Sven Eric Panitz

Lehrbriefe
Funktionale Programmierung

5 Parserkombinatoren

Parser Kombinatoren in Haskell

Sven Eric Panitz

9. März 2025

Inhaltsverzeichnis

1 Grammatiken

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].

1.1 Kontextfreie Grammatiken

Eine kontextfreie Grammatik ist ein 4-Tupel $(\mathcal{T}, \mathcal{N}, S, \Pi)$ bestehend aus:

- einer Menge $\mathcal{T}$ von Wörtern, den *Terminalsymbolen*.
- einer Menge $\mathcal{N}$ von *Nichtterminalsymbolen*.
- ein ausgezeichnetes Startsymbol $S \in \mathcal{N}$.
- einer endlichen Menge Π von Regeln der Form:
  $nt \rightarrow t_1 \ldots t_n$, wobei $nt \in \mathcal{N}, t_i \in \mathcal{N} \cup \mathcal{T}$.

Mit den Regeln einer kontextfreien Grammatik werden Sätze gebildet, indem ausgehend vom Startsymbol Regel angewendet werden. Bei einer Regelanwendung wird ein Nichtterminalzeichen t durch die Rechte Seite einer Regel, die t auf der linken Seite hat, ersetzt.

1.2 Backus-Naur-Form und Erweiterungen

In der Informatik haben sich Notationen etabliert, um Grammatiken zu notieren. Sie basieren meist auf der sogenannten Backus-Naur-Form, die zur Definition von Algol verwendet wurde.

Nichtterminalsymbole werden dabei in spitzen Klammern eingeschlossen. Terminalsymbole in Anführungszeichen. Statt mehrere Regeln für ein Nichtterminalsymbol vorzusehen, gibt es eine Regel und eine Notation für Alternativen. Hierzu wird ein vertikaler Strich verwendet.

Da es jetzt Alternativen und Sequenzen gibt, können auf der rechten Seite einer Regel Teile geklammert werden.

Häufig werden in Programmiersprachen Wiederholungen benötigt. Dieses wird durch geschweifte Klammern ausgedrückt.

Ebenso gibt es Teile, die optional sind. Dieses wird durch eckige Klammern ausgedrückt.

[1] Chomsky 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.

1.3 Grammatik einer kleinen Skriptsprache

Als Beispiel für eine Grammatik in erweiterter Backus-Naur-Form definieren wir eine Grammatik für eine kleine imperative Skriptsprache.

In dieser Grammatik wurden Schlüsselwörter nicht in Anführungszeichen gesetzt. Lediglich Operatoren und Symbole als Terminale der Sprache sind in einfachen Anführungszeichen eingeschlossen.

⟨*program*⟩ ::= { ⟨*fundef*⟩ } ⟨*expr*⟩

⟨*fundef*⟩ ::= fun ⟨*ident*⟩ '(' [⟨*params*⟩] ')' = ⟨*body*⟩

⟨*params*⟩ ::= ⟨*ident*⟩ { ',' ⟨*ident*⟩ }

⟨*body*⟩ ::= '{' { ⟨*stat*⟩ } '}'
| ⟨*expr*⟩

⟨*stat*⟩ ::= ⟨*assignment*⟩ ';'
| ⟨*while*⟩
| ⟨*simpleExpr*⟩ ';'

⟨*assignment*⟩ ::= ⟨*ident*⟩ := ⟨*expr*⟩

⟨*while*⟩ ::= while '(' ⟨*expr*⟩ ')' ⟨*body*⟩

⟨*simpleExpr*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*ifExpr*⟩
| ⟨*booleanExpr*⟩

⟨*ifExpr*⟩ ::= if ⟨*expr*⟩ then ⟨*expr*⟩ else ⟨*expr*⟩

⟨*booleanExpr*⟩ ::= ⟨*compareExpr*⟩ { ('&&' | '||') ⟨*compareExpr*⟩ }

⟨*compareExpr*⟩ ::= ⟨*addExpr*⟩ [('==' | '/=' | '<=' | '>=' | '<' | '>') ⟨*compareExpr*⟩]

⟨*addExpr*⟩ ::= ⟨*multExpr*⟩ { ('+' | '-') ⟨*multExpr*⟩ }

⟨*multExpr*⟩ ::= ⟨*atom*⟩ { ('*' | '/' | '%') ⟨*atom*⟩ }

⟨*atom*⟩ ::= ‘(’ ⟨*expr*⟩ ‘)’
| ⟨*varOrFuncall*⟩
| ⟨*number*⟩

⟨*varOrFuncall*⟩ ::= ⟨*ident*⟩ [‘(’ [⟨*args*⟩]‘)’]

⟨*args*⟩ ::= ⟨*expr*⟩ { ‘,’ ⟨*expr*⟩}

2 Parser

Ein Parser ist ein Programm, dass für eine Grammatik checkt, ob die Eingabe des Parsers mit den Regeln der Grammatik erzeugt werden kann.

