Compiler I: Grundlagen

Lexer/Parser-Generierung mit ANTLR

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Überblick

- ANTLR - Another Tool for Language Recognition
- Kurze Einführung in ANTLR 3.x
  - Inkompatibel zur ANTLR 2.x!
- Inhalt basiert wieder auf Material von Theo Ruys
  - “Vertalerbouw”, Universität Twente

ANTLR - Einführung 1

Embedded Systems & Applications

ANTLR

- Eingabe: Grammatik in EBNF (und mehr!)
- Ausgabe: Erkenner für Sprache

ANTLR - Einführung 1

Embedded Systems & Applications

ANTLR

- Eingabe: Grammatik in EBNF (und mehr!)
- Ausgabe: Erkenner für Sprache

Arten von Eingabedaten

- Zeichenströme (bearbeiten mit Scanner)
- Token-Ströme (bearbeiten mit Parser)
- Knoten-Ströme (bearbeiten mit Tree Walker)

ANTLR - Einführung 1

Embedded Systems & Applications

ANTLR

- Eingabe: Grammatik in EBNF (und mehr!)
- Ausgabe: Erkenner für Sprache

Arten von Eingabedaten

- Zeichenströme (bearbeiten mit Scanner)
- Token-Ströme (bearbeiten mit Parser)
- Knoten-Ströme (bearbeiten mit Tree Walker)

ANTLR 3.x

- LL(*) Compiler Generator
  - Erzeugt gut lesbaren Code für rekursiven Abstieg
- Erzeugt Erkenner in Java, C++, C#, Python, etc.

ANTLR - Einführung 2

ANTLR erzeugt prädiktive LL(k) oder LL(*) Erkenner

- Berechnet FIRST, FOLLOW und LOOKAHEAD Mengen
- Überprüft auf syntaktische Korrektheit
- Alle erzeugten Erkenner verwenden rekursiven Abstieg
  - Keine endlichen Automaten
  - Genau das Schema aus Watt & Brown

ANTLR - Einführung 2

ANTLR erzeugt prädiktive LL(k) oder LL(*) Erkenner

- Berechnet FIRST, FOLLOW und LOOKAHEAD Mengen
- Überprüft auf syntaktische Korrektheit
- Alle erzeugten Erkenner verwenden rekursiven Abstieg
  - Keine endlichen Automaten
  - Genau das Schema aus Watt & Brown

Alternative Compiler-Generatoren

- Lexer/Scanner: lex, flex, JFlex
- Parser: yacc/bison, JCup, JavaCC, SableCC, SLADE

Weiterführendes Material

Internet

- `http://www.antlr.org`
- Dort: Wiki, Thema “FAQ und Getting Started”
- Sehr umfangreiche Materialsammlung
  - Leider unstrukturiert

Weiterführendes Material

Internet

- `http://www.antlr.org`
- Dort: Wiki, Thema “FAQ und Getting Started”
- Sehr umfangreiche Materialsammlung
  - Leider unstrukturiert

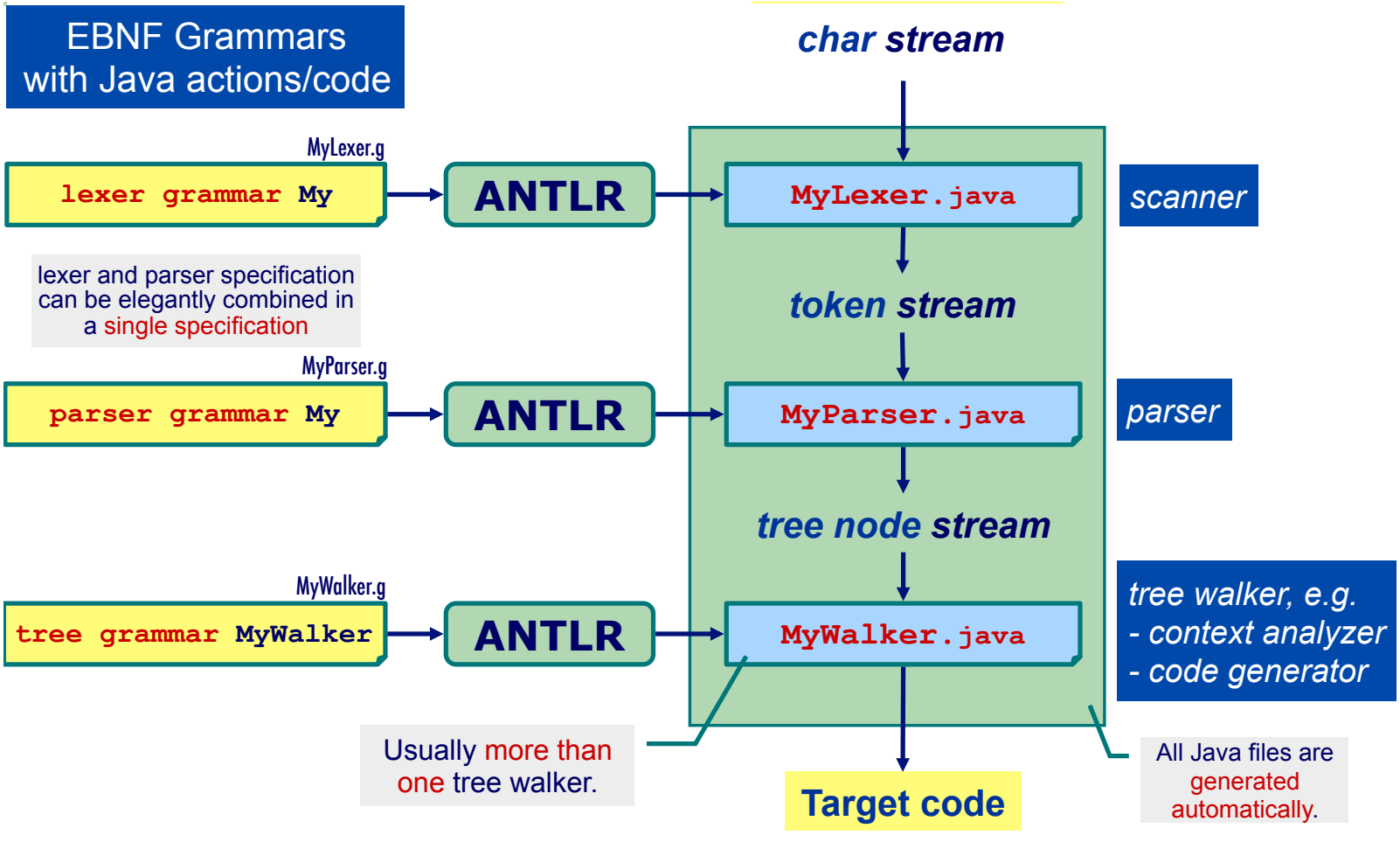
Besser: Buch *The Definitive ANTLR Reference*

- Terence Parr
- Pragmatic Bookshelf 2007
- Sehr gut lesbar!

Änderungen von ANTLR 2.x zu 3.x

- ANTLRWorks: IDE für Grammatiken und Debugging
- Unterstützung für LL(*)
- Aufbau von ASTs besser integriert (*rewrite rules*)
- Portableres Back-End (z.B. Ruby etc.)
- Bessere Fehlermeldungen und -behandlung
- Einbau von StringTemplate zur leichteren Texterzeugung
  - Sehr hilfreich für textuelle Code-Erzeugung
- Neue Syntax für Grammatiken
  - Inkompatibel zu ANTLR 2.x

Werkzeugfluß



Struktur der Eingabedatei

```
[gtype] grammar FooBar;

options {
    options for entire grammar file
}

tokens {
    token definitions
}

@header {
    will be copied to the generated Java file(s)
}

@rulecatch {
    error handling: how to deal with exceptions?
}

@members {
    optional class definitions: instance variables, methods
}

rulename : all rules for FooBar
```

gtype may be empty or **lexer**, **parser** or **tree**.

A single .g file can contain a Lexer and/or Parser, or a TreeParser.

e.g. **imports**

Aufbau einer Regel
Wird in Java-Methode umgesetzt

```
rulename [args] returns [T val]
  ( options { local options } )
    :  alternative1
    |  alternative2
    |  ...
    |  alternativen
    ;
```

optional, used for passing information around

An alternative is an EBNF regular expression containing:

- rulename
- TOKEN
- EBNF operator
- Java code in braces

+ optional code sections to insert at start of end of method

```
@init  { ... }
@after { ... }
```

EBNF operators

A|B	A or B
A*	zero or more A's
A+	one or more A's
A?	an optional A

When using EBNF operators in ANTLR: use parentheses to enclose more than one symbol..

