

Eine Parser Kombinator Bibliothek in Java

Sven Eric Panitz

14. Juni 2020

Inhaltsverzeichnis

1	Kontextfreie Grammatiken und Parser	2
1.1	Kontextfreie Grammatiken und Parser	2
1.1.1	Top-Down Parser	2
1.2	Java 14 und darüberhinaus	3
1.2.1	λ -Ausdrücke und funktionale Schnittstellen	3
1.2.2	Default-Methoden	4
1.2.3	Lokale Typinferenz	4
1.2.4	Record-Notation für Datenhaltungsklassen	4
1.2.5	<code>instanceof</code> -Pattern	4
1.2.6	Die Javainterpreter in der jshell	5
2	Parser Kombinator Bibliothek in Java	5
2.1	Importierte Klassen	5
2.2	Die Schnittstelle für Parser	5
2.3	Der Ergebnistyp	6
2.4	Die abstrakte Methode zum Parsen	6
2.5	Atomare Parser	7
2.5.1	ϵ -Parser	7
2.5.2	Erkennen von Terminalsymbolen	7
2.6	Alternativkombination	8
2.7	Sequenzkombination	9
2.8	Ergebnisberechnung	12
2.8.1	Beispiel einer rekursiven Grammatik	13
2.9	Wiederholungen	15
3	Ableitungsbaum	15
3.1	Grenzen des Ansatzes	17
3.1.1	Linksrekursion	17

3.1.2	Linker Präfix	17
4	Beispiel und Aufgabe	17
4.1	Tokenizer	17
4.2	Klassen des abstrakten Syntaxbaums (AST)	20
4.3	Grammatik	21
4.4	Parser	22
4.4.1	Konstante applicative Form	23
4.4.2	Regeln	23

1 Kontextfreie Grammatiken und Parser

1.1 Kontextfreie Grammatiken und Parser

Eine der bahnbrechenden Erfindungen des 20. Jahrhunderts geht auf den Sprachwissenschaftler Noam Chomsky[Cho56]¹ zurück.

Er präsentierte als erster ein formales Regelsystem, mit dem die Grammatik einer Sprache beschrieben werden kann. Dieses Regelsystem ist in seiner Idee verblüffend einfach. Es bietet Regeln an, mit denen mechanisch die Sätze einer Sprache generiert werden können.

Systematisch wurden Chomsky Ideen zum ersten Mal für die Beschreibung der Syntax der Programmiersprache Algol angewendet[NB60].

Für uns als Informatiker im Kontext von Programmiersprachen ist es wichtig, für eine gegebene Grammatik und einen Programmtext zu überprüfen, ob der Programmtext mit der entsprechenden Grammatik erzeugt werden kann. Ein Programm, dass diese Frage beantworten kann heißt ein Parser für eine bestimmte Grammatik. Ein Parser liegt in der Regel, sofern das Programm mit der Grammatik erzeugt werden kann, ein Ergebnis über die Erzeugung. Dieses ist oft in Form eines Ableitungsbaums, oder eines abstrakten Syntaxbaums. Manchmal wird aber auch direkt während des Parsens ein semantisches Ergebnis berechnet.

1.1.1 Top-Down Parser

Die denkbar einfachste Idee einen Parser zu implementieren ist eine Suche über alle möglichen Ableitungen der Grammatik. Diese Suche beginnt mit dem Startsymbol und versucht dann systematisch mit den Regeln der Grammatik das Programm zu erzeugen. Ein solcher Parser wird als Top-Down-Parser bezeichnet. Er wendet systematisch die Regeln der Grammatik an und versucht eine Linksableitung zu finden, die das Programm erzeugt. Es wird also immer das linkeste Nichtterminalsymbol durch die erste Regelalternative ersetzt, die dafür existiert. Sobald bei der Ableitung ein Wort entsteht, dass ein

¹Chomsky gilt als der am häufigsten zitierte Wissenschaftler des 20. Jahrhunderts. Heutzutage tritt Chomsky weniger durch seine wissenschaftlichen Arbeiten als vielmehr durch seinen Einsatz für Menschenrechte und bedrohte Völker in Erscheinung.

weiteres Terminalsymbol auf der linken Seite hat, wird dieses mit dem zu erzeugenden Programm verglichen. Fängt das Programm anders an, als generiert wurde kommt es zum Backtracking. Die zuletzt gewählt Ersetzung eines Nichtterminals für die es eine andere Regelalternative gab wird zurückgesetzt und mit der alternativen Regel versucht, das Programm zu erzeugen.

Es handelt sich also um eine vollständige Suche mit Backtracking. Phillip Wadler hat bereits 1985 gezeigt, wie diese einfach in einer verzögert ausgewerteten funktionalen Sprache umgesetzt werden kann bezeichnet [Wad85]. Wenige Jahre später wenden Frost Launchbury dieses Verfahren für die Entwicklung eines Parsers an [FL89].

Seither gilt es als eine Paradedisziplin der funktionalen Programmierung und es gibt Standardbibliotheken in funktionalen Sprachen zur Definition der Top-Down Parser mit Backtracking wie zum Beispiel in der Programmiersprache Haskell [Lei99].

In der Javawelt werden meist Tools verwendet die aus einer Beschreibung der Grammatik einen Parser Quelltext generieren. Hier wären zu nennen ANTLR [Par13], javacc oder sablecc.

1.2 Java 14 und darüberhinaus

Spätestens seit Java 1.8 aber schon beginnend mit Java 1.5 wurde das ursprüngliche Java immer mehr um Konzepte der funktionalen Programmierung erweitert. Vorreiter waren hier noch lange vor Java 1.5, das generische Typen in die Sprache aufgenommen hat, Phil Wadler und Martin Odersky mit der Spracherweiterung Pizza [OW97]. Martin Odersky hat viele der dabei entwickelten Ideen in die Programmiersprache scala eingebaut [OAC⁺04].

Heute hat Java genügend Konzepte der funktionalen Programmierung übernommen, die es ermöglichen einen Top-Down-Parser mit Backtracking in Java zu implementieren und ohne ein Parsergeneratortool auszukommen.

Im Rahmen dieser Aufgabe werden wir eine solche Bibliothek schrittweise entwickeln. Zunächst werfen wir aber kurz einen Blick auf die wichtigsten dabei verwendeten Konzepte der aktuellen Java 14 Sprache (inklusive noch vorläufiger Preview Features).

1.2.1 λ -Ausdrücke und funktionale Schnittstellen

Der wichtigste Schritt für Java, um komplexe Bibliotheken zu schreiben war die Einführung von λ -Ausdrücken mit Version 1.8 von Java. Seither ist es möglich durch eine einfache Syntax Schnittstellen zu implementieren, die genau eine Methode haben. Solche Schnittstellen werden als funktionale Schnittstellen bezeichnet. Wir werden bei der Entwicklung der Parserbibliothek exzessiv Gebrauch von λ -Ausdrücken machen. Insbesondere ein Parser selbst ist als eine funktionale Schnittstelle mit genau der einen abstrakten Methode `parse` definiert.

