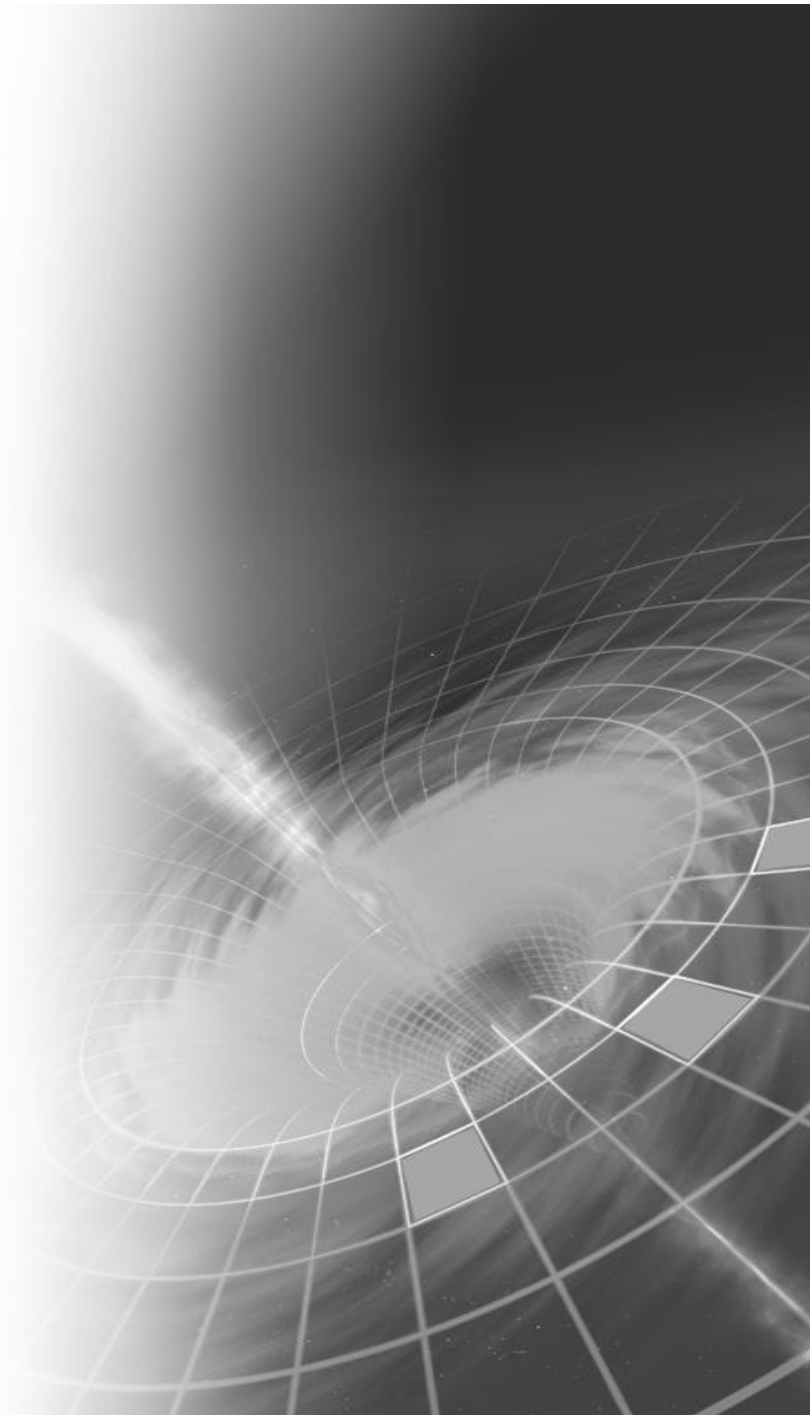
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dr. Andreas Kunert
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Position

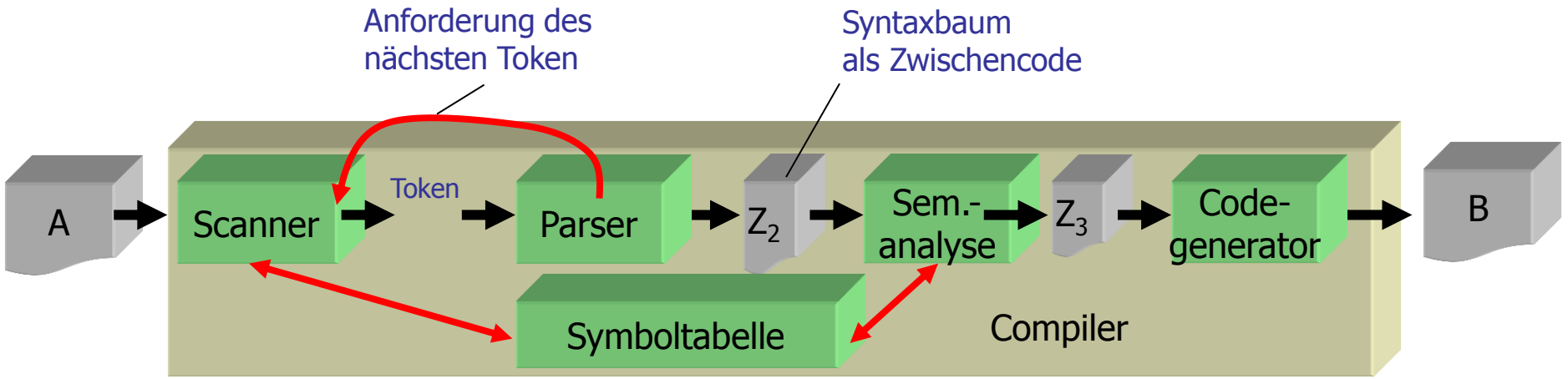
- **Teil I**
  Die Programmiersprache C
- **Teil II**
  Methodische Grundlagen des Compilerbaus
- **Teil III**
  Entwicklung eines einfachen Transcompilers

- **Kapitel 1**
  Compilationsprozess
- **Kapitel 2**
  Formalismen zur Sprachbeschreibung
- **Kapitel 3**
  Lexikalische Analyse: der Scanner
- **Kapitel 4**
  Syntaktische Analyse: der Parser
- **Kapitel 5**
  Parsergeneratoren: Yacc, Bison
- **Kapitel 6**
  Statische Semantikanalyse
- **Kapitel 7**
  Laufzeitsysteme
- **Kapitel 8**
  Ausblick: Codegenerierung

4.1 Einführung in die Syntaxanalyse

- Begriffsklärung und Klassifikation von Syntaxanalysatoren
- Prinzipielle Arbeitsweise eines Parsers
- Ein einfacher prädiktiver Parser (rekursiver Abstieg)
- Zusammenhang zwischen Parser und Grammatik-Unterklassen
- Mehrdeutigkeiten von Grammatiken

Der Parser



Parser: realisiert die syntaktische Analyse von Programmen

Syntaxanalyse

- Die syntaktische Analyse überprüft,
  ob der eingelesene Quellcode ein korrektes Programm der zu übersetzenden Quellsprache ist,
  d.h., der Syntax (Grammatik) der Quellsprache entspricht

- Dabei wird die Eingabe in einen Syntaxbaum umgewandelt.
  d.h. Teilfolgen von Symbolen/Token davon sind schrittweise zu immer größeren syntaktischen Einheiten zusammenzufassen
  Dieser Vorgang wird auch als Parsen bezeichnet.

