4 Tutorial

This tutorial introduces you to the basic concepts of FParsec. Our goal is to give you an intuition for how you can build parser applications using the FParsec library. We will only cover the basic ideas and only cursorily explore FParsec's API, but hopefully we will cover enough ground to enable you to further explore FParsec with the help of the user's guide, the API reference and the sample parsers in the `Samples` folder.

A Japanese translation of this tutorial by Gab_km is available here.
A Russian translation of this tutorial by Dmitry Vlasov is available here.

Contents

4.1 Preliminaries

FParsec is built as two DLLs: `FParsec.dll` and `FParsecCS.dll`. To use FParsec in your project, you can either let NuGet install one of the NuGet packages, or you can build the two FParsec DLLs from source. The easiest way to build FParsec from source is using the Visual Studio solution files in the `Build/VS11` folder of the source code package. Any project that uses FParsec has to reference both DLLs. See Download and Installation for more details.

All FParsec types and modules are declared in the `FParsec` namespace. This namespace contains some basic classes (such as `CharStream` and `Reply`) and four F# modules, namely

- `Primitives`, containing basic type definitions and parser combinators,
- `CharParsers`, containing parsers for chars, strings and numbers, and functions for applying parsers to input streams,
- `Error`, containing types and helper functions for creating, processing and formatting parser error messages,
- `StaticMapping`, containing functions for compiling static key-value mappings into optimized functions.

All code snippets in this tutorial assume that you've opened the `FParsec` namespace:

```
open FParsec
```

Opening the `FParsec` namespace also automatically opens the `Primitives`, `CharParsers` and `Error` modules.

Note
All code snippets in this tutorial are contained in the `Samples/Tutorial` project. Having this project open while reading the tutorial can be quite helpful. For example, you can hover the mouse over an identifier to get an Intellisense popup with the inferred type. And if you're curious how a library function is implemented, you can click the *Go to definition* context menu option to view its source code.

4.2 Parsing a single float

Parsing input with FParsec involves two steps:

1. building a parser and
2. applying the parser to the input.

Let's start with a simple example: parsing a single floating-point number in a string.

In this case the first step, building the parser, is trivial, because the `CharParsers` module already comes with a built-in float parser:

```
val pfloat: Parser<float,'u>
```

The generic type `Parser<'Result,'UserState>` is the type of all parsers in FParsec. If you follow the hyperlink into the reference, you'll see that `Parser` is a type abbreviation for a function type. However, at this point we don't need to go into the details of the `Parser` type. It's enough to note that the first type argument represents the type of the parser result. Thus, in the case of `pfloat` the type tells us that if the parser succeeds it returns a floating-point number of type `float`. We won't use a "user state" in this tutorial, so you can just ignore the second type argument for the time being.

To apply the `pfloat` parser to a string, we can use the `run` function from the `CharParsers` module:

```
val run: Parser<'Result, unit> -> string -> ParserResult<'Result,unit>
```

`run` is the simplest function out of several provided by the `CharParsers` module for running parsers on input. Other functions allow you, for example, to run parsers directly on the contents of a file or a `System.IO.Stream`.

`run` applies the parser passed as the first argument to the string passed as the second argument and returns the return value of the parser in form of a `ParserResult` value. The `ParserResult` type is a discriminated union type with the two cases: `Success` and `Failure`. In case the parser succeeds, the `ParserResult` value contains the result value, otherwise it contains an error message.

To simplify testing we write a little helper function that prints the result value or error message:

```
let test p str =
    match run p str with
    | Success(result, _, _)   -> printfn "Success: %A" result
    | Failure(errorMsg, _, _) -> printfn "Failure: %s" errorMsg
```

With this helper function in place, we can test `pfloat` by executing

```
test pfloat "1.25"
```

which produces the output

```
Success: 1.25
```

Testing `pfloat` with a number literal that has an invalid exponent

```
test pfloat "1.25E 3"
```

yields the error message

```
Failure: Error in Ln: 1 Col: 6
1.25E 3
     ^
Expecting: decimal digit
```

4.3 Parsing a float between brackets

Implementing parsers with FParsec typically means combining higher-level parsers from lower-level ones. You start with the parser primitives provided by the library and then successively combine these into higher-level parsers until you finally have a single parser for the complete input.

In the following sections we will illustrate this approach by discussing various sample parsers that build on each other. In this section we will begin with a very simple parser for a floating-point number between brackets:

```
let str s = pstring s
let floatBetweenBrackets = str "[" >>. pfloat .>> str "]"
```

Note

If you're trying to compile this or another code snippet and you get a compiler error mentioning F#'s "value restriction", please see section 4.10.

The definition of `str` and `floatBetweenBrackets` involves three library functions that we haven't yet introduced: `pstring`, `>>.` and `.>>`.

