

# Grundlagen des Übersetzerbaus (2)

Prof. Dr. Michael Philippsen



# Modulare Struktur von Übersetzern

## Analyse

*Token*

*Syntax*

Semantik

## Abbildung

Transf.

Optim.

## Codierung

Code-Erz.

Ass./Bind.

*Quelltext*

*Token-  
folge*

Struktur-  
baum

attrib.  
Struktur-  
baum

Zwischen-  
Code

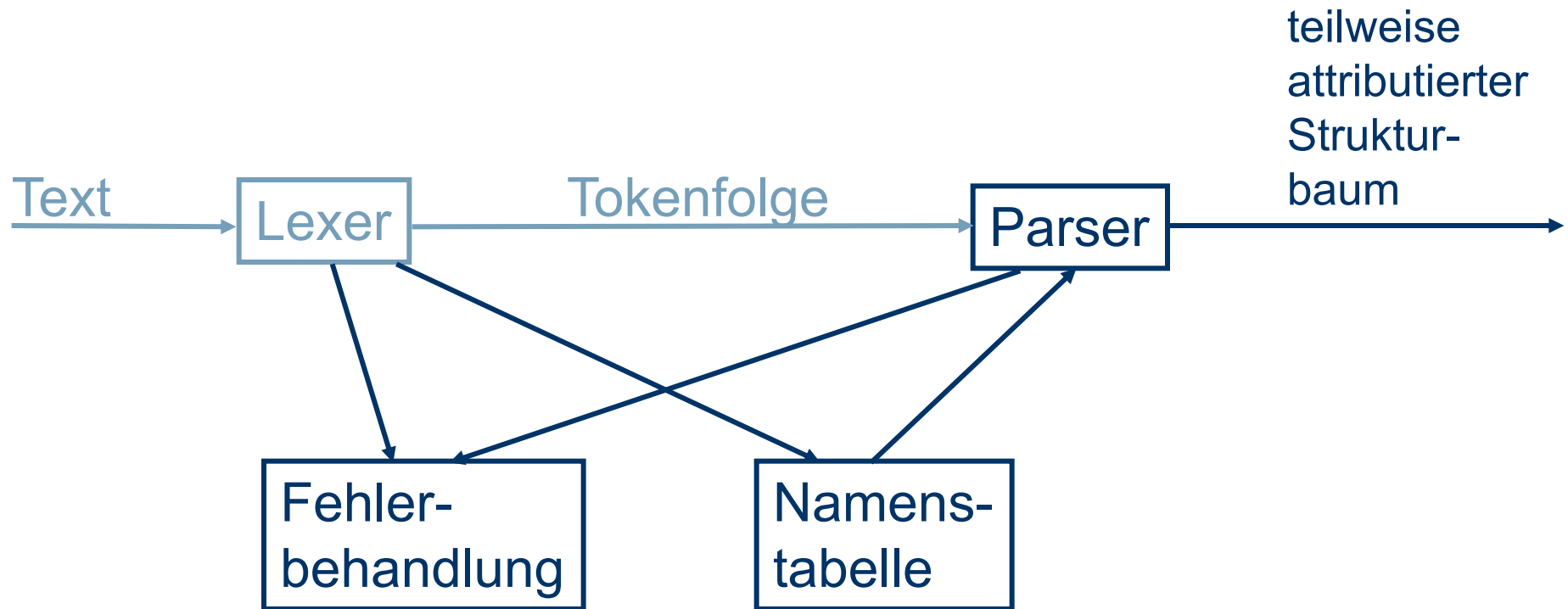
Ziel-  
programm

*Namens-  
tabelle*

Definitions-  
tabelle

- Aufgaben:
  - Feststellung der bedeutungstragenden Elemente
  - Zuordnung statischer Bedeutung
  - Konsistenzprüfung
- Schritte:
  - Lexikalische Analyse/Abtastung (Lexer)
  - Syntaktische Analyse/Zerteilung (Parser)
  - Semantische Analyse
- Noch keine Übersetzung, nur Analyse

# Zusammenspiel zwischen Lexer und Parser



# Lexikalische Analyse/Abtastung

- Ein **Lexer** (Scanner, lexikalischer Analysator/Abtaster/Symbolentschlüsseler) ...
  - zerlegt das Quellprogramm (Text → Folge von Zeichen) in eine Folge von *atomaren* bedeutungstragenden Einheiten (**Token**):
    - FLOAT, ID, LPAREN, CHAR, MUL, LBRACE, RELOP.
    - Manche Token tragen zusätzlich einen „semantischen Wert“.
      - Beispiel: ID-Token tragen (ggf. kodierten) Namen des Bezeichners
  - beseitigt überflüssige Zeichen(folgen) wie
    - Kommentare,
    - Leerzeichen, Tabulatoren, ...

Achtung: In Programmiersprachen wie Python trägt die Einrückung Semantik und darf daher nicht vom Lexer entfernt werden!

# Spezifikation mittels Regulärer Ausdrücke

- Heute üblich: Jeder Token-Typ wird mit Hilfe eines Regulären Ausdrucks beschrieben
  - auch: Reguläre Grammatik, Chomsky-3-Grammatik

- **PLUS**: "+" ;                      **MINUS**: "-" ;                      **MUL**: "\*" ;                      ...
- **LBRACE**: "(" ;                      **RBRACE**: ")" ;
- **INT**: "0" | ([1-9][0-9]\*) ;
- **VAR**: "var" ;
- **ID**: [a-zA-Z\_][a-zA-Z0-9\_]\* ;
- **SEP**: [\s\t\n]+
- ...

| → Alternative

Achtung: Definition von **ID** umfasst die von **VAR**!  
→ Erfordert Sonderbehandlung

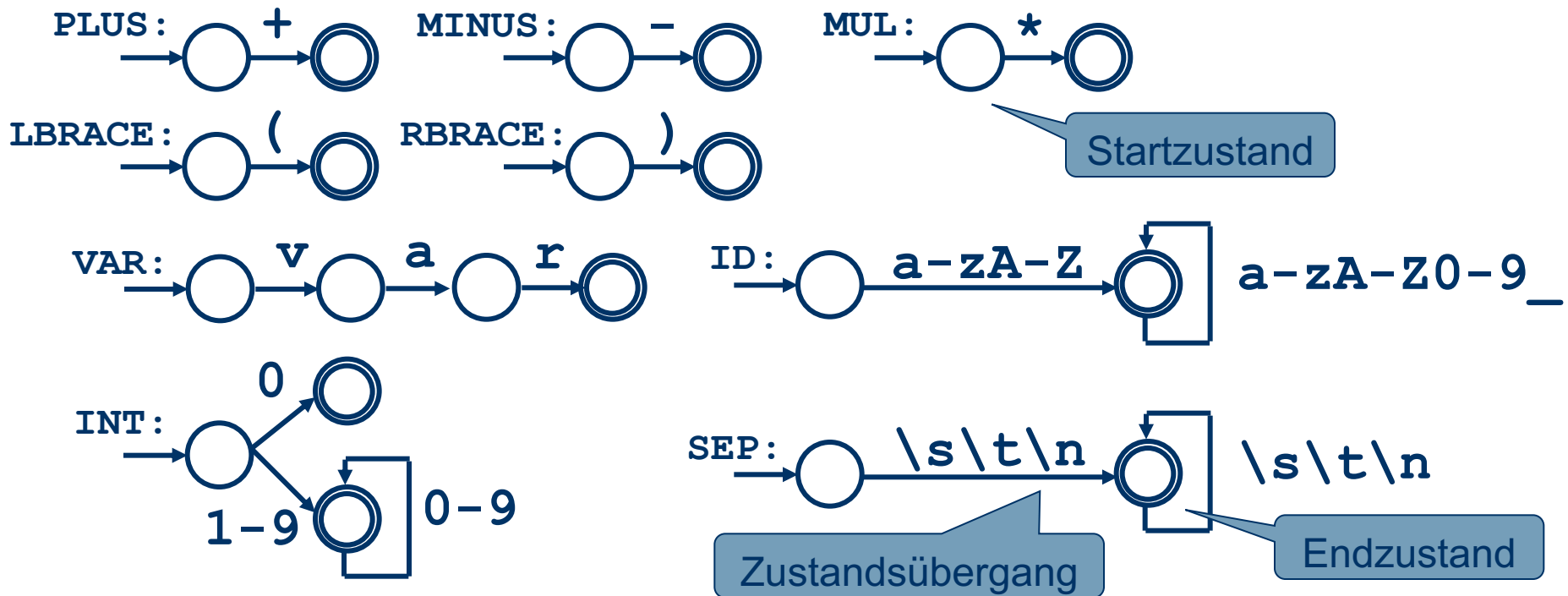
\* → „beliebig viele“

+ → „beliebig viele, aber mind. 1“

[...] → beliebiges Zeichen aus Menge

# Deterministische Endliche Automaten

- Zu jedem Reg. Ausdruck kann ein Deterministischer Endlicher Automat (DEA, engl. DFA) angegeben werden



- Laufzeit der DEA-Anwendung ist linear in Eingabegröße
  - → DEAs eignen sich prima für Abtastung

# DEA-Repräsentation per Tabelle

	Char <sub>1</sub>	Char <sub>2</sub>	Char <sub>3</sub>	...
Zust <sub>1</sub>				
Zust <sub>2</sub>		s <sub>3</sub>		
Zust <sub>3</sub>			⊗	
...				

s<sub>i</sub> Zeichen aus Eingabe konsumieren, dann weiter in Zustand i.

