



Andreas Koch  
Fachgebiet Eingebettete Systeme und ihre Anwendungen  
Fachbereich Informatik



# Kompilierung



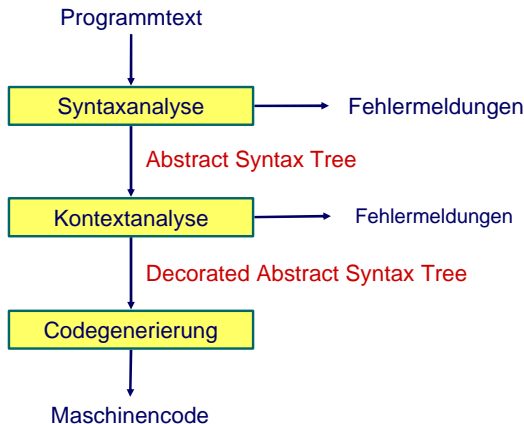
## Terminologie: Phase

- Transformationsschritte
  - ▣ Von Quellcode
  - ▣ ... zum Maschinencode
  
- Entspricht häufig den Teilen der Sprachspezifikation
  1. Syntax → Syntaxanalyse
  2. Kontextuelle Einschränkungen → Kontextanalyse
  3. Semantik → Codegenerierung

# Ablauf der Übersetzung 2



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

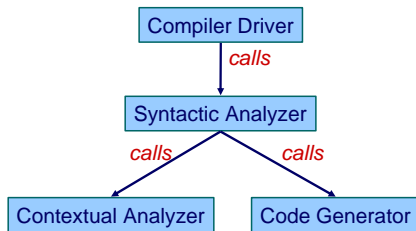




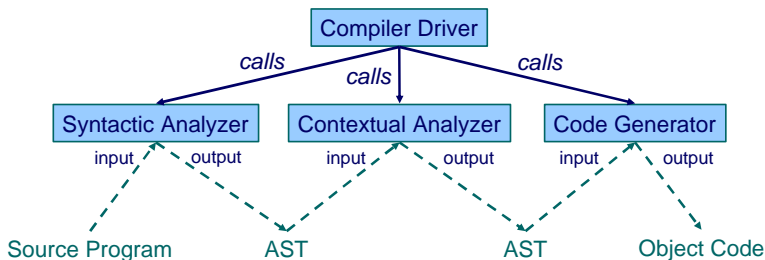
## Terminologie: Durchgang (*pass*)

- Kompletter Durchgang des Programmes
- Läuft über Quelltext oder IR
- Pass *kann* Phase entsprechen
- ... muss aber nicht!
- Einzelner Pass kann mehrere Phasen durchführen
- Aufbau des Compilers wird von der Anzahl der Passes dominiert

- Macht nur **einen** Pass über den Quelltext
  - ▢ Baut in der Regel **keine** echte IR auf
- Führt gleichzeitig aus
  - ▢ Syntaxanalyse (Parsing)
  - ▢ Kontextanalyse
  - ▢ Codegenerierung
- Pascal Compiler haben häufig Ein-Pass-Struktur



- Macht mehrere Passes über das Program
  - ▣ Quelltext und IR
- Datenweitergabe zwischen Passes über IR



# Vergleich Ein-Pass ./ Multi-Pass-Compiler



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

	Ein-Pass	Multi-Pass
Laufzeit	+	-
Speicher	+ für große Prog.	+ für kleine Prog.
Modularität	-	+
Flexibilität	-	+
Globale Optim.	--	+
Eingabesprachen	Nicht für alle	

Müssen Bezeichner vor Verwendung  
deklariert werden?



Java-Compilierung **erfordert** mehrere Passes

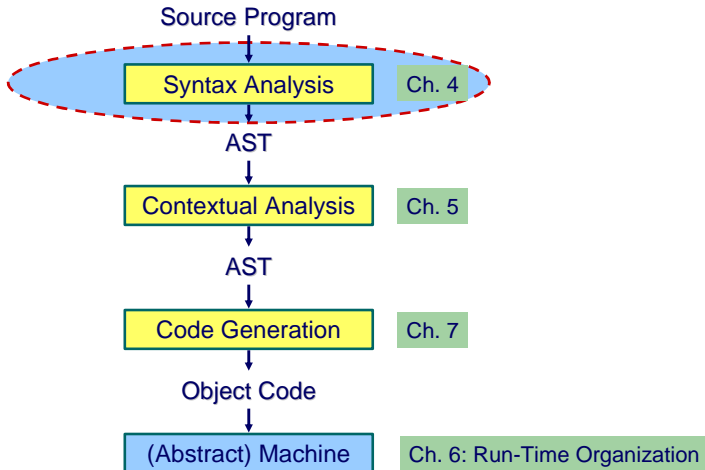
```
class Example {  
    void inc() { n = n + 1; }  
    int n;  
    void use() { n = 0; inc();}  
}
```

Beachte Reihenfolge Verwendung/Bindung von **n**!



- Ein-Pass wäre für Triangle möglich
- Aus pädagogischen Gründen aber Multi-Pass

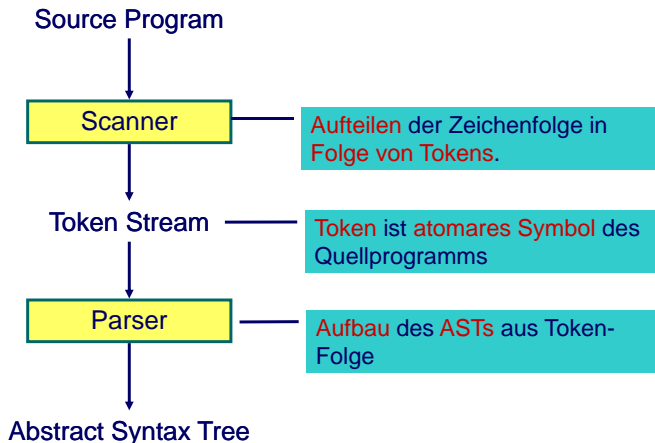
```
public class Compiler {  
    public static boolean compileProgram(...) {  
        Scanner scanner = new Scanner(...);  
        Parser parser = new Parser(...);  
        Checker checker = new Checker(...);  
        Encoder encoder = new Encoder(...);  
        ...  
        Program theAST = parser.parse();  
        checker.check(theAST);  
        encoder.encode(theAST);  
        ...  
    }  
  
    public static void main(String[] args) {  
        ... compileProgram(...); ...  
    }  
}
```



# Subphasen der Syntaxanalyse



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



## Beispielprogramm in Triangle

```
! Groesster Gemeinsamer Teiler
let func gcd(x: Integer, y: Integer) : Integer ~
    if x // y = 0                ! // -> Modulo
    then y
    else gcd(y, x // y);
in  putint(gcd(321,81))
```

