Monadische Parser

Syntax-Analyse mit Haskell

Mathias Mierswa (winf2920)
Thomas Stuht (inf6937)

8. Januar 2009

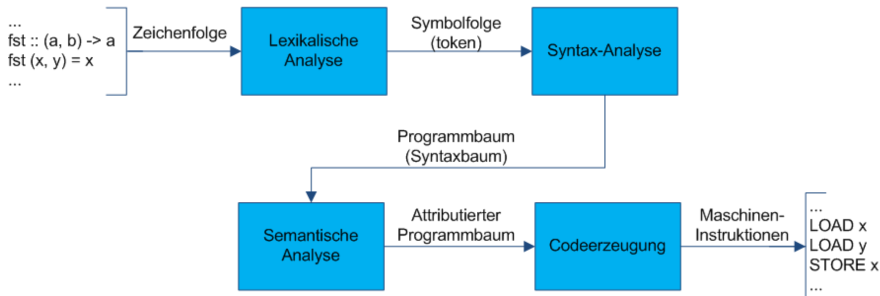
Gliederung

1. Motivation
2. Grundlagen des Compilerbaus
3. Entwicklung eines Parsers mit Haskell
4. Verwendung der Parser-Kombinator Bibliothek Parsec
5. Demobeispiel - Parsec

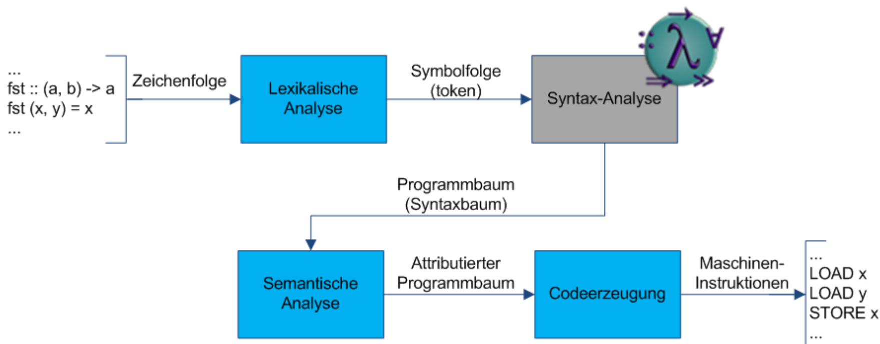
Gliederung

1. Motivation
   - Komponenten eines Compilers
   - Was ist die Syntax-Analyse?

Komponenten eines Compilers



Komponenten eines Compilers



Komponente: Syntax-Analyse

- **Eingabe**: Liste von Zeichen oder Symbolfolge (tokens)
  - eventuell durch einen Lexer übergeben
- **Ausgabe**: Syntaxbaum (parse tree)
  - repräsentiert die syntaktische Struktur der Eingabe
- **Spezifikation**: Kontextfreie Grammatik

Intuitive Parserdefinition

$\textbf{type}\ \text{Parser} = \text{String} \longrightarrow \text{Tree}$

Reichen die regulären Ausdrücke nicht aus?

Nein!

- reguläre Ausdrücke können schnell unhandlich werden
- Klammerschachtelungen können durch reguläre Ausdrücke nicht abgebildet werden (keine Rekursion!)
- Parser notwendig, um die meisten Programmiersprachen zu verarbeiten
- reguläre Ausdrücke können keine aussagekräftigen Fehlermeldungen liefern

Anwendungsgebiete

Syntax-Analyse wird zum Parsen verschiedener Objekte benötigt:

- Quelltexte von Programmiersprachen (z.B. Haskell, Java ...)
- HTML-Dokumente in Web-Browsern
- XML (z.B. HXmlToolbox: www.fh-wedel.de/~si/HXmlToolbox)
- RSS-Feeds
- CSV-Dateien
- TEX-Dateien
- Domain Specific Languages (DSLs)
- ...