⊗ Fehlerzustand.

Stichworte für die weitere Recherche:

- Teilmengenkonstruktion
- Minimierung des Automaten
- Tabellenkomprimierung



# Verknüpfung mehrerer DEAs

- Jeder Token-Typ wird durch „eigenen“ DEA beschrieben, Lexer muss jedoch *alle* Token-Typen erkennen können
- Zwei Möglichkeiten (grob!):

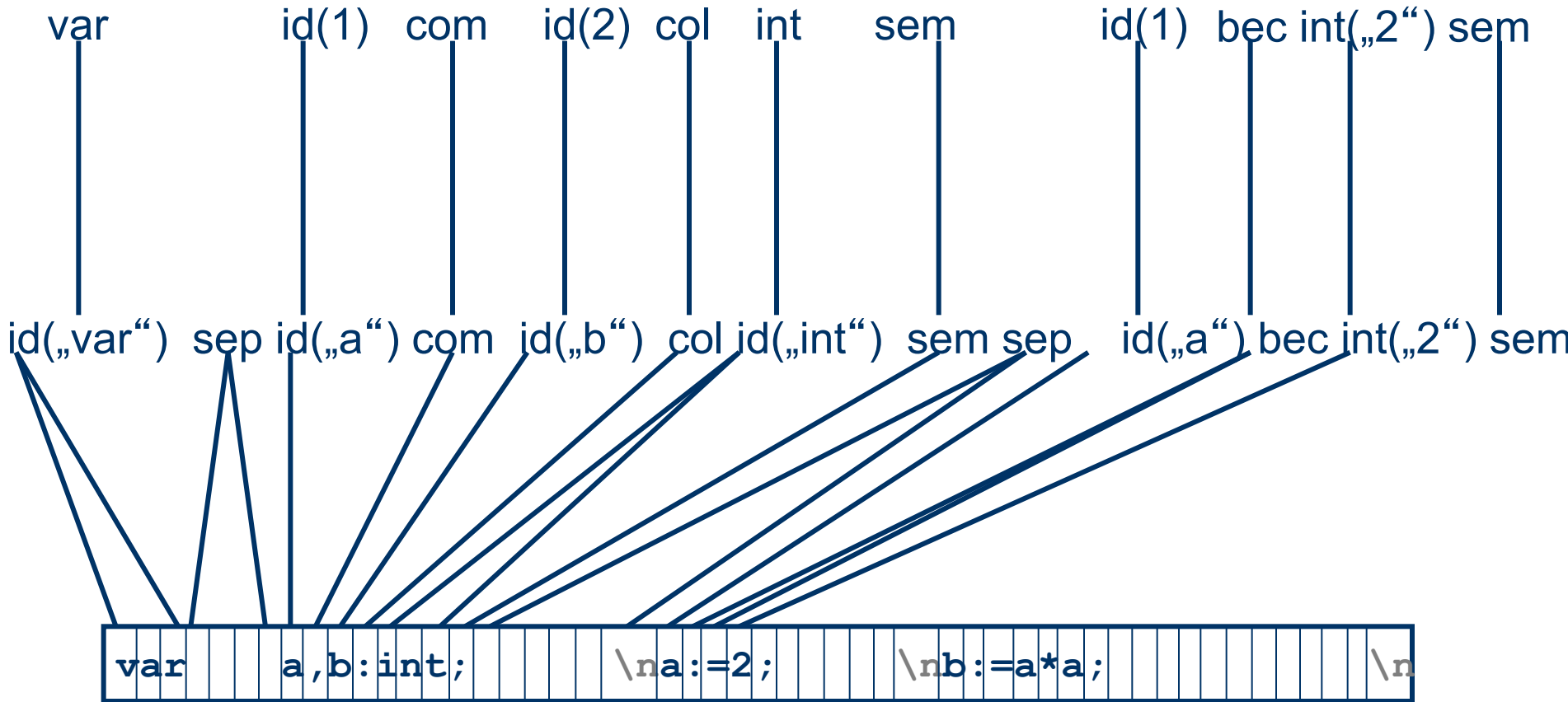
- Aus allen DEAs einen „großen“ DEA erzeugen
- Einzelne DEAs „gleichzeitig“ anwenden; konzeptuell:

//  $D_1, D_2, \dots, D_n$ : DEAs für die unterschiedlichen Token-Typen

```
while Eingabe is not empty do
   $s_i$  := längstes durch  $D_i$  akzeptiertes Präfix;
   $k$  :=  $\max\{|s_i|\}$  // Längstes Präfix aller  $D_i$  („maximal munch“)
  if  $k > 0$  then
     $j := \min\{i : |s_i| = k\}$ ; // falls nicht eindeutig → Sonderbehandlung
    entferne  $s_j$  aus Eingabe; //  $\hookrightarrow$  (z.B. anhand der spez. Reihenfolge)
    führe  $j$ -te Aktion aus // z.B.: Token mit  $j$ -tem Token-Typ ausgeben
  else
    Fehlermeldung // kein  $D_i$  kann Endzustand erreichen
  end
end
```

# Lexer: Beispiel

Zuordnung über Namenstabelle  
(Details folgen später)



- Es gibt Werkzeuge, die aus Spezifikationen effiziente Lexer (als Code in einer Programmiersprache) erzeugen.
  - Eingabe: Liste regulärer Ausdrücke  
plus ggf. zugehörige Aktionen (Programm-Code).
  - Ausgabe: Code zur effizienten Token-Erkennung;  
Programm-Code ist Nebeneffekt bei Erkennung  
eines Tokens.

# Beispiel: Generierung von Lexer (und Parser) mit ANTLR

Sample.g4

antlr

SampleLexer.java

javac

SampleLexer.class

WHITESPACE: (' ' | '\t' | '\r' | '\n') -> skip;

COMMENT: '#' ~('\n' | '\r')\* '\r'? '\n' -> skip;

Token wird nicht an Parser  
weitergereicht.

PLUS: '+';

MINUS: '-';

STAR: '\*';

Action, die bei Erkennung ausgeführt wird

SLASH: '/' {System.out.println("found slash");};

LPAREN: '(';

RPAREN: ')';

Regulärer Ausdruck  
beschreibt Menge der  
Zeichenfolgen, die durch  
Token abgebildet werden.

NUMBER: [0-9]+ ('.' [0-9]\*)? ;

IDENTIFIER: [a-zA-Z\_]+[a-zA-Z0-9\_]\*;

# ADT Lexer (abstrakter Datentyp)

```
abstract class Lexer is
    initialisiere();
    beende();
    naechstesToken(): Token;
end;
```

