

PEG-Parser, Pratt-Parser und Parser Combinators

BC George (FH Bielefeld)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

PEG Parser

Generative Systeme vs. Recognition basierte Systeme

- Generative Systeme:
 - formale Definition von Sprachen durch Regeln, die rekursiv angewendet Sätze/Strings der Sprache generieren
 - Sprachen aus der Chomsky Hierarchie definiert durch kontextfreie Grammatiken (CFGs) und reguläre Ausdrücke (REs)
- Recognition-basierte Systeme:
 - Sprachen definiert in Form von Regeln/Prädikaten, die entscheiden, ob ein gegebener String Teil der Sprache ist
 - Parsing Expression Grammar (PEG)

Sprache $L = \{\varepsilon, aa, aaaa, \dots\}$:

- generativ: $L = \{s \in a^* \mid s = (aa)^n, n \geq 0\}$
- recognition: $L = \{s \in a^* \mid |s| \bmod 2 = 0\}$

Motivation: Chomsky (CFGs + REs)

- ursprünglich für natürliche Sprachen
- adaptiert für maschinenorientierte Sprachen (Eleganz, Ausdrucksstärke)
- Nachteile:
 - maschinenorientierte Sprachen sollten präzise und eindeutig sein
 - CFGs erlauben mehrdeutige Syntax (gut für natürliche Sprachen)
 - Mehrdeutigkeiten schwer oder gar nicht zu verhindern
 - C++ Syntax
 - Lambda abstractions, `let` expressions und conditionals in Haskell
 - generelles Parsen nur in super-linearer Zeit

Motivation: PEG

- stilistisch ähnlich zu CFGs mit REs (EBNF)
- **Kernunterschied:** priorisierender Auswahloperator ($/$ statt $|$)
 - alternative Muster werden der Reihe nach getestet
 - erste passende Alternative wird verwendet
 - Mehrdeutigkeiten werden dadurch unmöglich
 - nahe an der Funktionsweise von Parsern (Erkennen von Eingaben)
 - PEGs können als formale Beschreibung eines Top-Down-Parsers betrachtet werden
- Beispiel:
 - EBNF: $A \rightarrow a b \mid a$ und $A \rightarrow a \mid a b$ sind äquivalent
 - PEG: $A \leftarrow a b / a$ und $A \leftarrow a / a b$ sind verschieden

Definition PEG

Eine Parsing Expression Grammar (PEG) ist ein 4-Tuple $G = (V_N, V_T, R, e_S)$ mit

- V_N eine endliche Menge von Nicht-Terminalen
- V_T eine endliche Menge von Terminalen
- R eine endliche Menge von Regeln
- e_S eine *Parsing Expression*, die als *Start Expression* bezeichnet wird.

Weiterhin gilt:

- $V_N \cap V_T = \emptyset$.
- jede Regel $e \in R$ ist ein Paar (A, e) , geschrieben als $A \leftarrow e$ mit $A \in V_N$, und e eine **Parsing Expression**
- für jedes Nicht-Terminal A existierte genau ein e mit $A \leftarrow e \in R$.

Definition Parsing Expression

Parsing Expressions werden rekursiv definiert: Seien e , e_1 und e_2 Parsing Expressions, dann gilt dies auch für

- den leeren String ε
- jedes Terminal $a \in V_T$
- jedes Nicht-Terminal $A \in V_N$
- die Sequenz $e_1 e_2$
- die priorisierende Option e_1 / e_2
- beliebige Wiederholungen e^*
- Nicht-Prädikate $!e$

Operatoren für Parsing Expression

Folgende Operatoren sind für die Konstruktion von **Parsing Expressions** erlaubt:

Operator	Priorität	Beschreibung
' '	5	String-Literal
" "	5	String-Literal
[]	5	Zeichenklasse
.	5	beliebiges Zeichen
(e)	5	Gruppierung
e?	4	Optional
e*	4	Kein-oder-mehr
e+	4	Ein-oder-mehr
&e	3	Und-Prädikat
!e	3	Nicht-Prädikat
e1 e2	2	Sequenz
e1 / e2	1	priorisierende Auswahl

- andere Sprachklasse als CFGs
- parsierbar in linearer Zeit mit unbegrenztem Lookahead (Packrat Parsing)
 - Nachteil: gesamter zu parsender Text muss im Speicher gehalten werden
- neue Möglichkeiten für das Syntax-Design von Sprachen
- Syntaxbeschreibung: keine Unterscheidung zwischen Hierarchie und lexikalischen Elementen nötig
- Herausforderung bei PEGs:
 - sind Alternativen vertauschbar, ohne die Sprache zu ändern?
 - Analog zur Frage der Mehrdeutigkeit bei CFGs