The function

```
val pstring: string -> Parser<string,'u>
```

takes a string as the argument and returns a parser for that string. When this parser is applied to an input stream it checks whether the following chars in the input stream match the given string. If the chars match the complete string, the parser consumes them, i.e. skips over them. Otherwise it fails without consuming any input. When the parser succeeds, it also returns the given string as the parser result, but since the string is a constant, you'll rarely make use of the result.

The `pstring` function isn't named `string` because otherwise it would hide the built-in F# function `string`. In general, parser names in FParsec that would otherwise conflict with built-in F# function names are prefixed by a single p char. `pfloat` is another example for this naming convention.

To save a few keystrokes we abbreviate `pstring` as `str`. So, for instance, `str "["` is a parser that skips over the char `'['`.

The binary operators `>>.` and `.>>` have the following types:

```
val (>>.): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'b,'u>
val (.>>): Parser<'a,'u> -> Parser<'b,'u> -> Parser<'a,'u>
```

As you can see from these signatures, both operators are parser combinators that construct a new parser from the two argument parsers. The parser `p1 >>. p2` parses `p1` and `p2` in sequence and returns the

result of `p2`. The parser `p1 .>> p2` also parses `p1` and `p2` in sequence, but it returns the result of `p1` instead of `p2`. In each case the point points to the side of the parser whose result is returned. By combining both operators in `p1 >>. p2 .>> p3` we obtain a parser that parses `p1`, `p2` and `p3` in sequence and returns the result from `p2`.

Note
With the somewhat imprecise wording "parses `p1` and `p2` in sequence" we actually mean: The parser `p1` is applied to the input and if `p1` succeeds then `p2` is applied to the remaining input; in case any of the two element parsers fails, the aggregate parser immediately propagates the error message.

In the documentation for FParsec we often use expressions such as "parses `p`" or "parses an occurrence of `p`" instead of the technically more accurate "applies the parser `p` to the remaining input and if `p` succeeds …", hoping that the exact meaning is obvious from the context.

The following tests show that `floatBetweenBrackets` parses valid input as expected and produces informative error messages when it encounters invalid input:

```
> test floatBetweenBrackets "[1.0]";;
Success: 1.0

> test floatBetweenBrackets "[]";;
Failure: Error in Ln: 1 Col: 2
[]
 ^
Expecting: floating-point number

> test floatBetweenBrackets "[1.0";;
Failure: Error in Ln: 1 Col: 5
[1.0
    ^
Note: The error occurred at the end of the input stream.
Expecting: ']'
```

4.4 Abstracting parsers

One of FParsec's greatest strengths is the ease with which you can define your own parser abstractions.

Take for instance the `floatBetweenBrackets` from the previous section. If you intend to also parse other elements between strings, you could define your own specialized combinator for this purpose:

```
let betweenStrings s1 s2 p = str s1 >>. p .>> str s2
```

You could then define `floatInBrackets` and other parsers with the help of this combinator:

```
let floatBetweenBrackets = pfloat |> betweenStrings "[" "]"
let floatBetweenDoubleBrackets = pfloat |> betweenStrings "[[" "]]"
```

Note
In case you're new to F#:
`pfloat |> betweenStrings "[" "]"` is just another way to write `betweenStrings "[" "]" pfloat`.

Once you notice that you frequently need to apply a parser between two others, you could go a step further and factor `betweenStrings` as follows:

```
let between pBegin pEnd p  = pBegin >>. p .>> pEnd
let betweenStrings s1 s2 p = p |> between (str s1) (str s2)
```

Actually, you don't need to define `between`, because this is already a built-in FParsec combinator.

These are all trivial examples, of course. But since FParsec is merely an F# library and not some external parser generator tool, there are no limits to the abstractions you can define. You can write functions that take whatever input you need, do some arbitrarily complex computations on the input and then return a special purpose parser or parser combinator.

For example, you could write a function that takes a regular-expression pattern as the input and returns a `Parser` for parsing input conforming to that pattern. This function could use another parser to parse the pattern into an AST and then compile this AST into a special-purpose parser function. Alternatively, it could construct a .NET regular expression from the pattern and then return a parser function that uses FParsec's `CharStream` API to directly apply the regex to the input stream (which is what the built-in `regex` parser actually does).

Another example are extensible parser applications. By storing parser functions in dictionaries or other data structures and defining an appropriate extension protocol, you could allow plugins to dynamically register new parsers or modify existing ones.

The possibilities are really endless. But before you can fully exploit these possibilities, you first need to be familiar with the fundamentals of FParsec.

4.5 Parsing a list of floats

We've already spent three sections on discussing how to parse a single floating-point number, so it's about time we try something more ambitious: parsing a list of floating-point numbers.

