ANTLR

(ANother Tool for Language Recognation)

1. Allgemeines / Geschichte (Chris)
2. Options (Phil)
3. Lexer / Parser (Ingmar)
4. Java Runtime Model (Chris)
5. Treeparsing (Phil)
6. Fragen

Allgemeines

- http://www.antlr.org
- Parsergenerator von PCCTS
  - Purdue Compiler Construction Tool Set
- Generierung von rekursiven Parsern
- Vorteile
  - EBNF – orientiert
  - hohe Flexibilität des Parsers
  - anschauliche Umsetzung
- LL (k) – Verfahren
- Generierung von Syntaxbäumen

Geschichte

- 1988 startete PCCTS als Parsergenerierungs – Projekt
- anfangs DFA - basiert
- bis 1990 ANTLR zweimal komplett neu geschrieben
- seit 1990 LL(k)
- danach Kleinigkeiten verbessert

ANTLR Optionen

Von „**a**nalyzerDebug“ bis „**w**arnWhenFollowAmbig“

Einleitung

- ANTLR bietet verschiedenste Optionen, wie
  - Dateioptionen
  - Grammatikoptionen
  - Optionen für Regeln
  - sowie Kommandozeilenoptionen

Datei, Grammatik und Regel Optionen

- Optionen können direkt in den Quellcode geschrieben werden

- allgemeiner Aufbau

```
options {
k = 2 ;
defaultErrorHandler = false ;
}
```

- wo können Optionen stehen:
  - direkt unter dem Header in einer normalen .g-Datei
  - in der Grammatik direkt hinter der Defintion des Parsers
  - in einer Regel an folgender Stelle:

```
myrule[args] returns [retval] options { defaultErrorHandler=false; }
: // body of rule... ;
```

Datei, Grammatik und Regel Optionen (Auswahl)

- **`language = „LanguageName“ ;`**
  - legt zu erzeugende Sprache fest
  - möglich Werte sind:
    - Java (Java)
    - Cpp (C++)
    - Csharp (C#)
- **`k = „NumberOfLookahead“ ;`**
  - bestimmt die Anzahl der Lookaheadsymbole
  - d.h. maximale Anzahl von Token die bei Alternativen durchsucht werden
  - außerdem wird nach Ausstiegsbedingungen bei EBNF Konstrukten gesucht
- **`testLiterals = false ;`**
  - unterdrückt automatische Änderung des Tokentyps
- **`buildAST = true ;`**
  - stellt das Interface für Treeparsing und Treewalking zur Verfügung

Datei, Grammatik und Regel Optionen (Auswahl)

- **`analyzerDebug`**
  - Debugginginformationen während Grammatikanalyse
- **`caseSensitive`**
  - Unterdrückt Unterscheidung von Groß- und Kleinschreibung
- **`caseSensitiveLiterals`**
  - ignoriert Groß- und Kleinschreibung beim Vergleich von Token und Litralen
- **`warnWhenFollowAmbig`**
  - liefert Exception bei leeren Alternativen

Kommandozeilenoptionen (Auswahl)

- **`-o outputDir`**
  - legt den Ordner für die Ausgabedateien fest
- **`-debug`**
  - öffnet den **ParseView** Debugger
  - ParseView ist separat erhältlich
- **`-html`**
  - generiert eine HTML - Datei der Grammatik
  - Funktion ist nur für den Parser verfügbar

Kommandozeilenoptionen (Auswahl)

- **`- docbook`**
  - wie –html, nur das eine SGML – Datei erzeugt wird
- **`-diagnostics`**
  - erzeugt eine Textdatei der Grammatik mit Debugging Informationen
- **`-h | -help | --help`**
  - Hilfe
- **`-trace | -traceLexer | -traceParser | -traceTreeParser`**
  - Ausgabe des Trace für alle oder entsprechenden Teil

PARSER & LEXER

But...we've always used automata for lexical analysis!

