3. Syntaxanalyse

Hanspeter Mössenböck, Institut für Systemsoftware, JKU

Kontextfreie Grammatiken

Problem

Reguläre Grammatiken können keine Zentralrekursion ausdrücken

```
E = id | "(" E ")".
```

Dafür braucht man zumindest kontextfreie Grammatiken

Definition

Eine Grammatik heißt ***kontextfrei*** (KFG), wenn alle Produktionen folgende Form haben:

`X = α.` X ∈ NTS, α Folge von TS und NTS

In EBNF kann α auch |, (), [] und {} enthalten

Beispiel

```
Expr = Term {("+" | "-") Term}.
Term = Factor {("*" | "/") Factor}.
Factor = id | "(" Expr ")".
```

← indirekte Zentralrekursion

Kontextfreie Sprachen werden durch ***Kellerautomaten*** erkannt

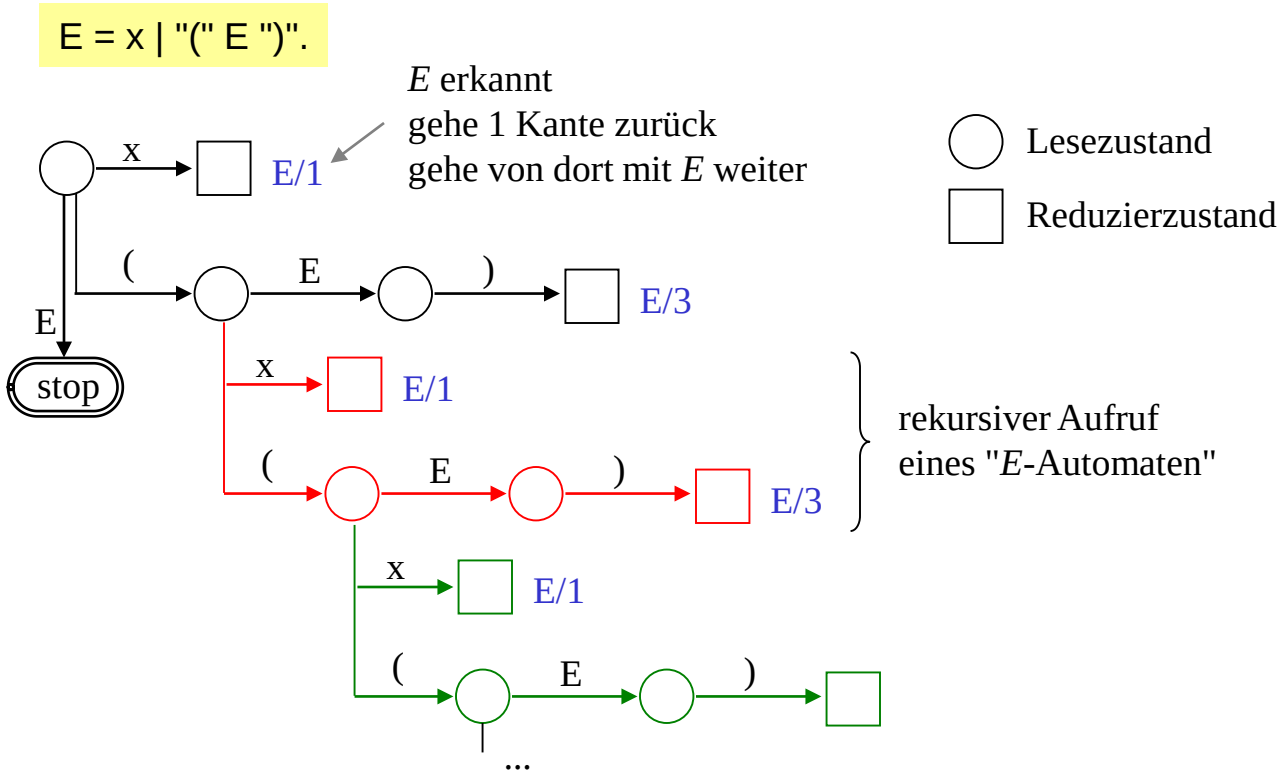
Kellerautomat

engl.: Push Down Automaton (PDA)

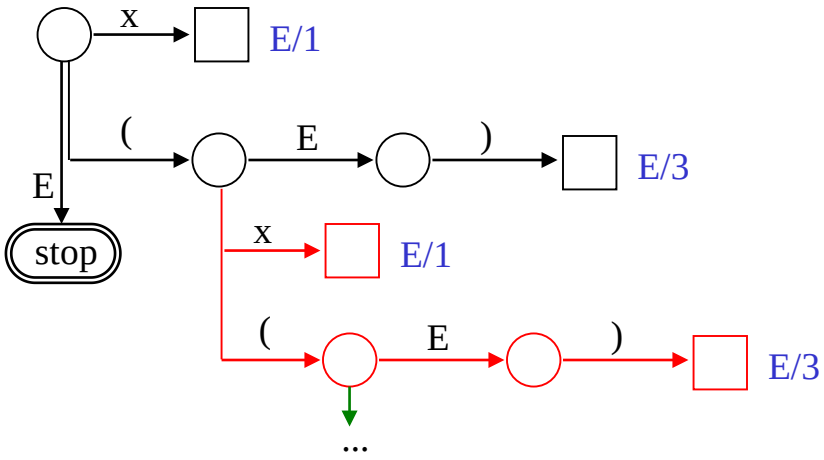
Eigenschaften

- erlaubt Zustandsübergänge mit Terminal- und Nonterminalsymbolen
- merkt sich Zustandsübergänge in einem Keller