## ■ Benötigte Operationen:

### □ Eingabe:

- `oeffne/schliesse(Eingabedatei)`
- `naechstesZeichen`

### □ Namenstabelle:

- `sucheOderTrageEin(Text)`

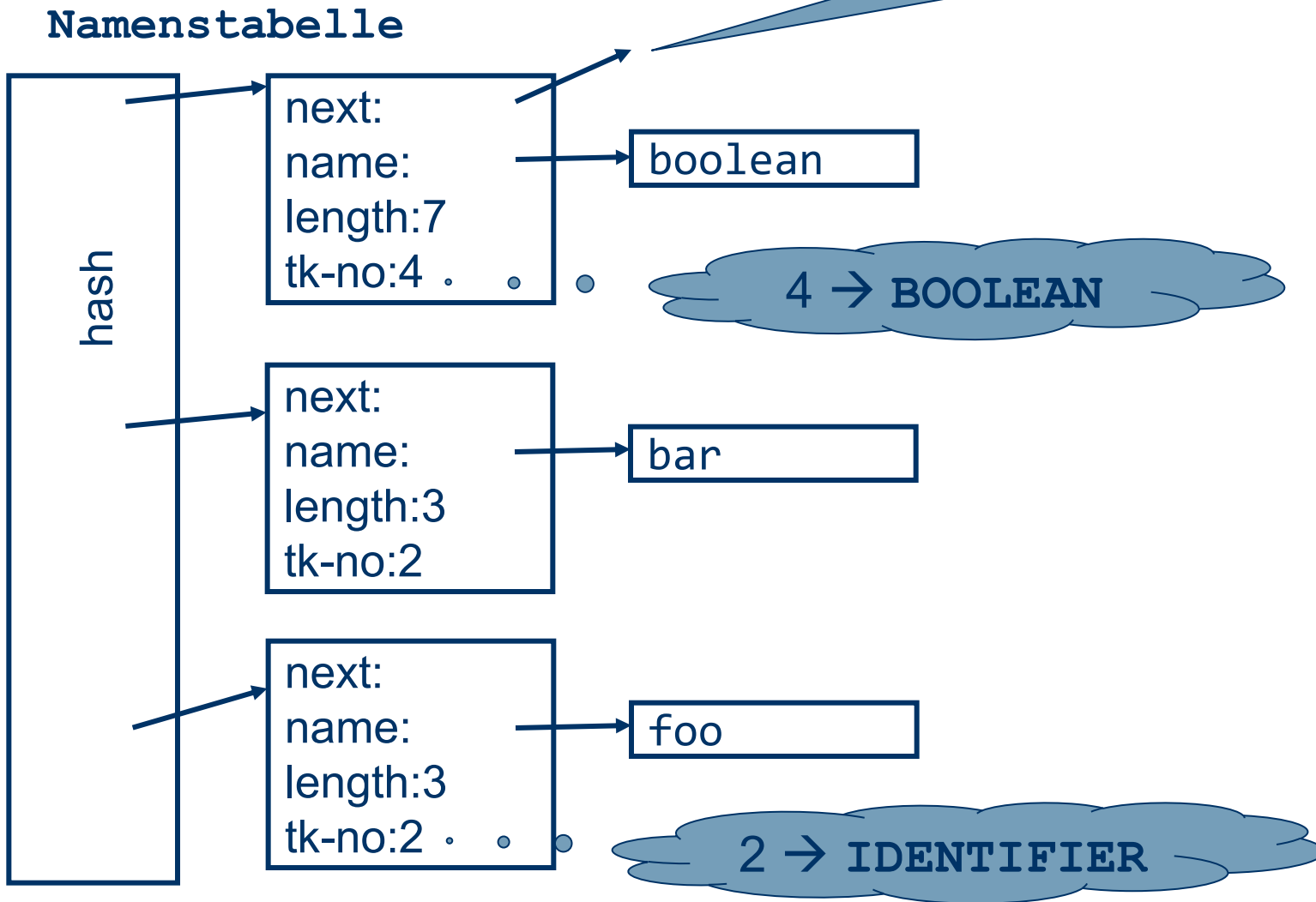
### □ Fehlerbehandlung:

- `Fehlereintrag(Nr, Text)`

- Kodierung von Bezeichnern
- Unterscheidung zwischen Bezeichnern und vordefinierten Schlüsselwörtern

# Organisation der Namenstabelle

Hash-Tabelle mit Verkettung



# ADTs Namenstabelle und NamenstabellenEintrag

```
abstract class Namenstabelle is
```

```
    initialisiere();
```

```
    beende();
```

```
    trageEin(Text, Tk-No);
```

```
    sucheOderTrageEin(Text): NamenstabellenEintrag;
```

```
end;
```

Der Lexer trägt damit die vordefinierten Schlüsselwörter in die Namenstabelle ein.

```
abstract class NamenstabellenEintrag is
```

```
    next: NamenstabellenEintrag; // Verkettung
```

```
    name: Text; // Zeichenfolge, z.B. als Bytes
```

```
    length: Integer; // Länge der Zeichenfolge
```

```
    tk-no: Integer; // Tokennummer
```

```
end;
```

# ADTs Token und Tokenfolge

```
abstract class Token is
    tk-no: Integer;                // Nummer des Token-Typs
    merkmal: NamenstabellenEintrag; // für Bezeichner und
                                    // Konstanten, sonst NIL
    pos: Position;                // für Fehlermeldungen
end;

abstract class Tokenfolge is Strom(Token);
```



# Token für Java

```
public interface Tokens {  
  
    public static final int  
        EOF = 0,  
        ERROR = EOF + 1,  
        IDENTIFIER = ERROR + 1,  
        ABSTRACT = IDENTIFIER + 1,  
        BOOLEAN = ABSTRACT + 1,  
        BREAK = BOOLEAN + 1,  
        BYTE = BREAK + 1,  
        CASE = BYTE + 1,  
        CATCH = CASE + 1,  
        CHAR = CATCH + 1,  
        CLASS = CHAR + 1,  
        CONST = CLASS + 1,  
        CONTINUE = CONST + 1,  
        DEFAULT = CONTINUE + 1,  
        DO = DEFAULT + 1,  
        DOUBLE = DO + 1,  
        ELSE = DOUBLE + 1,  
        EXTENDS = ELSE + 1,
```

...

```
        BAR = AMP + 1,  
        CARET = BAR + 1,  
        PERCENT = CARET + 1,  
        LTLT = PERCENT + 1,  
        GTGT = LTLT + 1,  
        GTGTGT = GTGT + 1,  
        PLUSEQ = GTGTGT + 1,  
        SUBEQ = PLUSEQ + 1,  
        STAREQ = SUBEQ + 1,  
        SLASHEQ = STAREQ + 1,  
        AMPEQ = SLASHEQ + 1,  
        BAREQ = AMPEQ + 1,  
        CARETEQ = BAREQ + 1,  
        PERCENTEQ = CARETEQ + 1,  
        LTLTEQ = PERCENTEQ + 1,  
        GTGTEQ = LTLTEQ + 1,  
        GTGTGTEQ = GTGTEQ + 1,  
        TokenCount = GTGTGTEQ + 1;
```

```
}
```



# Lexer in javac (nicht generiert), Hauptschleife

```
public void nextToken() {  
    while (true) {  
        pos = makePosition(line, col);  
        int start = bp;  
        switch (ch) {  
            // Abfackeln von Space, Tab, CR, NL,  
            ...  
            // Identifikator  
            case 'A': case 'B': case 'C': case 'D': ... case 'X': case 'Y': case 'Z':  
            case 'a': case 'b': case 'c': case 'd': ... case 'x': case 'y': case 'z':  
            case '$': case '_':  
                scanIdent();  
                return;  
            // „Satzzeichen“  
            case ',': scanChar(); token = COMMA; return;  
            case ';': scanChar(); token = SEMI; return;  
            case '(': scanChar(); token = LPAREN; return;  
            case ')': scanChar(); token = RPAREN; return;  
            case '[': scanChar(); token = LBRACKET; return;  
            case ']': scanChar(); token = RBRACKET; return;  
            ...  
        }  
    }  
}
```

Nächstes Zeichen wird betrachtet.

Rest des Identifikators wird gelesen.

# Lexer in javac (nicht generiert), Identifikator

```
private void scanIdent() {  
    do {  
        scanChar();  
        switch (ch) {  
            case 'A': case 'B': case 'C': case 'D': ... case 'X': case 'Y': case 'Z':  
            case 'a': case 'b': case 'c': case 'd': ... case 'x': case 'y': case 'z':  
            case '$': case '_':  
            case '0': case '1': case '2': case '3': ... case '7': case '8': case '9':  
                break;  
            default:  Zeichen erkannt, das nicht zum Identifikator gehört.  
                name = Name.fromChars(sbuf, 0, sp);  Eintrag in Namenstabelle.  
                if (name.index <= maxKey) token = key[name.index];  
                else token = IDENTIFIER;  
                return;  
            }  
        } while (true);  
    }  
}
```

Erkennung eines reservierten  
Wortes → besonderes Token.

Parser liest Variablen token, name und pos.

Bemerkung: Fehlerbehandlung nur eingeschränkt möglich: Lexer kann  
z.B. **x987897** hier nicht als Zahl mit überschüssigem **X** erkennen.

# Literale in Java (1)

Notation aus Java-Sprachspezifikation

Literal:

IntegerLiteral:

DecimalIntegerLiteral:

DecimalNumeral IntegerTypeSuffix<sub>opt</sub>

HexIntegerLiteral:

HexNumeral IntegerTypeSuffix<sub>opt</sub>

OctalIntegerLiteral:

OctalNumeral IntegerTypeSuffix<sub>opt</sub>

FloatingPointLiteral

BooleanLiteral

CharacterLiteral

StringLiteral

NullLiteral

one of: I L

Für Abtastung relevantes Merkmal

DecimalNumeral:

0, 120, ...

0

NonZeroDigit Digits<sub>opt</sub>

HexNumeral:

0x0, 0XaF1, ...

0x

HexDigit

0X

HexDigit

HexNumeral HexDigit

OctalNumeral:

00, 0120, ...

0

OctalDigit

OctalNumeral OctalDigit

# Literale in Java (2)

Literal:

IntegerLiteral

FloatingPointLiteral:

Digits **•** Digits<sub>opt</sub> ExponentPart<sub>opt</sub> FloatTypeSuffix<sub>opt</sub>

**•** Digits ExponentPart<sub>opt</sub> FloatTypeSuffix<sub>opt</sub>

Digits ExponentPart **•** FloatTypeSuffix<sub>opt</sub>

Digits ExponentPart<sub>opt</sub> FloatTypeSuffix **•**

BooleanLiteral

CharacterLiteral

StringLiteral

NullLiteral

one of: f F d D

Für Abtastung relevantes Merkmal

ExponentPart:

ExponentIndicator SignedInteger

ExponentIndicator: one of

e E

SignedInteger:

Sign<sub>opt</sub> Digits

Sign: one of

+ -

# Lexer in javac (nicht generiert), Umgang mit Zahl-Literalen in nextToken()

```
case '0':  
    scanChar();  
    if (ch == 'x' || ch == 'X') {  
        scanChar();  
        if (Character.digit(ch, 16) < 0) {  
            lexError("hexadecimal numbers must contain at least one hexadec...");  
        }  
        scanNumber(16); //radix  
    } else {  
        putChar('0');  
        scanNumber(8);  
    }  
    return;  
case '1': case '2': case '3': ... case '7': case '8': case '9':  
    scanNumber(10);  
    return;  
case '.':  
    scanChar();  
    if ('0' <= ch && ch <= '9') {  
        putChar('.');  
        scanFractionAndSuffix();
```

Geht's mit einer 0 los?

