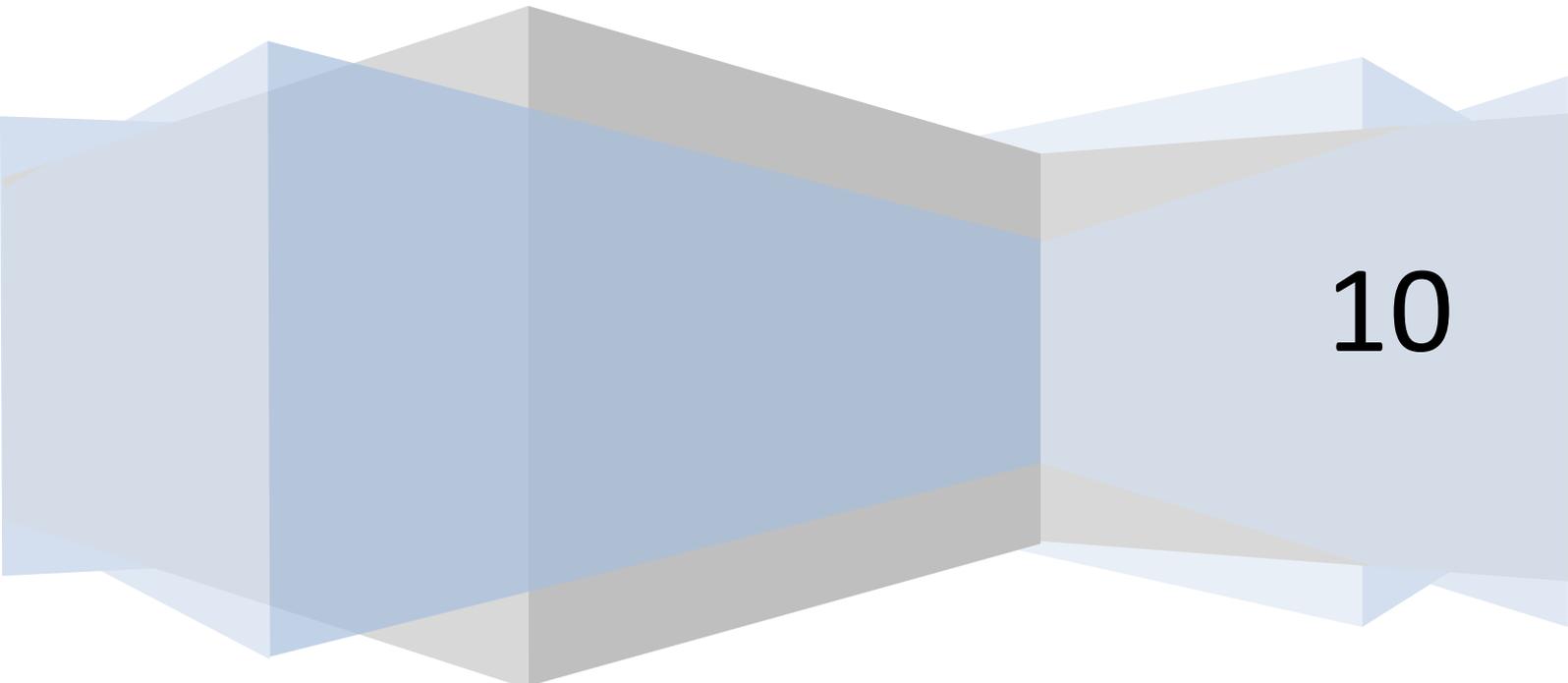


# Parser Combinators in C#

A Monadic Parser Combinator Framework in C#

Thomas Krause



10

## Inhaltsverzeichnis

Einleitung.....	3
Lookahead und Mehrdeutigkeit.....	5
Theorie .....	6
Was ist ein Parser? .....	6
Parserprimitive .....	6
Return(result) .....	6
Fail .....	7
Accept.....	7
Kombinator-Primitive .....	7
Or.....	7
Then.....	7
Probleme und Einschränkungen des kombinatorischen Parsens .....	10
Implementierung.....	13
Spracherweiterungen von C# 3.0 .....	13
Projektstruktur .....	16
State.....	16
Combinator.....	16
Parser.....	16
Helper .....	17
Beispiele .....	17
Fazit .....	18
Quellen und weiterführende Literatur.....	19
Monadic Parser Combinators using C# 3.0, LukeH's WebLog.....	19
Monadic parser combinators (I-VII), Brian McNamara .....	19
The X-SAIGA Project, Hafiz, R. and Frost, R.....	19
FParsec - A Parser Combinator Library for F# .....	19

## Einleitung

Die Technik des kombinatorischen Parsens ist ein Top-Down-Ansatz und ähnelt damit dem rekursiven Abstieg. Wie beim Ansatz des rekursiven Abstiegs, werden Grammatiken bei Parser Combinators ebenfalls direkt in einer regulären (in der Regel aber funktionalen) Programmiersprache beschrieben, statt auf spezielle Grammatikbeschreibungen auszuweichen.

Das Grundprinzip beim kombinatorischen Parsen ist es kleine einfache Parser zu immer komplexeren Parsern zusammensetzen (zu kombinieren). Grundvoraussetzung dafür sind Higher-Order-Funktionen, also Funktionen die Funktionen als Parameter entgegennehmen oder andere Funktionen als Ergebnis zurückgeben.

Das kombinatorische Parsen ist ein Top-Down-Ansatz und ähnelt in gewisser Hinsicht der Methode des rekursiven Abstiegs. So werden beim kombinatorischen Parsen ebenfalls Funktionen verwendet um den eigentlichen Parse Vorgang durchzuführen und die Grammatikbeschreibung liegt direkt als ausführbares Programm vor, so dass keine separate Sprache hierfür benötigt wird.

