Informatik Seminar 2003 - Parsen

Jens Kulenkamp

8. Dezember 2003

Das Programm

Parser

Aufgaben und Eigenschaften eines Parsers:

- Die Eingabe ist ein Text oder Token
- Analysiert wird die Syntax anhand einer Grammatik
- Die Ausgabe/Erzeugung ist Zwischencode bestehend aus einem Syntax- oder Programmbaum

Grammatik

Eine kontextfreie Grammatik ist ein Quadrupel $G = (T, N, P, S)$ wobei gilt:

- T: ist ein Alphabet von Terminalsymbolen
- N: ist ein Alphabet von Nichtterminalensymbolen und beschreiben strukturierte Einheiten.
- P: ist eine Menge von Produktionsregeln. Eine Produktionsregel besteht aus einem Paar von Nichtterminal und einer Folge von Nichtterminal- und/oder Terminalsymbolen.
- S: ist das Startsymbol. Die Menge aller aus dem Startsymbol S ableitbaren Terminalsymbolen T ist die erzeugte Sprache G.

1. Grammatik für arithmetische Ausdrücke

$$G = (T, N, P, S)$$
$$T = \{num, +, -, *, /\}$$
$$N = \{E\}$$
$$P = \{E\}$$
$$E = \{E -> num | E + E | E * E | E - E | E/E\}$$
$$S = E$$

E steht für *expression*
Beispiel: “10 + 2” und “4 * 5 - 8”

1. Grammatik für arithmetische Ausdrücke

Zusammenfassung:

- Der Syntaxbaum wird von oben nach unten aufgebaut. Das Prinzip nennt sich *top-down*.
- Die Grammatik ist mehrdeutig (unterschiedliche Syntaxbäume für den gleichen Ausdrück)!
- Lösung: Prioritäten, Assoziativität oder eine natürlich strukturierte Grammatik (z.B. durch Klammerung).

2. Grammatik für arithmetische Ausdrücke

$$G = (T, N, P, S)$$
$$T = \{num, +, -, *, /\}$$
$$N = \{E, T, F\}$$
$$P = \{E -> E + T | E - T | T, T -> T * F | T / F | F, F -> num\}$$
$$S = E$$

E steht für *expression*
F steht für *factor*
Beispiel: "`4 * 5 - 8`"

2. Grammatik für arithmetische Ausdrücke

Zusammenfassung:

- Für jeden Ausdruck gibt es nur noch einen gültigen Syntaxbaum
- zusätzliches nichtterminal Symbol F.
- Die Grammatik ist durch F eindeutig geworden
- links Assoziativ
- Achtung Linksrekursion!

Motivation:

einen eigenen Parser bauen.

- Entwicklung eines Typ `Parse`
- Es werden keine Monaden verwendet
- elementare Parser (*engl. elementary parsers*)
- Kombinatoren für elementare Parser (*engl. parser combinators*)

Der Typ "Parse"

und das Problem:

- Gegeben sei ein Programmbaum repräsentiert durch ein `String`
- Das Ergebnis soll ein Baum sein zum Beispiel der Datentyp `Tree`
- Der Datentyp in Haskell könnte so aussehen:

```
type Parse = String -> Tree
```

Der Typ "Parse"

Der alte Datenyp:

```
type Parse = String -> Tree
```

und das Problem:

- Ein Parser ruft andere Parser auf oder rekursiv sich selbst
- Es muss somit möglich sein, das Zwischenergebnis und den Rest zu speichern
- Ohne die Verwendung von globalen Datenstrukturen
- Der neue Datentyp in Haskell könnte so aussehen:

```
type Parse = String -> ( Tree, String )
```

Der Typ "Parse"

Der alte Datenyp:

```
type Parse = String -> ( Tree, String )
```

und das Problem:

- Der `Tree` ist nicht deklariert und ist abhängig vom Anwendungsfall (XML-Baum, Programmbaum, arithmetische Ausdrücke...)
- Der neue Datentyp in Haskell könnte so aussehen:

```
type Parse  a = String -> ( a, String )
```

- Beispiel: ein Parser für arithmetische Ausdrücke `Parser Expr` und `Expr`

Der Typ "Parse"

Der alte Datenyp:

```
type Parse a = String -> ( a, String )
```

und das Problem:

- ein Parser hat kein oder mehr als ein Ergebnis
- Der neue Datentyp in Haskell könnte so aussehen:

```
type Parse  a = String -> [ ( a, String ) ]
```

- In diesem Fall können mehrere Ergebnisse gespeichert werden (list of successes, *backtracking*).

Der Typ "Parse"

Der alte Datenyp:

```
type Parse a = String -> [ ( a, String ) ]
```

und das Problem:

- Bislang kann der Typ nur mit Strings arbeiten
- Lösung weiteres Abstrahieren `String -> b`
- Der neue Datentyp in Haskell könnte so aussehen:

```
type Parse b a = [b] -> [ ( a, [b] ) ]
```

oder etwas lesbarer:

```
type Parse symbol result = [symbol] -> [ ( result, [symbol] ) ]
```

- Dieser Datentyp wird auf den folgenden Seiten verwendet

Elementarer Parser - succeed

Ein Parser der die Parameter nicht verarbeitet sondern nur Typ `Parse` liefert.