Example

```
expr    : operand (PLUS operand)*
        ;
operand : LPAREN expr RPAREN
        | NUMBER
        ;
```

Benutzung von ANTLR

Kommandozeile

`java org.antlr.Tool` *eingabe*`.g`

Benutzung von ANTLR

Embedded Systems & Applications

Kommandozeile

`java org.antlr.Tool` *`eingabe`*`.g`

CLASSPATH muss enthalten

- antlr.jar stringtemplate.jar antlr3.jar antlr3-runtime.jar

Benutzung von ANTLR

Embedded Systems & Applications

Kommandozeile

`java org.antlr.Tool` *eingabe*`.g`

CLASSPATH muss enthalten

- antlr.jar stringtemplate.jar antlr3.jar antlr3-runtime.jar

GUIs

- ANTLRWorks (IntelliJ): **`http://www.antlr.org/works`**
- AntlrDT (Eclipse): **`http://www.certiv.net/projects/plugins/antlrdt.html`**
- ANTLR IDE (Eclipse): **`http://antlrv3ide.sourceforge.net/`**

Beispielsprache: Calc 1
Einfache Sprache für Taschenrechner

- Deklarationen
  - Nur Integer-Variablen
  - Müssen vor Anweisungen stehen
- Anweisungen
  - Zuweisung zu Variablen
  - Ausgabe von ausgewerteten Ausdrücken
- Ausdrücke
  - Zunächst nur Addition und Subtraktion

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

Beispielsprache: Calc 2

Embedded Systems & Applications

EBNF for Calc

```
program         ::=  declarations statements EOF
declarations    ::=  (declaration SEMICOLON)*
declaration     ::=  VAR IDENTIFIER COLON type
statements      ::=  (statement SEMICOLON)+
statement       ::=  assignment
                 |   printStatement
assignment      ::=  lvalue BECOMES expr
printStatement  ::=  PRINT LPAREN expr RPAREN
lvalue          ::=  IDENTIFIER
expr            ::=  operand ((PLUS | MINUS) operand)*
operand         ::=  IDENTIFIER
                 |   NUMBER
                 |   LPAREN expr RPAREN
type            ::=  INTEGER
```

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

All terminals are written as UPPERCASE symbols.

Vorgehensweise

ANTLR wird vier unterschiedliche Erkenner erzeugen

- CalcLexer (erweitert `Lexer`)
  Übersetzt Zeichenstrom in Tokenstrom

Vorgehensweise

ANTLR wird vier unterschiedliche Erkenner erzeugen

- CalcLexer (erweitert `Lexer`)
  Übersetzt Zeichenstrom in Tokenstrom
- CalcParser (erweitert `Parser`)
  Übersetzt Token-Strom in Knoten-Strom (von AST-Knoten)

Vorgehensweise

ANTLR wird vier unterschiedliche Erkenner erzeugen

- CalcLexer (erweitert `Lexer`)
  Übersetzt Zeichenstrom in Tokenstrom
- CalcParser (erweitert `Parser`)
  Übersetzt Token-Strom in Knoten-Strom (von AST-Knoten)
- CalcChecker (erweitert `TreeParser`)
  Läuft über Knoten-Strom des AST und führt kontextuelle Überprüfung durch
  ➥automatisch erzeugtes Visitor-Pattern

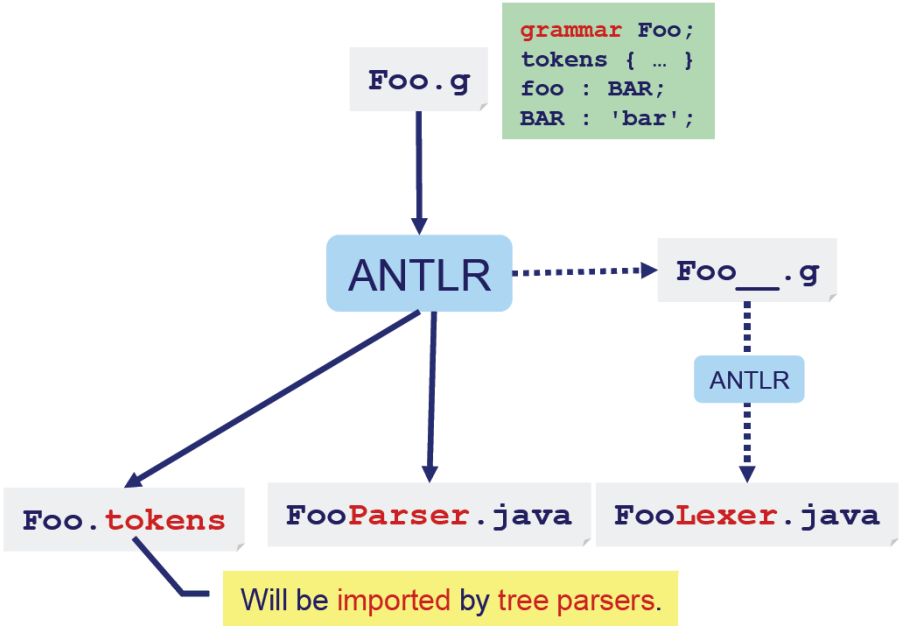
Vorgehensweise

ANTLR wird vier unterschiedliche Erkenner erzeugen

- CalcLexer (erweitert `Lexer`)
  Übersetzt Zeichenstrom in Tokenstrom
- CalcParser (erweitert `Parser`)
  Übersetzt Token-Strom in Knoten-Strom (von AST-Knoten)
- CalcChecker (erweitert `TreeParser`)
  Läuft über Knoten-Strom des AST und führt kontextuelle Überprüfung durch
  ➥automatisch erzeugtes Visitor-Pattern
- CalcInterpreter (erweitert `TreeParser`)
  Läuft über Knoten-Strom des AST und interpretiert Programm
  ➥automatisch erzeugtes Visitor-Pattern

Parser und Lexer in ANTLR

- Enge Zusammenarbeit zwischen Lexer und Parser
  - Lexer produziert Tokens aus Zeichenstrom
  - Parser konsumiert Tokens
- ANTLR 3.x erlaubt kombinieren von Lexer und Parser in einer Spezifikation



Namensregeln für Lexer und Parser

- Literale Zeichenketten sind in einfache Anführungszeichen eingeschlossen
  - Beispiele: `'foo'`, `'bar'`
- Token-Namen im Lexer beginnen immer mit Großbuchstaben
  - Beispiele: `PLUS`, `MINUS`, `Div`
- Nichtterminalsymbole im Parser beginnen immer mit einem Kleinbuchstaben
  - Beispiele: `program`, `statement`, `dSpace`

Parser und Lexer für Calc 1
Optionen und Tokens

```
grammar Calc;

options {
  k = 1;
  language = Java;
  output = AST;
}

tokens {
    PLUS         =   '+'   ;
    MINUS        =   '-'   ;
    BECOMES      =   ':='  ;
    COLON        =   ':'   ;
    SEMICOLON    =   ';'   ;
    LPAREN       =   '('   ;
    RPAREN       =   ')'   ;

    // keywords
    PROGRAM      =   'program' ;
    VAR          =   'var'     ;
    PRINT        =   'print'   ;
    INTEGER      =   'integer' ;
}
```

This is a combined specification (not prefixed by lexer, parser or tree).

amount of lookahead, disables LL(*)

Target language is Java.

build an AST

token definitions (literals)

tokens always start with an uppercase letter and specify the text for a token

Parser und Lexer für Calc 2

Parser-Produktionen für reine Erkennung, noch keine AST-Konstruktion

```
program      :   declarations statements EOF
             ;
declarations :   (declaration SEMICOLON)*
             ;
statements   :   (statement SEMICOLON)+
             ;
declaration  :   VAR IDENTIFIER COLON type
             ;
statement    :   assignment
             |   print
             ;
assignment   :   lvalue BECOMES expr
             ;
print        :   PRINT LPAREN expr RPAREN
             ;
lvalue       :   IDENTIFIER
             ;
expr         :   operand ((PLUS | MINUS) operand)*
             ;
operand      :   IDENTIFIER
             |   NUMBER
             |   LPAREN expr RPAREN
             ;
type         :   INTEGER
             ;
```

parser specific rules

special “end-of-file” token

parser rules start with a lowercase letter

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

In this example, all tokens are explicitly named (as UPPERCASE tokens). It is also possible to use literals in the parser specification.
For example:

```
print : 'print' '(' expr ')'
expr  : operand (('+' | '-') operand)*
```

Parser und Lexer für Calc 3
Lexer-Regeln

```
IDENTIFIER :   LETTER (LETTER | DIGIT)*
           ;

NUMBER     :   DIGIT+
           ;

COMMENT    :   '//' .* '\n'
                   { $channel=HIDDEN; }
           ;

WS         :   (' ' | '\t' | '\f' | '\r' | '\n')+
                   { $channel=HIDDEN; }
           ;

fragment DIGIT  :   ('0'..'9') ;

fragment LOWER  :   ('a'..'z') ;

fragment UPPER  :   ('A'..'Z') ;

fragment LETTER :   LOWER | UPPER ;
```

lexer specific rules

"`.*`" matches everything except the character that follows it (i.e. `'\n'`).

There are multiple token channels. The parser reads from the DEFAULT channel. By setting a token's channel to HIDDEN it will be ignored by the parser.

shorthand for (the complete)
`'a'|'b'|'c'| ...|'y'|'z'`

fragment lexer rules can be used by other lexer rules, but do not return tokens by themselves

No need to worry about counting the newlines; the lexer takes care of this automatically.

Parser und Lexer für Calc 4
Parser mit AST-Konstruktion

Embedded Systems & Applications

parser building the AST

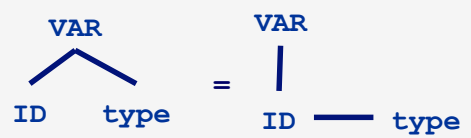
```
program     :   declarations statements EOF
                  -> ^(PROGRAM declarations statements)
            ;
declarations:   (declaration SEMICOLON!)*
            ;
statements  :   (statement SEMICOLON!)+
            ;
declaration :   VAR^ IDENTIFIER COLON! type
            ;
statement   :   assignment
            |   print
            ;
assignment  :   lvalue BECOMES^ expr
            ;
print       :   PRINT^ LPAREN! expr RPAREN!
            ;
lvalue      :   IDENTIFIER
            ;
expr        :   operand ((PLUS^ | MINUS^) operand)*
            ;
operand     :   IDENTIFIER
            |   NUMBER
            |   LPAREN! expr RPAREN!
            ;
type        :   INTEGER
            ;
```

Imaginary token that is used as the root AST node (not really needed).

