MicroParser – How to get started

THIS IS A DRAFT

MicroParser – a minimal parser combinator framework for C# with the focus on light dependencies, small size and reasonable performance for strings that fits in memory.

What is parsing?

The process of converting a data stream into an object graph representing the content of that data stream, i.e. *"x = 3"*; can be parsed into *AssignmentExpression(VariableReference("x"), Constant (3))*.

Why use a parser combinator framework?

Parsing is one of the interesting problems that if one apply standard structured programming the parser tends to grow large, unwieldy and with lot of redundancies. Therefore numerous support tools have emerged to help developers perform simple parsing, i.e. Regular Expression. Regular Expression has a couple of drawbacks:

1. No compile-time checking
2. Problematic to parse with operator precedence
3. Problematic to parse string value that contains escaped string terminators
4. No good error reporting if parsing failed
5. Difficult to reuse and combine regular expressions

A parser combinator framework can help with this.

Why is it called parser combinator?

A parser combinator function is a function that takes one or more parser as argument and combines it into a new parser. A parser is basically a function that takes a string and a position and replies with the parser result and the new position. In MicroParser this represented by Parser<T> which is declared as:

```
partial class Parser<TValue>
{
   // ParserState is basically a string with a position
   // ParserReply contains the updated state and the result of the parser
   // operation depending on if the operation was successful
   public delegate ParserReply<TValue> Function(ParserState state);

   public readonly Function Execute;

   public Parser (Function function);
   public static implicit operator Parser<TValue> (Function function);
}
```

Essentially Parser is a placer holder for Parser.Function delegate. The main reason for introducing it as a class is because delegates can't be partial. Secondary reasons are that it enables operator overloading and better performance pinpointing.

As parsers are functions essentially a parser combinator function is by nature a so-called higher-order functions i.e. it takes zero or more functions as arguments and produces a function. This has been a trademark of functional programming languages for years it gives a lot of expressiveness but these days.NET developers are using it every day as almost all functions in System.Linq.Enumerable are higher-order functions.

To most developers higher-order functions seems confusing the first time one starts to use it, especially in the context of a parser combinator framework that so heavily relies on higher-order functions. My advice is to ignore the underlying details and try to read the parser combinator code as a configuration of a parser just as Regular Expression is a string configuration of a parser.

After you have convinced yourself that it actually works it can be interesting to look into how actually a parser combinator framework works:

Parser combinator resources	
Excellent article on monadic parser combinators	http://www.cs.nott.ac.uk/~gmh/monparsing.pdf
Parser combinators (Wikipedia)	http://en.wikipedia.org/wiki/Parser_combinator

What parser combinator frameworks exist?

This is just a small excerpt of available frameworks:

Name	Description	Language	Silverlight compatible	Pros	Cons (for .NET devs)
Parsec	The mother of modern parser combinator frameworks. Pragmatic design that aims to give good performance.	Haskell	No	Reasonable speed. Good expressiveness.	Requires Haskell runtime. Uncertain how well it integrates with .NET. Binary size (as you have to include Haskell runtime). Haskell language unknown to many .NET developers.
GParsec	Parsec for Groovy.	Groovy (Java)	No	Reasonable speed. Good expressiveness.	Requires Java runtime. .NET and Java doesn't interop cleanly Binary size (as you have to include JRT).
FParsec	Parsec for F#.	F# (.NET)	Yes	Reasonable speed. Good expressiveness. Handles very large inputs.	Binary Size (as FParsec is a very complete framework). F# language not known to all developers.
Boost.Spirit	C++ framework developed by Boost community.	C++	No	Reasonable speed. Mature.	Requires a C++/C LI project to integrate well with .NET. Quite advanced for non C++ devs. Poor error reporting.
MicroParser		C# (.NET)	Yes	Reasonable speed. Good expressiveness. Small binary size. Is designed to be internalized in your assemblies.	Not as complete as FParsec. Can't deal with input strings that are larger than what can be read into memory.

Why develop MicroParser?

Stephan Tolksdorf has done a brilliant job with FParsec and he deserves every bit of respect. However, when getting to the point where I need to deploy FParsec based parsers I have run into some issues:

1. The size of the FParsec framework is around 600kb – in a Silverlight project size matters.
2. FParsec is a visible dependency which can create conflict with other assemblies. This is not unique to FParsec but all third party assemblies suffer from this.
3. Many C# developers are interested but wary of F#, also to maintain an FParsec based parser you have to know a fair bit about FParsec. For a C# dev it can be hard to maintain a parser when one lacks both knowledge of F# and FParsec.

