Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

College 2012-2013
10. Ingebedde talen

Doaitse Swierstra

Utrecht University

October 15, 2012

Vandaag:

1. combinator talen
2. parser combinators
3. backtracking & list of successes methode
4. rekenmachines

Combinator languages

In plaats van voor elk toepassingsgebied een speciale, nieuwe taal te definiëren, handelen we als volgt:

- definieer een bibliotheek met

Combinator languages

In plaats van voor elk toepassingsgebied een speciale, nieuwe taal te definiëren, handelen we als volgt:

- definieer een bibliotheek met
  1. primitieve concepten

Combinator languages

In plaats van voor elk toepassingsgebied een speciale, nieuwe taal te definiëren, handelen we als volgt:

- definieer een bibliotheek met
  1. primitieve concepten
  2. samenstellingsmechanismen

Combinator languages

In plaats van voor elk toepassingsgebied een speciale, nieuwe taal te definiëren, handelen we als volgt:

- definieer een bibliotheek met
  1. primitieve concepten
  2. samenstellingsmechanismen
- zorg dat de bibliotheek er een beetje "lekker" uitziet, door bijv een mooie verzameling operatoren en bijbehorende prioriteiten te kiezen

Combinator languages

In plaats van voor elk toepassingsgebied een speciale, nieuwe taal te definiëren, handelen we als volgt:

- definieer een bibliotheek met
  1. primitieve concepten
  2. samenstellingsmechanismen
- zorg dat de bibliotheek er een beetje "lekker" uitziet, door bijv een mooie verzameling operatoren en bijbehorende prioriteiten te kiezen
- omdat de betekenis van een taal vaak met behulp van een functie kan worden uitgedrukt lenen met name functionele talen zich bij uitstek voor deze methode

Een expressie ontleden

We willen functies schrijven die ons in staat stellen een expressie in te lezen en als een *Boom* op te slaan (zogenaamde Parsers of Ontleders).

Een expressie ontleden

We willen functies schrijven die ons in staat stellen een expressie in te lezen en als een *Boom* op te slaan (zogenaamde Parsers of Ontleders). We lezen de expressie stukje voor stukje in, en houden dus telkens een kleiner stukje van de invoer over.

Een expressie ontleden

We willen functies schrijven die ons in staat stellen een expressie in te lezen en als een *Boom* op te slaan (zogenaamde Parsers of Ontleders). We lezen de expressie stukje voor stukje in, en houden dus telkens een kleiner stukje van de invoer over. Voor het type van een inleesfunctie zouden we dus kunnen kiezen:

$$\textbf{type}\ Parser = String \rightarrow (Boom, String)$$

Een expressie ontleden

We willen functies schrijven die ons in staat stellen een expressie in te lezen en als een *Boom* op te slaan (zogenaamde Parsers of Ontleders). We lezen de expressie stukje voor stukje in, en houden dus telkens een kleiner stukje van de invoer over. Voor het type van een inleesfunctie zouden we dus kunnen kiezen:

$$\textbf{type}\ Parser = String \rightarrow (Boom, String)$$

Omdat we niet vast willen leggen wat we precies aan het herkennen zijn kunnen we beter een polymorf type definiëren:

$$\textbf{type}\ Parser\ a = String \rightarrow (a, String)$$

Ontleden kan falen, of op meerder manieren slagen

Sommige strings kunnen we echter op meerdere manieren ontleden. We maken dus het type van de ontleedfuncties zó, dat ze niet één boom-met-reststring opleveren, maar alle mogelijke manieren waarop dat kan:

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

Ontleden kan falen, of op meerder manieren slagen

Sommige strings kunnen we echter op meerdere manieren ontleden. We maken dus het type van de ontleedfuncties zó, dat ze niet één boom-met-reststring opleveren, maar alle mogelijke manieren waarop dat kan:

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

We simuleren op deze manier niet-determinisme.

Ontleden kan falen, of op meerder manieren slagen

Sommige strings kunnen we echter op meerdere manieren ontleden. We maken dus het type van de ontleedfuncties zó, dat ze niet één boom-met-reststring opleveren, maar alle mogelijke manieren waarop dat kan:

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

We simuleren op deze manier niet-determinisme. Als voorbeeld schrijven we een parsefunctie die de letter 'a' herkent:

$$\begin{array}{lll} \mathit{pLettera} & & :: \mathit{Parser\ Char} \\ \mathit{pLettera}\ [\,] & & = [\,] \\ \mathit{pLettera}\ (x : xs) & \mid x \equiv \texttt{'a'} & = [(\texttt{'a'}, xs)] \\ & \mid \mathit{otherwise} & = [\,] \end{array}$$

En als het niet lukt?