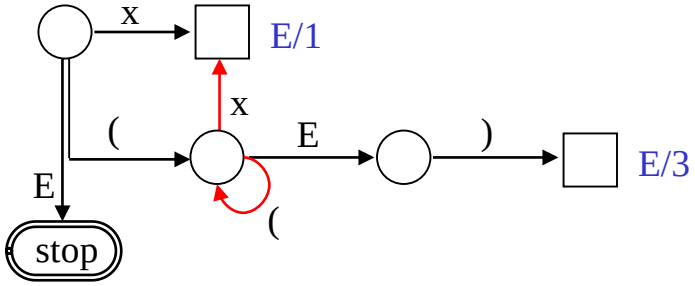
Beispiel



Kellerautomat (Fortsetzung)



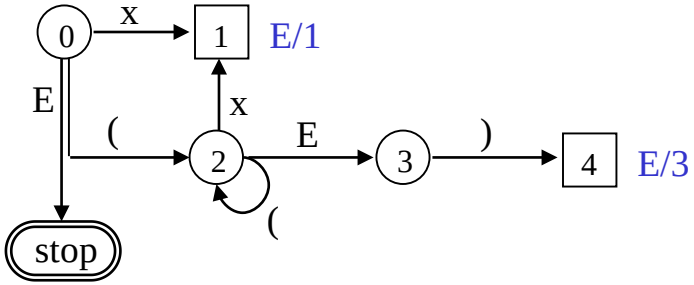
Kann vereinfacht werden zu



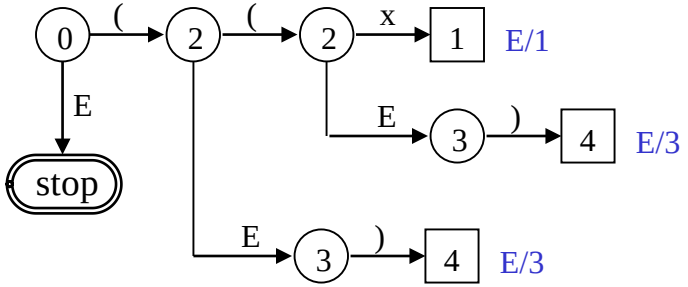
Erfordert Keller, um den Rückweg durch alle bisher durchlaufenen Zustände zu finden

Arbeitsweise eines PDA

Beispiel: Erkennung von ((x))



Durchlaufene Zustände werden in einem Keller gespeichert



Einschränkung kontextfreier Grammatiken

KFGs können keine *Kontextbedingungen* ausdrücken

Zum Beispiel:

- ***Jeder verwendete Name muss deklariert worden sein***
  Deklaration gehört zum Kontext der Verwendung
  ```
  x = 3;
  ```
  kann je nach Kontext richtig oder falsch sein

- ***In Ausdrücken müssen die Typen der Operanden übereinstimmen***
  Typen werden in der Deklaration festgelegt ⇒ gehört zum Kontext der Verwendung

Lösungsmöglichkeiten

- ***Kontextsensitive Grammatiken***
  zu kompliziert

- ***Überprüfung der Kontextbedingungen in der semantischen Analyse***
  d.h. die Grammatik erlaubt Konstrukte, die laut Kontextbedingungen verboten sind, z.B.:
  ```
  char x; ...
  x = 3;
  ```
  syntaktisch korrekt aber semantisch falsch
  Fehler wird erst bei semantischer Analyse gemeldet.

Kontextbedingungen

Für jede Syntaxregel werden die semantischen Zusatzbedingungen angegeben

Zum Beispiel für MicroJava

Statement = Designator "=" Expr ";".

- *Designator* muss eine Variable, ein Arrayelement oder ein Objektfeld bezeichnen.
- Der Typ von *Expr* muss mit dem Typ von *Designator* zuweisungskompatibel sein.

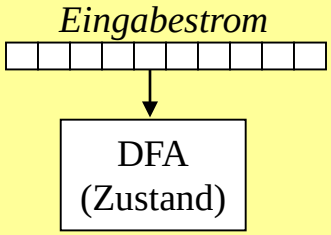
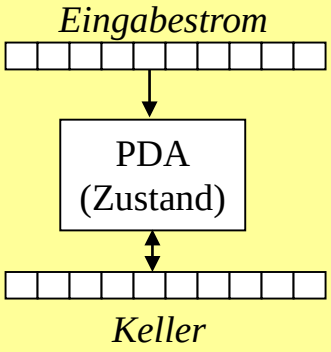
Factor = "new" ident "[" Expr "]".

- *ident* muss einen Typ bezeichnen.
- Der Typ von *Expr* muss *int* sein.

$Designator_1$ = $Designator_2$ "[" Expr "]".

- $Designator_2$ muss eine Variable, ein Arrayelement oder ein Objektfeld bezeichnen.
- Der Typ von $Designator_2$ muss ein Arraytyp sein.
- Der Typ von *Expr* muss *int* sein.

Reguläre versus kontextfreie Grammatiken

	Reguläre Grammatiken	**Kontextfreie Grammatiken**
Anwendung	Lexikalische Analyse	Syntaxanalyse
Erkennung	**DFA** (kein Keller) 	**PDA** (Keller) 
Produktionen	X = a | b Y.	X = α.
Probleme	Klammerkonstrukte	Kontextsensitive Konstrukte (z.B. Typprüfungen, ...)

3. Syntaxanalyse

Syntaxanalyse mit Rekursivem Abstieg

engl.: *Recursive Descent Parsing*

- Topdown-Technik
- Syntaxbaum wird von oben nach unten aufgebaut

Beispiel

Grammatik

X = a X c | b b.