- Falls der Quellcode nicht zur Grammatik der Quellsprache passt,
  gibt der Parser einen Syntaxfehler aus.
  Fehlerbehandlung, so dass sichere Fortsetzung der Analyse des Restquelltextes möglich ist

Prinzipielle Behandlung von Fehlern

- **Normalfall**: Programme sind fehlerhaft
- zu behandelnde **Fehlerklassen**:
  1. lexikalische Fehler
     (z.B. nicht beendete Kommentare)
  2. Syntaxfehler
     (z.B. Klammerstrukturierung)
  3. Fehler in der statischen Semantik
     (z.B. Typfehler)

Schritte

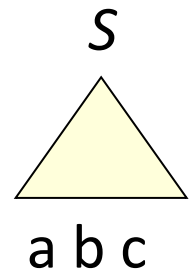
- melde und lokalisiere den Fehler
- diagnostiziere den Fehler
- korrigiere den Fehler (wenn möglich)

  um zumindest das nächste Token verarbeiten oder
  Endlos-Schleifen in der Analyse abwehren zu können
- fasse wieder Tritt, um evtl. vorhandene weitere Fehler zu entdecken

  dazu müssen wenigstens bis zum nächsten Token (z.B. Semikolon) Zeichen »geschluckt« werden

- effektive Umsetzung ist jedoch schwierig
- manche Fehler werden erst viel später entdeckt nachdem sie auftraten

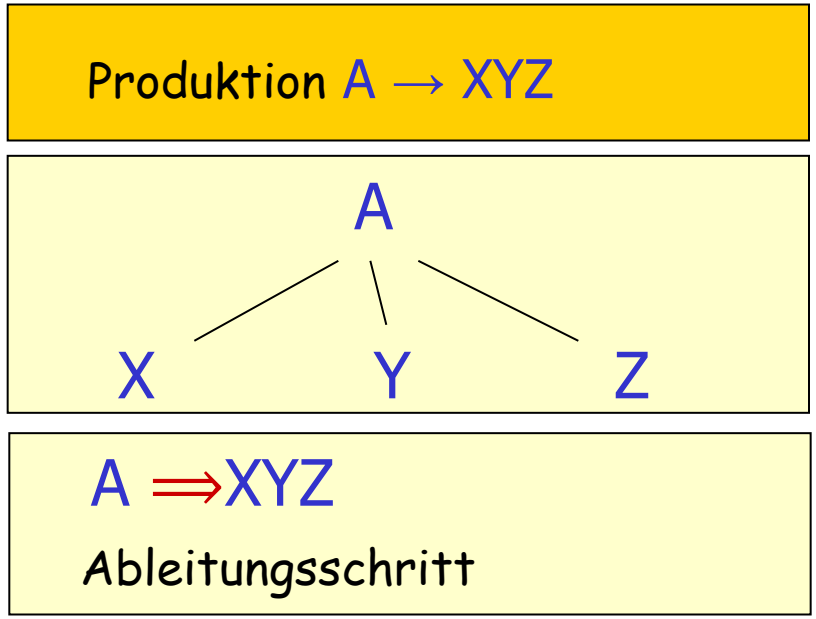
Syntaxbaum (Parse-Baum) allgemein



- … ist ein Baum, der die wichtigsten Bestandteile einer Ableitung zeigt.
- … stellt grafisch dar,
  wie aus dem Start-Symbol einer Grammatik ein Wort der Sprache hergeleitet wird.

Aufbau

- innere Knoten sind mit Variablen und
  Blattknoten mit Terminalsymbolen
  oder ε
  beschriftet.
- zu jedem inneren Knoten muss eine Produktion so vorhanden sein, dass
  - die linke Seite der Produktion
    die Beschriftung des Knotens bildet und
  - die Nachfolgerknoten (von links nach rechts gelesen)
    die rechte Seite der Produktion ergeben



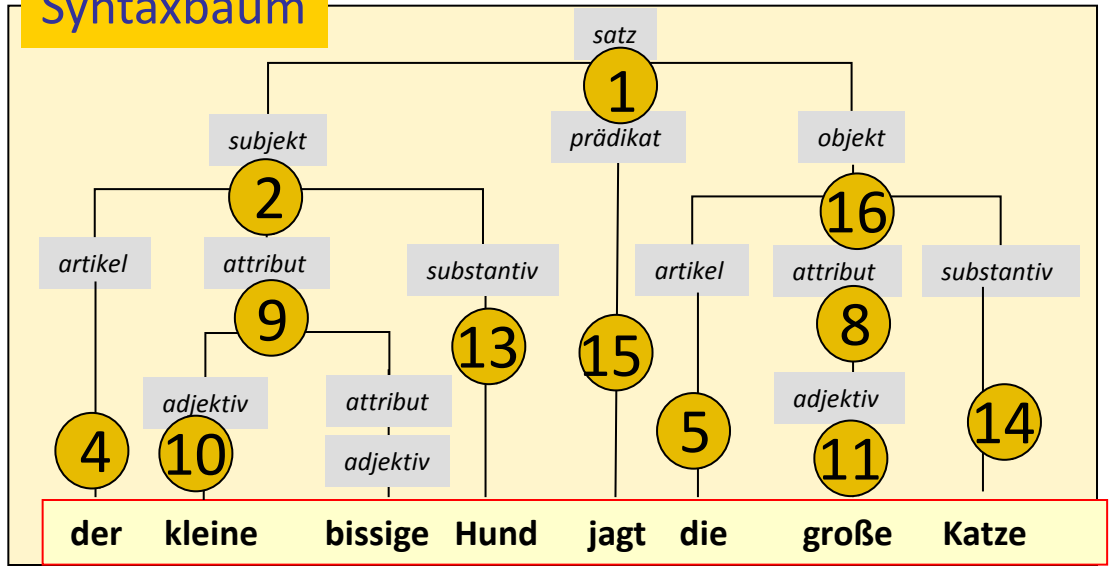
Grammatiken und Syntaxbäume

1. *satz* → *subjekt prädikat objekt*
2. *subjekt* → *artikel attribut substantiv*
3. *artikel* → ε
4. → **der**
5. → **die**
6. → **das**
7. *attribut* → ε
8. → *adjektiv*
9. → *adjektiv attribut*
10. *adjektiv* → **kleine**
11. → **große**
12. → **bissige**
13. *Substantiv* → **Hund**
14. → **Katze**
15. *prädikat* → **jagt**
16. *objekt* → *artikel attribut substantiv*