Het komt goed uit dat we een *lijst* van ontledingen opleveren, zodat we de lege lijst kunnen opleveren als ontleding niet mogelijk is (omdat de invoer niet met een ‘a’ begint).

En als het niet lukt?

Het komt goed uit dat we een *lijst* van ontledingen opleveren, zodat we de lege lijst kunnen opleveren als ontleding niet mogelijk is (omdat de invoer niet met een 'a' begint). Deze functie maken we natuurlijk parametriseerbaar, zodat hij gebruikt kan worden om elke letter te herkennen:

$$
\begin{array}{lll}
\mathit{pLetter} & & :: \mathit{Char} \rightarrow \mathit{Parser\ Char} \\
\mathit{pLetter}\ a\ [\,] & & = [\,] \\
\mathit{pLetter}\ a\ (x : xs) & \mid x \equiv a & = [\,(a, xs)\,] \\
 & \mid \mathit{otherwise} & = [\,] \\
\mathit{pLettera} & & = \mathit{pLetter}\ \texttt{'a'}
\end{array}
$$

Woordjes herkennen

Een andere elementaire ontleedfunctie is een functie die een string herkent, bijvoorbeeld het woord 'begin'. We noemen deze functie *pToken*, omdat hij een 'token' herkent:

$$
\begin{array}{lll}
pToken & :: & String \rightarrow Parser\ String \\
pToken\ k\ xs & \mid k \equiv take\ n\ xs & = [(k, drop\ n\ xs)] \\
 & \mid otherwise & = [\,] \\
 & \textbf{where}\ n = length\ k &
\end{array}
$$

Woordjes herkennen

De volgende functie kijkt of het eerste element van de invoer aan een bepaalde conditie voldoet:

$$
\begin{aligned}
&\mathit{pSatisfy} && :: (\mathit{Char} \rightarrow \mathit{Bool}) \rightarrow \mathit{Parser\ Char} \\
&\mathit{pSatisfy}\ p\ [\,] && = [\,] \\
&\mathit{pSatisfy}\ p\ (x : xs) && = \textbf{if}\ p\ x\ \textbf{then}\ [(x, xs)]\ \textbf{else}\ [\,]
\end{aligned}
$$

Woordjes herkennen

De volgende functie kijkt of het eerste element van de invoer aan een bepaalde conditie voldoet:

$$
\begin{aligned}
&\mathit{pSatisfy} && :: (\mathit{Char} \to \mathit{Bool}) \to \mathit{Parser}\ \mathit{Char} \\
&\mathit{pSatisfy}\ p\ [\,] && = [\,] \\
&\mathit{pSatisfy}\ p\ (x : xs) && = \textbf{if}\ p\ x\ \textbf{then}\ [(x, xs)]\ \textbf{else}\ [\,]
\end{aligned}
$$

Merk op, voor de minimalisten:

$$\mathit{pLetter}\ a = \mathit{pSatisfy}\ (a \equiv)$$

Twee dingen na elkaar herkennen

De functie $\langle * \rangle$, bouwt een ontleedfunctie die twee constructies naast elkaar herkent. De te herkennen constructies worden aangeleverd in de vorm van twee functie-parameters van het type *Parser* ....

Twee dingen na elkaar herkennen

De functie $\langle * \rangle$, bouwt een ontleedfunctie die twee constructies naast elkaar herkent. De te herkennen constructies worden aangeleverd in de vorm van twee functie-parameters van het type *Parser* .... In plaats van het cartesisch product te nemen als het resultaat van een dergelijke sequentiële compositie beginnen we heel slim met een ingewikkeld type dat de andere resultaten "opeet":

$$(\langle * \rangle) :: Parser\ (b \to a) \to Parser\ b \to Parser\ a$$

in plaats van:

$$(\langle * \rangle) :: Parser\ a \to Parser\ b \to Parser\ (a, b)$$

Twee dingen na elkaar herkennen

De functie $\langle * \rangle$, bouwt een ontleedfunctie die twee constructies naast elkaar herkent. De te herkennen constructies worden aangeleverd in de vorm van twee functie-parameters van het type *Parser* .... In plaats van het cartesisch product te nemen als het resultaat van een dergelijke sequentiële compositie beginnen we heel slim met een ingewikkeld type dat de andere resultaten "opeet":

$$(\langle * \rangle) :: Parser\ (b \to a) \to Parser\ b \to Parser\ a$$

in plaats van:

$$(\langle * \rangle) :: Parser\ a \to Parser\ b \to Parser\ (a, b)$$

We kennen deze operator uit de klasse *Applicative*.