MicroParser aims to be:

1. Smallish – around 30kb
2. Easy to internalize – can be source code included into an assembly to completely hide the dependency
3. Fairly easy for C# developers to understand
4. Deployable using .NET 3.5 runtime or Silverlight runtime. MicroParser requires VS2010 though.

Parsing with MicroParser

This sample is called Sample1 in the project SampleParsers

Let's assume we like to be able to parse the following line:

```
AnIdentifier = 3
```

The expression consist of 2 parts, an identifier and an integer separated by an equal sign '='. In parser combinator framework one builds the parser from the bottom up.

We start by declaring a parser for the integer:

```
// Int () is a builtin parser for ints
Parser<int> p_int = CharParser.Int ();
```

The identifier parser would look like this:

```
Parser<SubString> p_identifier = CharParser
   .ManyCharSatisfy2 (                // Creates a string parser
      CharSatisfy.Letter,             // A test function applied to the
                                      // first character
      CharSatisfy.LetterOrDigit,      // A test function applied to the
                                      // rest of the characters
      minCount:1                      // We require the identifier to be
                                      // at least 1 character long
      );
```

We also needs to some help parsers to consume the '=' token and superfluous whitespace:

```
Parser<Empty> p_spaces     = CharParser.SkipWhiteSpace ();
Parser<Empty> p_assignment = CharParser.SkipChar ('=');
```

Combining the parsers into the complete parser:

```
Parser<Tuple<SubString,int>> p_parser = Parser.Group (
   p_identifier.KeepLeft (p_spaces),
   p_assignment.KeepRight (p_spaces).KeepRight (p_int));
```

In the end we execute the parser on the string:

```
ParserResult<Tuple<string,int>> result = Parser.Parse (
   p_parser,
   "AnIdentifier = 3");

if (result.IsSuccessful)
{
   Console.WriteLine (
      "{0} = {1}",
      result.Value.Item1,
      result.Value.Item2
      );
}
else
{
   Console.WriteLine (
      result.ErrorMessage
      );
}
```

Complete sample:

```
using System;
using MicroParser;
// ReSharper disable InconsistentNaming
namespace SampleParsers
{
   class Program
   {
      static void Main (string[] args)
      {
         // Int () is a builtin parser for ints
         Parser<int> p_int = CharParser.Int ();

         Parser<SubString> p_identifier = CharParser
            .ManyCharSatisfy2 (                 // Creates a string parser
               CharSatisfy.Letter,              // A test function applied to the
                                                // first character
               CharSatisfy.LetterOrDigit,       // A test function applied to the
                                                // rest of the characters
               minCount: 1                      // We require the identifier to be
                                                // at least 1 character long
               );

         Parser<Empty> p_spaces     = CharParser.SkipWhiteSpace ();
         Parser<Empty> p_assignment = CharParser.SkipChar ('=');

         Parser<Tuple<SubString, int>> p_parser =Parser.Group(
            p_identifier.KeepLeft(p_spaces),
            p_assignment.KeepRight(p_spaces).KeepRight(p_int));

         ParserResult<Tuple<SubString,int>> result = Parser.Parse (
            p_parser,
            "AnIdentifier = 3");

         if (result.IsSuccessful)
         {
            Console.WriteLine (
               "{0} = {1}",
               result.Value.Item1,
               result.Value.Item2
               );
         }
         else
         {
            Console.WriteLine (
               result.ErrorMessage
               );
         }

         Console.ReadKey ();
      }
   }
}
```

Implicit Error Messages

A killer feature in Parsec and FParsec is the ability to generate readable error messages without explicit coding. MicroParser attempts to provide this feature as well although admittedly FParsec does a better job at the time of writing.

When using the sample code above:

Input	MicroParser response
AnIdentifier = @	Pos: 15 ('@') - expected : digit
1nIdentifier = 3	Pos: 0 ('1') - expected : letter
AnIdentifier @ 3	Pos: 13 ('@') - expected : '='

This feature alone makes parser combinator frameworks such as Parsec, FParsec and MicroParser a very interesting alternative to Regular Expressions.