Symbole

a, b, c, ...	$\in \Sigma$
fettgedruckte Namen/Zeichen	$\in \Sigma$
u, v, w, ...	$\in \Sigma^*$
A, B, C, ...	$\in V$
kleingeschriebene kursive Namen	$\in V$
$\alpha, \beta, \gamma, \ldots$	$\in V^*$

Syntaxbaum



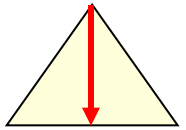
Methoden der Syntaxanalyse

Unterscheidung

in Abhängigkeit der Richtung, in der die Knoten des Syntaxbaums konstruiert werden

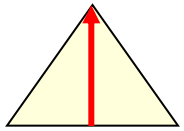
(1) Top-Down- Methoden

- Knotenkonstruktion beginnend mit Wurzel zu den Blättern
- sehr einfach,
  aber nur für eingeschränkte kfG-Teilklasse anwendbar !
  (genauere Charakterisierung erfolgt später)
- effiziente Parser lassen sich per Hand (ohne Zustandstabellen) erstellen



(2) Bottom-Up-Methoden

- umgekehrt: Knotenkonstruktion von den Blättern zur Wurzel
- erlaubt die Behandlung größerer Klassen von kfGs
  deshalb beliebter bei automatischer Parser-Generierung



Betrachten hier nur Parser mit **Leserichtung** des Eingabewortes:
von **links** nach **rechts**

Konstruktion von Syntaxanalysatoren

Varianten

(1) per Hand-Konstruktion als

- Umsetzung der allgemeinen Methode des rekursiven Abstiegs
- (Deterministischer) Kellerautomat
  universelle Automatenimplementation & individuelle Zustandsübergangstabellen (ähnlich dem Scanner-Ansatz)

(2) automatische Generierung

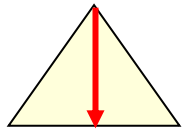
bei Identifikation verschiedener Unterklassen von kfGs

Von-Hand-Implementationen empfehlen sich **nicht**,
solange sich die Sprache, für den Parser erstellt werden sollen,
selbst noch in Entwicklung befindet

4.1 Einführung in die Syntaxanalyse

- Begriffsklärung und Klassifikation von Syntaxanalysatoren
- Prinzipielle Arbeitsweise eines Parsers
- Ein einfacher prädiktiver Parser (rekursiver Abstieg)
- Zusammenhang zwischen Parser und Grammatik-Unterklassen
- Mehrdeutigkeiten von Grammatiken

Prinzip des Parsens (1)



einfaches Beispiel einer kfG (Startsymbol: *type*)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

Pascal-Syntax (Ausschnitt)

Aufgabe: Parsen des Eingabewortes **array [num dotdot num] of integer**

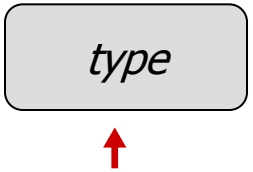
bei Aufbau des Ableitungsbaums

Prinzip des Parsens (1)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

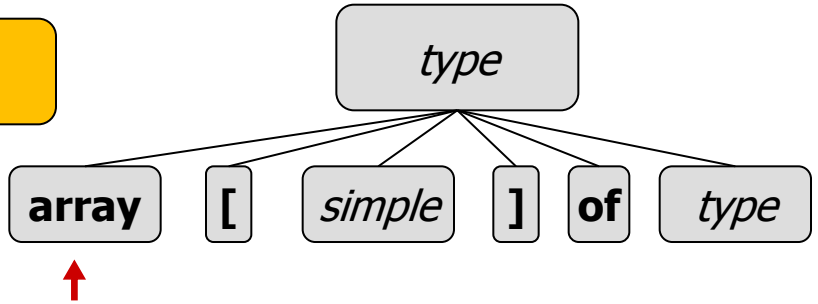
Schritt 1

Wurzel:= Startsymbol
Setzen von Markierungen
in Syntaxbaum u. Eingabe



array [num dotdot num] of integer

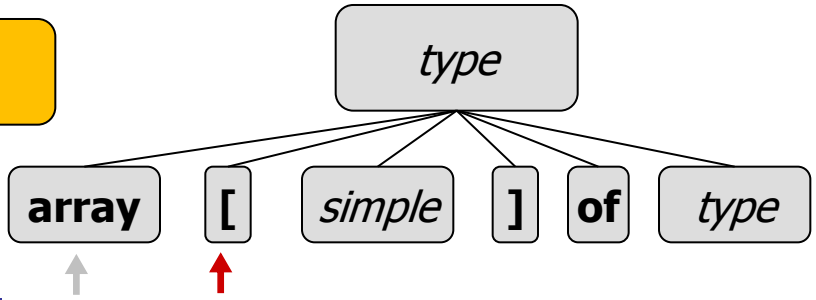
Schritt 2



array [num dotdot num] of integer

lookAhead → Auswahl von Regel 3
(eindeutig!)

Schritt 3



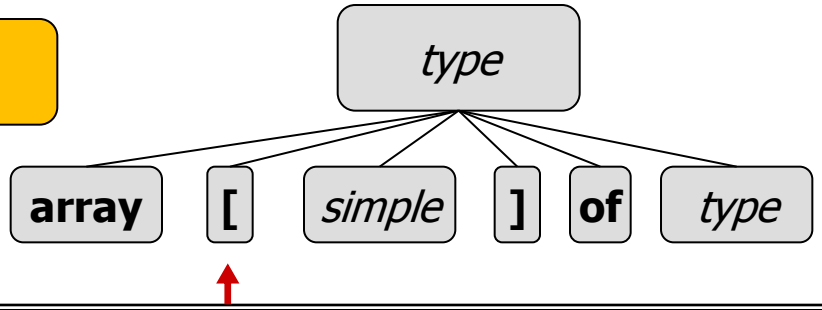
array **[** num dotdot num] of integer

wenn der gerade erzeugte Knoten (hier: **array**) ein Terminalknoten ist und mit lookAhead übereinstimmt, wird in der Eingabe und im Baum ein Schritt weiter gerückt