Gliederung

2 Grundlagen des Compilerbaus
- Grammatiken

Kontextfreie Grammatiken

- hohe Praxisrelevanz: Festlegung der Syntax vieler Programmiersprachen
- 4-Tupel: G = (T,N,P,S) mit $T \cap N = \emptyset$ und $S \in N$
  - T: nicht-leere, endliche Menge der Terminalsymbole
  - N: nicht-leere, endliche Menge der Nichtterminalsymbole
  - P: nicht-leere, endliche Produktionsmenge (Regeln)
    - $P \subseteq N \times (T \cup N)^*$
  - S: Startsymbol
- häufig in (Erweiterter) Backus-Naur-Form (*[E]BNF*)angegeben

Analyseverfahren

- für Top-Down-Parser relevant
  - LL(1)
    - von links nach rechts gelesen
    - Linksableitung
    - Lookahead 1 = ein Zeichen vorausschauen
  - LL(k)
    - Lookahead k = k Zeichen vorausschauen
  - kein LL(1), wenn ...
    - rechte Produktionsseiten nicht eindeutig →Faktorisierung
    - Mehrdeutigkeiten (z.B. zwei Ableitungsbäume für *ein* Wort)
    - Linksrekursion→Transformation in Rechtsrekursion

Gliederung

3 Entwicklung eines Parsers mit Haskell
- Monadischer Ansatz
- Parser-Implementierungen
- Effizienz

Erläuterungen

Achtung, Monaden!

der im Folgenden entwickelte Parser ist ein Beispiel für die Verwendung von Monaden in Haskell

- Schrittweise Entwicklung eines recursive descent parsers
  - Top-Down-Parser
  - jedes Nichtterminalsymbol von einem "Teil"-Parser verarbeitet
  - Regel-Auswahl erfolgt anhand des nächsten Zeichens in der Eingabe
- Modularisierung führt zu "Parser-Baukasten"
  - kleine Parser erfüllen Spezialaufgaben
  - kleine Parser werden mittels Kombinatoren zu größeren Einheiten verbunden

Typdefinition I

Intuitive Definition des Parsers

Listing: intuitive Definition

```
newtype Parser = MkParser (String -> Tree)
```

- newtype zum Verbergen der Implementierung der Funktionen vom Typ (String -> Tree)
- ✖ Definition nicht ausreichend, da ein Parser nur Spezialaufgabe erfüllt
  - nicht verarbeiteter Reststring wird nicht zurückgeliefert
  - keine Kombination mehrerer Teilparser möglich

Typdefinition II

Verbesserung 1

Listing: erste Verbesserung

```
newtype Parser = MkParser (String -> (Tree,String))
```

- ✔ Rückgabe eines Paars, bestehend aus erzeugtem Teilbaum und nicht verarbeitetem Reststring
- ✖ Definition nicht ausreichend
  - Parsing-Vorgang kann fehlschlagen
  - mehrdeutige Grammatik können mehrere Ergebnisse liefern

Typdefinition III

Verbesserung 2

Listing: zweite Verbesserung

```
newtype Parser = MkParser (String -> [(Tree,String)])
```

- ✔ Rückgabe einer leeren Liste, wenn Parser fehlschlägt
- ✔ Liste kann mehrere mögliche Ergebnisse beinhalten
- ✖ Definition kann noch verallgemeinert werden

Typdefinition IV

Verbesserung 3

Listing: dritte Verbesserung

```
newtype Parser a = MkParser (String -> [(a,String)])
```

- ✔ Abstraktion vom konkreten Ergebnistyp
  - Teil-Parser können unterschiedliche Typen zurückliefern (Char, Integer,...)
  - Abstraktion von String denkbar, aber für diesen Zweck nicht notwendig

Parser Monade I

Funktion zum Anwenden eines Parsers auf einen Eingabestring:

Funktion parse

```
parse :: Parser a -> String -> [(a,String)]
parse (MkParser p) = p
```

Rückgabe eines *einzelnen* Resultats:

Funktion parseOne

```
parseOne :: Parser a -> String -> a
parseOne p = fst . head . parse p
```

Parser Monade II

Definition des Typs Parser als Monadeninstanz

Listing: Monadeninstanz

```
instance Monad Parser where
  return a = MkParser (\s -> [(a,s)])
  p >>= f = MkParser (\s ->
                concat [parse (f a) s' | (a, s') <- parse p s])
```

- Lamda-Ausdruck spiegelt Funktion vom Typ String -> [(a,String)] wider

Parser Monade III

Verwendung der eingebauten Typklasse MonadPlus

Listing: MonadPlus (vordefiniert)

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

- mzero : Fehlerfall wird als leeres Ergebnis implementiert
- mplus : Kombination zweier Resultate

Parser Monade IV

Verwendung der eingebauten Typklasse MonadPlus

Listing: MonadPlus-Instanz

```
instance MonadPlus Parser where
  mzero = MkParser (\s -> [])
  mplus p q = MkParser (\s -> parse p s ++ parse q s)
```