Token-Folge: Ohne Leerzeichen, Zeilenvorschub und Kommentare

```
let func gcd ( x : Integer , y : Integer )
: Integer ~ Integer if x // y = 0 then y
else gcd ( y , x // y ) ; in putint ( gcd
( 321 , 81 ) )
```

- **Token** ist atomares Symbol des Programms
- Verwendet zwischen Scanner und Parser
- Kann auch aus mehreren Zeichen bestehen
- Zeichen selbst i.d.R. uninteressant, Ausnahmen:
  - ▢ Bezeichnernamen
  - ▢ Konstante Werte (Zahlen, Zeichen), sog. *Literale*
- ...Parser ist nur an der **Art** des Tokens interessiert

```
class Token {  
    TokenKind      kind;      // enum TokenKind {...}  
    String          spelling;  
    SourcePosition position; // Zeilennummer, Spalte  
}
```

# Aufzählung für Token-Arten (Auszug)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
enum TokenKind {
    // literals, identifiers, operators...
    INTLITERAL ( 0, "<int>"),
    IDENTIFIER ( 2, "<identifier>"),
    OPERATOR   ( 3, "<operator>"),

    // reserved words
    BEGIN      ( 5, "begin"),
    CONST      ( 6, "const"),
    DO         ( 7, "do"),
    ELSE       ( 8, "else"),
    END        ( 9, "end"),
    IF         (11, "if"),
    IN         (12, "in"),
    LET        (13, "let"),
    THEN       (17, "then"),
    VAR        (19, "var"),
    WHILE      (20, "while"),

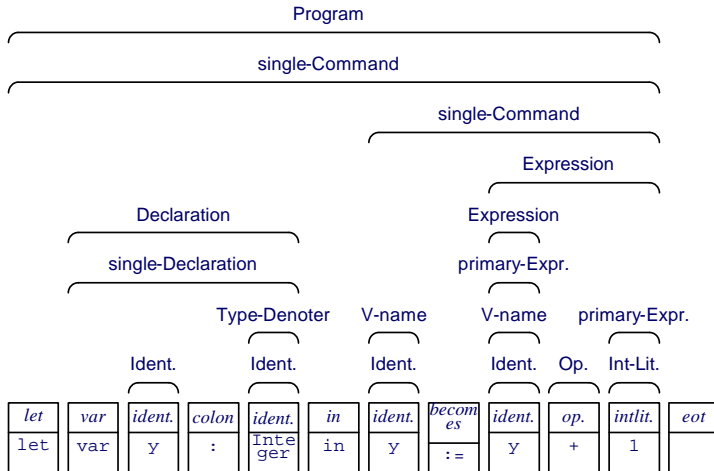
    // punctuation and brackets
    SEMICOLON  (23, ";"),
    COMMA      (24, ","),
    BECOMES    (25, "!="),
    IS         (26, "~"),
    LPAREN     (27, "("),
    RPAREN     (28, ")"),
    ...
    final int   id;
    final String spelling;
    ...
}
```

Beispiel: `t = new Token(TokenKind.OPERATOR, "+", position);`

# Parsen der Token-Folge



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

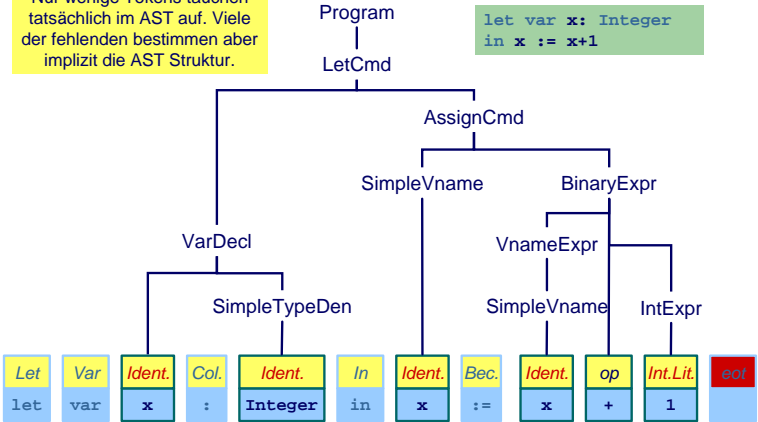




# Aufbau des AST aus Token-Folge

Nur wenige Tokens tauchen tatsächlich im AST auf. Viele der fehlenden bestimmen aber implizit die AST Struktur.

```
let var x: Integer
in x := x+1
```





- Kontextfreie Grammatiken (CFG)
- Spezifiziert durch  $(N, T, P, S)$
- CFG definiert Menge von Zeichenketten
  - ▢ Elemente sind *Sätze* bestehend aus Terminalsymbolen
  - ▢ Gesamtmenge ist *Sprache* der CFG
- Hier: Sätze haben eindeutige Phrasenstruktur
- $P$  häufig in Backus-Naur-Form (BNF) angegeben
- Übersichtlicher: Extended BNF
  - ▢ BNF + Reguläre Ausdrücke auf rechter Seite der Produktionen

# Beispiel: Produktionen in EBNF



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

BNF

```
Program    ::= single-Command
Command    ::= single-Command
            | Command ; single-Command
....
Expression ::= primary-Expression
            | Expression operator primary-Expression
```

EBNF

```
Command    ::= single-Command ( ; single-Command ) *
....
Expression  ::= primary-Expression
                ( operator primary-Expression ) *
```



- Auch REs definieren eine Sprache
  - ▢ Reguläre Sprache
  - ▢ Weniger komplex als durch CFG beschreibbare Sprachen
- CFG erlaubt Beschreibung von Selbsteinbettung
  - ▢ Ausdruck  $\mathbf{a^*(b+c)/d}$  bettet Ausdruck  $\mathbf{b+c}$  ein
  - ▢ Vergleichbar dem Konzept der Rekursion
- REs erlauben **keine** Beschreibung von Selbsteinbettung

Ziel: Systematische Herleitung von Parsern aus CFG



## Hilfsmittel

- CFG kann transformiert (umgestellt) werden
- ... unter Beibehaltung der beschriebenen Sprache



- Zusammenfassen von Produktionen mit gleichem Nicht-Terminal auf linker Seite
  - ▣ *Left-Hand Side* (LHS), analog RHS

## Vor Transformation

$$S ::= X + S$$

$$S ::= X$$

$$S ::= \varepsilon$$

## Nach Gruppierung

$$S ::= X + S | X | \varepsilon$$

# Grammatik-Transformation durch Linksauklammern



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Zusammenfassen von gleichen Anfängen in einer Produktion
- $X Y \mid X Z \rightarrow X(Y \mid Z)$

Beispiel:



```
cmd  := if Expr then cmd  
      |  if Expr then cmd else cmd
```

```
cmd  := if Expr then cmd ( $\epsilon$  | else cmd)
```

- Linksrekursion in Produktion
  - ▣  $N ::= X \mid N Y$
  - ▣  $L(N) = \{X, XY, XYY, XYYY, XYYYY, \dots\}$
- Ersetzung durch
  - ▣  $N ::= X(Y)^*$

Beispiel:



Identifier ::= Letter  
                  | Identifier Letter  
                  | Identifier Digit

Identifier ::= Letter (Letter | Digit)\*



## Vor Transformation

$$\mathbf{N} ::= \mathbf{X}_1 \mid \dots \mid \mathbf{X}_m \mid \mathbf{N} \mathbf{Y}_1 \mid \dots \mid \mathbf{N} \mathbf{Y}_n$$

## Nach Linksausklammern

$$\mathbf{N} ::= (\mathbf{X}_1 \mid \dots \mid \mathbf{X}_m) \mid (\mathbf{N}(\mathbf{Y}_1 \mid \dots \mid \mathbf{Y}_n))$$

## Nach Beseitigen der Linksrekursion

$$\mathbf{N} ::= (\mathbf{X}_1 \mid \dots \mid \mathbf{X}_m)(\mathbf{Y}_1 \mid \dots \mid \mathbf{Y}_n)^*$$



- Wenn  $N ::= X$  einzige Produktion mit LHS  $N$  ist
- ...  $N$  durch  $X$  in RHS aller Produktionen ersetzen

Beispiel:

## Vor Transformation

single-Declaration ::= **var** Identifier : Type-denoter | ...  
Type-denoter ::= Identifier

## Nach Ersetzung

single-Declaration ::= **var** Identifier : Identifier | ...

## Aber ...

Solche “überflüssigen” Nicht-Terminals können nützlichen Dokumentationscharakter für den menschlichen Leser haben!



- Hier auf den ersten Blick noch nicht erkennbar
- Erlauben kompaktere und lesbarere Beschreibung von CFGs
- **Sehr nützlich** bei der Konstruktion von Parsern für CFGs



**Erkennung:** Entscheidung, ob ein Eingabetext ein Satz der Grammatik  $G$  ist.

**Parsing:** Erkennung und zusätzlich Bestimmung der Phrasen-Struktur

- Beispiel: Durch *konkreten* Syntaxbaum

**Eindeutigkeit:** Eine Grammatik ist eindeutig falls jeder Eingabetext auf maximal eine Weise geparsed werden kann,

- Ein syntaktisch korrekter Eingabetext hat genau einen eindeutigen Syntaxbaum



- Zwei wesentliche Verfahren
- Unterscheiden sich in der Art ihres Vorgehens
  - Top-Down Beispiel: Rekursiver Abstieg
  - Bottom-Up Beispiel: Shift/Reduce



## Produktionen

**Sentence** ::= **Subject Verb Object .**  
**Subject** ::= **I | a Noun | the Noun**  
**Object** ::= **me | a Noun | the Noun**  
**Noun** ::= **cat | mat | rat**  
**Verb** ::= **like | is | see | sees**

## Beispiele der erzeugten Sprache

**the cat sees a rat .**  
**I like the cat .**  
**the cat see me .**  
**I like me .**  
**a rat like me .**



## Vorgehensweise

- Untersuche Eingabetext zeichenweise, von links nach rechts
- Baue Syntaxbaum von **unten nach oben** auf
  - ▣ Von den Terminalzeichen in den Blättern
  - ▣ ... zum S Nicht-Terminal in der Wurzel



## Zwei Arten von Aktionen

**Shift** Lese Zeichen ein

- Zusätzlich: Und lege es auf dem Stack ab

**Reduce** Erkenne ein Nicht-Terminal LHS der Produktion  $p$

- Zusätzlich: Oberste Elemente des Stapels müssen RHS von  $p$  entsprechen, ersetze durch LHS von  $p$  (Zusammenfassen)
- Ende wenn Startsymbol  $S$  erreicht und Eingabetext komplett gelesen

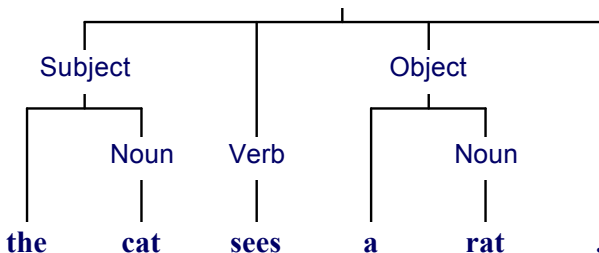
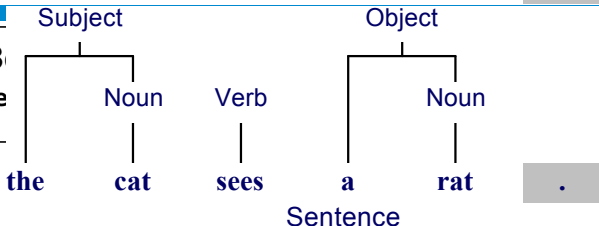


## Beispiel B

the cat see a rat .



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



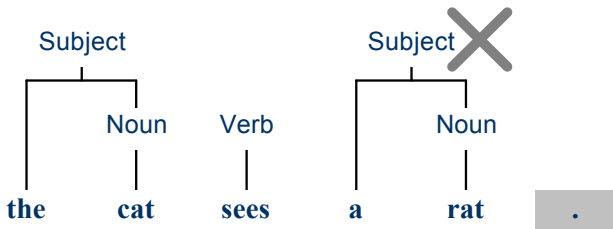
Sentence ::= Subject Verb Object .  
 Subject ::= **I** | **a** Noun | **the** Noun  
 Object ::= **me** | **a** Noun | **the** Noun  
 Noun ::= **cat** | **mat** | **rat**  
 Verb ::= **like** | **is** | **see** | **sees**

# Schwierigkeit bei Bottom-Up Parsing



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Welche Produktion beim Zusammenfassen anwenden?



Lösung: Nicht nur bekannte Zeichen betrachten, sondern auch noch Zustand ("schon Subject gesehen") einbeziehen.

... aber hier nicht weiter vertieft!



## Vorgehensweise

- Untersuche Eingabetext zeichenweise, von links nach rechts
- Baue Syntaxbaum von **oben nach unten** auf
  - ▣ Vom Start-Nicht-Terminal  $S$  in der Wurzel
  - ▣ ... zu den Terminalzeichen in den Blättern



## Aktion

- Expandiere jeweils das am weitesten links gelegene Nicht-Terminal **N**
- ... durch Anwendung einer Produktion **N ::= X**
- Wähle Produktion aus durch Betrachten der nächsten  $n$  Zeichen des Eingabetextes (Annahme hier:  $n = 1$ )
- Falls keine Produktion auf Zeichen passt → Fehler!
- Ende wenn Eingabetext komplett gelesen und kein unexpandiertes Nicht-Terminal mehr existiert

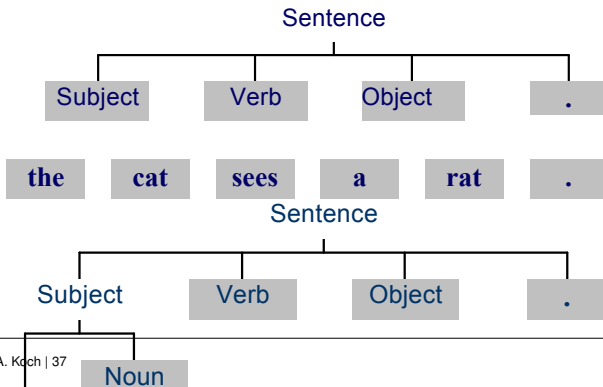
# Beispiel Top-Down Parsing

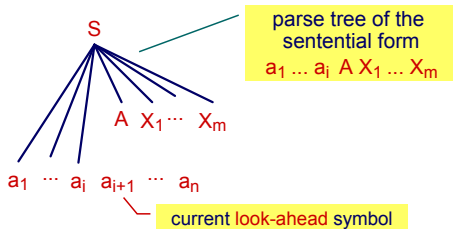
the cat sees a rat .



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Sentence ::= Subject Verb Object .  
Subject ::= I | a Noun | the Noun  
Object ::= me | a Noun | the Noun  
Noun ::= cat | mat | rat  
Verb ::= like | is | see | sees





Falls es möglich ist,

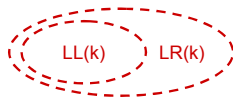
- ... bei Betrachten der nächsten  $k$  Zeichen des Textes
- ... immer die richtige Produktion zu finden

dann ist die Grammatik  $LL(k)$

- L: Lese Eingabetext von links nach rechts
- L: Leite immer vom am weitesten links stehenden Nicht-Terminal ab.



- Probleme mit Top-Down-Parsing
  - ▣ Konstruktion einer  $LL(k)$  Grammatik für die gewünschte Sprache gelegentlich mühsam
  - ▣ Linksausklammern und Beseitigen von Linksrekursion können Lesbarkeit der Grammatik erschweren
- Lösung: Bottom-Up-Parsing mit  $LR(k)$ -Techniken
  - ▣ **L**: Lese Eingabetext von **links nach rechts**
  - ▣ **R**: Fasse die am weitesten **rechts** stehenden Terminal-Symbole zusammen und baue den Baum **rückwärts** auf
  - ▣ Mächtigeres Beschreibungsinstrument als  $LL(k)$
  - ▣ Nachteil: Parsing-Vorgang komplexer und schlechter verständlich

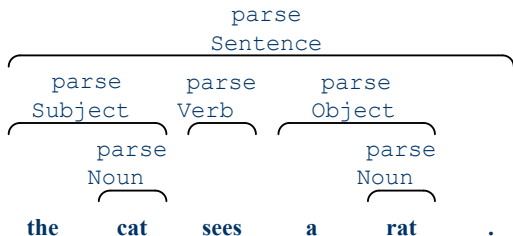




Einfache Implementierung der Top-Down Strategie, Idee:

- Struktur des konkreten Syntaxbaumes (Parse-Baum) entspricht
- ... Aufrufmuster von sich wechselseitig aufrufenden Prozeduren
- Für jedes Nicht-Terminal **XYZ** existiert
- ... Parse-Prozedur **parseXYZ**, die genau dieses Nicht-Terminal parst

Beispiel:





# Beispiel für Micro-English 1



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Sentence ::= Subject Verb Object .