Auch beim rekursiven Abstieg werden verschiedene Parse-Methoden miteinander „kombiniert“.

```
IfStatement parseIfStatement()
{
    accept(IF);
    var e = parseExpression();
    accept(THEN);
    var s = parseStatement();

    return new IfStatement(e, s);
}
```

Im obigen Beispiel für die Methode des rekursiven Abstiegs ruft die Parse-Methode `parseIfStatement` intern die beiden Parse-Methoden `parseExpression` und `parseStatement`, sowie die Methode `Accept` zum Lesen eines Tokens auf. Betrachtet man `parseIfStatement`, `parseExpression`, `parseStatement` und `Accept` als separate „Parser“, so kann man sagen, dass `parseIfStatement` ein Parser ist, der aus einer Kombination der Parser `parseExpression`, `Accept` und `parseStatement` besteht. Die Art der Kombination ist in diesem Falle eine *Sequenz*. Komplexere Parse-Methoden können durchaus auch eine andere Arten der Kombination enthalten. Zum Beispiel *Alternative* nach dem Schema: `if (lookahead() == XYZ) ...`

Beim kombinatorischen Parsen geschieht die Kombination verschiedener Parser nicht, wie beim rekursiven Abstieg, implizit durch den Programmfluss, sondern vielmehr explizit durch den Aufruf von Kombinatormethoden.

Ein Kombinator ist also in diesem Kontext eine Methode die ein oder mehrere Parser als Argument entgegennimmt und einen neuen „kombinierten“ Parser zurückgibt.

Beispiel:

```
var statement = ifStatement.Or(functionCall);
```

`ifStatement` und `functionCall` sind Parser (Funktionen), die Methode `Or` ist ein Kombinator (Alternative) der die beiden Parser zu einem neuen Parser `statement` verknüpft.

Gegenüber dem rekursiven Abstieg mag diese Art einen Parser zu bauen zunächst umständlich erscheinen, sie bietet aber einige erhebliche Vorteile.

Während man beim rekursiven Abstieg zwar sehr komplexe Parser bauen kann, so sind die Möglichkeiten die einzelnen Parse-Methoden zu kombinieren bzw. wiederzuverwenden durch die Kontrollstrukturen (z.B. `if`, `while` oder *Sequenz*) der jeweiligen Programmiersprache beschränkt. Beim kombinatorischen Parsen können dagegen sowohl Parser als auch die Kombinatoren nahezu beliebig komplex sein und lassen sich zudem besser auf die spezifischen Anforderungen beim Parsen anpassen. Nehmen wir zum Beispiel folgenden Code um eine kommagetrennte Liste von Zahlen mit dem Verfahren des rekursiven Abstiegs zu parsen:

```
int[] parseNumberList()
{
    List<int> numberList = new List<int>();

    numberList.Add(parseNumber);
    while (lookahead() != COMMA)
    {
        accept(COMMA);
        numberList.Add(parseNumber);
    }

    return numberList.ToArray();
}
```

Das gleiche Ergebnis könnte beim kombinatorischen Parsen durch einen speziellen Kombinator sehr viel kürzer und zudem noch deutlich lesbarer ausgedrückt werden:

```
var parseNumberList = parseNumber.RepeatWithSeparator(COMMA);
```

Der Kombinator kann dabei ebenso wie ein Parser wiederverwendet werden:

```
var parseStatementList = parseStatement.RepeatWithSeparator(SEMICOLON);
```

Auch wenn es aussieht als würden bei den vorangegangenen Beispielen für kombinatorische Parser Objektinstanzen verwendet (`var parseStatementList = ...`), so handelt es sich tatsächlich um reguläre Funktionen die direkt aufgerufen werden können:

```
var result = parseStatementList(...);
```

Diese Eigenschaft Funktionen je nach Bedarf wie Objekte zu behandeln ermöglicht vereinfacht viele Probleme die beim rekursiven Abstieg entstehen können. So kann man zum Beispiel vergleichsweise trivial ein Rekursionslimit für alle Parser einführen, so dass auch linksrekursive Grammatiken unterstützt werden (mit bestimmten Einschränkungen, dazu später mehr).

## Lookahead und Mehrdeutigkeit

Ein fundamentaler Unterschied vom kombinatorischen Parsern (sowie sie hier implementiert sind) zu herkömmlichen Parsern ist, dass beim Parsen in der Regel kein look-ahead verwendet wird. Vielmehr arbeiten kombinatorische Parser nach dem Verfahren Trial-and-Error. Statt mit Hilfe eines look-aheads zunächst eine von möglicherweise vielen Alternativen auszuwählen, wird beim kombinatorischen Parsen jede mögliche Alternative untersucht (angewendet) und erst im Anschluss geprüft welche Alternative(n) zu einem Ergebnis geführt hat/haben und welche nicht. Dabei ist es durchaus legitim wenn mehr als eine Alternative zu einem Ergebnis führt, die Grammatik als mehrdeutig ist. Das Ergebnis eines Parse-Vorgangs ist also immer eine Liste von allen möglichen Ableitungen die die Grammatik für eine bestimmte Eingabe erlaubt. Im Falle einer eindeutigen Grammatik besteht diese Liste folglich am Ende nur aus einem Element. Zwischenergebnisse können aber auch in diesem Fall zunächst mehrdeutig sein.

Während zur Programmierung eines Compilers für eine Programmiersprache in der Regel eindeutige Grammatiken wünschenswert sind, so gibt es durchaus Anwendungsgebiete in denen mehrdeutige Grammatiken und die zugehörigen Parser essenziell sind. Ein Beispiel ist die Verarbeitung natürlicher Sprache, die oftmals sehr mehrdeutig ist. Dazu ein kurzer Beispielsatz zur Verdeutlichung:

„Wie viele Deutsche vertragen Schweizer Universitäten?“

Wer verträgt hier wen? Subjekt, Prädikat, Objekt oder Objekt, Prädikat, Subjekt

Die „deutsche Grammatik“ erlaubt also (mindestens) zwei fundamental verschiedene Ableitungsbäume für diesen Satz.