1.2.2 Default-Methoden

Auch seit Java 1.8 können Schnittstellen bereits ausimplementierte Methoden haben, die also nicht abstrakt sind. Diese werden als default-Methoden bezeichnet.

1.2.3 Lokale Typinferenz

Komplexe Bibliotheken, die mit vielen funktionalen Schnittstellen hantieren und auch sonst viele generische Typen verwenden, haben viele sehr lange und komplexe Typen. In Java gibt es keine Möglichkeit für diese Typen kurze und griffige Typnamen einzuführen. Würde man für jede Variable komplett immer den Typ in den Quelltext schreiben müssen, dann könnte man vor lauter Typangaben kaum noch Anwendungslogik erkennen. Zum Glück kennt Java seit Java 10 die Möglichkeit, im Quelltext den Typ einer lokalen Variablen nicht hinzuschreiben sondern stattdessen nur das Schlüsselwort **var** zu verwenden. Dieses bedeutet nicht, dass der Typ der Variable beliebig ist, sondern dass der Compiler den Typen berechnet. Man spricht dabei auch von Typinferenz.

In λ -Ausdrücken gibt das auch für die Parameter der Methode, die von dem λ -Ausdruck implementiert wird.

In Rahmen unserer Parserbibliothek werden wir wann immer möglich, den Typ nicht hinschreiben und zur Not auf eine Entwicklungsumgebung vertrauen, die auf Wunsch uns für jede Variable den inferierten Typ anzeigen kann.

1.2.4 Record-Notation für Datenhaltungsklassen

Bisher nur als preview feature vorhanden gibt es in Java jetzt eine einfache Möglichkeit eine Klasse mit Attributen zu definieren, so dass automatisch ein Standardkonstruktor, eine Methode **toString** sowie Methoden für die Gleichheit und den Hash-Code generiert werden. Solche Klassen werden als **record**-Klassen bezeichnet. Statt das Schlüsselwort **class** wird das Schlüsselwort **record** verwendet. Dem Klassennamen folgt in Form einer Parameterliste die Definition der Attribute. Dieses sind dann auch die Parameter der konstruierten Klasse.

Damit lassen sich typische Datenhaltungsklassen in einer Zeile definieren.

Die **record**-Klassen können generisch sein und Schnittstellen implementieren, aber von keinen anderen Klassen erben.

1.2.5 instanceof-Pattern

Der Operator **instanceof** führte in Java immer dazu, dass eine **instanceof**-Abfrage gefolgt war von einer Typzusicherung. Das sah immer umständlich aus und war unnötig kompliziert. Es ist nun möglich direkt nach einem **instanceof**-Ausdruck eine Variable anzugeben, die dann von dem Typ ist, auf den geprüft wurde. Folgendes kleines Beispiel

prüft für ein beliebiges Objekt `o`, ob es vom Typ `String` ist und gegebenenfalls kann man dann mit diesem Objekt unter den neuen Variablenbezeichner `s` als `String` weiterarbeiten.

```
if (o instanceof String s) return s.toUpperCase();
```

1.2.6 Die Javainterpreter in der jshell

Durch die Verfügbarkeit eines Interpreters lassen sich Funktionen direkter zu kleinen Tests einmal aufrufen und ausprobieren, was in der Entwicklung und im Verständnis für eine Bibliothek von großer Hilfe sein kann.

2 Parser Kombinator Bibliothek in Java

Parserkombinatorbibliotheken sind schon in den Anfängen der funktionalen Programmierung, noch vor den Zeiten von Haskell entworfen worden. Ein frühes sehr schönes Beispiel ist [FL89], das eine Grammatik für natürlichsprachliche Anfragen implementiert und mit einer Semantik versieht.

2.1 Importierte Klassen

Wir benötigen eine Reihe von Klassen aus dem Paket `java.util` und werden folgende Standardklassen verwenden.

Java: Parser

```
package name.panitz.parser;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.function.Predicate;
import java.util.function.BinaryOperator;
import java.util.stream.Collectors;
```

2.2 Die Schnittstelle für Parser

Ein Parser ist ein Objekt, das für eine Eingabe, die eine Folge von Token besteht, ein Ergebnis erzeugt. Wir lassen die Typen der Token und die Typen des Ergebnisses variable. Daher ist die Schnittstelle **Parser** generisch mit zwei Typvariablen. Zum einen die Typvariable `T` für einen beliebigen aber festen Typ von Token und zum anderen die Typvariable `R` für irgendein Ergebnistyp. `R` wird in den meisten Praxisfällen ein abstrakter Syntaxbaum sein.

Ein Parser hat nur eine zentrale Funktionalität: prüfe, ob eine Folge von Terminalsymbolen mit den Regeln der Grammatik aus dem Startsymbol erzeugt werden kann. Es bietet sich an, für Parser eine funktionale Schnittstelle zu definieren.

Java: Parser

```
@FunctionalInterface public interface Parser<T,R> {
```

2.3 Der Ergebnistyp

Bevor wir die abstrakte Methode zum Parsen definieren, müssen wir uns Gedanken darüber machen, was das Ergebnis eines Parses sein soll. Der Parser wird die Eingabe versuchen mit seiner Grammatik zu erzeugen. Dieses wird von links nach rechts geschehen. Eventuell kann ein Parser nicht die komplette Eingabe mit der zugrundeliegenden Grammatik erzeugen, sondern nur einen Teil. dann ist es wichtig zu wissen, welchen Teil der Eingabe übrig geblieben ist. Zusätzlich soll der Parser ein Ergebnis erzeugen.

Um diesem gerecht zu werden, sei eine Recordklasse definiert, die zwei Attribute hat. Eine Liste der noch nicht verarbeiteten Terminale der Eingabe und ein Ergebnis.

Java: Parser

```
static record Result<T,R>(List<T> rest,R result){}
```

2.4 Die abstrakte Methode zum Parsen

Mit der zuletzt definierten Recordklasse, lässt sich die abstrakte Methode zum Parsen definieren.

Der Parameter ist eine Liste von Token, also Terminalsymbolen der Grammatik. Für das Ergebnis verwenden wir die Recordklasse, die wir definiert haben. Wir sehen aber vor, dass ein Parser nicht genau ein Ergebnis hat, sondern einer Liste von Ergebnissen. Ist diese Ergebnisliste leer, dann konnte kein Präfix der Eingabe mit der Grammatik erzeugt werden. Dieses Prinzip, dass seit den frühen Tagen der Programmiersprache Lisp bekannt ist, wird auch als »replace failure by a list of successes« bezeichnet[Wad85].

Hat die Ergebnisliste mehr als ein Element, dann war die Grammatik unter Umständen mehrdeutig und führte zu unterschiedlichen Ableitung der Eingabe.