```
protected void parseSentence() {  
    parseSubject();  
    parseVerb();  
    parseObject();  
    accept(".");  
}
```

`accept(t)` prüft, ob aktuelles  
Token das erwartete Token `t` ist.

# Beispiel für Micro-English 2



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Subject ::= **I** | **a** Noun | **the** Noun

```
protected void parseSubject() {  
    if (currentToken matches "I") {  
        accept("I");  
    } else if (currentToken matches "a") {  
        accept("a");  
        parseNoun();  
    } else if (currentToken matches "the") {  
        accept("the");  
        parseNoun();  
    } else  
        report a syntax error  
}
```

Die Methode **muß** immer anhand von currentToken die **passende** Alternative auswählen können.

# Beispiel für Micro-English 3



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
public class MicroEnglishParser {  
    protected Token currentToken;  
  
    public void parse() {  
        currentToken = first token;  
        parseSentence();  
        check that no token follows the sentence  
    }  
  
    protected void accept(Token expected) { ... }  
    protected void parseSentence() { ... }  
    protected void parseSubject() { ... }  
    protected void parseObject() { ... }  
    protected void parseNoun() { ... }  
    protected void parseVerb() { ... }  
  
    ...  
}
```

```
public class MicroEnglishParser {  
    protected Token currentToken;  
}
```



- **currentToken** enthält nacheinander die Tokens des Eingabetextes
- Ablauf einer Methode **parseN**
  - ▢ Bei Eintritt enthält **currentToken** eines der Token, mit denen **N** beginnen kann
  - ▢ ...sonst wäre eine andere Parse-Methode aufgerufen werden (oder Syntaxfehler)
  - ▢ Bei Austritt enthält **currentToken** das auf die **N**-Phrase folgende Token
- Ablauf der Methode **accept(t)**
  - ▢ Bei Eintritt muß **currentToken** =  $t$  sein
  - ▢ ...sonst Syntaxfehler
  - ▢ Bei Austritt enthält **currentToken** das auf  $t$  folgende Token



## Entwicklung von Parseern mit rekursivem Abstieg

### 1. Formuliere Grammatik (CFG) in EBNF

- ▣ Eine Produktion pro Nicht-Terminal
- ▣ Beseitige **immer** Linksrekursion
- ▣ Klammere gemeinsame Teilausdrücke nach links aus wo **möglich**

### 2. Erstelle Klasse für den Parser mit

- ▣ **protected** Variable **currentToken**
- ▣ Schnittstellenmethoden zum Scanner
  - **accept(*t*)** und **acceptIt()**
- ▣ **public** Methode **parse**, welche ...
  - erstes Token via Scanner aus dem Eingabetext liest
  - die Parse-Methode des Start Nicht-Terminals **S** der CFG aufruft

### 3. Implementiere **protected** Parsing-Methoden

- ▣ Methode **parseN** für jedes Nicht-Terminalsymbol **N**

## starters[[**X**]] mit EBNF-Ausdruck **X**

Menge aller Terminal-Symbole, die am Anfang einer aus **X** herleitbaren Zeichenkette stehen können.