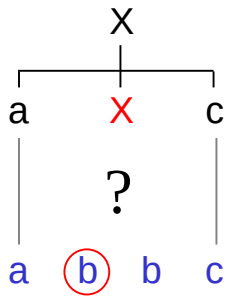
Eingabesatz

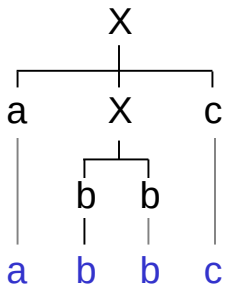
a b b c

Startsymbol X

?

welche Alternative passt?





Eingabesatz a b b c

Auswahl der passenden Alternative auf Grund von

- Vorgriffssymbol aus dem Eingabestrom
- Terminale Anfänge der einzelnen Alternativen

Globale Variablen des Parsers

Der Parser ist eine Klasse mit statischen Feldern und Methoden

Vorgriffssymbol

Parser schaut immer um 1 Token voraus (look ahead)

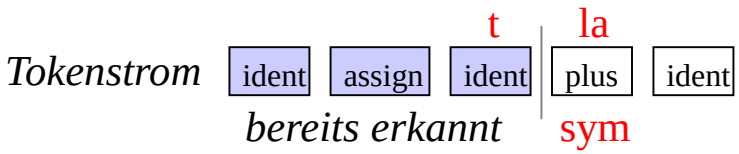
```
private static int sym;     // Tokencode des Vorgriffssymbols
```

Parser merkt sich die beiden aktuellen Token (für Semantikverarbeitung)

```
private static Token t;     // zuletzt erkanntes Token
private static Token la;    // look ahead token (noch nicht erkannt)
```

Diese Variablen werden von der Hilfsmethode *scan()* gefüllt

```
private static void scan() {
   t = la;
   la = Scanner.next();
   sym = la.kind;
}
```



scan() wird zu Beginn der Syntaxanalyse aufgerufen ⇒ erstes Symbol steht in *sym*

Erkennung von Terminalsymbolen

Muster

Terminalsymbol: a
Parser-Aktion: check(a);

Dazu nötige Hilfsmethoden

```
private static void check (int expected) {
   if (sym == expected) scan();  // erkannt, daher weiterlesen
   else error( name[expected] + " expected" );
}
```

```
private static void error (String msg) {
   System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
   System.exit(1);  // bessere Lösung siehe später
}
```

```
private static String[] name = {"?", "identifier", "number", ..., "+", "-", ...};
```

geordnet nach Tokencodes

Namen der Terminalsymbole sind als Konstanten deklariert

```
static final int
   none = 0,
   ident = 1,
   ... ;
```

Erkennung von Nonterminalsymbolen

Muster

Nonterminalsymbol: X
Parser-Aktion: X(); // Aufruf der Erkennungsmethode von X

Für jedes Nonterminalsymbol gibt es eine gleichnamige Erkennungsmethode

```
private static void X() {
   ... Aktionen zur Erkennung von X ...
}
```

Initialisierung des MicroJava-Parsers

```
public static void Parse() {
   scan();          // füllt la und sym
   MicroJava();     // ruft die Erkennungsmethode des Startsymbols auf
   check(eof);      // nach dem Programm muss die Eingabedatei zu Ende sein
}
```

Erkennung von Sequenzen

Muster

Grammatikregel: X = a Y c.

Erkennungsmethode:

```
private static void X() {
   // sym enthält das erste Symbol von X
   check(a);
   Y();
   check(c);
   // sym enthält ein Nachfolgesymbol von X
}
```

Simulation

X = a Y c.
Y = b b.

```
private static void X() {          a b b c
   check(a);                       b b c
   Y();                            c
   check(c);
}

private static void Y() {          b b c
   check(b);                       b c
   check(b);                       c
}
```

restliche Eingabe

Erkennung von Alternativen

Grammatikmuster: $\alpha \mid \beta \mid \gamma$ α, β, γ sind beliebige EBNF-Ausdrücke

Erkennungsaktion:

```
if (sym ∈ First(α)) { ... erkenne α ... }
else if (sym ∈ First(β)) { ... erkenne β ... }
else if (sym ∈ First(γ)) { ... erkenne γ ... }
else error("..."); // sprechende Fehlermeldung finden
```

Beispiel

```
X = a Y | Y b.
Y = c | d.
```

First(aY) = {a}
First(Yb) = First(Y) = {c, d}

```
private static void X() {
    if (sym == a) {
        check(a);
        Y();
    } else if (sym == c || sym == d) {
        Y();
        check(b);
    } else error ("invalid start of X");
}
```

Erkennung von EBNF-Optionen

Grammatikmuster: [α] α ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: if (sym ∈ First(α)) { ... *erkenne* α ... } // kein Fehlerzweig!

Beispiel

```
X = [a b] c.
```

```
private static void X() {
    if (sym == a) {
        check(a);
        check(b);
    }
    check(c);
}
```

Erkennung der Phrasen a b c und c

Erkennung von EBNF-Iterationen

Grammatikmuster: $\{\alpha\}$ α ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: while (sym $\in$ First(α)) { ... *erkenne* α ... }

Beispiel

```
X = a {Y} b.
Y = c | d.
```

```
private static void X() {
    check(a);
    while (sym == c || sym == d) Y();
    check(b);
}
```

Erkennung der Phrasen a c d b und a b

Alternativ dazu:

```
private static void X() {
    check(a);
    while (sym != b  && sym != eof) Y();
    check(b);
}
```

