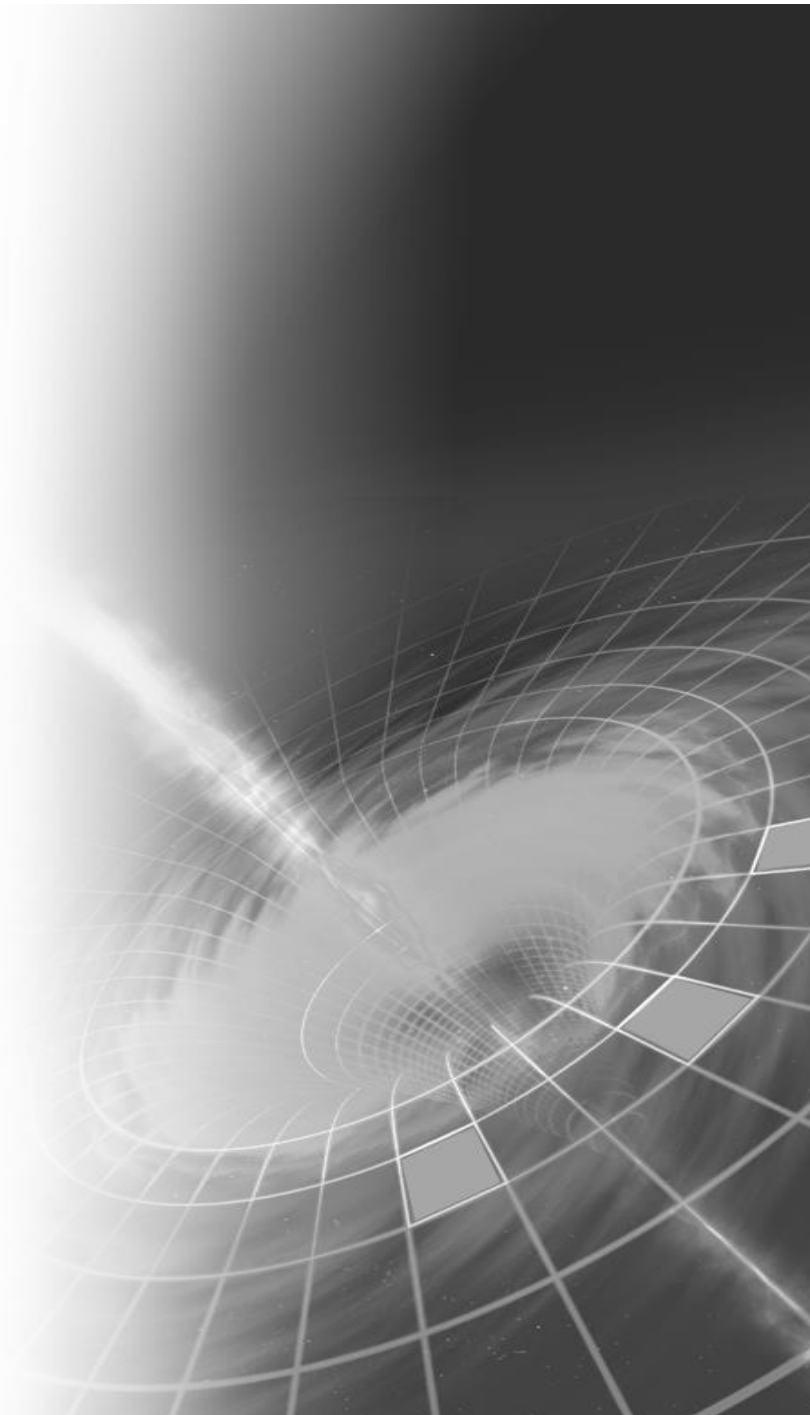
Bachelor-Programm

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 Programmie
- Teil II
  Methodische Gru
- Teil III
  Entwicklung ein

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

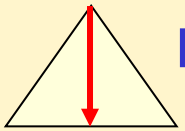
- 4.1
  Einführung in die Syntaxanalyse
- 4.2
  Restrukturierung von Grammatiken
- 4.3
  LL-Parser
- 4.4
  . . .

Linksseitige und Rechtsseitige Ableitungen (Wdh.)

Ableitung: ist i.allg. ein nichtdeterministischer Prozess, wobei
zunächst nicht vorgeschrieben, welches Nichtterminal abzuleiten ist,
wenn mehrere zur Auswahl stehen

Um diese Vielfalt besser zu beherrschen,
schränkt man sich etwas ein:

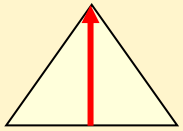
Man „normiert“ die Ableitungen: **linksseitig / rechtsseitig**



LL(k)-Grammatiken

nur
Linksseitige
Ableitungen

Beispielverfahren
Rekursiver Abstieg



LR(k)-Grammatiken

nur
Rechtsseitige
Ableitungen

LL(1)-Analysealgorithmus (Wdh.)

Herangehensweise

Benutzung von Parse-Prozeduren, um Terminalsymbole zu erkennen,
die durch Meta-Variablen akzeptiert werden (bei linksseitigem Aufbau des Parse-Baumes)

D.h.: Grammatik,
konnte unmittelbar in ein Programm umgewandelt werden kann!