### Beispiele

$$\text{starters}[[\mathbf{ab}]] = \{\mathbf{a}\}$$

$$\text{starters}[[\mathbf{a|b}]] = \{\mathbf{a, b}\}$$

$$\text{starters}[[\mathbf{(re) * set}]] = \{\mathbf{r, s}\}$$

# Berechnungsregeln für starters[[X]]



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

$$\text{starters}[[\varepsilon]] = \{\}$$

$$\text{starters}[[t]] = \{t\}$$

$$\text{starters}[[XY]] = \begin{cases} \text{starters}[[X]] & \text{falls aus } X \text{ kein } \varepsilon \text{ herleitbar} \\ \text{starters}[[X]] \cup \text{starters}[[Y]] & \text{sonst} \end{cases}$$

$$\text{starters}[[X|Y]] = \text{starters}[[X]] \cup \text{starters}[[Y]] \text{ noch nicht ganz richtig!}$$

$$\text{starters}[[X^*]] = \text{starters}[[X]]_{\text{dito!}}$$

$$\text{starters}[[N^*]] = \text{starters}[[X]], \text{ wenn } N ::= X \text{ dito!}$$

Ausbügeln der Ungenauigkeiten später (siehe Folie 52)



Annahme:  $N ::= X$ , nun **schrittweise** Zerlegung von  $X$

$\varepsilon$  ; (=leere Anweisung)

$t$  **accept**( $t$ );

**P** **parseP**();

**P Q** **parseP**();  
    **parseQ**();

**P|Q** **if** (**currentToken**  $\in$  **starters**[[**P**]]) *was bei P =  $\varepsilon$ ?*  
    **parseP**();  
    **else if** (**currentToken**  $\in$  **starters**[[**Q**]])  
        **parseQ**();  
    **else**  
        *melde Syntaxfehler*

**P\*** **while** (**currentToken**  $\in$  **starters**[[**P**]])  
    **parseP**();



Analog:  $\text{follow}[[\mathbf{X}]]$  ist Menge der Tokens, die in der CFG nach  $\mathbf{X}$  folgen können.

1. Ausdruck **innerhalb** rechter Seite von Produktion:

$$\mathbf{A} ::= (\mathbf{a} \mid \mathbf{b}) \mathbf{c} \quad \rightarrow \text{follow}[[\mathbf{a} \mid \mathbf{b}]] = \{\mathbf{c}\}$$

2. Ausdruck **am Ende** rechter Seite von Produktion:

$$\mathbf{A} ::= (\mathbf{a} \mid \mathbf{b})$$

$$\mathbf{B} ::= \mathbf{A} \mathbf{u}$$

$$\mathbf{C} ::= \mathbf{A} \mathbf{v}$$

$$\rightarrow \text{follow}[[\mathbf{a} \mid \mathbf{b}]] = \{\mathbf{u}, \mathbf{v}\}$$

3. Nichtterminal auf linker Seite von Produktion:

Vereinigung der Folgemenge aller Vorkommen auf rechten Seiten:

$$\text{follow}[[\mathbf{A}]] = \{\mathbf{u}, \mathbf{v}\}$$

4. Startsymbol: leere Menge



Funktionieren nur dann, wenn in Grammatik  $G$  gilt:

- Falls  $G$   $X|Y$  enthält und sich weder  $X$  noch  $Y$  zu  $\varepsilon$  ableiten lassen:  
 $\text{starters}[[X]] \cap \text{starters}[[Y]] = \emptyset$
- Falls  $G$   $X|Y$  enthält und sich beispielsweise  $Y$  zu  $\varepsilon$  ableiten lässt:  
 $\text{starters}[[X]] \cap (\text{starters}[[Y]] \cup \text{follow}[[X|Y]]) = \emptyset$
- Falls  $G$   $X^*$  enthält:  $\text{starters}[[X]] \cap \text{follow}[[X^*]] = \emptyset$

➡ Wenn alles gilt:  $G$  ist  $LL(k)$  mit  $k = 1$

Hinweis: Definition in PLPJ, p. 104 ist nicht ausreichend!

## Überprüfen von Alternativen und Wiederholungen

### ■ $S ::= Xd, B ::= b$

- Nichts zu prüfen

$S ::= Xd$

### ■ $X ::= A \mid B \mid c$

- Produktion enthält 3 Alternativen → paarweise prüfen!

$X ::= A \mid B \mid c$

- $B \mid c$

$$\text{starters}[[B]] \cap \text{starters}[[c]] = \emptyset \checkmark$$

- $A \mid B$

$A$  kann zu  $\varepsilon$  abgeleitet werden!

$$(\text{starters}[[A]] \cup \text{follow}[[A \mid B]]) \cap \text{starters}[[B]] = \emptyset \checkmark$$

- $A \mid c$  analog

### ■ $A ::= a^*$

- $a^*$

$$\text{starters}[[a]] \cap \text{follow}[[a^*]] = \emptyset \checkmark$$



Bisher gezeigt für  $P|Q$

```
if (currentToken ∈ starters[[P]])
    parseP();
else if (currentToken ∈ starters[[Q]])
    parseQ();
else
    melde Syntaxfehler
```

Problematisch, wenn  $\varepsilon$  aus  $P$  oder  $Q$  ableitbar.

Korrekt: Verwende statt starters[[X]]

$$\text{dirset}[[X]] = \begin{cases} \text{starters}[[X]] & \text{falls aus } X \text{ kein } \varepsilon \text{ herleitbar} \\ \text{starters}[[X]] \cup \text{follow}[[X]] & \text{sonst} \end{cases}$$

Analog für  $P^*$ . Korrigiere so Folie 48.