Annotations for building AST nodes	
`T^`	make `T` the root of this (sub)rule
`T!`	discard `T`
`-> ^(...)`	tree construction for a rule

For example:

```
  VAR^ IDENTIFIER COLON! type
= ^(VAR IDENTIFIER type)
=
```



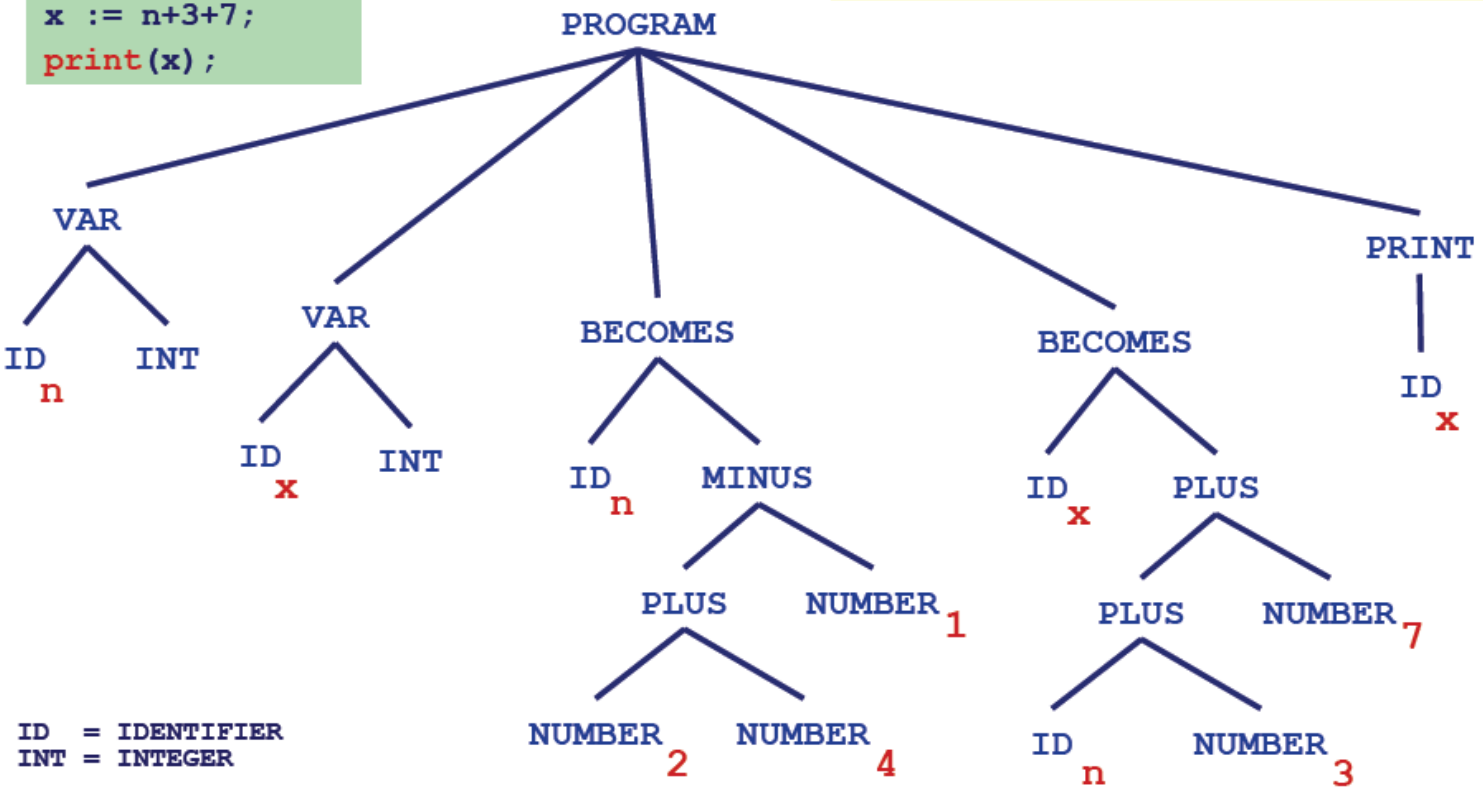
(due to rule of operand) this builds:
`^(PLUS expr expr)`

first child, next sibling notation

Konstruierter AST

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

```
program : decls stats EOF
          -> ^(PROGRAM decls stats);
decls   : (decl SEMICOLON!)*
stats   : (stat SEMICOLON!)+ ;
decl    : VAR^ ID COLON! type ;
stat    : assign | print ;
assign  : lvalue BECOMES^ expr ;
print   : PRINT^ LPAREN! expr RPAREN! ;
lvalue  : ID ;
expr    : oper ((PLUS^ | MINUS^) oper)* ;
oper    : ID | NUM | LPAREN! expr RPAREN! ;
type    : INT ;
```



ID = IDENTIFIER
INT = INTEGER

Automatische Erzeugung von Visitor

TreeWalker anhand von Baum-Grammatik, führt noch keine Aktionen aus

```
program : decls stats EOF -> ^(PROGRAM decls stats);
decls   : (decl SEMICOLON!)*
stats   : (stat SEMICOLON!)+ ;
decl    : VAR^ IDENTIFIER COLON! type ;
stat    : assign | print ;
assign  : lvalue BECOMES^ expr ;
print   : PRINT^ LPAREN! expr RPAREN! ;
lvalue  : ID ;
expr    : operand ((PLUS^ | MINUS^) operand)* ;
operand : ID | NUM | LPAREN! expr RPAREN! ;
type    : INT ;
```

```
tree grammar CalcTreeWalker;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}

program     :   ^(PROGRAM (declaration | statement)+)
            ;

declaration :   ^(VAR IDENTIFIER type)
            ;

statement   :   ^(BECOMES IDENTIFIER expr)
            |   ^(PRINT expr)
            ;

expr        :   operand
            |   ^(PLUS expr expr)
            |   ^(MINUS expr expr)
            ;

operand     :   IDENTIFIER | NUMBER ;
type        :   INTEGER ;
```

This is a specification of a tree walker.

Import tokens from `Calc.tokens`.

The AST nodes are of type `CommonTree`.

The AST has a root node `PROGRAM` with many (declaration or statement) children.

Match a tree whose root is a PLUS token with two children that match the expr rule.

This tree walker does not do anything (yet). Note the conciseness of the grammar and the correspondence with the "abstract syntax" of the language Calc.

Kontextuelle Überprüfung für Calc 1
Infrastruktur

The CalcChecker checks the context rules of the language:
- each identifier can be declared only once
- identifiers that are used must have been declared.

```
tree grammar CalcChecker;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}

@header {
import java.util.Set;
import java.util.HashSet;
}

@rulecatch {
catch (RecognitionException e) {
  throw e;
}
}

@members {
   private Set<String> idset = new HashSet<String>();
   public boolean isDeclared(String s) {
       return idset.contains(s);
   }
   public void declare(String s) {
       idset.add(s);
   }
}
```

`@header`: code block which is copied verbatim to the beginning of CalcChecker.java.

`@rulecatch`: specify your own error handler. Here: no error handler; exceptions are propagated to the method calling this checker.

`@members`: code block which is copied verbatim to the class definition of CalcChecker.java.

The Calc language uses a monolithic block structure. For checking the scope rules we can use a `Set`.

The methods `isDeclared` and `declare` become methods of the class `CalcChecker`.

Kontextuelle Überprüfung für Calc 2

Regeln für kontextuelle Einschränkungen prüfen

```
program
    :   ^(PROGRAM (declaration | statement)+)
    ;

declaration
    :   ^(VAR id=IDENTIFIER type)
        { if (isDeclared($id.getText()))
              throw new CalcException($id.getText() +
                                      " is already declared");
          else
              declare($id.getText());
        }
    ;

statement
    :   ^(BECOMES id=IDENTIFIER expr)
        { if (!isDeclared($id.text))
              throw new CalcException($id.text +
                                      " is used but not declared");
        }

    |   ^(PRINT expr)
    ;

...
```

With `name=NODE` we can refer to the AST node using `name` ...

... and get its String representation.

Within Java code, the ANTLR variables are (usually) prefixed with $.