De definitie van $<*>$; List of Successes

Een lijst-comprehensie komt nu goed van pas om alle mogelijkheden te combineren:

$$
\begin{array}{lll}
(<*>) & :: & \mathit{Parser}\ (b \to a) \to \mathit{Parser}\ b \to \mathit{Parser}\ a \\
(p_1 <*> p_2)\ xs = & [\ & (v_1\ v_2, xs_2) \\
 & \mid & (v_1\quad, xs_1) \leftarrow p_1\ xs \\
 & , & (v_2\quad, xs_2) \leftarrow p_2\ xs_1 \\
 &] &
\end{array}
$$

De definitie van $<*>$; List of Successes

Een lijst-comprehensie komt nu goed van pas om alle mogelijkheden te combineren:

$$
\begin{array}{ll}
(<*>) & :: \mathit{Parser}\ (b \rightarrow a) \rightarrow \mathit{Parser}\ b \rightarrow \mathit{Parser}\ a \\
(p_1 <*> p_2)\ xs = & [\ (v_1\ v_2, xs_2) \\
 & \mid (v_1 \quad , xs_1) \leftarrow p_1\ xs \\
 & ,\ (v_2 \quad , xs_2) \leftarrow p_2\ xs_1 \\
 &]
\end{array}
$$

De functie levert dus alle mogelijke waarden *v1 v2* met reststring *xs2* op, waarbij *v1* het resultaat van aanroep van parse-functie *p1* is, en waarvan de reststring *xs1* aan parse-functie *p2* wordt gevoed om waarde *v2* te bepalen.

Keuze; List of Successes

Behalve 'opeenvolging' moeten we ook iets hebben om 'keuze' te representeren. Daarvoor introduceren we de parser-combinator $<|>$:

$$
\begin{aligned}
&(<|>) && :: \mathit{Parser}\ a \rightarrow \mathit{Parser}\ a \rightarrow \mathit{Parser}\ a \\
&(p_1 <|> p_2)\ xs && = p_1\ xs +\!\!+\ p_2\ xs
\end{aligned}
$$

Deze functie probeert eenvoudigweg beide parsers op de input uit, en concateneert alle gevonden ontledingen.

Extra combinators

Naast de parser-combinators $<*>$ en $<|>$ zijn er functies om parsers te transformeren in andere parsers. De eerste parser-transformeerder is *pJust*. Gegeven een parser p levert deze functie een parser die hetzelfde doet als p, maar bovendien garandeert dat de rest-string leeg is:

```
pFinal        :: Parser a→Parser a
pFinal p xs = [ (v, "")
              | (v, ys) ← p xs
              , ys ≡ ""
              ]
```

Extra combinators

Naast de parser-combinators $<*>$ en $<|>$ zijn er functies om parsers te transformeren in andere parsers. De eerste parser-transformeerder is *pJust*. Gegeven een parser p levert deze functie een parser die hetzelfde doet als p, maar bovendien garandeert dat de rest-string leeg is:

```
pFinal        :: Parser a→Parser a
pFinal p xs = [ (v, "")
              | (v, ys) ← p xs
              , ys ≡ ""
              ]
```

```
pSucceed v input = [(v, input)]
p `option` v     = p <|> pSucceed v
pFail      input = []
```

$<\$> :: (a \to b) \to Parser\ a \to Parser\ b$

Een andere handige parser-transformeerfunctie is $<\$>$. Deze functie maakt van een parser een andere parser, door op het resultaat van de parser een functie toe te passen.

$$
\begin{aligned}
&(<\$>) && :: (a \to b) \to Parser\ a \to Parser\ b \\
&(f <\$> p)\ xs && = [\ (f\ v, ys) \\
& && \quad \mid (\ \ v, ys) \leftarrow p\ xs \\
& && \quad]
\end{aligned}
$$

$<\$> :: (a \to b) \to Parser\ a \to Parser\ b$

Een andere handige parser-transformeerfunctie is $<\$>$. Deze functie maakt van een parser een andere parser, door op het resultaat van de parser een functie toe te passen.

$$
\begin{aligned}
&(<\$>) && :: (a \to b) \to Parser\ a \to Parser\ b \\
&(f <\$> p)\ xs && = [\ (f\ v, ys) \\
& && \quad \mid (\ \ v, ys) \leftarrow p\ xs \\
& && \quad]
\end{aligned}
$$

Als we naar het type kijken dan zien we dan we dat *Parser* dus ook een instantie is van de klasse *Functor*: $<\$>$ is gewoon *fmap*.