- Aus Algol Grammatik

**Block** ::= **begin Declaration ( ; Declaration)\* ; Command end**

- Prüfe Regel für **X\***

- ❑  $\text{starters}[[\text{ ; Declaration}]] = \{ ; \}$
- ❑  $\text{follow}[[\text{ ( ; Declaration)*}] = \{ ; \}$
- ❑  $\text{starters}[[\text{ ; Declaration}]] \cap \text{follow}[[\text{ ( ; Declaration)*}]] \neq \emptyset$

- Produktion ist aber transformierbar

**Block** ::= **begin Declaration ; (Declaration ;)\* Command end**

- Annahme:  $\text{starters}[[\text{Declaration ;}]] \cap \text{starters}[[\text{Command}]] = \emptyset$



## Annahme bis 1992

Rekursiver Abstieg funktioniert sinnvoll nur für  $k = 1$ , exponentieller Worst-Case-Aufwand bei  $k > 1$ .

## Gegenbeispiel 1992: PCCTS (jetzt ANTLR)

Worst-case kann für Grammatiken typischer Programmiersprachen in der Regel vermieden werden, sogar bei  $k = \infty$ .

- Konstruktion von Top-Down-Parsern gut automatisierbar
- Für Java beispielsweise
  - ▣ ANTLR: LL( $k$ ) bis LL( $*$ )
  - ▣ JavaCC: LL( $k$ )



Command ::= single-Command (; single-Command)\*

```
Command parseCommand() {  
    commandAST = parseSingleCommand();  
    while (currentToken.kind ==  
        TokenKind.SEMICOLON) {  
        acceptIt();  
        parseSingleCommand();  
    }  
}
```

## acceptIt()

- Könnte auch **accept(TokenKind.SEMICOLON)** sein
- Würde aber überflüssige Fehlerüberprüfung vornehmen
  - ▣ Token wurde schon vorher in **while(...)** geprüft
- Also ohne weitere Bearbeitung akzeptieren

# Parser für Mini-Triangle: parseSingleCommand



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
single-Command ::= Identifier ( := Expression
                             | ( Expression ) )
                | ...
```

```
Command parseSingleCommand() {
    switch (currentToken.kind) {
        case IDENTIFIER: {
            parseIdentifier();
            switch (currentToken.kind) {
                case BECOMES:
                    acceptIt();
                    parseExpression();
                    break;
                case LPAREN:
                    acceptIt();
                    parseExpression();
                    accept(TokenKind.RPAREN);
                    break;
                default: /* melde Syntaxfehler */
            }
            break; // case IDENTIFIER
        }
    }
    ...
}
```

Weitere Beispiele in PLPJ.





- Aufpassen bei
  - ▣ **parseIdentifizier**
  - ▣ **parseIntegerLiteral**
  - ▣ **parseOperator**
- ... hier nicht nur **Art** des Tokens relevant
- sondern **tatsächlicher** Text
  - ▣ **TokenKind.IDENTIFIER**: foo, bar, pi, k9, ...
  - ▣ **TokenKind.INTLITERAL**: 23, 42, 2006, ...
  - ▣ **TokenKind.OPERATOR**: +, -, /, ...

➡ Eingabetext nicht nur auf Token-**Art** reduzieren, Text selbst muß **erhalten** bleiben

# Häufige Fehler: Grammatik ist nicht LL(1)

## Grammatik



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Auszug aus Grammatik

```
single-Command ::= V-name := Expression
                | Identifier ( Expression )
                | if Expression then single-Command
                  else single-Command
                | ...
```

## Anfangsmengen

```
starters[[ V-name := Expression ]] = starters[[ V-name ]]
                                     = { Identifier }

starters[[ Identifier ( Expression ) ]] = { Identifier }

starters[[ if Expression then ... ]] = { if }
```

# Häufige Fehler: Grammatik ist nicht LL(1)

## Implementierung des Parsers



### Durch Zerlegung gewonnener Java-Code

```
Command parseSingleCommand() {  
    switch (currentToken.kind) {  
        case IDENTIFIER:  
            parseVname();  
            accept(TokenKind.BECOMES);  
            parseExpression();  
            break;  
  
        case IDENTIFIER:  
            parseIdentifier();  
            accept(TokenKind.LPAREN);  
            parseExpression();  
            accept(TokenKind.RPAREN);  
            break;  
  
        case IF: ...
```

# Häufige Fehler: Linksausklammern vergessen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Auszug aus Grammatik nach Ersetzen von **V-name** durch **Identifizier**

```
single-Command ::= Identifizier := Expression  
                | Identifizier ( Expression )  
                | if Expression then single-Command  
                  else single-Command
```

Anfangsmengen

$\text{starters}[[ \text{Identifizier} := \text{Expression} ]] = \{ \text{Identifizier} \}$

$\text{starters}[[ \text{Identifizier} ( \text{Expression} ) ]] = \{ \text{Identifizier} \}$

# Häufige Fehler: Linksausklammern vergessen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Jetzt mit Linksausklammern

```
single-Command ::= Identifier ( := Expression | ( Expression ) )  
                | if Expression then single-Command  
                  else single-Command
```

Neue Anfangsmengen

```
starters[[ := Expression ]] = { := }  
starters[[ ( Expression ) ]] = { ( }
```



Auszug aus Grammatik vor Korrektur

```
Command ::= single-Command  
          | Command ; single-Command
```

Anfangsmengen

```
starters[[ single-Command ]]  
        = { Identifier, if, while, let, begin }  
starters[[ Command ; single-Command ]]  
        = { Identifier, if, while, let, begin }
```

# Häufige Fehler: Linksrekursion nicht beseitigt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
Command parseCommand() {  
    switch (currentToken.kind) {  
        case IDENTIFIER:  
        case IF:    case WHILE:  
        case LET:  case BEGIN:  
            parseSingleCommand();  
            break;  
  
        case IDENTIFIER:  
        case IF:    case WHILE:  
        case LET:  case BEGIN:  
            parseCommand();  
            accept(";")  
            parseSingleCommand();  
            break;  
  
        default: /* melde Syntaxfehler */  
    }  
}
```

# Parser für Mini-Triangle: Grammatikanpassung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
Program      ::= single-Command
Command      ::= single-Command
               | Command ; single-Command
single-Command ::= V-name := Expression
               | Identifier ( Expression )
               | ...
```

```
Program      ::= single-Command
Command      ::= single-Command
               | Command ; single-Command
single-Command ::= V-name := Expression
               | Identifier ( Expression )
               | ...
```

Linksrekursion

Linksauklammern