LEXER & PARSER

- Kein DFA LEXER
- beide verwenden LL(k) mit k beliebig aber fest

DFA vs. hand build scanner

- Vorteile DFA
  - einfach aus regulären Ausdrücken herzustellen
  - kommen mit Linksrekursion klar:

```
integer     : "[0-9]+" ;

real        : "[0-9]+{.[0-9]*}|.[0-9]+" ;
```

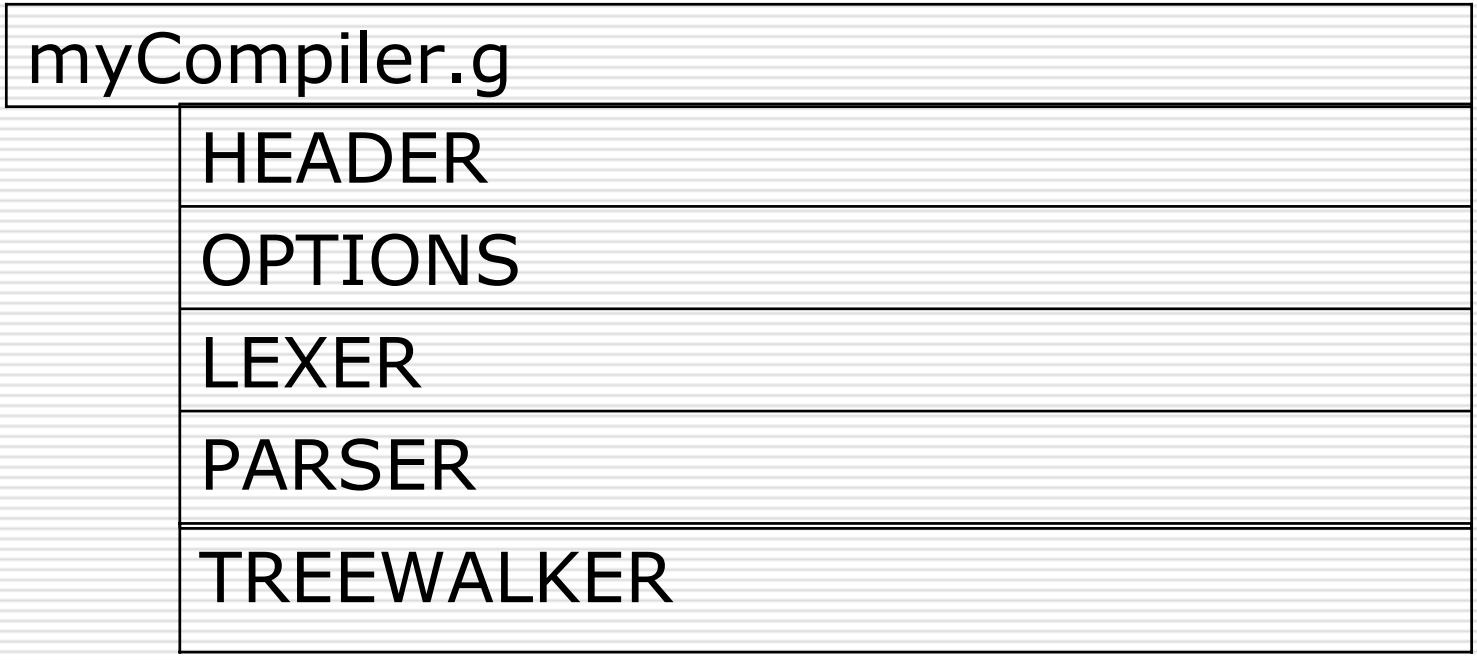
DFA vs. hand build scanner

- Vorteile hand build scanner (hbs)
  - nicht beschränkt auf die Klasse der regulären Sprachen
  - semantische Aktionen sind möglich
  - DFA´s bestehen aus Integertabellen (schwer zu debuggen)
  - guter hbs ist schneller als DFA
  - kommt mit UNICODE (16bit) klar