## Theorie

Nachdem die grundsätzliche Funktionsweise des kombinatorischen Parsens hoffentlich ein wenig klar geworden ist, können wir uns nun ein wenig mit der Theorie des kombinatorischen Parsens beschäftigen.

### Was ist ein Parser?

Ein Parser ( $p$ ) ist in diesem Kontext definiert als eine Funktion die eine Liste von zu parsenden Tokens (**currentstate**) entgegennimmt, null, ein, oder mehrere Tokens aus dieser Eingabe liest bzw. verarbeitet und als Rückgabewert eine Liste von Tupeln der Form (**(Teil) ergebnis, verbleibende Tokens**) produziert. Formaler ausgedrückt:

```
p: currentstate => {(result_1, newstate_1), ...}
```

Jedes Tupel stellt dabei eine mögliche Ableitung bzw. eine Parse-Alternative dar.

Da ein Parser in der Regel nur einen Teil der Eingabe liest, muss die verbleibende Eingabe (**newstate**) für jede Alternative als Teil des Ergebnisses zurückgegeben werden, damit diese z.B. von nachfolgenden Parsern in einer Sequenz verwendet werden kann.

Das (Teil)Ergebnis im Tupel (**result\_x**), stellt das eigentliche Resultat des Parse-Vorgangs dar. Ein Parser für natürliche Zahlen könnte z.B. die gelesene Zahl als Ganzzahltyp (Integer) zurückgeben, ein Parser für Anweisungen einer Programmiersprache dagegen z.B. einen AST-Knoten vom Typ **Statement**.

### Beispiele

#### Parser für einzelne Ziffer

Ziffer: „123“ => {(1, „23“)}

#### Parser für eine Zahl (Parser muss nicht alle Tokens konsumieren)

Zahl: „123 “ => {(1, „23“), (12, „3“), (123, „“)}

#### Mehrdeutige Ableitung (Berechnungsreihenfolge nicht festgelegt)

Ausdruck: „1+2\*3“ => {(9, „“), (7, „“)}

#### Bei erfolglosem Parsen wird eine leere Menge zurückgegeben

Zahl: „abc“ => { }

## Parserprimitive

Prinzipiell bauen alle im Framework vorhandenen Parser auf drei grundlegenden Parsern auf (auch als Parserprimitive bezeichnet). Alle anderen Parser entstehen aus Kombination von ein oder mehrerer dieser Parserprimitive. Die drei Parser sind **Return**, **Fail** und **Accept**.

### Return(result)

Dieser Parser konsumiert keine Eingabe und gibt immer ein bestimmtes Ergebnis zurück. Genauer gesagt, handelt es sich hierbei nicht um einen einzelnen Parser, sondern um eine Fabrikmethode, die für ein bestimmtes Ergebnis einen Parser erzeugt, der dieses Ergebnis zurückgibt.

Beispiel:

```
Return(2): „xyz“ => {(2, „xyz“)}
```

## Fail

Dieser Parser gibt unabhängig von der Eingabe eine leere Menge zurück, oder anders ausgedrückt, der Parser lässt den Parse-Vorgang immer fehlschlagen.

Beispiel:

```
Fail: „xyz“ => { }
```

## Accept

Der Accept Parser konsumiert ein einzelnes Zeichen aus der Eingabe und gibt dieses als Parseergebnis zurück. Bei einer leeren Eingabe wird eine leere Menge als Ergebnis zurückgegeben.

Beispiel:

```
Accept: „xyz“=> {(„x“, „yz“)}
```

```
Accept: „“ => { }
```

## Kombinator-Primitive

Um aus den oben genannten Parsern komplexere Parser zusammensetzen werden nur zwei grundlegende Kombinator-Primitive benötigt. Aus diesen lassen sich fast beliebig komplexe Parser oder auch andere Kombinatoren zusammensetzen.

### Or

Der Or-Kombinator nimmt zwei Parser entgegen und gibt einen neuen Parser zurück der zu einer bestimmten Eingabe die Vereinigung beider Parser-Ergebnismengen zurückgibt. Er ähnelt damit dem Alternativoperator in Grammatiken (|).

```
Char: “123” => {(“1”, “23”)}  
Digit: “123” => {(1, “23”)}  
CharOrDigit = Char.Or(Digit);  
CharOrDigit: “123” => {{{“1”, “23”}, (1, “23”)}}
```

### Then

Der Then-Kombinator entspricht logisch einer Sequenz in einer Grammatik. Er konkateniert also 2 Parser. Die vom ersten Parser nicht konsumierten Token werden an den zweiten Parser als Eingabe übergeben.

Eine Schwierigkeit besteht darin aus beiden Parsern ein gemeinsames Parseergebnis zu produzieren, denn der zweite Parser bekommt als Eingabe lediglich die Restmenge der Tokens, kann aber nicht auf das Parseergebnis des ersten Parsers zurückgreifen.

Beispiel:

```
Zahl: „123“ => {(123, „“)}  
Plus: „+“ => {(PLUS, „“)}  
Addition = Zahl.Then(Plus).Then(Zahl)  
Addition: „12+5“ => {{{???, „“}}}
```

Die Parser `Zahl` und `Plus` in diesem Beispiel sind voneinander völlig unabhängig und können nicht auf die Ergebnisse der jeweils anderen Parser zugreifen. Das Ergebnis soll aber aus einer Kombination der Einzelergebnisse bestehen (in diesem Fall z.B. das Additionsergebnis `17`). Es wird also ein

Verfahren benötigt mit dem die Zwischenergebnisse der Parser gespeichert werden können und später ein kombiniertes Ergebnis erzeugt werden kann. Genau dies ermöglicht der Then-Kombinator, wenn auch auf eine zunächst ein wenig schwer verständliche Art und Weise.