Expression parsing with operator precedence

This sample is called Sample2 in the project SampleParsers

The sample above is very straight forward to implement using regular expression and there's little to gain by adding a dependency to MicroParser. However imagine we have an expression such as this:

```
2*(x + 1) + y + 3
```

The expression is provided by our user and we like to compute the user defined expression. One idea is to use the shipped C# compiler that is shipped with .NET to compile the expression and execute it. That will work but there are some issues:

1. The user is writing C#, for simple formulas like above that's not a problem but in some cases we like to add domain specific operators or keywords.
2. Potential security issues – a black hat might use it to execute code in an unsecure context

A different approach is to specify a Domain Specific Language and parse that language using a parser framework such as MicroParser or FParsec. This is an area where Regular Expression isn't particular good at because Regular Expression struggles with:

1. Operator precedence – i.e. "*/" binds *harder* than "+-"
2. Parenthesis matching – it's difficult to express a regular expression that handles nested parenthesizes
3. Poor error message when the user types a faulty expression

A parser combinator framework has no problems with this.

In this sample we use MicroParser to parse an expression such as the one above and the result is an Expression tree (System.Linq.Expression). This expression tree is then compiled into a delegate and executed. The variables "x" and "y" will be provided as members of a dictionary.

Let's start with some basic parsers:

```
// Define a parameter expression that represent a dictionary, this dictionary
// will contain the variable values
var inputParameter = Expression.Parameter (
   typeof (IDictionary<string, double>),
   "input"
   );

Func<string, Parser<Empty>> p_str = CharParser.SkipString;

var p_spaces = CharParser.SkipWhiteSpace ();

// Parse a double and map it into a ConstantExpression
var p_value = CharParser.Double ().Map (d =>(Expression)Expression.Constant (d));
// Parse a identifier and map it into a Expression that uses FindVariableValue
// to locate a value based on the identifier
var p_variable = CharParser
   .ManyCharSatisfy2 (
      CharSatisfy.Letter,
      CharSatisfy.LetterOrDigit,
      minCount: 1
   )
   .Map (identifier => (Expression) Expression.Call (
      null,
      s_findVariableValue,
      inputParameter,
      Expression.Constant (identifier.ToString ()))
   );
```

It's time to define the term parser; that is a value (i.e. 3.14), a variable (i.e. "x") or a sub expression (i.e. "(x + 3)"). As the parser here will need to call itself recursively if it's a sub expression we will have to use a special type of parser, the redirect parser.

```
var p_astRedirect = Parser.Redirect<Expression> ();

// p_ast is the complete parser (AST = Abstract Syntax Tree)
var p_ast = p_astRedirect.Parser;

// Choice applies each parser consequentially until one matches.
var p_term = Parser.Choice (
   p_ast.Between (p_str ("(").KeepLeft (p_spaces), p_str (")")),
   p_value,
   p_variable
   ).KeepLeft (p_spaces);
```

Complete the parser

```
// p_level is a support parser generator
// it accepts a parser it will apply on the input separated by the operators
// in the ops parameter
Func<Parser<Expression>, string, Parser<Expression>> p_level =
   (parser, ops) => parser.Chain (
      CharParser.AnyOf (ops , minCount:1, maxCount:1).KeepLeft (p_spaces),
      (left, op, right) =>
         Expression.MakeBinary (OperatorToExpressionType (op), left, right)
      );

// By splitting */ and +- like this we ensure */ binds _harder_
var p_lvl0 = p_level (p_term, "*/");
var p_lvl1 = p_level (p_lvl0, "+-");

// This completes the parser
p_astRedirect.ParserRedirect = p_lvl1;
```

The parser p_ast will now parse the following expression:

```
2*(x + 1) + y + 3
```

And if X = 1 and Y = 2 the resulting delegate should produce the value 9.

JSON parsing sample

This sample is in JsonSerializer.cs in the project MicroParser.Json.

JSON is getting more and more popular as an alternative to XML. While .NET has excellent XML tools it's somewhat lacking in JSON support.

The JSON specification (www.json.org) is a small grammar that can be implemented with the help of a parser combinator such as MicroParser or FParsec.