Let us first assume that we need to parse a sequence of floating-point numbers in brackets, i.e. text in the following EBNF format: `("[" float "]")*`. Valid input strings in this format are for example: `""`, `"[1.0]"`, `"[2][3][4]"`.

Since we already have a parser for a float between brackets, we only need a way to repeatedly apply this parser to parse a sequence. This is what the `many` combinator is for:

```
val many: Parser<'a,'u> -> Parser<'a list,'u>
```

The parser `many p` repeatedly applies the parser `p` until `p` fails, i.e. it "greedily" parses as many occurrences of `p` as possible. The results of `p` are returned as a list in the order of occurrence.

Some simple tests show that `many floatInBrackets` works as expected:

```
> test (many floatBetweenBrackets) "";;
Success: []
> test (many floatBetweenBrackets) "[1.0]";;
Success: [1.0]
> test (many floatBetweenBrackets) "[2][3][4]";;
Success: [2.0; 3.0; 4.0]
```

If `floatBetweenBrackets` fails *after consuming input*, then the combined parser fails too:

```
> test (many floatBetweenBrackets) "[1][2.0E]";;
Failure: Error in Ln: 1 Col: 9
```

```
[1][2.0E]
        ^
Expecting: decimal digit
```

Note that `many` also succeeds for an empty sequence. If you want to require at least one element, you can use `many1` instead:

```
> test (many1 floatBetweenBrackets) "(1)";;
Failure: Error in Ln: 1 Col: 1
(1)
^
Expecting: '['
```

Tip

If you'd prefer the last error message to be worded in terms of the higher level `floatBetweenBrackets` parser instead of the lower level `str "["` parser, you could use the `<?>` operator as in the following example:

```
> test (many1 (floatBetweenBrackets <?> "float between brackets")) "(1)";;
Failure: Error in Ln: 1 Col: 1
(1)
^
Expecting: float between brackets
```

Please see section 5.8 of the user's guide to learn more about customizing error messages.

If you just want to skip over a sequence and don't need the list of parser results, you could use the optimized combinators `skipMany` or `skipMany1` instead of `many` and `many1`.

Another frequently used combinator for parsing sequences is `sepBy`:

```
val sepBy: Parser<'a,'u> -> Parser<'b,'u> -> Parser<'a list, 'u>
```

`sepBy` takes an "element" parser and a "separator" parser as the arguments and returns a parser for a list of elements separated by separators. In EBNF notation `sepBy p pSep` could be written as `(p (pSep p)*)?`. Similar to `many`, there are several variants of `sepBy`.

With the help of `sepBy` we can parse a more readable list format, where floating-point numbers are separated by a comma:

```
floatList: "[" (float ("," float)*)? "]"
```

Valid input strings in this format are for example: `"[]"`, `"[1.0]"`, `"[2,3,4]"`.

The straightforward implementation of this format is

```
let floatList = str "[" >>. sepBy pfloat (str ",") .>> str "]"
```

Testing `floatList` with valid test strings yields the expected result:

```
> test floatList "[]";;
Success: []
> test floatList "[1.0]";;
Success: [1.0]
> test floatList "[4,5,6]";;
Success: [4.0; 5.0; 6.0]
```

Testing with invalid input shows that `floatList` produces helpful error messages:

```
> test floatList "[1.0,]";;
Failure: Error in Ln: 1 Col: 6
[1.0,]
     ^
Expecting: floating-point number

> test floatList "[1.0,2.0";;
Failure: Error in Ln: 1 Col: 9
[1.0,2.0
        ^
Note: The error occurred at the end of the input stream.
Expecting: ',' or ']'
```

4.6 Handling whitespace

FParsec treats whitespace (spaces, tabs, newlines, etc) just as any other input, so our `floatList` parser can't yet deal with whitespace:

```
> test floatBetweenBrackets "[1.0, 2.0]";;
Failure: Error in Ln: 1 Col: 5
[1.0, 2.0]
    ^
Expecting: ']'
```

If we want the parser to ignore whitespace, we need to make this explicit in the parser definition.