Java: Parser

```
List<Result<T, R>> parse(List<T> toks);
```

2.5 Atomare Parser

Die am einfachsten zu erkennenden Konstrukte einer Grammatik sind die Terminalsymbole und die ϵ -Produktionen.

2.5.1 ϵ -Parser

Wir schreiben eine statische Methode, die einen solchen Parser für ϵ -Produktionen erzeugt. Auch ein Parser, der kein Symbol der Eingabe verarbeitet, soll ein Ergebnis haben. Das Ergebnisobjekt wird der Methode zur Generierung des ϵ -Parsers als Parameter übergeben.

Da die Schnittstelle **Parser** eine funktionale Schnittstelle ist, kann das Objekt über einen λ -Ausdruck erzeugt werden. Im **return** steht per λ eine Funktion, die eine Tokenliste erhält und eine einelementige Ergebnisliste erzeugt, mit der unverbrauchten Eingabeliste als Rest und dem vorgegebenen Ergebnisobjekt.

Java: Parser

```
static<T,R> Parser<T, R> epsilon(R r) {  
    return (toks) -> List.of(new Result<>(toks, r));  
}
```

Mit ein paar ersten kleinen Aufrufe auf dem Javainterpreter **jshell** können wir uns mit der Arbeitsweise dieses Parsers vertraut machen:

Shell

```
jshell> Parser.epsilon("fertig").parse(List.of("bla","blub"))  
$1 ==> [Result[rest=[bla, blub], result=fertig]]  
jshell> Parser.result("Hello").parse(List.of())  
Parser.result("Hello").parse(List.of())  
$2 ==> [Result[rest=[], result=Hello]]
```

Wie man sieht, hat dieser Parser immer genau ein Ergebnis.

2.5.2 Erkennen von Terminalsymbolen

Der wichtigste elementare Parser ist dazu da, um das nächste Zeichen der Eingabe als ein bestimmtes Terminalsymbol zu erkennen. Dieses entspricht einer Regel der Form $A \rightarrow a$ der Grammatik mit $A \in N$ und $a \in T$.

Wir schreiben wieder eine statische Methode, die einen solchen Parser erzeugt. Allerdings werden wir die Methode zunächst etwas allgemeiner halten. Es wird nicht direkt nach einem Terminalsymbol geschaut, sondern nach Terminalsymbolen die eine gewisse Prädikateigenschaft haben. Hierzu wird das Prädikat als Parameter übergeben.

Mit dem λ -Ausdruck wird wieder der Parser definiert.

Es ist dabei zu prüfen, ob die Eingabe überhaupt noch ein Symbol enthält. Ist dieses der Fall, dann muss das erste Symbol die verlangte Eigenschaft haben. Nur in diesem Fall ist das Ergebnis eine einelementige Liste, mit dem ersten Token der Eingabe als Ergebnis und der Restliste ohne dem ersten Element der Eingabe als Rest. Ansonsten ist das Ergebnis des Parsers die leere Liste, die anzeigt, dass mit dieser Regel der Grammatik kein Präfix der Eingabe erzeugt werden konnte.

Java: Parser

```
static<T> Parser<T, T> terminal(Predicate<T> t) {
    return (toks) ->
        !toks.isEmpty() && t.test(toks.get(0))
        ? List.of(new Result<>(toks.subList(1, toks.size()), toks.get(0)))
        : List.of();
}
```

Auch hier können wir mit der JShell erste Tests machen.

Shell

```
jshell> Parser.terminal(x->x.equals("bla")).parse(List.of("bla", "blub"))
$3 ==> [Result[rest=[blub], result=bla]]
jshell> Parser.terminal(x->x.equals("blub")).parse(List.of("bla", "blub"))
$4 ==> []
```

2.6 Alternativkombination

Jetzt werden wir entsprechend der Regeln einer Grammatik Methoden schreiben, die aus existierenden Parserobjekte neue Parserobjekte erzeugen. Solche Methoden werden als Parserkombinatoren bezeichnet. Die erste Kombination bezieht sich auf die Wahl aus zwei Alternativen. Dieses sind Regeln der Form $A \rightarrow X|Y$. Angenommen wir haben schon die Parser für $X, Y \in (N \cup T)^*$, dann können wir diese kombinieren zu einem Parser der beide Alternativen gehen kann.

Hierzu schreiben wir die *default* Methode **alt** für die Schnittstelle **Parser**. Dabei soll das **this**-Objekt mit einem zweiten Parser zu einem neuen Parser kombiniert werden. Der zweite Parser wird als Parameter übergeben. Allerdings ist dieser nicht vom Typ **Parser<T, R>**, wie man erwarten würde, sondern vom Typ **Supplier<Parser<T, R>>**. Es ist also ein Objekt, das den zweiten Parser berechnen kann. Der Grund dafür wird in rekursiven Grammatikregeln liegen.

Die Alternativkombination lässt sich in wenigen Zeilen formulieren. Das Ergebnis ist eine Parserfunktion mit einer Tokenliste als Eingabe. Zunächst wird versucht mit dem **this**-Objekt die Tokenliste zu parsen. Wenn das fehlschlägt, spricht eine leere Liste als Ergebnis hat, dann wird der **that**-Parser für das Gesamtergebnis genommen.

Java: Parser

```
default Parser<T, R> alt(Supplier<Parser<T, R>> that) {  
    return (toks) -> {  
        var result = parse(toks);  
        return result.isEmpty() ? that.get().parse(toks) : result;  
    };  
}
```

Bei mehrdeutigen Grammatiken bevorzugt diese Kombination, die erste erfolgreiche Alternativen. Man hätte auch wenn der erste Parser ein nichtleere Ergebnisliste gahabt hätte, den zweiten Parser auf dieselbe Tokenliste starten können und beide Ergebnislisten dann vereinigen.

Man beachte bei der Alternativkombination, dass die beiden Parser, die zu einem neuen kombiniert werden, denselben Ergebnistyp haben müssen. Der Ergebnistyp muss ja eindeutig sein und kann nicht je nach Einzelfall, welche der beiden Alternativen erfolgreich ist, variieren.

Auch hier lassen sich erste Tests auf dem Javainterpreter ausprobieren.

Shell

```
jshell> Parser.terminal(x->x.equals("bla")).  
    alt()->Parser.terminal(x->x.equals("blub"))).  
    parse(List.of("blub","bla"))  
$20 ==> [Result[rest=[bla], result=blub]]  
  
jshell> Parser.terminal(x->x.equals("bla")).  
    alt()->Parser.terminal(x->x.equals("blub"))).  
    parse(List.of("bla","blub"))  
$21 ==> [Result[rest=[blub], result=bla]]
```

2.7 Sequenzkombination

Die zweite Kombination wird die Sequenz von zwei Parsern ausdrücken. Dieses entspricht Grammatikregeln der Form: $A \rightarrow XY$, für $X, Y \in (N \cup T)^*$.