Prinzip des Parsens (2)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

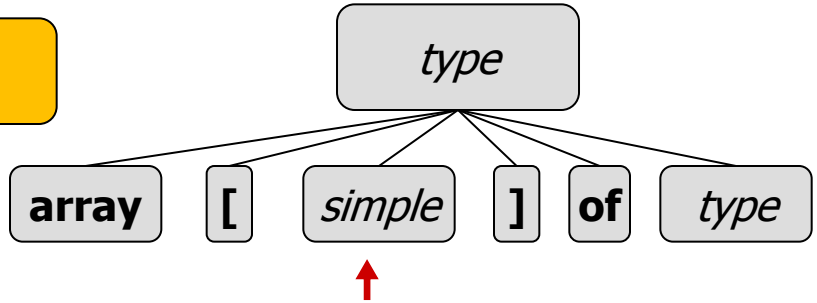
Schritt 4



array [num dotdot num] of integer

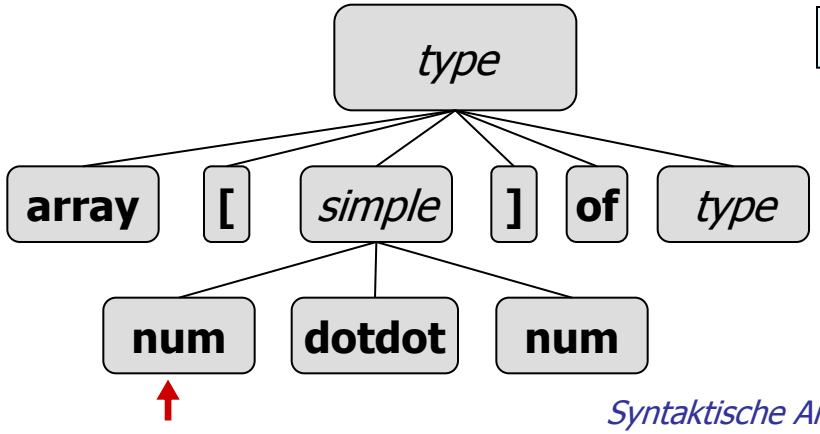
Klammer [kann „geschluckt" werden

Schritt 5



array [**num** dotdot num] of integer

es gibt nur eine Regel (6) zur Auswahl, die **num** aus *simple* produziert (1-deutig)



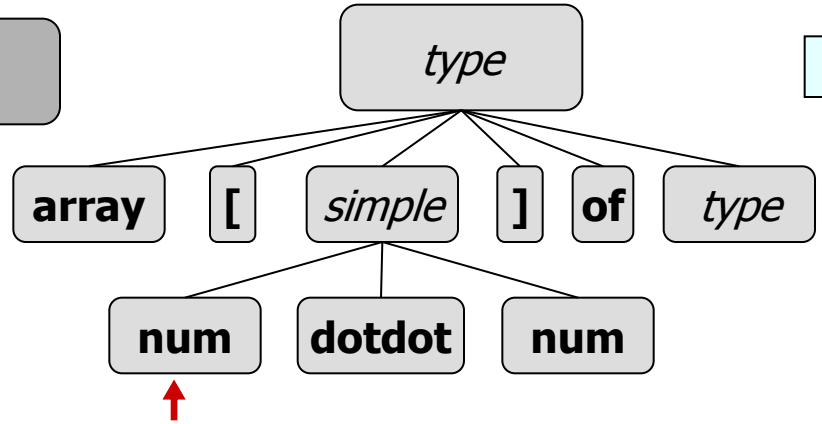
array [**num** dotdot num] of integer

Anwendung von Regel: [6]

Prinzip des Parsens (3)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

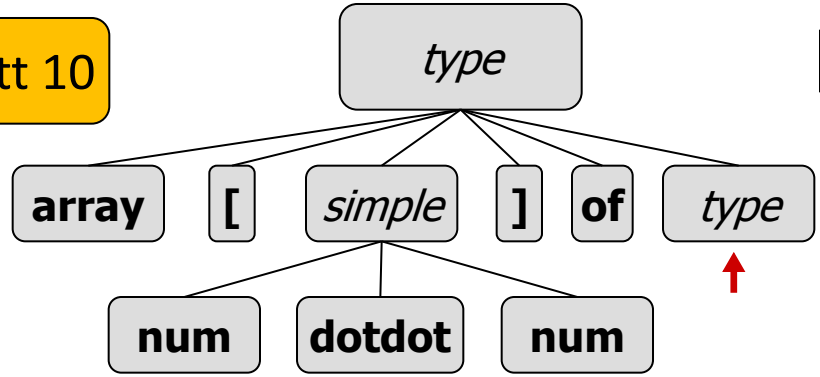
Schritt 5



array [**num** dotdot num] of integer

Folge von 5 Schritten, da Baumsymbole und entspr. lookAhead-Symbole übereinstimmen:
num, dotdot, num,], of

nach Schritt 10

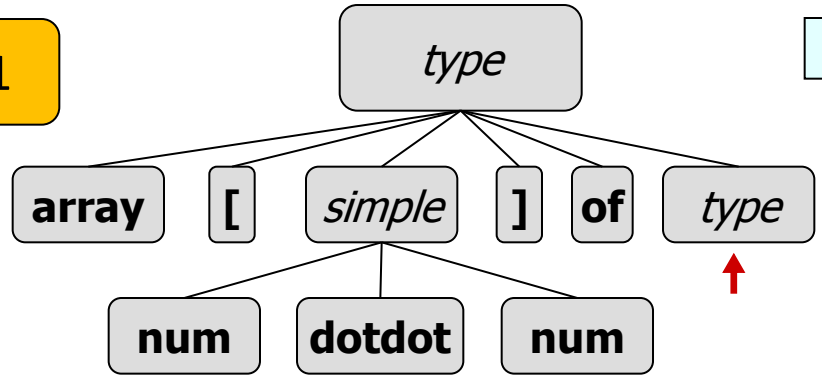


array [**num dotdot num**] of integer

Prinzip des Parsens (4)

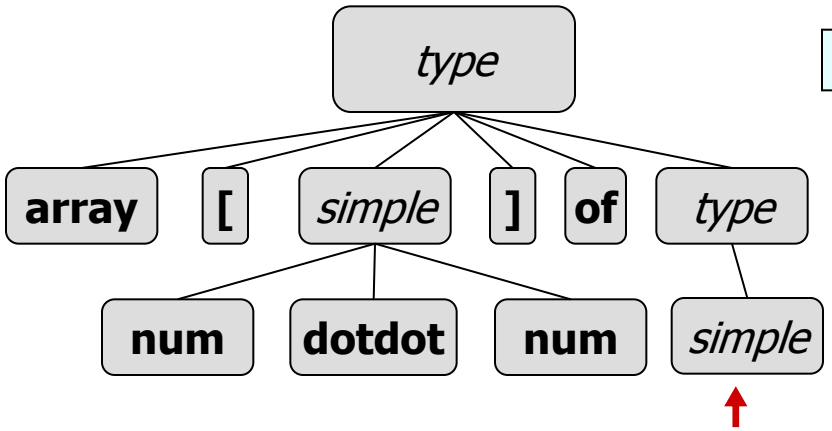
1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

Schritt 11



array [num dotdot num] of **integer**

es gibt nur eine Regel zur Auswahl, die für *type* anwendbar ist: [1]
([2] und [3] liefern nicht lookAhead)