Prinzipiell kann ein Parser zwei Strategien verfolgen.

- Man kann ausgehend vom ersten Wort der Eingabe systematisch versuchen rückwärts die Regeln der Grammatik anzuwenden bis man zum Startsymbol gelangt. Ein solcher Parser wird als botton-up Parser bezeichnet.
- Man kann systematisch versuchen alle Sätze ausgehend vom Startsymbol zu bilden. Wenn man von links nach rechts gelesen dabei Terminale erzeugt, die nicht zum zu parsenden Satz gehören, wird eine andere Alternative versucht (Backtracking). Ein solcher Parser wird als top-down Parser bezeichnet

So allgemein ist also ein Parser durch eine vollständige Suche mit Backtracking implementiert. Man kann aber Grammatiken so konstruieren, dass immer nur eine Regelalternative zur zur parsenden Eingabe passt. Bei top-down Parsern ist dieses die LL(1) Eigenschaft einer Grammatik.

Wir werden nun eine Bibliothek entwickeln, die es erlaubt direkt Grammatiken als Haskell-Funktionen zu schreiben. Man erhält dadurch einen top-down Parser mit Backtracking.

3 Parserkombinatoren

Die Idee der Parserkombinatoren ist, dass jedes Nichtterminalsymbol zu einer Funktion wird, die Sätze, die mit diesem Symbol als Startsymbol gebildet wurden, parsen kann.

Die rechten Seite der Regeln für ein Nichtterminalsymbol kombinieren andere Symbole. Entweder als Sequenz, erst das eine dann das andere, oder als Alternative, entweder das eine oder das andere. Für diese Kombinationen werden Funktionen höherer Ordnung entwickelt, die als Operatoren definiert werden.

So lässt sich im Prinzip eine Grammatik direkt als Haskell-Code abschreiben.

Parserkombinatorbibliotheken sind ein 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.

OurParserLib.lhs

```
> module OurParserLib where
> import Data.Char
> import Data.Maybe
```

3.1 Der Parsertyp

Parser sind Funktionen. Sie verarbeiten einen Eingabeliste von Token, prüfen, ob diese Liste mit den Regeln einer Grammatik erzeugt werden kann. Es wird dann ein Ergebnis für diesen Parsvorgang konstruiert. Dieses Ergebnis ist ein Paar aus zwei Teilen: den restlichen Token, die durch den Parsvorgang noch nicht verarbeitet wurden und dem eigentlichen Ergebnis.

Da Grammatiken auch mehrdeutig sein können, werden unsere Parserfunktion nicht ein Parsergebnis aus einem solchen Paar erzeugen, sondern eine Liste von Parserergebnissen. Ist diese Liste leer, dann konnte nichts nach der Grammatik geparst werden. Hat die Liste mehr als ein Element, dann war die Grammatik mehrdeutig und es gab mehrere Möglichkeiten etwas aus der Eingabeliste zu parsen.

OurParserLib.lhs

```
> type ParsResult token result = (result,[token])
> type Parser token result = [token] -> [ParsResult token result]
```

3.2 Atomare Parserfunktionen

Die einfachsten Parser kombinieren nicht bestehende, sondern erkennen genau ein Zeichen. Dieses sind die Parser für die Terminalsymbolde der Grammatik.

3.2.1 Terminalsymbole erkennen

Der einfachste Parser ist der, der ein Terminalsymbol aus der Grammatik erkennt. Verallgemeinert ist das ein Parser zum Erkennen von Token mit bestimmter Eigenschaft.

OurParserLib.lhs

```
> satisfy :: (t -> Bool) -> Parser t t
> satisfy p  [] = []
> satisfy p (x:xs)
>  |p x = [(x,xs)]
>  |otherwise = []
```

Zum Erkennen eines Terminalsymbols soll das erste Zeichen gleich dem Terminalsymbol sein.

OurParserLib.lhs

```
> terminal :: Eq t => t ->  Parser t t
> terminal t = satisfy ((==) t)
```

3.2.2 Das leere Wort

Grammatiken enthalten manchmal das leere Wort bezeichnet als ϵ. Hierfür ist der Parser denkbar einfach definiert. Er konsumiert nichts vom Eingabestrom und erzeugt genau einen erfolgreichen Parsvorgang.

OurParserLib.lhs

```
> epsilon ::  Parser t [t]
> epsilon xs = [([],xs)]
```

3.3 Kombinatorfunktionen

Jetzt werden Möglichkeiten definiert, bestehende Parserfunktionen zu neuen zu kombinieren.

3.3.1 Infix Parserkombinatoren

Die Parserkombinatoren werden als Infixoperatoren definiert. Wir werden fünf Operatoren definieren, die unterschiedliche Präzedenzen haben.