Wir möchten 2 Parser `p1` und `p2` konkatenieren. Die Ergebnisse von `p1` sollen dabei nicht verloren gehen. Die Lösung besteht darin `p2` nicht direkt zu übergeben, sondern dynamisch und abhängig von dem Ergebnis von `p1` zu erstellen. Dazu wird dem Then-Kombinator anstelle von `p2` eine Funktion übergeben, die als Parameter das Parse-Ergebnis von `p1` entgegennimmt und dann als Rückgabewert den neuen Parser `p2` „produziert“.

Der von Then zurückgegebene Parser ruft also zunächst `p1` auf, ruft dann die als Argument übergebene Funktion auf (mit dem Parseergebnis von `p1` als Parameter). Zuletzt ruft er den von dieser Funktion erstellten Parser `p2` auf (mit den von `p1` verbleibenden Tokens als Parameter) und gibt das Ergebnis dieses Parsers als Ergebnis des kombinierten Parsers zurück.

Dazu ein kleines Beispiel zum Verständnis:

```
Digit = Char.Then(d => Return(int.Parse(d)))
```

Dieser Ausdruck entspricht einer Grammatikregel der Form:

```
Digit2Num ::= Digit:d { RESULT = int.Parse(d); }
```

`Digit` ist also ein Parser der ein Zeichen aus der Eingabe liest, als Nummer interpretiert und den dazugehörigen Ganzzahlwert (`int.Parse`) zurückgibt:

```
Digit: "123" => {(1, "23")}
```

Der Then-Kombinator kombiniert in diesem Fall einen Char-Parser und **einen** (von vielen möglichen) Return-Parsern. Das Argument von Then ist **kein** Parser, sondern eine Funktion die einen Parameter (`d`) entgegennimmt und einen neuen Parser (das Ergebnis des Ausdrucks `Return(int.Parse(d))`) zurückgibt. Die obige aus C# entnommene Schreibweise ist dabei eine verkürzte Schreibweise für den folgenden äquivalenten Ausdruck:

```
Digit = Char.Then(ReturnInt)
```

**Mit der Funktion:**

```
Parser ReturnInt(string d)
{
    return Return(int.Parse(d));
}
```

Die Schreibweise:

```
Parameter => ReturnExpression
```

Wird auch als Lambda-Ausdruck oder anonyme Funktion bezeichnet. Anonym deshalb weil die Funktion keinen Namen besitzt.

In unserem Beispiel führt der von Then erzeugte Parser für die Eingabe „123“ die folgenden Schritte aus:

1. Der Char-Parser wird mit dem Argument „123“ aufgerufen. Der Rückgabewert ist { („1“, „23“) }. Das Parserergebnis ist also „1“ und die verbleibenden Tokens sind „23“.
2. Das Parseergebnis („1“) wird an die an Then übergebene Funktion (`d => Return(int.Parse(d))`) übergeben. Das entspricht einem Aufruf der Funktion `ReturnInt` mit dem Parameter „1“. Diese Funktion gibt den Ausdruck `Return(int.Parse(„1“))` bzw. `Return(1)` zurück. `Return` ist wie im vorherigen Kapitel beschrieben eine Fabrikmethode für Parser. `Return` gibt für den Parameter 1 also einen Parser zurück, der unabhängig von der Eingabe keine Tokens konsumiert und immer den Wert 1 zurückgibt.
3. Der zurückgegebene, neue Parser (`Return(1)`) wird mit der verbleibenden Tokenmenge „23“ aufgerufen. Das Ergebnis ist das Tupel (1, „23“):

```
Return(1): „23“ => {(1, „23“)}
```

Natürlich lassen sich auch kompliziertere Anwendungsbeispiele realisieren. Hier das zu Anfang verwendete Beispiel der Addition mit einer Gegenüberstellung einer regulären Grammatik:

Combinator-Grammatik	Herkömmliche Grammatikregel
<pre>Addition = Zahl.Then(n1 =&gt; Plus.Then(a =&gt; Zahl.Then(n2 =&gt; Return(n1 + n2))))</pre>	<pre>ADDITION ::= ZAHL:n1 PLUS:a ZAHL:n2 {{ RESULT = n1+n2 }}</pre>

Die spezielle Funktionsweise des Then-Kombinators ist eine in der funktionalen Programmierung häufig anzutreffendes Programmier-Muster und eignet sich für viele Anwendungsgebiete. In der Literatur wird diese Funktion allgemein als Bind bezeichnet. Zusammen mit dem Return-Kombinator bildet er einen sogenannten Monad.

## Probleme und Einschränkungen des kombinatorischen Parsens

Das kombinatorische Parsen bietet einige Vorteile gegenüber herkömmlichen Parsern. Das trotzdem die Verbreitung außerhalb von akademischen Kreisen nur sehr begrenzt ist, liegt neben der beschränkten Unterstützung für funktionale Programmierung in vielen Mainstream-Sprachen auch an einigen mehr oder weniger leicht lösbaren Problemen die beim kombinatorischen Parsen oft auftreten.

### Laufzeit

Die Laufzeit beim kombinatorischen Parsen ist bei naiven, unoptimierten Implementierungen exponentiell. Das liegt unter anderem daran, dass der Parser ohne Lookahead arbeitet und sämtliche Alternativen in der Grammatik durchprobiert. Dazu kommt, dass der gleiche Parser oft mehrmals für die gleiche Eingabe aufgerufen wird. Zum Beispiel:

```
Ausdruck = Term + Rest | Term - Rest
```

Ruft man den Parser **Ausdruck** für die Eingabe „5-3“ auf, so wird zunächst die Alternative **Term+Rest** geprüft und anschließend die Alternative **Term-Rest**. In beiden Fällen wird aber zu Beginn der **Term**-Parser für die Eingabe „5-3“ aufgerufen, obwohl das Ergebnis eines bestimmten Parsers für eine bestimmte Eingabe immer konstant sein sollte.