- mzero : Fehlerfall wird alsleeres Ergebnis implementiert
- mplus : Kombination zweier Resultate

Parser Monade V

Beschränkung auf das erste Ergebnis des Parsevorgangs

Funktion limit

```
limit :: Parser a -> Parser a
limit p = MkParser (\s -> case parse p s of
          [] -> []
          (x:xs) -> [x])
```

Einfache Parser: item

Lesen eines einzelnen Zeichens oder Rückgabe eines Fehlers, wenn leere Eingabe

Funktion item

```
item :: Parser Char
item = MkParser (\s -> case s of
            [] -> []
            (x:xs) -> [(x,xs)])
```

Einfache Parser: sat

Verarbeitung eines Zeichens, falls Prädikat erfüllt wird

Funktion sat

```
sat :: (Char -> Bool) -> Parser Char
sat p = do {x <- item; if p x then return x else mzero}
```

Einfache Parser: char und string

Funktion char

```
char :: Char -> Parser Char
char c = sat (== c)
```

Funktion string

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do {char x; string xs; return (x:xs)}
```

Einfache Parser: digit, letter, ...

Funktion digit

```
digit :: Parser Char
digit = sat isDigit
```

Funktion letter

```
letter :: Parser Char
letter = sat isAlpha
```

Einfache Parser: ... alphanum, lower, upper

Funktion alphanum

```
alphanum :: Parser Char
alphanum = sat isAlphaNum
```

Funktion lower

```
lower :: Parser Char
lower = sat isLower
```

Funktion upper

```
upper :: Parser Char
upper = sat isUpper
```

Kombinatoren I

Funktion exOr

```
exOr :: Parser a -> Parser a -> Parser a
exOr p q = MkParser (\s -> case parse p s of
            [] -> parse q s
            ps -> ps)
```

Funktion zeroOrMore

```
zeroOrMore :: Parser a -> Parser [a]
zeroOrMore p = exOr (oneOrMore p) (return [])
```

Funktion oneOrMore

```
oneOrMore :: Parser a -> Parser [a]
oneOrMore p = do {x <- p; xs <- zeroOrMore p; return (x:xs)}
```

Kombinatoren II

Funktion identifier

```
identifier :: Parser String
identifier = do {x <- lower; xs <- zeroOrMore alphanum; return (x:xs)}
```

Funktion nat

```
nat :: Parser Int
nat = do {xs <- oneOrMore digit; return (read xs)}
```

Funktion integer

```
integer :: Parser Int
integer = exOr (do {char '-'; n <- natural; return (-n)}) natural
```

Lexer-Funktionalität

Funktion space

```
space :: Parser ()
space = do {zeroOrMore (sat isSpace); return ()}
```

Funktion token

```
token :: Parser a -> Parser a
token p = do {space; x <- p; space; return x}
```

Funktionen symbol und natural

```
symbol :: String -> Parser String
symbol s = token (string s)
natural :: Parser Int
natural = token nat
```

Links-Rekursion I

BNF

expr ::= expr addop factor | factor
factor ::= const | '(' expr ')'
addop ::= '+' | '-'
const ::= integer

Links-Rekursion II

Listing: 1. Versuch

```
data Expr = Con Int | Bin Op Expr Expr deriving Show
data Op = Plus | Minus deriving Show
expr :: Parser Expr
expr = exOr term factor
term = do {t <- expr; op <- addop; u <- factor; return (Bin op t u)}
addop = exOr (do {symbol "+"; return Plus})
             (do {symbol "-"; return Minus})
factor = exOr (token const')
              (do {symbol "("; e <- expr; symbol ")"; return e})
const' = do {n <- integer; return (Con n)}
```

- ✖ Endlosschleife, da expr term aufruft und term wieder expr ...

Links-Rekursion III

- Transformation in rechtsrekursive Regeln

verbesserte BNF

expr ::= factor rest
rest ::= addop factor rest | ϵ
factor ::= const | '(' expr ')'
addop ::= '+' | '-'
const ::= integer

Links-Rekursion IV

Listing: 2. Versuch

```
data Expr = Con Int | Bin Op Expr Expr deriving Show
data Op = Plus | Minus deriving Show
expr' :: Parser Expr
expr' = do {t <- factor; rest t}
rest t = exOr (do {op <- addop; u <- factor; rest (Bin op t u)})
              (return t)
.... {- keine Änderungen -}
```

- ✔ Parser rest verarbeitet ggf. zweiten Teil von expr oder liefert einzelnen Wert zurück

Begrenzung der Resultate

Funktion limit