OurParserLib.lhs

```
> infixl 9 +-
> infixl 9 +--
> infix  6 <<-
> infixl 3 !|+
> infixl 3 |+
```

3.3.2 Die Sequenz zweier Parser

Die wichtigste Kombination zweier Parser ist deren Sequenz. Es soll mit einem Parser geparst werden und anschließend die verbleibenden Token mit einem zweiten Parser. Der Ergebnistyp

ist dann ein Paar aus den beiden Ergebnissen der zwei Parser. kann schon mit dem ersten Parser nicht geparst werden, dann wird der zweite Parser gar nicht versucht.

Der Sequenzkombinator lässt sich in einer Zeile umsetzen:

OurParserLib.lhs

```
> (+-) :: Parser t r1 -> Parser t r2 -> Parser t (r1,r2)
> (+-) p1 p2 = \xs -> [((r1,r2),ts2)| (r1,ts1)<-p1 xs, (r2,ts2)<-p2 ts1]
```

Wir nutzen eine List-Comprehension. Zunächst werden alle Ergebnisse des ersten Parser betrachtet: `(r1,ts1) <- p1 xs`. Dann werden für deren Resttoken alle Ergebnisse des zweiten Parsers betrachtet: `(r2,ts2) <- p2 ts1`.

3.3.3 Die Alternativkombination

Wenn zwei Parser alternativ miteinander kombiniert werden, soll das bedeuten, versuche den einen oder versuche den anderen Parser. Das Ergebnis sind dann alle Ergebnisse des ersten Parsers und zusätzlich alle Ergebnisse des zweiten Parsers. Da die Parsergebnisse Listen sind, können diese einfach aneinander gehängt werden.

Wie man sieht, kann man auf diese Art und Weise nur zwei Parser mit demselben Ergebnistyp kombinieren.

OurParserLib.lhs

```
> (|+) :: Parser t r -> Parser t r -> Parser t r
> (|+) p1 p2 xs = p1 xs ++ p2 xs
```

Wir definieren einen zweiten Operator für die Alternative. Die zweite Version macht eine Art Shortcut. Wenn der erste Parser erfolgreich ist, wird der zweite ignoriert und nicht mehr probiert.

OurParserLib.lhs

```
> (!|+) :: Parser t r -> Parser t r -> Parser t r
> (!|+) p1 p2 xs
>   |null p1res =  p2 xs
>   |otherwise = p1res
>   where p1res = p1 xs
```

3.3.4 Map auf Parsers

Wenn wir das Ergebnis eines Parsvorgangs ändern wollen, dann soll für alle Parsvorgänge eine Funktion auf das Ergebnis angewendet werden. Das entspricht der klassischen `map` Funktion auf Listen, oder der Funktion `fmap` in der Typklasse `Functor`.

Insbesondere wenn wir die Sequenzen von Parsern haben, bekommen wir Tupel als Ergebnis, die wir gerne zu einem gemeinsamen Ergebnis weiterverarbeiten wollen.

Unsere Parser sind allerdings keine Instanz der Typklasse `Functor`. Stattdessen definieren wir einen Operator, der an einen Parser eine Funktion zur Manipulation des Ergebnisses anhängt.

OurParserLib.lhs

```
> (<<-) :: Parser t r1 -> (r1 -> r2) -> Parser t r2
> (<<-) p f = \xs ->  [(f result,rest) |(result,rest) <-  p xs]
```

3.3.5 Optionale

Manche Teile einer Grammatik sind optional. Sie können da sein, aber auch fehlen. Hierzu könnte man den Datentyp `Maybe` in Haskell verwenden. Der Einfachheit halber nutzen wir eine Liste als Ergebnis. Die leere Liste, wenn mit dem Parser nicht geparst werden konnte, die einelementige Liste sonst. Zur Implementierung lässt sich der epsilon-Parser für das leere Wort verwenden.

OurParserLib.lhs

```
> optional p
>   =     p       <<- (\x-> [x])
>     !|+ epsilon <<- (\_-> [])
```

3.3.6 0 bis n-fache Wiederholung

Während die Option die 0 bis 1-fache Wiederholung darstellt, benötigt man oft die 0 bis n-fache Wiederholung. Das Ergebnis ist eine Liste aller Parsergebnisse.

Die folgende Implementierung nutzt eine lokale Funktion mit einem Akkumulator für das Ergebnis.

OurParserLib.lhs

```
> zeroToN :: Parser t r -> Parser t [r]
> zeroToN  = zeroToN2 []
>   where
>     zeroToN2 acc p xs
>      | null res1 = [(reverse  acc,xs)]
>      | otherwise = zeroToN2 (r1: acc) p rest
>       where
>        res1 = p xs
>        (r1,rest) = head res1
```

1 bis n-fache Wiederholung Manchmal verlangt eine Wiederholung, dass ein Parser mindestens einmal erfolgreich gewesen sein muss.