Offen geblieben: Allgemeines Problem rekursiver Parse-Techniken

Bestimmung der anzuwendenden Produktionsregel

Ausgangssituation

sei G eine kf-Grammatik und

der Stand der Analyse sei mit der Auswertung der Regel

$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

bestimmt,

wo entschieden werden muss,
welche Produktionsregel für A anzuwenden ist

Beispielgrammatik (letzte Vorlesung)
bereitete kein Auswahlproblem

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

LL(1)-Grammatik (Wdh.)

bisherige Antwort:

- Regelauswahl sollte durch noch zu fordernde Eigenschaften einer LL(1)-Grammatik 1-deutig werden
- aber noch ohne Definition !

allgemein musste aber bereits gelten:

(1) das jeweilige **terminale** Wort ist an den Blättern des Baumes ablesbar

(2) für Sprachcompiler ist es notwendig, **1-deutige** Grammatiken zu entwerfen bzw. Mehrdeutigkeiten mit speziellen Regeln aufzulösen

Grund: sonst wären mehrdeutige semantische Interpretationen möglich

Einfache LL(1)-Eigenschaft

Die einfache LL(1)-Eigenschaft einer kfG ist gegeben, wenn

- die Grammatik keine ε-Produktion hat
- jede rechte Seite von Produktionen eines Nichtterminalsymbols A mit einem anderen Terminalsymbol beginnt

➔ Eine solche Grammatik heißt **einfache LL(1)-Grammatik**

Beispiel (Ausschnitt Java-Grammatik):

Stmnt → **identifier** = *Expr*;
| { *Stmnt* }
| **if** (*Expr*) *Stmnt* **else** *Stmnt*
| **for** (*Stmnt*; *Expr*; *Stmnt*) *Block_Stmnt*

für *Stmnt*

mit einem Lookahead von 1
kann die Regel 1-deutig bestimmt werden

Für die Praxis sind diese Grammatiken aber zu eingeschränkt

Verletzung der LL(k)-Eigenschaft

Es gibt zwei Klassen kotextfreier Grammatiken,
die in der Praxis zur Syntaxdefinition von Programmiersprachen
verwendet werden und die LL(k)-Eigenschaft für kein k erfüllen:

- mehrdeutige Grammatiken
- linksrekursive Grammatiken

➔ Einige mehrdeutige Grammatiken kann man in 1-deutige Grammatiken überführen (es gibt aber inhärent mehrdeutige Grammatiken)

➔ Linksrekursion kann durch Transformation der Grammatik eliminiert werden

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

Mehrdeutige kontextfreie Grammatiken

Äquivalenzen und Mehrdeutigkeiten

- **Äquivalenz von Parsebäumen und Ableitungen:**
  Ein Wort gehört zur Sprache der Grammatik
  g.d.w., es das Ergebnis mindestens einer entsprechenden Parsebaum-Konstruktion ist

  Das Vorhandensein
  - einerseits von
    links- und rechtsseitigen Ableitungen
  - sowie andererseits
    von Parse-Bäumen

  sind äquivalente Bedingungen für die Definition der Worte der Sprache einer kfG

- **Mehrdeutige Grammatiken:**
  Bei einigen kfGs ist es möglich,
  dass ein Wort über **mehrere** linksseitige und **mehrere** rechtseitige Ableitungen verfügt

  Eine solche Grammatik wird **mehrdeutig** genannt

„Fluch und Segen“ von Mehrdeutigkeit

- Für viele nützliche mehrdeutige Grammatiken (typische Programmiersprachen) lassen sich jeweils eine 1-deutige Grammatik finden, die dieselbe Sprache erzeugt

  Leider sind diese 1-deutige Grammatiken häufig viel komplexer als die Ausgangsgrammatik (deshalb sind mehrdeutige Grammatiken durchaus bei Sprachdefinitionen beliebt)

- ABER es gibt auch kf Sprachen, die als **inhärent mehrdeutig** bezeichnet werden, da jede Grammatik dieser Sprache mehrdeutig ist.

- Das Problem, ob eine (beliebige) kontextfreie Grammatik mehrdeutig oder nicht-mehrdeutig ist, ist allgemein **nicht entscheidbar**.

  Häufig kann aber Mehrdeutigkeit ausgeschlossen werden, falls die kontextfreie Eingabe-Grammatik **Element einer bestimmten Teilklasse** von kontextfreien Grammatiken ist.

Man muss also vermeiden, eine Sprache so zu konstruieren, für die man keine eindeutige Grammatik finden kann.

Überprüfung einiger Aussagen zu Ableitungen

1) in jedem Ableitungsschritt wird eine Variable zur Ersetzung gewählt,
Frage: aber, welche?

Wahl bestimmt unterschiedlichen Ableitungstyp, z.B.:

- **Linksableitung**
- **Rechtsableitung**

2) **Frage**: Sind die entstehenden Syntaxbäume trotzdem gleich?

3) Falls mehrere Ableitungsmöglichkeiten bei Ersetzung einer Variablen existieren
Fragen:
- Sind die entstehenden Syntaxbäume immer noch gleich?
- **Haben die so analysierten/akzeptierten Worte dieselbe Bedeutung?**

z.B. Baum-Interpretation: Links-Tiefe-zuerst?

Mehrere Ableitungsbäume für eine kfG

Beispiel: Grammatik für einfache Ausdrücke

1.	*goal* → *E*
2.	*E* → *E op E*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| +
7.	| -
8.	| /

Eingabewort: x + 2*y

… erzeugt **aber mehr als eine** Links- und **mehr als eine** Rechtsableitung für dasselbe Eingabewort.