DFA vs. hand build scanner

- ANTLR macht beides
- reguläre Ausdrücke → gut zu verstehender LEXER, der mächtiger ist, UNICODE versteht und leicht zu debuggen ist
- PROBLEME:
  - UNICODE noch nicht vollständig implementiert
  - Linksfaktorisierung

Aufbau



- Reihenfolge egal
- *.g ist Konvention

HEADER

- Aufbau:

```
header {
...übernommener code in compiler...
}
```

- muss Zielcode entsprechen
- guter Bereich für eigentlichen Compiler der Lexer, Parser… nutzt

LEXER

- Aufbau:

```
{ ...optionaler Zielcode... }
class MyLexer extends Lexer;
options {
    ...Optionen...
}
tokens {
    ...optionale Tokendefinitionen...
}
{ ...optionaler klasseninterner Zielcode... }
Regeln
```

PARSER

- Aufbau:

```
{ ...optionaler Zielcode... }
class MyParser extends Parser;
options {
    ...Optionen...
}
tokens {
    ...optionale Tokendefinitionen...
}
{ ...optionaler klasseninterner Zielcode... }
Regeln
```

REGELN

- Tokensektion:

```
tokens {
    KEYWORD_VOID="void";
```

  - Definition von Schlüsselwörtern

```
    EXPR;
```

  - Definition von imaginären Token

```
}
```

LEXER & PARSER

- string

```
"for" == 'f' 'o' 'r'
```

- end of file

```
EOF
```

- Zeichen auslassen:

```
WS :  ( ' ' | '\t'
      | '\n'| '\r' )+
      { $setType(Token.SKIP); };
```

LEXER & PARSER

- Aufbau:

```
class MyParser extends Parser;
idList : ( ID )+;
```

  - beginnt mit Kleinbuchstabe

```
class MyLexer extends Lexer;
ID : ( 'a'..'z' )+ ;
```

  - beginnt mit Großbuchstabe

REGELN

- Aufbau:

```
Regelname:
    Alternative_1
     |    Alternative_2
    ...
     |    Alternative_n
     ; mit exception:
```

- mit exception:

```
Regelname throws MyException: A;
```

- mit init action:

```
Regelname { init-action}:    ...;
```

- interne token:

```
protected rulename: … ;
```

REGELN

- versteht EBNF:

  (...) Unterregel
  (...) * beliebig oft
  (...) + mindestens 1 mal
  (...) ? 1 oder 0 mal
  | or

- weitere Symbole:

  .. Bereich (z.B. ('a'..'z'))
  ~ not (z.B. (~'\n')*)
  . wildcard

REGELN

- weitere Symbole:

{...}	semantische Aktionen
{...}?	semantische Bedingung
(...)=>	syntaktische Bedingung
[...]	Regelparameter (später)

REGELN

- semantische Aktionen:

```
Regelname:
(   {init-action}:
                {action of 1st production}
        production_1
   |    {action of 2nd production}
        production_2
)?;
```

REGELN

- semantische Bedingung:

```
Regelname:
    { expr }? nur_wenn_expr_true;
```

- muss 'boolean' in Java zurückgeben oder 'bool' in C++

REGELN

- syntaktische Bedingungen:

```
Regelname:
        ( list "=" )=> list "=" list
        |    list;
```

- 'list' beliebig oft wiederholt
- selektives Backtracking
- implementiert durch Exceptions

REGELN

- lokal lookahead:

```
COMMENT:
    "/*"
    (
        { LA(2) != '/' }? '*'
        |   ~('*')
    )*
    "*/";
```

REGELN

- GREEDY Schleifen

```
stat:       "if" expr "then" stat
            (
                  "else" stat
            )?;
```

liefert:

```
    warning: nondeterminism between alts 1 and 2 of  block
upon
    k==1:"else"
```