Birgt aber Gefahr einer Endlosschleife, wenn b vergessen wird

Umgang mit großen First-Mengen

Falls Menge > 4 Elemente: Klasse *BitSet* verwenden

z.B.: First(X) = {a, b, c, d, e}
First(Y) = {f, g, h, i, j}

Am Anfang des Parsers alle First-Mengen aufbauen

```
import java.util.BitSet;
private static BitSet firstX = new BitSet();
firstX.set(a); firstX.set(b); firstX.set(c); firstX.set(d); firstX.set(e);
private static BitSet firstY = new BitSet();
firstY.set(f); firstY.set(g); firstY.set(h); firstY.set(i); firstY.set(j);
```

Verwendung

Z = X | Y.

```
private static void Z() {
    if (firstX.get(sym)) X();
    else if (firstY.get(sym)) Y();
    else error("invalid Z");
}
```

Falls Menge ≤ 4 Elemente: auscodieren (ist schneller)

z.B.: First(X) = {a, b, c}

```
if (sym == a || sym == b || sym == c) ...
```

Optimierungen beim rekursiven Abstieg

Vermeiden von Mehrfachtests

X = a | b.

unoptimiert

```
private static void X() {
    if (sym == a) check(a);
    else if (sym == b) check(b);
    else error("invalid X");
}
```

optimiert

```
private static void X() {
    if (sym == a) scan();  // kein check(a);
    else if (sym == b) scan();
    else error("invalid X");
}
```

X = {a | Y d}.
Y = b | c.

unoptimiert

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) check(a);
        else if (sym == b || sym == c) {
            Y(); check(d);
        } else error("invalid X");
    }
}
```

optimiert

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) scan();
        else { // keine Prüfung mehr
            Y(); check(d);
        } // kein Fehlerfall
    }
}
```

Optimierungen beim rekursiven Abstieg

Effizienteres Schema für Alternativen in einer Iteration

X = {a | Y d}.

wie vorhin

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) scan();
        else {
            Y(); check(d);
        }
    }
}
```

noch mehr optimiert

```
private static void X() {
    for (;;) {
        if (sym == a) scan();
        else if (sym == b || sym == c) {
            Y(); check(d);
        } else break;
    }
}
```

kein Mehrfachtest auf a mehr

Optimierungen beim rekursiven Abstieg

Häufiges Iterationskonstrukt

α {separator α}

herkömmlich

```
... erkenne α ...
while (sym == separator) {
    scan();
    ... erkenne α ...
}
```

kürzer

```
for (;;) {
    ... erkenne α ...
    if (sym == separator) scan(); else break;
}
```

Konkretes Beispiel

Type ident {"," Type ident}

```
Type(); check(ident);
while (sym == comma) {
    scan();
    Type(); check(ident);
}
```

```
for (;;) {
    Type(); check(ident);
    if (sym == comma) scan(); else break;
}
```

Eingabe z.B.: int a , int b ;

Richtige Bestimmung der terminalen Anfänge

Grammatik

terminale Anfänge der Alternativen

```
X = Y a.
Y = {b} c
  | [d]
  | e.
```

b und *c*

d und *a* (!)

e

```
U = V e
  | f.
V = {d}.
```

d und *e* (weil *V* löschbar!)

f

Parser-Methoden

```
private static void X() {
    Y(); check(a);
}
```

```
private static void Y() {
    if (sym == b || sym == c) {
        while (sym == b) scan();
        check(c);
    } else if (sym == d || sym == a) {
        if (sym == d) scan();
    } else if (sym == e) {
        scan();
    } else error("invalid Y");
}
```

```
private static void U() {
    if (sym == d || sym == e) {
        V(); check(e);
    } else if (sym == f) {
        scan();
    } else error("invalid U");
}
```

```
private static void V() {
    while (sym == d) scan();
}
```

Rekursiver Abstieg und Syntaxbaum

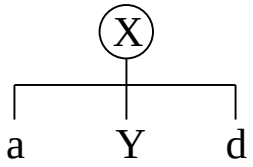
Syntaxbaum wird nur implizit aufgebaut

- steckt in den Parser-Methoden, die gerade aktiv sind
- d.h. in den Produktionen, an denen gerade gearbeitet wird

Beispiel X = a Y d.
Y = b c.

Aufruf von *X()*

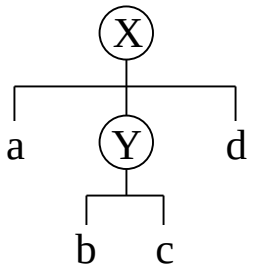
```
private static void X() {
    check(a); Y(); check(d);
}
```



X in Arbeit

Erkennung von *a*
Aufruf von *Y()*

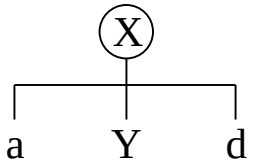
```
private static void Y() {
    check(b); check(c);
}
```



X in Arbeit
Y in Arbeit

"Keller"

Erkennung von *b* und *c*
Rückkehr von *Y()*



X in Arbeit

3. Syntaxanalyse

LL(1)-Eigenschaft

Vorbedingung für den rekursiven Abstieg

LL(1) ... erkennbar von **L**inks nach rechts
mit **L**inkskanonischen Ableitungen (linkstes NTS zuerst ableiten)
und **1** Vorgriffssymbol

Definition

1. Eine Grammatik ist LL(1), wenn alle ihre Produktionen LL(1) sind.
2. Eine Produktion ist LL(1), wenn für alle Alternativen
   $\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
   in dieser Produktion gilt:
   $First(\alpha_i) \cap First(\alpha_j) = \{\}$ (für beliebige $i \neq j$)

Mit anderen Worten

- Die terminalen Anfänge aller Alternativen einer Alternativenkette müssen paarweise disjunkt sein.
- Der Parser muss sich auf Grund des Vorgriffssymbols für eine der Alternativen entscheiden können.

Beseitigung von LL(1)-Konflikten

Faktorisierung