 Mehrdeutigkeit

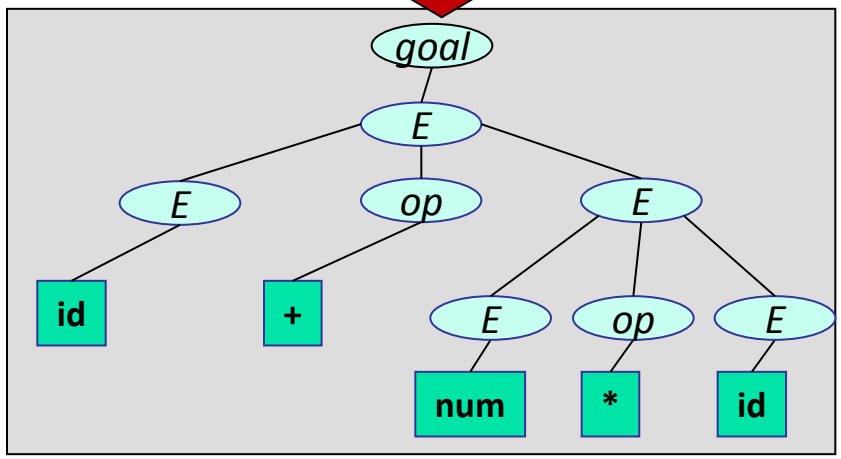
Experiment: eine weitere Linksableitung

Eingabewort: x + 2*y

goal	⇒	*E*	[1]
	⇒	*E op E*	[2]
	⇒	**id** *op E*	[4]
	⇒	**id +** *E*	[6]
	⇒	**id +** *E op E*	[2]
	⇒	**id + num** *op E*	[3]
	⇒	**id + num *** *E*	[5]
	⇒	**id + num * id**	[4]

alternativ [2]

1.	*goal* → *E*
2.	*E* → *E op E*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| **+**
7.	| **-**
8.	| **/**



Erzeugung eines semantisch verschiedenen Syntaxbaumes → Grammatik ist mehrdeutig

x + (2 * y)

Zweite mögliche Rechts-Ableitungen

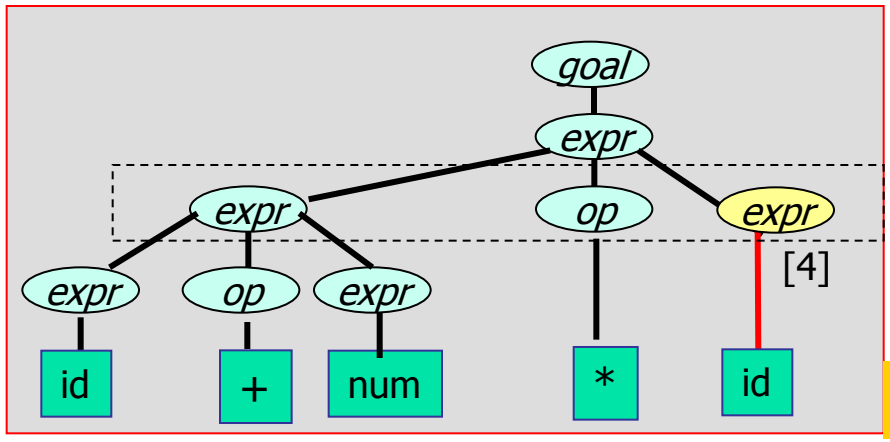
Eingabewort: x + 2*y

1.	*goal* → *expr*
2.	*expr* → *expr op expr*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| +
7.	| -
8.	| /

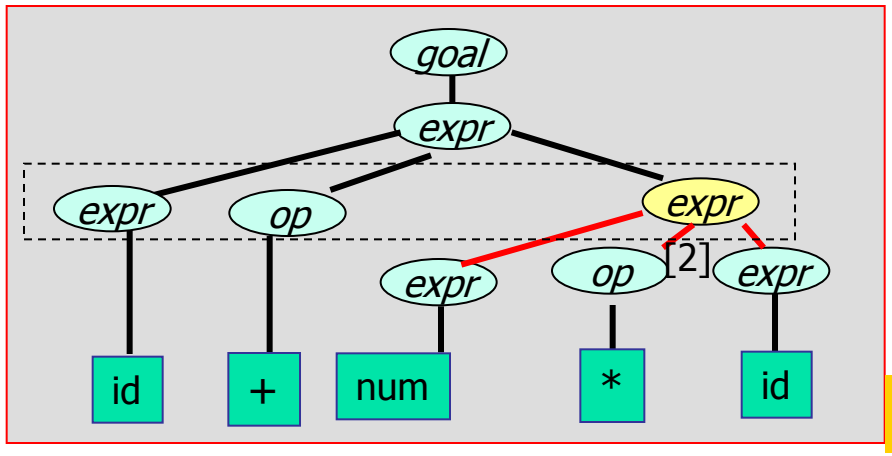
1. Rechts-Ableitung: [1] [2] [4]

Auswahl: nicht eindeutig

2. Rechts-Ableitung: [1] [2] [2]



(x+2)*y



x+(2*y)

unterschiedliche Bedeutungen

Zusammenfassung: Experimentergebnisse

1. für Eingabewort: x + 2*y

   - 2. Links- und 2. Rechtsableitung → x+(2*y)
   - 1. Links- und 1. Rechtsableitung → (x+2)*y

2. die Art der Ableitung (Links- oder Rechtsableitung)
   hat keinen Einfluss
   auf den abgeleiteten Syntaxbaum,
   solange bei den unterschiedlichen Varianten die gleichen Entscheidungen getroffen werden, welche der von den möglichen Ersetzungsregeln jeweils zum Einsatz kommt

3. bei Änderung der Regelauswahl (falls es mehr als eine Regel zur Ableitung einer Variablen gibt) wird ein anderer Syntaxbaum aufgebaut

   wichtige Nebenbedingung:
   dabei muss sich natürlich stets das Eingabewort ableiten lassen