Beispiel: Formeln

```
Expr  <- Term ([-+] Term)*  
Term  <- Factor ([*/] Factor)*  
Factor <- '(' Expr ')' / [0-9]+
```

Achtung: Grammatik ist rechtsassoziativ

Beispiel: Verschachtelte Kommentare

Verschachtelte Kommentare sind möglich, da die lexikalische Syntax von PEGs nicht auf REs beschränkt ist:

```
Comment      <- CommentStart CommentBody* CommentEnd
CommentStart <- '/*'
CommentEnd   <- '*/'
CommentBody  <- Comment / (!CommentEnd AnySingleChar)
AnySingleChar <- .
```

oder in Kurzform

```
Comment <- '/*' (Comment / !'*/' .)* '*/'
```

Beispiel: Beliebige Escape-Sequenzen

Escape-Sequenzen haben meist nur eine stark eingeschränkte Syntax. Eine PEG kann beliebige Ausdrücke in eine Escape-Sequenz erlauben:

```
Expression  <- ...  
Literal     <- ["] (!["] [Character EscSequence])* ["]  
Character   <- !'\\' .  
EscSequence <- '\\(' Expression ')'
```

Zum Beispiel könnte man dadurch in einer Escape-Sequenz auf Variablen zugreifen (`\(var)`) oder arithmetische Ausdrücke verarbeiten (`\(1+2)`).

Beispiel: Verschachtelte Template Typen in C++

Bekanntes Problem mit Template-Definitionen in C++: Leerzeichen zwischen Winkelklammern nötig um Interpretation als Pipe-Operator (\gg) zu verhindern:

```
TypeA<TypeB<TypeC> > MyVar;
```

PEG erlaubt kontextsensitive Interpretation:

```
Expression      <- ...
TemplateType    <- PrimaryType (LEFTANGLE TemplateType RIGHTANGLE)?
ShiftExpression <- Expression (ShiftOperator Expression)*
ShiftOperator   <- LEFTSHIFT / RIGHTSHIFT
Spacing         <- any number of spaces, tabs, newlines or comments

LEFTANGLE      <- '<' Spacing
RIGHTANGLE     <- '>' Spacing
LEFTSHIFT      <- '<<' Spacing
RIGHTSHIFT     <- '>>' Spacing
```

Beispiel: Dangling-Else

In CFGs sind verschachtelte if-then(-else) Ausdrücke mehrdeutig (Shift-Reduce-Konflikt). Dies wird häufig durch informelle Meta-Regeln oder Erweiterung der Syntax aufgelöst. In PEGs sorgt der priorisierende Auswahloperator für das korrekte Verhalten.

```
Statement <- IF Cond Statement ELSE Statement  
           / IF Cond Statement  
           / ...
```

Beispiel: Nicht-CFG-Sprachen

Ein klassisches Beispiel einer nicht-CFG Sprache ist $a^n b^n c^n$. Diese Sprache lässt sich mit der folgenden PEG darstellen:

$$G = (\{A, B, S\}, \{a, b, c\}, R, S)$$

$$A \leftarrow a A b \mid \varepsilon$$

$$B \leftarrow b B c \mid \varepsilon$$

$$S \leftarrow (A ! b) a^* B !.$$

Operator Precedence Parsing

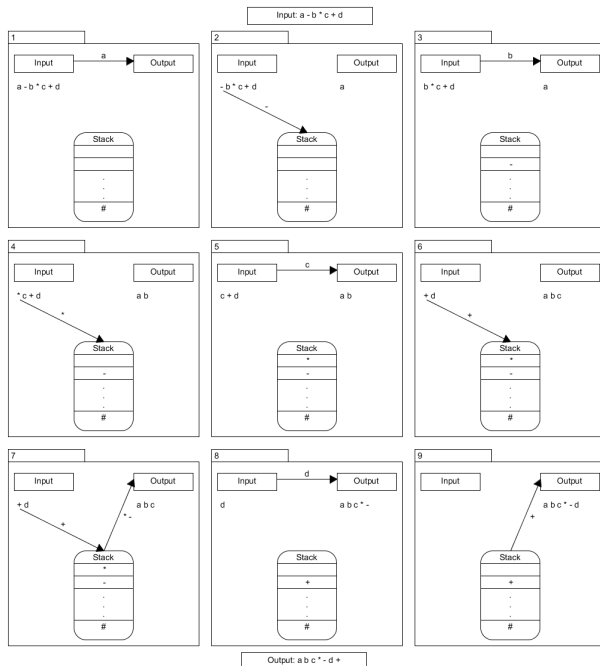
- Probleme mit CFG und BNF
- Recursive Decent Parser (RD)
 - Modelliere Grammatik(-regeln) durch rekursive Funktionen
 - Top-Down-Ansatz
 - i.d.R. LL(1) Parser (handgeschrieben)
 - Generatoren: ANTLR, Boost.Spirit
- Probleme mit RD
 - Linksrekursion (für handgeschriebene Parser von gringer Bedeutung; ersetzbar durch Schleife, EBNF)
 - Vorrangregeln (Precedence/Binding Power) und Assoziativität von Operatoren
 - Effizienz (Backtracking)