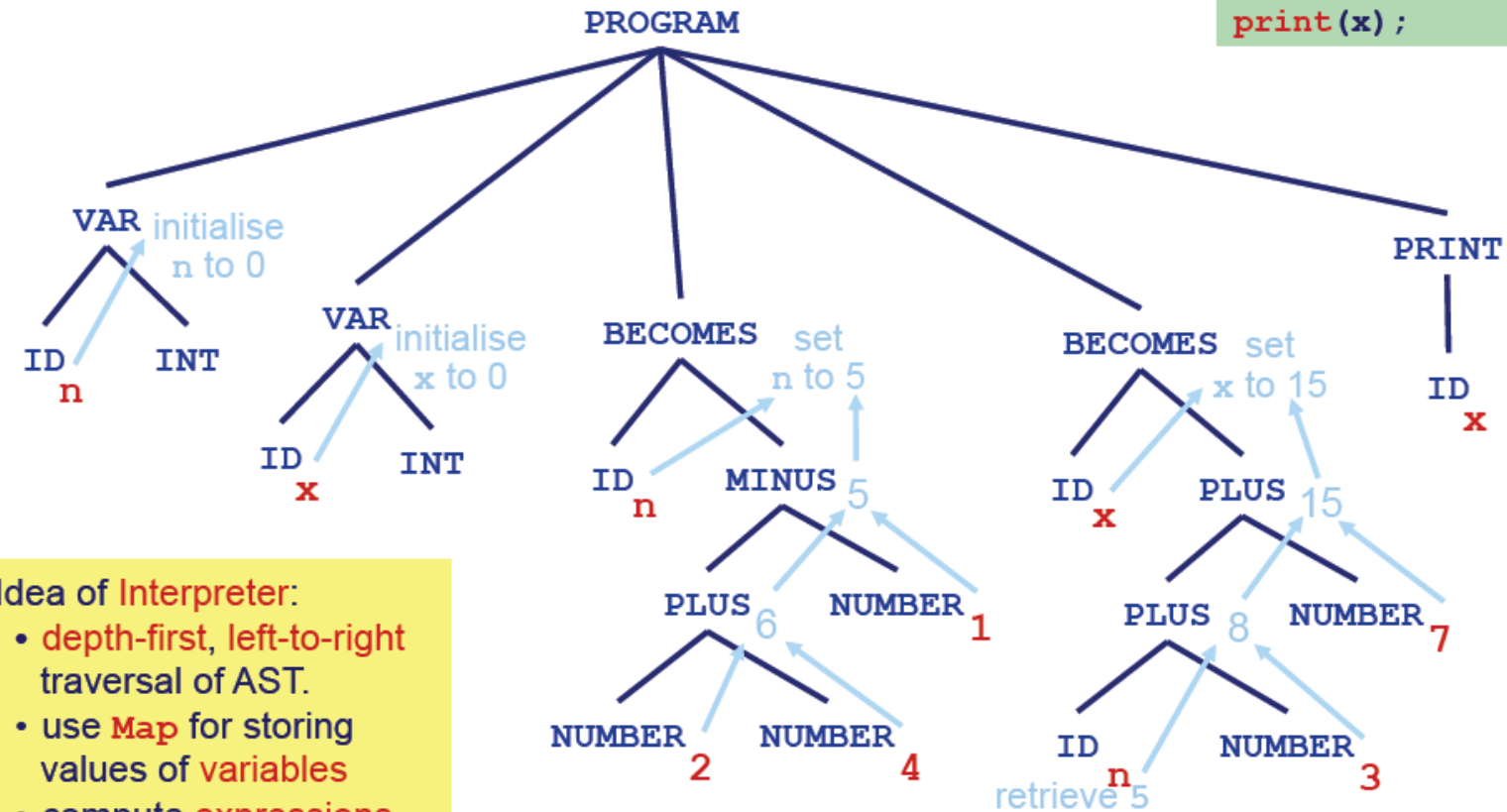
Java code block which is copied verbatim to the parse method of '`declaration`' in CalcChecker.java.

... or use the attribute `text`.

`CalcException` is an user-defined Exception (subclass of `org.antlr.runtime RecognitionException`) to express some problem in the input.

Interpreter für Calc 1
Grundsätzliche Vorgehensweise

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```



Idea of Interpreter:
- depth-first, left-to-right traversal of AST.
- use `Map` for storing values of variables
- compute expressions bottom up

Interpreter für Calc 2
Verwende weiteren TreeWalker

Idea of Interpreter:
- depth-first, left-to-right traversal of AST.
- use `Map` for storing values of variables
- compute expressions bottom up

```
tree grammar CalcInterpreter;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}

@header {
import java.util.Map;
import java.util.HashMap;
}

@members {
    Map<String,Integer> store = new HashMap<String,Integer>();
}

program      :   ^(PROGRAM (declaration | statement)+)
             ;

declaration  :   ^(VAR id=IDENTIFIER type)
                 { store.put($id.text, 0); }
             ;

...
```

To store the values of the variables.

Initialized on 0.

Interpreter für Calc 3
Verwende weiteren TreeWalker

```
statement
    :   ^(BECOMES id=IDENTIFIER v=expr)
        { store.put($id.text, $v); }

    |   ^(PRINT v=expr)
        { System.out.println("" + $v); }
    ;

expr returns [int val = 0]
    :   z=operand                   { val = z;     }
    |   ^(PLUS  x=expr y=expr)      { val = x + y; }
    |   ^(MINUS x=expr y=expr)      { val = x - y; }
    ;

operand returns [int val = 0]
    :   id=IDENTIFIER { val = store.get($id.text); }
    |   n=NUMBER      { val = Integer.parseInt($n.text); }
    ;
```

The rule expr returns a value.

The value returned by expr is put into the store for id.

ANTLR deduces from the context the types of the variables: `id` is a `CommonTree`, `v` is an `int`.

A rule can return a value: `rulename returns [T x]` The type of the return value is `T` and the value returned is the value of `x` at the end of the rule.

Note that it is also possible to pass arguments to a rule.

Get the value of `IDENTIFIER` out of the store.

Parse the string representation of the `NUMBER`.

Hauptprogramm für Interpreter
compiler driver

```
public static void main(String[] args) {

    CalcLexer  lex  = new CalcLexer(
                          new ANTLRInputStream(System.in));
    CommonTokenStream tokens = new CommonTokenStream(lex);
    CalcParser parser = new CalcParser(tokens);

    CalcParser.program_result result = parser.program();
    CommonTree tree = (CommonTree) result.getTree();

    CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);
    CalcChecker checker = new CalcChecker(nodes);
    checker.program();

    CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);
    CalcInterpreter interpreter = new CalcInterpreter(nodes);
    interpreter.program();
}
```

lexer

parser

checker

inter-preter

A lexer gets an ANTLR stream as input.

The parser gets the lexer's output tokens.

Call the start symbol to start parsing.

The recognition methods may all throw Exceptions (e.g. `RecognitionException`, `TokenStreamException`); These have to be caught in `main`-method. See `Calc.java`.

AST Visualisierung 1
Textuell und graphisch

```
public static void main(String[] args) {

    CalcLexer lexer = new CalcLexer(
                            new ANTLRInputStream(System.in));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    CalcParser parser = new CalcParser(tokens);

    CalcParser.program_return result = parser.program();
    CommonTree tree = (CommonTree) result.getTree();
    ...

    // show S-Expression respresentation of the AST
    String s = tree.toStringTree();
    System.out.println(s);

    // print the AST as DOT specification
    DOTTreeGenerator gen = new DOTTreeGenerator();
    StringTemplate st = gen.toDOT(tree);
    System.out.println(st);
}
```

-ast

-dot

.dot files can be visualized using the GraphViz program: http://www.graphviz.org/

`DOTTreeGenerator` is defined in package `org.antlr.stringtemplate`

AST Visualisierung 2

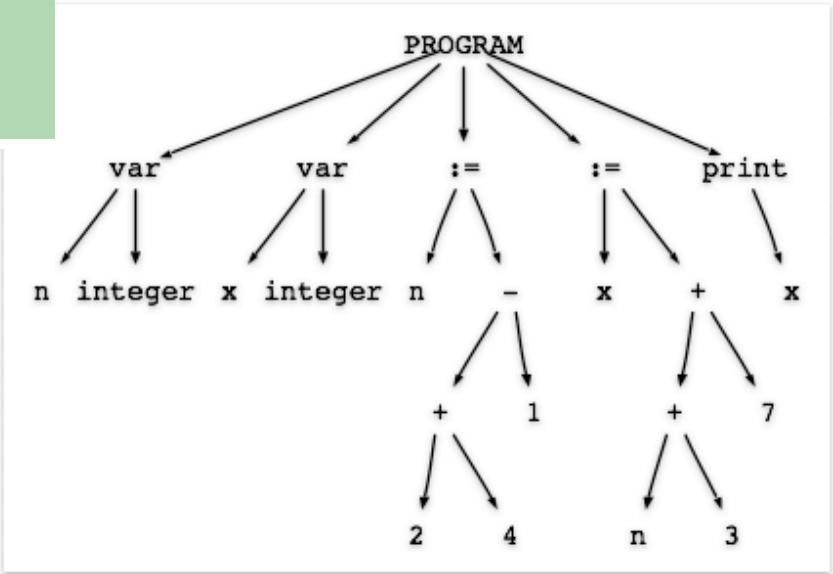
```
Terminal — bash — 100x8

[ruys@jay]$ java Calc -no_interpreter -ast < ex1.calc
(PROGRAM (var n integer) (var x integer) (:= n (- (+ 2 4) 1)) (:= x (+ (+ n 3) 7)) (print x))
[ruys@jay]$
```