Position

- Teil I
  Die Programmie
- Teil II
  Methodische Gru
- Teil III
  Entwicklung ein

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Lexikalische 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
- 4.2
  Restrukturierung von Grammatiken
- 4.3
  LL-Parser

4.2 Restrukturierung von Grammatiken

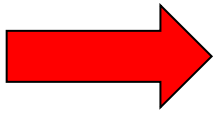
- Ursache von Mehrdeutigkeiten (kontextfreie Mehrdeutigkeiten)
- Eliminierung von Mehrdeutigkeiten
  als Voraussetzung für die Konstruktion von LL- und LR-Parsern
- erste Betrachtung von Übersetzungstechniken
  (am Beispiel von Übersetzungsschemata)
- Problem der Linksrekursion
  rekursiver Parser und Übersetzungsschemata

Prinzipieller Umgang mit Mehrdeutigkeiten

- Für Parser ist eine mehrdeutige Grammatik **nicht** wünschenswert

  In den meisten Fällen gelingt es aber,
  durch **Umformulierungen** eine 1-deutige Grammatik zu konstruieren

1.	*goal* → *E*
2.	*E* → *E op E*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| **+**
7.	| **-**
8.	| **/**



Suche nach
1-deutiger Grammatik,
die **übliche Vorrangregeln von Operatoren** festlegt

- Es gibt aber Fälle (wenn die 1-deutige Grammatik zu komplex wird),
  wo es günstiger ist,
  mehrdeutige Grammatiken in Verbindung mit **Eindeutigkeitsregeln** zu benutzen, die unerwünschte Parse-Bäume verwerfen

Typische Ursache für Mehrdeutigkeit

- häufige Ursache:
  unklare/verwirrende Spezifikation der Regeln einer kontextfreien Grammatik

  z.B. durch Überladung a = f (17)

  f kann dabei (in Algol-ähnlichen Sprachen)
  - eine Funktion oder
  - ein Array sein

- Auflösung der Mehrdeutigkeit durch Kontextwissen
  - Zugriff auf Deklarationen über Symboltabelle (später) (Typerkennung)
  - Einführung von Regelprioritäten (in generierten Parsern)

- für die **deterministisch-kontextfreien Sprachen LL(k) und LR(k)** sind die Grammatiken jedoch eindeutig
  anders ausgedrückt:

  Grammatiken für LL(k)- und LR(k)-Parser müssen eindeutig sein

4.2 Restrukturierung von Grammatiken

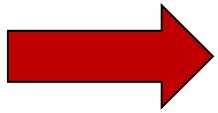
- Ursache von Mehrdeutigkeiten (kontextfreie Mehrdeutigkeiten)
- Eliminierung von Mehrdeutigkeiten
  als Voraussetzung für die Konstruktion von LL- und LR-Parsern
- erste Betrachtung von Übersetzungstechniken
  (am Beispiel von Übersetzungsschemata)
- Problem der Linksrekursion
  rekursiver Parser und Übersetzungsschemata

Beispiel-2: Eliminierung von Mehrdeutigkeiten

Restrukturierung einer mehrdeutigen Ausdrucksgrammatik
bei Beachtung von Operatorpräzedenzen

ursprüngliche mehrdeutige Grammatik

1.	*goal* → *E*
2.	*E* → *E op E*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| +
7.	| -
8.	| /



Einführung
von zwei zusätzlichen
Variablen: T, F

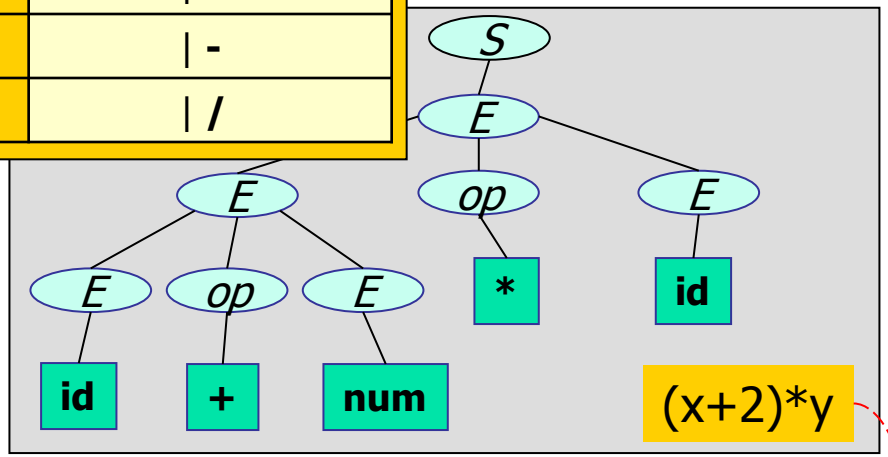
neue eindeutige Grammatik

1.	*goal* → *E*
2.	*E* → *E* + *T*
3.	| *E* - *T*
4.	| *T*
5.	*T* → *T* * *F*
6.	| *T* / *F*
7.	| *F*
8.	*F* → **num**
9.	| **id**

