

Compiler-FrontEnd

Pascal Hof

Haskell-Stammtisch, 19.10.2011

- 1 Einführung
- 2 Werkzeuge
- 3 Monadische Parser
Einführung
Kombination von Parsern
- 4 Applikative Parser
- 5 Annotationen
- 6 Fazit

Aufbau eines Compilers

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

Frontend

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse

Backend

- Zwischencodeerzeugung
- Codeoptimierung
- Codegenerator

“Cross-cutting layers”

- Symboltabelle
- Fehlerbehandlung

Hier und heute:

lexikalische und syntaktische Analyse

Lex. und synt. Analyse in Haskell

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

Zwei Möglichkeiten zum Entwurf der ersten beiden Phasen:

Seperate Umsetzung

- Lexikalische Analyse: `lex :: String -> [Token]`
- Syntaktische Analyse: `parse :: [Token] -> Ast`

Kombination

- Syntaktische Analyse: `parse :: String -> Ast`

Lexer

```
1 lexer :: String → [Token] → [Token]
2 lexer('&':xs) ts = ts ++ [TokAnd] ++ lexer xs
3 -- weitere Fallunterscheidungen
4 lexer xs ts = takeWhile cond xs -- div. Listenoperationen
```

Parser

```
1 parse :: [Token] → Ast
2 parse (TokOpenBrace
3       :TokIdent i1
4       :TokAnd
5       :TokIdent i2
6       :TokCloseBrace
7       :[]) = And i1 i2
```

Sehr aufwändig, v.a. bei wechselseitig rekursiven Datentypen

lexikalische Analyse mit Alex

Alex ist eine DSL zur Definition lexikalischen Analysen.
Ausschnitt einer Lexer-Definition mit *Alex*:

```
1 data Token = TBrOp | TBrCl | TConst Int | TAdd | TMul
3 tokens :-
4   $white+ ;
5   (      { λs → TBrOp }
6   )      { λs → TBrCl }
7   $digit+ { λs → TConst (read s) }
8   +      { λs → TAdd }
9   *      { λs → TMul }
11 alexScanToken :: String → [Token]
```

Syntaktische Analyse mit Happy

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

Mit *Alex* definierte Parser können benutzt werden.

```

1 data Ast = Const Int | Add Ast Ast | Mul Ast Ast
3 %token
4   '('      { TBrOp }
5   ')'      { TBrCl }
6   const    { TConst $$ }
7   '+'      { TAdd }
8   '*'      { TMul }

10 Ast : '(' Ast '+' Ast ')'      { Add $2 $4 }
11     | '(' Ast '*' Ast ')'      { Mul $2 $4 }
12     | const                      { Const $1 }

```

Parser mit BNFC

BNFC erzeugt *Alex*- und *Happy*-Dateien aus annotierter Grammatik.

Eingabedatei

```
1  Con. Ast ::= digit+;  
2  Add. Ast ::= "(" Ast "+" Ast " )";  
3  Mul. Ast ::= "(" Ast "*" Ast " )";
```

Was wird erzeugt?

- Abstrakte Syntax
- Datei für Lexer-Generator (für Alex, JLex oder Flex)
- Datei für Parser-Generator (für Happy, CUP oder Bison)
- Pretty-Printer
- tex-Datei mit Sprachspezifikation (in BNF)

... für die Nutzung mit Haskell, Java, C, C++ oder O'Caml

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

Pro

Mit *BNFC*, *Alex* und *Happy* lassen sich sehr schnell und einfach Parser definieren.

Contra

Die so erzeugten Parser lassen sich nur schwierig an eigene Zwecke (z.B. Positionen für Token) anpassen.

Typ eines Parsers

```
1 | type P a = String → a
```

Nicht komplette Eingabe konsumieren

```
1 | type P a = String → (a,String)
```

Liste zur Fehlerbehandlung

```
1 | type P a = String → [(a,String)]
```

In Datentyp einbetten

```
1 | data Parser a = Parser {unP :: String → [(a,String)]}
```

Variante

```
1 | data Parser s a = Parser {unP :: s → [(a,s)]}
```

Typ eines Parsers

Wiederholung

```
1 | data Parser a = Parser {unP :: String → [(a,String)]}
```

Idee für monadische Parser

Liste kann ...

- leer sein → Parser kann Eingabe nicht lesen.
- ein Element enthalten → Parser hat genau eine Möglichkeit die Eingabe zu lesen.
- mehrere Elemente enthalten → Parser hat mehrere Möglichkeiten die Eingabe zu lesen.