via **tree.toStringTree()**

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

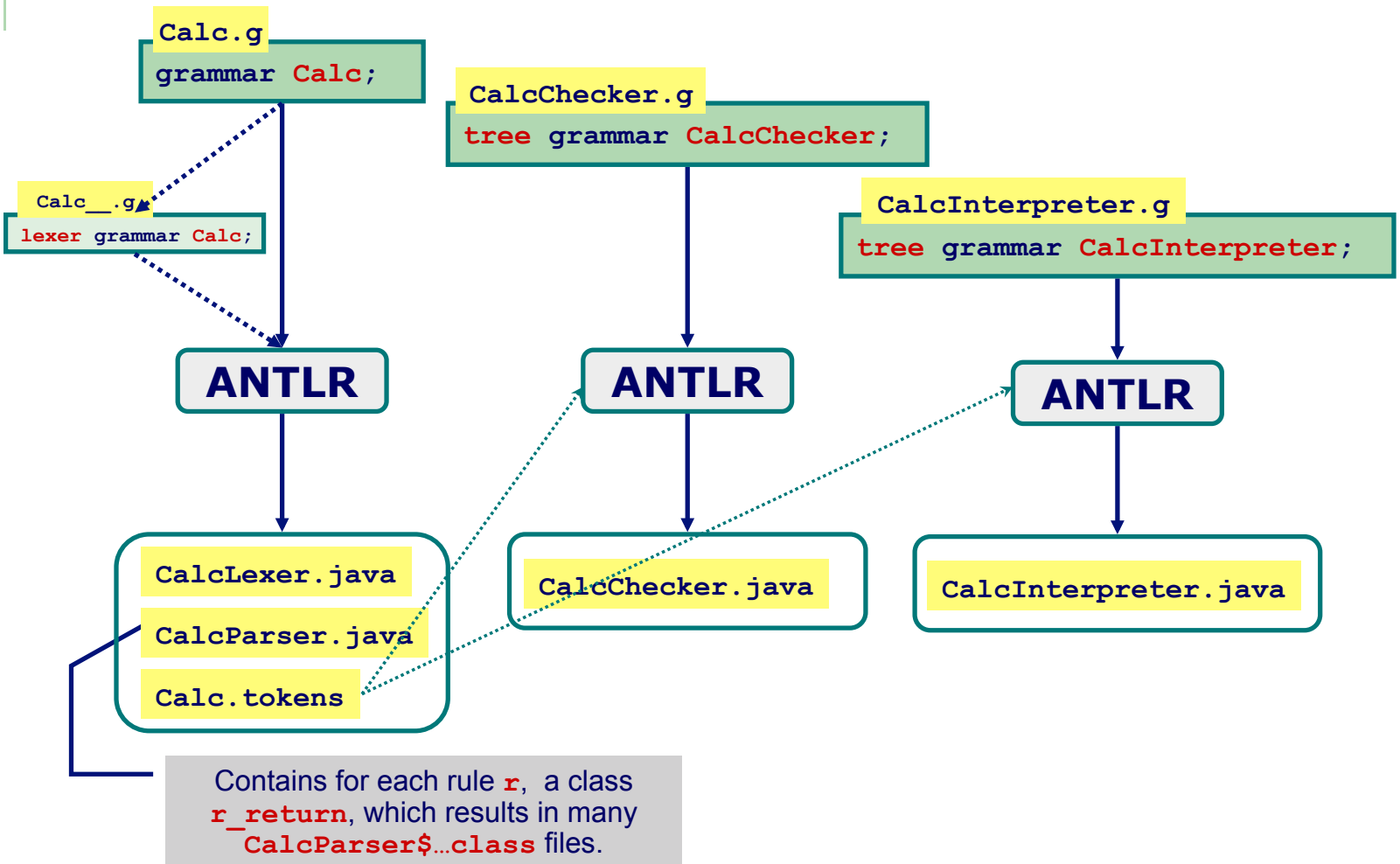
ex1.dot

Render Stop Zoom: 100%



via **DOTTreeGenerator** and GraphViz

Struktur des erzeugten Java-Codes



Auszug aus Parser
Hauptmethode `program()`

```
public class CalcParser extends Parser {
    ...
  public final program_return program() throws RecognitionException {
      program_return retval = new program_return();
      ...
      try {
          // Calc.g:44:9: declarations statements EOF
          {
          pushFollow(FOLLOW_declarations_in_program412);
          declarations1=declarations();
          _fsp--;

          stream_declarations.add(declarations1.getTree());
          pushFollow(FOLLOW_statements_in_program414);
          statements2=statements();
          _fsp--;

          stream_statements.add(statements2.getTree());
          EOF3=(Token)input.LT(1);
          match(input,EOF,FOLLOW_EOF_in_program416);
          stream_EOF.add(EOF3);
          ...
          }
      }
      catch (RecognitionException re) {
          reportError(re);
          recover(input,re);
      } ...
      return retval;
  }
```

Most code that builds the AST is omitted!

```
program
   : declarations statements EOF!
   ;
```

Auszug aus Parser

Parse Deklarationen mit `declarations()`

```
public final declarations_return declarations() throws RecognitionException {
    declarations_return retval = new declarations_return();
    ...
    try {
        ...
        loop1:
        do {
            int alt1=2;
            int LA1_0 = input.LA(1);

            if ( (LA1_0==VAR) )
                alt1=1;

            switch (alt1) {
                case 1 :
                    {
                    pushFollow(FOLLOW_declaration_in_declarations463);
                    declaration4=declaration();
                    ...
                    match(input,SEMICOLON,FOLLOW_SEMICOLON_in_declarations465);
                    }
                    break;
                default :
                    break loop1;
            }
        } while (true);
    } catch (RecognitionException re) {
        ...
    return retval;
}
```

LA(1) - current lookahead Token.

```
declarations
  : (declaration SEMICOLON!)*
  ;
```

Vorteile von ANTLR

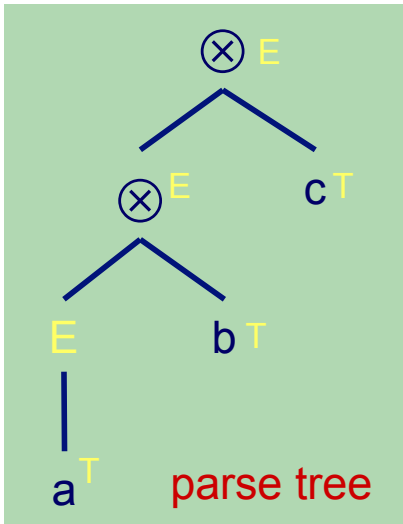
- ANTLR erlaubt Entwickler . . .
  - . . . sich auf Spezifikation des Compiler zu konzentrieren
  - Übernimmt dann Implementation des Compilers
- Gleiche Syntax zur Spezifikation von
  - Lexer/Scanner
  - Parser
  - TreeWalker
- Portable Code-Generierung
  - Java, C, C#, Python, Objective-C, etc
  - . . . gilt aber nicht für in Spezifikationen eingebetteten Code
- Gut unterstützt und aktive Benutzergemeinschaft

Tipps

- Links- und Rechtsassoziativität
- Operatorpräzedenz
- Hängendes `else`

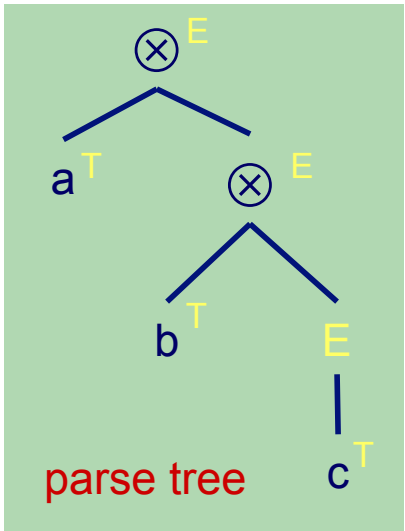
Linksassoziativität

- Linksassoziativer Operator $\otimes$:
  $a \otimes b \otimes c = (a \otimes b) \otimes c$
- Produktion (linksrekursiv!)
  $E ::= E \otimes T | T$
- In EBNF
  $E ::= T(\otimes T)^*$



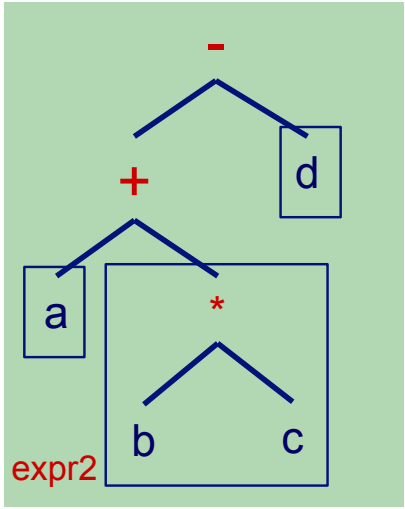
Rechtsassoziativität

- Rechtsassoziativer Operator $\otimes$:
  $a \otimes b \otimes c = a \otimes (b \otimes c)$
- Produktion (linksrekursiv!)
  $E ::= T \otimes E | T$
- In EBNF (? = 0- oder 1-mal)
  $E ::= T(\otimes E)?$



Operatorpräzedenz 1

- Beispiel:
  $a + b \times c - d$
- . . . sollte geparsed werden als
  $(a + (b \times c)) - d$



- Operator $\times$ hat höhere Präzedenz als $+$ und $-$
- In Grammatik ausdrücken, durch Platzieren von $\times$ “näher an Operanden” als $+$ und $-$

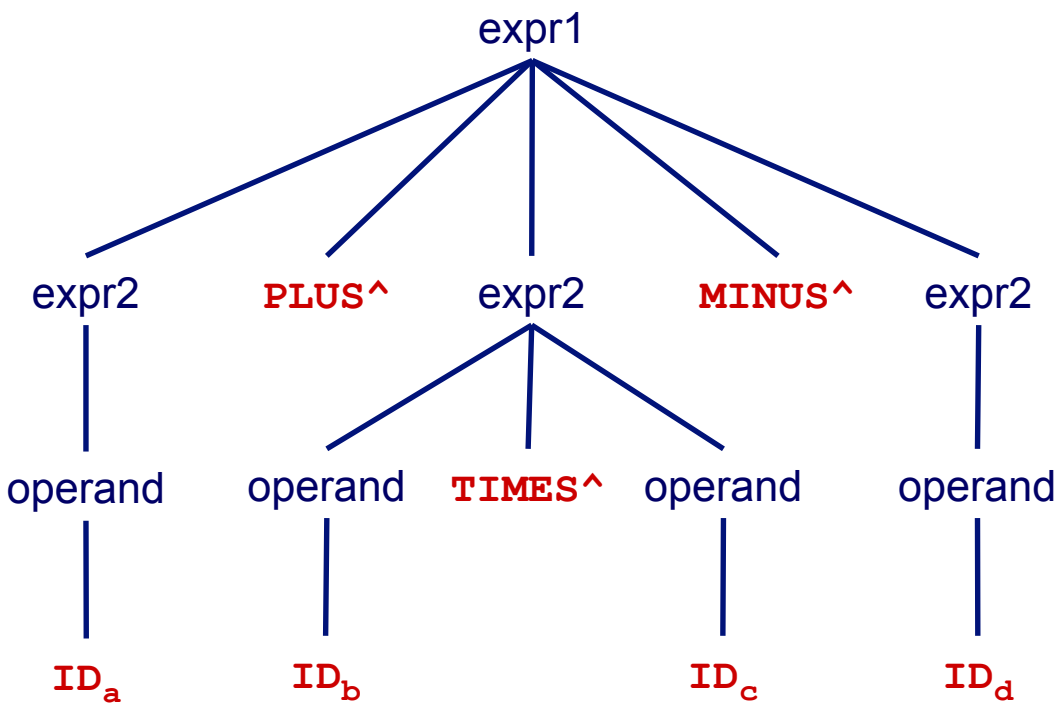
```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```

Operatorpräzedenz 2

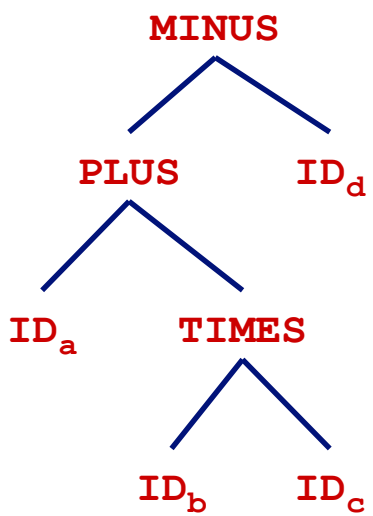
$a + b \times c - d$