- LÖSUNG:

```
stat:       "if" expr "then" stat
            (
                  options {greedy=true;} : "else" stat
            )?;
```

REGELN

- NONGREEDY Schleifen

```
CURLY_BLOCK_SCARF:   '{' (.)* '}';
```

matched alles mit '(.)*' auch abschließende '}'

- LÖSUNG:

```
CURLY_BLOCK_SCAR:
   '{'
   ( options { greedy=false; }: . )*
   '}';
```

REGELN

- anderes Beispiel (C-Kommentare):

```
class L extends Lexer;
options {
        k=2;
}
CMT: "/*"
        (options {greedy=false;}: . )*
      "*/";
```

REGELN

- gebräuchliche Vorzeichen:

```
class MyLexer extends Lexer;
options {
   k=4;
}
GT                          : ">";
GE                          : ">=";
RSHIFT                      : ">>";
RSHIFT_ASSIGN                   : ">>=";
UNSIGNED_RSHIFT             : ">>>";
UNSIGNED_RSHIFT_ASSIGN      : ">>>=";
```

- gelöst durch Lookaheadanzahl

REGELN

- Textmanipulation von TOKEN:

```
BR :   '<'! "br" '>'! ;
```

  - vernachlässigt '<' und '>'

```
REGEL! : ... ;
```

  - Text der Regel ist leer

```
REGEL : ... |! ...;
```

  - Text der Regel bei bestimmter Alternative leer

REGELN

- Textmanipulation von TOKEN:

`$append(x)`

- hänge x an Text von TOKEN heran

`$setText(x)`

- setze Text von Regel auf x

`$getText`

- liefert Text von Regel

`$setToken(x)`

- setzt Tokenobject auf x

`$setType(x)`

- setzt Typ des Token auf x

REGELN

- TOKEN labeln:

```
INDEX:
   '[' i:INT ']'
   {System.out.println(i.getText());};
INT:
   ('0'..'9')+ ;
```

REGELN

- Filtern:

```
class T extends Lexer;
options {
   k=2;
   filter=true;
}
P : "<p>" ;
BR: "<br>" ;
```

  - nur
 und <p> weitergeleitet. Rest wird ignoriert

REGELN

- Filtern:

```
class T extends Lexer;
options {
   k=2;
   filter = true;
}
P : "<p>" ;
BR: "<br>" ;
TABLE : "<table" (WS)? (ATTRIBUTE)* (WS)? '>' ;
WS : ' ' | '\t' | '\n' ;
ATTRIBUTE : ... ;
```

  - "<table 8 = width ;>" würde gefiltert werden, da nicht richtig

REGELN

- LÖSUNG:

```
setCommitToPath(boolean commit)

TABLE:
   "<table" (WS)?
   {setCommitToPath(true);}
   (ATTRIBUTE)* (WS)? '>';
```

REGELN

- Filtern bestimmter Regeln:

```
class T extends Lexer;
options {
   k=2;
   filter = P;
}
P : "<p>" ;
...
```

Java Runtime Model

Java Runtime Model

- Regeln in file.g
  - **`class MyParser extends Parser;`**

    `options {…}`

    … ***`Regeln`*** …
  - **`class MyLexer extends Lexer;`**

    `options {…}`

    … ***`Regeln`*** …
  - **`class MyTreeParser extends TreeParser;`**

    `options {…}`

    … ***`Regeln`*** …

Java Runtime Model

- erzeugte Dateien:
  - MyParser.java
  - MyLexer.java
  - MyTokenTypes.java
  - MyTokenTypes.txt
  - (MyTreeParser.java)

MyParser.java / Bsp 1

- Regeldefinition in myCompiler.g
  - type :

```
KW_BOOLEAN
| KW_FLOAT
| KW_INT
| KW_VOID ;
```