$<\$> :: (a \to b) \to Parser\ a \to Parser\ b$

Een andere handige parser-transformeerfunctie is $<\$>$. Deze functie maakt van een parser een andere parser, door op het resultaat van de parser een functie toe te passen.

$$
\begin{aligned}
&(<\$>) && :: (a \to b) \to Parser\ a \to Parser\ b \\
&(f <\$> p)\ xs && = [\ (f\ v, ys) \\
& && \quad \mid (\ \ v, ys) \leftarrow p\ xs \\
& && \quad]
\end{aligned}
$$

Als we naar het type kijken dan zien we dan we dat *Parser* dus ook een instantie is van de klasse *Functor*: $<\$>$ is gewoon *fmap*. We kiezen handige prioriteiten:

```
infixl 5 <*>
infixr 3 <|>
infix  7 <$>
```

Resultaten berekenen

Met de tot nu toe geschreven parsergenereer- en -transformeerfuncties kunnen we een ontleedfunctie schrijven die correct geneste haakjesparen ontleedt.

Resultaten berekenen

Met de tot nu toe geschreven parsergenereer- en -transformeerfuncties kunnen we een ontleedfunctie schrijven die correct geneste haakjesparen ontleedt. Een voorbeeld-sessie met de functie *haakjes*:

```
? pFinal haakjes "()(())()"
[(2,[])]
(254 reductions, 615 cells)
? pFinal haakjes "())"
[]
(63 reductions, 131 cells)
```

Inderdaad herkent *pFinal haakjes* alleen correct geformeerde haakjesparen, en berekent daarbij de nesting-diepte.

Gebruik van <$>

Met behulp van <$> berekenen we als functiereultaat de nestingsdiepte (dit staat model voor het berekenen van een semantische functie naar keuze). Door de gekozen prioriteiten is het net alsof de functie die voor de <$> staat los gelaten wordt op de resultaten van de parsers die achter de <$> staan.

$$
\begin{aligned}
haakjes = \quad & (\quad (\lambda\_\ h1\ \_\ h2 \rightarrow (1 + h1)\ \text{`}max\text{`}\ h2) \\
& <\$>\ pLetter\ \text{'('} \\
& <*>\ haakjes \\
& <*>\ pLetter\ \text{')'} \\
& <*>\ haakjes) \\
& <|>\ pSucceed\ 0 \\
pSucceed\ v\ inp = & [(v, inp)]
\end{aligned}
$$

<$> combineert rijtjes resultaten

De functie voor de <$> levert samen met de eerste parser na de <$> weer een parser op, maar wel een die als resultaat een functie oplevert met één parameter minder. Omdat we niet in de resultaten van de parsers voor de haakjes zelf zijn geintereseerd hebben hiervoor een _ op de parameterplaats gebruikt.

<$> combineert rijtjes resultaten

De functie voor de <$> levert samen met de eerste parser na de <$> weer een parser op, maar wel een die als resultaat een functie oplevert met één parameter minder. Omdat we niet in de resultaten van de parsers voor de haakjes zelf zijn geintereseerd hebben hiervoor een _ op de parameterplaats gebruikt. Merk op dat zowel <*> als <$> links associatief zijn.

Handige combinaties

In de *haakjes* parser kwamen we de functie $\lambda\_\, h1\ \_\, h2 \rightarrow (1 + h1)\ `max`\ h2$ tegen. Kennelijk zijn we niet altijd in alle onderdelen die we herkend hebben geinteresseerd. Omdat dit veel voor komt definiëren we speciale gevallen van de combinatoren $<\$>$ en $<*>$, die het resultaat van hun rechter argument weggooien:

Handige combinaties

In de *haakjes* parser kwamen we de functie $\lambda\_\ h1\ \_\ h2 \rightarrow (1 + h1)\ `max`\ h2$ tegen. Kennelijk zijn we niet altijd in alle onderdelen die we herkend hebben geinteresseerd. Omdat dit veel voor komt definiëren we speciale gevallen van de combinatoren $<\$>$ en $<*>$, die het resultaat van hun rechter argument weggooien:

```
infixl    5 <*
infixl    7 <$
pSucceed v inp = [(v, inp)]
f <$  p = const <$> pSucceed f <*> p
p <*  q = const <$> p          <*> q
```