```
IfStatement =  "if" "(" Expr ")" Statement
             |  "if" "(" Expr ")" Statement "else" Statement.
```

Gleiche Anfänge herausziehen

```
IfStatement =  "if" "(" Expr ")" Statement (
                                           | "else" Statement
                                           ).
```

... oder in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement ["else" Statement].
```

Faktorisierung mit Auflösen von Nonterminalsymbolen

```
Statement   =  Designator "=" Expr ";"
             |  ident "(" [ActualParameters] ")" ";".
Designator  =  ident {"." ident}.
```

Designator in *Statement* einsetzen

```
Statement   =  ident {"." ident} "=" Expr ";"
             |  ident "(" [ActualParameters] ")" ";".
```

dann faktorisieren

```
Statement   =  ident  (   {"." ident} "=" Expr ";"
                      |   "(" [ActualParameters] ")" ";"
                      ).
```

Beseitigung von Linksrekursion

Linksrekursion ist immer ein LL(1)-Konflikt

Zum Beispiel

```
IdentList = ident | IdentList "," ident.
```

erzeugt folgende Phrasen

```
ident
ident "," ident
ident "," ident "," ident
...
```

kann immer durch Iteration ersetzt werden

```
IdentList = ident {"," ident}.
```

Versteckte LL(1)-Konflikte

EBNF-Optionen und -Iterationen sind versteckte Alternativen

X = [α] β. ⇔ X = α β | β. α und β sind beliebige EBNF-Ausdrücke

Regeln

X = [α] β.	First(α) ∩ First(β) muss {} sein
X = {α} β.	First(α) ∩ First(β) muss {} sein

X = α [β].	First(β) ∩ Follow(X) muss {} sein
X = α {β}.	First(β) ∩ Follow(X) muss {} sein

X = α | .	First(α) ∩ Follow(X) muss {} sein

Beseitigung versteckter LL(1)-Konflikte


```
Name = [ident "."] ident.
```

Wo steckt der Konflikt und wie kann er beseitigt werden?

```
Prog = Declarations ";" Statements.
Declarations = D {";" D}.
```

Wo steckt der Konflikt und wie kann er beseitigt werden?

Dangling Else

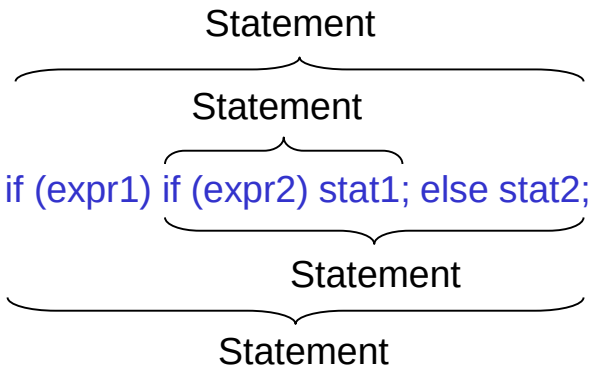
If-Anweisung in Java

```
Statement =  "if" "(" Expr ")" Statement ["else" Statement]
           |  ... .
```

Das ist ein LL(1)-Konflikt!

First("else" Statement) ∩ Follow(Statement) = {"else"}

Es ist sogar eine Mehrdeutigkeit, die sich nicht beheben lässt



Es gibt 2 verschiedene Syntaxbäume!

Behandlung von LL(1)-Konflikten

LL(1)-Konflikt ist nur eine Warnung

Der Parser nimmt einfach die erste passende Alternative

```
X = a b c
  | a d.
```

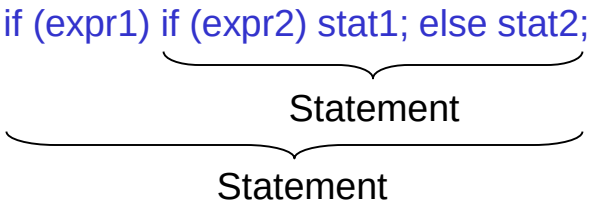
mit *a* als Vorgriffssymbol nimmt der Parser diese Alternative

Dieser LL(1)-Konflikt muss daher beseitigt werden (durch Faktorisierung)

Beispiel: Dangling Else

```
Statement = "if" "(" Expr ")" Statement [ "else" Statement ]
          | ... .
```

Wenn das Vorgriffssymbol hier "else" ist,
betritt der Parser die Option;
d.h. das "else" gehört zum innersten "if"



Das ist hier "zufällig" die gewünschte Interpretation.

Weitere Anforderungen an eine Grammatik

(Vorbedingungen für die Syntaxanalyse)

Vollständigkeit

Für jedes NTS muss es eine Produktion geben

```
X = a Y Z .
Y = b b .
```

falsch!
Keine Produktion für *Z*

Terminalisierbarkeit

Jedes NTS muss sich (direkt oder indirekt) in eine Kette von Terminalsymbolen ableiten lassen

```
X = a Y | c .
Y = b Y .
```

falsch!
Y kann nicht in lauter Terminalsymbole abgeleitet werden

Zirkularitätsfreiheit

Ein NTS darf nicht (direkt oder indirekt) in sich selbst ableitbar sein ($X \Rightarrow Y_1 \Rightarrow Y_2 \Rightarrow ... \Rightarrow X$)

```
X = a | Y .
Y = b | X .
```

falsch!
Zirkulär, weil $X \Rightarrow Y \Rightarrow X$

3. Syntaxanalyse

Ziele der Syntaxfehlerbehandlung

Anforderungen

1. Der Parser soll möglichst viele Fehler pro Übersetzung finden
2. Der Parser darf auch bei schlimmen Fehlern nicht abstürzen
3. Die Fehlerbehandlung soll die fehlerfreie Analyse nicht bremsen
4. Die Fehlerbehandlung soll den Parser-Code nicht aufblähen

