

# Parser Combinators in C#

Thomas Krause

# Gliederung

---

- ▶ Einführung
- ▶ Was ist ein Parser?
- ▶ Einfache und komplexe Parser
- ▶ Probleme
- ▶ DEMO: Combinator Framework in C#

# Parser Combinators

---

- ▶ Parsing-Technik
- ▶ Top-Down-Ansatz
- ▶ Ähneln rekursivem Abstieg
- ▶ Einfache Parser werden zu immer komplexeren Parsern kombiniert
- ▶ Keine separate Sprache für Syntaxbeschreibung
  - ▶ Syntax wird als Serie von Funktionsaufrufen beschrieben
  - ▶ Direkt ausführbare Grammatik!
- ▶ Basiert auf Higher Order Functions

# Was ist ein Parser?

---

- ▶ Funktion:  $\text{input} \Rightarrow \{(\text{result\_1}, \text{newstate\_1}), \dots\}$
- ▶ Eingabe
  - ▶ Zu parsende Tokens
- ▶ Ausgabe
  - ▶ Liste von Tupeln: ((Teil)Ergebnis, Verbleibende Tokens)

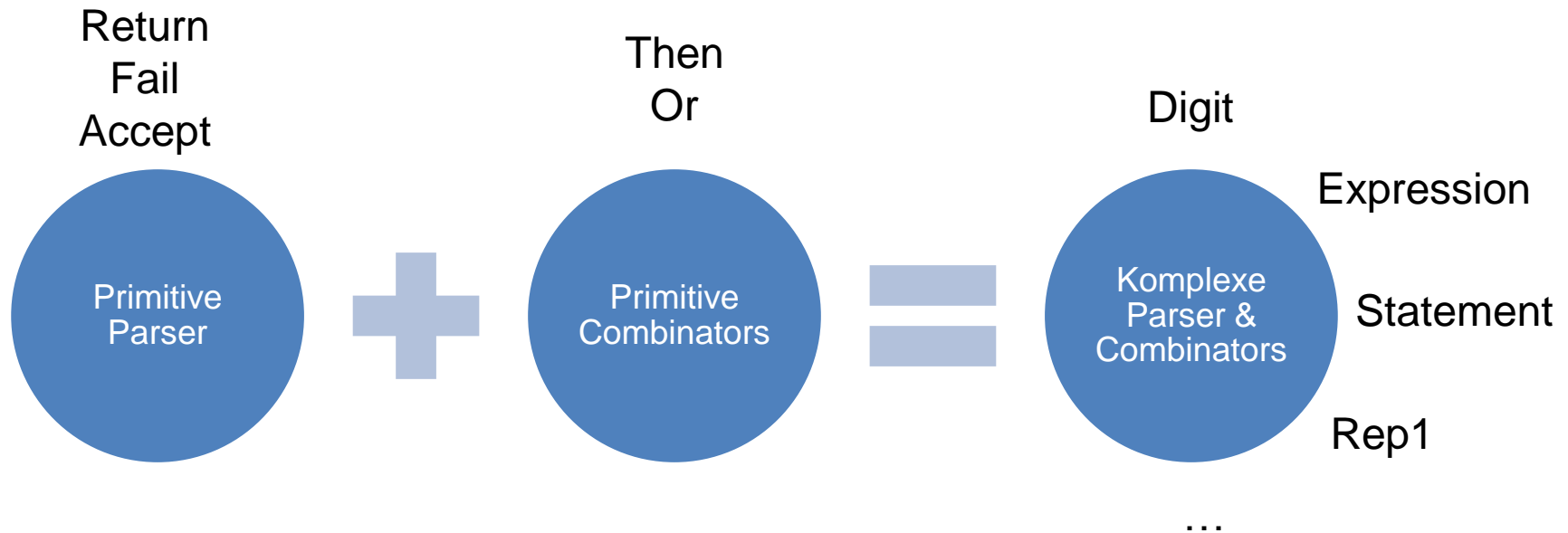
# Beispiele

---

- ▶ Parser für einzelne Ziffer:
  - ▶ Ziffer: „123“  $\Rightarrow \{(1, \text{„23“})\}$
- ▶ Parser für eine Zahl:
  - ▶ Zahl: „123 “  $\Rightarrow \{(1, \text{„23“}), (12, \text{„3“}), (123, \text{„“})\}$
- ▶ Mehrdeutige Grammatiken sind möglich:
  - ▶ Ausdruck: „1+2\*3“  $\Rightarrow \{(9, \text{„“}), (7, \text{„“})\}$
- ▶ Bei erfolglosem Parsen  $\rightarrow$  leere Menge:
  - ▶ Zahl: „abc“  $\Rightarrow \{ \}$