First, we need to define what we want to accept as whitespace. For simplicity we will just use the built-in `spaces` parser, which skips over any (possibly empty) sequence of `' '`, `'\t'`, `'\r'` or `'\n'` chars.

```
let ws = spaces
```

Next, we need to insert the `ws` parser at every point where we want to ignore whitespace. In general it's best to skip whitespace *after* one parses elements, i.e. skip trailing instead of leading whitespace, because that reduces the need for backtracking (which will be explained below). Hence, we insert `ws` at two places to skip over any whitespace after brackets or numbers:

```
let str_ws s = pstring s .>> ws
let float_ws = pfloat .>> ws
let numberList = str_ws "[" >>. sepBy float_ws (str_ws ",") .>> str_ws "]"
```

A simple test shows that `numberList` ignores whitespace:

```
> test numberList @"[ 1 ,
                        2 ] ";;
Success: [1.0; 2.0]
```

If we introduce an error on the second line, we see that `FParsec` automatically keeps track of the line count:

```
> test numberList @"[ 1,
                       2; 3]";;

Failure: Error in Ln: 2 Col: 27
                       2; 3]
                        ^
Expecting: ',' or ']'
```

Our `numberList` parser still doesn't skip leading whitespace, because that's not necessary when we put it together with other parsers that skip all trailing whitespace. If we wanted to parse a whole input stream with only a list of floating-point numbers, we could use the following parser:

```
let numberListFile = ws >>. numberList .>> eof
```

The end-of-file parser `eof` will generate an error if the end of the stream hasn't been reached. This is useful for making sure that the complete input gets consumed. Without the `eof` parser the following test wouldn't produce an error:

```
> test numberListFile " [1, 2, 3] [4]";;
Failure: Error in Ln: 1 Col: 12
 [1, 2, 3] [4]
           ^
Expecting: end of input
```

4.7 Parsing string data

FParsec contains various built-in parsers for chars, strings, numbers and whitespace. In this section we will introduce a few of the char and string parsers. For an overview of all available parsers please refer to the parser overview in the reference.

You've already seen several applications of the `pstring` parser (abbreviated as `str`), which simply skips over a constant string in the input. When the `pstring` parser succeeds, it also returns the skipped string as the parser result. The following example demonstrates this:

```
> test (many (str "a" <|> str "b")) "abba";;
Success: ["a"; "b"; "b"; "a"]
```

In this example we also used the `<|>` combinator to combine two alternative parsers. We'll discuss this combinator in more detail below.

Note
We refer to both `pstring` and `pstring "a"` as "parsers". Strictly speaking, `pstring` is function taking a string argument and returning a `Parser`, but it's more convenient to just refer to it as a (parametric) parser.

When you don't need the result of the `pstring` parser, you can alternatively use the `skipString` parser, which returns the `unit` value `()` instead of the argument string. In this case it doesn't make any difference to performance whether you use `pstring` or `skipString`, since the returned string is a constant. However, for most other built-in parsers and combinators you should prefer the variants with the "skip" name prefix when you don't need the parser result values, because these will generally be faster. If you look at the parser overview, you'll see "skip" variants for many of the built-in parsers and combinators.

If you want to parse a **c**ase **i**nsensitive string constant you can use `pstringCI` and `skipStringCI`. For example:

```
> test (skipStringCI "<float>" >>. pfloat) "<FLOAT>1.0";;
Success: 1.0
```

Frequently one needs to parse string variables whose chars have to satisfy certain criteria. For instance, identifiers in programming languages often need to start with a letter or underscore and then need to continue with letters, digits or underscores. To parse such an identifier you could use the following

parser:

```
let identifier =
    let isIdentifierFirstChar c = isLetter c || c = '_'
    let isIdentifierChar c = isLetter c || isDigit c || c = '_'

    many1Satisfy2L isIdentifierFirstChar isIdentifierChar "identifier"
    .>> ws // skips trailing whitespace
```

Here we have used the `many1Satisfy2L` string parser, which is one of several primitives for parsing strings based on char predicates (i.e. functions that take a char as input and return a boolean value). It parses any sequence of one or more chars (hence the "many1" in the name) whose first char satisfies the first predicate function and whose remaining chars satisfy the second predicate (hence the "Satisfy2"). The string label given as the third argument (hence the "L") is used in error message to describe the expected input.

The following tests show how this parser works:

```
> test identifier "_";;
Success: "_"
> test identifier "_test1=";;
Success: "_test1"
> test identifier "1";;
Failure: Error in Ln: 1 Col: 1
1
^
Expecting: identifier
```

Tip

If you want to parse identifiers based on the Unicode XID syntax, consider using the built-in `identifier` parser.

Many string formats are complicated enough that you need to combine several char and string parser primitives. For example, consider the following string literal format:

```
  stringLiteral: '"' (normalChar|escapedChar)* '"'
  normalChar:    any char except '\' and '"'
  escapedChar:   '\\' ('\\'|'"'|'n'|'r'|'t')
```

A straightforward translation of this grammar to FParsec looks like:

```
let stringLiteral =
    let normalChar = satisfy (fun c -> c <> '\\' && c <> '"')
    let unescape c = match c with
                     | 'n' -> '\n'
                     | 'r' -> '\r'
                     | 't' -> '\t'
                     | c   -> c
    let escapedChar = pstring "\\" >>. (anyOf "\\nrt\"" |>> unescape)
    between (pstring "\"") (pstring "\"")
            (manyChars (normalChar <|> escapedChar))
```

In this example we use several library functions that we haven't yet introduced:

- `satisfy` parses any char that satisfies the given predicate function.
- `anyOf` parses any char contained in the argument string.
- The pipeline combinator `|>>` applies the function on the right side (`unescape`) to the result of the

parser on the left side (`anyOf "\\nrt\""`).