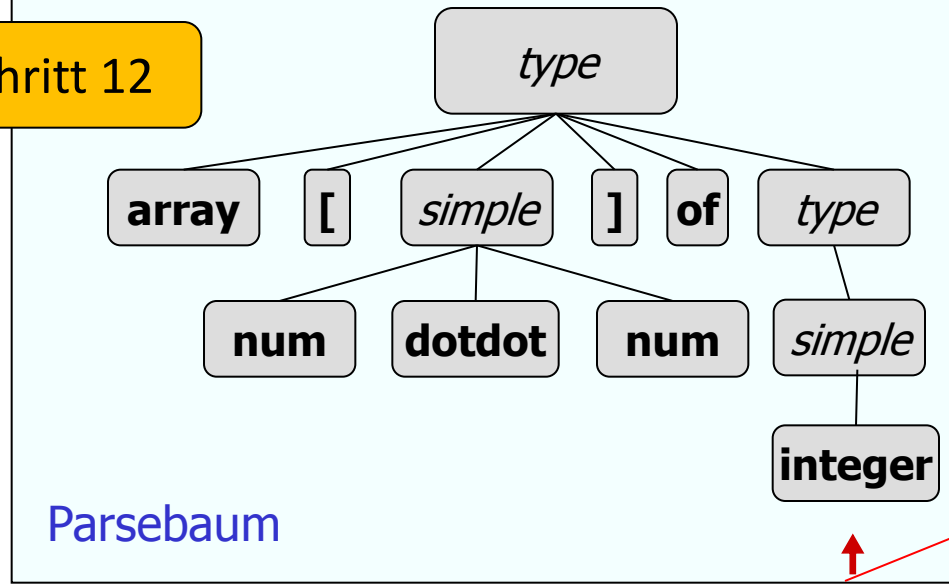


array [num dotdot num] of **integer**

Prinzip des Parsens (5)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

Schritt 12



Parsebaum

array [num dotdot num] of integer

nach Anwendung von [4]

letztes Zeichen wird „geschluckt"
FERTIG

Fazit

- Grammatik-Eigenschaften bestimmen Effizienz der Analyse
- es kann zu aufwändigen Rücksetzungen kommen, wenn gewählte Regel nicht zum Erfolg führt

Rekursiver Abstieg (falls anwendbar)
kommt ohne Rücksetzen aus

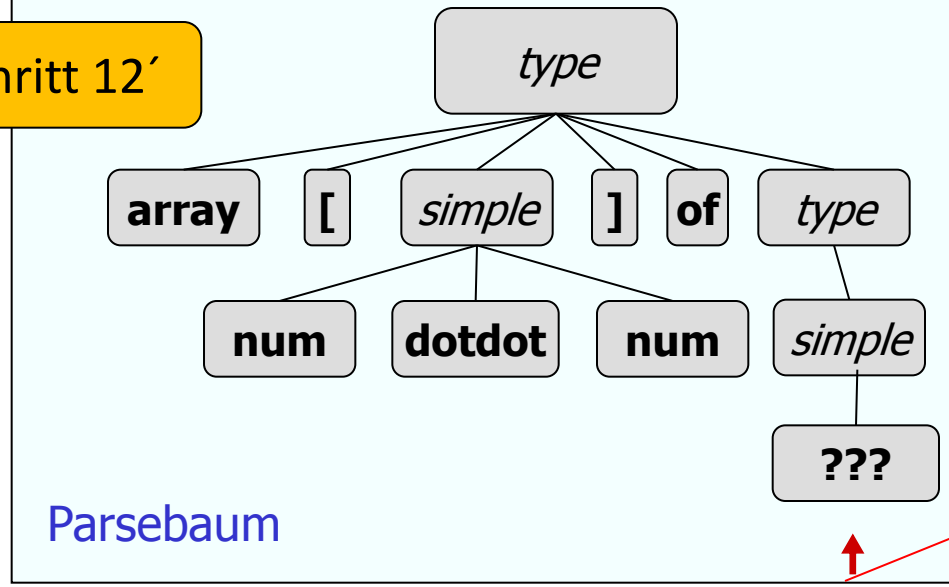
Beispiel-Grammatik:
sehr einfach: Regelauswahl problemlos

Allg. Fall:
Regelauswahl ist nicht immer 1-deutig

Prinzip des Parsens (6)

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

Schritt 12´



Parsebaum

array [num dotdot num] of id

Es gibt keine Regel, die *simple* nach **id** ableitet

und auch von *type* gibt es keinen Weg zu **id**

letztes Zeichen kann nicht „geschluckt" werden ABBRUCH

mit Fehlermeldung

Nehmen fehlerhafte Eingabe an:
array [num dotdot num] of id

bisheriger Ablauf bleibt erhalten, bis auf Schritt 12

4.1 Einführung in die Syntaxanalyse

- Begriffsklärung und Klassifikation von Syntaxanalysatoren
- Prinzipielle Arbeitsweise eines Parsers
- Ein einfacher prädiktiver Parser (rekursiver Abstieg)
- Zusammenhang zwischen Parser und Grammatik-Unterklassen
- Mehrdeutigkeiten von Grammatiken

Verfahren des rekursiven Abstiegs

Eingabe wird durch Menge rekursiver Prozeduren abgearbeitet

1 - für **jede Variable** X der Grammatik gibt es eine spezielle Prozedur p_X , die entscheidet, welche der Produktionen $X \rightarrow \alpha$ der Grammatik i. Abh. von LookAhead anzuwenden ist

- α ist **Variable**: **rekursive** Anwendung der Prozedur p_α
- α ist **Terminalsymbol**

  falls Übereinstimmung mit LookAhead : Anwendung der Prozedur match

  sonst: Anzeige eines Syntaxfehlers

2 - match- Prozedur: dient zur Vereinfachung des Codes der Nichtterminalsymbole:

falls Argument von match mit dem LookAhead-Symbol übereinstimmt,

wird zum nächsten Eingabesymbol übergegangen

sonst: Syntaxfehler

(match aktualisiert den Wert von lookAhead!)

3 - Beginn mit Prozedur für das Startsymbol p_S

Beispiel: Rekursiver Abstieg (1)

```
procedure simple();
begin
        if lookAhead == integer then
                match(integer)
        else if lookAhead == char then
                match(char)
        else if lookAhead == num then begin
                match(num); match(dotdot); match(num)
        end
        else error()
end;
```

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

```
procedure type();
begin
        if lookAhead is in {integer, char, num} then
                simple()
        else if lookAhead == '↑' then begin
                match('↑');match (id)
        end
        else if lookAhead == array then begin
                match(array); match('['); simple(); match(']'); match(of); type()
        end
        else error()
end;
```

```
procedure match (t: Tokentyp);
begin
        if lookAhead == t
            then
                lookAhead= nextToken()
            else
                error();
end;
```

Beispiel: Rekursiver Abstieg (2)

Parse-Prozess

- Start mit Prozedur für das Startsymbol (Top-Down, prädiktiv, rekursiv)

- Fortsetzung:
  das jeweils aktualisierte LookAhead-Symbol bestimmt,
  welche Produktion anzuwenden ist

  - beginnt rechte Seite mit einem Terminalsymbol, dann ist diese Produktion anwendbar
    wenn denn das Terminalsymbol mit dem LookAhead-Symbol übereinstimmt
  - …