De opgeschoonde *haakjes*

De functie *haakjes* wordt nu:

$$
\begin{array}{lll}
haakjes = & (max \circ (1+) & \\
 & <\$\ \ pLetter\ \texttt{'('} & \text{-- deze niet} \\
 & <\!*\!>\ haakjes & \text{-- deze wel} \\
 & <\!*\ \ pLetter\ \texttt{')'} & \text{-- deze niet} \\
 & <\!*\!>\ haakjes) & \text{-- deze wel} \\
 & `option`\ 0 &
\end{array}
$$

We vatten het patroon samen

$$
\begin{aligned}
&\mathit{pParens}\ p = \mathit{pSymbol}\ \text{'('}\ \ *\!\!> p <\!\!*\ \ \mathit{pSymbol}\ \text{')'}\\
&\mathit{haakjes} \quad = (\mathit{max} <\!\$\!> ((1+) <\!\$\!> \mathit{pParens}\ \mathit{haakjes})\\
&\qquad\qquad\qquad\qquad <\!\!*\!\!> \mathit{haakjes})\\
&\qquad\qquad\quad \text{`}\mathit{option}\text{`}\ 0\\
&p\text{`}\mathit{option}\ v = p <\!|\!> \mathit{pSucceed}\ v
\end{aligned}
$$

De essentie van $\langle * \rangle$

We hebben gezien hoer we *Maybe* een instantie van *Applicative* konden maken, teneinde falende berekeningen te modelleren: $e_1\ e_2$.

- some het functiegdeeltet e_1 als het argument gedeelte e_2 kunnen falen
- we modelleren dit met een *Maybe*
- dus we willen een *Maybe* $(b \to a)$ toepassen op een *Maybe b* en een *Maybe a* opleveren

```
func <*> arg = case func of
                 Just b2a -> case arg of
                               Just b    -> Just (b2a b)
                               Nothing   -> Nothing
                 Nothing  -> Nothing
```

De essentie van of $<*>$ (Cont)

Dit patroon wordt gevangen door::

```
class Applicative p where
  (<*>)   :: p (b→a)→p b→p a
  pure    :: a          →p a
  (<$>)   :: (b→a)   →p b→p a
  f <$> p = pure f <*> p
  ...
instance Applicative Maybe where
  Just f <*> Just v = Just (f v)
  _      <*> _      = Nothing
  pure a            = Just a
```

De essentie van $<*>$ (Cont)

Als we nu schrijven:

$$f <*> a\_1 <*> a\_2 <*> a\_3$$

dan hebben we de originele impliciete functie toepassing in $f\ a\_1\ a\_2\ a\_3$ "ge-overload".

De essentie van $<*>$ (Cont)

Als we nu schrijven:

$$f <*> a\_1 <*> a\_2 <*> a\_3$$

dan hebben we de originele impliciete functie toepassing in $f\ a\_1\ a\_2\ a\_3$ "ge-overload". **Conclusie:**
In plaats van een waarde van type $b \rightarrow a$ toe te passen op een waarde van type type b teninde een waarde van type a te krijgen, laat de operator $<*>$ een p-waarde met label/index type $b \rightarrow a$ los op p-value met label/index type b teneindeeen p-waarde met label/index type a te krijgen.

All *Monad*s zijn *Applicative*

Elke type constructor die in *Monad* zit kan op triviale wijze ook een instantie van *Applicative* gemaakt worden:

```
instance Monad m ⇒ Applicative m where
  f <*> v = do fv ← f
               vv ← v   -- bevat geen referentie naar fv
               return (fv vv)
  pure    = return
```

Combinatie van *Applicative* instanties

Merk op dat ons huidige *Parser* type een combinatie is van twee *Applicative*'s die we eerder hebben gezien:

- we houden een toestand bij (de nog niet gebruikte invoer)
- we modelleren niet-determinisme

Combinatie van *Applicative* instanties

Merk op dat ons huidige *Parser* type een combinatie is van twee *Applicative*'s die we eerder hebben gezien:

- we houden een toestand bij (de nog niet gebruikte invoer)
- we modelleren niet-determinisme

Dit is een veel voorkomend patroon. Denk bijv. aan het doorzoeken van een graaf:

- de toestand is de lijst met al bezichte knopen
- de wordt bepaald door de uitgaande takken

Case Study: Pocketcalculators

1. een groot aantal voorbeelden
2. elk voorbeeld is een opvolger van de vorige

Een driver programma (zie website vak)

```
module PocketCalculators where
import ParseLib
import Data.Char

run :: (Show t) ⇒ Parser Char t→String→IO ()
run p ex =
  do putStrLn ("Expected: " ++ ex ++ " or (q) to quit")
     inp ← getLine
     case inp of
       "q" → return ()
        _ → do putStrLn (
                  case p (filter (≢ ' ') inp) of
                  ((v, ""): _) → "Result is: " ++ show v
                  _            → "Incorrect input")
               run p ex
```