OurParserLib.lhs

```
> oneToN :: Parser t r -> Parser t [r]
> oneToN p = p +- zeroToN p <<- \(x,xs)-> (x:xs)
```

Löschen von Whitespace Für einen Parser auf Strings, also mit Token Char, wird ein neuer Parser erzeugt, der Whitespace am Anfang der Tokenliste ignoriert.

OurParserLib.lhs

```
> ignoreSpace :: Parser Char r -> Parser Char r
> ignoreSpace p = \xs -> p (dropWhile isSpace xs)
```

Damit lässt sich eine Alternative Sequenz definieren, in der weißer Zwischenraum zwischen den Token, die die beiden Parser verarbeiten, ignoriert wird. Vor dem zweiten Parser ist Zwischenraum zu ignorieren.

OurParserLib.lhs

```
> (+--) p1 p2 = p1 +- ignoreSpace p2
```

Auch Wiederholungen, wenn wir auf einem String parsen, erlauben oft beliebig viele Leerzeichen, zwischen zwei Parsvorgängen.

OurParserLib.lhs

```
> rep p  = zeroToN (ignoreSpace p)
```

Filtern der Ergebnisse Folgende Funktion erlaubt es, aus der Ergebnisliste erfolgreiche Parsergebnisse zu löschen, wenn Sie eine bestimmte Eigenschaft nicht haben. Hilfreich, wenn etwas nicht durch kontextfreie Grammatik darstellbar, zum Beispiel verschiedene Tagnamen bei öffnenden und schließenden XML-Tags.

OurParserLib.lhs

```
> filtere :: (r -> Bool) -> Parser t r -> Parser t r
> filtere  pred p = \xs ->  filter (\(r,_) -> pred r) (p xs)
```

Wir werden diese Funktion im Rahmen dieser Aufgabe nicht verwenden.

4 Interpreter einer Skriptsprache

Nun soll an einem Beispiel die Kombinatorparser Bibliothek angewendet werden. Es soll für die Grammatik aus 1.3 ein Parser definiert werden. Der dabei entstandene Syntaxbaum wird dann mit einer Funktion ausgewertet.

4.1 Abstrakter Syntaxbaum

Zunächst der Datentyp für den Syntaxbaum. Ein Programm besteht aus einer Liste von Funktionsdefinitionen und einem Ausdruck, der auszuwerten ist.

OurParserLib.lhs

```
> data Prog = Prog [Fundef] Expr
>   deriving (Show,Eq)
```

Eine Funktionsdefinition hat einen Funktionsnamen, eine Liste von Parameternamen und als Rumpf eine Liste von Anweisungen.

OurParserLib.lhs

```
> data Fundef = Fun String [String] [Statement]
>   deriving (Show,Eq)
```

Wir kennen laut Grammatik drei verschiedene Anweisungen:

OurParserLib.lhs

```
> data Statement = Simple Expr
>  |While Expr [Statement]
>  |Assignment String Expr
>   deriving (Show,Eq)
```

Es gibt in der Sprache fünf verschiedene Ausdrücke.

OurParserLib.lhs

```
> data Expr =
>   Number Integer
>  |Variable String
>  |FunCall String [Expr]
>  |BinOp Expr Operator Expr
```

```
>  |IfExpr Expr Expr Expr
>   deriving (Eq,Show)
```

Wir haben dabei einen festen Satz an Operatoren.

OurParserLib.lhs

```
> data Operator = OR|AND|OEQ|NEQ|LE|GE|OLT|OGT|ADD|SUB|MULT|DIV|MOD
>   deriving (Eq,Show)
```

4.2 Parser

In der Folge soll ein Parser der Grammatik der kleinen Skriptsprache entwickelt werden.

4.2.1 Terminale

Die Terminalsymbole der Sprache sind die Schlüsselwörter, Bezeichner und Zahlenliterale. Für diese schreiben wir je einen Parser.

Beginnen wir mit Bezeichnern, die aus einer Folge von Buchstaben bestehen sollen, und natürlich nicht leer sein dürfen. Hier hilft die Funktion `isLetter` aus dem Modul `Data.Char`:

OurParserLib.lhs

```
> ident = oneToN (satisfy isLetter)
```

Zahlenliterale bestehen nur aus Ziffern. Hier hilft die Funktion `isDigit` aus dem Modul `Data.Char`. Die Ziffern lassen sich dann per `read` als ganze Zahl lesen.