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

Elementarer Parser - symbola

Ein Parser der das Symbol 'a' erwartet

```
> symbola :: Parse Char Char
> symbola []     = []
> symbola (x:xs)
>     | x=='a'    = [ ('a',xs) ]
>     | otherwise = []

? symbola "abc"
[('a',"bc")]

? symbola "bcd"
[]
```

Besser: Ein Parser der ein übergebenes Symbol erwartet

Elementarer Parser - symbol

Ein Parser der ein Symbol s erwartet und unabhängig vom Datentyp Char ist (Vorraussetzung Gleichheitstest)

```
> symbol :: Eq a => a -> Parse a a
> symbol s []       = []
> symbol s (x:xs)
>     | s==x        = [ (x,xs) ]
>     | otherwise = []

? symbol 'a' "abc"
[('a',"bc")]

? symbol 'a' "bcd"
[]
```

Elementarer Parser - spot

Ein Parser der eine beliebige Funktion (Ergebnistyp `Bool`) auf ein Symbol anwendet.

```
> spot            :: (a-> Bool) -> Parse a a
> spot  f []       = []
> spot  f (x:xs)
>     | f x        = [(x,xs)]
>     | otherwise = []
```

Beispiel: Eine verfeinerte Version des Parsers `symbol`

```
> symbol  :: Eq a => a -> Parse a a
> symbol t = spot (==t)

? symbol 'a' "abc"
[('a',"bc")]
```

Parser kombinator

- Parser kombinator *engl. Parser combinator*
- setzen elementare Parser zusammen
- sind Parser für terminal- und nichterminal- Symbole
- Sequenzierung (erst `p1`, dann `p2`) `<*>`
- Alternative (`p1` oder `p2`) `<|>`

Parser kombinatoren - Alternative

Auswahl: Parser p1 oder p2. Das Ergebnis p1 oder p2

```
> infixr 4 <|>
> (<|>)              :: Parse a b -> Parse a b -> Parse a b
> (p1 <|> p2) input  = (p1 input) ++ (p2 input)

? (dig <|> symbol 'a') "abc"
[('a',"bc")]

? (dig <|> symbol 'a') "123"
[('1',"23")]
```

Parser kombinatoren - Sequenzierung

Parser-Folge: erst Parser p1 und den Rest xs1 als Eingabe für Parser p2. Das Ergebnis von x1 und x2 als Paar und den Rest

```
> infixr 6 <*>
> (<*>)              :: Parse a b -> Parse a c -> Parse a (b,c)
> (p1 <*> p2) input  = [( (x1, x2), xs2 )
>                       | (x1, xs1) <- p1 input
>                      ,  (x2, xs2) <- p2 xs1
>                       ]

? (symbol 'a' <*> symbol 'b') "abc"
[(('a','b'),"c")]

? (symbol 'a' <*> symbol 'b') "123"
[]
```

Parser kombinatoren - Umwandler

Umwandler: Wende auf den Parser `p` die Funktion `f` an

```
> infixr 5 <@
> (<@)            :: Parse a b -> ( b -> c ) -> Parse a c
> (p <@ f) input   = [ (f x,xs)
>                     | (x,xs) <- p input
>                   ]
```

Anwendungsbeipiel: ein Parser der ein Character zu Integer umwandelt

```
> digit :: Parse Char Int
> digit  = spot isDigit <@ f
>    where f c = ord c - ord '0'

? digit "123"
[(1,"23")]
```

Parser kombinatoren - Listen

Beispiel: Ein Zahl z.B. "123", repräsentiert aus einer Liste von Ziffern, muss in ein Symbol umgewandelt werden

```
> many   :: Parse a b -> Parse a [b]
> many p  = (p <*> many p <@ list)
>         <|>
>         (succeed [])
>    where list (x,xs) = x:xs

? many (spot isDigit) "123abc"
[("123","abc"),("12","3abc"),("1","23abc"),("","123abc")]

? many (spot isAlpha) "test"
[("test",""),("tes","t"),("te","st"),("t","est"),("","test")]
```

Parser kombinatoren - Option

Beispiel: Eine Zahl kann positiv 12 oder negativ -12 sein. Das Symbol ”-” ist optional!

```
> option          :: Parse a b -> Parse a [b]
> option p input   = ( ( p <@ (:[]) )
>                     <|>
>                       ( succeed [] ) ) input

? option (symbol '-') "-123"
[("-","123"),("","-123")]

? option (symbol '-') "123"
[("","123")]
```

Wiederholung

```
-  type Parse symbol result = [symbol] -> [ ( result, [symbol] ) ]
-  succeed :: b -> Parse a b
-  symbol  :: Eq a => a -> Parse a a
-  spot    :: (a-> Bool) -> Parse a a
-  (<|>)   :: Parse a b -> Parse a b -> Parse a b
-  (<*>)   :: Parse a b -> Parse a c -> Parse a (b,c)
-  (<@)    :: Parse a b -> ( b -> c ) -> Parse a c
-  many    :: Parse a b -> Parse a [b]
```

Idee

Ein Taschenrechner zum Berechnen von ganzzahligen Operationen.