Een enkel cijfer

$$
\begin{aligned}
c1 = \mathit{run}\ \mathit{pDigit}\ \texttt{"Digit"} \\
\quad \textbf{where}\ \mathit{pDigit} = \mathit{pSatisfy}\ (\lambda x \rightarrow \mathit{ord}\ \texttt{'0'} \leqslant \mathit{ord}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{ord}\ x \quad \leqslant \mathit{ord}\ \texttt{'9'})
\end{aligned}
$$

Een enkel cijfer

$$
\begin{aligned}
c1 = \mathit{run}\ \mathit{pDigit}\ \texttt{"Digit"} \\
\quad \textbf{where}\ \mathit{pDigit} = \mathit{pSatisfy}\ (\lambda x \rightarrow \mathit{ord}\ \texttt{'0'} \leqslant \mathit{ord}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{ord}\ x \quad \leqslant \mathit{ord}\ \texttt{'9'})
\end{aligned}
$$

```
*PocketCalculators> c1
Expected: Digit or (q) to quit
3
Result is: '3'
Expected: Digit or (q) to quit
a
Incorrect input
Expected: Digit or (q) to quit
q
*PocketCalculators>
```

Nog een keer: een cijfer

We herkennen nog steeds een cijfer, maar het resultaat is nu een getal:

$$
\begin{aligned}
c2 = {} & run\ pDigAsInt\ \texttt{"Digit"} \\
& \quad \textbf{where}\ pDigitAsInt = f\ <\$>\ pDigit \\
& \qquad\qquad f\ c = ord\ c - ord\ \texttt{'0'}
\end{aligned}
$$

Nog een keer: een cijfer

We herkennen nog steeds een cijfer, maar het resultaat is nu een getal:

$$
\begin{aligned}
c2 = {} & run\ pDigAsInt\ \texttt{"Digit"} \\
& \quad \textbf{where}\ pDigitAsInt = f\ <\$>\ pDigit \\
& \qquad\qquad f\ c = ord\ c - ord\ \texttt{'0'}
\end{aligned}
$$

Of zelfs:

$$pDigitAsInt = ((-ord\ \texttt{'0'}) \circ ord)\ <\$>\ pDigit$$

Een volledig natuurlijk getal

$$c3 = run\ pNatural\ \texttt{"Natural Number"}$$

$$pNatural = foldl\ (\lambda n\ c \rightarrow n * 10 + c)\ 0$$
$$<\$>\ pMany1\ pDigAsInt$$

Een volledig natuurlijk getal

```
c3 = run pNatural "Natural Number"
pNatural = foldl (λn c→n * 10 + c) 0
                <$> pMany1 pDigAsInt
```

```
pMany   :: Parser a→Parser [a]
pMany p = (:) <$> p <*> pMany p `option` []
  --
pMany1  :: Parser s a→Parser s [a]
pMany1 p = (:) <$> p <*> pMany p
```

Ook negatieve getallen

$$
\begin{aligned}
c4 &= run\ pInteger\ \texttt{"Integer (possibly negative)"} \\
&\quad \textbf{where}\ pInteger = (negate <\$\ \ (pToken\ \texttt{"-"})\ `option`\ id) \\
&\qquad\qquad\qquad\qquad <*>\ pNatural \\
negate\ x &= -x
\end{aligned}
$$

De som van twee getallen

$$
\begin{aligned}
c5 = \mathit{run}\ \mathit{pPlus}\ &\texttt{"e.g. 2+-5"} \\
\textbf{where}\ \mathit{pPlus} = (+)\ &<\$>\ \mathit{pInteger} \\
&<*\ \ \mathit{pToken}\ \texttt{"+"} \\
&<*>\ \mathit{pInteger}
\end{aligned}
$$

Een rij optellingen

```
c6 = run pPlus' "e.g. 2+-5"
  --
pPlus' = applyall <$> pInteger
                  <*> pMany ((+) <$  pToken "+" <*> pInteger
  --
applyall x (f : fs)  = applyall (f x) fs
applyall x []        = x
```

Aftrekken i.p.v. optellen

Pas op: aftrekken is niet commutatief, dus operanden omwisselen.

```
c7 = run pMinus' "e.g. 5-3-2"
  where pMinus' = applyall <$> pInteger
                <*> pMany (flip (-) <$  pToken "-"
                                    <*> pInteger
                          )
flip f x y = f y x
```