Angenommen es existiert bereits ein Parser für die Wörter X und Y , so können diese kombiniert werden zu einem neuen Parser, der erst von der Eingabe einen Präfix mit X versucht zu erzeugen und dann auf den Rest der Eingabe, sofern der erste Parser erfolgreich war, versucht einen Präfix mit Y zu erzeugen.

Anders als bei der Alternativkombination kann man nicht davon ausgehen, dass beide Parser denselben Ergebnistyp haben. Was soll also der gemeinsame Parser, der aus der Sequenz der beiden Parser besteht für einen Ergebnistypen haben?

Hierzu sei ein kleiner Hilfsdatentyp für Paare definiert:

Java: Parser

```
static record Pair<A, B>(A e1, B e2){}
```

Wir wollen also die Sequenz zweier Parser bilden. Der erste Parser wird gestartet mit der Eingabe. Für jedes erfolgreiche Ergebnis dieses Parsers wird der zweite Parser auf die übriggebliebenen Resttoken gestartet. Das Gesamtergebnis besteht aus den Paaren des ersten und zweiten Parserergebnisses. Die Resttoken sind die vom zweiten Parser übrig gebliebenen Resttoken.

Auch diese Kombination definieren wir in einer *default*-Methode und bilden die Sequenz aus *this* und *that*.

Wir brauchen dabei zwei Schleifen, eine die über die Ergebnisse des ersten Parsers iteriert und eine innere Schleife, die für jedes dieser Ergebnisse den zweiten Parser startet und über dessen Ergebnisse iteriert.

Java: Parser

```
default <R2> Parser<T, Pair<R,R2>> seq(Supplier<Parser<T, R2>> that) {
    return (toks)->{
        var result1s = parse(toks);
        List<Result<T, Pair<R,R2>>> r = new ArrayList<>();
        for (var r1:result1s){
            var result2 = that.get().parse(r1.rest);
            for (var r2:result2)
                r.add(new Result<>(r2.rest, new Pair<>(r1.result, r2.result)));
        }
        return r;
    };
}
```

Für die Leser, die Gefallen an funktionaler Programmierung gefunden haben, geben wir eine zweite Version dieser Kombinatormethode. In dieser Version verwenden wir statt der Schleifen das Java-Stream-API. Die beiden *for*-Schleifen der ersten Version übersetzen hier zu Aufrufen der methode `map` auf Stream-Objekten.

Java: Parser

```
default <R2> Parser<T, Pair<R,R2>> seq2(Supplier<Parser<T, R2>> that) {  
    return (toks)->  
        parse(toks).stream()  
            .map( r1 ->  
                that.get()  
                    .parse(r1.rest).stream()  
                    .map(r2-> new Result<>(r2.rest,new Pair<>(r1.result,r2.result)))  
                    .collect(Collectors.toList()) )  
            .reduce(new ArrayList<>(), (r,r2s) -> {r.addAll(r2s);return r;}) ;  
}
```

Die Arbeitsweise auch dieser Methode lässt sich im Javainterpreter bei kleinen Beispielen ausprobieren.

Hierfür seien zunächst zwei Parser die ein Terminalsymbol in Form eines Stringobjektes erkennen definiert.

Shell

```
jshell> var bla = Parser.terminal(x->x.equals("bla"))  
var bla = Parser.terminal(x->x.equals("bla"))  
bla ==> Parser$$Lambda$20/0x0000000800b87840@23ab930d  
  
jshell> var blub = Parser.terminal(x->x.equals("blub"))  
var blub = Parser.terminal(x->x.equals("blub"))  
blub ==> Parser$$Lambda$20/0x0000000800b87840@619a5dff
```

Aus diesen zwei Parsern wird ein dritter Parser kombiniert, der die Sequenz der beiden Terminalsymbole erkennt.

Shell

```
jshell> var blablub = bla.seq(()->blub)  
var blablub = bla.seq(()->blub)  
blablub ==> Parser$$Lambda$22/0x0000000800b87440@7ab2bfe1
```

Diesen können wir jetzt auf unterschiedliche Eingabeliste anwenden.

Shell

```
jshell> blablub.parse(List.of("bla","blub","ende"))  
$16 ==> [Result[rest=[ende], result=Pair[e1=bla, e2=blub]]]  
  
jshell> blablub.parse(List.of("blub","bla","ende"))  
$17 ==> []
```

Der Leser ist aufgefordert weitere Tests im Javainterpreter zu machen.

2.8 Ergebnisberechnung

Die Sequenz zweier Parser, ergibt einen Parser, der als Ergebnis ein Paar der beiden Ergebnisse der miteinander kombinierten Parser hat. Oft will man definieren, wie die beiden Teilergebnisse zu einem eigenen neuen Ergebnis zusammen verrechnet werden. Daher bieten wir in der Bibliothek eine weitere *default*-Methode an, mit der das Ergebnis des Parsers über ein Funktionsobjekt verändert werden kann.

Diese Methode nennen wir naheliegend auch wieder **map**, wie wir es aus Listen gewohnt sind.

Das Ergebnis eines Parsers ist eine Liste von Ergebnisobjekte. Über diese Ergebnisliste wird die Standardmethode **map** des Stream-APIs angewendet, um neue Ergebnisobjekte zu erhalten.

Java: Parser

```
default <R2> Parser<T, R2> map(Function<R,R2> f) {  
    return (toks) ->  
        parse(toks).stream()  
            .map(p->new Result<>(p.rest, f.apply(p.result)))  
            .collect(Collectors.toList());  
}
```

Auch für diese Methode seien erste Beispiel im Javainterpreter ausgeführt. Zunächst sei ein Parser für Strings, die nur aus Ziffern bestehen definiert:

Shell

```
jshell> var ip = Parser.terminal((String s) ->  
↪ s.chars().allMatch(c->Character.isDigit(c)))  
ip ==> Parser$$Lambda$20/0x0000000800b87840@6bf2d08e
```

Diesen Parser verändern wir mit der Methode **map** zu einem Parser, der auch die gelsenen Zahlen der Strings als Integer-Objekte zum Ergebnis hat.

Shell

```
jshell> var i = ip.map(x->Integer.parseInt(x))  
i ==> Parser$$Lambda$27/0x0000000800b98840@73a8dfcc
```

Als weiteren Test definieren wir die Sequenz dieses Parsers mit sich selbst und addieren über die Methode **map** die beiden Teilergebnisse mit einander.

Shell

```
jshell> var ii = i.seq()->i).map(p -> p.e1()+p.e2())  
ii ==> Parser$$Lambda$27/0x0000000800b98840@aec6354
```

Ein Beispielaufruf kann uns von der Arbeitsweise dieses Parsers überzeugen.

Shell

```
jshell> ii.parse(List.of("17","25","4"))
$52 ==> [Result[rest=[4], result=42]]
```

Mit Hilfe der jetzt definierten Methode `map` können wir auch eine dritte Version des Sequenzkombinators definieren.