MyParser.java / Bsp1

```
public final void type() throws RecognitionException,
TokenStreamException {
    try {       //für Fehlerbehandlung
            switch ( LA(1)) {
            case KW_BOOLEAN:
                    {
                    match(KW_BOOLEAN);
                    break;
                    }
                    ...
                    default:{
                    throw new NoViableAltException(LT(1),
getFilename());
                    }//ende default
            } //ende switch
    }catch …
}// ende void type
```

MyParser.java / Bsp 2

- Regeldefinition in myCompiler.g
  - ```
    parameterlist :
    type id ( "," type id )* ;
    ```
    ```

MyParser.java / Bsp 2

```
public final void parameterlist() throws
    RecognitionException, TokenStreamException {

    try {
        type();
        id();
        _loop70:
        do {
                if ((LA(1)==45)) {
                        match(45);
                        type();
                        id();
                }else {break _loop70;}
        } while (true);
    }catch (//…) {//Fehlerbehandlung…}
}
```

MyParser.java

- Zusammenfassung:
  - jede Regel hat Methode
  - Probe mit **`if`** oder **`switch`** möglich:
    - Token als Integer-Konstanten
    - **`import MyLexerTokenTypes`**
  - lokale Fehlerbehandlung

MyLexer.java / Bsp 1

- Tokendefinition in myCompiler.g
  - `PLUS:  "+";`
  - `MINUS:  "-";`

MyLexer.java / Bsp 1

```
public Token nextToken() throws TokenStreamException {
        Token theRetToken=null;
    tryAgain:
        for (;;) {
                Token _token = null;
                int _ttype = Token.INVALID_TYPE;
                resetText();
                switch ( LA(1)) {
                        case '+':
                                mPLUS(true);
                                theRetToken=_returnToken;
                                break;
                        case '-':
                                mMINUS(true);
                                theRetToken=_returnToken;
                                break;
                        …
```

MyLexer.java / Bsp 2

- Tokendefinition in myCompiler.g
  - **LETTER:**

```
(('a'..'z') | ('A'..'Z'));
```

MyLexer.java / Bsp 2

```
case 'A': case 'B': case 'C': case 'D': case 'E': case 'F': case 'G':
case 'H': case 'I': case 'J': case 'K': case 'L': case 'M': case 'N':
case 'O': case 'P': case 'Q': case 'R': case 'S': case 'T': case 'U':
case 'V': case 'W': case 'X': case 'Y': case 'Z': case 'a': case 'b':
case 'c': case 'd': case 'e': case 'f': case 'g': case 'h': case 'i':
case 'j': case 'k': case 'l': case 'm': case 'n': case 'o': case 'p':
case 'q': case 'r': case 's': case 't': case 'u': case 'v': case 'w':
case 'x': case 'y': case 'z':

    mID(true);
    theRetToken=_returnToken;
    break;
default:
    if ((LA(1)=='=') && (LA(2)=='=')) {
            mEQU(true);
            theRetToken=_returnToken;
    }
```

MyLexer.java / Methoden

```
public final void mPLUS(boolean _createToken) throws … {
        int _ttype;
        Token _token=null;
        int _begin=text.length();
        _ttype = PLUS;
        int _saveIndex;
        match("+");
        if ( _createToken && _token==null &&_ttype!=Token.SKIP ) {
                _token = makeToken(_ttype);
                _token.setText(new String(text.getBuffer(), _begin,
    text.length()-_begin));
        }

        _returnToken = _token;
}
```

MyLexer.java

- Zusammenfassung
  - ***implements LexerTokenTypes***
  - jeder Token – Typ eigene Methode
  - **`nextToken();`**

MyTokenTypes.java