- Beendigung bei vollständiger Verarbeitung der Eingabe oder Fehlererkennung

Beispiel-Ablauf

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

array [num dotdot num] **of integer**

Start

mit Prozedur für das Startsymbol

```
procedure type();
begin
        if lookAhead is in {integer, char, num} then
                simple()
        else if lookAhead ='↑' then begin
                match('↑'); match(id)
        end
        else if lookAhead = array then begin
                match(array); match('['); simple(); match(']'); match(of); type()
        end
        else error()
end;
```

entspricht der RS [3]

jedes Terminalsymbol wird mit lookAhead verglichen

jedes Metasymbol führt zum Aufruf der zugeordneten Prozedur

Beispiel-Ablauf

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

array **[num dotdot num] of integer**

Schritte 1,2

```
procedure type();
begin
        if lookAhead is in {integer, char, num} then
                simple()
        else if lookAhead ='↑' then begin
                match('↑'); match(id)
        end
        else if lookAhead = array then begin
                match(array); match('['); simple(); match(']');match(of); type()
        end
        else error()
end;
```

match(array) führt zur Weitersetzung von lookAhead: **[**

match('[');) zu lookAhead: **num**

Beispiel-Ablauf

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

array [num dotdot num] of integer

↑

weitere Schritte

```
procedure type();
begin
        if lookAhead is in {integer, char, num} the
                simple()
        else if lookAhead ='↑' then begin
                match('↑'); match(id)
        end
        else if lookAhead = array then begin
→               match(array); match('['); simple(); match(']');match(of); type()
        end
        else error()
end;
```

```
procedure simple();
begin
    if lookAhead = integer then
            match(integer)
    else if lookAhead =char then
            match(char)
    else if lookAhead = num then begin
            match(num); match(dotdot); match(num)
    end
    else error()
end;
```

1. Metasymbol: Aufruf der Prozedur für *simple*
   Akzeptanz von: **num**, **dotdot**, **num**
2. **]**, **of** und Aufruf der Prozedur für *type*

Beispiel-Ablauf

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**
6.	| **num dotdot num**

array [num dotdot num] of integer

weitere Schritte

```
procedure type();
begin
        if lookAhead is in {integer, char, num} then
                simple()
        else if lookAhead ='↑' then begin
                match('↑'); match(id)
        end
        else if lookAhead = array then begin
                match(array); match('['); simple(); match(']');match(of); type()
        end
        else error()
end;
```

1. type kann nur nach simple abgeleitet werden:
   Aufruf der Prozedur von *simple*

Beispiel-Ablauf

array [num dotdot num] of **integer**

1.	*type* → *simple*
2.	| ↑ **id**
3.	| **array [** *simple* **] of** *type*
4.	*simple* → **integer**
5.	| **char**

weitere Schritte

```
procedure type();
begin
        if lookAhead is in {integer, char, num} the
                simple()
        else if lookAhead ='↑' then begin
                match('↑'); match(id)
        end
        else if lookAhead = array then begin
                match(array); match('['); simple(); match(']');match(of); type()
        end
        else error()
end;
```

```
procedure simple();
begin
   if lookAhead = integer then
           match(integer)
   else if lookAhead =char then
           match(char)
   else if lookAhead = num then begin
           match(num); match(dotdot); match(num)
   end
   else error()
end;
```

1. Akzeptanz von: **integer** → Ende der Eingabe
2. Beendigung der Prozeduren

FRAGE

Kann man

die jeweiligen **Grammatikklassen** angeben,

die eine Syntaxanalyse von Programmen ihrer jeweils erzeugten Sprachen

durch prädiktive Parsern unterschiedlicher Bauart erlauben ?

Prädiktive Parser

zunächst der Typ: Rekursiver Top-Down-Analysator

4.1 Einführung in die Syntaxanalyse

- Begriffsklärung und Klassifikation von Syntaxanalysatoren
- Prinzipielle Arbeitsweise eines Parsers
- Ein einfacher prädiktiver Parser (rekursiver Abstieg)
- Zusammenhang zwischen Parser und Grammatik-Unterklassen
- Mehrdeutigkeiten von Grammatiken

Bekannt: Wortproblem und Grammatiktypen

Frage

- Gibt es einen Algorithmus, der bei Eingabe
  - einer Grammatik $G = (V, \Sigma, P, S)$ und
  - eines Wortes $x \in \Sigma^*$

  in endlicher Zeit entscheidet, ob $x \in L(G)$ oder $x \notin L(G)$?

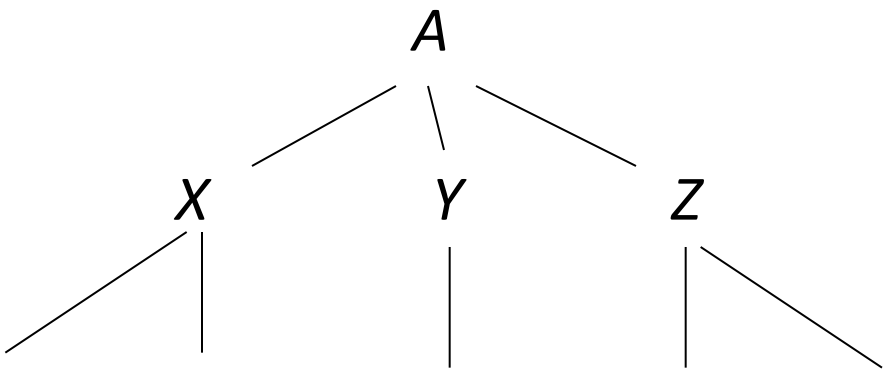
Antwort

- erfolgt in Abhängigkeit der Eigenschaften von G
  (und motiviert damit eine Klasseneinteilung von Grammatiken)

Ableitungen

Ableitung
ist der Prozess des Nachweises, dass ein
Eingabewort einer vorgegebenen Grammatik entspricht

$L(G) = \{w \in \Sigma^*, S \Rightarrow^* w\}$



$A \Rightarrow XYZ$

Ableitungsschritt

$X \Rightarrow \ldots$

$Y \Rightarrow \ldots$

$Z \Rightarrow \ldots$

z.B. immer linksseitige
oder rechtsseitige Ableitungen

Konflikt:
welcher Schritt zuerst?

LL(k)-Grammatiken

LR(k)-Grammatiken

LL(k)- und LR(k)-Grammatiken

LL-Grammatik: Grammatik, die von einem LL-Parser analysiert/ geparst werden kann

- L: Leserichtung (links → rechts)
- L: Ableitungsbildung (am weitesten links)

LR-Grammatik: Grammatik, die von einem LR-Parser analysiert/ geparst werden kann

- L: Leserichtung (links → rechts)
- R: Ableitungsbildung (am weitesten rechts)

k ist die Anzahl von LookAhead-Symbolen

LL(1) == LL

LR(1) == LR (k>1 möglich, aber praktisch ungünstiger)

Weitere Ableitungsvarianten

(1) zwar theoretisch möglich, aber schwierig zu implementieren

- rechtsseitige Top-Down-Ableitungen

  bzw.