Java: Parser

```
default <R2> Parser<T, Pair<R,R2>> seq3(Supplier<Parser<T, R2>> that) {
    return (toks)->
        parse(toks).stream()
            .map(r1->that.get().map(r->new Pair<>(r1.result,r)).parse(r1.rest))
            .reduce(new ArrayList<>(),(r,r2s) -> {r.addAll(r2s);return r;});
}
```

2.8.1 Beispiel einer rekursiven Grammatik

Kontextfreie Grammatiken werden erst interessant, wenn sie rekursiv sind. Der typische Vertreter einer Sprache, die kontextfrei und nicht mehr regulär ist, ist die Sprache die eine Klammerung darstellt. Als erstes Beispiel eines Parser für eine solche rekursive Grammatik schreiben wir den Parser für eine Grammatik der Sprache: $\{a^n b^n | n \in \{0, 1, 2, \dots\}\}$.

Eine Grammatik für diese Sprache ist:

$$S \rightarrow aSb | \epsilon$$

Auch diese kleine Grammatik können wir interaktiv im Javainterpreter durchspielen.

Hierzu definieren wir einen Parser der den String "a" erkennt:

Shell

```
jshell> Parser<String,String> a = Parser.terminal(a->a.equals("a"))
Parser<String,String> a = Parser.terminal(a->a.equals("a"))
a ==> Parser$$Lambda$20/0x0000000800b87840@3fa77460
```

Und den Parser der als Terminal den String "b" erkennt.

Shell

```
jshell> Parser<String,String> b = Parser.terminal(a->a.equals("b"))
Parser<String,String> b = Parser.terminal(a->a.equals("b"))
b ==> Parser$$Lambda$20/0x0000000800b87840@619a5dff
```

Jetzt aus diesen mit Hilfe der Kombinatoren einen Parser für die Grammatik definieren. Da die regel rekursiv ist, geht dieses nur über eine Methode, die rekursiv ist. Eine relursive Variable ist nicht nötig. Somit definieren wir für das Startsymbol eine Methode, die den Parser für das Startsymbol erzeugt. Diese Methode wird einen rekursiven Aufruf auf sich haben. Das würde aber zu einer nicht terminierenden Rekursion führen. Daher ist die Rückgabe der Methode lediglich ein Supplier-Objekt für den Parser. Er wird noch nicht erzeugt sondern hat eine get-Methode, um ihn zu erzeugen.

Als Ergebnis der Parsers wollen wir eine ganze Zahl haben, die zählt, wieviel Paare der Buchstaben a und b geparst werden konnten.

Und hier nun die Umsetzung der Regel. Es ist zunächst die Sequenz aus einem a-Symbol und dem rekursiven Aufruf. Als Ergbnis wird 1 auf das Ergebnis der rekursiven Regelanwendung aufaddiert. Dann wird die Sequenz zu einem Parser für das b-Symbol gebildet. Dessen Ergebnis ist die Übernahme des Ergebnisses des ersten Parsers.

Insgesamt gibt es dann noch die Alternative eines ϵ -Parsers, der das Ergenis 0 hat.

Shell

```
jshell> Supplier<Parser<String,Integer>> s(){
    return ()->
        a.seq(s()).map(p->1+p.e2())
        .seq(()->b).map(p->p.e1())
        .alt(()->Parser.epsilon(0));
}
| created method s()
```

Diesen Parser können wir versuchen auf eine Folge der Buchstaben a und b aufzurufen. Hierzu erzeugen wir uns eine entsprechende Eingabeliste:

Shell

```
jshell> var ds = java.util.Arrays.asList("aaaaaaaaabbbbbbbbbb".split("|"))
var ds = java.util.Arrays.asList("aaaaaaaaabbbbbbbbbb".split("|"))
ds ==> [a, a, a, a, a, a, a, a, b, b, b, b, b, b, b, b, b]
```

Der Parser kommt zu dem Ergebnis, dass 8 a gefolgt von 8 b in der Eingabe waren und zwei weitere b nicht mit der Grammatik erzeugt werden konnten.

Shell

```
jshell> s().get().parse(ds)
$36 ==> [Result[rest=[b, b], result=8]]
```

2.9 Wiederholungen

Eigentlich sind wir soweit mit unseren Ausdrucksmöglichkeiten für kontextfreie Grammatiken fertig. Die Regeln lassen sich mit den Kombinatoren für Sequenz und Alternative stückweise in Programme umsetzen.

In vielen Grammatiken kommen Wiederholungen vor. Hierzu gibt es Erweiterungen in der Notation von Grammatiken, die diese Wiederholungen direkt und nicht über rekursive Regeln ausdrücken. So sehen wir schließlich auch noch einen Kombinator vor, mit dem die 0 bis n-fache Sequenz eines Parser ausgedrückt werden kann.

Der Ergebnistyp des entstehenden Parsers ist eine Liste. Diese wird leer sein, wenn der Parser keinmal erfolgreich war.

Wir können die n-fache Wiederholung mit den bestehenden Kombinatoren einfach ausdrücken:

Java: Parser

```
default Parser<T, List<R>> zeroToN() {  
    return  
        seq(()->zeroToN()).map(p->{p.e2().add(0,p.e1());return p.e2();})  
        .alt(()->epsilon(new ArrayList<R>()));  
}
```

Mit dem Parser für Strings, die nur aus Ziffern bestehen, lässt sich diese Funktion schnell im Interpreter ausprobieren.

Shell

```
jshell> is.parse(List.of("1","2","3","4","5","sechs"))  
$63 ==> [Result[rest=[sechs], result=[1, 2, 3, 4, 5]]]
```

Wenn bereits das erste Token nicht mit dem Parser zu erkennen ist, dann gibt es trotzdem ein erfolgreiches Ergebnis einer leeren Liste.

Shell

```
jshell> is.parse(List.of("12ww","2","3","4","5","sechs"))  
$64 ==> [Result[rest=[12ww, 2, 3, 4, 5, sechs], result=[]]]
```

3 Ableitungsbaum

Wir können für kontextfreie Grammatiken nun Parser definieren und dabei über die Methode `map` angeben, welches Ergebnis erzielt werden soll.

Wir können aber auch eine Version der Grammatik definieren, die immer einen Ableitungsbaum als Ergebnis liefert. Hierzu definieren wir uns eine Struktur für Ableitungsbäume.

Java: Parser

```
static interface Tree {}
static record Epsilon() implements Tree {}
static record Terminal(String image) implements Tree {}
static record Node(String name, List<Tree> childNodes) implements Tree {}
```

Ein Ableitungsbaum ist ein Objekt der Schnittstelle `Tree`. Hiervon gibt es drei Implementierungen. Eine für das Blatt, das mit ϵ markiert ist, eine für Blätter an denen ein Terminalsymbol hängt und einen für innere Knoten, die Nichtterminalsymbolen entsprechen.

Wir können für die Generierung spezialisierte Parser schreiben, die als Ergebnis einen Ableitungsbaum generieren.