Setzt token auf **INTLITERAL** oder **LOGLITERAL**, falls nicht **.**, **e**, **E**, **f**, **F**, **d** oder **D** gefunden wird. Dann **FLOATLITERAL** oder **DOUBLELITERAL**.

Setzt token auf **FLOATLITERAL** oder **DOUBLELITERAL**.

# Anekdote: Problemfälle in FORTRAN 77

- Problem in FORTRAN 77:
  - Zwischenräume sind *optional* (→ können sogar innerhalb von Token vorkommen).
  - Unklar, ob Zeichenfolge ein reserviertes Wort oder Bezeichner. Was ist das längste Präfix?
- Beispiel:
  - `DO 10 I = 1.5` ist Zuweisung an Variable `DO10I`
  - `DO10I = 1,5` ist Schleifensteuerung (Zähler `I`, Endmarke `10`)
- Verfahren:
  - Einlesen der gesamten Anweisung.
  - Anweisung beginnt mit einem reservierten Wort, wenn
    - Anweisung kein Gleichheitszeichen enthält,
    - nach einem Gleichheitszeichen ein Komma folgt,
    - ...
  - Andernfalls ist die Anweisung eine Zuweisung, die mit einem Bezeichner beginnt.

# Ein Lexer alleine reicht nicht aus!

- Frage: Warum reicht der Lexer nicht aus, um ein Programm in einer Programmiersprache zu akzeptieren?
- Antwort: *Reguläre Sprachen/DEAs* reichen nicht aus.
- Beispiel: Mit folgender Spezifikation kann man ausgeglichene Klammerung festlegen.

expr: „(“ expr „)“ | ...



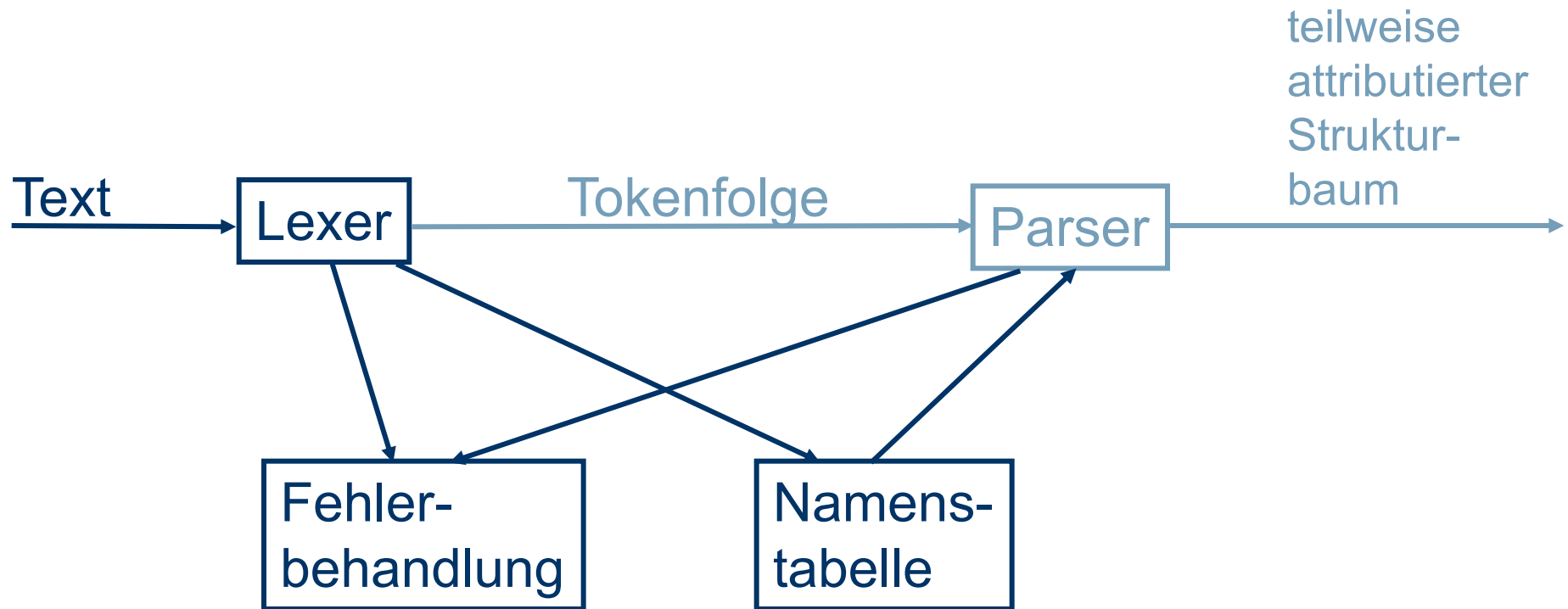
Rekursion!

- Kein *endlicher* Automat (mit n Zuständen) kann sich beliebige Klammeranzahlen „merken“ (>n)
  - Korrekte Klammerung in Ausdrücken und Programmen („{ ... }“, „**begin** ... **end**“, ...)
  - Korrekte Schachtelung von Programmkonstrukten



- Aufgaben:
  - Feststellung der bedeutungstragenden Elemente
  - Zuordnung statischer Bedeutung
  - Konsistenzprüfung
- Schritte:
  - Lexikalische Analyse/Abtastung (Lexer)  
→ Syntaktische Analyse/Zerteilung (Parser)
  - Semantische Analyse
- Noch keine Übersetzung möglich, nur Analyse

# Zusammenspiel zwischen Lexer und Parser



# Warum getrennter Lexer?

## Kann man nicht alles mit dem Parser machen?

- Separater, effizienter Lexer lohnt sich, weil höhere „Kompression“ als in anderen Teilen des Übersetzers.
  - Beispielanweisung: `var := var + const;`  
→ 6 Token, aber 19 Zeichen
  - Potenziell viele Leerzeichen pro Zeile wegen Einrückung und Kommentaren.
- Parser wird erheblich komplizierter, wenn z.B. Regeln für Kommentare, Leerräume etc. in der Grammatik festgelegt werden müssen.
- DEA schneller als der für Parser benötigte Kellerautomat.
- Portabilität steigt: Der Lexer kann sich z.B. mit Zeichensatz des Eingabetextes auseinandersetzen.

- Ein Parser (Zerteiler) analysiert die in der Tokenfolge enthaltene Struktur.
  - Grundlage: Kontextfreie Grammatiken
  - Parser entscheidet, ob eine Tokenfolge zu der Sprache gehört, die durch die Grammatik definiert ist.
  - Dabei baut der Parser (implizit oder explizit) den zugehörigen konkreten Syntaxbaum für die Eingabe auf.
- Analog zu Lexer-Generatoren: Es gibt Werkzeuge zur Generierung effizienter Parser aus Spezifikationen.

# Kontextfreie Grammatiken im Überflug

- Eine Grammatik besteht aus Produktionen, die über Terminalen und Nicht-Terminalen definiert sind.
  - Terminale entsprechen den vom Lexer generierten Token
    - Atomar → Keine innere Struktur
  - Nicht-Terminale bilden die verschiedenen Sprachkonstrukte ab
    - Locker: Jedes Sprachkonstrukt entspricht einem Nicht-Terminal in der Grammatik (+ ggf. notwendige „Hilfs-Nicht-Terminale“)
  - Produktionen setzen die (Nicht-)Terminale zueinander in Beziehung und beschreiben so die synt. Regeln der Sprache
  - Kontextfreie Grammatik: Auf der „linken Seite“ jeder Produktion steht nur *ein* Nicht-Terminal, auf der „rechten Seite“ eine beliebige Folge von Terminalen und Nicht-Terminalen.
- Locker: Grammatik legt fest, in welcher „Reihenfolge“ Token in der Tokenfolge vorkommen dürfen.