- linksseitige Bottom-Ableitungen

(2) ebenso wäre auch ein Lesen von rechts nach links prinzipiell möglich

Theoretische Basis der syntaktischen Analyse

Wichtige Zusammenhänge: kfG – kf Sprachen - Kellerautomaten

- Zu jeder kontextfreien Grammatik lässt sich ein Kellerautomat konstruieren, der die von der Grammatik definierte Sprache akzeptiert.
- Die von einem Kellerautomaten akzeptierte Sprache ist kontextfrei, sie besitzt eine (sogar effektiv konstruierbare) kontextfreie Grammatik.
- Eine Sprache ist deterministisch kontextfrei genau dann, wenn sie von einem deterministischen Kellerautomaten erkannt wird

Grammatiktypen
nach Noam Chomsky

Antwort auf das **Wortproblem**:
erfolgt in Abhängigkeit der Eigenschaften von G
(und motiviert damit eine Klasseneinteilung von Grammatiken)

- **Typ-0**
  Phrasenstrukturgrammatik:
  keinerlei Einschränkung, *jede Grammatik ist vom Typ 0*

- **Typ-1**
  kontextsensitive Grammatik: falls zusätzlich für alle Regeln $w_1 \rightarrow w_2$ in P gilt:
  $|w_1| \leq |w_2|$

- **Typ-2**
  kontextfreie Grammatik: falls zusätzlich für alle Regeln w1→ w2 in P gilt:
  w1 ist eine einzelne Meta-Variable ($w_1 \in V_N$)

Suche nach geeigneten Unterklassen

- **Typ-3**
  reguläre Grammatik: falls zusätzlich gilt: dass $w_2 \in (\Sigma \cup \Sigma V)$
  d.h. rechte Seiten sind entweder ein einzelnes Terminalsymbol oder ein Terminalsymbol gefolgt von einem Meta-Symbol

Kontextfrei gegenüber kontextsensitiv

- **bei kontextfreier Regel:** $A \longrightarrow x$
  kann die Variable A – unabhängig vom Kontext, in dem A steht – bedingungslos durch x ersetzt werden

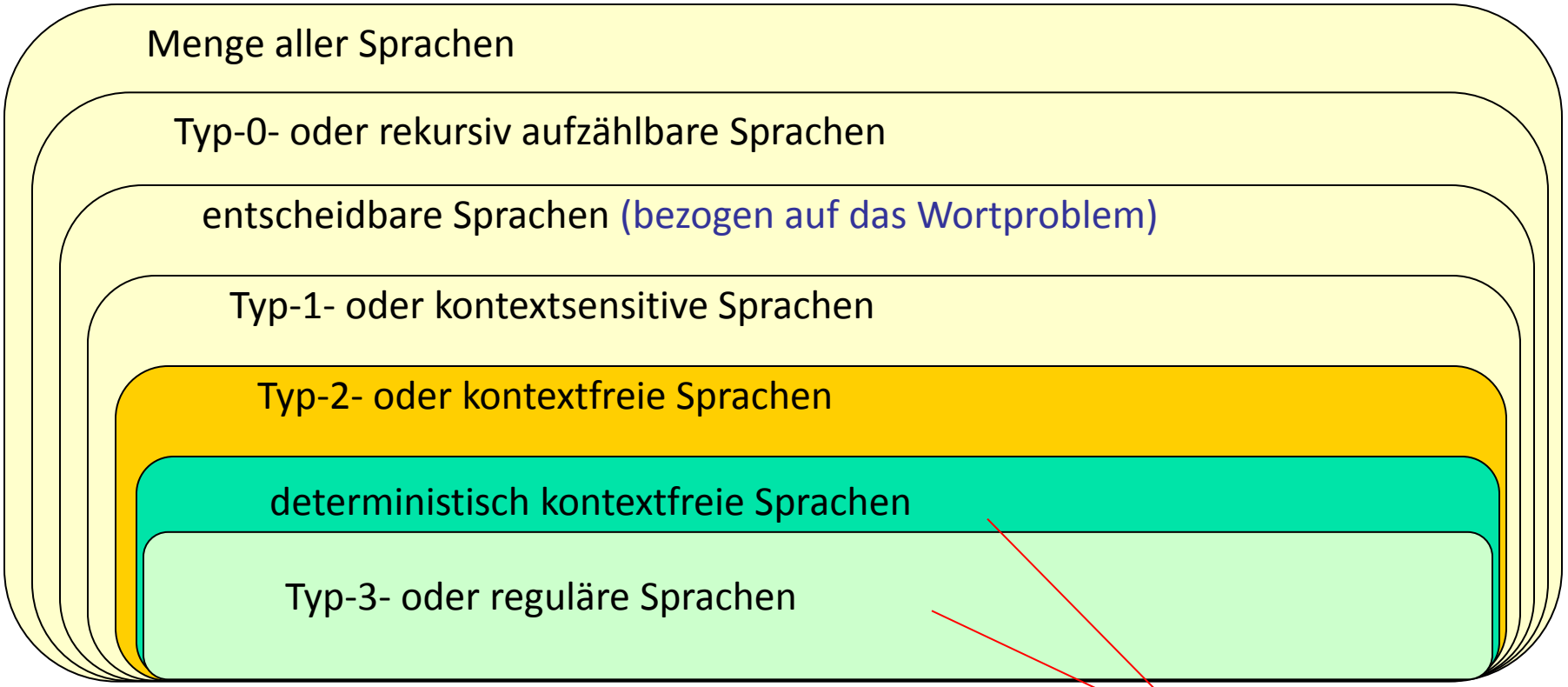
  stmt ⟶ **if (** *expr* **)** *stmt* **else** *stmt*

- **bei kontextsensitiver Regel**: $uAv \longrightarrow uxv$
  kann die Variable A durch x nur dann ersetzt werden,
  wenn die Variable A im Kontext zwischen u und v steht

(auch wenn sich dabei der Kontext (RS) ändern sollte: $uAv \longrightarrow u'xv'$)

Typen von Sprachen

Chomsky-Hierarchie: Typ-3 ⊂ Typ-2 ⊂ Typ-1 ⊂ Typ-0



spannende Sprachen für Compilerbau

Sprachen in der Praxis

... haben / benötigen aber Ausdruckskraft

- kontextsensitiver oder sogar Typ-0-Grammatiken

dennoch arbeitet man wegen schwieriger algorithmischer Handhabung von Typ-0,1-Sprachen lieber mit kontextfreien Grammatiken

z.B. in C: - Typverträglichkeit,
- korrekte Parameteranzahl beim Funktionsruf,
- ausschließliche Verwendung vorab deklarierter Objekte

lassen sich **nicht** durch kontextfreie Grammatiken beschreiben

Problemlösung

- Behandlung von real vorhandenen Kontextbedingungen und Sonderfällen erfolgt durch **nicht-grammatikalische** Zusatzalgorithmen

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität
Typ-3	Reguläre Grammatik DFA ←→ NFA ←→ RA	ja	linear, *O(n)*