Als erstes der Parser für Epsilonknoten im Baum:

Java: Parser

```
static<T> Parser<T,Tree> epsilonTree() {
    return epsilon(new Epsilon());
}
```

Dann der Parser, der ein Blatt für ein bestimmtes Terminalsymbol erzeugt.

Java: Parser

```
static<T> Parser<T,Tree> terminalTree(Predicate<T> t) {
    return terminal(t).map(x->new Terminal(x.toString()));
}
```

Schließlich eine Methode, die die Sequenz aus zwei Parsern erzeugt und im Ergebnisbaum einen inneren Knoten hat. Dieser wird mit einem String markiert, der der Methode mit übergeben wurde.

Java: Parser

```
static<T> Parser<T,Tree> seq
    (String name,Parser<T,Tree> dies, Supplier<Parser<T,Tree>> that) {
    return dies.seq(that).map(p->new Node(name,List.of(p.e1(),p.e2())));
}
```

Und schließlich lässt sich auch für die Wiederholung ein Baumknoten generieren.

Java: Parser

```
static<T> Parser<T, Tree> zeroToN(String name, Parser<T, Tree> dies) {  
    return dies.zeroToN().map(xs->new Node(name, xs));  
}
```

Auch diese Methoden können in einem ersten kleinen Test im Javainterpreter einmal aufgerufen werden.

Shell

```
jshell> var it = Parser.terminalTree((String s) ->  
  ↪ s.chars().allMatch(c->Character.isDigit(c)))  
it ==> Parser$$Lambda$77/0x0000000800ba3040@3498ed  
  
jshell> var itt = Parser.seq("zweiInt", it, ()->it)  
itt ==> Parser$$Lambda$77/0x0000000800ba3040@380fb434  
  
jshell> itt.parse(List.of("325", "57658", "678"))  
$78 ==> [Result[rest=[678], result=Node[name=zweiInt,  
  ↪ childNodes=[Terminal[image=325], Terminal[image=57658]]]]]
```

3.1 Grenzen des Ansatzes

3.1.1 Linksrekursion

3.1.2 Linker Präfix

4 Beispiel und Aufgabe

Jetzt soll die vorgestellte Parser-Kombinator Bibliothek genutzt werden, um einen Parser für eine kleine Programmiersprache zu definieren. Die Sprache wird Funktionsdefinitionen, und Ausdrücke für Arithmetik, veränderbare Variablen sowie if-Ausdrücke und while-Schleifen enthalten. Es soll ein abstrakter Syntaxbaum erzeugt werden.

4.1 Tokenizer

Bevor wir eine Eingabetliste von Token mit einem Parser verarbeiten können, benötigen wir einen Tokenizer, der diese Liste erzeugen kann. Der Tokenizer ist nicht Teil dieser Aufgabe und liegt als fertig implementierte Klasse vor.

Dabei ist ein Token eine Objekt, dass die eigentliche Token kapselt mit zusätzlichen Informationen über die Zeile und Spalte, in der das Token im Quelltext auftaucht, sowie dem Originalbild des Tokens im Quelltext. Letzteres wird benötigt, wenn der Tokenizer nach einem regulären Ausdruck ein bestimmtes Token erkannt hat, aber es sich bei dem

Token nicht um ein Schlüsselwort handelt, dass immer gleich aussieht, sondern zum Beispiel um einen beliebigen Bezeichner oder ein Zahlenliteral handelt. In diesem Fall ist das Token (Terminalzeichen) nur ein Token, das anzeigt, es handelt sich um einen Bezeichner oder eine Zahl, die konkrete Ausprägung ist dann im Attribut `image` gespeichert.

Java: Parser

```
public static record Token<T>(int l, int c, String image, T token) {
    @Override public boolean equals(Object obj) {
        return (obj instanceof Token t) && t.token == token;
    }
}
```

Eine Methode, die einen Parser für ein bestimmtes Token erkennt, lässt sich über die Methode `terminal` einfach definieren.

Java: Parser

```
static<T> Parser<Token<T>, Token<T>> token(T t) {
    return terminal(tok -> tok.token == t);
}
```

Für die Token der kleinen Programmiersprache wird eine Aufzählungsklasse definiert. Reguläre Ausdrücke geben für die einzelnen Token an, zu welcher Sprache sie gehören und welches Zeichen nach dem Token folgen kann.

Java: Parser

```
public static enum Tok {
    ifT("if\\W"), elseT("else\\W"), thenT("then\\W")
    , whileT("while\\W")
    , funT("fun\\W"), doT("do\\W")
    , lparT("\\(\\.", rparT("\\)\\.")
    , lbraceT("\\{\\.", rbraceT("\\}\\.")
    , commaT(",\\.", semicolonT(";\\.")
    , assignT(":=\\.", eqT("=\\.")
    , plusT("\\+\\.", minusT("-\\.", multT("\\*\\.", divT("/\\.", modT("%\\.")
    , identT("[\\_a-zA-Z]\\w*\\W")
    , intConstantT("(?:\\d+\\.|\\.|\\d\\.|\\d*(?:[Ee][-+]?\\d+)?\\.")
    ;
    public String regex;

    private Tok(String regex) {
        this.regex = regex;
    }
}
```

Mit dieser Aufzählungsklasse lässt sich mit der vorgegebenen Klasse `Tokenizer` ein Tokenizer implementieren, der genau diese Token erkennt und eine Liste von Token erstellen

kann.

Java: Parser

```
public static class LTokenizer extends Tokenizer<Tok> {  
    public LTokenizer() {  
        for (Tok tok:Tok.values()){  
            put(tok.regEx,tok);  
        }  
    }  
}
```

Entscheidend ist hier durchaus auch die Reihenfolge der Token in der Aufzählungsklasse. Der Tokenizer wird versuchen, das erste Token, dass mit seinem regulären Ausdruck matcht zu generieren. Deshalb ist es wichtig, dass die Token der Schlüsselwörter vor den Token für allgemeine Bezeichner in der Aufzählungsklasse definiert ist.

Wir können den so definierten Tokenizer auch im Javainterpreter schon einmal austesten.