# Kontextfreie Grammatiken: Beispiel

- Beispiel für eine einfache kontextfreie Grammatik:

$s \rightarrow s \text{ „;“ } \text{asgn}$

$s \rightarrow \text{asgn}$

Nicht-Terminal

$\text{asgn} \rightarrow \text{ID } \text{„=“ } \text{addexpr}$

$\text{addexpr} \rightarrow \text{addexpr } \text{„+“ } \text{term}$

$\text{addexpr} \rightarrow \text{term}$

Alternative Produktionen für  
das Nicht-Terminal  $\text{addexpr}$

$\text{term} \rightarrow \text{NUMBER}$

$\text{term} \rightarrow \text{ID}$

Terminal



Nicht-Terminal

# (Konkreter) Syntaxbaum: Beispiel 1

- $x = 13 + y ; y = 3$

ID = NUMBER + ID ; ID = NUMBER

- $s \rightarrow s \text{ „;“ asgn}$

$s \rightarrow \text{asgn}$

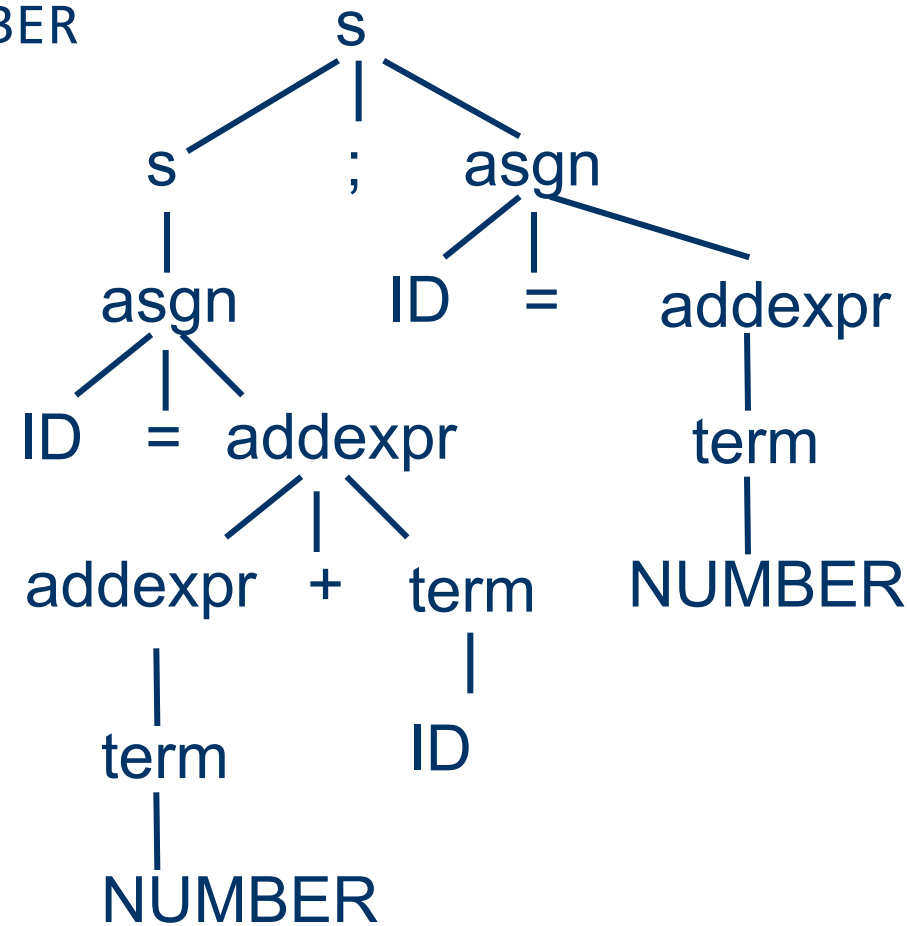
$\text{asgn} \rightarrow \text{ID „=“ addexpr}$

$\text{addexpr} \rightarrow \text{addexpr „+“ term}$

$\text{addexpr} \rightarrow \text{term}$

$\text{term} \rightarrow \text{NUMBER}$

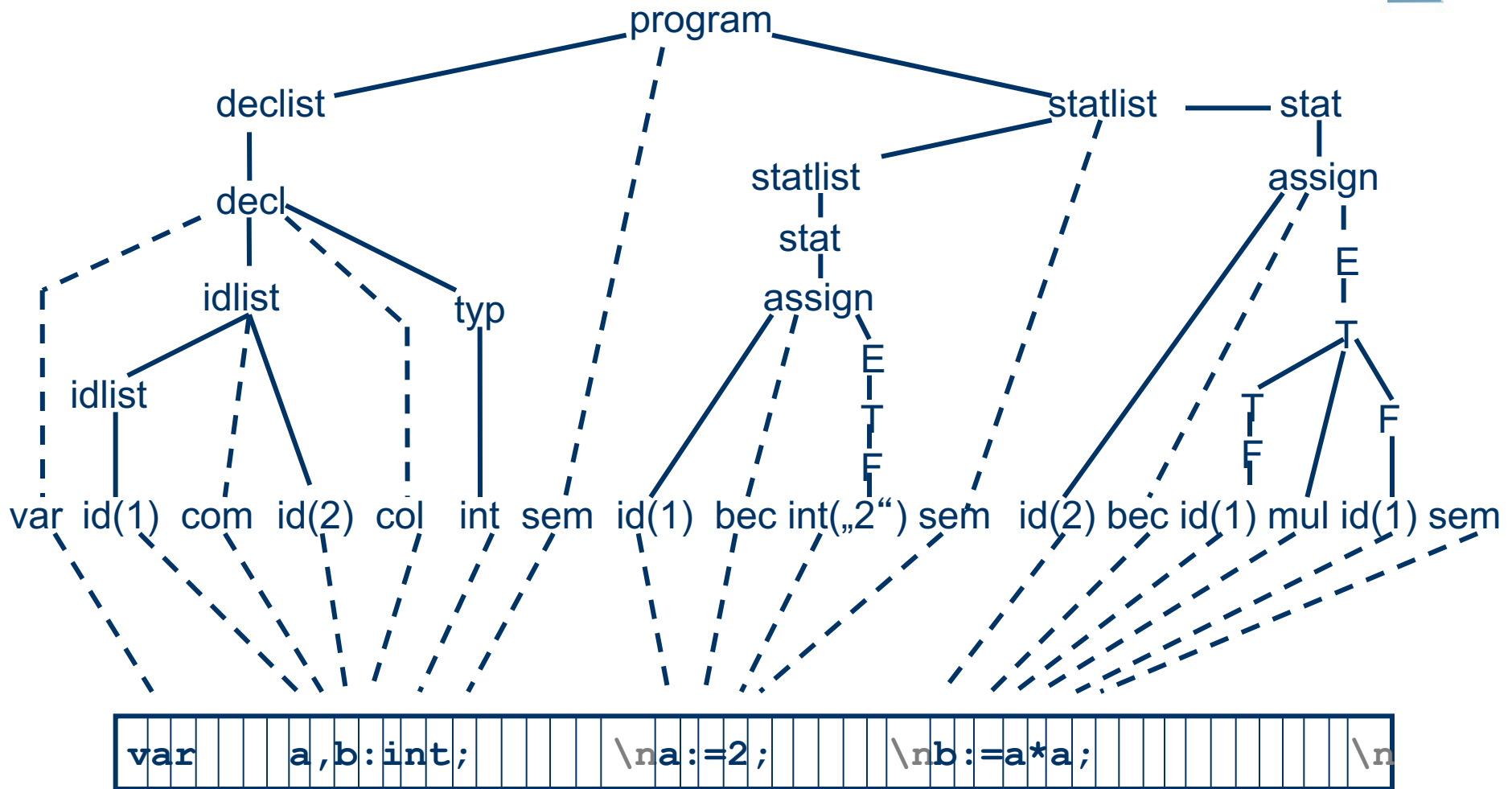
$\text{term} \rightarrow \text{ID}$



Ein Blatt pro Terminalsymbol.

Ein innerer Knoten pro angewendeter Produktion.

# (Konkreter) Syntaxbaum: Beispiel 2



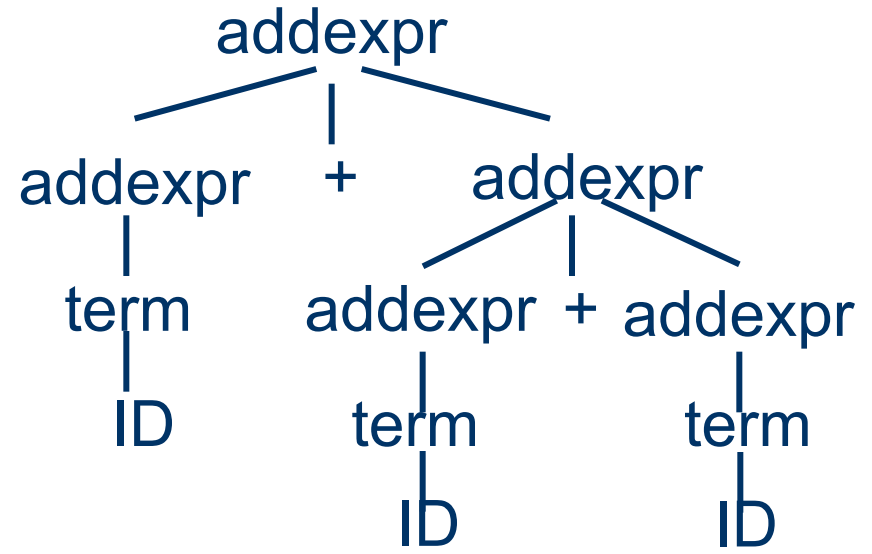
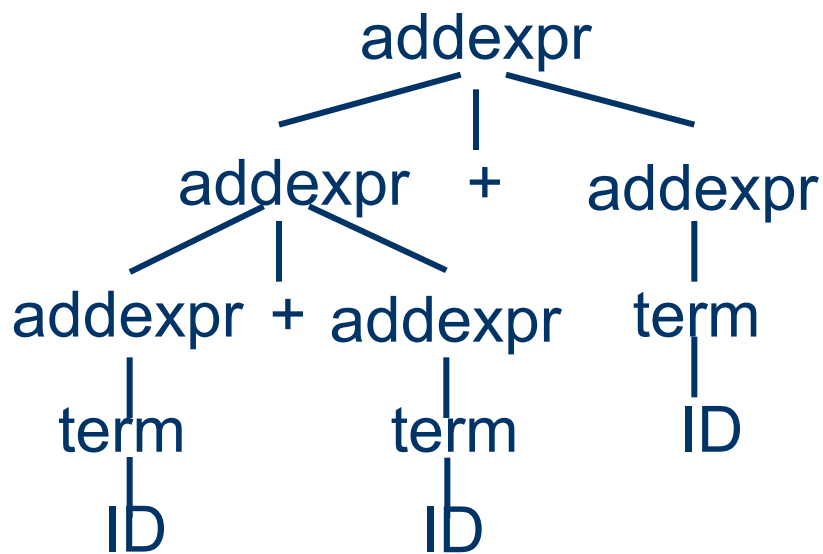


```
abstract class Parser is
    initialisiere();
    beende();
    zerteile(Tokenfolge);
end;
```