# Parser & Combinators

---



- ▶ **Combinator:** Funktion die ein oder mehrere Parser zu einem neuen Parser verknüpft
  - ▶ Beispiel:  $\text{Symbol} = (\text{Letter} \mid \text{Digit})^*$

# Primitive Parser - Return

---

- ▶ Return(result) – Fabrik für Return-Parser
- ▶ Konsumiert keine Eingabe
- ▶ Gibt immer einen bestimmten Wert (result) zurück
- ▶ Beispiel:
  - ▶ Return(2): „xyz“ => {(2, „xyz“)}

# Primitive Parser - Fail

---

- ▶ Fail – Fail-Parser
- ▶ Gibt immer eine leere Menge zurück (Parse-Vorgang fehlgeschlagen)
- ▶ Beispiel:
  - ▶ Fail: „xyz“ => { }

# Primitive Parser: Accept

---

- ▶ Accept – Accept-Parser
- ▶ Konsumiert einzelnes Zeichen aus der Eingabe und gibt es zurück
- ▶ Beispiel:
  - ▶ Accept: „xyz“=> {(„x“, „yz“)}
- ▶ Ist die Eingabe leer (Ende der Eingabe), wird leere Menge zurückgegeben:
  - ▶ Accept: „“=> { }

# Combinator Primitive - Or

---

- ▶  $p1.Or(p2): input \Rightarrow p1(input) \cup p2(input)$
- ▶ Ergebnismenge ist die Vereinigung der Ergebnismengen der einzelnen Parser
- ▶ Beispiel:
  - ▶ Char: "123"  $\Rightarrow \{("1", "23")\}$
  - ▶ Digit: "123"  $\Rightarrow \{(1, "23")\}$
  - ▶ CharOrDigit = Char.Or(Digit);
  - ▶ CharOrDigit: "123"  $\Rightarrow \{("1", "23"), (1, "23")\}$

# Combinator Primitive - Then

---

- ▶ In Literatur als Bind bekannt
- ▶ Entspricht einer Sequenz in Grammatik
- ▶ Konkatenieren von 2 Parsern p1, p2
- ▶ **Problem:** Wie werden die Ergebnisse von p1 und p2 Zusammengeführt?
- ▶ **Lösung:** p2 wird nicht direkt übergeben, sondern jeweils aus dem Ergebnis von p1 erstellt

# Beispiel: Then-Combinator

---

- ▶ Digit2Num: Digit.**Then**(d => Return(int.Parse(d)))
  - ▶ Digit2Num ::= Digit:d { RESULT = int.Parse(d); }
- ▶ Then erstellt hier einen Parser, der:
  - ▶ Zunächst eine Ziffer aus dem Eingabestrom liest (Digit)
  - ▶ Mithilfe dieser Ziffer einen Return-Parser erstellt, der den Zahlenwert der Ziffer zurückgibt (d => ...)
  - ▶ Diesen neuen Parser mit der verbleibenden Eingabe aufruft
  - ▶ Und das Ergebnis des neuen Parsers zurückgibt

## Beispiel: Then-Combinator (2)

---

- ▶ Digit2Num: `Digit.Then(d => Return(int.Parse(d)))`
  - ▶ `Digit.Then(ReturnInt)`
  - ▶ `ReturnInt(d) { return Return(int.Parse(d)); }`
- ▶ Parsen des Strings „1“
  1. Digit-Parser wird angewendet:  
„1“ => {(„1“, „“)} (daher: Ergebnis: „1“, Rest-Tokens: „“)
  2. Funktion `ReturnInt` wird mit „1“ als Parameter aufgerufen
  3. `ReturnInt(„1“)` gibt den Parser `Return(1)` zurück
  4. `Return(1)` wird mit dem Rest der Eingabe aufgerufen („“)  
`Return(1): „“ => {(1, „“)}`
- ▶ Also: Digit2Num: „1“ => {(1, „“)}

# Beispiel: Then-Combinator (3)

---

## Combinator-Grammatik

- ▶ `AddExpression =  
number.Then(n1 =>  
AddToken.Then(a =>  
number.Then(n2 =>  
Return(n1 + n2))))`

## „Normale“ Grammatik

- ▶ `AddExpression ::=`  
EX:n1  
ADD:a  
EX:n2  
{ { RESULT = n1+n2 } }

# Zusammengesetzte Parser

---

- ▶ `Accept(Token t)` – Parser für bestimmtes Token
  - ▶ `Accept().Then(token =>  
    token == t  
    ? Return(token)  
    : Fail()  
);`
- ▶ `Accept('a'): „abc“ => { („a“, „bc“) }`      `//Return(„a“)`
- ▶ `Accept('b'): „abc“ => { }`      `//Fail()`