```
public interface c0LexerTokenTypes {
   int EOF = 1;
   int NULL_TREE_LOOKAHEAD = 3;
   int KW_BOOLEAN = 4;
   int KW_DO = 5;
   int KW_ELSE = 6;
   …
}
```

Tree Parsing mit ANTLR

Von Bäumen und Waldläufern

Inhalt

Einleitung

- Was ist ein Treeparser
- Notation
- Grammatikregeln
- Transformation und Operatoren
- Beispiele
  - einfach
  - Rückgabewert
  - Patternmatching

- Die letzte Seite

Einleitung

- ANTLR bietet Hilfsmittel um abstrakte Syntaxbäume (ASTs) zu konstruieren
- Erweiterung der Grammatik notwendig
  - Hinzufügen von Baumoperatoren
  - Regeln neu definieren
  - zusätzliche Aktionen ergänzen
- Veränderung der grammatikalen Struktur der ASTs ermöglicht einfaches „Treewalking“ während der Übersetzung

Was ist ein Tree Parser

- Parsing dient der Ermittlung einer grammatikalen Struktur
- ANTLR sieht Baum als Eingabe von Knoten in zwei Dimensionen
- Unterschied zwischen „normalem“ Tokenparsing und Treeparsing:
  - Testen nach dem Lookahead
  - neuen Regelmethoden definiert
  - zweidimensionale Baumstruktur
- ANTLR kann alle Bäume parsen, die das AST Interface implementieren
- Die beiden wichtigsten Funktionen sind dabei
  - **`getFirstChild`** – liefert eine Referenz auf das erste Kind
  - **`getNextSibling`** – liefert eine Referenz auf das nächste Kind

Notation

- Aufbau eines Knotens
  - besteht aus einer Liste mit Kindern
  - mit „etwas“ Text (z.B. Wert des Variableninhaltes)
  - und einem Knotentyp

- Somit ist jeder Knoten eines Baumes auch ein Baum

- Notation wie in LISP
  - Einfach: `#(A B C)`
    - Baum mit Wurzel A und Kindern B und C

Notation

- Aufbau eines Knotens
  - besteht aus einer Liste mit Kindern
  - mit „etwas“ Text (z.B. Wert des Variableninhaltes)
  - und einem Knotentyp

- Somit ist jeder Knoten eines Baumes auch ein Baum

- Notation wie in LISP
  - Einfach: `#(A B C)`
    - Baum mit Wurzel A und Kindern B und C
  - Verschachtelung: `#(A B #(C D E))`

Notation

- Aufbau eines Knotens
  - besteht aus einer Liste mit Kindern
  - mit „etwas“ Text (z.B. Wert des Variableninhaltes)
  - und einem Knotentyp

- Somit ist jeder Knoten eines Baumes auch ein Baum

- Notation wie in LISP
  - Einfach: #(A B C)
    - Baum mit Wurzel A und Kindern B und C
  - Verschachtelung: #(A B #(C D E))
    - Baum mit Wurzel A, einem Kind B und einem Unterbaum mit C als Wurzel und D sowie E als Kindern

Die AST Definition

```
public interface AST {

/** Get the token type for this node */
public int getType();

/** Set the token type for this node */
public void setType(int ttype);

/** Get the token text for this node */
public String getText();

/**Set the token text for this node */
public void setText(String text);

/** Get the first child of this node; null if no children */
public AST getFirstChild();

/** Set the first child of a node. */
public void setFirstChild(AST c);

/** Get the next sibling in line after this one */
public AST getNextSibling();

/** Set the next sibling after this one. */
public void setNextSibling(AST n);

/**Add a (rightmost) child to this node */
public void addChild(AST c);

}
```

Grammatikregeln für Bäume

- Baumgrammatiken Sammlung von EBNF Regeln
- mit Aktionen sowie semantischen und syntaktischen Prädikaten
- ```
  rule:   alternative1
      | alternative2
      | ...
      | alternativen ;
  ```
- jede alternative Produktion ist zusammengesetzt aus Elementliste:
  - `# (root-token child1 child2 ... child n)`
- Beispiel: Additionsbaum mit zwei Integer Kindern:
  ```