Bei komplexeren Grammatiken führt dies zu vielen unnötigen Aufrufen und einer exponentiellen Laufzeit. Eine einfache und in der Algorithmik bzw. funktionalen Programmierung häufig angewendete Lösung dafür ist Memoization. Dabei werden einmal produzierte Ergebnisse eines Parsers für eine bestimmte Eingabe gespeichert und beim nächsten Aufruf mit der gleichen Eingabe wiederverwendet. Allein durch diesen vergleichsweise einfach zu realisierenden Schritt lässt sich eine polynomiale Laufzeit von  $O(n^3)$  erreichen. Das ist zwar ein enormer Fortschritt, der komplexere Grammatiken bzw. größere Eingaben überhaupt erst möglich macht. Im Vergleich zu herkömmlichen Compilern, die eine lineare Laufzeit  $O(n)$  erreichen, ist dies aber für viele Anwendungen immer noch zu langsam. Eine lineare Laufzeit lässt sich auch mit kombinatorischen Parsern erreichen, dazu muss aber die Grammatik angepasst werden und spezielle Kombinatoren verwendet werden, die die Laufzeitkomplexität einschränken. Zum Beispiel lässt sich ein neuer Or-Kombinator definieren, der die 2. Alternative nur dann prüft, wenn der erste Parser keine Eingabe konsumiert hat. Dies hat die gleiche Wirkung wie ein Lookahead bei herkömmlichen Parsern. Man erreicht damit zwar eine deutlich höhere Geschwindigkeit, das Verfahren setzt aber voraus, dass die Grammatik entsprechend angepasst wird (Faktorisierung).

### Linksrekursion

Schwieriger zu lösen ist das Problem der Linksrekursion, das auch bei LL-Parsern bzw. beim rekursiven Abstieg auftritt. Wie bei den letztgenannten Parsern lässt sich das Problem ebenfalls durch Anpassung der Grammatik und Auflösung der Linksrekursion lösen. Die resultierenden Grammatiken sind aber in der Regel nicht nur schlecht lesbar, sondern erschweren auch das Erstellen von ASTs erheblich.

Ein anderer Lösungsansatz, der in den letzten Jahren insbesondere von Frost (R.) und Hafiz (R.) entwickelt wurde (siehe Literatur), besteht darin die Linksrekursion zuzulassen, aber in der Tiefe zu beschränken. Man kann zeigen, dass die maximal benötigte Rekursionstiefe für einen einzelnen Parser durch die Länge der Eingabe beschränkt ist. Weitere Rekursionsschritte würden an dieser

Stelle zu Nichtterminalen führen, die zu  $\epsilon$  abgeleitet werden müssen damit die endgültige Ableitung die Länge der Eingabe nicht überschreitet.

Besonders elegant lässt sich dies innerhalb der Memoization Funktion lösen, so dass auch keine weiteren Anpassungen der Grammatik oder der Parser nötig sind um Linksrekursion zuzulassen. Dies ist auch der Ansatz der in diesem Parser-Kombinator-Framework verwendet wird.

Leider ist eine korrekte Implementierung dieses Ansatzes alles andere als trivial, da viele Sonderfälle berücksichtigt werden müssen, damit der Parser tatsächlich in allen Fällen korrekte Ableitungen liefert. Zudem erhöht sich die Laufzeit (bei Verwendung von Memoization) im Worst-Case von  $O(n^3)$  auf  $O(n^4)$ .

### Fehlerbehandlung

Die hier vorgestellte Technik und die zugehörige Implementierung geben, wenn für eine bestimmte Eingabe keine Ableitung möglich ist, lediglich eine leere Menge zurück ohne dem Aufrufer mitzuteilen an welcher Stelle und warum der Parse-Vorgang fehlgeschlagen ist.

Da im Laufe des kombinatorischen Parsens prinzipbedingt eine große Anzahl von Parse-Vorgängen auftreten die zu keinem Ergebnis führen (z.B. Austesten einer bestimmten, nicht zutreffenden Alternative in einem Or-Kombinator) ist es schwierig zu entscheiden an welcher Stelle ein Parse-Vorgang fehlgeschlagen ist. Zudem hat ein einzelner Parser keinerlei Informationen darüber in welchem Kontext er aufgerufen wurde.

### Beispiel

An einer Stelle in der Grammatik sind entweder eine Zahl oder ein Variablenname zulässig:

```
Value = Number.Or(Variable)
```

Wendet man auf den Value-Parser nun bestimmte Eingaben an, so könnten die einzelnen Parser Number und Variable jeweils ein Ergebnis oder eine Fehlermeldung zurückgeben:

#### Eingabe ist Zahl

```
Number: „123“ => {(123, „“)}  
Variable: „123“ => {#Variable expected#}  
Value: „123“ => {(123, „“)}
```

#### Eingabe ist Variablenname

```
Number: „abc“ => {#Number expected#}  
Variable: „abc“ => {(Variable: abc, „“)}  
Value: „abc“ => { (Variable: abc, „“)}
```

#### Eingabe ist weder Zahl noch Variablenname

```
Number: „%“ => {#Number expected#}  
Variable: „%“ => {#Variable expected#}  
Value: „%“ => {#Number or Variable expected#}
```

Das korrekte Verhalten im Ersten Fall ist die vom **Variable**-Parser erzeugte Fehlermeldung zu verwerfen und lediglich das Ergebnis des **Number**-Parsers zu berücksichtigen. Ebenso muss im zweiten Fall die Fehlermeldung des **Number**-Parsers ignoriert werden. Im letzten Fall handelt es sich aber (im Kontext des **Value**-Parsers) tatsächlich um eine fehlerhafte Eingabe und es muss eine entsprechende (möglichst aussagekräftige) Fehlermeldung produziert werden. Die Fehlermeldung

kann in diesem Fall durch Kombination der Einzelfehlermeldungen produziert werden. Je nach Grammatik ist dies aber nicht trivial und benötigt gute Heuristiken um Fehler an der richtigen Stelle zu erkennen und gute Fehlermeldungen zu produzieren.