- Zahlen: “12” und “-88”, negative Zahlen werden durch “-” dargestellt
- Operatoren: “+”, “-”, “*”, “/”, “%”
- “%” steht für mod
- Nur ganzzahlige Division
- Leerzeichen sind nicht erlaubt
- Jeder Ausdruck muss geklammert sein!
  Beispiel: “(23-(20/2))”

Was wird benötigt?

- Einen Parser
- Eine Grammatik
- Hilfsfunktionen `isOp`, `charToOp`, `charListToExpr`

Hilfsfunktionen

Umwandeln einer Ziffernliste in den Datentyp Expr

```
> charListToExpr :: [Char] -> Expr
> charListToExpr []     = Lit 0
> charListToExpr (c:xs) = Lit ( natural (c:xs) )

> litParse :: Parse  Char Expr
> litParse
>     = (
>        option (symbol '-') <*>  many1   (spot isDigit)
>       ) <@  charListToExpr.join
>     where join = uncurry (++)

? litParse "12"
[(Lit 12,""),(Lit 1,"2")]

? litParse "-12"
[(Lit (-12),""),(Lit (-1),"2")]
```

Der Parser

```
> parser :: Parse Char Expr
> parser  = litParse <|> opExprParse

> opExprParse :: Parse Char Expr
> opExprParse  = (symbol '('<*>
>                 parser    <*>
>                 spot isOp <*>
>                 parser    <*>
>                 symbol ')'
>                ) <@ makeExpr
>    where
>    makeExpr :: (a,(Expr,(Char,(Expr,b)))) -> Expr
>    makeExpr (_,(e1,(bop,(e2,_))))  = Op (charToOp bop) e1 e2
```

Was ist Parsec?

- mächtige Parser Bibliothek
- top-down Parser mit Monaden
- LL[1] Grammatik (default) und Backtracking (optional)
- lexikalische Analyse und Parser zusammen
- ermöglicht aussagekräftige Fehlermeldungen
- `ParsecExpr`: erweiterung für Ausdrücke
- `ParsecLanguage`: Spracherweiterung für Programmiersprachen
- `ParsecToken`: parsen von lexikalischen Tokens
- `ParsecErr`: für Warnungen und Fehlermeldungen
- und viele weitere Bibliotheken...

Beispiel mit Parsec

```
> -- Ein Parser fuer Identifier
> simple1 :: Parser [Char]
> simple1  = many letter

? parseTest simple1 "abc123"
"abc"

> -- Ein Parser fuer Hex Zahlen
> simple2 :: Parser [Char]
> simple2  = many hexDigit

? parseTest simple2 "ABC123xyz"
"ABC123"
```

Sequenz

Ein Parser der eine Sequenz von Klammernpaar(en) “()” erwartet.

```
> openClose1 :: Parser ()
> openClose1  = do{ char '('
>                 ; openClose1
>                 ; char ')'
>                 ; openClose1
>                 }
>              <|> return ()

? parseTest openClose1 "(()())"
()

? parseTest openClose1 "(()"
parse error at (line 1, column 4):
unexpected end of input
expecting "(" or ")"
```

Berechnung ausführen

Berechne die Anzahl der Klammernpaare

```
> noOfBrace = do{ char '{'
>               ; m <- noOfBrace
>               ; char '}'
>               ; n <- noOfBrace
>               ; return (1+m+n);
>               }
>             <|> return 0

? parseTest noOfBrace "{}{}"
2

? parseTest noOfBrace "{{}}{}"
3
```

1. Auswahl

Die Auswahl entweder “(a)” oder “(b)”.

```
> or0 = string "(a)"
>       <|>
>       string "(b)"

? parseTest or0 "(a)"
"(a)"

? parseTest or0 "(b)"
parse error at (line 1, column 1):
unexpected "b"
expecting "(a)"
```

2. Auswahl

Die verbesserte Auswahl entweder “(a)” oder “(b)”.

```
> or1 = do { char '('
>        ; char 'a' <|> char 'b'
>        ; char ')'
>        ; return "ok"
>        }

? parseTest or1 "(a)"
"ok"

? parseTest or1 "(b)"
"ok"
```

3. Auswahl

Die verbesserte Auswahl unter der Verwendung von `try`.

```
> or2 = try( string "(a)" )
>       <|>
>       try( string "(b)" )

? parseTest or2 "(a)"
"(a)"

? parseTest or2 "(b)"
"(b)"

? parseTest or2 "(c)"
parse error at (line 1, column 1):
unexpected "c"
expecting "(a)" or "(b)"
```

Ein Taschenrechner realisiert mit Parsec