Der Tokenizer wird instanziiert und ein String, der in Token der Sprache auszuteilen ist als Eingabe gesetzt:

Shell

```
jshell> import name.panitz.parser.*;  
  
jshell> var t = new Parser.LTokenizer()  
  
jshell> t.setInputString("fun f(x,y,z)={u:=42;if u==x then y else z;}")
```

Jetzt kann die Liste der Token generiert werden:

Shell

```
jshell> t.asList()
$4 ==> [Token[l=1, c=3, image=fun, token=funT], Token[l=1, c=4, image=f,
↪ token=identT], Token[l=1, c=5, image=(, token=lparT], Token[l=1, c=6,
↪ image=x, token=identT], Token[l=1, c=7, image=,, token=commaT],
↪ Token[l=1, c=8, image=y, token=identT], Token[l=1, c=9, image=,,
↪ token=commaT], Token[l=1, c=10, image=z, token=identT], Token[l=1,
↪ c=11, image=), token=rparT], Token[l=1, c=12, image==, token=eqT],
↪ Token[l=1, c=13, image={, token=lbraceT], Token[l=1, c=14, image=u,
↪ token=identT], Token[l=1, c=16, image:=, token=assignT], Token[l=1,
↪ c=18, image=42, token=intConstantT], Token[l=1, c=19, image=;,
↪ token=semicolonT], Token[l=1, c=21, image=if, token=ifT], Token ...
↪ Token[l=1, c=24, image==, token=eqT], Token[l=1, c=25, image=x,
↪ token=identT], Token[l=1, c=29, image=then, token=thenT], Token[l=1,
↪ c=30, image=y, token=identT], Token[l=1, c=34, image=else,
↪ token=elseT], Token[l=1, c=35, image=z, token=identT], Token[l=1,
↪ c=36, image=;, token=semicolonT], Token[l=1, c=37, image=},
↪ token=rbraceT]]
```

4.2 Klassen des abstrakten Syntaxbaums (AST)

Nachdem wir nun die Eingabe für den Parser in Form einer Liste von Token erhalten können, betrachten wir das Ergebnis des Parsers. Dieses soll ein abstrakter Syntaxbaum werden. Mit Hilfe der Recordklassen lassen sich die Knoten des AST als reine Datenhaltungsklassen definieren, die die Schnittstelle `AST` implementieren.

Java: Parser

```
static interface AST{}
static record OpExpr(AST l,BinOP op,AST r) implements AST{}
static record Var(String n) implements AST{}
static record IntLiteral(int n) implements AST{}
static record Assign(String name,AST r) implements AST{}
static record If(AST cond,AST then,AST otherwise) implements AST{}
static record While(AST cond,AST body) implements AST{}
static record Sequence(List<AST> sts) implements AST{}
static record FunCall(String name,List<AST> args) implements AST{}
```

Zwei weitere Datenklassen seien für Funktionsdefinitionen und für ein Programm gegeben:

Java: Parser

```
static record Fun(String name,List<String> args,AST body){}
static record Prog(List<Fun> funs,AST main){}
```

Die Klasse `OpExpr` hat ein Attribut für den binären Operator. Die binären Operatoren sind durch eine Aufzählungsklasse definiert:

Java: Parser

```
public static enum BinOP{
    add ((x,y)->x+y,"+"),
    sub ((x,y)->x-y,"-"),
    mult((x,y)->x*y,"*"),
    div((x,y)->x/y,"/"),
    mod((x,y)->x%y,"%"),

    eq  ((x,y)->x==y?1:0,"=");

    BinOP(BinaryOperator<Integer> op,String name){
        this.op = op;
        this.name = name;
    }
    BinaryOperator<Integer> op;
    String name;
}
```

4.3 Grammatik

Schließlich geben wir die Grammatik unserer Sprache an. Ein Programm besteht aus einer Folge von Funktionsdefinitionen und eine abschließende Anweisung, die ausgeführt werden soll. 0 bis *sn*-fache Wiederholungen sind mit einem *-Symbol gekennzeichnet.

$\langle prog \rangle \quad ::= \langle fundef \rangle^* \langle statement \rangle$

$\langle fundef \rangle \quad ::= \text{'fun' } \langle ident \rangle \text{'(' } \langle params \rangle \text{')' '=' } \langle statement \rangle$

$\langle params \rangle \quad ::= \langle ident \rangle \text{' ,' } \langle ident \rangle^*$
 $\quad \quad \quad | \quad \epsilon$

$\langle statement \rangle \quad ::= \langle whileStat \rangle$
 $\quad \quad \quad | \quad \langle assignStat \rangle$
 $\quad \quad \quad | \quad \langle blockStat \rangle$
 $\quad \quad \quad | \quad \langle expression \rangle$

$\langle whileStat \rangle \quad ::= \text{'while' '(' } \langle expression \rangle \text{')' } \langle statement \rangle$

$\langle ifStat \rangle$	$::= \text{'if'} \langle expression \rangle \text{'then'} \langle statement \rangle \text{'else'} \langle statement \rangle$
$\langle assignStat \rangle$	$::= \langle ident \rangle \text{' := ' } \langle expression \rangle$
$\langle blockStat \rangle$	$::= \text{'{' } \langle stats \rangle \text{'}'}$
$\langle stats \rangle$	$::= \langle statement \rangle (\text{' ; ' } \langle statement \rangle)^*$ $\langle statement \rangle$
$\langle expression \rangle$	$::= \langle ifStat \rangle$ $\langle cmpExpr \rangle$
$\langle cmpExpr \rangle$	$::= \langle addExpr \rangle \text{' = ' } \langle addExpr \rangle$ $\langle addExpr \rangle$
$\langle addExpr \rangle$	$::= \langle multExpr \rangle (\langle addOp \rangle \langle multExpr \rangle)^*$
$\langle addOp \rangle$	$::= \text{' + '}$ ' - '
$\langle multExpr \rangle$	$::= \langle atom \rangle (\langle multOp \rangle \langle atom \rangle)^*$
$\langle multOp \rangle$	$::= \text{' * '}$ ' / ' ' \% '
$\langle atom \rangle$	$::= \langle ident \rangle \text{' (' } \langle args \rangle \text{') '}$ $\langle ident \rangle$ $\langle number \rangle$ $\text{' (' } \langle expression \rangle \text{') '}$
$\langle args \rangle$	$::= \langle expression \rangle (\text{' , ' } \langle expression \rangle)^*$ $\langle expression \rangle$

4.4 Parser

Wir beginnen für die Grammatik den Parser zu definieren.

4.4.1 Konstante applicative Form

zunächst als kleine Optimierung sei eine Hilfsklasse gegeben, die die Schnittstelle `Supplier` implementiert. Bisher haben wir `Supplier` mit einem λ -Ausdruck erzeugt als `()->expr`. Wenn `expr` dabei eine komplexe Rechnung ist, so ist bei jedem Aufruf von `get()` diese Rechnung von neuem zu machen.

`Supplier` sind nichts anderes als 0-stellige Funktionen, auch konstante applicative Form genannt. Hierfür lässt sich nach einmaliger Berechnung des Ergebnisses dieses speichern und bei weiteren Aufrufen verwenden.

Für diesen Zweck sei die folgende Klasse definiert:

Java: Parser

```
static class CAF<X> implements Supplier<X>{
    CAF(Supplier<X> sup){this.sup=sup;}
    private X result = null;
    private Supplier<X> sup;
    @Override public X get(){
        if (result==null) result=sup.get();
        return result;
    }
}
```

Statt `()->expr` kann jetzt ein `Supplier` mit einem Speicher erzeugt werden durch:
`new CAF<>(()->expr).`

4.4.2 Regeln

Wie beginnen mit der Umsetzung der ersten Grammatikregeln. Dabei beginnen wir mit den Ausdrücken. Anweisungen und Funktionsdefinitionen, sowie die Startregel für ein komplettes Programm sind Teil der Aufgabe.