OurParserLib.lhs

```
> number = oneToN (satisfy isDigit) <<- \xs -> (read xs)::Integer
```

Schließlich definieren wir noch eine Parserfunktion, die Schlüsselwörter erkennt. Ein Schlüsselwort besteht aus einem String, das ist eine Folge von Zeichen. Einzelnen Zeichen können mit der Funktion `terminal` der Parserbibliothek erkannt werden. Deshalb erzeugen wir mit `(map terminal xs)` die Liste der Parser für die einzelnen Zeichen der Liste. Diese werden gefaltet mit dem Sequenzoperator.

OurParserLib.lhs

```
> keyword xs
>   = (foldl (\p1 p2 -> p1 +- p2 <<- (\_ -> []))
>            epsilon
>            (map terminal xs))
>      <<- \_ -> xs
```

Testweise können wir diese Parser schon einmal in einer Interpretersession aufrufen:

ghci Session

```
OurParserLib> number "12345x"
[(12345,"x")]
OurParserLib> ident "xyz123"
[("xyz","123")]
OurParserLib> (ident+-number) "xyz123"
[(("xyz",123),"")]
OurParserLib> keyword "witzelbritz" "witzelbritz234  nhjkj"
[("witzelbritz","234  nhjkj")]
OurParserLib>
```

4.2.2 Ausdrücke

Jetzt setzen wir die Regeln der Grammatik eins zu eins als Parserfunktion um. Wir beginnen mit der Regel für Ausdrücke. Diese hat zwei Alternativen. Ein Ausdruck ist entweder ein if-Ausdruck oder ein bool'scher Ausdruck. Über den Alternativoperator lässt sich dieses direkt hinschreiben und sieht aus, wie in der eigentlichen Grammatik.

OurParserLib.lhs

```
> expr :: Parser Char Expr
> expr = ifExpr !|+ booleanExpr
```

If-Ausdrücke Kümmern wir uns zunächst um die if-Ausdrücke. Hier trennen laut Grammatik drei Schlüsselwörter die drei Unterausdrücke für Bedingung und die zwei Alternativen. Es ist also fünf Mal der Sequenzoperator zu bemühen, um eine Sequenz aus den 6 Symbolen der rechten Seite der Regel zu bilden. Zwischen Schlüsselwörtern und Ausdrücken kann beliebiger Zwischenraum stehen, so dass wir die für Stringeingabe spezialisierte Version der Sequenz verwenden, die dieses berücksichtigt.

OurParserLib.lhs

```
> ifExpr =
>   keyword "if"   +-- expr +--
>   keyword "then" +-- expr +--
>   keyword "else" +-- expr
>     <<- (\(((((_,c),_),a1),_),a2) -> IfExpr c a1 a2)
```

Das Ergebnis dieser fünffachen Sequenz sind fünffach verschachtelte Paare. Per *Pattern-Matching* im λ-Ausdruck identifizieren wir die Teilergebnisse, wobei wir die Ergebnisse für die Schlüsselwörter ignorieren. Das Ergebnis für den Syntaxbaum wird mit diesen Teilergebnissen dann erzeugt: `(IfExpr c a1 a2)`.

Bool'sche-Ausdrücke Setzen wir als nächstes die bool'schen Ausdrücke um.

Laut Grammatik beginnt ein bool'scher Ausdruck mit einem Vergleichsausdruck. Dann folgt eine 0 bis n-fache Wiederholung der Sequenz aus einem bool'schen Operator und einem weitern Vergleichsausdruck.

Mit den Mitteln unserer Parserbibliothek lässt sich dieses direkt ausdrücken. Wir müssen uns nur Gedanken machen, wie wir das Ganze zu einem Ausdruck des Syntaxbaum zusammen setzen. Das wird in einer Hilfsfunktion `mkOpEx` erledigt.

OurParserLib.lhs

```
> booleanExpr :: Parser Char Expr
> booleanExpr = compareExpr +-- (rep (booleanOperator+--compareExpr))
>    <<- (\(x,os) -> mkOpEx x os)
```

Zunächst aber kümmern wir uns um die Operatoren. Dieses sind genau zwei.

OurParserLib.lhs

```
> booleanOperator = pOp [("&&",AND),("||",OR)]
```

Für eine Liste aus Paaren von Operatornamen und Operatorwert, können wir mit dem Alternativoperator einen Parser definieren.

OurParserLib.lhs

```
> pOp ops = foldl1 (!|+) [keyword s <<- \_-> op |(s,op)<-ops]
```

Schließlich die Hilfsfunktion, um den Syntaxbaum zu konstruieren:

OurParserLib.lhs

```
> mkOpEx e = foldl (\e1 (op,e2)-> BinOp e1 op e2) e
```

Weitere Regeln als Aufgabe Soweit der Einstieg. Jetzt sind Sie dran. Die übrigen Regeln der Grammatik sind als Aufgaben umzusetzen.

Aufgabe 1 Schreiben Sie die Parserfunktion für das Nichtterminal <*compareExpr*>.