- The choice combinator `<|>` applies the parser on the right side if the parser on the left side fails, so that `normalChar <|> escapedChar` can parse both normal and escaped chars. (We will discuss this operator in more detail two sections below.)
- `manyChars` parses a sequence of chars with the given char parser and returns it as a string.

Let's test the `stringLiteral` parser with a few test inputs:

```
> test stringLiteral "\"abc\"";;
Success: "abc"
> test stringLiteral "\"abc\\\"def\\\\ghi\"";;
Success: "abc"def\ghi"
> test stringLiteral "\"abc\\def\"";;
Failure: Error in Ln: 1 Col: 6
"abc\def"
     ^
Expecting: any char in '\nrt"'
```

Instead of parsing the string literal char-by-char we could also parse it "snippet-by-snippet":

```
let stringLiteral2 =
    let normalCharSnippet = many1Satisfy (fun c -> c <> '\\' && c <> '"')
    let escapedChar = pstring "\\" >>. (anyOf "\\nrt\"" |>> function
                                                            | 'n' -> "\n"
                                                            | 'r' -> "\r"
                                                            | 't' -> "\t"
                                                            | c   -> string c)
    between (pstring "\"") (pstring "\"")
            (manyStrings (normalCharSnippet <|> escapedChar))
```

Here we have used the `manyStrings` combinator, which parses a sequence of strings with the given string parser and returns the strings in concatenated form.

Note

We have to require `normalCharSnippet` to consume at least one char, i.e. use `many1Satisfy` instead of `manySatisfy`. Otherwise `normalCharSnippet` would succeed even if doesn't consume input, `escapedChar` would never be called and `manyStrings` would eventually throw an exception to prevent an infinite loop.

Parsing a string chunk-wise using an optimized parser like `many1Satisfy` is usually a bit faster than parsing it char-wise using `manyChars` and `satisfy`. In this case we can optimize our parser even a bit further – once we realize that two normal char snippets must be separated by at least one escaped char:

```
let stringLiteral3 =
    let normalCharSnippet = manySatisfy (fun c -> c <> '\\' && c <> '"')
    let escapedChar = (* like in stringLiteral2 *)
    between (pstring "\"") (pstring "\"")
            (stringsSepBy normalCharSnippet escapedChar)
```

The `stringsSepBy` combinator parses a sequence of strings (with the first argument parser) separated by other strings (parsed with the second argument parser). It returns all parsed strings, including the separator strings, as a single, concatenated string.

Note that `stringLiteral3` uses `manySatisfy` instead of `many1Satisfy` in its `normalCharSnippet` definition, so that it can parse escaped chars that are not separated by normal chars. This can't lead to an infinite loop because `escapedChar` can't succeed without consuming input.

4.8 Sequentially applying parsers

Whenever you need to apply multiple parsers in sequence and only need the result of one of them, a suitable combination of `>>.` and `.>>` operators will do the job. However, these combinators won't suffice if you need the result of more than one of the involved parsers. In that case you can use the `pipe2`, …, `pipe5` combinators, which apply multiple parsers in sequence and pass all the individual results to a function that computes the aggregate result.

For instance, with the `pipe2` combinator

```
val pipe2: Parser<'a,'u> -> Parser<'b,'u> -> ('a -> b -> 'c) -> Parser<'c,'u>
```

you can construct a parser `pipe2 p1 p2 f` that sequentially applies the two parsers `p1` and `p2` and then returns the result of the function application `f x1 x2`, where `x1` and `x2` are the results returned by `p1` and `p2`.

In the following example we use `pipe2` to parse a product of two numbers:

```
let product = pipe2 float_ws (str_ws "*" >>. float_ws)
                    (fun x y -> x * y)
```

```
> test product "3 * 5";;
Success: 15.0
```

The `pipe2-5` combinators are particularly useful for constructing AST objects. In the following example we use `pipe3` to parse a string constant definition into a `StringConstant` object:

```
type StringConstant = StringConstant of string * string

let stringConstant = pipe3 identifier (str_ws "=") stringLiteral
                           (fun id _ str -> StringConstant(id, str))
```

```
> test stringConstant "myString = \"stringValue\"";;
Success: StringConstant ("myString","stringValue")
```

If you just want to return the parsed values as a tuple, you can use the predefined `tuple2-5` parsers. For instance, `tuple2 p1 p2` is equivalent to `pipe2 p1 p2 (fun x1 x2 -> (x1, x2))`.