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität
Typ-3	Reguläre Grammatik DFA ←→ NFA ←→ RA	ja	linear, *O(n)*
deterministisch kontextfrei	LR(k)- und LL(k)- Grammatiken Deterministischer Kellerautomat (DPDA)	ja	linear, *O(n)*

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität
Typ-3	Reguläre Grammatik DFA ←→ NFA ←→ RA	ja	linear, *O(n)*
deterministisch kontextfrei	LR(k)- und LL(k)- Grammatiken Deterministischer Kellerautomat (DPDA)	ja	linear, *O(n)*
Typ-2	Kontextfreie Grammatik (nichtdeterministischer) Kellerautomat (PDA)	ja	kubisch, $O(n^3)$

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität
Typ-3	Reguläre Grammatik DFA ←→ NFA ←→ RA	ja	linear, $O(n)$
deterministisch kontextfrei	LR(k)- und LL(k)- Grammatiken Deterministischer Kellerautomat (DPDA)	ja	linear, $O(n)$
Typ-2	Kontextfreie Grammatik (nichtdeterministischer) Kellerautomat (PDA)	ja	kubisch, $O(n^3)$
Typ-1	Kontextsensitive Grammatik Linear beschränkter Automat (LBA)	ja	exponentiell (NP-hart)

Zusammenfassung: Grammatiktypen, Wortproblem & Komplexität

Zeitbedarf für Erkennung eines Wortes bestehend aus n Token

Grammatik	Darstellungsmittel	Wortproblem entscheidbar?	Komplexität
Typ-3	Reguläre Grammatik DFA ←→ NFA ←→ RA	ja	linear, *O(n)*
deterministisch kontextfrei	LR(k)- und LL(k)- Grammatiken Deterministischer Kellerautomat (DPDA)	ja	linear, *O(n)*
Typ-2	Kontextfreie Grammatik (nichtdeterministischer) Kellerautomat (PDA)	ja	kubisch, $O(n^3)$
Typ-1	Kontextsensitive Grammatik Linear beschränkter Automat (LBA)	ja	exponentiell (NP-hart)
Typ-0	Typ-0- Grammatik, Phrasenstrukturgrammatik Turing-Maschine (TM)	nein	unlösbar

auf der linken Seite können auch bereits analysierte Terminale stehen, aus denen immer noch weiter abgeleitet werden kann

Zusammenfassung: Automatenäquivalenz

Nichtdeterministischer Automat	Deterministischer Automat	Äquivalenz ?
NFA	DFA	ja
PDA	DPDA	nein
LBA	DLBA	?
TM	DTM	ja

Zusammenfassung: Sprachen und Keller-Automaten

- **PushDown-Automat (PDA):**
  …ist ein nichtdeterministischer endlicher Automat
  mit einem Stack (zum Speichern von ZK beliebiger Länge).

  Der Stack kann nur von oben gelesen oder verändert werden.

- **Bewegung eines PDAs**:
  … erfolgt in Abhängigkeit vom
  - aktuellen Zustand,
  - des nächsten Eingabesymbols und
  - des obersten Kellerelementes.

  Bewegung kann auch erfolgen, ohne ein Eingabezeichen zu lesen.

  Bem: Da der PDA **nichtdeterministisch** ist, hat er die Wahl zwischen endlich vielen Bewegungsmöglichkeiten,
  wobei jede einen Zustand und eine ZK von Stacksymbolen angibt, durch die das obere Stacksymbol ersetzt wird.

Zusammenfassung: Sprachen und Keller-Automaten

- **Akzeptanz**:
  Ein PDA kann auf zweierlei Weise Akzeptanz signalisieren:
  - durch den Übergang in einen finalen Zustand oder
  - durch Leeren des Stacks.

- **Konfiguration**:
  … bestehend aus
  - Zustand,
  - verbleibende Eingabe und
  - Stackinhalt

  beschreibt die momentane Konfiguration.

  Eine Übergangsfunktion zwischen Konfigurationen beschreibt einzelne Bewegungen eines PDA.

- **PDA und Grammatiken:**
  die von einem PDA akzeptierten Sprachen sind die kf Sprachen.

Zusammenfassung: Sprachen und deterministische Keller-Automaten

- **Deterministischer PushDown-Automat (DPDA)**: … ist ein PDA,

  der zu
  - jedem gegebenen Zustand,
  - einem Eingabesymbol (inkl. ϵ) und
  - einem Stacksymbol

  höchstens eine Bewegungsmöglichkeit hat.

  Des weiteren kann er **nie** zwischen
  - einer Bewegung, die in Reaktion auf ein echtes Eingabesymbol erfolgt und
  - einer ϵ-Bewegung

  wählen.

- **vom DPDA akzeptierte Sprachen**:

  Alle regulären Sprachen werden durch Erreichen des **Endzustandes** akzeptiert,
  des weiteren werden auch nicht-reguläre Sprachen erkannt, und zwar kf Sprachen,
  die eine **eindeutige** Grammatik besitzen.

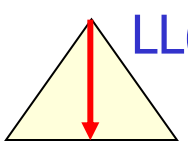
- Damit liegen die **Sprachen der DPDAs** echt zwischen den regulären und den kf Sprachen

Zusammenfassung: Wichtige Zusammenhänge

LL(k) $\subset$ LR(k)= det.kontextfreien Sprachen $\subset$ kontextfreien Sprachen

DPDA (determinisitscher Kellerautomat)

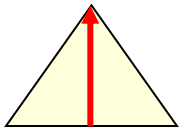
verlangt für Worterkennung die Eindeutigkeit der Grammatik



LL(k)-Grammatiken

Linksseitige Ableitungen

Beispielverfahren
Rekursiver Abstieg



LR(k)-Grammatiken

Rechtsseitige Ableitungen

Zusammenfassung: Parser-Techniken im Überblick (1)

Universelle Parse-Methoden (praktisch nicht interessant)
(Cocke-Younger-Kasami, Early)

- funktionieren für alle Grammatiken, aber nicht effizient

Deterministische Methoden (praktisch bedeutsam)

- Top-Down-Methoden
- Bottom-Up-Methoden

prädiktive Verfahren

- Leserichtung: links → rechts
- sind auf Grammatik-Teilklassen beschränkt

Simulation nichtdeterministischer Methoden (praktisch nicht interessant)

Backtracking-Verfahren

- Top-Down-Methoden
- Bottom-Up-Methoden

Zusammenfassung: Parser-Techniken im Überblick (2)

	prädiktive Verfahren	Backtracking-Verfahren
	Deterministisches Verfahren	Nichtdeterministisches Verfahren
Top-Down	überwiegend handgeschrieben (**LL-Sprachen**) z.B.: Rekursiver Abstieg	entspr. deterministische Simulationen
Bottom-Up	erzeugt durch Parser-Generatoren (**LR-Sprachen**)	entspr. deterministische Simulationen
Zeitverhalten	linear	exponentiell

Problem: wir benötigen (notwendige und hinreichende) Kriterien dafür,
wann eine kfG eine
LL- bzw.
LR-Grammatik ist