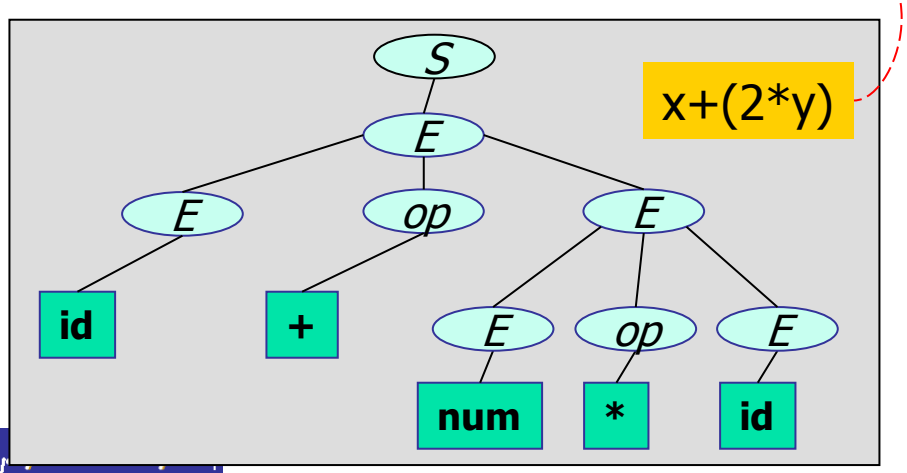
1.	*S* → *E*
2.	*E* → *E op E*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| +
7.	| -
8.	| /

1.	*S* → *F*
2.	*E* → *E* + *T*
3.	| *E* - *T*
4.	| *T*
5.	*T* → *T* * *F*
6.	| *T* / *F*
7.	| *F*
8.	*F* → **num**
9.	| **id**

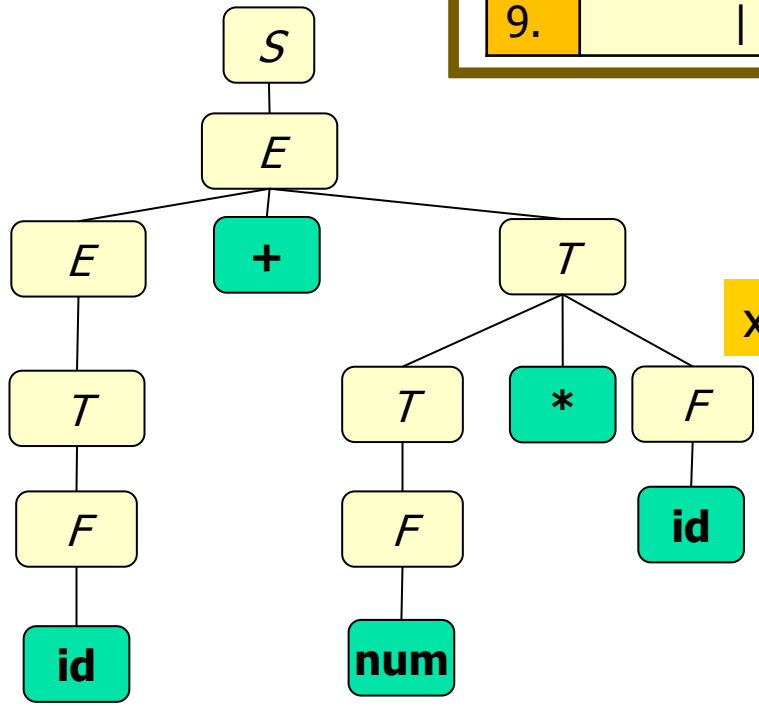
x+2*y



(x+2)*y



x+(2*y)



x+(2*y)

Kontextfreie Mehrdeutigkeit von Grammatiken

Mehrdeutigkeiten in Grammatiken für reale Sprachen
sind häufig „hausgemacht“,

Ursache ist die kontextfreie Formulierung kontextabhängiger
Sachverhalte

Beispiel-3

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| ... *weitere*

- für die syntaktische Einheit:
  if E_1 **then if** E_2 **then** S_1 **else** S_2
  gibt es mehr als einen links- bzw. rechtsseitigen Ableitungsbaum
  (und damit unterschiedliche Bedeutungen)

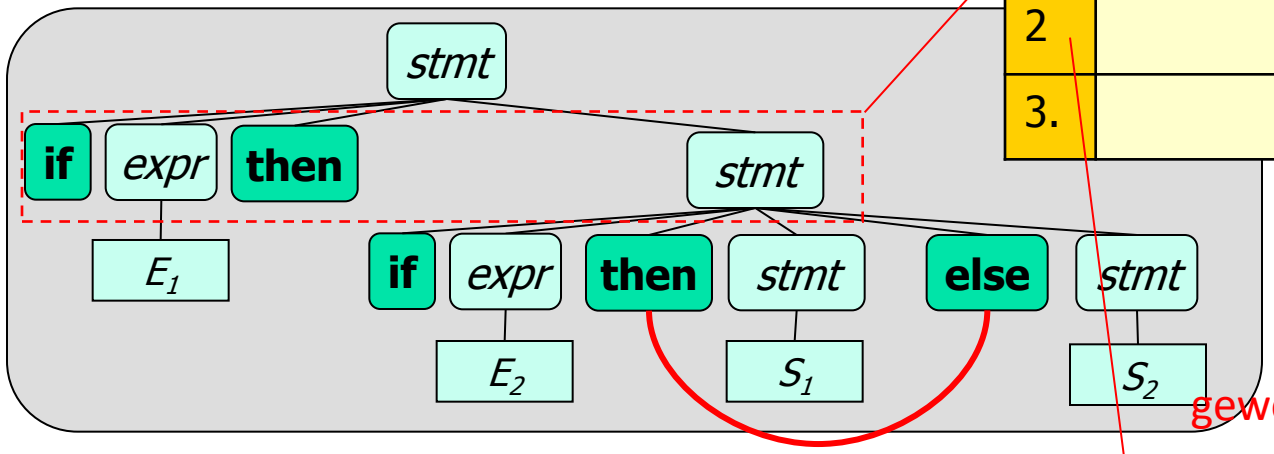
- Grammatik ist mehrdeutig
  sie ist durch die Kontextfreiheit der Grammatik bestimmt (kontextfreie Mehrdeutigkeit)