OurParserLib.lhs

```
> compareExpr :: Parser Char Expr
> compareExpr = \xs ->[]
```

Aufgabe 2 Schreiben Sie die Parserfunktion für das Nichtterminal <*addExpr*>.

OurParserLib.lhs

```
> addExpr :: Parser Char Expr
> addExpr = \xs ->[]
```

Aufgabe 3 Schreiben Sie die Parserfunktion für das Nichtterminal <*multExpr*>.

OurParserLib.lhs

```
> multExpr :: Parser Char Expr
> multExpr = \xs ->[]
```

Aufgabe 4 Schreiben Sie die Parserfunktion für das Nichtterminal <*atom*>.

OurParserLib.lhs

```
> atom :: Parser Char Expr
> atom = \xs ->[]
```

Aufgabe 5 Schreiben Sie die Parserfunktion für das Nichtterminal <*varOrFuncall*>.

OurParserLib.lhs

```
> varOrFunCall :: Parser Char Expr
> varOrFunCall = \xs ->[]
```

In einer Interpretersession sollten Sie jetzt in der Lage sein, Ausdrücke unserer kleinen Sprache zu parsen:

ghci Session

```
OurParserLib> expr  "(17+4)*  2 == 42 "
[(BinOp (BinOp (BinOp (Number 17) ADD (Number 4)) MULT (Number 2)) OEQ
(Number 42),"")]
OurParserLib> expr "if (17+4)* 2 == 42 ||19<=0 &&1/=56 then 42 else 18"
[(IfExpr (BinOp (BinOp (BinOp (BinOp (BinOp (Number 17) ADD (Number 4))
MULT (Number 2)) OEQ (Number 42)) OR (BinOp (Number 19) LE (Number 0)))
AND (BinOp (Number 1) NEQ (Number 56))) (Number 42) (Number 18),"")]
OurParserLib> expr  "f(x,1,f(2,g(x,f(y))),17+4)"
[(FunCall "f" [Variable "x",Number 1,FunCall "f" [Number 2,FunCall "g"
[Variable "x",FunCall "f" [Variable "y"]]],BinOp (Number 17) ADD (Numbe
r4)],"")]
OurParserLib>
```

4.2.3 Anweisungen

Widmen wir uns jetzt den Anweisungen unserer kleinen Skriptsprache. Wir kennen laut Grammatik drei Anweisungen, von denen zwei mit einem Semikolon enden.

OurParserLib.lhs

```
> stat :: Parser Char Statement
> stat = whileStat
>    !|+ assignment +--terminal ';' <<- fst
>    !|+ simpleExpr +--terminal ';' <<- fst
```

Aufgabe 6 Schreiben Sie die Parserfunktion für das Nichtterminal <*assignment*>.

OurParserLib.lhs

```
> assignment :: Parser Char Statement
> assignment = \xs ->[]
```

Aufgabe 7 Schreiben Sie die Parserfunktion für das Nichtterminal <*body*>.

OurParserLib.lhs

```
> body :: Parser Char [Statement]
> body = \xs ->[]
```

Aufgabe 8 Schreiben Sie die Parserfunktion für das Nichtterminal <*whileStat*>

OurParserLib.lhs

```
> whileStat :: Parser Char Statement
> whileStat = \xs ->[]
```

Aufgabe 9 Schreiben Sie die Parserfunktion für das Nichtterminal <*simpleExpr*>.

OurParserLib.lhs

```
> simpleExpr :: Parser Char Statement
> simpleExpr = \xs ->[]
```

Programm Es verbleibt das komplette Sktriptprogramm zu parsen, das aus einer Folge von Funktionsdefinitionen und einem auszuwertenden Ausdruck besteht

Aufgabe 10 Schreiben Sie die Parserfunktion für das Nichtterminal <*fundef*>.

OurParserLib.lhs

```
> fundef :: Parser Char Fundef
> fundef = \xs ->[]
```

Aufgabe 11 Schreiben Sie die Parserfunktion für das Nichtterminal <*prog*>.

OurParserLib.lhs

```
> prog :: Parser Char Prog
> prog = \xs ->[]
```

In einer Interpretersession sollten Sie jetzt in der Lage sein, Programme unserer kleinen Sprache zu parsen:

ghci Session

```
> prog "fun fac(n)={r:=1;while(n>0){r:=r*n;n:=n-1;} r;} fac(5)"
[(Prog [Fun "fac" ["n"] [Assignment "r" (Number 1),While (BinOp (Varia
ble "n") OGT (Number 0)) [Assignment "r" (BinOp (Variable "r") MULT (V
ariable "n")),Assignment "n" (BinOp (Variable "n") SUB (Number 1))],Si
mple (Variable "r")]] (FunCall "fac" [Number 5]),"")]
```

Interpreter Der erste Schritt zu einer eigenen kleinen Skriptsprache ist getan. Wir haben eine Syntax und einen dazugehörigen Parser, der für die entsprechende Grammatik einen Syntaxbaum erzeugt. Dabei wollen wir es nicht belassen, sondern die Programme, die wir parsen können, auch ausführen. Hierzu schreiben wir einen Interpreter, der über den Syntaxbaum geht und die entsprechenden Rechnungen ausführt.

