Deskriptive Programmierung

Parserkombinatoren

Zur Erinnerung, 2. Vorlesung: Verarbeitung arithmetischer Ausdrücke

- Wir wollten gültige arithmetische Ausdrücke beschreiben, zum Beispiel angelehnt an eine formale Grammatik:

```
expr   ::=  term + expr | term
term   ::=  factor * term | factor
factor ::=  nat | (expr)
```

- … um dann zum Beispiel sowas zu können:

```
data Expr = Lit Int | Add Expr Expr | Mul Expr Expr
```

```
parse :: String → Expr
???
```

```
> parse "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

```
eval :: Expr → Int
…
```

```
calc :: String → Int
calc s = eval (parse s)
```

```
> calc "2+3*5"
17
```

Zur Erinnerung, 2. Vorlesung: Verarbeitung arithmetischer Ausdrücke

- Wir hatten schon diskutiert, dass man natürlich die Regeln (zu Vorrang etc.) der Grammatik berücksichtigen muss, dann „naiv“ könnte man "2+3*5" ja statt als Add (Lit 2) (Mul (Lit 3) (Lit 5)) auch als Mul (Add (Lit 2) (Lit 3)) (Lit 5) lesen.

- Gezeigt bzw. erstmal nur behauptet hatte ich, dass man die Grammatik selbst als „Programm“ ausdrücken kann:

```
expr   =  ( Add  <$>  term  <*  char '+'  <*>  expr )  |||  term

term   =  ( Mul  <$>  factor  <*  char '*'  <*>  term )  |||  factor

factor =  ( Lit  <$>  nat )  |||  ( char '('  *>  expr  <*  char ')' )
```

und dann erhält:

```
>  parse expr "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

Zur Erinnerung, 2. Vorlesung: Verarbeitung arithmetischer Ausdrücke

- Genau das wollen wir jetzt realisieren.
- Da ähnliche Problemstellungen natürlich in vielen Bereichen auftauchen; wollen wir eine möglichst allgemeine Lösung entwickeln.
- Zum Beispiel eine Bibliothek:

```
type Parser = …

parse :: Parser → String → …

integer, identifier :: Parser

(| | |) :: Parser → Parser → Parser

(+++) :: Parser → Parser → Parser
…
```

… wobei wir die notwendigen Kombinatoren für die spezielle syntaktische Form des Beispiels (<$>, <*>, <*, *>) zunächst noch zurückstellen.

Design einer Parser-Bibliothek

- Schon die zweiten „…“ in:

```
type Parser = …

parse :: Parser → String → …
```

zeigen, dass man wohl etwas allgemeiner starten muss.

- Besser:

```
type Parser a = …

parse :: Parser a → String → a

integer :: Parser Int

identifier :: Parser String

(| | |) :: Parser a → Parser a → Parser a

(+++) :: Parser a → Parser b → Parser ?
```

Design einer Parser-Bibliothek

- Was wäre nun eine gute Repräsentation für

```
type Parser a = …
```

?

- Naheliegende Idee, unter Verwendung von Higher-Order:

```
type Parser a = String → a
```

- Aber im Allgemeinen könnte ein (Teil-)Parser nur einen Teil der Eingabe „verbrauchen“, es bliebe also ein Rest-String übrig.

```
type Parser a = String → (a, String)
```

- Und wir sollten darauf vorbereitet sein, dass ein Parse-Versuch auch fehlschlagen kann.

```
type Parser a = String → Maybe (a, String)
```

Design einer Parser-Bibliothek

- Wie ist für

```
type Parser a = String → Maybe (a, String)
```

nun die Funktion

```
parse :: Parser a → String → a
```

zu implementieren?
(Zur Verwendung nachdem man einen Parser mittels der anderen Operationen der Bibliothek entwickelt hat.)