## Implementierung

Grundvoraussetzung für das kombinatorische Parsen sind Higher-Order-Funktionen. Da dies in einigen Sprachen nicht ohne weiteres möglich ist, oder die Syntax dafür recht umständlich sein kann, eignen sich einige Programmiersprachen besser als andere für die Implementierung dieser Technik. Gerade funktionale Programmiersprachen eignen sich aus naheliegenden Gründen im besonderen Maße für solche Ansätze, aber auch in anderen Sprachen lassen sich (gegebenenfalls mit Einschränkungen) gute Implementierungen realisieren.

Für die hier vorgestellte Implementierung wurde die Sprache C# in der Version 3.0 gewählt. Die Version ist deshalb wichtig, weil mit Version 3.0 wichtige Neuerungen insbesondere im Bereich der funktionalen Programmierung dazukamen, die für dieses Projekt sehr hilfreich waren:

- Language INtegrated Query (LINQ)
- Lambda Ausdrücke
- Expression Trees
- Extension Methods
- Type Inference für lokale Variablen (var)

Diese neuen Features sind eng miteinander verzahnt. Genau genommen sind Lambda Ausdrücke, Extension Methods und Expression Trees Features die benötigt wurden um LINQ zu unterstützen.

Da diese Sprachfeatures für das Verständnis des Combinator-Frameworks wichtig sind, möchte ich im nächsten Abschnitt kurz auf die einzelnen Features eingehen.

### Spracherweiterungen von C# 3.0

LINQ, oder auch **Language Integrated Query** erweitert C# um die Fähigkeit beliebige Datenquellen abzufragen ohne dabei auf spezielle Abfragesprachen wie SQL zurückgreifen zu müssen. Die Abfragen werden stattdessen direkt in C# ausgedrückt:

```
var adultNames = from p in People where p.Age >= 18 select p.Name;
```

Die obige Abfrage würde aus der Datenquelle **People** die Namen aller volljährigen Personen zurückliefern. Wichtig zu wissen ist noch, dass **adultNames** nach der Zuweisung vom Typ **IEnumerable<string>** ist und die Abfrage erst im Falle einer **Enumeration** (z.B. in einer **foreach**-Schleife) tatsächlich ausgeführt wird. Dank der neuen **Type Inference** wird der Typ der Variablen automatisch vom Compiler ermittelt, so dass stattdessen das Schlüsselwort **var** verwendet werden kann.

Die obige Syntax ist für den C#-Compiler lediglich eine schönere Schreibweise für den folgenden äquivalenten Code:

```
var adultNames = People.Where(p => p.Age >= 18).Select(p => p.Name);
```

Die LINQ-Schlüsselwörter **from**, **where** und **select** werden also vom C# Compiler in einfache Methodenaufrufe umgewandelt. Jede Klasse die entsprechende benannte Methoden (mindestens **select**) mit der richtigen Signatur besitzt kann also prinzipiell als Datenquelle für LINQ dienen.

Im Parser Combinator Framework wird dieses Feature benutzt um Parser zusammzusetzen:

```
var symbol = from first in Letter
             from rest in LetterOrDigit.Rep()
             select first + String(rest);
```

In diesem Beispiel wird ein neuer Parser (**symbol**) aus Kombination der Parser **Letter** und **LetterOrDigit** erstellt. Die Schreibweise ohne LINQ ist deutlich weniger lesbar:

```
var symbol2 = Letter.Then(first => LetterOrDigit.Rep()).Then(rest =>
Return<char, string>(first + String(rest)));
```

Ein weiteres neues und wichtiges Konzept in C# 3.0, das in den beiden letzten beiden Beispielen zu sehen ist, sind die sogenannten **Lambda-Ausdrücke**:

```
var adultNames = People.Where(p => p.Age >= 18).Select(p => p.Name);
```

Lambda Ausdrücke sind anonyme Methodendeklarationen und haben in C# die Form (**parameter**) => **ausdruck** oder (**parameter**) => { **block** }. Anonym heißen sie deshalb, weil die Methoden keinen Namen besitzen. Die obige Abfrage ließe sich ohne Hilfe von Lambda Ausdrücken auch so formulieren:

```
bool ageCheck(Person p) { return p.Age >= 18; }
string nameFromPerson(Person p) { return p.Name; }
var adultNames = People.Where(ageCheck).Select(nameFromPerson);
```

Lambda Ausdrücke werden im Combinator Framework an vielen Stellen verwendet. So zum Beispiel zum Erstellen eines Return-Parsers:

```
public static P<TTok, TRes> Return<TTok, TRes>(TRes result)
{
    return input => new[] { new ParseResult<TTok, TRes>(input, result) };
}
```

Man sieht hier, dass im Körper des Lambda Ausdrucks auch auf lokale Variablen oder Parameter der Funktion zugegriffen werden darf, die den Ausdruck enthält, ohne dass diese als direkte Parameter des Lambdas übergeben werden. Dieses Konzept ist als Closure bekannt.

Eine Besonderheit von C# erlaubt es Lambda Ausdrücke vom Compiler in einen abstrakten Syntaxbaum (AST, **Expression Tree**) umwandeln zu lassen, der zur Laufzeit ausgewertet werden kann:

```
Expression<Func<int>> constantEx = () => 5;
```

Da der Typ von **constantEx** vom Typ **Expression<T>** ist, wird der rechte Ausdruck vom Compiler in einen abstrakten Syntaxbaum (AST) umgewandelt und der Variable zugewiesen:

```
Expression<Func<int>> constantEx =
Expression.Lambda<Func<int>>(Expression.Constant(5, typeof(int)));
```

Diese Funktion wird im Rahmen von LINQ benötigt um Lambda Ausdrücke (je nach Datenquelle) zum Beispiel in effiziente SQL-Ausdrücke umzuwandeln statt diese (lokal ausgeführte) Methoden zu interpretieren.