Ausdrücke Auswerten Unsere Skriptsprache ist imperativ und hat veränderbare Variablen. Bei der Aufführung eines Programms müssen wir darüber Buch führen, welcher Wert jeweils in den Variablen gespeichert ist. Hierzu benötigen wir einen Map. Die einfachste Art einen Map zu implementieren, ist als Liste von Schlüssel-Wert-Paaren.

Entsprechend sei die Umgebung, in der wir unser Programm auswerten definiert. Unsere Skriptsprache soll als einzigen Datentypen Integer-Werte verwenden. Somit sind die Werte in der Umgebung der Auswertung vom Typ `Integer`.

OurParserLib.lhs

```
> type VarName = String
> type Env = [(VarName,Integer)]
```

Bevor wir uns um die Auswertung von Ausdrücken kümmern, definieren wir für die Operatorsymbole eine binäre Funktion. Da wir auf Integer-Werten rechnen, ist dieses für arithmetische Operatoren eine direkte Abbildung auf die Operatorfunktion.

OurParserLib.lhs

```
> getOp :: Operator -> Integer -> Integer -> Integer
> getOp ADD  = (+)
> getOp SUB  = (-)
> getOp MULT = (*)
> getOp DIV  = div
> getOp MOD  = mod
```

Für Vergleichsoperatoren und bool'sche Operatoren ist dieses nicht direkt möglich, da diese in Haskell bool'sche Ergebnistypen haben. Diese müssen wir als Zahl interpretieren. Der Typ `Bool` ist Instanz der Typklasse `Enum` und kann mit `fromEnum` zu einer ganzen Zahl, der Position in der Aufzählung umgewandelt werden. Umgekehrt kann aus einem Integer-Wert ein bool'scher Wert mit `fromIntegral` erhalten werden.

Wie man sieht, muss man sich gut in den Standardtypklassen der Basistypen auskennen.

OurParserLib.lhs

```
> getOp OEQ  = \x y -> toInteger$fromEnum (x==y)
> getOp NEQ  = \x y -> toInteger$fromEnum (x/=y)
> getOp GE   = \x y -> toInteger$fromEnum (x>=y)
> getOp LE   = \x y -> toInteger$fromEnum (x<=y)
> getOp OLT  = \x y -> toInteger$fromEnum (x<y)
> getOp OGT  = \x y -> toInteger$fromEnum (x>y)
> getOp AND  = \x y -> toInteger$fromEnum
>    ((toEnum$fromIntegral x)&&(toEnum$fromIntegral y))
> getOp OR   = \x y -> toInteger$fromEnum
>    ((toEnum.fromIntegral) x||(toEnum.fromIntegral) y)
```

In einer weiteren Hilfsfunktion suchen wir aus einer Liste von Funktionsdefinitionen nach einer bestimmten Funktion oder brechen mit einem Fehler ab.

OurParserLib.lhs

```
> getFun :: VarName -> [Fundef] -> Fundef
> getFun n [] = error ("function not defined "++n)
> getFun n (f@(Fun fn args body):fs)
```

```
>   |n==fn = f
>   |otherwise = getFun n fs
```

Damit kann die Auswertungsfunktion für Ausdrücke geschrieben werden. Wir werten innerhalb einer Umgebung mit Variablenbelegungen aus und haben zusätzlich die Liste der Funktionsdefinitionen.

OurParserLib.lhs

```
> eval :: Env -> [Fundef] -> Expr -> Integer
```

Für Zahlenliterale ist die Auswertung denkbar einfach. Das Ergebnis ist die dargestellte Zahl des Literals.

OurParserLib.lhs

```
> eval _ _ (Number i) = i
```

Für Variablen muss die Variable in der Umgebung nachgeschlagen werden. Hierzu gibt es die Standardlistenfunktion `lookup` im Prelude.

OurParserLib.lhs

```
> eval env _ (Variable s) = fromJust $lookup s env
```

Aufgabe 12 Implementieren Sie die Auswertungsfunktion für die verbleibenden drei Arten von Ausdrücken.

OurParserLib.lhs

```
> eval env fs (IfExpr c a1 a2) = 0
> eval env fs (BinOp left op right) = 0
> eval env fs (FunCall n args) = 0
```

Tipp: Für die Auswertung eines Funktionsaufrufs benötigen Sie die erst im späteren Verlauf implementierte Funktion `runStats`. Sie sollten bei diesem Aufruf eine komplett neue Umgebung erzeugen, in der die formalen Parameter der Funktion den Wert der konkreten Argumente des Aufrufs bekommen. Die neue Umgebung können Sie gut mit der Standardfunktion `zip` erreichen.