Due to JSONs dynamic nature it fits very well together with languages such as JavaScript or Groovy. In .NET4 we got something called ExpandoObject which also fits very well together with JSON:

```
// ExpandoObject sample
// What's cool is that ExpandoObjects properties are bindable using WPF bindings
dynamic expando         = new ExpandoObject ();
expando.Test            = "This is a new property called Test";
expando.AnotherProperty = 3.14;

IDictionary<string, object> expandoAsDictionary = expando;
foreach (var kv in expandoAsDictionary)
{
   Console.WriteLine ("{0} - {1}", kv.Key, kv.Value);
}
```

In this sample we will unserialize a JSON string into an ExpandoObject for fun and fortune.

Some simple parsers to parse basic JSON tokens:

```
Func<char, Parser<Empty>> p_char = CharParser.SkipChar;
Func<string, Parser<Empty>> p_str = CharParser.SkipString;
var p_spaces = CharParser.SkipWhiteSpace ();

var p_null = p_str ("null").Map (null as object);
var p_true = p_str ("true").Map (true as object);
var p_false = p_str ("false").Map (false as object);

var p_number = CharParser.Double ().Map (d => d as object);
```

The string value parser is a bit more complicated as JSON strings can be \-escaped:

```
const string simpleEscape     = "\"\\/bfnrt";
const string simpleEscapeMap  = "\"\\/\b\f\n\r\t";
Debug.Assert (simpleEscape.Length == simpleEscapeMap.Length);

var simpleSwitchCases = simpleEscape
   .Zip (
      simpleEscapeMap,
      (l, r) => Tuple.Create (
         l.ToString (),
         Parser.Return (new StringPart (r))
      )
   );

var otherSwitchCases =
   new[]
      {
         Tuple.Create (
            "u",
            CharParser
               .Hex (minCount: 4, maxCount: 4)
               .Map (ui => new StringPart ((char) ui)))
      };

var switchCases = simpleSwitchCases.Concat (otherSwitchCases).ToArray ();

var p_escape = Parser.Switch (
   Parser.SwitchCharacterBehavior.Consume,
   switchCases
   );

var p_string = Parser
   .Choice (
      CharParser
         .NoneOf ("\\\"", minCount:1)
         .Map (ss => new StringPart (ss.Position, ss.Length)),
      CharParser.SkipChar ('\\').KeepRight (p_escape))
   .Many ()
   .Between (
      p_char ('"'),
      p_char ('"')
      )
   .CombineStringParts ();
```

Now we are ready to define the complete JSON value parser:

```
var p_array_redirect    = Parser.Redirect<object> ();
var p_object_redirect   = Parser.Redirect<object> ();

var p_array             = p_array_redirect.Parser;
var p_object            = p_object_redirect.Parser;

// Parser.Switch is used as we can tell by looking at the first character which
// parser to use
var p_value = Parser
   .Switch (
      Parser.SwitchCharacterBehavior.Leave,
      Tuple.Create ("\"", p_string),
      Tuple.Create ("0123456789", p_number),
      Tuple.Create ("{", p_object),
      Tuple.Create ("[", p_array),
      Tuple.Create ("t", p_true),
      Tuple.Create ("f", p_false),
      Tuple.Create ("n", p_null)
      )
   .KeepLeft (p_spaces);
```

The array parser:

```
var p_elements = p_value.Array (p_char (',').KeepLeft (p_spaces));

p_array_redirect.ParserRedirect = p_elements.Between (
   p_char ('[').KeepLeft (p_spaces),
   p_char (']')
   )
   .Map (objects => objects as object);
```

The object parser:

```
var p_member = Parser.Group (
   p_string.KeepLeft (p_spaces),
   p_char (':').KeepLeft (p_spaces).KeepRight (p_value)
   );

var p_members = p_member.Array (p_char (',').KeepLeft (p_spaces));

p_object_redirect.ParserRedirect =
   p_members
      .Between (
         p_char ('{').KeepLeft (p_spaces),
         p_char ('}')
         )
      .Map (values =>
               {
                  IDictionary<string, object> exp = new ExpandoObject ();
                  foreach (var value in values)
                  {
                     exp.Add (value.Item1.ToString (), value.Item2);
                  }

                  return exp as object;
               });
```

The complete parser ensures leading whitespace is consumed:

```
s_parser = p_spaces.KeepRight (p_value);
```

This parser will now be able to parse JSON data such as below into an ExpandoObject:

```
{
   "Test": "This is a new property called Test",
   "AnotherProperty": 3.14,
   "AnArray": ["GML", "XML"]
}
```