Fehlerbehandlungsmethoden für den rekursiven Abstieg

- Fehlerbehandlung im "Panic Mode"
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

3. Syntaxanalyse

3.1 Kontextfreie Grammatiken und Kellerautomaten

3.2 Rekursiver Abstieg

3.3 LL(1)-Eigenschaft

3.4 Fehlerbehandlung

- Panic Mode
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Panic Mode

Die Syntaxanalyse wird nach dem ersten Fehler abgebrochen

```
private static void error (String msg) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    System.exit(1);
}
```

Vorteile

- billig
- für kleine Kommandosprachen oder für Interpreter ausreichend

Nachteile

- für größere Sprachen nicht zufriedenstellend

3. Syntaxanalyse

3.1 Kontextfreie Grammatiken und Kellerautomaten

3.2 Rekursiver Abstieg

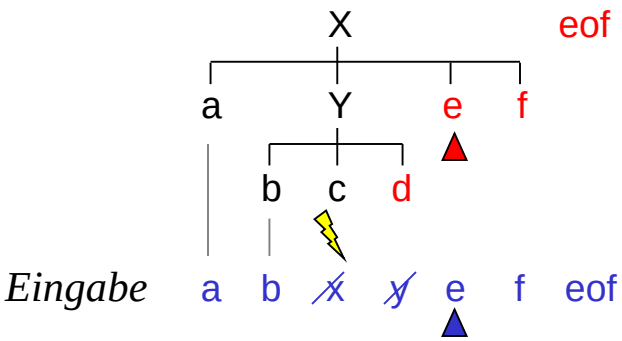
3.3 LL(1)-Eigenschaft

3.4 Fehlerbehandlung

- Panic Mode
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Fehlerbehandlung mit allgemeinen Fangsymbolen

Beispiel



Wiederaufsatz (Synchronisation der restlichen Eingabe mit der Grammatik)

1. **Aus der Grammatik "Fangsymbole" berechnen, mit denen nach der Fehlerstelle fortgesetzt werden kann**
   (Fangsymbole = Nachfolger aller Symbole, an denen der Parser gerade arbeitet).

   Fangsymbole hier: {d} + {e, f} + {eof}

2. **Fehlerhafte Symbole überlesen, bis ein Fangsymbol auftritt.**
   x und *y* werden hier überlesen; mit *e* kann fortgesetzt werden.

3. **Parser an die Stelle in der Grammatik steuern, an der fortgesetzt werden kann.**

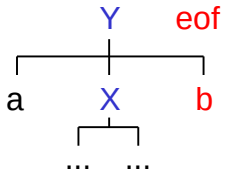
Berechnung der Fangsymbole

Jeder Erkennungsmethode eines Nonterminalsymbols *X* werden die aktuellen Nachfolger von *X* als Parameter mitgegeben

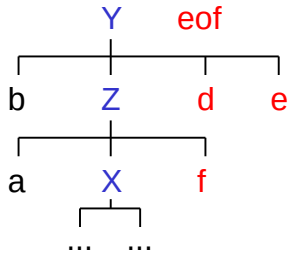
```
private static void X (BitSet sux) {
   ...
}
```

sux ... Nachfolger aller NTS, an denen gerade gearbeitet wird

sux_X kann je nach Kontext eine andere Menge sein



$sux_X = \{b, eof\}$



$sux_X = \{f, d, e, eof\}$

sux enthält immer *eof* (Nachfolger des Startsymbols)

Erkennung von Terminalsymbolen

Grammatik

X = ... a α_1 α_2 ... α_n .

α_i ... EBNF-Konstrukte

Erkennungsmuster

check(a, First(α_1) ∪ First(α_2) ∪ ... ∪ First(α_n) ∪ sux_X);

kann statisch vorausberechnet werden

muss zur Laufzeit berechnet werden

```
private static void check (int expected, BitSet sux) {...}
```

Beispiel

X = a b c.

```
private static void X (BitSet sux) {
   check(a, {b, c} ∪ sux);
   check(b, {c} ∪ sux);
   check(c, sux);
}
```

Pseudocode

Symbolmengen am Programmbeginn vorberechnen:

```
static BitSet fs1 = new BitSet();
fs1.set(b); fs1.set(c);
```

```
check(a, ((BitSet)fs1.clone()).or(sux));
```

Erkennung von Nonterminalsymbolen

Grammatik

X = ... Y α_1 α_2 ... α_n .