- Recognition-basiert (insofern als der Parser nicht aus einer Grammatik generiert, sondern von Hand geschrieben wird)
- verwendet Vorschautoken
- benutzt sowohl Rekursion als auch Schleifen
- interpretiert Operator-Precedence Grammatik
 - Untermenge der deterministischen kontextfreien Grammatiken
- simpel zu programmieren und kann die Operator-Tabelle während der Programmlaufzeit konsultieren
- verwendet eine Reihenfolge (precedence) für die Operatoren

Klassische Methode (RD)

- siehe Parsing Expressions by Recursive Descent
- neues Nicht-Terminal für jeden Precedence-Level
- Nachteile:
 - Anzahl der Precedence Level bestimmt Größe und Geschwindigkeit des Parsers
 - Operatoren und Precedence Levels fest in Grammatik eingebaut

Dijkstras Shunting Yard Algorithmus (SY)



Dijkstras Shunting Yard Algorithmus-Beispiel

```
while there are tokens to be read:
    read a token
    if the token is a number:
        output.add(token)
    if the token is an operator:
        if operator is a bracket:
            if left bracket:
                stack.put(token)
            else:
                check if left bracket is in stack
                while stack.top != left bracket:
                    output.add(stack.pop())
                output.add(stack.pop())    // pop left bracket
        else:
            while token.precedence < stack.top.precedence:
                output.add(stack.pop())
            stack.put(token)
```

Precedence Climbing (PC)

- Climb **down** the precedence levels (Norvell)
- Behandelt eine Expression wie verschachtelte Unterausdrücke
 - Jeder Unterausdruck enthält das gemeinsame precedence level

2 + 3 * 4 * 5 - 6

|-----| : prec 1
|-----| : prec 2

- Die Operatoren stehen mit Gewicht und Association (left, right) in einer Tabelle

Precedence Climbing-Beispiel

```
fn expr(prec_bound) {  
    result = atom() // number or an expression in braces  
  
    loop {  
        operator = nextToken()  
        (prec, assoc) = lookup(operator)  
        if (prec <= prec_bound) { break; }  
  
        next_bound = if (assoc.left?()) { prec + 1 } else { prec }  
        rhs = expr(next_bound)  
  
        result = compute(operator, result, rhs)  
    }  
  
    return result  
}
```


Top Down Operator Precedence (TDOP) Pratt Parsing

- Generalisierung von Precedence Climbing
- Verwendet eine Gewichtung (binding Power) statt einer Reihenfolge (precedence)
 - lbp = left Binding Power
 - rbp = right Binding Power

Tokenhandler

- Die Tokenhandler behandeln Notationen unterschiedlich
 - infix notation: $a = b - c$
 - led (left denotation)
 - prefix from: $a = -b$
 - nud (Null denotation)

```
op_token_sub = {  
  lbp: 10,  
  // '-' has two roles  
  prefix: () => { -expression(100) }, // negation  
  infix: (left) => { left - expression(10) }, // subtraction  
}  
  
op_token_mul = {  
  lbp: 20,  
  prefix: none, // '*' has only one role  
  infix: (left) => { left * expression(20) }, // multiplication  
}
```

Top Down Operator Precedence-Beispiel

```
fn expression(rbp = 0) {  
  token = tokens.next()  
  left = token.prefix()  
  while rbp < tokens.current().lbp {  
    token = tokens.next()  
    left = token.infix(left)  
  }  
  
  return left  
}
```

Right Associative TDOP

- Der Parser behandelt die folgende Potenzierungsoperatoren als Unterausdrücke des ersten Unterausdrucks
- Dies wird erreicht, indem wir den Ausdruck im Handler der Potenzierung mit einem rbp aufrufen, der niedriger ist als der lbp der Potenzierung

```
op_token_pow = {  
  lbp: 30,  
  infix: left => {  
    left ** expression(30 - 1) // RECURSIVE CALL  
  },  
}
```

- Wenn expression zum nächsten ^ in seiner Schleife gelangt, stellt er fest, dass noch `rbp < tokens.current().lbp` ist und gibt das Ergebnis nicht sofort zurück, sondern sammelt zunächst den Wert des Unterausdrucks.