The `tuple2` parser is also available under the operator name `.>>.`, so that you can write `p1 .>>. p2` instead of `tuple2 p1 p2`. In the following example we parse a pair of comma separated numbers with this operator:

```
> test (float_ws .>>. (str_ws "," >>. float_ws)) "123, 456";;
Success: (123.0, 456.0)
```

Hopefully you find the *>>-with-1-or-2-dots-notation* intuitive by now.

If you need a `pipe` or `tuple` parser with more than 5 arguments, you can easily construct one using the existing ones. For example, do you have an idea how to define a `pipe7` parser? This footnote gives a possible solution: [1]

4.9 Parsing alternatives

In the section on `Parsing string data` we already shortly introduced the choice combinator `<|>`:

```
val (<|>): Parser<'a,'u> -> Parser<'a,'u> -> Parser<'a,u>
```

This combinator allows you to support multiple alternative input formats at a given input position. For

example, in the above section we used `<|>` to combine a parser for unescaped chars and a parser for escaped chars into a parser that supports both: `normalChar <|> escapedChar`.

Another example that shows how `<|>` works is the following parser for boolean variables:

```
let boolean =     (stringReturn "true"  true)
              <|> (stringReturn "false" false)
```

Here we have also used the `stringReturn` parser, which skips the string constant given as the first argument and, if successful, returns the value given as the second argument.

Testing the `boolean` parser with some inputs yields:

```
> test boolean "false";;
Success: false
> test boolean "true";;
Success: true
> test boolean "tru";;
Failure: Error in Ln: 1 Col: 1
tru
^
Expecting: 'false' or 'true'
```

The behaviour of the `<|>` combinator has two important characteristics:

- `<|>` only tries the parser on the right side if the parser on the left side fails. It does *not* implement a longest match rule.
- However, it only tries the right parser if the left parser fails *without consuming input*.

A consequence of the second point is that the following test fails because the parser on the left side of `<|>` consumes whitespace before it fails:

```
> test ((ws >>. str "a") <|> (ws >>. str "b")) " b";;
Failure: Error in Ln: 1 Col: 2
 b
 ^
Expecting: 'a'
```

Fortunately, we can easily fix this parser by factoring out `ws`:

```
> test (ws >>. (str "a" <|> str "b")) " b";;
Success: "b"
```

If you're curious why `<|>` behaves this way and how you can handle situations where you need `<|>` to try the alternative parser even if the first parser fails after consuming input, please see section 5.6 and section 5.7 in the user's guide.

If you want to try more than two alternative parsers, you can chain multiple `<|>` operators, like in `p1 <|> p2 <|> p3 <|> ...`, or you can use the `choice` combinator, which accepts a sequence of parsers as the argument, like in `choice [p1; p2; p3; ...]`.

4.10 F#'s value restriction

When you start writing your own parsers with FParsec or try to compile some individual code snippets from above, you'll come across a compiler issue that often causes some head-scratching among new users of F# and FParsec: the *value restriction*. In this section we'll explain the value restriction and how you can handle it in your FParsec programs.

Note
If you find the discussion in this section too technical for the moment, just skip to the next section and come back later when you actually see a compiler message mentioning "value restriction" for the first time.

F#'s value restriction is the reason that the following code snippet does not compile

```
open FParsec
let p = pstring "test"
```

even though the following snippet compiles without a problem[2]:

```
open FParsec
let p = pstring "test"
run p "input"
```

The compiler error generated for the first sample is the following:

```
error FS0030: Value restriction.
The value 'p' has been inferred to have generic type
    val p : Parser<string,'_a>
Either make the arguments to 'p' explicit or,
if you do not intend for it to be generic, add a type annotation.
```

When you work with FParsec you'll sooner or later see this or similar error messages, in particular if you work with the interactive console prompt. Fortunately, this kind of error is usually easy to workaround.

The problem with the first snippet above is that the F# compiler infers the `p` value to have an unresolved generic type, although F# doesn't permit a generic value in this situation. The return type of the `pstring` function is `Parser<string,'u>`, where the type parameter `'u` represents the type of the `CharStream` user state. Since there is nothing in the first snippet that constrains this type parameter, the compiler infers the type `Parser<string,'_a>` for the parser value `p`, with `'_a` representing an unresolved type parameter.

In the second snippet this problem doesn't occur, because the use of `p` as the first argument to the `run` function constrains the user state type. Since `run` only accepts parsers of type `Parser<'t,unit>`, the compiler infers the non-generic type `Parser<string,unit>` for `p`.