Einige einfache Parser (1/2)

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

immer fehlschlagender Parser

```
1 | zero :: Parser a
2 | zero = Parser $ \_ → []
```

immer erfolgreicher Parser

```
1 | result :: a → Parser a
2 | result v = Parser $ \input → [(v,input)]
```

Einige einfache Parser (2/2)

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

Parser für Ziffern

```
1 digit :: Parser Int
2 digit = Parser $ \input → case input of
3   (x:xs) → if isDigit x then [(read [x],xs)] else []
4   []     → []
```

Parser für Character

```
1 char :: Char → Parser Char
2 char c = Parser $ \input → case input of
3   (x:rest) → if c ≡ x then [(x,rest)] else []
4   _       → []
```

Wie können Parser kombiniert werden?

Sequentielle Komposition

```
1 | seq :: Parser a → (a → Parser b) → Parser b
2 | (Parser p) 'seq' f = Parser $
3 |   λinput → concat [unP (f a) rest | (a,rest) ← p input]
```

“Parallele” Komposition

```
1 | par :: Parser a → Parser a → Parser a
2 | (Parser p1) 'par' (Parser p2) = Parser $ λinput →
3 |   case p1 input of
4 |     r@(_:_) → r
5 |     []      → p2 input
```

Parser ist eine Monade

Monad-Instanz

```
1 instance Monad Parser where
2   --return :: a → Parser a
3   return = result
5   --(>>=) :: Parser a → (a → Parser b) → Parser b
6   (>>=) = seq
```

MonadPlus-Instanz

```
1 instance Plus Parser where
2   --mzero :: Parser a
3   mzero = zero
5   --mplus :: Parser a → Parser a → Parser a
6   mplus = par
```

Beispiel: Monadische Parser (1/2)

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

Parser-Funktion

```
1 parse :: Parser a → String → [a]
2 parse (Parser p) = map fst ∘ p
```

Erkennung des Präfixes 'ABC'

```
1 pABC :: Parser ()
2 pABC = char 'A' >>> char 'B' >>> char 'C' >>> return ()

4 ghci> parse pABC "ABc"  ⇒ []
5 ghci> parse pABC "ABC"  ⇒ [()]
6 ghci> parse pABC "ABCA" ⇒ [()]
```

Beispiel: Monadische Parser (2/2)

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

Monadischer Parser mit Rückgabewert

```
1 pDigitCharA :: Parser (Int,Char)
2 pDigitCharA = do
3   i ← digit
4   c ← char 'A'
5   return (i,c)
```

Kurz:

```
1 pDigitCharA :: Parser (Int,Char)
2 pDigitCharA = liftM2 (,) digit (char 'A')
```

Applicative vs. Monad

Jede Monad-Instanz ist auch Applicative-Instanz. Die andere Richtung gilt im Allgemeinen nicht.

Häufig ist Applicative schon mächtig genug und man braucht keine Monade.

```
1 class Functor f => Applicative f where
2   pure  :: a -> f a
3   (<*>) :: f (a -> b) -> f a -> f b
4   (*>)  :: f a -> f b -> f b
5   (<*)  :: f a -> f b -> f a
6   some  :: f a -> f [a]
7   many  :: f a -> f [a]
```

```
1 instance Monad m => Applicative m where
2   pure = return
3   (<*>) = ap
```

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

```
1 pDZ :: Parser (Char,Char)
2 pDZ = (,) <$> char 'd' <*> char 'z'

4 pDZ :: Parser (Char,Char)
5 pDZ = liftA2 (,) (char 'd') (char 'z')
```

```
1 pId :: Parser Int
2 pId = char '(' *> pDigit <*> char ')'
```



```
4 ghci> parse pId "(3)" ==> [3]
```

... ist ähnlich zu MonadPlus.

```
1 class Applicative f => Alternative f where
2   empty :: f a
3   (<|>) :: f a -> f a -> f a
4   some  :: f a -> f [a]
5   many  :: f a -> f [a]
```

Ableitbar aus MonadPlus

```
1 instance MonadPlus m => Alternative m where
2   empty = mzero
3   (<|>) = mplus
```

Beispiele für Alternative

```

1 | pId :: Parser [Int]
2 | pId = char '(' *> many pDigit <*> char ')'