Erkennungsmuster

Y(First(α_1) $\cup$ First(α_2) $\cup$ ... $\cup$ First(α_n) $\cup$ sux_X);

Beispiel

X = a Y c.
Y = b b.

```
private static void X (BitSet sux) {
    check(a, {b, c} ∪ sux);
    Y({c} ∪ sux);
    check(c, sux);
}
```

```
private static void Y (BitSet sux) {
    check(b, {b} ∪ sux);
    check(b, sux);
}
```

Das Startsymbol *S* wird mit S({eof}); aufgerufen

Überlesen fehlerhafter Symbole

Fehler werden in *check()* bemerkt

```
private static void check (int expected, BitSet sux) {
   if (sym == expected) scan();
   else error(name[expected] + " expected", sux);
}
```

Nach Meldung des Fehlers wird bis zum nächsten Fangsymbol überlesen

```
private static void error (String msg, BitSet sux) {
   System.out.println("line " + la.line + " col " + la.col + ": " + msg);
   errors++;
   while (!sux.get(sym)) scan();  // while (sym ∉ sux) scan();
   // sym ∈ sux
}
```

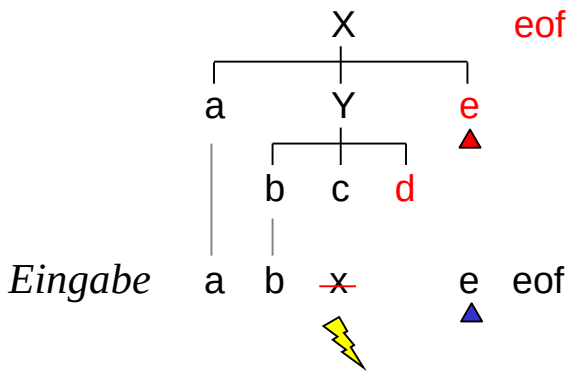
```
public static int errors = 0;  // Anzahl gefundener Syntaxfehler
```

Synchronisation mit der Grammatik

Beispiel

X = a Y e.
Y = b c d.

```
private static void X (BitSet sux) {
    check(a, {b, e} ∪ sux);
    Y({e} ∪ sux);
    check(e, sux);
}
private static void Y (BitSet sux) {
    check(b, {c, d} ∪ sux);
    check(c, {d} ∪ sux);
    check(d, sux);
}
```



$sux_X = \{eof\}$

$sux_Y = \{e, eof\}$

Fehler wird hier entdeckt; Fangsymbole = {*d*, *e*, *eof*}

1. *x* wird überlesen; *sym* == *e* ∈ Fangsymbole
2. Parser setzt fort: *check(d, sux);*
3. liefert wieder einen Fehler; Fangsymbole = {*e*, *eof*}
4. es wird nichts überlesen, weil *sym* == *e* ∈ Fangsymbole
5. Parser kehrt aus *Y()* zurück und macht *check(e, sux);*
6. Wiederaufsatz geglückt!

Parser "holpert" nach der Fehlerstelle weiter, bis er zu einer Stelle in der Grammatik kommt, an der das gefundene Fangsymbol erwartet wird.

Unterdrücken von Folgefehlermeldungen

Während des Wiederaufsatzes produziert der Parser Folgefehlermeldungen

Abhilfe durch einfache Heuristik

Fehler werden nur dann gemeldet,
wenn seit dem letzten Fehler mindestens 3 Symbole korrekt erkannt wurden.
=> Folgefehlermeldungen werden unterdrückt

```
private static int errDist = 3;  // "letzter Fehler" liegt mehr als 3 Symbole zurück
```

```
private static void scan() {
   ...
   errDist++;  // wieder ein Symbol erkannt
}
```

```
private static void error (String msg, BitSet sux) {
   if (errDist >= 3) {
      System.out.println("line " + la.line + " col " + la.col + ": " + msg);
      errors++;
   }
   while (!sux.get(sym)) scan();
   errDist = 0;  // Zählung beginnt von neuem
}
```

Erkennung von Alternativen

X = α | β | γ .

α, β, γ sind beliebige EBNF-Ausdrücke

Konventionelle Implementierung

```
private static void X () {
   if (sym ∈ First(α)) ... parse α ...
   else if (sym ∈ First(β)) ... parse β ...
   else if (sym ∈ First(γ)) ... parse γ ...
   else error(...);
}
```

Fehler wird zu spät entdeckt!
Keine Chance mehr, mit First(α), First(β) oder First(γ) wieder fortzusetzen.

Daher mit Fehlerbehandlung so:

```
private static void X (BitSet sux) {
   if (sym ∉ (First(α) ∪ First(β) ∪ First(γ)))
      error("invalid X", First(α) ∪ First(β) ∪ First(γ) ∪ sux);
   // sym passt zu einer der nächsten Alternativen
   // oder ist ein legaler Nachfolger von X
   if (sym ∈ First(α)) ... parse α ...
   else if (sym ∈ First(β)) ... parse β ...
   else if (sym ∈ First(γ)) ... parse γ ...
}
```

vorgezogene Fehlerabfrage, damit im Fehlerfall mit den Anfängen der Alternativen synchronisiert werden kann

kein Fehlerzweig mehr

First(α) ∪ First(β) ∪ First(γ) kann wieder statisch vorberechnet werden
sux ∪ ... muss zur Laufzeit berechnet werden

Erkennung von Optionen und Iterationen

Optionen

X = [α] β.

```
private static void X (BitSet sux) {
   if (sym ∉ (First(α) ∪ First(β)))
      error("...", First(α) ∪ First(β) ∪ sux);
   // sym passt zu α oder β
   // oder ist ein Nachfolger von X
   if (sym ∈ First(α)) ... parse α ...;
   ... parse β ...
}
```

vorgezogene Fehlerabfrage, damit im Fehlerfall auch mit α synchronisiert werden kann

Iterationen

X = {α} β.

```
private static void X (BitSet sux) {
   for (;;) {
      // Schleife auch betreten, wenn sym ∉ First(α)
      if (sym ∈ First(α)) ... parse α ...;              // korrekter Fall 1
      else if (sym ∈ First(β) ∪ sux) break;             // korrekter Fall 2
      else error("...", First(α) ∪ First(β) ∪ sux);     // Fehlerfall
   }
   ... parse β ...
}
```