Optellen en aftrekken

```
c8 = run pPlusMinus " e.g. 2+3-5"
pPlusMinus = applyall
                <$>  pInteger
                <*>  pMany (   (   flip (-) <$  pToken "-"
                             <|> ↑↑
                                 flip (+) <$  pToken "+"
                               )    <*> pInteger
                           )
```

Geef dit patroon een naam: *pChainL*

We definiëren een nieuwe combinator *pChainl* met twee parameters:

1. de scheider, die als waarde een binaire functie oplevert

$$
\begin{aligned}
&\mathit{pChainL}\ s\ p = \mathit{applyall} <\$> p <*> \mathit{pMany}\ (\mathit{flip} <\$> s <*> p) \\
&c9 = \mathit{run}\ \mathit{pPlusMinus}'\ \texttt{" e.g. 2+3-5"} \\
&\qquad \textbf{where}\ \mathit{pPlusMinus}' = \mathit{pChainL}\ (\ (-) <\$\ \ \mathit{pToken}\ \texttt{"-"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad <|> \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (+) <\$\ \ \mathit{pToken}\ \texttt{"+"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ )\ \mathit{pInteger}
\end{aligned}
$$

Geef dit patroon een naam: *pChainL*

We definiëren een nieuwe combinator *pChainl* met twee parameters:

1. de scheider, die als waarde een binaire functie oplevert
2. de operand

$$
\begin{aligned}
&\mathit{pChainL}\ s\ p = \mathit{applyall} <\$> p <*> \mathit{pMany}\ (\mathit{flip} <\$> s <*> p) \\
&c9 = \mathit{run}\ \mathit{pPlusMinus}'\ \texttt{" e.g. 2+3-5"} \\
&\qquad \textbf{where}\ \mathit{pPlusMinus}' = \mathit{pChainL}\ (\ (-) <\$\ \ \mathit{pToken}\ \texttt{"-"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad <|> \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (+) <\$\ \ \mathit{pToken}\ \texttt{"+"} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ )\ \mathit{pInteger}
\end{aligned}
$$

Abstraheer over de operatoren

```
c10 = run pPlusMinus'' " e.g. 2+3-5"
  --
pPlusMinus'' = pChainL addops pInteger
  --
addops = pLookup [((+),"+"),((-),"-")]
  --
pLookup ops =
  foldr <|>
        pFail
        [λ(op,sym)→op <$ pToken sym | (op,sym) ← ops]
```

We willen ook kunnen vermenigvuldigen

Vermenigvuldigen heeft een hogere prioriteit dan optellen en aftrekken:

$$
\begin{aligned}
&c11 = run\ pPlusMinusTimes\ \texttt{"e.g. 2*3-5"}\\
&pPlusMinusTimes = pChainL\ addops\ \ pTimes\\
&pTimes \qquad\qquad\quad = pChainL\ mulops\ pInteger\\
&mulops \qquad\qquad\quad = pLookup\ [((*), \texttt{"*"})]\\
&addops \qquad\qquad\quad = pLookup\ [((+), \texttt{"+"}), ((-), \texttt{"-"})]
\end{aligned}
$$

Nog meer prioriteiten?

$$
\begin{aligned}
&c12 = run\ pPlusMinusTimes\ \texttt{"e.g. 2*3-5"}\\
&pPlusMinusTimes = pChainL\ addops\ (pChainL\ mulops\ pInteger)\\
&pPlusMinusTimes = addops\ `pChainL`\ mulops\ `pChainL`\ pInteger\\
&mulops \qquad\qquad\quad = pLookup\ [((*), \texttt{"*"})]\\
&addops \qquad\qquad\quad = pLookup\ [((+), \texttt{"+"}), ((-), \texttt{"-"})]
\end{aligned}
$$

Nog meer prioriteiten?

$$
\begin{aligned}
&c12 = run\ pPlusMinusTimes\ \texttt{"e.g. 2*3-5"}\\
&pPlusMinusTimes = pChainL\ addops\ (pChainL\ mulops\ pInteger)\\
&pPlusMinusTimes = addops\ `pChainL`\ mulops\ `pChainL`\ pInteger\\
&mulops \qquad\qquad = pLookup\ [((*), \texttt{"*"})]\\
&addops \qquad\qquad = pLookup\ [((+), \texttt{"+"}), ((-), \texttt{"-"})]
\end{aligned}
$$

$$
\begin{aligned}
&c12 = run\ pPlusMinusTimes\ \texttt{"e.g. 2*3-5"}\\
&pPlusMinusTimes = foldr\ pChainL\ pInteger\ [addops, mulops]\\
&mulops \qquad\qquad = pLookup\ [((*), \texttt{"*"})]\\
&addops \qquad\qquad = pLookup\ [((+), \texttt{"+"}), ((-), \texttt{"-"})]
\end{aligned}
$$