```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```

parse tree:



constructed AST:



Hängendes `else` 1

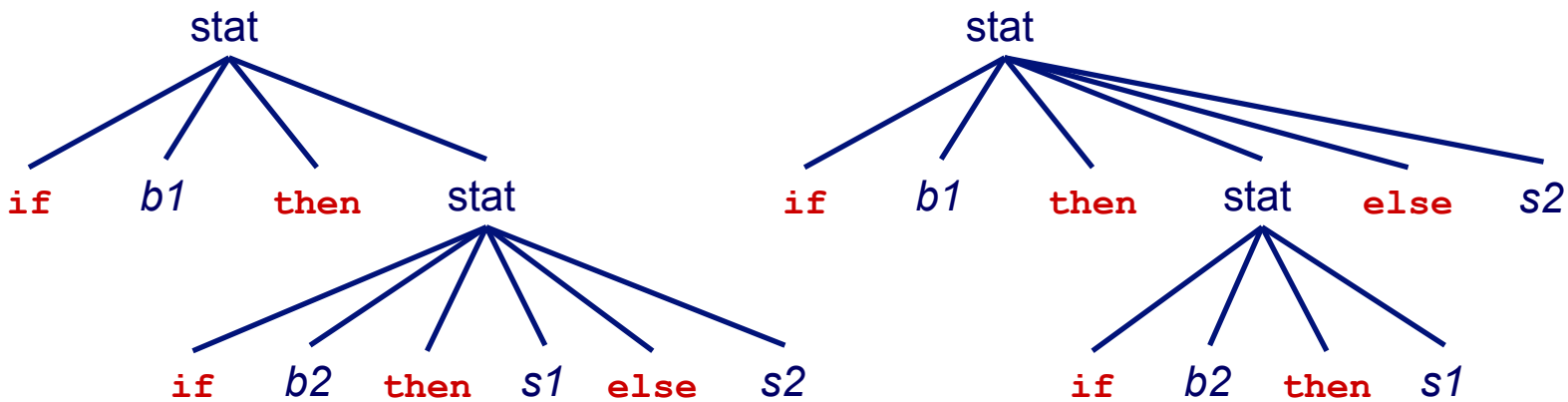
dangling else

Klassisches Problem von Mehrdeutigkeit beim Parsen

```
stat : 'if' expr 'then' stat ('else' stat)?
     | ...      ;
```

```
if b1 then if b2 then s1 else s2
```

Zwei mögliche Parse-Bäume



Hängendes `else` 2

ANTLR gibt Warnung aus

```
warning(200): Foo.g:12:33: Decision can match input
such as "'else'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that
input
```

. . . tut aber das Richtige (*greedy matching* $\rightarrow$ 1. Baum):

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

Muss nicht explizit hingeschrieben werden, da Default.

Weiterführende Themen

Embedded Systems & Applications

Vertiefung von

- Aufbauen von ASTs beim Parsen
- Heterogene ASTs
- Fehlerbehandlung
- Syntaktische Prädikate
- Semantische Prädikate
- Texterzeugung mit StringTemplate
- Automatisches Testen mit gUnit

Aufbau von ASTs
Annotationen in Produktion oder *rewrite rule*

becomes top-node

will be discarded

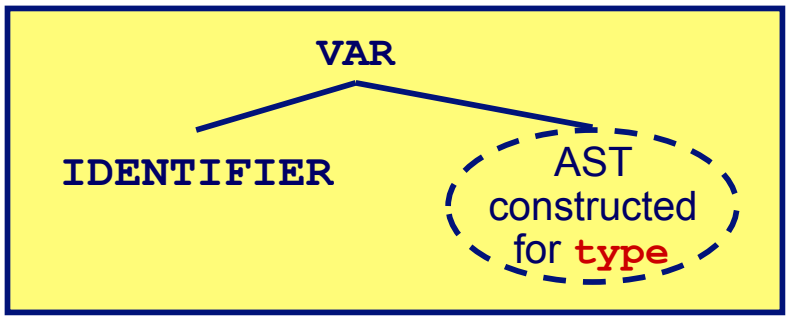
e.g.
```
declaration : VAR^ IDENTIFIER COLON! type ;
```

or:
```
declaration : VAR IDENTIFIER COLON type
              -> ^(VAR IDENTIFIER type);
```

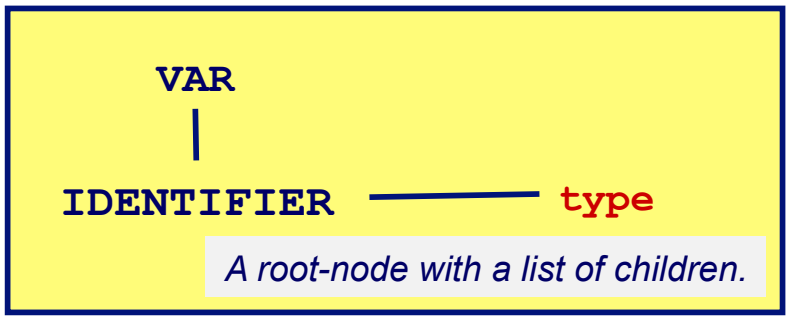
Notations cannot be mixed!

Ergebnis

constructs the following AST



or seen alternatively:



AST kann nun mit ANTLR Tree Parser geparsed werden:

```
declaration : ^(VAR IDENTIFIER type) ;
```

ANTLR uses a prefix pattern language for AST nodes.

Benutzerdefinierte AST-Klassen 1

Embedded Systems & Applications

Bisher: AST aufgebaut aus *Default*-Knoten `tree.CommonTree`

- Ausreichend für viele einfache Sprachen

Benutzerdefinierte AST-Klassen 1

Embedded Systems & Applications

Bisher: AST aufgebaut aus *Default*-Knoten `tree.CommonTree`

- Ausreichend für viele einfache Sprachen
- Ungeeignet, wenn Knoten noch weitere Informationen halten sollen
  - Typen, Bezeichner, runtime entities, etc.

Benutzerdefinierte AST-Klassen 1

Embedded Systems & Applications

Bisher: AST aufgebaut aus *Default*-Knoten `tree.CommonTree`

- Ausreichend für viele einfache Sprachen
- Ungeeignet, wenn Knoten noch weitere Informationen halten sollen
  - Typen, Bezeichner, runtime entities, etc.

- Dann benutzerdefinierte AST-Klasse verwenden
  - Benötigt zwei Klassendefinitionen
  - **`MyTree extends CommonTree`**
    benutzerdefinierte AST-Knoten
  - **`MyTreeAdaptor extends CommonTreeAdaptor`**
    Adapter-Entwurfsmuster zum Anlegen neuer **`MyTree`** Knoten

Benutzerdefinierte AST-Klassen 2
Verwendung in ANTLR

1. Definiere Unterklasse `MyTree` von `CommonTree`

```
public class MyTree extends CommonTree { ...
```

2. Definiere Unterklasse `MyTreeAdaptor` von `CommonTreeAdaptor`

```
class MyTreeAdaptor extends CommonTreeAdaptor { ...
```

3. Veranlasse Parser, AST aus `MyTree`-Knoten aufzubauen

```
MyParser parser = new MyParser(tokens);
parser.setTreeAdaptor(new MyTreeAdaptor());
```

4. Wähle `MyTree` als AST-Klasse in Optionen im Tree Parser aus

```
options { ... ASTLabelType = MyTree; }
```

Beispiel: LIST
Einfache Sprache für Listen und Operationen

- Aufbau von (verschachtelten) Listen
- Atomare Elemente sind Zahlen
- Arithmetische Operationen auf Listen
- Beispiel

```
+[3, 5, *[2, 5], +[3, 7, +[2, 5], 11], 27, 51]
```

Beispiel: LIST
Einfache Sprache für Listen und Operationen

- Aufbau von (verschachtelten) Listen
- Atomare Elemente sind Zahlen
- Arithmetische Operationen auf Listen
- Beispiel

```
+[3, 5, *[2, 5], +[3, 7, +[2, 5], 11], 27, 51]
```

➥Verwende eigene AST-Knoten für Operationen

Operationen auf AST

1a) Berechne Teilsummen/produkte für jede der Unterlisten

1b) Speichere Zwischenergebnisse für die (Unter)Listen ab

- Nicht: Nur ein Ergebnis nach oben weitergeben

2) Fasse Teilergebnisse aller echten Unterlisten zusammen

- `+[3, +[2, 7], *[4, +[2, 3]], 7]` $\rightarrow$ `+[3, 9, 20, 7]`
- Oberste Liste besteht nur noch aus Operator und Zahlen

LIST – Eigene AST-Knoten

`ListNode` is a subclass of ANTLR's default AST class: `CommonTree`.

```
public class ListNode extends CommonTree {
    protected int value = 0;

    public ListNode()            { super();      }
    public ListNode(Token t)     { super(t);     }

    /** Get the List value of this node. */
    public int getValue()        { return value; }

    /** Set the List value of this node. */
    public void setValue(int value) { this.value = value; }

    public String toString() {
        String s = super.toString();
        try { Integer.parseInt(this.getText()); }
        catch (NumberFormatException ex)
            { s = s + " {=" + getValue() + "}"; }
        return s;
    }
}
```

For the string representation, add the value to non-numeric nodes.