Ableitungsbäume für if-then-else

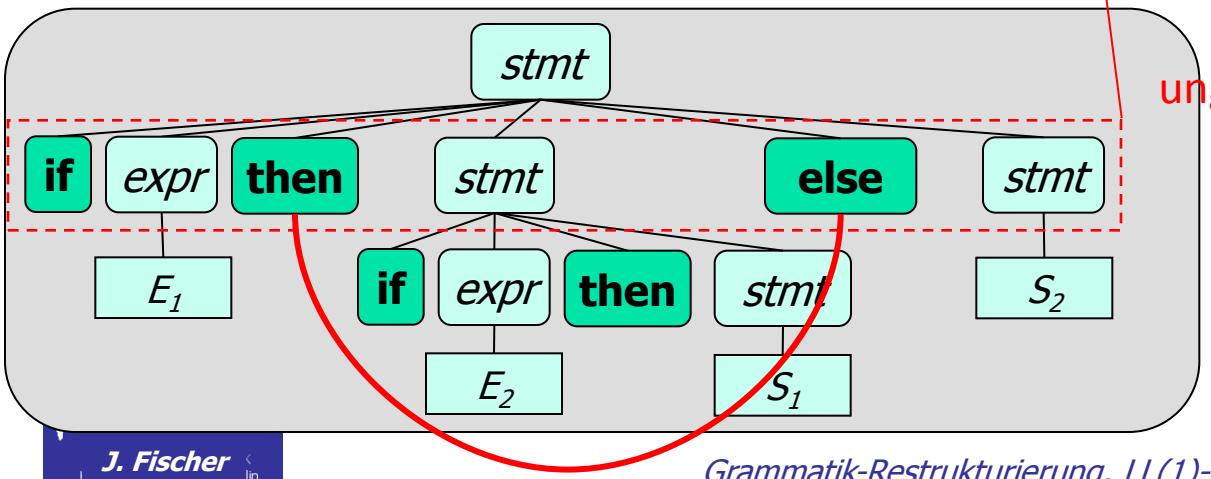
Eingabe: if *E1* then if *E2* then *S1* else *S2*

1.	*stmt* ::= **if** *expr* **then** *stmt*
2	| **if** *expr* **then** *stmt* **else** *stmt*
3.	| *other*

other - irgendeine andere Anweisung



gewöhnliche Semantik



ungewöhnliche Semantik

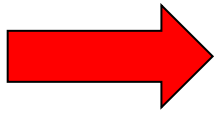
semantischer Unterschied:
für Zugehörigkeit vom
else-stmt

Beispiel-3: Eliminierung von Mehrdeutigkeiten

1.	*stmt ::=* **if** *expr* **then** *stmt*
2	| **if** *expr* **then** *stmt* **else** *stmt*
3.	| *other*

zwar schlechter lesbar, aber dennoch häufig verwendete **Restrukturierungsvariante** der Grammatik:

zwei zusätzliche Metasymbole



1.	*stmt*	→ *matched*
2.		| *unmatched*
3.	*matched*	→ **if** *expr* **then** *matched* **else** *matched*
4.		| *other*
5.	*unmatched*	→ **if** *expr* **then** *stmt*
6.		| **if** *expr* **then** *matched* **else** *unmatched*

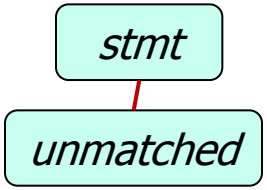
- generiert die gleiche Sprache mit der üblichen Regel:

Ordne ein **else** dem am nächsten stehenden (noch) nicht zugeordneten **then** zu

Eindeutige Grammatik für: if-then-else (1)

Eingabe: **if** *E1* **then if** *E2* **then** *S1* **else** *S2*

Ableitungen: [2]

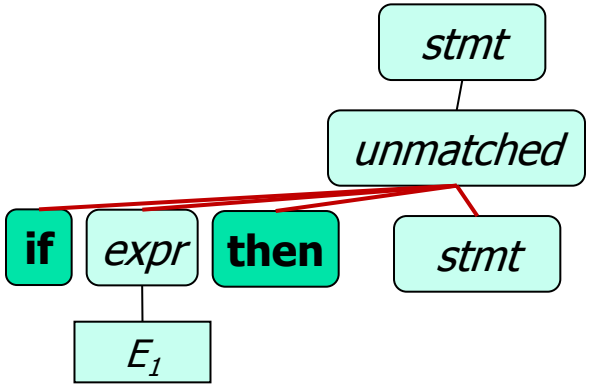


1.	*stmt* → *matched*
2.	| *unmatched*
3.	*matched* → **if** *expr* **then** *matched* **else** *matched*
4.	| *other stmts*
5.	*unmatched* → **if** *expr* **then** *stmt*
6.	| **if** *expr* **then** *matched* **else** *unmatched*

Eindeutige Grammatik für: if-then-else (1)

Eingabe: **if** *E1* **then if** *E2* **then** *S1* **else** *S2*

Ableitungen: [2],[5]