```

```

1 | pAorB :: Parser Char
2 | pAorB = char 'A' <|> char 'B'

```

Beispiel aus der Einleitung

```

1 | data Ast = Const Int | Add Ast Ast | Mul Ast Ast
3 | pAst :: Parser Ast
4 | pAst =
5 |     Const <$> digit
6 |     <|> Add   <$> (char '(' *> pAst <*> char '+')
7 |               <*> pAst <*> char ')'
8 |     <|> Mul   <$> (char '(' *> pAst <*> char '*')
9 |               <*> pAst <*> char ')'

```

... ist die wohl bekannteste Bibliothek für Parserkombinatoren in Haskell.

```
1 | data ParsecT s u m a = -- kompliziert
```

Typvariablen von ParsecT

- Typ des Eingabestreams s (z.B. String oder Token)
- Zustand u (z.B. Anzahl gewisser Strukturen zählen)
- eingebettete Monade m (hier: Identity)
- Rückgabetyt des Parsers a

Mit *Parsec* lassen sich Parser in monadischer oder applikativer Schreibweise definieren.

Annotationen im Syntaxbaum

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

Häufig enthält ein Syntaxbaum auch Annotationen, z.B. Positionen in der Quelldatei.

In der Regel werden diese in die abstrakte Syntax eingebettet:

```
1 data SrcPoint = SrcPoint { column :: Int , row :: Int}
3 data SrcPos = SrcSpan SrcPoint SrcPoint
4             | SrcPos SrcPoint
6 data Ast = Foo Int SrcPos
7           | Bar Ast SrcPos
8           | Foobar Tree SrcPos
10 data Tree = Node String SrcPos
11           | Branch String SrcPos Tree Tree
```

Abstraktion von Annotationen

Probleme

- Änderungen des Annotationstyps erfordern Überarbeitung der kompletten abstrakten Syntax.
- keine Schnittstelle, um Annotationen an einem beliebigen Knoten zu erhalten.

Erstes Problem kann durch Abstraktion von Annotationen umgangen werden.

```
1 data Ast a = Foo {i :: Int, annAst :: a}
2             | Bar {ast :: Ast, annAst :: a}
3             | Foobar {tree :: Tree, annAst a}

5 data Tree a = Node { str :: String, annTree a}
6              | Branch {strB :: String, annTree :: a
7                      ,t1 :: Tree, t2 :: Tree}
```

Schnittstelle zu Annotationen

Einführung

Werkzeuge

Monadische
ParserEinführung
Kombination
von ParsernApplikative
Parser

Annotationen

Fazit

```
1 class Annotated t where
2   getAnn :: t a → a
4 instance Annotated Ast where getAnn = annAst
5 instance Annotated Tree where getAnn = annTree
```

Mit *TemplateHaskell* können diese Instanzen für beliebige Datentypen abgeleitet werden. (noch nicht implementiert)

Den Syntaxbaum annotieren

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

Wie lässt sich ein Syntaxbaum mit Positionen erzeugen?

- bei *Parsec* kann die aktuelle Position mit `getSourcePosition` erhalten werden.
- *Alex* und *Happy* können mit ein wenig Zusatzaufwand annotierte Syntaxbäume erzeugen.
- *BNFC* kann es (noch?) nicht.

Je mächtiger das Werkzeug, desto schwieriger fällt Umsetzung von "Sonderwünschen".

Wir haben ...

- ... gängige Werkzeuge zum Entwurf von Compilern eingeführt.
- ... monadische Parser definiert.
- ... applikative Parser definiert.
- ... Syntaxbäume mit Positionen annotiert.

Werkzeuge zum Compilerbau

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

Parsekombinatoren als EDSL, z.B. *Parsec*

- keine neue Sprache zu lernen.
- Typsystem von Haskell und vertraute Hilfsfunktionen nutzbar.
- nahezu jeder Sonderwunsch umsetzbar.

DSL-Lösung, z.B. *Alex*, *Happy* oder *BNFC*

- jeweilige Sprache ist zu erlernen.
- Haskellfunktionen können nur bedingt genutzt werden.
- Entwurf “einfacher” Übersetzer sehr leicht, v.a. mit *BNFC*

Einführung

Werkzeuge

Monadische
Parser

Einführung
Kombination
von Parsern

Applikative
Parser

Annotationen

Fazit

- Monadic Parser Combinators (Hutton,Meijer)
- Packet language-python von Hackage (Annotationen)