- Benötigte Operationen
  - Lexer/Tokenfolge:  
    naechstesToken
  - Fehlerbehandlung  
    Fehlereintrag(Nr, pos)
  - Für den Aufbau des Syntaxbaums  
    produktion(nr)

# Mehrdeutige Grammatiken

- Eine Grammatik heißt **mehrdeutig**, wenn es mehrere verschiedene Syntaxbäume zu einer Tokenfolge gibt.
- Beispiel:
  - $\text{addexpr} \rightarrow \text{addexpr} \text{ „+“ } \text{addexpr}$  (Rest wie vorher)
  - (Relevanter Teil der) Eingabe:  $\text{ID} + \text{ID} + \text{ID}$



# Eindeutige Grammatiken (1)

- Zu Programmiersprachen sollte es immer eindeutige Grammatiken geben. Andernfalls ist unklar, was mit einem Programm gemeint ist.
- Beispiel: if-Anweisung *ohne Regelung* in Sprachdefinition

```
if () then ... if () then ... else ...
```



```
if () then ... if () then ... else ...
```



Sog. „dangling else“-Problem

Das ist die typische Regelung in C, Java, ...

Es ist *ohne Regelung* nicht klar, zu welchem if ein else gehört.

Alternative: zwingend Klammern, **fi**, **endif**, ...

# Eindeutige Grammatiken (2)

- Prinzipiell:
  - Oft kann man zu einer mehrdeutigen Grammatik eine eindeutige Grammatik angeben, die dieselbe Sprache beschreibt.
  - Es gibt Sprachen, die nur von mehrdeutigen Grammatiken beschrieben werden können.
- Manche Parser(-generatoren) behandeln „Assoziativität“:
  - Mehrdeutigkeit wird in einer bestimmten Richtung aufgelöst.
  - Beispiel
    - „ $\text{id} := \text{id} + \text{id} + \text{id}$ “ wird immer wie „ $\text{id} := \text{id} + \text{id} + \text{id}$ “ zerteilt.



- Man unterscheidet „links-assoziativ“ und „rechts-assoziativ“.

- Ziel: Parser sollte Eingabe in linearer Zeit verarbeiten
- Nur für Unterklassen von eindeutigen kontextfreien Grammatiken möglich!
- Für Programmiersprachen wichtigste Unterklassen:
  - LL(k):
    - Leserichtung Links→rechts; Linksableitung
    - Vorausschau von k Token
    - z.B. Zerteiler mit rekursivem Abstieg, zerteilen „top-down“
  - LR(k):
    - Leserichtung Links→rechts; Rechtsableitung
    - Vorausschau von k Token
    - Speichereffiziente Unterklasse „Look-Ahead LR“: LALR(1)
    - z.B. tabellengesteuerte Zerteiler, zerteilen „bottom-up“

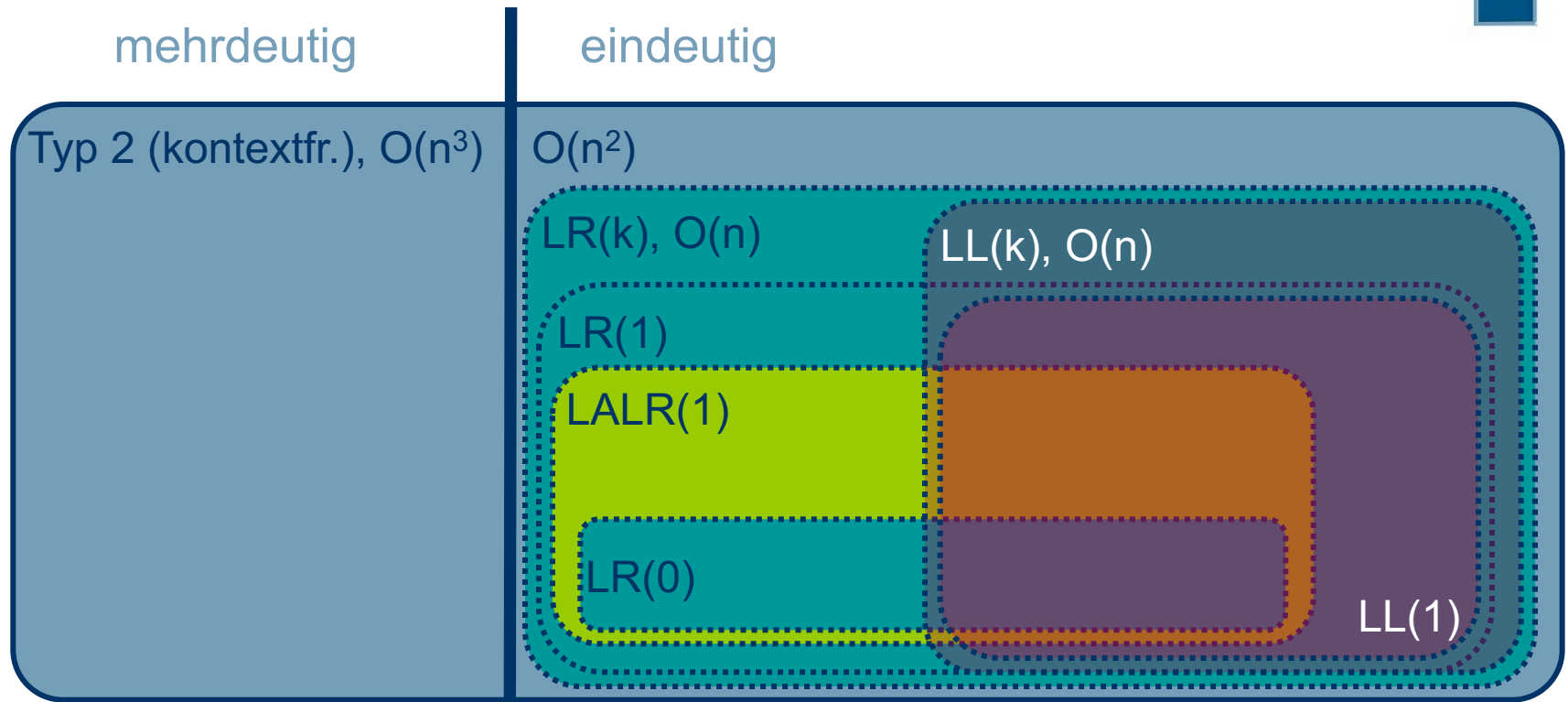
# Beziehungen zwischen Grammatik-Klassen (1)

- In Bezug auf die *Grammatik* gilt stets
  - $LR(0) \subset LALR(1) \subset LR(1) \subset \dots \subset LR(k)$
  - $LL(1) \subset \dots \subset LL(k)$

PDA: push-down automaton  
DPDA: deterministic  
push-down automaton

- Für die *akzeptierten Sprachen* hingegen gilt:
  - $\mathcal{L}(LL(1)) \subset \dots \subset \mathcal{L}(LL(k)) \subset \mathcal{L}(LR(k)) = \mathcal{L}(DPDA) \subset \mathcal{L}(PDA)$
  - $\mathcal{L}(LR(0)) \subset \mathcal{L}(LR(1)) = \mathcal{L}(LR(2)) = \dots = \mathcal{L}(LR(k)) = \mathcal{L}(LR(k))$

# Beziehungen zwischen Grammatik-Klassen (2)



$$LR(0) \subset LALR(1) \subset LR(1) \subset \dots \subset LR(k)$$

Weitere Informationen:

<http://amor.cms.hu-berlin.de/~kunert/papers/lr-analyse/lr.pdf>

# Rekursiver Abstieg: Parser in javac (1)

```
/** CompilationUnit = [PACKAGE Qualident ";" ] optional  
                        {ImportDeclaration} {TypeDeclaration}  
*/
```