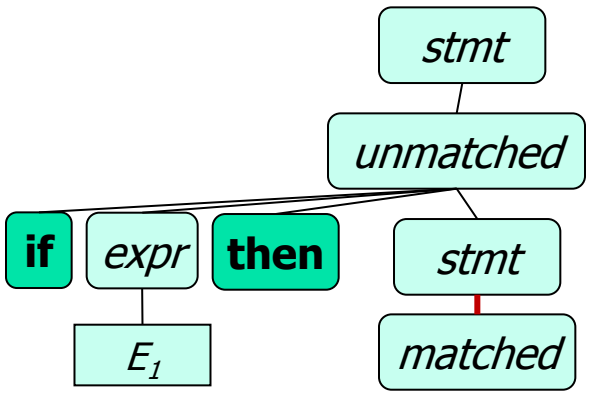


1.	*stmt* → *matched*
2.	| *unmatched*
3.	*matched* → **if** *expr* **then** *matched* **else** *matched*
4.	| *other stmts*
5.	*unmatched* → **if** *expr* **then** *stmt*
6.	| **if** *expr* **then** *matched* **else** *unmatched*

Eindeutige Grammatik für: if-then-else (1)

Eingabe: **if** *E1* **then if** *E2* **then** *S1* **else** *S2*

Ableitungen: [2],[5],[1]



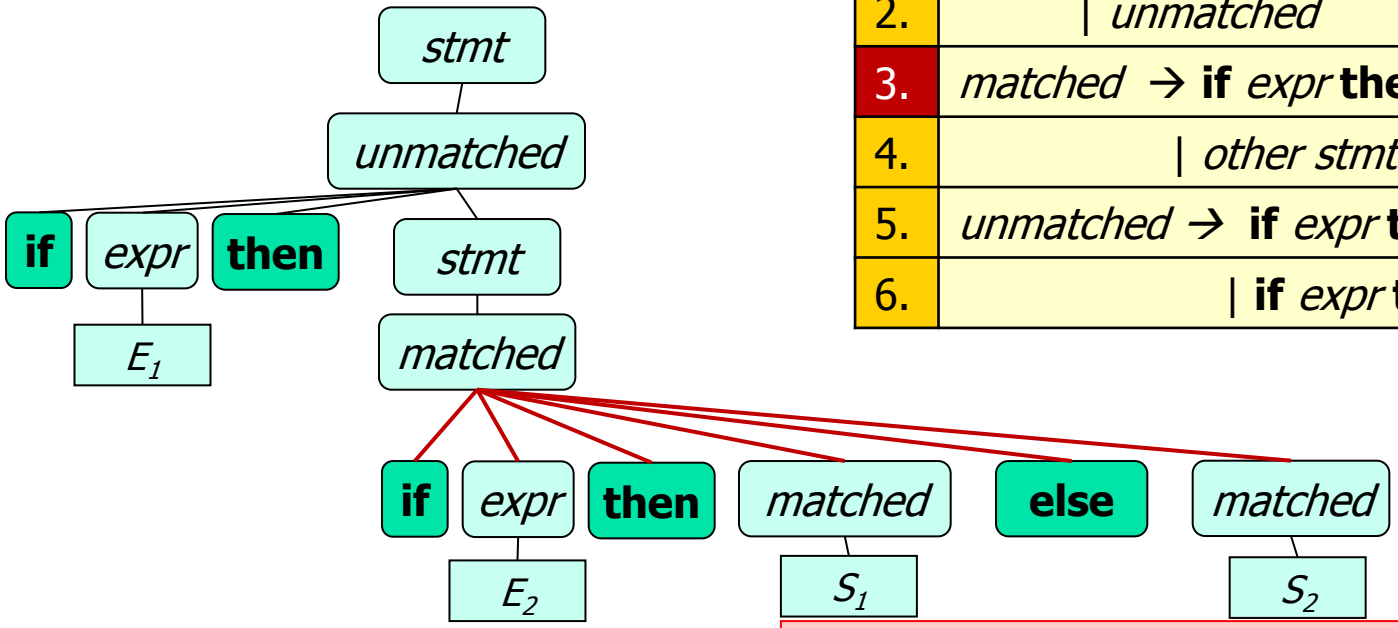
1.	*stmt* → *matched*
2.	| *unmatched*
3.	*matched* → **if** *expr* **then** *matched* **else** *matched*
4.	| *other stmts*
5.	*unmatched* → **if** *expr* **then** *stmt*
6.	| **if** *expr* **then** *matched* **else** *unmatched*

Eindeutige Grammatik für: if-then-else (1)

Eingabe: **if** *E1* **then if** *E2* **then** *S1* **else** *S2*

Ableitungen: [2],[5],[1],[3]

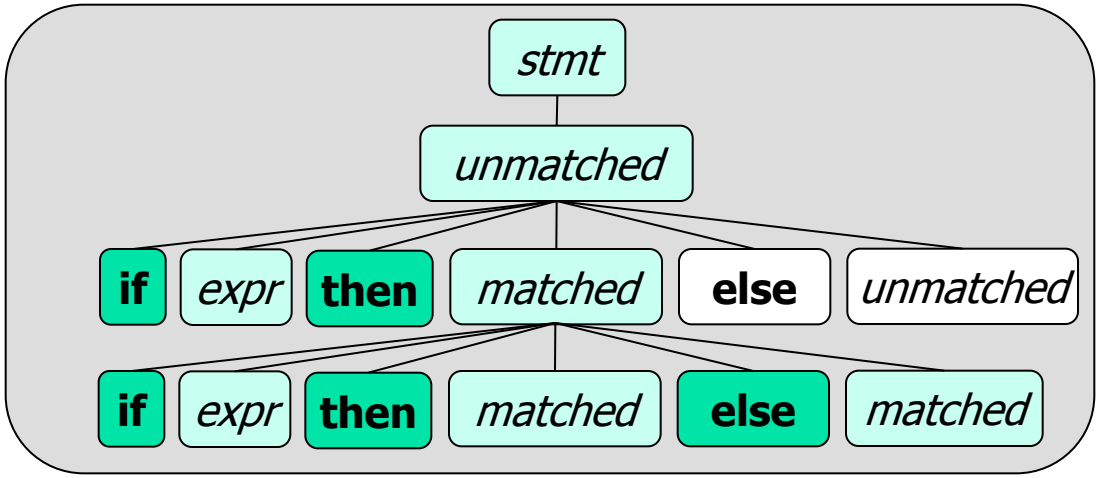
1.	*stmt* → *matched*
2.	| *unmatched*
3.	*matched* → **if** *expr* **then** *matched* **else** *matched*
4.	| *other stmts*
5.	*unmatched* → **if** *expr* **then** *stmt*
6.	| **if** *expr* **then** *matched* **else** *unmatched*