Beispiel


```
X = a Y
   |  b {c d}.
Y = [b] d.
```

```
private static void X (BitSet sux) {
   if (sym != a && sym != b)
      error("invalid X", {a, b} ∪ sux);
   if (sym == a) {
      scan(); Y(sux);
   } else if (sym == b) {
      scan();
      for (;;) {
         if (sym == c) {
            scan();
            check(d, {c} ∪ sux);
         } else if (sym ∈ sux) {
            break;
         } else {
            error("c expected", {c} ∪ sux);
         }
      }
   }
}
```

```
private static void Y (BitSet sux) {
   if (sym != b && sym != d)
      error("invalid Y", {b, d} ∪ sux);
   if (sym == b) scan();
   check(d, sux);
}
```

Zusammenfassung

Fehlerbehandlung mit allgemeinen Fangsymbolen

Vorteil

+ systematisch anwendbar
+ gibt guten Wiederaufsatz

Nachteile

- bremst die fehlerfreie Syntaxanalyse
- bläht den Parsercode auf
- ist kompliziert

3. Syntaxanalyse

3.1 Kontextfreie Grammatiken und Kellerautomaten

3.2 Rekursiver Abstieg

3.3 LL(1)-Eigenschaft

3.4 Fehlerbehandlung

- Panic Mode
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Fehlerbehandlung mit speziellen Fangsymbolen

Synchronisation erfolgt nur an besonders "sicheren" Stellen

d.h. an Stellen, die mit Schlüsselwörtern beginnen, die an keiner anderen Stelle in der Grammatik vorkommen

Zum Beispiel

- Statement-Anfang: if, while, do, ...
- Deklarationsanfang: public, static, void, ...

Fangsymbolmengen

Allerdings Problem, dass an beiden Stellen auch *ident* vorkommen kann, das kein sicheres Fangsymbol ist ⇒ *ident* aus Fangsymbolmengen weglassen

Code, der an der Synchronisationsstelle eingefügt werden muss

```
...                                  erwartete Symbole an dieser Synchronisationsstelle
if (sym ∉ expectedSymbols) {
   error("...");  // keine Nachfolgermengen; kein Überlesen in error()
   while (sym ∉ (safeExpectedSymbols ∪ {eof})) scan();
}
...          Fangsymbole                damit es zu keiner Endlosschleife kommt
```

- Es müssen keine Nachfolgermengen als Parameter mitgegeben werden
- Fangsymbolmengen können statisch berechnet werden
- Nach einem Fehler "holpert" der Parser bis zur nächsten Synchronisationsstelle weiter

Beispiel

Synchronisation am Statement-Anfang

```
private static void Statement() {
   if (!firstStat.get(sym)) {
      error("invalid start of statement");
      while (!syncStat.get(sym)) scan();
      errDist = 0;
   }
   if (sym == if_) {
      scan();
      check(lpar); Expr(); check(rpar);
      Statement();
      if (sym == else_) { scan(); Statement(); }
   } else if (sym == while_) {
      ...
}
```

```
static BitSet firstStat = new BitSet();
firstStat.set(while_);
firstStat.set(if_);
...
```

firstStat ... mit *ident*
syncStat ... *firstStat* ohne *ident* aber mit *eof*

Rest der Syntaxanalyse bleibt wie wenn es keine Fehlerbehandlung gäbe

Keine Synchronisation in *error()*

```
private static void error (String msg) {
   if (errDist >= 3) {
      System.out.println("line " + la.line + " col " + la.col + ": " + msg);
      errors++;
   }
   errDist = 0;
}
```

Heuristik mit *errDist* kann auch hier verwendet werden

Simulation eines Wiederaufsatzes

```
private static void Statement() {
   if (!firstStat.get(sym)) {
      error("invalid start of statement");
      while (!syncStat.get(sym)) scan();
      errDist = 0;
   }
   if (sym == if_) {
      scan();
      check(lpar); Condition(); check(rpar);
      Statement();
      if (sym == else_) { scan(); Statement(); }
   ...
}
```

```
private static void check (int expected) {
   if (sym == expected) scan();
   else error(...);
}
```

```
private static void error (String msg) {
   if (errDist >= 3) {
      System.out.println(...);
      errors++;
   }
   errDist = 0;
}
```

Fehlerhafte Eingabe: if a > b , max = a; while ...

sym	*Aktion*	
if	scan();	*if* ∈ *firstStat* ⇒ ok
$ident_a$	check(lpar);	Fehlermeldung: (expected
	Condition();	erkennt a > b
comma	check(rpar);	Fehlermeldung:) expected
	Statement();	*comma* passt nicht ⇒ Fehler, aber keine Meldung
		Überlesen von ", *max = a"* , Synchronisation mit *";"*
semicolon		Wiederaufsatz geglückt!

Synchronisation am Anfang einer Iteration

Zum Beispiel

```
Block = "{" {Statement} "}".
```

Standardbehandlung bei Synchronisation am Statementanfang

```
private static void Block() {
   check(lbrace);
   while (sym ∈ First(Statement))
      Statement();
   check(rbrace);
}
```

Bei falschem *Statement*-Anfang wird die Schleife nicht betreten oder sie wird vorzeitig verlassen.
Synchronisationsstelle in *Statement* wird gar nicht erreicht.

Daher besser

```
private static void Block() {
   check(lbrace);
   while (sym ∉ {rbrace, eof})
      Statement();
   check(rbrace);
}
```

Zusammenfassung

Fehlerbehandlung mit speziellen Fangsymbolen

Vorteile

+ bremst fehlerfreie Syntaxanalyse nicht
+ bläht Code des Parsers nicht auf
+ einfach

Nachteil

- erfordert Erfahrung und "Tuning"