```
Program      ::= single-Command
```

```
Command      ::= single-Command
               ( : single-Command )*
```





- Parser mit rekursivem Abstieg baut impliziten Syntaxbaum auf
  - ▢ Durch den Aufrufgraph der Parse-Methoden
- In einem Ein-Pass-Compiler unproblematisch
- Reicht nicht für Multi-Pass Compiler
  - ▢ **Weitergabe** der Daten zwischen Passes erforderlich



- Beobachtung: Jedes Nicht-Terminalsymbol **XYZ** wird durch eine Parse-Methode **parseXYZ** bearbeitet  
**protected void parseXYZ ( )**
  - ▣ Bisher nicht benutzt: Funktionsergebnis und Parameter
- Idee: Ausnutzung der Möglichkeiten zum Aufbau eines AST

# AST Knoten von Mini-Triangle



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Program	::= Command	Program
Command	::= Command ; Command   V-name := Expression   Identifier ( Expression )   <b>if</b> Expression <b>then</b> single-Command   <b>else</b> single-Command   <b>while</b> Expression <b>do</b> single-Command   <b>let</b> Declaration <b>in</b> single-Command	SequentialCmd AssignCmd CallCmd IfCmd WhileCmd LetCmd
Expression	::= Integer-Literal   V-name   Operator Expression   Expression Operator Expression	IntegerExpr VnameExpr UnaryExpr BinaryExpr
V-name	::= Identifier	SimpleVname
Declaration	::= Declaration ; Declaration   <b>const</b> Identifier ~ Expression   <b>var</b> Identifier : Type-denoter	SeqDecl ConstDecl VarDecl
Type-denoter	::= Identifier	SimpleTypeDen

# Sub-ASTs von Mini-Triangle



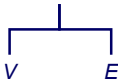
TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Command	::= Command ; Command	SequentialCmd
	V-name := Expression	AssignCmd
	Identifier ( Expression )	CallCmd
	if Expression then single-Command	IfCmd
	else single-Command	
	while Expression do single-Command	WhileCmd
	let Declaration in single-Command	LetCmd

SequentialCmd



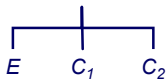
AssignCmd



CallCmd



IfCmd



WhileCmd



LetCmd





- Abstrakte Basisklasse  
**public abstract class** AST { ... }
- Eigene Subklassen für alle Arten von AST-Knoten

Jede Subklasse hat Instanzvariablen für ihre Unterknoten

```
public class Program extends AST {  
    public Command C;  
    ...  
}
```

Abstrakte Basisklasse aller **Command** AST-Knoten

```
public abstract class Command extends AST {
```

# Unterklassen der Command-Klasse



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
abstract class Command  
  extends AST { ... }
```

Command	
::= Command ; Command	SequentialCmd
V-name := Expression	AssignCmd
Identifier ( Expression )	CallCmd
if Expression then single-Command	IfCmd
else single-Command	
while Expression do single-Command	WhileCmd
let Declaration in single-Command	LetCmd

```
public class SequentialCmd extends Command {  
  public Command c1, c2;  
  ...  
}  
public class AssignCmd extends Command {  
  public Vname v;  
  public Expression e;  
  ...  
}  
public class CallCmd extends Command {  
  public Identifier i;  
  public Expression e;  
  ...  
}  
public class IfCmd extends Command {  
  public Expression e;  
  public Command c1, c2;  
  ...  
}
```

etc.

Die AST Subklassen haben  
auch entsprechende  
Konstruktoren zur korrekten  
Initialisierung der Objekte.

# Sonderfall: Terminal-Knoten



- Blätter des ASTs, hier ist **Text** des Tokens relevant
- Bezeichner, Zahlen, Operatoren

## Abstrakte Superklasse aller Terminal-Knoten

```
public abstract class Terminal extends AST {  
    public String spelling;  
    ...  
}
```

## Konkrete Unterklasse für Bezeichner

```
public class Identifier extends Terminal {  
    public Identifier(String spelling) {  
        this.spelling = spelling;  
    }  
}
```

- Während des rekursiven Abstiegs
- Idee: **parseN**-Methode liefert AST für **N**-Phrase
- AST für **N**-Phrase wird durch Zusammensetzen der ASTs der Subphrasen erstellt

## Beispiel für Produktion $N ::= X$

```
protected ASTN parseN () {  
    ASTN itsAST;  
    Parse X, sammle Subphrasen-ASTs in itsAST  
    return itsAST  
}
```



# Zusammensetzen von Subphrasen ASTs 1



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## EBNF

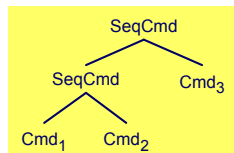
**Command ::= single-Command ( ; single-Command)\***

## AST

**Command ::= Command ; Command**

SequentialCmd

```
Command parseCommand() {  
    Command c1AST = parseSingleCommand();  
    while (currentToken.kind ==  
        TokenKind.SEMICOLON) {  
        acceptIt();  
        Command c2AST = parseSingleCommand();  
        c1AST = new SequentialCmd(c1AST, c2AST);  
    }  
    return c1AST;  
}
```



# Zusammensetzen von Subphrasen ASTs 2



```
Declaration parseSingleDeclaration() {
    Declaration declAST;
    switch (currentToken.kind) {
    case CONST: // single-Declaration ::= const Identifier ~ Expression
        acceptIt();
        Identifier iAST = parseIdentifier();
        accept(TokenKind.IS);
        Expression eAST = parseExpression();
        declAST = new ConstDeclaration(iAST, eAST);
        break;

    case VAR: // single-Declaration ::= var Identifier : Type-denoter
        acceptIt();
        Identifier iAST = parseIdentifier();
        accept(TokenKind.COLON);
        TypeDenoter tAST = parseTypeDenoter();
        declAST = new VarDeclaration(iAST, tAST);
        break;

    default: /* melde Syntaxfehler */
    }
    return declAST; }
```

# Scanning - Woher kommen die Tokens?



Zwei relevante Methoden im Parser

```
public class Parser {  
    Scanner scanner;  
    Token currentToken;  
  
    void accept(TokenKind tokenExpected) {  
        if (currentToken.kind == tokenExpected)  
            currentToken = scanner.scan();  
        else  
            /* melde Syntaxfehler */  
    }  
    void acceptIt() {  
        currentToken = scanner.scan();  
    }  
}
```



- Auch genannt lexikalische Analyse oder Lexer
- Ähnlich Parsing, aber auf einer Ebene feinerer Details
  - ▣ Parser: Arbeitet mit Tokens, die zu Phrasen gruppiert werden
  - ▣ Scanner: Arbeitet mit Zeichen, die zu Tokens gruppiert werden
- Aufgaben des Scanners
  - ▣ Bilde Tokens aus Zeichen
  - ▣ Entferne unerwünschte Leerzeichen, Zeilenvorschübe, etc. (white space)
  - ▣ Führe Buch über Zeilennummern und Eingabedateinamen

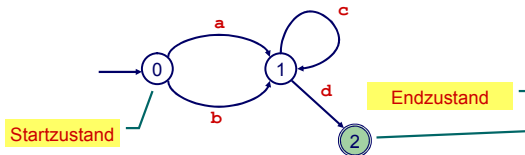


Tokens werden durch REs definiert, bestehend aus:

- Einzelzeichen
- Operatoren
  - ▣ Konkatenation: **A B**
  - ▣ Alternative: **A | B**
  - ▣ Optionalität: **A?**
  - ▣ Wiederholung: **A\***
  - ▣ Vordefinierte REs (sog. Macros)
- **aber:** keine rekursiven Definitionen

- Reguläre Ausdrücke können durch **Übergangsdiagramme** dargestellt werden
  - ▣ Endliche Automaten
  - ▣ Kanten/Transitionen beschriftet mit **Eingabesymbolen**
  - ▣ Zustände/Knoten
    - Genau ein Startzustand
    - Beliebig viele Endzustände (akzeptierende Zustände)

Beispiel: **(a | b) c\* d**





## Systematische Konstruktion von Scannern

1. Formuliere lexikalische Grammatik in EBNF
  - ▣ Falls nötig: Transformiere für rekursiven Abstieg
2. Implementiere Scan-Methoden **scanN** für jede Produktion **N ::= X**, mit Rumpf passend zu **X**
3. Implementiere Scanner-Klasse, bestehend aus
  - ▣ **protected** Instanzvariable **currentChar**
  - ▣ **protected** Methoden **take** und **takeIt**
    - Analog zu **accept/acceptIt** im Parser
    - Lesen diesmal aber zeichenweise in **currentChar**
  - ▣ **protected** Scan-Methoden aus 2., erweitert um Erstellen von Token-Objekten
  - ▣ Eine **public** Methode **scan**, die den nächsten Token liefert
    - Überspringt dabei white space und Kommentare



```
public class Scanner {
    char          currentChar;
    StringBuilder currentSpelling;

    public Token scan() {
        ... // Kommentare und Whitespace ueberlesen

        currentSpelling = new StringBuilder();
        TokenKind currentKind = scanToken();

        return new Token(currentKind, currentSpelling.toString());
    }

    TokenKind scanToken() {
        switch (currentChar) {
            ...
        }
    }

    void take(char expectedChar) { ... }
    void takeIt() { ... }
}
```

Hänge **currentChar** an **currentSpelling**  
und lese nächstes Zeichen in **currentChar**





## 1. Lexikalische Grammatik in EBNF verfassen

```
Token ::= Identifier | Integer-Literal | Operator |  
        ; | : | := | ~ | ( | ) | eot  
Identifier ::= Letter (Letter | Digit)*  
Integer-Literal ::= Digit Digit*  
Operator ::= + | - | * | / | < | > | =  
Separator ::= Comment | space | eol  
Comment ::= ! Graphic* eol
```

## 2. Umstellen für rekursiven Abstieg: Ersetzung und Linksausklammern

```
Token ::= Letter (Letter | Digit)*  
        | Digit Digit*  
        | + | - | * | / | < | > | =  
        | ; | : | (=|ε) | ~ | ( | ) | eot  
Separator ::= ! Graphic* eol | space | eol
```

Hier eigentlich nicht nötig. Aber: Schneller!



- EBNF kann **nicht** trennen zwischen
    - ▣ Schlüsselworten
    - ▣ Bezeichnen
  - Wird beides als **Identifizier** beschrieben
- ➡ während des Scannens reparieren.



```
public class Scanner {
    private char      currentChar = ... // hole erstes Zeichen
    private StringBuilder currentSpelling;

    private void take(char expectedChar) {
        if (currentChar == expectedChar)
            takeIt();
        else
            /* melde lexikalischen Fehler */
    }

    private void takeIt() {
        currentSpelling.append(currentChar);
        currentChar = ... // hole naechstes Zeichen
    }
    ...
}
```

```
...  
public Token scan() {  
    while (currentChar == '!' ||  
        || currentChar == ',' ||  
        || currentChar == '\n')  
        scanSeparator();  
  
    currentSpelling = new StringBuilder();  
    TokenKind currentKind = scanToken();  
  
    return new Token(kind, currentSpelling.toString(),  
                    /* Position */);  
}
```

```
private void scanSeparator() { ... }  
private void scanToken() { ... }
```

Entwicklung sehr ähnlich zu Parse-Methode

...



```
private TokenKind scanToken() {  
    switch (currentChar) {  
        case 'a': case 'b': ... case 'z':  
        case 'A': case 'B': ... case 'Z':  
            ... // Token ::= Letter (Letter | Digit)*  
            return TokenKind.IDENTIFIER;  
        case '0': ... case '9':  
            ... // Token ::= Digit Digit*  
            return TokenKind.INTLITERAL;  
        case '+': case '-': ... case '=':  
            takeIt();  
            return TokenKind.OPERATOR;  
        ...  
    }  
}
```



```
case 'a': case 'b': ... case 'z':  
case 'A': case 'B': ... case 'Z':  
    takeIt();  
    while (isLetter(currentChar) || isDigit(currentChar))  
        takeIt();  
    return TokenKind.IDENTIFIER;  
  
case '0': ... case '9':  
...  
...
```

# Hauptmethode `scan()`



```
...  
public Token scan() {  
    while (currentChar == '!'  
        || currentChar == '  
        || currentChar == '\n')  
        scanSeparator();  
  
    currentSpelling = new StringBuilder();  
    TokenKind currentKind = scanToken();  
  
    return new Token(kind, currentSpelling.toString(),  
        /* Position */);  
}
```

Wo nun Unterscheidung zwischen Bezeichnern und Schlüsselworten?

# Ändern von Token-Art während der Konstruktion



```
enum TokenKind {  
    ...  
    static final Map<String, TokenKind> reservedWords;  
    static {  
        // Trage Schlüsselwoerter in Hash-Tabelle ein  
        reservedWords = Stream.of(  
            ARRAY, BEGIN, CONST, DO, ELSE, END, FUNC, IF, IN,  
            LET, OF, PROC, RECORD, THEN, TYPE, VAR, WHILE  
        ).collect(toMap(t -> t.spelling, identity()));  
    }  
}  
  
final class Token {  
    ...  
    Token(TokenKind kind, String spelling) {  
        if (kind == TokenKind.IDENTIFIER)  
            if (TokenKind.reservedWords.contains(spelling))  
                kind = TokenKind.reservedWords.get(spelling)  
        ...  
    }  
}
```





- Sehr mechanischer Ablauf
- Gut automatisierbar
- Beispiele
  - ▣ JLex/JFlex: Scanner basiert auf endlichem Automaten
  - ▣ Eingebaute Scanner in Parser-Generatoren ANTLR/JavaCC