- Shunting Yard, TDOP und Precedence Climbing sind im wesentlichen der gleiche Algorithmus:
- Im Gegensatz zum klassischen RD ist das Hinzufügen/Ändern von Operatoren einfach
 - RD: Hinzufügen/Ändern von Funktionen im Parser
 - SY/TDOP/PC: Daten liegen in Tabellenform vor
- Mischformen möglich (siehe Shunting Yard in Parsing Expressions by Recursive Descent)
- Shunting Yard verwendet einen Stack anstatt Rekursion
- Precedence Climbing wird am häufigsten eingesetzt

Vergleich TDOP vs. Precedence Climbing

- TDOP(Pratt Parsing) vs. Precedence Climbing
 - Pratt Parsing and Precedence Climbing Are the Same Algorithm
 - From Precedence Climbing to Pratt Parsing (Norvell)
 - Eine While Schleife mit rekursiven Aufruf mit Abbruchbedingung (binding Power/precedence)
 - Rechts Assoziativität
 - In precedence climbing: $\text{next_min_prec} = \text{prec} + 1$ für left Assoziativität
 - In Pratt Parsing: rechte binding power rbp auf lbp-1 gesetzt und der rekursive Aufruf mit ihr durchgeführt
 - Klammern
 - In precedence climbing in der rekursiven Parsing Funktion behandelt
 - In Pratt Parsing können sie als nud-Funktion für das Token (behandelt werden

- Haskell
 - benutzerdefinierte Infix-Operatoren mit individueller Assoziativität und Vorrang-Regeln
 - Ein Beispiel für das Konvertieren der Operator während der Laufzeit
- Raku
 - im Verbund mit zwei weiteren Parsern (Speed-up beim Parsen von arithmetischen Ausdrücken)
- Taschenrechner
 - Umwandlung Infix-Notation (menschenslesbar) in umgekehrte polnische Notation (optimiert für Auswertung)

Parser Kombinatoren

- High-order Funktion
 - Funktionen als Parameter
 - Funktion als Rückgabewert
- Verwendet mehrere Parser als Input und gibt den Output des kombinierten Parser zurück:
 - Parse Tree
 - Index der Stelle im String die zum Stoppen des Parsers geführt hat
- Die Output der verwendeten Parser:
 - Success: {result, restString}
 - Failure: Error Message und Position

Simple Parser

Simple Parser, die nachher als Inputparameter für einen Kombinierten Parser verwendet werden:

```
fn integer(input) {  
  match = /^d+/.applyTo(input)  
  if (match) {  
    matchedText = match.first  
    return Ok({data: matchedText, rest: input.skip(len(matchedText))})  
  }  
  return Err({dataDesc: "an integer", input: input})  
}
```

Kombinierter Parser

Der kombinierte Parser sieht wie folgt aus:

```
fn apply(func, parsers) {  
  return (input) => {  
    accData = []  
    currentInput = input  
    parsers.each((parser) => {  
      result = parser(currentInput)  
      case result {  
        Err: return result  
        Ok(data, rest):  
          accData.append(data)  
          currentInput = rest  
      }  
    })  
    return Ok({data: func(accData), rest: currentInput})  
  }  
}
```

Kombinierte Parser definieren

Der kombinierte Parser kann nun so definiert werden:

```
fn plusExpr(input) {  
  // `_` because data of plus/eof is not needed for calculation  
  return apply((num1, _, num2, _) => { num1 + num2 }, [  
    integer,  
    plus,  
    integer,  
    eof  
  ])  
}
```

Diese Zusammensetzung der Parser überprüft eine Plus-Expression mit Integern. Wichtig hierbei ist die richtige Reihenfolge der Parser.

Verwendung des Parsers

Nun muss noch eine Parse-Funktion geschrieben werden, um die kombinierten Parser auszuführen.

```
fn parse(parser, input) {  
  result = parser(input)  
  case result {  
    Err(dataDesc, input):  
      raise "Parse error!  
        expected: '${dataDesc}'  
        got: '${input}'  
        "  
    Ok:  
      return result  
  }  
}
```

Ausführung des Kombinierten Parsers

Führt man nun den Parser aus, kann es wie folgt aussehen:

```
# parse(plusExpr, "12+34")  
-> {data: 46, rest: ""}  
  
# parse(plusExpr, "12+34rest")  
-> Uncaught: Parse error!  
    expected: 'end of input'  
    got: 'rest'
```

LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.