Usual set- and get-methods for the extra instance variable of `ListNode`.

Warning: do not override `CommonTree`'s `getType` or `getText`.

LIST – Adapter für eigene AST-Knoten

```
class ListNodeAdaptor extends CommonTreeAdaptor {
    public Object create(Token t) {
        return new ListNode(t);
    }
}
```

The method `create` is used to build to `ListNode` objects.

- Agiert als Adapter zwischen ANTLR-Innereien und eigenen Klassen
- Fungiert als *Factory*-Objekt
  - Ist verantwortlich für ein API zur Erzeugung neuer Objekte der gewünschten Klasse

LIST – Lexer und Parser
Baut AST auf

```
grammar List;

options {
    k=1;
    language=Java;
    output=AST;
}

tokens {
    ...
}

top         :   list EOF! ;
list        :   operator^ elems ;
elems       :   LBRACKET! elem (COMMA! elem)* RBRACKET! ;
elem        :   NUMBER
            |   list
            ;
operator    :   PLUS
            |   TIMES
            ;
...
```

As usual, we only let the parser construct the AST.

build an AST

A (List)AST node is created: the operator-TOKEN is the root-node, and the elements of elems are the children.

= "+"

= "*"

Straightforward lexer rules for NUMBER, whitespace and comments have been omitted.

LIST – Tree Parser 1

Wertet Ausdrücke im AST aus (Op 1a) und speichert Ergebnisse (Op 1b)

Embedded Systems & Applications

Computes the values of List-nodes (i.e. `PLUS-` or `TIMES`-nodes) and stores this value in the corresponding `ListNode` node.

```
list     : operator^ elems ;
elems    : elem+
elem     : NUMBER | list ;
operator : PLUS | TIMES ;
```

```
tree grammar ListWalker;

options { ... ASTLabelType=ListNode; }
@members { /* ... see next slide ... */ }

list        :       { int sum=0; ListNode l=null; }
                ^(p=PLUS
                    (       { l=(ListNode)input.LT(1); }
                        list
                            { sum += l.getValue(); }
                    )+
                )   { $p.setValue(sum); }

            |   ^(t=TIMES list+)
                    { $t.setValue(product(t)); }

            |   n=NUMBER
                    { $n.setValue(Integer.parseInt($n.text)); }
            ;
```

The alternative for `PLUS` computes the sum while walking its children (preferred way).

We need to refer to the actual `ListNodes` of the elements of the sublist.

The alternative for `TIMES` computes the product after all children have been parsed (see the method `product` on the next slide).

Private Methode im `ListWalker` Tree Parser

```
list
  :  ...
  | ^(t=TIMES list+)
    { $t.setValue(product(t)); }
  ;
```

```
tree grammar ListWalker;
...
@members {
    ...

    private int product(ListNode root) {
        int prod = 1;
        for (int i=0; i<root.getChildCount(); i++)
            prod *= ((ListNode) root.getChild(i)).getValue();
        return prod;
    }
}
```

Walk the children of a node `root` and computes the product.

Basis-Klassen für AST Knoten

In ANTLR: **BaseTree** und CommonTree

```
public class BaseTree implements Tree
{
  public int     getChildCount()
  public Tree    getChild(int i)
  public List    getChildren()

  public void    addChild(Tree t)
  public void    addChildren(List kids)
  public void    setChild(int i,Tree t)

  public int     getChildIndex()
  public void    setChildIndex(int ix)
  public Tree    getParent()
  public void    setParent(Tree t)

  public String  toString();
  public String  toStringTree();

  ...
}
```

```
public class CommonTree extends BaseTree
{

  public Token   getToken()
  public Tree    dupNode()
  public boolean isNil()

  public int     getType()
  public String  getText()
  public int     getLine()

  ...
}
```

- The **BaseTree** is a generic tree implementation with no payload. You must subclass **BaseTree** to actually have any user data.
- A **CommonTree** node is wrapper for a **Token** object.

LIST – ListTopLevel 1

Baut Liste nur aus Ergebnissen aller Unterlisten (Op 2)

Embedded Systems & Applications

- Alle Elemente sind `NUMBER`-Knoten
  - Werden neu angelegt (*imaginary nodes*)
  - Verweisen auf Ursprungstokens (Zeile/Spalte, Text)
- Liste wird direkt umgeschrieben
  - Alle Elemente ohne *rewrite rules* werden unmodifiziert ausgegeben

LIST – ListTopLevel 2

```
tree grammar ListTopLevel;

options {
   tokenVocab=List;
   ASTLabelType=ListNode;
   output=AST;
   rewrite=true;
}

root    :   ^(PLUS   (elem)+)
        |   ^(TIMES  (elem)+)
        ;

list    :   ( ^(p=PLUS   (elem)+) -> ^(NUMBER[p.getToken(),""+p.getValue()])
            | ^(t=TIMES  (elem)+) -> ^(NUMBER[t.getToken(),""+t.getValue()])
            )
        ;

elem    :    NUMBER
        |    list
        ;
```

output a (new) AST

in-line replacement of nodes

in-line replacement of nodes

LIST – Hauptprogramm

Für Operation 1: Berechnen und Speichern der Zwischenergebnisse

```
public static void main(String[] args) {
   ...
   try {
       ListLexer lexer =
           new ListLexer(new ANTLRInputStream(System.in));
       CommonTokenStream tokens = new CommonTokenStream(lexer);
       ListParser parser = new ListParser(tokens);
       parser.setTreeAdaptor(new ListNodeAdaptor());

       ListParser.top_return result = parser.top();
       ListNode tree = (ListNode) result.getTree();

       TreeNodeStream nodes = new CommonTreeNodeStream(tree);
       ListWalker walker = new ListWalker(nodes);
       walker.top();
       ...
       System.out.println(">> Total: " + tree.getValue());

    } catch (RecognitionException e) { ... }
}
```

Make sure that `ListNode` objects are created.

Print the value of the root node.

Heterogene ASTs 1

Embedded Systems & Applications

Bisher in ANTLR: Homogene ASTs, alle Knoten haben denselben Typ

- Problem: Was, wenn unterschiedliche Attribute gespeichert werden müssen?

Heterogene ASTs 1

Embedded Systems & Applications

Bisher in ANTLR: Homogene ASTs, alle Knoten haben denselben Typ

- Problem: Was, wenn unterschiedliche Attribute gespeichert werden müssen?

Ein Ansatz: `Map<String,Object> properties` als Feld in Knoten

- Flexibel, beliebige Dinge abspeicherbar
- Nachteil: Nicht typsicher, schwer wartbar

Heterogene ASTs 1

Embedded Systems & Applications

Bisher in ANTLR: Homogene ASTs, alle Knoten haben denselben Typ

- Problem: Was, wenn unterschiedliche Attribute gespeichert werden müssen?

Ein Ansatz: `Map<String,Object> properties` als Feld in Knoten

- Flexibel, beliebige Dinge abspeicherbar
- Nachteil: Nicht typsicher, schwer wartbar

➥Heterogene Bäume
Verschiedene Knotenarten in einem Baum

Heterogene ASTs 2

Gezielt verschiedene Knotenarten anlegen durch < · · · >

```
program       :   declarations statements EOF
                    -> ^(PROGRAM declarations statements)
              ;
declarations  :   (declaration SEMICOLON!)*
              ;
statements    :   (statement SEMICOLON!)+
              ;
declaration   :   VAR^ IDENTIFIER<IdNode> COLON! type
              ;
statement     :   assignment
              |   print
              ;
assignment    :   lvalue BECOMES^ expr
              ;
print         :   PRINT^ LPAREN! expr RPAREN!
              ;
lvalue        :   IDENTIFIER<IdNode>
              ;
expr          :   operand (( PLUS<BinExprNode>^
                           | MINUS<BinExprNode>^) operand)*
              ;
operand       :   IDENTIFIER<IdNode>
              |   NUMBER
              |   LPAREN! expr RPAREN!
              ;
type          :   INTEGER<TypeNode>
              ;
```

With the `<...>` suffix annotation, one can specify the node type of a node.

In this example for Calc, there are three extra node types:

- `IdNode`
- `BinExprNode`
- `TypeNode`

All these classes have to be defined as subclasses of (a subclass of) `CommonTree`. Just like we did for `ListNode`.

Resist the urge to define and use many (>10) heterogeneous AST nodes.

With ANTLR (usually) at most a handful is needed. Due to the complete OO approach, W&B had to use a complete heterogenous approach.

