**. (We created a nonterminal in the above example when we called .Name("digit")). A grammar for the above *domain* pattern would be:

public class DomainGrammar : Grammar<DomainGrammar>  
{  
 public virtual Expression Domain()  
 {  
 return Domain() + '.' + Word() | Word();  
 }  
  
 public virtual Expression Word()  
 {  
 return +'a'.To('Z');  
 }  
}

As you can see, using this syntax it is possible for Domain() to recursively call itself. Also, the nonterminals in a grammar are automatically named, saving you the step of calling the .Name(...) method as earlier. Grammars also have the advantage of publicly exposing the nonterminals in a way that is accessible for later mapping. They are especially suited for constructing grammars for more complex languages, though can be used for the simplest patterns if desired.

Now let's take a look at all the expressions you have available to you.

## Sequence (+)

A sequence allows you to define a string of expressions that must be satisfied. It overloads the + operator so use it in the same way to concatenate two expressions together.

'@' + Peg.Any

This expression would allow any string that starts with a '@' symbol.

## Ordered Choice (|)

The ordered choice allows you to provide a set of possible patterns that will match. **Important**: this operator evaluates from left to right. That means the *first* match will be chosen, even if there is a potentially better match in a later choice. As this is C#, it has lower precedence than the sequence (+) operator. This means that by default (without parentheses) an ordered choice is composed of sequences (or even simpler expressions). For example,

'a' | 'b' + 'c'

This pattern allows "a" or "bc", but not "ab" and not "bc". In other words, it's a *choice* between 'a' **or** 'b' + 'c'. What if we wanted a choice between 'a' or 'b' and *then* plus 'c'? Just use parentheses:

('a' | 'b') + 'c'

Now the strings "ac" and "bc" are accepted.

## One Or More (+)

While the same *symbol* as the *Sequence* operator, this uses the + unary prefix operator, and not the + binary operator. Whatever expression follows the + may repeat any number of times but at least *one* iteration must match.

+'a'.\_() + 'b'

This pattern must start with the 'a' character and can be followed by any number of *additional* 'a' characters, so long as it ends with a 'b'. So "ab", "aab", and "aaaab" are all valid, but "b" is not.

## Zero Or More (-)

Exactly like the *one-or-more* operator except it does not require at least one match. In practice this acts like the "optional" variant of the *one-or-more* operator.

-'a'.\_() + 'b'

This pattern will accept any number of 'a' characters (including none) followed by a 'b'. So "ab", "aab", and "b" are all valid.

## Optional (~)

Whatever expression follows the operator is considered optional. This means that if the expression accepts the input, then it is used. If, however, the expression fails to accept the input, it is ignored and parsing proceeds to the subsequent expression.

'a' + ~'b'.\_() + 'c'

Here the 'b' is optional so "abc" and "ac" are valid.

## And

The operand must be satisfied. However, even upon success no input is consumed. Often used at the end of a sequence to verify that the input that *follows* the current sequence has a certain value.

'a' + 'b' +

## Not (!)

The operand must *not* be satisfied. If it *is* satisfied, the