0-n mal

```
public Tree.TopLevel compilationUnit() {  
    int pos = S.pos;  
    Tree pid = null;  
    if (S.token == PACKAGE) {  
        S.nextToken();  
        pid = qualident();  
        accept(SEMI);  
    }  
    ListBuffer<Tree> defs = new ListBuffer<Tree>();  
    while (S.token == IMPORT) defs.append(importDeclaration());  
    while (S.token != EOF) defs.append(typeDeclaration());  
    return F.at(pos).TopLevel(pid, defs.toList());  
}
```

Scanner (Lexer)

1 Token look-ahead

Rekursiver Abstieg

Tree-Fabrik



## Einschub: Fabrik?

- Das Entwurfsmuster Fabrik entkoppelt den Parser von den Tree-Unterklassen.
- Man kann die Implementierung der Tree-Unterklassen ändern, ohne den Parser ebenfalls ändern zu müssen.
- Der Parser gibt der Fabrik nur diejenigen Tree-Attribute, die für ihn wichtig sind. Die Fabrik ergänzt vor dem Aufruf der Konstruktoren Default-Werte für andere Konstruktorparameter.
  - Der Parser braucht also nichts über die weiteren Konstruktorparameter und deren geeignete Initialisierung zu wissen.
  - Die Tree-Unterklassen brauchen keine Konstruktoren speziell für den Parser vorzuhalten.

## Rekursiver Abstieg: Parser in javac (2)

```
/** ImportDeclaration = IMPORT Ident { "." Ident }
                                [ "." "*" ] ";"

 */
Tree importDeclaration() {
    int pos = S.pos;
    S.nextToken();
    Tree pid = F.at(S.pos).Ident(ident());
    boolean starImport = false;
    while (S.token == DOT && !starImport) {
        S.nextToken();
        if (S.token == STAR) {
            pid = F.at(S.pos).Select(pid, Names.star);
            S.nextToken();
            starImport = true;
        } else {
            pid = F.at(S.pos).Select(pid, ident());
        }
    }
    accept(SEMI);
    return F.at(pos).Import(pid);
}
```

# Rekursiver Abstieg: Parser in javac (3)

```
/** ClassOrInterfaceDeclaration = ModifiersOpt  
 *  
 * (ClassDeclaration | InterfaceDeclaration)  
 */
```

```
Tree classOrInterfaceDeclaration(int flags) {  
    flags = flags | modifiersOpt();  
    if (S.token == CLASS)  
        return classDeclaration(flags);  
    else if (S.token == INTERFACE)  
        return interfaceDeclaration(flags);  
    else  
        return syntaxError("'class' or 'interface' expected");  
}
```

 Fehlerbehandlung

# Rekursiver Abstieg: Parser in javac (4)

// Fehlerbehandlung: Wenn ein bestimmtes Token (z.B. Semikolon) fehlt, Fehler melden und alle Token überspringen, bis ein „Stop-Token“ gefunden wurde, ab dem Parser fortsetzen kann:

```
private void skip() {
    int nbraces = 0; int nparens = 0;
    while (true) {
        switch (S.token) {
            case EOF: case CLASS: case INTERFACE:
                return;
            case SEMI:
                if ( nbraces == 0
                    && nparens == 0)
                    return;
                break;
            case RBRACE:
                if (nbraces == 0) return;
                nbraces--;
                break;
            case RPAREN:
                if (nparens > 0) nparens--;
                break;
            case LBRACE:
                nbraces++;
                break;
            case LPAREN:
                nparens++;
                break;
            default:
                S.nextToken();
        }
    }
}
```

# Shift-Reduce-Parser

- Füge neues Startsymbol  $s'$  und eine EOF-Marke  $\$$  ein.

- Beispiel:

$s' \rightarrow s\$$

$s \rightarrow s \text{ „;“ asgn}$

$s \rightarrow \text{asgn}$

$\text{asgn} \rightarrow \text{ID „=“ addexpr}$

$\text{addexpr} \rightarrow \text{addexpr „+“ term}$

$\text{addexpr} \rightarrow \text{term}$

$\text{term} \rightarrow \text{NUMBER}$

$\text{term} \rightarrow \text{ID}$

- Zum Akzeptieren wähle nach Betrachtung der nächsten  $k$  Eingabesymbole (look-ahead) einen der folgenden Schritte:
  - „shift“: Schiebe das nächste Eingabesymbol auf Stapel.
  - „reduce“: Ersetze ein  $abc$  oben auf dem Stapel durch ein  $X$ , wenn es eine Produktion  $X \rightarrow cba$  gibt.
  - Die ganze Eingabe ist Bestandteil der Sprache, wenn  $s\$$  reduziert wurde.

# Shift-Reduce mit look-ahead(1) am Beispiel

Grammatik:

$s' \rightarrow s\$$

$s \rightarrow s \text{ „;“ asgn}$

$s \rightarrow \text{asgn}$

$\text{asgn} \rightarrow \text{ID „=“ addexpr}$

$\text{addexpr} \rightarrow \text{addexpr „+“ term}$

$\text{addexpr} \rightarrow \text{term}$

$\text{term} \rightarrow \text{NUMBER}$

$\text{term} \rightarrow \text{ID}$

	$x=13+y; y=3\$$	shift
ID	$=13+y; y=3\$$	shift
ID=	$13+y; y=3\$$	shift
ID=NUM	$+y; y=3\$$	$\text{term} \rightarrow \text{NUM}$
ID=term	$+y; y=3\$$	$\text{addexpr} \rightarrow \text{term}$
ID=addexpr	$+y; y=3\$$	shift
ID=addexpr+	$y; y=3\$$	shift
ID=addexpr+ID	$; y=3\$$	$\text{term} \rightarrow \text{ID}$
ID=addexpr+term	$; y=3\$$	$\text{addexpr} \rightarrow \text{addexpr+term}$
ID=addexpr	$; y=3\$$	$\text{asgn} \rightarrow \text{ID=addexpr}$
asgn	$; y=3\$$	$s \rightarrow \text{asgn}$
s	$; y=3\$$	shift
s;	$y=3\$$	shift
s; ID	$=3\$$	shift
s; ID=	$3\$$	shift
s; ID=NUM	$\$$	$\text{term} \rightarrow \text{NUM}$
s; ID=term	$\$$	$\text{addexpr} \rightarrow \text{term}$
s; ID=addexpr	$\$$	$\text{asgn} \rightarrow \text{ID=addexpr}$
s; asgn	$\$$	$s \rightarrow s;\text{asgn}$
s	$\$$	shift
s\$		$s' \rightarrow s\$$
s`		accept

# Shift-Reduce-Konflikte

- Im Beispiel war immer „klar“, ob „shift“ oder „reduce“ notwendig war.
- Bei mehrdeutigen Grammatiken kommt es zu sog. „shift-reduce“- oder „reduce-reduce“-Konflikten.
  - Werkzeuge geben diese Konflikte als Warnungen aus.
  - Das Auflösen der Konflikte durch Umschreiben der Grammatik ist dann Aufgabe des Benutzers.

# Tabellen-Repräsentation für LR-Parser

	Aktionen				„goto“-Regeln			
	Tok <sub>1</sub>	Tok <sub>2</sub>	Tok <sub>3</sub>	...	NT <sub>1</sub>	NT <sub>2</sub>	NT <sub>3</sub>	...
Zust <sub>1</sub>						g <sub>2</sub>		
Zust <sub>2</sub>		s <sub>3</sub>						
Zust <sub>3</sub>			r <sub>1</sub>					
...								

- LR-Parser verwaltet Stapel mit *Zuständen* (anders als Kellerautomat)
- s<sub>i</sub> „shift“
  - Konsumiere Token, lege Zustand i auf Stapel
- r<sub>k</sub> „reduce“ für k-te Regel: NT<sub>x</sub> → a<sub>1</sub>..a<sub>m</sub>
  - Nimm m Zustände vom Stapel (j sei neuer oberster Zustand)
  - Führe „goto“ g<sub>y</sub> aus Tabelleneintrag für (j, NT<sub>x</sub>) aus
- g<sub>i</sub> „goto“
  - Lege Zustand i auf Stapel
- Alle anderen Einträge: Fehler



# Shift-Reduce am Beispiel

```

0      x=13+y; y=3$
0 1    =13+y; y=3$
0 1 4   13+y; y=3$
0 1 4 8  +y; y=3$
0 1 4 10 +y; y=3$
0 1 4 9  +y; y=3$
0 1 4 9 12 y; y=3$
0 1 4 9 12 7 ; y=3$
0 1 4 9 12 13 ; y=3$
0 1 4 9 ; y=3$
0 3      ; y=3$
...