```
limit :: Parser a -> Parser a
limit p = MkParser (\s -> case parse p s of
          [] -> []
          (x:xs) -> [x])
```

- Speicherplatzvorteil gegenüber take 1
- take 1 (r:rs) = r:take 0 rs
- take 0 rs = [] -> wird aber erst bei Bedarf ausgewertet
- Folge: die übrigen Resultate müssen sich gemerkt werden (Pointer)

Lazy-Evaluation I

bisherige Funktionen

zeroOrMore p = exOr (oneOrMore p) (return [])
oneOrMore p = do {x <- p; xs <- zeroOrMore p; return (x:xs)}
as = zeroOrMore (char 'a')

Aufruf für "aab"

parse as $\perp = \perp$
parse as ('a' :$\perp$) = $\perp$
parse as ('a' : 'a' :$\perp$) = $\perp$
parse as ('a' : 'a' : 'b' :$\perp$) = [("aa", 'b' :$\perp$)]

Lazy-Evaluation II

Funktion force

```
force :: Parser a -> Parser a
force p = MkParser
          (\s -> (fst (head (rs s)), snd (head (rs s))) : tail (rs s))
             where
             rs s = parse p s
```

Lazy-Evaluation III

neue Funktionen (mit force)

zeroOrMore' p = force (exOr (oneOrMore' p) (return []))
oneOrMore' p = do { x <- p; xs <- zeroOrMore' p; return (x:xs)}
as' = zeroOrMore' (char 'a')

Aufruf für "aab"

parse as' $\perp$ = ($\perp$,$\perp$) :$\perp$
parse as' ('a' :$\perp$) = ('a' :$\perp$,$\perp$) :$\perp$
parse as' ('a' : 'a' :$\perp$) = ('a' : 'a' :$\perp$,$\perp$) :$\perp$
parse as' ('a' : 'a' : 'b' :$\perp$) = [("aa", 'b' :$\perp$)]

Gliederung

4 Verwendung der Parser-Kombinator Bibliothek Parsec
- Grundlegendes
- Implementierung
- Code - Beispiele

Was ist Parsec?

- Parser-Kombinator Bibliothek
- top-down-Parser
- Verwendung von Monaden
- unterstützte Grammatiken
  - kontext-sensitiv
  - insbesondere LL(1)
- in Wirtssprache implementiert ⟶ Vorteile von Haskell verfügbar ✔

Parsec: Eigenschaften

- umfangreiche Fehlerausgabe
- Bibliothek implementierter Parser-Bausteine
- Backtracking möglich
- kann lexikalische Analyse mit ausführen

Schnittstellen: Char Parser (Text.Parsec.Char)

```
oneOf :: Stream s m Char => [Char] -> ParsecT s u m Char
noneOf :: Stream s m Char => [Char] -> ParsecT s u m Char
spaces :: Stream s m Char => ParsecT s u m ()
space :: Stream s m Char => ParsecT s u m Char
newline :: Stream s m Char => ParsecT s u m Char
tab :: Stream s m Char => ParsecT s u m Char
upper :: Stream s m Char => ParsecT s u m Char
lower :: Stream s m Char => ParsecT s u m Char
alphaNum :: Stream s m Char => ParsecT s u m Char
letter :: Stream s m Char => ParsecT s u m Char
digit :: Stream s m Char => ParsecT s u m Char
hexDigit :: Stream s m Char => ParsecT s u m Char
octDigit :: Stream s m Char => ParsecT s u m Char
char :: Stream s m Char => Char -> ParsecT s u m Char
anyChar :: Stream s m Char => ParsecT s u m Char
satisfy :: Stream s m Char => (Char -> Bool) -> ParsecT s u m Char
string :: Stream s m Char => String -> ParsecT s u m String
```

Schnittstellen: Kombinatoren (Text.Parsec.Combinator)