This example suggests two ways to handle the value restriction in FParsec programs:

- Either make sure that the type of a parser value is constrained to a non-generic type by subsequent uses of this parser value *in the same compilation unit*,
- or provide an explicit type annotation to manually constrain the type of the parser value (usually, a few type annotations in key locations are enough for a whole parser module).

Often it is convenient to define some type abbreviations like the following

```
type UserState = unit // doesn't have to be unit, of course
type Parser<'t> = Parser<'t, UserState>
```

With such an abbreviation in place, type annotations become as simple as

```
let p : Parser<_> = pstring "test"
```

Of course, constraining the type of a parser value to a non-generic type is only a solution if you don't actually need a generic type. If you do need a generic value, you'll have to apply other techniques, as they are for example explained in the F# reference or in a blog entry by Dmitry Lomov. However, FParsec `Parser` values (not parametric parser functions) are usually only used in the context of a specific

parser application with a fixed user state type. In that situation constraining the type is indeed the appropriate measure to avoid a value restriction error.

4.11 Parsing JSON

Now that we have discussed the basics of FParsec we are well prepared to work through a real world parser example: a JSON parser.

JSON (JavaScript Object Notation) is a text-based data interchange format with a simple and lightweight syntax. You can find descriptions of the syntax on json.org and in RFC 4626.

In many applications one only has to deal with JSON files describing one particular kind of object. In such a context it sometimes can be appropriate to write a specialized parser just for that specific kind of JSON file. In this tutorial, however, we will follow a more general approach. We will implement a parser that can parse any general JSON file into an AST, i.e. an intermediate data structure describing the contents of the file. Applications can then conveniently query this data structure and extract the information they need. This is an approach comparable to that of XML parsers which build a data structure describing the document tree of an XML document. The great advantage of this approach is that the JSON parser itself becomes reusable and the document specific parsing logic can be expressed in the form of simple functions processing the AST of the JSON document.

The natural way to implement an AST in F# is with the help of a discriminated union type. If you look at the JSON specification, you can see that a JSON value can be a string, a number, a boolean, null, a comma-separated list of values in square brackets, or an object with a sequence of key-value pairs in curly brackets.

In our parser we will use the following union type to represent JSON values:

```
type Json = JString of string
          | JNumber of float
          | JBool   of bool
          | JNull
          | JList   of Json list
          | JObject of Map<string, Json>
```

Here we've chosen the F# `list` type to represent a sequence of values and the `Map` type to represent a sequence of key-value pairs, because these types are particularly convenient to process in F#.[3] Note that the `Json` type is recursive, since both `JList` and `JObject` values can themselves contain `Json` values. Our parser will have to reflect this recursive structure.

Tip

If you're new to FParsec and have a little time, it would be a good exercise to try to implement the JSON parser on your own (with the help of the reference documentation). This tutorial already covered almost everything you need and the JSON grammar is simple enough that this shouldn't take too much time. Of course, you can always peek at the implementation below if you get stuck.

We start the actual parser implementation by covering the simple `null` and boolean cases:

```
let jnull  = stringReturn "null" JNull
let jtrue  = stringReturn "true"  (JBool true)
let jfalse = stringReturn "false" (JBool false)
```

Handling the number case is just as simple, because the JSON number format is based on the typical floating-point number format used in many programming languages and hence can be parsed with FParsec's built-in `pfloat` parser:

```
let jnumber = pfloat |>> JNumber
```

(Note that F# allows us to pass the object constructor `JNumber` as a function argument.)

If you compare the precise number format supported by `pfloat` with that in the JSON spec, you'll see that `pfloat` supports a superset of the JSON format. In contrast to the JSON format the `pfloat` parser also recognizes `NaN` and `Infinity` values, accepts a leading plus sign, accepts leading zeros and even supports the hexadecimal float format of Java and C99. Depending on the context this behaviour can be considered a feature or a limitation of the parser. For most applications it probably doesn't matter, and the JSON RFC clearly states that a JSON parser may support a superset of the JSON syntax. However, if you'd rather only support the exact JSON number format, you can implement such a float parser rather easily based on the configurable `numberLiteral` parser (just have a look at how this is currently done in the `pfloat` source).