Subexpressie met haakjes hebben de hoogste prioriteit.

```
c14 = run pExpr "e.g. 2*(3-5)"
pExpr   = foldr pChainL pFactor [addops, mulops]
pFactor = pInteger <|> pParens pExpr
  --
pParens p = pPack '(' p ')'
  --
pPack       :: Parser Char→Parser b→Parser Char→Parser b
pPack p r q = pSymbol p  *> r <*  pSymbol q
```

Conditionele Expressies

```
c15 = run pExpr "e.g. if true then 5 else 7 "
pExpr        = foldr pChainL pFactor [addops, mulops]
               <|> pIfThenElse
  --
pIfThenElse = choose <$  pToken "if"
                     <*> pBoolExpr
                     <*  pToken "then"
                     <*> pExpr
                     <*  pToken "else"
                     <*> pExpr
pBoolExpr   =      True <$  pToken "true"
               <|> False <$  pToken "false"
choose cond t e = if cond then t else e
```

Relationele Operatoren

```
c16 = run pExpr "e.g. if 3 <5 && 6 <= 7  then 5 else 7
pBoolExpr = foldr pChainL pRelExpr [andops, orops]
pRelExpr  =          True <$  pToken "true"
           <|>       False <$  pToken "false"
           <|>       pExpr <**> (flip <$> pRelOp <*> pExpr)
p <**> q  = (λpr qr→qr pr) <$> p <*> q
  --
andops  = pLookup [((∧), "&&")]
orops   = pLookup [((∨), "||")]
pRelOp  = pLookup [((⩽), "<="), ((⩾), ">="),
                   ((≡), "=="), ((≢), "/="),
                   ((<), "<"), ((>) , ">")]
```

Variabelen: type $Env = [(Char, Int)]$

```
run' :: Parser Char (Env→Int)→String→Env→IO ()
run' p ex env
↑↑ = do putStrLn ("Expected: " ++ ex ++
                 " or (q) to quit or (d) to define")
        inp ← getLine
        case inp of
         ↑↑ "q"→return ()
            "d"→do {putStrLn "Variable? ";inp ← getLine
                   ; let var = head inp
                   ; putStrLn "Value?";      inp ← getLine
                   ; let val = read inp
                   ; run' p c ((var,val) : env) }
            _→do case p (filter (≢ ' ') inp) of
                 ↑↑ ((v,""): _)→putStrLn (show (v env))
                  _            →putStrLn "Input?"
                 run' p ex env
```

Overal een extra omgeving *env*

```
applyall' x fs env = applyall (x env) (map ($env) fs)
  --
pChainL' s p = applyall'
                 <$> p
                 <*> pMany ((λs p env→flip s (p env))
                            <$> s <*> p
                   )
  --
choose' cond t e env = if cond env then t env else e env
```

Uiteindelijke Calculator

```
c17 = run' pExpr "e.g. if ?x < ?y then ?x else ?y " []
pExpr      = foldr pChainL' pFactor [addops, mulops] <|> pIfThen
pFactor    = pInteger' <|> pParens pExpr <|> pVar
pVar       = (λident env→  case lookup ident env of
                              Nothing→error "illegal id"
                              Just r→r)
                <$  pSymbol '?' <*> pChar
pChar      = pSatisfy (λx→'a' ⩽ x ∧ x ⩽ 'z')
pBoolExpr = foldr pChainL' pRelExpr [andops, orops]
pRelExpr  =       const True <$  pToken "true"
                <|> const False <$  pToken "false"
                <|> pExpr
                      <**> ((λop ex env→flip op (ex env))
                              <$> pRelOp <*> pExpr)
p <**> q   = (λpr qr env→(qr env) (pr env)) <$> p <*> q
pInteger'  = const <$> pInteger
```

Meer parser-combinators

De bibliotheek `Parselib.hs` bevat een aantal handige extra combinatoren zoals:

$$
\begin{aligned}
&\mathit{pSepBy} && :: \mathit{Parser}\ a \rightarrow \mathit{Parser}\ b \rightarrow \mathit{Parser}\ [a] \\
&p\ `\mathit{pSepBy}`\ s && = (:) <\$> p <*> \mathit{pMany}\ (s\ \ *> p)\ `\mathit{option}`\ [\,] \\
&\mathit{pCommaList}\ p && = p\ `\mathit{pSepBy}`\ (\mathit{pSymbol}\ ',')
\end{aligned}
$$