```
choice :: Stream s m t => [ParsecT s u m a] -> ParsecT s u m a
count :: Stream s m t => Int -> ParsecT s u m a -> ParsecT s u m [a]
between :: Stream s m t => ParsecT s u m open -> ParsecT s u m close -> ParsecT s u m a -> ParsecT s u m a
option :: Stream s m t => a -> ParsecT s u m a -> ParsecT s u m a
optionMaybe :: Stream s m t => ParsecT s u m a -> ParsecT s u m (Maybe a)
optional :: Stream s m t => ParsecT s u m a -> ParsecT s u m ()
skipMany1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m ()
many1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
sepBy :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
sepBy1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
endBy :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
endBy1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
sepEndBy :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
sepEndBy1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
chainl :: Stream s m t => ParsecT s u m a -> ParsecT s u m (a -> a -> a) -> a -> ParsecT s u m a
chainl1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m (a -> a -> a) -> ParsecT s u m a
chainr :: Stream s m t => ParsecT s u m a -> ParsecT s u m (a -> a -> a) -> a -> ParsecT s u m a
chainr1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m (a -> a -> a) -> ParsecT s u m a
eof :: (Stream s m t, Show t) => ParsecT s u m ()
notFollowedBy :: (Stream s m t, Show t) => ParsecT s u m t -> ParsecT s u m ()
manyTill :: Stream s m t => ParsecT s u m a -> ParsecT s u m end -> ParsecT s u m [a]
lookAhead :: Stream s m t => ParsecT s u m a -> ParsecT s u m a
anyToken :: (Stream s m t, Show t) => ParsecT s u m t
```

Verwendung von Parsec

Verwendung von Parsec

```
module ParsecUse where
import Text.ParserCombinators.Parsec
identifier :: Parser String
identifier = string "Mathias"
             <|> string "Thomas"
doParse :: Show a => Parser a -> String -> IO ()
doParse p xs = case (parse p "" xs) of
             Left err -> print err
             Right xs -> print xs
```

Aufruf: doParse identifier “Mathias”

Verbesserte Fehlerbehandlung

Fehlerbehandlung

```
module ParsecUse where
import Text.ParserCombinators.Parsec
identifier' :: Parser String
identifier' = string "Mathias"
           <|> string "Thomas"
           <?> "Name eines Vortragenden"
doParse :: Show a => Parser a -> String -> IO ()
doParse p xs = case (parse p "" xs) of
           Left err -> print err
           Right xs -> print xs
```

Aufruf: doParse identifier' "Ernie"

Folgen, Verzweigungen und Aktionen

Klammerschachtelung

```
nesting :: Parser Int
nesting = do {char '('
             ; n <- nesting
             ; char ')'
             ; m <- nesting
             ; return (max (n+1) m)
             }
         <|> return 0
```

Aufruf: doParse nesting “(()()(()()))”

Lookahead I

naiver Ansatz

```
day = string "Montag"
      <|> string "Dienstag"
      <|> string "Mittwoch"
      <|> string "Donnerstag"
      <|> string "Freitag"
      <|> string "Samstag"
      <|> string "Sonntag"
```

Aufruf: doParse day "Donnerstag"

Lookahead II

richtigerer Ansatz

```
day' = try (string "Montag")
       <|> try (string "Dienstag")
       <|> try (string "Mittwoch")
       <|> try (string "Donnerstag")
       <|> try (string "Freitag")
       <|> try (string "Samstag")
       <|> string "Sonntag"
```

- Aufruf: doParse day' "Donnerstag"
- Beeinträchtigung der Effizienz, wenn sehr viel Backtracking :-(
- Alternative: Links-Faktorisierung, um Eindeutigkeit herzustellen

Gliederung

5 Demobeispiel - Parsec
- Scanner und Parser

EBNF

ident ::= letter {letter | digit}
boolean ::= "TRUE" | "FALSE"
block ::= stmt {";" stmt}
stmt ::= "DoIt" | ident ":=" expr |
    "IF" expr "THEN" block ["ELSE" block] "ENDIF" |
    "WHILE" expr "DO" block "ENDWHILE"
expr :: = ident | boolean | UnOp expr | expr BinOp expr |
    "(" expr ")"
UnOp ::= "!"
BinOp ::= "&&" | "=="

- Kommentare: {- hier steht ein Kommentar -}

Syntaxbaum

```
type Ident = String
data Block = Block [Stmt] deriving Show
data Stmt = DoIt | Assign Ident Expr | If Expr Block Block |
          While Expr Block deriving Show
data Expr = Id Ident | Boolean Bool | UnEx UnOp Expr |
          BinEx BinOp Expr Expr deriving Show
data UnOp = Not deriving Show
data BinOp = And | Eq deriving Show
```

Definition der Syntaxbestandteile