The JSON string format takes a little more effort to implement, but we've already parsed a similar format with the `stringLiteral` parsers in section 4.7, so we can just adapt one of those parsers for our purpose:

```
let str s = pstring s

let stringLiteral =
    let escape =  anyOf "\"\\/bfnrt"
                  |>> function
                      | 'b' -> "\b"
                      | 'f' -> "\u000C"
                      | 'n' -> "\n"
                      | 'r' -> "\r"
                      | 't' -> "\t"
                      | c   -> string c // every other char is mapped to itself

    let unicodeEscape =
        /// converts a hex char ([0-9a-fA-F]) to its integer number (0-15)
        let hex2int c = (int c &&& 15) + (int c >>> 6)*9

        str "u" >>. pipe4 hex hex hex hex (fun h3 h2 h1 h0 ->
            (hex2int h3)*4096 + (hex2int h2)*256 + (hex2int h1)*16 + hex2int h0
            |> char |> string
        )

    let escapedCharSnippet = str "\\" >>. (escape <|> unicodeEscape)
    let normalCharSnippet  = manySatisfy (fun c -> c <> '"' && c <> '\\')

    between (str "\"") (str "\"")
            (stringsSepBy normalCharSnippet escapedCharSnippet)

let jstring = stringLiteral |>> JString
```

`stringLiteral` parses string literals as a sequence of normal char snippets separated by escaped char snippets. A normal char snippet is any sequence of chars that does not contain the chars `'"'` and `'\\'`. An escaped char snippet consists of a backslash followed by any of the chars `'\\'`, `'\"'`, `'/'`, `'b'`, `'f'`, `'n'`, `'r'`, `'t'`, or an Unicode escape. An Unicode escape consists of an `'u'` followed by four hex chars

representing an UTF-16 code point.

The grammar rules for JSON lists and objects are recursive, because any list or object can contain itself any kind of JSON value. Hence, in order to write parsers for the list and object grammar rules, we need a way to refer to the parser for any kind of JSON value, even though we haven't yet constructed this parser. Like it is so often in computing, we can solve this problem by introducing an extra indirection:

```
let jvalue, jvalueRef = createParserForwardedToRef<Json, unit>()
```

As you might have guessed from the name, `createParserForwardedToRef` creates a parser (`jvalue`) that forwards all invocations to the parser in a reference cell (`jvalueRef`). Initially, the reference cell holds a dummy parser, but since the reference cell is mutable, we can later replace the dummy parser with the actual value parser, once we have finished constructing it.

The JSON RFC sensibly only permits spaces, (horizontal) tabs, line feeds and carriage returns as whitespace characters, which allows us to use the built-in `spaces` parser for parsing whitespace:

```
let ws = spaces
```

Both JSON lists and objects are syntactically represented as a comma-separated lists of "elements" between brackets, where whitespace is allowed before and after any bracket, comma and list element. We can conveniently parse such lists with the following helper function:

```
let listBetweenStrings sOpen sClose pElement f =
    between (str sOpen) (str sClose)
            (ws >>. sepBy (pElement .>> ws) (str "," >>. ws) |>> f)
```

This function takes four arguments: an opening string, a closing string, an element parser and a function that is applied to the parsed list of elements.

With the help of this function we can define the parser for a JSON list as follows:

```
let jlist   = listBetweenStrings "[" "]" jvalue JList
```

JSON objects are lists of key-value pairs, so we need a parser for a key-value pair:

```
let keyValue = stringLiteral .>>. (ws >>. str ":" >>. ws >>. jvalue)
```

(Remember, the points on both sides of `.>>.` indicate that the results of the two parsers on both sides are returned as a tuple.)

By passing the `keyValue` parser to `listBetweenStrings` we obtain a parser for JSON objects:

```
let jobject = listBetweenStrings "{" "}" keyValue (Map.ofList >> JObject)
```

Having defined parsers for all the possible kind of JSON values, we can combine the different cases with a `choice` parser to obtain the finished parser for JSON values:

```
do jvalueRef := choice [jobject
                        jlist
                        jstring
                        jnumber
                        jtrue
                        jfalse
                        jnull]
```

The `jvalue` parser doesn't accept leading or trailing whitespace, so we need to define our parser for complete JSON documents as follows:

```
let json = ws >>. jvalue .>> ws .>> eof
```

This parser will try to consume a complete JSON input stream and, if successful, will return a `Json` AST of the input as the parser result

And that's it, we're finished with our JSON parser. If you want to try this parser out on some sample input, please take a look at the JSON project in the `Samples` folder.

4.12 What now?

If this tutorial has whet your appetite for a more in-depth introduction to FParsec, just head over to the user's guide. If you can't wait to write your own parser, then bookmark the parser overview page, maybe take a short look at the example parsers in the `Samples` folder and just start hacking. You can always consult the user's guide at a later point should you get stuck somewhere.

Footnotes:

[1]
```
let pipe7 p1 p2 p3 p4 p5 p6 p7 f =
    pipe4 p1 p2 p3 (tuple4 p4 p5 p6 p7)
          (fun x1 x2 x3 (x4, x5, x6, x7) -> f x1 x2 x3 x4 x5 x6 x7)
```

[2] Assuming you referenced the two FParsec DLLs.

[3] If you need to parse huge sequences and objects, it might be more appropriate to use an array and dictionary for JList and JObject respectively.