```

1.  $s \rightarrow \text{asgn}$
2.  $s \rightarrow s ; \text{asgn}$
3.  $\text{asgn} \rightarrow \text{ID} = \text{addexpr}$
4.  $\text{addexpr} \rightarrow \text{addexpr} + \text{term}$
5.  $\text{addexpr} \rightarrow \text{term}$
6.  $\text{term} \rightarrow \text{NUM}$
7.  $\text{term} \rightarrow \text{ID}$

	ID	NUM	;	+	=	\$	s	asgn	term	addexpr
0	s1						g2	g3		
1					s4					
2			s6			acc				
3			r1			r1				
4	s7	s8							g10	g9
5										
6	s1							g11		
7			r7	r7		r7				
8			r6	r6		r6				
9			r3	s12		r3				
10			r5	r5		r5				
11			r2			r2				
12	s7	s8							g13	
13			r4	r4		r4				

- Es gibt Werkzeuge, die aus einer gegebenen Spezifikation einer kontextfreien Grammatik Code erzeugen, der einen effizienten Parser für diese Sprache realisiert.
- Theorie ist so gut erforscht,
  - dass es heute zahlreiche Werkzeuge gibt, die einfach verwendbar sind und die effiziente Parser generieren können
    - Meldung von Konflikten in der Grammatik
    - Konfliktauflösung
    - Fehlerbehandlung
    - ...
    - → In vielen Fällen sind generierte Parser das Mittel der Wahl
  - dass wir hier das „Wie“ der Tabellenkonstruktion nicht besprechen.

# Beispiel: Generierung von (Lexer und) Parser mit ANTLR

Sample.g4

antlr

SampleParser.java

javac

SampleParser.class

```
assignStatement
    : lvalue ASSIGN addExpression SEMICOLON
    ;

ifStatement
    : IF orExpression THEN block
      ( ELSE block )? END
    ;

identifier
    : IDENTIFIER
    ;

lvalue
    : identifier | arrayAccess
    ;

orExpression
    : andExpression ( OR andExpression )*
    ;

addExpression
    : multExpression ( ( PLUS | MINUS ) multExpression )*
    ;
```

Spezifikation der Regeln in Erweiterter Backus-Naur-Form (EBNF)

# Grammatikschwierigkeiten von Java

## Zu spezifische Namen (1)

- Beispiel von Produktionen-Gruppen in der Java-Spezifikation:

TypeName:	AmbiguousName:
Identifier	Identifier
PackageName . Identifier	AmbiguousName . Identifier
PackageName:	MethodName:
Identifier	Identifier
PackageName . Identifier	AmbiguousName . Identifier

```
class Problem1 {int m() { foo.
```



- Parser kann bei Untersuchung von `foo` mit einem Token Vorausschau (.) nicht entscheiden, ob `foo`

- ein PackageName `foo.Bar ref = new foo.Bar(2)`
- oder ein AmbiguousName `foo.print("Hello World!");`

ist.

## Zu spezifische Namen (2)

- Lösung: Statt `PackageName`, `TypeName`, `ExpressionName`, `MethodName`, `AmbiguousName` lieber nur eine Namenssorte.

Name:

`SimpleName`

`QualifiedName`

`SimpleName`:

`Identifier`

`QualifiedName`:

`Name . Identifier`

- Damit ist dieser Teil der Grammatik eindeutig. Spätere Übersetzerphasen müssen sich damit auseinandersetzen, welche Rolle die Namen wirklich spielen.

# Grammatikschwierigkeiten von Java

## Zu spezifische Modifikatoren (1)

- Beispiel von Produktionen-Gruppen in der Java-Spezifikation:

- FieldDeclaration:

FieldModifiers<sub>opt</sub> Type VariableDeclarators ;  
**public protected private final static**  
**transient volatile**

- MethodHeader:

MethodModifiers<sub>opt</sub> ResultType MethodDeclarator Throws<sub>opt</sub>  
**public protected private final static**  
**abstract native synchronized**

```
class Problem2 { public static int
```



- Parser kann bei Untersuchung von **public** mit einem Token Vorausschau (**static**) nicht entscheiden, ob

- eine Feld-Deklaration

```
int foo = 0;
```

- oder eine Methoden-Deklaration

```
int foo(String a) {...}
```

folgen wird (und zu welcher Modifier-Sorte reduziert werden soll.)

- Lösung: Statt ClassModifiers, FieldModifiers, MethodModifiers, ConstructorModifiers, InterfaceModifiers, ConstantModifiers lieber nur eine Sorte von Modifikatoren.

Modifiers:

Modifier

Modifiers Modifier

Modifier: one of

`public protected private final static abstract`  
`native synchronized transient volatile`

- Damit ist dieser Teil der Grammatik (auch bei einem Token Vorausschau) eindeutig. Spätere Übersetzerphasen müssen sich damit auseinandersetzen, welche der akzeptierten Modifikatoren an einer Stelle zulässig sind.

# Grammatikschwierigkeiten von Java

## Felddeklaration vs. Methodendeklaration (1)

- Beispiel von Produktionen-Gruppen in der Java-Spezifikation:

- FieldDeclaration:

~~Field~~Modifiers<sub>opt</sub> Type VariableDeclarators ;

- MethodHeader:

~~Method~~Modifiers<sub>opt</sub> ResultType MethodDeclarator Throws<sub>opt</sub>

`class Problem3 { int foo`



Hier ist auch `void` zulässig.

- Parser kann bei Untersuchung von `int` mit einem Token Vorausschau (`foo`) nicht entscheiden, ob

- `int` Type ist (Feld-Deklaration)      `int foo = 0;`
  - oder ResultType (Methoden-Dekl.).    `int foo(String a) {...}`



# Grammatikschwierigkeiten von Java

## Felddeklaration vs. Methodendeklaration (2)

- Lösung: Entfernung des ResultTyps aus der Grammatik. Stattdessen explizite Behandlung des `void`-Falls.
  - FieldDeclaration:  
~~Field~~Modifiers<sub>opt</sub> Type VariableDeclarators ;
  - MethodHeader:  
~~Method~~Modifiers<sub>opt</sub> `void` MethodDeclarator Throws<sub>opt</sub>  
~~Method~~Modifiers<sub>opt</sub> Type MethodDeclarator Throws<sub>opt</sub>
- Die Entscheidung, ob eine Feld- oder Methodendeklaration vorliegt, fällt später im Parser.

# Grammatikschwierigkeiten von Java

## Array-Typ vs. Array-Zugriff (1)

- Beispiel von Produktionen-Gruppen in der Java-Spezifikation:

- ArrayType:

Type [ ]

- ArrayAccess:

Name [ Expression ]

PrimaryNoNewArray [ Expression ]

Ausdrücke außer `new Typ [...]`, weil `new Typ [...][...]` sonst sowohl Erzeugung eines zweidim. Arrays als auch Zugriff auf ein gerade erstelltes eindim. Array sein kann.

```
class Problem4 { Problem() { foo[  
                                ↑
```

- Parser kann bei Untersuchung von `foo` mit einem Token Vorausschau ( `[` ) nicht entscheiden, ob

- `foo` Typname ist `foo[] bar;`
- oder Arrayzugriff („Name“). `foo[3]=12;`

- Lösung: „Name“ auch in ArrayType zulassen.
  - ArrayType:
    - PrimitiveType [ ]
    - Name [ ]
    - ArrayType [ ]
  - ArrayAccess:
    - Name [ Expression ]
    - PrimaryNoNewArray [ Expression ]
- Parser akzeptiert `foo` als „Name“ und entscheidet später, ob es sich um einen Typnamen oder um einen Array-Zugriff handelt.

# Grammatikschwierigkeiten von Java

## Typwandlung vs. Klammerausdruck (1)

- Typumwandlungsproduktion:

- CastExpression:
  - ( PrimitiveType ) UnaryExpression
  - ( ReferenceType ) UnaryExpressionNotPlusMinus

```
class Problem5 { Problem5() { super((foo)
```



- Zerteiler kann bei Untersuchung von `foo` mit einem Token Vorausschau ( ) nicht entscheiden, ob

- `foo` geklammerter Ausdruck ist `super ( (foo) , 9)`
- oder Typwandlung. `super ( (foo) baz , 9)`

Entweder wird „Name“ über „PostfixExpression“ zu „Expression“ oder über „ClassOrInterfaceType“ zu „ReferenceType“.

# Grammatikschwierigkeiten von Java

## Typwandlung vs. Klammerausdruck (2)

- Lösung: ReferenceType in CastExpression vermeiden.
  - CastExpression:
    - ( PrimitiveType Dims<sub>opt</sub>) UnaryExpression
    - ( Expression ) UnaryExpressionNotPlusMinus
    - ( Name Dims ) UnaryExpressionNotPlusMinus
- Damit kann der Parser `foo` zu „Expression“ machen und die Entscheidung aufschieben.
- Illegal akzeptierte Ausdrücke wie
  - `(int[])+3`
  - `(foo+1)baz`muss der Übersetzer in der semantischen Analyse entdecken.

# Modulare Struktur von Übersetzern

## Analyse

*Token*

*Syntax*

Semantik

## Abbildung

Transf.

Optim.

## Codierung

Code-Erz.

Ass./Bind.

*Quelltext*

*Token-  
folge*

Struktur-  
baum

attrib.  
Struktur-  
baum

Zwischen-  
Code

Ziel-  
programm

*Namens-  
tabelle*

Definitions-  
tabelle