```
synDef = emptyDef { commentStart = "{-"
                  , commentEnd = "-}"
                  , nestedComments = True
                  , identStart = letter
                  , identLetter = alphaNum
                  , opStart = oneOf ":!&="
                  , opLetter = oneOf "&="
                  , reservedOpNames= [":=", "!", "&&","=="]
                  , reservedNames = ["TRUE", "FALSE", "DoIt", "IF", "THEN",
                      "ELSE", "ENDIF", "WHILE", "DO", "ENDWHILE"]
                  , caseSensitive = True
                  }
```

- Spezifizieren der Syntax

Erzeugung einfacher Parser

```
TokenParser{ parens = m_parens
           , identifier = m_identifier
           , reservedOp = m_reservedOp
           , reserved = m_reserved
           , semiSep1 = m_semiSep1
           , whiteSpace = m_whiteSpace } = makeTokenParser synDef
```

✔ Selbst geschriebene Parser können auf Tokenebene arbeiten
✖ Whitespace vor den Tokens wird *nicht* automatisch entfernt

Parser für Expression

```
exprparser :: Parser Expr
exprparser = buildExpressionParser table term <?> "expression"
table = [ [Prefix (m_reservedOp "!" >> return (UnEx Not))]
        , [Infix (m_reservedOp "&&" >> return (BinEx And)) AssocLeft]
        , [Infix (m_reservedOp "==" >> return (BinEx Eq)) AssocLeft]
        ]
term = m_parens exprparser
        <|> liftM Id m_identifier
        <|> (m_reserved "TRUE" >> return (Boolean True))
        <|> (m_reserved "FALSE" >> return (Boolean False))
```

Parser für Statements I

```
mainparser :: Parser Block
mainparser = m_whiteSpace >> stmtparser
        where
        stmtparser :: Parser Block
        stmtparser = liftM Block (m_semiSep1 stmt1)
        stmt1 = (m_reserved "DoIt" >> return DoIt)
        <|> do { i <- m_identifier
            ; m_reservedOp ":="
            ; e <- exprparser
            ; return (Assign i e)
            }
        <|> do { m_reserved "IF"
            ; c <- exprparser
            ; m_reserved "THEN"
```

Parser für Statements II

```
          ; p <- stmtparser
          ; q <- option (Block [])
                   (do{m_reserved "ELSE"; s <- stmtparser; return s})
          ; m_reserved "ENDIF"
          ; return (If c p q)
          }
     <|> do { m_reserved "WHILE"
          ; c <- exprparser
          ; m_reserved "DO"
          ; p <- stmtparser
          ; m_reserved "ENDWHILE"
          ; return (While c p)
          }
```

Aufruf des Parsers

```
runParser :: String -> IO ()
runParser s = case parse mainparser "" s of
          { Left err -> print err
          ; Right out -> print out
          }
```

✔ Hiermit können beliebige Codefragmente geparst werden

Fragen

Hier ist Platz für Eure Fragen

Vielen Dank

Vielen Dank für Eure Aufmerksamkeit !!!

Bücher

Bird, Richard
Introduction to Functional Programming using Haskell
Prentice Hall, 1998.

Hutton, Graham
Programming in Haskell
Cambridge University Press, 2007.

Online-Quellen I

HaskellWiki
Parsing expressions and statements
Internet: http://www.haskell.org/haskellwiki/Parsing_expressions_and_statements, Abruf: 2008-12-14, Stand: 2008.

Hutton, Graham/ Meijer, Erik
Monadic Parser Combinators
Internet: http://www.cs.nott.ac.uk/~gmh/pearl.pdf, Abruf: 2008-12-17, Stand: 1998.

Online-Quellen II

Hutton, Graham/ Meijer, Erik
Monadic Parser Combinators
Internet: http://www.cs.nott.ac.uk/~gmh/monparsing.pdf, Abruf: 2008-12-17, Stand: 1996.

O'Sullivan, Bryan/ Stewart, Don/ Goerzen, John
Real World Haskell (E-Book; O'Reilly Media)
Internet: http://book.realworldhaskell.org/read/, Abruf: 2008-12-16, Stand: 2008.

Online-Quellen III

Schmidt, Prof. Dr. Uwe
Compilerbau & Formale Sprachen (FH Wedel - Vorlesungsunterlagen)
Internet: http://www.fh-wedel.de/~si/vorlesungen/cb/cb.html, Abruf: 2008-12-14, Stand: 2008.