Bezeichner Ein beliebiger Bezeichner ist durch das Token `identT` gegeben. Hierfür sieht der `Supplier` des Parser wie folgt aus.

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> ident(){
    return new CAF<>(() -> token(Tok.identT).map(x->new Var(x.image)));
}
```

Zahlenlitterale Ebenso verfahren wir mit dem Token für Zahlenlitteralen:

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> number(){
    return new CAF<>(() ->
        token(Tok.intConstantT)
        .map(x->new IntLiteral(Integer.parseInt(x.image))));
}
```

Vergleichsausdrücke Die Regel der Vergleichsausdrücke lässt sich direkt in unsere Bibliothek übersetzen.

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> cmpExpr(){return new CAF<>(()->
    addExpr().get()
    .seq(()->token(Tok.eqT))
    .seq(addExpr())
    .map(x->(AST)new OpExpr(x.e1().e1,BinOP.eq,x.e2()))
    .alt(addExpr())
    .alt(ifStat())
    );
}
```

Additionsausdrücke Die Additionsausdrücke verwenden die Wiederholung.

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> addExpr(){return new CAF<>(()->
    multExpr().get()
    .seq(()->addOp().get().seq(multExpr()).zeroToN())
    .map(x->{
        AST r = x.e1;
        for (var snd:x.e2) r = snd.e1.apply(r, snd.e2);
        return r;
    }));
}
```

Die Additionsoperatoren sind die beiden operatoren der Addition und Subtraktion.

Java: Parser

```
static Supplier<Parser<Token<Tok>, BinaryOperator<AST>>> addOp(){
    BinaryOperator<AST> aop= (AST l,AST r)->(AST)new OpExpr(l,BinOP.add,r);
    BinaryOperator<AST> sop= (AST l,AST r)->(AST)new OpExpr(l,BinOP.sub,r);
    return new CAF<>(()->
        token(Tok.plusT).map(x-> aop)
        .alt(()->token(Tok.minusT).map(x->sop)));
}
```

Multiplikationsausdrücke Analog hierzu können die Ausdrücke der Punktrechnung umgesetzt werden:

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> multExpr(){return new CAF<>(()->
    atom().get()
    .seq(()->multOp().get().seq(atom()).zeroToN())
    .map(x->{
        AST r = x.e1;
        for (var snd:x.e2) {
            r = new OpExpr(r, BinOP.mult, snd.e2);
        }
        return r;
    })
    );
}
```

Hierzu gibt es drei verschiedene Operatorsymbole.

Java: Parser

```
static Supplier<Parser<Token<Tok>, BinaryOperator<AST>>> multOp(){
    BinaryOperator<AST> muop=(AST l,AST r)->(AST)new
        OpExpr(l,BinOP.mult,r);
    BinaryOperator<AST> dop= (AST l,AST r)->(AST)new OpExpr(l,BinOP.div,r);
    BinaryOperator<AST> moop=(AST l,AST r)->(AST)new OpExpr(l,BinOP.mod,r);
    return new CAF<>(()->
        token(Tok.multT).map(x-> muop)
        .alt(()->token(Tok.divT).map(x->dop))
        .alt(()->token(Tok.modT).map(x->moop))
    );
}
```

Atomare Ausdrücke Schließlich die Regel für atomare Ausdrücke, die entweder, Ein Funktionsaufruf, eine Variable, ein Zahlenliteral oder ein geklammerter Ausdruck sein

können.

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> atom(){return new CAF<>(()->
    funCall().get()
    .alt(ident())
    .alt(number())
    .alt(()->token(Tok.lparT)
        .seq(addExpr())
        .seq(()->token(Tok.rparT))
        .map(x->x.e1.e2))
    );
}
```

Für Funktionsaufrufe wurde eine eigene Regel definiert:

Java: Parser

```
static Supplier<Parser<Token<Tok>,AST>> funCall(){return new CAF<>(()->
    token(Tok.identT).map(x->x.image)
    .seq(()->token(Tok.lparT))
    .seq(args())
    .seq(()->token(Tok.rparT))
    .map(p->(AST)new FunCall(p.e1.e1.e1,p.e1.e2))
    );
}
```

Die Argumente eines Funktionsaufrufes werden schließlich durch die letzte Regel definiert.

Java: Parser

```
static Supplier<Parser<Token<Tok>,List<AST>>> args(){
    return new CAF<>(()->
        cmpExpr().get()
        .seq(()->token(Tok.commaT).seq(cmpExpr()).map(p->p.e2).zeroToN())
        .map(p->{p.e2.add(0,p.e1);return p.e2;})
        .alt(()->epsilon(List.of()))
    );
}
```

Damit ist die Grammatik für Ausdrücke vollständig umgesetzt. Einen ersten kleinen Test können wir auch hier wieder mit dem Javainterpreter durchführen-

Shell

```
jshell> import name.panitz.parser.*;

jshell> var tokenizer = new Parser.LTokenizer()
tokenizer ==>

jshell> tokenizer.setInputString("f(x,y,z)+2*(u-43)=g(g(g(1)))")

jshell> Parser.cmpExpr().get().parse(tokenizer.asList())

$5 ==> [Result[rest=[], result=OpExpr[l=OpExpr[l=FunCall[name=f,
↪ args=[Var[n=x], Var[n=y], Var[n=z]]], op=add,
↪ r=OpExpr[l=IntLiteral[n=2], op=mult, r=OpExpr[l=Var[n=u], op=sub,
↪ r=IntLiteral[n=43]]], op=eq, r=FunCall[name=g, args=[FunCall[name=g,
↪ args=[FunCall[name=g, args=[IntLiteral[n=1]]]]]]]]]]]
```

Aufgabe 1

Vervollständigen Sie den Parser um die fehlenden Regeln der Grammatik.

Java: Parser

```
static Supplier<Parser<Token<Tok>,Prog>> program(){
    return new CAF<>(()->null);
}
static Supplier<Parser<Token<Tok>,Fun>> fundef(){
    return new CAF<>(()->null);
}
static Supplier<Parser<Token<Tok>,AST>> statement(){
    return new CAF<>(()->null);
}
```

Die folgende schließende Klammer beendet die hier entwickelte Schnittstelle **Parser**.

Java: Parser

```
}
```

Literatur

- [Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions of Information Theory*, 2:113–124, 1956.
- [FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

- [Lei99] Leijen, Daan and Martini, Paolo and Latter, Antoine. parsec: Monadic parser combinators. <http://hackage.haskell.org/package/parsec>, 1999. [Online; accessed 17-April-2019].
- [NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.
- [OAC⁺04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.
- [OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 146–159. ACM Press, 1997.
- [Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2 edition, 2013.
- [Wad85] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 1985.