# Zusammengesetzte Kombinatoren

---

- ▶ `p.Rep()` – Wiederholung
  - ▶  $p^* = p^+ \mid \text{eps}$
  - ▶ `p.Rep1().Or(Return({ })))`
- ▶ `p.Rep1()` – Wiederholung (mindestens einmal)
  - ▶  $p^+ = pp^*$
  - ▶ `p.Then(first => p.Rep1.Then(remainder => Return({first} U remainder))`
- ▶ Beispiel
  - ▶ `Letter.Rep1(): „abc“ => {(„a“, „bc“), („ab“, „c“), („abc“, „“)}`

# Probleme

---

- ▶ Laufzeit
- ▶ Linksrekursion
- ▶ Fehlerbehandlung

# Problem: Laufzeit

---

- ▶ Es werden immer alle Möglichkeiten durchprobiert
- ▶ Ein Parser parst oft mehrmals gleiche Eingabe
- ▶ Beispiel:
  - ▶ Ausdruck = Term + Rest | Term – Rest
  - ▶ „5-3“: Doppelter Aufruf des Term-Parsers für „5“
  - ▶ Bei komplexeren Grammatiken sehr ineffizient
- ▶ Exponentielle Laufzeit!
- ▶ Einfache Lösung: Memoization
  - ▶ Teilergebnisse der Parser speichern →  $O(n^3)$
- ▶ Komplexer: Spezielle Combinators + Grammatiken
  - ▶ Lineare Laufzeit, aber dafür Verlust der Flexibilität


# Problem: Linksrekursion

---

- ▶ `ex = ex.Then(...)`
  - ▶ „ex“-Parser ruft „ex“-Parser auf, der „ex“-Parser aufruft, ...
  - ▶ Stack-Overflow
- ▶ Lösungsansatz:
  - ▶ Für jeden Parser-Aufruf Rekursionstiefe merken
  - ▶ Maximale Rekursionstiefe ist durch die Länge der Eingabe begrenzt
  - ▶ Lässt sich mit Memoize Funktion kombinieren
  - ▶ Laufzeit inkl. Memoization:  $O(n^4)$
- ▶ Korrekte Implementierung nicht trivial!
  - ▶ Literatur: Frost, R. and Hafiz, R. (2006-2010)
  - ▶ PoC-Implementierung in Haskell

# Fehlerbehandlung

---

- ▶ In der vorgestellten Form keine Fehlerbehandlung
  - ▶ Parse-Fehler resultieren in leerer Ergebnismenge
- ▶ Lösungsansatz:
  - ▶ Parser-Ergebnisse mit Fehlermeldungen anreichern
  - ▶ Zusammensetzen der Fehlermeldungen auf höheren Ebenen
- ▶ Beispiel:
  - ▶ Letter: „%“ => #Letter expected#
  - ▶ Digit: „%“ => #Digit expected# 
  - ▶ Letter.Or(Digit): „%“ => #Letter or Digit expected#

# Implementierung

---

- ▶ Geschrieben in C#
  - ▶ Stark vereinfacht dank Lambdas+LINQ in C# 3.0
- ▶ Unterstützt beliebige Tokens als Eingabe (generisch)
- ▶ Unterstützt Mehrdeutige Grammatiken
- ▶ Unterstützt Linksrekursion!
- ▶ Inklusive Beispiel:
  - ▶ Einfacher Parser für mathematische Ausdrücke und Funktionen



DEMO

Parser Combinator Framework in C#

# Fazit

---

- ▶ Akademisch sehr interessant
- ▶ Aktives Forschungsgebiet
- ▶ „Sehr einfach“ und trotzdem sehr mächtig
- ▶ Nachteile
  - ▶ Schlechte Laufzeit
  - ▶ Tief verschachtelte Funktionsaufrufe (→ Stack-Overflow)
  - ▶ Oft schwierig zu debuggen
  - ▶ Oft schwierig nachzuvollziehen
  - ▶ Nicht in jeder Programmiersprache gut umsetzbar

# Quellen

---

- ▶ Monadic Parser Combinators using C# 3.0
  - ▶ LukeH's WebLog – <http://blogs.msdn.com/lukeh>
  - ▶ mit LINQ, keine Linksrekursion oder Mehrdeutigkeit
- ▶ Monadic parser combinators (I-VII)
  - ▶ Brian McNamara – <http://lorgonblog.spaces.live.com/blog/>
  - ▶ Sehr Ausführlich, inklusive Fehlerbehandlung
  - ▶ Keine Linksrekursion oder Mehrdeutigkeit
- ▶ X-SAIGA Project
  - ▶ Hafiz, R. and Frost, R –  
<http://www.cs.uwindsor.ca/~hafiz/xsaiga/pub.html>
  - ▶ Publikationen zum Thema Linksrekursion und Mehrdeutigkeit
  - ▶ Implementierung in Haskell



FRAGEN?

Vielen Dank für die Aufmerksamkeit!