Grammatikregeln für Bäume

- Baumgrammatiken Sammlung von EBNF Regeln
- mit Aktionen sowie semantischen und syntaktischen Prädikaten
- ```
  rule:   alternative1
      | alternative2
      | ...
      | alternativen ;
  ```
- jede alternative Produktion ist zusammengesetzt aus Elementliste:
  - `# (root-token child1 child2 ... child n)`
- Beispiel: Additionsbaum mit zwei Integer Kindern:
  - `# (PLUS INT INT)`
  ```

Grammatikregeln für Bäume

- **man beachte:**
  - **Wurzel eines Baumpatterns muss eine Tokenreferenz sein**
  - **Kinder dürfen auch „Subrules" sein**
- **Beispiel „If then else" Statement – „else" optional**
  - `# (IF expr stat (stat)?)`
- **Wichtig für Baumgrammatiken und Strukturen:**
  - **keine exakten Matches**
  - **ausreichende Matches genügen**
- **Übereinstimmung wird geliefert auch wenn Teile noch ungeparst sind**
  - **Bsp.:** `# (A B)` **# paßt zu allen Bäume gleicher Struktur,**
    wie `# (A # (B C) D)`

Syntaktische Prädikate

- Treeparsing benutzt Lookahead von k = 1
- Möglichkeit unterschiedliche Baumstrukturen zu beschreiben

- Ziel: Beschränkung durch den fixen Lookahead umgehen

- Lösung: Syntaktische Prädikate

- Beispiel: Unterscheidung des unären und binären Minusoperators:
  - ```
    expr:  ( # (MINUS expr expr) ) => # (MINUS expr expr)
         | # (MINUS expr)
          ... ;
    ```

- Reihenfolge beachten, da zweite Alternative Teilmenge der ersten ist
    ```

Baumschule - Transformation

- ANTLR Tree Parse unterstützt buildAST Option
- ohne Codetransformation wird Eingabebaum zum Ausgabebaum
- jede Regel hat implizit einen (automatisch definierten) Ergebnisbaum
- getAST liefert den Ergebnisbaum der Startregel

Der "!" Operator

- Regeln können um Suffix "!" erweitert werden
  - unterdrückt automatisches Hinzufügen zum Ergebnisbaum
  - interne Verarbeitung findet jedoch trotzdem statt
- Kann mit Regelreferenzen und Regeldefinitionen verwendet werden
  - Beispiel:
  - `begin!  :  INT PLUS i:INT { #begin = #(PLUS INT i); } ;`
- "!" kann auch als Präfixoperator verwendet werden
  - Funktionsweise wie oben
  - Filtert jedoch nur die entsprechende Alternative

Der "^" Operator

- Blätter
  - jeder Tokenreferenz sowie jedem Tokenbereich wird ein Blatt zugeordnet
  - ohne Suffixe wird Tokenliste erzeugt und der Baum kopiert

- Wurzeln
  - jeder Token mit dem Suffix "^" wird als Wurzelknoten behandelt
  - Sinn: Token ist Operatorknoten und gleichzeitig auch Wurzel eines neuen Teilbaums
  - Beispiel:
    - Eingabe: `a : A B^ C^ ;`
    - Ergebnisbaum: ??

Der "^" Operator

- Blätter
  - jeder Tokenreferenz sowie jedem Tokenbereich wird ein Blatt zugeordnet
  - ohne Suffixe wird Tokenliste erzeugt und der Baum kopiert
- Wurzeln
  - jeder Token mit dem Suffix "^" wird als Wurzelknoten behandelt
  - Sinn: Token ist Operatorknoten und gleichzeitig auch Wurzel eines neuen Teilbaums
  - Beispiel:
    - Eingabe: `a : A B^ C^ ;`
    - Ergebnisbaum: `#(C #(B A))`
    - Ergebnisbaum ohne "^": Liste A B C