- Relativ natürlich:

```
parse :: Parser a → String → a
parse p inp = case p inp of
                Nothing       → error "invalid input"
                Just (x, "")  → x
                Just (_, rest) → error ("unused input: " ++ rest)
```

Design einer Parser-Bibliothek

- Machen wir uns nun an die Implementierung einiger „Parser-Bausteine“:

```
item :: Parser Char
item = \inp → case inp of
                ""     → Nothing
                c : cs → Just (c, cs)
```

- Dann:

```
> parse item "a"
'a'

> parse item "b"
'b'

> parse item ""
Program error: invalid input

> parse item "ab"
Program error: unused input: b
```

Design einer Parser-Bibliothek

- Es geht auch etwas wählerischer:

```
digit :: Parser Char
digit = \inp → case inp of
                 ""      → Nothing
                 c : cs  → if '0' <= c && c <= '9' then Just (c, cs) else Nothing
```

- Dann:

```
> parse digit "a"
Program error: invalid input

> parse digit "5"
'5'

> parse digit ""
Program error: invalid input

> parse digit "ab"
Program error: invalid input
```

Design einer Parser-Bibliothek

- Analog:

```
lower :: Parser Char
lower = …
```

- Zum Ausdrücken von Alternativen:

```
infixr 3 |||

(|||) :: Parser a → Parser a → Parser a
p ||| q = …
```

- Gewünscht:

```
> parse (digit ||| lower) "a"
'a'

> parse (digit ||| lower) "5"
'5'
```

- Implementierung des Auswahl-Operators:

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                   Just (x, rest)  →  Just (x, rest)
                   Nothing         →  q inp
```

- Der Operator ist assoziativ (daher macht auch die Deklaration „infixr 3 |||“ Sinn), und es gibt ein neutrales Element:

```
failure :: Parser a
failure = \inp → Nothing
```

```
> parse failure "a"
Program error: invalid input

> parse (failure ||| lower) "a"
'a'
```

Design einer Parser-Bibliothek

- Implementierung eines Operators zur Konkatenation von Parsern:

```
(+++) :: Parser a → Parser b → Parser ?
p +++ q = …
```

- Denkbar wäre `(+++) :: Parser a → Parser b → Parser (a, b)` , aber dann wäre es nicht möglich, Abhängigkeitsbedingungen auszudrücken.

- Stattdessen:

```
(++>) :: Parser a → (a → Parser c) → Parser c

(+++) :: Parser a → Parser b → Parser b
```

- Dann zum Beispiel ausdrückbar:

```
digit  ++>  \d → if d < '5' then digit else lower
```

- Implementierung der beiden Operatoren zur Konkatenation von Parsern:

```
infixr 4 +++, ++>

(+++) :: Parser a → Parser b → Parser b
p +++ q = \inp → case p inp of
                   Nothing      → Nothing
                   Just (_, rest) → q rest

(++>) :: Parser a → (a → Parser b) → Parser b
p ++> f = \inp → case p inp of
                   Nothing      → Nothing
                   Just (x, rest) → f x rest
```

- Tatsächlich, für den Operator, der das Ergebnis des ersten Parsers ignoriert:

```
(+++) :: Parser a → Parser b → Parser b
p +++ q = p ++> \_ → q
```

- Können wir dennoch eine Konkatenation implementieren, bei der beide Ergebnisse kombiniert werden?

- Zum Beispiel zum Parsen eines Kleinbuchstaben und einer Ziffer, und Rückgabe beider Komponenten:

```
> parse pair "a1"
('a', '1')
```

- Möglich, unter Verwendung einer weiteren Primitive:

```
yield :: a → Parser a
yield x = \inp → Just (x, inp)
```

- Dann nämlich:

```
pair = lower ++> \x → digit ++> \y → yield (x, y)
```

- Das scheint ein allgemein nützliches Kombinationsprinzip zu sein, daher Abstraktion in eine extra Hilfsfunktion:

```
liftP :: (a → b → c) → Parser a → Parser b → Parser c
liftP f p q  =  p ++> \x → q ++> \y → yield (f x y)
```

- Dann:

```
>  parse (liftP (,) lower digit) "a1"
('a', '1')

>  parse (liftP (,) digit lower) "a1"
Program error: invalid input

>  parse (liftP max lower lower) "nm"
'n'
```

Design einer Parser-Bibliothek

- Wenn wir schon mal beim Abstrahieren sind:

```
mapP :: (a → b) → Parser a → Parser b
mapP f p  =  p ++> \x → yield (f x)
```

- Dann etwa:

```
digitAsInt :: Parser Int
digitAsInt  =  mapP (\d → length ['1' .. d]) digit
```

- Sowie:

```
sat :: (Char → Bool) → Parser Char
sat p  =  item ++> \x → if p x then yield x else failure
```

- Dann etwa:

```
digit :: Parser Char
digit = sat isDigit                                  -- isDigit :: Char → Bool
```

```
char :: Char → Parser ()
char x  =  sat (== x) +++ yield ()
```

Parser-Bibliothek: Zwischenbilanz

```
parse :: Parser a → String → a

item, digit, lower :: Parser Char

yield :: a → Parser a

failure :: Parser a

(|||) :: Parser a → Parser a → Parser a

(++>) :: Parser a → (a → Parser b) → Parser b

(+++) :: Parser a → Parser b → Parser b

liftP :: (a → b → c) → Parser a → Parser b → Parser c

mapP :: (a → b) → Parser a → Parser b

sat :: (Char → Bool) → Parser Char
```

Design einer Parser-Bibliothek

- Eigentlich wollen wir im Beispiel ja (unter anderem) natürliche Zahlen parsen.
- Wir könnten wie folgt vorgehen:

nat_1 :: Parser Int
nat_1 = digitAsInt

nat_2 = digitAsInt ++> \$d_1$ → digitAsInt ++> \$d_2$ → yield (10 * d_1 + d_2)

nat_3 = digitAsInt ++> \$d_1$ → nat_2 ++> \$n_2$ → yield (100 * d_1 + n_2)

nat_4 = liftP (\d_1 n_3 → 1000 * d_1 + n_3) digitAsInt nat_3

…

nat :: Parser Int
nat = nat_9 ||| nat_8 ||| nat_7 ||| nat_6 ||| nat_5 ||| nat_4 ||| nat_3 ||| nat_2 ||| nat_1

- Das ist natürlich nicht wirklich befriedigend!

Design einer Parser-Bibliothek

- Es wäre gut, allgemeine Wiederholungen ausdrücken zu können.
- Also:

```
many :: Parser a → Parser [a]
many p  =  (p ++> \x → many p ++> \xs → yield (x : xs))
           |||  yield [ ]

many1 :: Parser a → Parser [a]
many1 p  =  p ++> \x → many p ++> \xs → yield (x : xs)
```

- Oder, äquivalent:

```
many :: Parser a → Parser [a]
many p = many1 p ||| yield [ ]

many1 :: Parser a → Parser [a]
many1 p = liftP (:) p (many p)
```

Parsen arithmetischer Ausdrücke

Nun, zum Parsen natürlicher Zahlen:

1. Versuch:

```
nat :: Parser [Int]
nat = many1 digitAsInt
```

```
> parse nat "123"
[1, 2, 3]
```

2. Versuch, mit Nachbearbeitung:

```
nat :: Parser Int
nat = mapP (foldl (\n d → 10 * n + d) 0) (many1 digitAsInt)
```

```
> parse nat "123"
123
```

Parsen arithmetischer Ausdrücke

- Ausgangsspezifikation/Grammatik zum Erkennen arithmetischer Ausdrücke als Ganzes:

```
expr    ::=  term + expr | term
term    ::=  factor * term | factor
factor  ::=  nat | (expr)
```

- Nun, Umsetzung:

```
expr :: Parser ()
expr = term +++ char '+' +++ expr  |||  term

term :: Parser ()
term = factor +++ char '*' +++ term  |||  factor

factor :: Parser ()
factor = nat +++ yield ()  |||  char '(' +++ expr +++ char ')'
```

- Test:

```
>  parse expr "2+3*5"
()
```

Parsen arithmetischer Ausdrücke

- Wir wollen natürlich auch die Ergebnisse des Parsens sehen, also:

```
expr :: Parser Expr
expr  =  (term  ++>  \t → char  '+'  +++  expr  ++>  \e → yield  (Add t e))
         |||  term

term :: Parser Expr
term  =  (factor  ++>  \f → char  '*'  +++  term  ++>  \t → yield  (Mul f t))
         |||  factor

factor :: Parser Expr
factor  =  (nat  ++>  \n → yield (Lit n))
           |||  char  '('  +++  expr  ++>  \e → char ')'  +++  yield e
```

- Oder, unter geeigneter Nutzung der eingeführten Higher-Order Funktionen:

```
expr    =  liftP Add term (char '+' +++ expr)  |||  term
term    =  liftP Mul factor (char '*' +++ term)  |||  factor
factor  =  mapP Lit nat  |||  char '(' +++ expr ++> \e → char ')' +++ yield e
```

Parsen arithmetischer Ausdrücke

- Tests:

```
> parse expr "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

```
> parse expr "2*3+5"
Add (Mul (Lit 2) (Lit 3)) (Lit 5)
```

- Also mit:

```
calc :: String → Int
calc s = eval (parse expr s)
```

dann:

```
> calc "2+3*5"
17
```

```
> calc"2*3+5"
11
```

„Lektion“:

- allgemeine Parser-Bibliothek (ParserCore.hs, Parser.hs):

```
type Parser a      parse     item      yield     failure   (| | |)   (++>)
```

```
(+++)    digit    lower    …      sat      liftP    mapP

many     many1   char
```

- darauf aufbauend, Parser für spezifische Anwendungen, z.B. (Calc.hs):

```
import ParserCore
import Parser

data Expr

nat                factor             term               expr
```

Ein zusätzliches Interface zur Parser-Bibliothek

Etwas „(gar nicht so) dunkle Magie“ (MParserCore.hs):

```
import qualified ParserCore

newtype Parser a  =  P (ParserCore.Parser a)
unP :: Parser a → ParserCore.Parser a
unP (P p)  =  p

parse :: Parser a → String → a
parse  =  ParserCore.parse . unP

item :: Parser Char
item  =  P ParserCore.item
...

instance Monad Parser where
  return  =  yield
  (>>=)  =  (++>)
  fail _  =  failure
```

Ein zusätzliches Interface zur Parser-Bibliothek

Nun Verwendung von do-Blöcken möglich (nicht erzwungen), zum Beispiel:

```
term :: Parser Expr
term = do f ← factor
          char '*'
          t ← term
          return (Mul f t)
       ||| factor
```

```
factor :: Parser Expr
factor = mapP Lit nat
         ||| do char '('
                e ← expr
                char ')'
                return e
```

statt:

```
term :: Parser Expr
term = (factor ++> \f → char '*' +++ term ++> \t → yield (Mul f t))
       ||| factor

factor :: Parser Expr
factor = mapP Lit nat
         ||| char '(' +++ expr ++> \e → char ')' +++ yield e
```

Ein zusätzliches Interface zur Parser-Bibliothek

- Allgemeine Regeln zur Verwendung von do-Notation für unsere Parser-Bibliothek:

$$\begin{array}{l}\text{do } prs_1\\ \quad x_2 \leftarrow prs_2\\ \quad x_3 \leftarrow prs_3\\ \quad prs_4\\ \quad x_5 \leftarrow prs_5\\ \quad \ldots\end{array}$$

Der do-Block insgesamt hat den Typ des letzten prs_n. Zu diesem ist generell kein x_n vorhanden.

wobei die prs_i jeweils einen Typ der Form Parser t_i haben und an die x_i (sofern explizit vorhanden) dann jeweils ein Wert des Typs t_i gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das durch prs_i gelieferte Ergebnis.

- Es handelt sich tatsächlich nur um „syntaktischen Zucker“. Obiges Beispiel würde automatisch umgewandelt in:

$$prs_1 \mathbin{+\!\!+\!\!+} (prs_2 \mathbin{+\!\!+\!\!>} (\backslash x_2 \rightarrow prs_3 \mathbin{+\!\!+\!\!>} (\backslash x_3 \rightarrow prs_4 \mathbin{+\!\!+\!\!+} (prs_5 \mathbin{+\!\!+\!\!>} (\backslash x_5 \rightarrow \ldots)))))$$