Im Rahmen dieses Projektes werden Expression-Trees lediglich für die Beispielimplementierung des Mathe-Parsers benutzt. Gegenüber eigenen ASTs haben diese den Vorteil sich direkt mithilfe einer `Compile()`-Methode zur Laufzeit kompilieren zu lassen, so dass keine eigene Codeerzeugung mehr benötigt wird.

Das letzte mit C# 3.0 eingeführte neue Feature sind die sogenannten **Extension Methods**, also „Erweiterungsmethoden“.

```
var adultNames = People.Where(p => p.Age >= 18).Select(p => p.Name);
```

Schauen wir uns den obigen Code nochmal an und gehen wir davon aus, dass es sich bei `People` um den Typ `IEnumerable<People>` handelt. Die Abfrage wird funktionieren, ohne dass das Interface `IEnumerable<T>` eine Methode namens `Where` kennt. Die einzige Methode von `IEnumerable<T>` lautet `GetEnumerator`.

Bei der benutzen `Where`-Methode handelt es sich um eine Erweiterungsmethode, die also den Typ `IEnumerable<T>` um die zusätzliche Funktion „erweitert“. Die in diesem Beispiel benutzte Methode liegt in der Klasse `System.Linq.Enumerable` und hat die Signatur:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)
```

Es handelt sich also um eine reguläre statische Methode, die für den ersten Parameter den zu erweiternden Typ und das spezielle Schlüsselwort `this` angibt.

Wichtig ist, dass es sich bei Erweiterungsmethoden lediglich um eine reine Syntaxvereinfachung handelt. Die Erweiterungsmethoden können nicht auf private Member der erweiterten Klasse zugreifen und der Methodenaufruf wird schon zur Kompilierzeit entsprechend aufgelöst:

```
var adultNames = Enumerable.Select(Enumerable.Where(People, p => p.Age >= 18), p => p.Name);
```

Außerdem werden Erweiterungsmethoden nur berücksichtigt wenn die Klasse die diese enthält mit einem `using`-Statement importiert wurde. Für obiges Beispiel ist also die Zeile

```
using System.Linq;
```

zwingend erforderlich.

Erweiterungsmethoden spielen im Combinator Framework eine große Rolle. Nicht nur für die LINQ Unterstützung werden diese gebraucht, sondern auch die normalen Kombinator-Funktionen wie `Then`, `Or` oder `Repeat` machen davon Gebrauch. Ohne Erweiterungsmethoden ließe sich ein Aufruf wie `Letter.Or(Number)` nur als `Combinators.Or(Letter, Number)` schreiben. Dabei muss man bedenken, dass Parser vom .NET Typ `Delegate` sind, der einem typsicheren Funktionszeiger

ähnelt und normalerweise nicht vom Benutzer erweiterbar ist. Erweiterungsmethoden ermöglichen dies jedoch ohne weiteres.

## Projektstruktur

Das Parser Combinator Framework gliedert sich intern in vier Namespaces **State**, **Combinator**, **Parser** und **Helper**.

### State

Der Namespace **State** enthält die Klassen **ParseState**, **ParseResult** und **Sequence**. Dies sind die einzigen Klassen im Framework die instanziiert werden und Daten speichern. Sie repräsentieren Zustände (State) im System. Alle drei Klassen sind immutable, das bedeutet, dass die in ihnen gespeicherten Daten nach dem Instanziiieren nicht mehr geändert werden können.

**Sequence** ist eine generische, immutable, verkettete Liste. Jede **Sequence**-Instanz besteht aus einem gespeichertem Element (**head**) und einem Zeiger auf eine weitere **Sequenz**-Instanz, die die restlichen Elemente enthält (**tail**).

**Sequence** wird intern verwendet um die Liste der verbleibenden Tokens vor oder nach einem Parse-Vorgang darzustellen. Diese Datenstruktur bietet sich an, da Tokens immer vom Beginn der Eingabe gelesen werden und die Liste der verbleibenden Tokens immer aus den letzten n Token der Eingabe besteht, wobei n eine natürliche Zahl zwischen 0 und der Länge der Eingabe sein kann. Wird ein Token aus einer **Sequence** „gelesen“ (daher entfernt), so geschieht dass in dem einfach ein Verweis auf die **tail-Sequence** zurückgegeben wird, die die restlichen Tokens enthält.

Für eine Eingabe der Länge n müssen also insgesamt nur n Instanzen der Sequenz-Klasse erstellt werden, auch wenn viele Millionen Parse-Vorgänge durchgeführt werden.

Die Klasse **ParseState** stellt den aktuellen Zustand eines Parse-Vorgangs dar, sie dient als Input für einen Parser und wird als Teil des Ergebnisses vom Parser zurückgegeben. Sie kapselt intern ein **Sequence**-Objekt, das wie oben beschrieben die verbleibenden Tokens im Eingabestrom darstellt.

Instanzen der Klasse **ParseResult** werden als Ergebnis eines Parsers zurückgegeben. Ein **ParseResult** besteht immer aus dem (partiellen) Ergebnis des Parsers und dem neuen **ParseState** nach Anwendung des Parsers, also primär den verbleibenden Tokens.

### Combinator

Im Combinator Namespace befinden sich alle vordefinierten Kombinatoren. Die Grundkombinatoren **Then** und **Or** (Primitive) befinden sich in der Klasse **PrimitiveCombinators**. Alle anderen Kombinatoren bauen auf diesen auf.

Nennenswert sind dabei insbesondere die Kombinatoren für Wiederholung (**RepeatCombinators**) und für die LINQ-Unterstützung (**LinqCombinators**).

### Parser

Entsprechend enthält der Parser Namespace alle vordefinierten Parser des Frameworks. Hier gibt es ebenfalls eine Klasse **PrimitiveParsers** die die Parser-Primitive **Return**, **Fail** und **Accept** enthält. Die Klasse **CharacterParser** enthält vordefinierte Parser zum Lesen von Character-Tokens (Typ **char**) und Strings.

## Helper

Der **Helper** Namespace enthält 2 wichtige Erweiterungsklassen für Parser.

Die **MemoizeExtension**-Klasse ermöglicht die Memoization von Parse-Ergebnissen sowie die Limitierung der Rekursionstiefe bei linksrekursiven Parsern. Sie ist damit essenziell um eine hohe Geschwindigkeit zu erreichen.

Der in diesem Framework verwendete Algorithmus zur Beschränkung der Rekursionstiefe ist direkt aus den entsprechenden Veröffentlichungen von Frost und Hafiz abgeleitet und ist neben der Proof-of-Concept Implementierung der Autoren in Haskell eine der wenigen (mir bekannten) Frameworks die Linksrekursion direkt unterstützen. Die gängigen Parser-Combinator-Frameworks scheitern an dieser Aufgabe. Da die korrekte Implementierung schwierig ist, ist allerdings nicht auszuschließen dass der verwendete Algorithmus fehlerhaft ist und nicht in allen Fällen zu korrekten Ergebnissen führt. Da die Rekursionstiefe direkt von der Länge der Eingabe abhängt, kann es zudem auch mit Rekursionsbeschränkung zu Stack-Overflows kommen, wenn die Eingabe zu lang wird. Eine Lösung dafür könnte die Verwendung von Tail-Recursion sein, die von C# aber nicht direkt unterstützt wird.

Die **ParseExtension**-Klasse bietet die Hilfsmethoden **Parse** und **ParsePartial** die einen Parse-Vorgang starten und eine Liste der Ergebnisse an den Aufrufer zurückgeben. Sie nehmen als Argument ein **IEnumerable<T>**, konvertieren dieses in eine **Sequence<T>** bzw. einen **ParseState**, rufen den als weiteres Argument übergebenen Parser mit diesem **ParseState** auf und transformieren die Liste vom Typ **ParseResult<TResult>** in ein Array vom Typ **TResult**, welches an den Aufrufer zurückgegeben wird.

## Beispiele

Hier sind einige Verwendungsbeispiele für das Parser Combinator Framework. Ausführlichere Beispiele finden sich in dem beigefügten **MathParser** Testprojekt.