In einer Interpretersession sollten Sie jetzt in der Lage sein, Ausdrücke unserer kleinen Sprache zu parsen und auszuwerten:

ghci Session

```
OurParserLib> eval [][]$fst$head$expr "17+4*2"
25
OurParserLib> eval [("x",42)][]$fst$head$expr "17+x*2"
101
```

Anweisungen Auswerten Schließlich sollen Anweisungen ausgeführt werden. Anders als Ausdrücke, die nur zu einem Zahlenergebnis auswerten, können Anweisungen die Umgebung verändern, indem einer Variablen ein neuer Wert zugewiesen wird. Daher ist das Ergebnis der Ausführung einer Anweisung nicht nur eine Zahl, sondern die Umgebung nach Ausführung der Anweisung:

OurParserLib.lhs

```
> run :: Env -> [Fundef] -> Statement -> (Env,Integer)
```

Wenn die Anweisung nur ein einfacher Ausdruck ist, so wird die Umgebung nicht verändert und das Ergebnis dieser Auswertung genommen.

OurParserLib.lhs

```
> run env fs (Simple e) = (env,eval env fs e)
```

Anders sieht es bei der Zuweisung aus. Die Variable erhält einen neuen Wert. Wir hängen diesen einfach vorne an die Umgebung an. Dann wird der neue Wert vor dem alten gefunden.

OurParserLib.lhs

```
> run env fs (Assignment v e) = ((v,r):env,r)
>   where r = eval env fs e
```

Aufgabe 13 Implementieren Sie die Ausführungsfunktion für die while-Schleife.

OurParserLib.lhs

```
> run env fs w@(While c body) = (env,0)
```

Sowohl Funktionsrümpfe als auch der Rumpf einer while-Schleife bestehen aus einer Liste von Anweisungen. Diese kann nun mit folgender Funktion ausgeführt werden.

OurParserLib.lhs

```
> runStats env fs [stat] = run env fs stat
> runStats env fs (st:sts) = runStats (fst$run env fs st) fs sts
```

Ein komplettes Programm kann nun ausgeführt werden.

OurParserLib.lhs

```
> runProg (Prog fs e) = eval [] fs e
```

In einer Interpretersession sollten Sie jetzt in der Lage sein, Programme unserer kleinen Sprache zu parsen und auszuführen:

ghci Session

```
> (runProg.fst.head.prog)
    "fun fac(n)={r:=1;while(n>0){r:=r*n;n:=n-1;} r;} fac(5)"
120
```

Aufgabe 14 Eine Parserkombinatorbibliothek gibt es auch im Standard-API von Haskell.[Lei99]. Machen Sie sich mit dem Modul vertraut. Schauen Sie sich die Implementierung an und vergleichen Sie diese mit der Implementierung unserer Aufgabe.

Projektidee 1

Sie haben den ersten Schritt zu einem Compiler bewältigt. Wie wäre es eine eigene kleine Programmiersprache zu entwickeln und einen Compiler für diese zu implementieren? Sind Sie an der Implementierung einer lazy-evaluierten funktionalen Sprache interessiert, so sei das folgendes Tutorial empfohlen[JLPJ92].

5 Lernzuwachs

Folgende Erkenntnisse sollte sich nach Studium dieses Kurses gesammelt haben.

- mit Funktionen höherer Ordnung und Operatoren lassen sich gut domainspezifische Bibliotheken fast wie eine eigene Sprache in Haskell definieren.
- Currying ist eine ausdrucksstarke Methodik. Wir haben oft Funktionen nur auf einen Teil ihrer Argumente angewendet und so neue Funktionen erhalten.
- Typklassen auf numerischen und Aufzählungstypen insbesondere deren Konvertierungsfunktion sollten gut studiert werden.

Literatur

[Cho56] N. Chomsky, “Three models for the description of language,” *IRE Transactions of Information Theory*, vol. 2, pp. 113–124, 1956.

[FL89] R. Frost and J. Launchbury, “Constructing natural language interpreters in a lazy functional language,” *The Computer Journal*, vol. 32, no. 2, pp. 108–121, 1989.

[JLPJ92] S. P. Jones, D. Lester, and S. Peyton Jones, *Implementing functional languages: a tutorial*. Prentice Hall, January 1992. [Online]. Available: https://www.microsoft.com/en-us/research/publication/implementing-functional-languages-a-tutorial/

[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] P. Naur and J. Backus, “Report on the algorithmic language ALGOL 60,” *Communications of the ACM*, vol. 3, no. 5, pp. 299–314, may 1960.

Atomare Parser
Alternativkombinator
Sequenzkombinator
Wiederholungen
Map auf Ergebnis