Fehlerbehandlung 1

- ANTLR-generierte Erkenner behandeln Fehler durch Java Exceptions
  - **`RecognitionException`** ist Basisklasse aller ANTLR Exceptions

Fehlerbehandlung 1

- ANTLR-generierte Erkenner behandeln Fehler durch Java Exceptions
  - **`RecognitionException`** ist Basisklasse aller ANTLR Exceptions

- Schon gesehen in: **`CalcChecker`**
  - Wirft **`CalcException`** bei kontextuellen Fehlern
  - Bricht dann Programm ab

```
@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}
```

With this **`@rulecatch`** clause, we specified that an **`RecognitionError`** is not handled, but re-thrown to the **`main`** method. This essentially means that the Calc compiler stops at the first error.

Fehlerbehandlung 2

Default Exception Handler in `Parser` und `TreeParser`

- Fängt alle `RecognitionExceptionS`
- Gibt Fehlermeldung aus
- Setzt dann Parsing fort

`ListWalker`

```
list  :  ...
      |  n=NUMBER
           { if ($n.text.equals("211035"))
                throw new RecognitionException(
                        "211035 on line " + $n.getLine() +
                        " is not a valid number");
             else
                $n.setValue(Integer.parseInt($n.text));
           }
      ;
```

The number "211035" is tagged as a `RecognitionException`. The `ListWalker` class will catch the `Exception` and report the error. Then it will proceed in walking the tree. Note that we use the line number that is associated with the Token of the NUMBER node.

Fehlerbehandlung 3

Beispiel für benutzerdefinierte Fehlerbehandlung

- Definiere eigene Exception-Klasse **`ListException`**
- Redefinieren von **`displayRecognitionError()`**
  - Definiert in **`BaseRecognizer`**

```
tree grammar ListWalker;
...
@members {
  protected int nrErr = 0;
  public    int nrErrors() { return nrErr; }

  public void displayRecognitionError(
                String[] tokenNames, RecognitionException e) {
      nrErr = nrErr+1;
      if (e instanceof ListException)
          emitErrorMessage("[List] error: " + e.getMessage());
      else
          super.displayRecognitionError(tokenNames, e);
    }
}
```

Counting the total number of errors.

`ListException` is an user defined exception (in the style of **`CalcException`**).

Fehlerbehandlung 4

Embedded Systems & Applications

Exceptions können auch direkt in Produktionen behandelt werden

```
rule : foo BAR SEMI!
     ;
     catch [RecognitionException re] {
         reportError(re);
         consumeUntil(input, SEMI);
         input.consume();
     }
```

Error recovery: consume all tokens until and including the SEMI token.

Syntaktische Prädikate 1

Beispiel für nicht-LL(1) Grammatik

```
rule : X Y
     | X Z
     ;
```

LL(1) problem.

can be solved using left-factorization:

```
rule : X (Y | Z) ;
```

Syntaktische Prädikate 1

Beispiel für nicht-LL(1) Grammatik

```
rule : X Y
     | X Z
     ;
```

LL(1) problem.

can be solved using left-factorization:

```
rule : X (Y | Z) ;
```

Anderer Lösungsansatz: Syntaktische Prädikate

```
rule : (X Y) => X Y
     | X Z
     ;
```

Only when X Y appears in the tokenstream take this alternative.

This can be regarded as 'locally setting k to 2'.

Syntaktische Prädikate 1

Beispiel für nicht-LL(1) Grammatik

```
rule : X Y
     | X Z
     ;
```

LL(1) problem.

can be solved using left-factorization:

```
rule : X (Y | Z) ;
```

Anderer Lösungsansatz: Syntaktische Prädikate

```
rule : (X Y) => X Y
     | X Z
     ;
```

Only when X Y appears in the tokenstream take this alternative.

This can be regarded as 'locally setting k to 2'.

Syntax für syntaktische Prädikate

```
( prediction block ) => production
```

Syntaktische Prädikate 2

- Können beliebig weiten Lookahead benutzen
- Mächtiger als LL(*)
  - Lokale CFG statt lokalem DFA für Lookahead
- Führen selektives Backtracking durch, um Mehrdeutigkeiten aufzulösen
  - Eventuell vorhandene Aktionen werden dabei ignoriert

```
expr : (ID LPAREN) => ID LPAREN params RPAREN
     | ID BECOMES ...
     | ...
;
```

```
foo(int x);
x=...
```

Semantische Prädikate 1

- Erlauben Angabe von beliebiger Bedingung beim Parsing
- Beschreibung der Bedingung durch Java-Code

```
{ semantic-predicate-expression } ?
```

Verwendung auf zwei Arten

- Validierende Prädikate
- Vereindeutigende Prädikate (*disambiguating predicates*)

Semantische Prädikate 2
Validierende Prädikate

Lösen Exception aus, wenn Bedingung verletzt wird
Beispiel:

```
decl : ^(VAR id=IDENTIFIER type)
       { if (isDeclared($id.text))
             throw new CalcException(...);
         else
             declare($id.text); };
```

CalcChecker

```
decl : ^(VAR id=IDENTIFIER type)
       { !isDeclared($id.text) }?
       { declare($id.text); }
     ;
```

with validating predicate

Validierendes Prädikat steht nach erkanntem Symbol (Terminal/Nichtterminal/Knoten)

Semantische Prädikate 3
Vereindeutigende Prädikate

Lösen Mehrdeutigkeiten beim Parsen aufgebaut

- Stehen als Erstes in jeder Parsing-Alternative

```
stat    :   // declaration "type varName;"
            { isTypeName(input.LT(1).getText()) }?
            ID ID SEMICOLON                  // declaration
        |   ID BECOMES expr SEMICOLON        // assignment
        ;
```

```
int x;
x=3;
```

Disambiguating predicates must be the first item of an alternative.

first lookahead-**Token**

```
public interface TokenStream {
  ...

  /** Return the i-th token of lookahead */
  public Token LT(int i);
}
```

StringTemplate 1

- Inverse Operation zum Parsen
- Erzeuge strukturierten Text
- “Ausfüllen” von Textfeldern in Vorlagen

StringTemplate 1

- Inverse Operation zum Parsen
- Erzeuge strukturierten Text
- “Ausfüllen” von Textfeldern in Vorlagen

Beispiel: Code-Generierung für TAM in ANTLR Tree Walker

```
assignment
   : ^(BECOMES id=IDENTIFIER expr)
         { int addr = dict.get($id.text);
           emit("STORE(1)" + addr + "[SB]");
         }
   ;
```

```
expr
   : operand
   | ^(PLUS expr expr)
         { emit("CALL  add");
   | ^(TIMES expr expr)
         { emit("CALL  mult");
   ...
   ;
```

```
operand
   : id=IDENTIFIER
         { int addr = dict.get($id.text);
           emit("LOAD(1)" + addr + "[SB]");
         }
   | n=NUMBER
         { emit("LOADL" + $n.text); }
   ;
```

ANTLR’s StringTemplates can be used to collect all these emit strings in a separate file (i.e., a template).

StringTemplate 2
Code-Erzeugung für TAM

A string template allows another level of indirection to isolate the target instructions.

CalcCodeGeneratorStringTemplate.g

```
tree grammar Generator;
options { ...
   output=template;
}
```

build a template

```
statement
   : ^(BECOMES id=IDENTIFIER expr)
     -> assign(addr={dict.get($id.text)},
               expr={$expr.st})
   | ^(PRINT expr)
     -> print(expr={$expr.st})
```

```
operand
   : id=IDENTIFIER
     -> loadvar(addr={dict.get($id.text)})
   | n=NUMBER
     -> loadnum(val={$n.text})
   ;
```

Every non-terminal has a variable `st` which corresponds with the string template that it returns.

tam.stg

```
assign(addr,expr) ::= <<
<expr>
STORE(1) <addr>[SB]
>>
```

```
print(expr) ::= <<
<expr>
CALL  putint
CALL  puteol
>>
```

```
loadvar(addr) ::= <<
LOAD(1) <addr>[SB]
>>
```

```
loadnum(val) ::= <<
LOADL <val>
>>
```

Note the resemblance between string templates and W&B's code templates.

StringTemplate 3
Code-Erzeugung für TAM

original code generator

```
expr
    : operand
    | ^(PLUS expr expr)
          { emit("CALL  add");
    | ^(TIMES expr expr)
          { emit("CALL  mult");
    ;
```

Note that each node (expr) of the AST is responsible for generating its code.

CalcCodeGeneratorStringTemplate.g

```
expr
  : operand
    { $st = $operand.st; }

  | ^(PLUS  x=expr y=expr
    -> binexpr(e1={x.st}, e2={y.st},
               instr="add")

  | ^(TIMES x=expr y=expr)
    -> binexpr(e1={x.st}, e2={y.st},
               instr="mult")
  ;
```

You have to explicitly state that the child's string template has to be copied.

tam.stg

```
binexpr(e1,e2,instr) ::= <<
<e1>
<e2>
CALL <instr>
>>
```

All machine specific code generation definitions appear together within the template definition (i.e., **tam.stg**).

It is 'easier' to port the code generator to a different target machine: "just" write a new template definition.

Automatischer Test von ANTLR-Grammatiken mit gUnit

```
gunit SimpleC;

// Teste Produktion variable
variable:
"int x"  FAIL     // Erwartet Fehler wegen fehlendem ';'
"int x;" OK       // Erwartet fehlerfreies Parsen

// Test Produktion functionHeader
functionHeader:
"void bar(int x)" returns ["int"]  // erwartet Ergebnis "int" von Produktion

// Teste Produktion program mit mehrzeiliger Eingabe
program:
<<
char c;
int x;
>> OK        // Erwarte erfolgreiches Parsen

// Teste lexikalische Regeln
ID:
"abc123"  OK    // Erfolg erwartet
"XYZ@999" OK    // Erfolg erwartet
"123abc"  FAIL  // erwarte Fehler

INT:
"00000"     OK
"123456789" OK
```