```
var letterResult = Letter.ParsePartial("abcde");

var letterRepeatResult = Letter.Rep1().CharsToString().ParsePartial("abc");

//LINQ
var switchLetters =
    from x in Letter
    from y in Letter
    select y.ToString() + x;

var switchLettersResult = switchLetters.Parse("ab");

var number =
    from n in Digit.Rep1()
    select int.Parse(String(n));

var numberResult = number.Parse("12345");
```

## Fazit

Parser-Kombinatoren sind akademisch sehr interessant und bilden ein aktives Forschungsgebiet im Bereich der Parsing-Techniken. In bestimmten Bereichen wie dem Parsen von natürlicher Sprache sind sie bereits jetzt durchaus konkurrenzfähig zu anderen Parsing-Techniken und bestechen dabei unter anderem durch eine einfache Umsetzung bei gleichzeitig großer Flexibilität. Im Bereich des Compilerbaus überwiegen momentan allerdings noch die Nachteile wie das schlechte Laufzeitverhalten, aber zum Teil auch kleinere Schwierigkeiten, wie die Nachvollziehbarkeit des Programmflusses (Debugging) oder die teilweise sehr theoretischen und schwierig verständlichen zugrundeliegenden Konzepte (Bind, Monads, Kombinatoren, ...). Außerdem bieten die wenigsten Mainstream-Programmiersprachen alle Sprach- (z.B. Lambda-Ausdrücke) bzw. Laufzeit-Features (z.B. Tail-Recursion) die benötigt werden um eine elegante und gleichzeitig effiziente Implementierung zu ermöglichen. Die zurzeit stärker werdende Verbreitung von funktionalen Programmierkonzepten, sowie die weiterhin sehr aktive Forschung in diesem Gebiet, lassen aber hoffen das in Zukunft Parser-Kombinatoren auch im Mainstream eine größere Verbreitung finden.

## Quellen und weiterführende Literatur

### Monadic Parser Combinators using C# 3.0, LukeH's WebLog

Adresse: <http://blogs.msdn.com/lukeh>

- Beispielimplementierung in C#
- mit LINQ- Unterstützung
- keine Unterstützung von Linksrekursion oder Mehrdeutigkeit

### Monadic parser combinators (I-VII), Brian McNamara

Adresse: <http://orgonblog.spaces.live.com/blog/>

- Sehr Ausführlich, inklusive Fehlerbehandlung
- keine Unterstützung von Linksrekursion oder Mehrdeutigkeit

### The X-SAIGA Project, Hafiz, R. and Frost, R

Adresse: <http://www.cs.uwindsor.ca/~hafiz/xsaiga/pub.html>

- Publikationen zum Thema Linksrekursion und Mehrdeutigkeit
- Proof-of-Concept Implementierung in Haskell

### FParsec - A Parser Combinator Library for F#

Adresse: <http://www.quanttec.com/fparsec/>

- Portierung der populären Haskell-Bibliothek Parsec zu der funktionalen .NET Sprache F#
- Voll funktionsfähige Parser-Kombinator-Bibliothek inklusive Fehlerbehandlung
- Keine Unterstützung für Linksrekursion oder Mehrdeutigkeit