Beispiel

- einfacher Taschenrechner mit folgender Grammatik

```
class CalcLexer extends Lexer;
WS :        (' ' | '\t' | '\n' | '\r') { _ttype = Token.SKIP; } ;
LPAREN :    '(' ;       RPAREN :    ')' ;
STAR:       '*' ;       PLUS:       '+' ;
SEMI:       ';' ;       INT :       ('0'..'9')+ ;
```

```
class CalcParser extends Parser;
options {
            buildAST = true;        // uses CommonAST by default
}
expr :      mexpr (PLUS^ mexpr)* SEMI! ;
mexpr :     atom (STAR^ atom)* ;
atom: INT ;
```

- Beispiel: Eingabe 3 * 4 + 5
  - erzeugt folgenden Baum:

Beispiel

- einfacher Taschenrechner mit folgender Grammatik

```
class CalcLexer extends Lexer;
WS :        (' ' | '\t' | '\n' | '\r') { _ttype = Token.SKIP; } ;
LPAREN :    '(' ;       RPAREN :    ')' ;
STAR:       '*' ;       PLUS:       '+' ;
SEMI:       ';' ;       INT :       ('0'..'9')+ ;
```

```
class CalcParser extends Parser;
options {
            buildAST = true;        // uses CommonAST by default
}
expr :      mexpr (PLUS^ mexpr)* SEMI! ;
mexpr :     atom (STAR^ atom)* ;
atom: INT ;
```

- Beispiel: Eingabe 3 * 4 + 5
  - erzeugt folgenden Baum: `# ( + ( * 3 4 ) 5 )`

Präzedenz

- Präzedenz von Operatoren wird nicht mit angegeben
- wird komplett über die Baumstruktur kodiert.
  - Eingabesymbole sind in Knoten verstaut
  - die Struktur jedoch in der Beziehung der Knoten untereinander

Beispiel – mit Rückgabewert

- Struktur muss rekursiv beschrieben werden:

```
class CalcTreeWalker extends TreeParser;

expr :      # (PLUS expr expr)
          | # (STAR expr expr)
          | INT ;
```

```
class CalcTreeWalker extends TreeParser;

expr returns [int r]
{
          int a,b;
          r=0;
} :
             # (PLUS a = expr b = expr) {r = a + b ;}
          | # (STAR a = expr b =e xpr)  {r = a * b ;}
          | i : INT                     {r = Integer.parseInt(i.getText()); }
;
```

Anbindung an Treeparser und Treewalker

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Calc {
  public static void main(String[] args) {
     try {
           CalcLexer lexer         = new CalcLexer(new DataInputStream(System.in));
           CalcParser parser       = new CalcParser(lexer);

           // Parse the input expression
           parser.expr();

           CommonAST t             = (CommonAST)parser.getAST();

           // Print the resulting tree out in LISP notation
           System.out.println(t.toStringList());

           CalcTreeWalker walker   = new CalcTreeWalker();

           // Traverse the tree created by the parser
           int r = walker.expr(t);
           System.out.println("value is "+r);
        } catch(Exception e) { System.err.println("exception: "+e); }
     }
}
```

Beispiel Patternmatching

```
class CalcTreeWalker extends TreeParser;
options {
                  buildAST = true;         // "transform" mode
}

expr:!  #(PLUS left:expr right:expr)      // '!' turns off auto transform
        {
            // x + 0 = x
            if ( #right.getType() == INT && Integer.parseInt( #right.getText() ) == 0)
            {
                #expr = #left ;
            }
            // 0 + x = x
            else if ( #left.getType() == INT && Integer.parseInt( #left.getText() ) == 0)
            {
                #expr = #right ;
            }
            // x + y
            else
            {
                #expr = # (PLUS, left, right) ;
            }
        }
    |   # (STAR expr expr)                // use auto transformation
    |   i: INT
    ;
```

Erwähnenswert

- was ANTLR nicht bietet:
  - Referenzknoten, die nichts weiter tun, als auf andere Knoten zu zeigen
- weiteres Feature des ANTLR Tree Parsers:
  - Ausgabe des Baumes in XML (Drittanbieter)