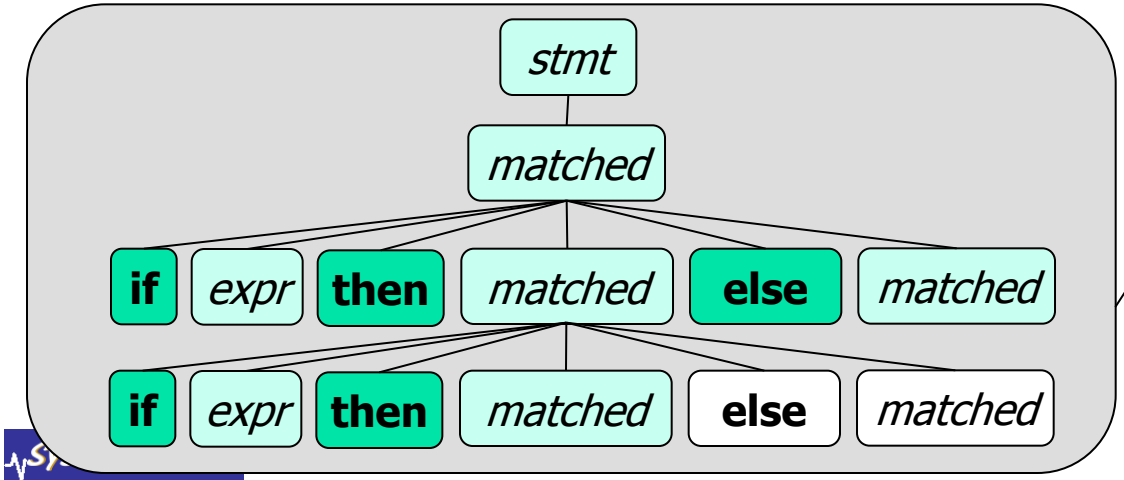


Grammatik erzeugt die gleiche Sprache, lässt aber nur einen Syntaxbaum für die Eingabe zu

Eindeutige Grammatik für: if-then-else (2)

Eingabe: if *E1* then if *E2* then *S1* else *S2*





1.	*stmt* → *matched*
2.	| *unmatched*
3.	*matched* → **if** *expr* **then** *matched* **else** *matched*
4.	| *other stmts*
5.	*unmatched* → **if** *expr* **then** *stmt*
6.	| **if** *expr* **then** *matched* **else** *unmatched*

Ableitungen: [2],[6],[3]

Anwendung von Alternativ-Regel führt nicht zu einer erfolgreichen Wortableitung

Ableitungen: [1],[3],[3]

4.2 Restrukturierung von Grammatiken

- Ursache von Mehrdeutigkeiten (kontextfreie Mehrdeutigkeiten)
- Eliminierung von Mehrdeutigkeiten
  als Voraussetzung für die Konstruktion von LL- und LR-Parsern
- erste Betrachtung von Übersetzungstechniken
  (am Beispiel von Übersetzungsschemata)
- Problem der Linksrekursion
  rekursiver Parser und Übersetzungsschemata

Motivation für kurzen Einschub

- Parser-Funktionalität (z.B. Prädiktiver Parser)
  setzt gewisse Grammatikeigenschaften voraus.

- Ausgangsgrammatiken verletzten häufig einige dieser benötigten Eigenschaften.

- Es gibt jedoch Transformationen von Grammatiken (Restrukturierungen), die diese Eigenschaften sichern, wobei die Resultate die gleichen Sprachen erzeugen (!)

- Teilweise sind aber diese Transformationen nur für die Analyse von Eingabeworten der Sprache günstig,
  aber ungünstig für Codeerzeugungen aus so entstehenden Syntaxbäumen.

- Um die neu entstehenden (aber durchaus beherrschbaren!) Probleme diskutieren zu können,
  benötigen wir zumindest eine erste Vorstellung von der Codegenerierung.

Übersetzungsformalismen

(1) **Syntaxgesteuerte Definition** (später)

- Formalismus zur Beschreibung von Übersetzungen programmiersprachlicher Konstrukte
- Ausgangspunkt: Komponenten eines syntaktischen Konstrukts, angereichert mit Attributen (Typ, Speicherplatz, …)

(2) **Übersetzungsschemata** (jetzt)

- englische Bezeichnung: syntax directed translation scheme (SDT)
- prozeduraler Formalismus
- SDT ist eine kfG mit eingestreuten semantischen Aktionen (Programmfragmente) in den rechten Seiten der Produktionen

  eine Aktion wird dabei in {…} eingeschlossen

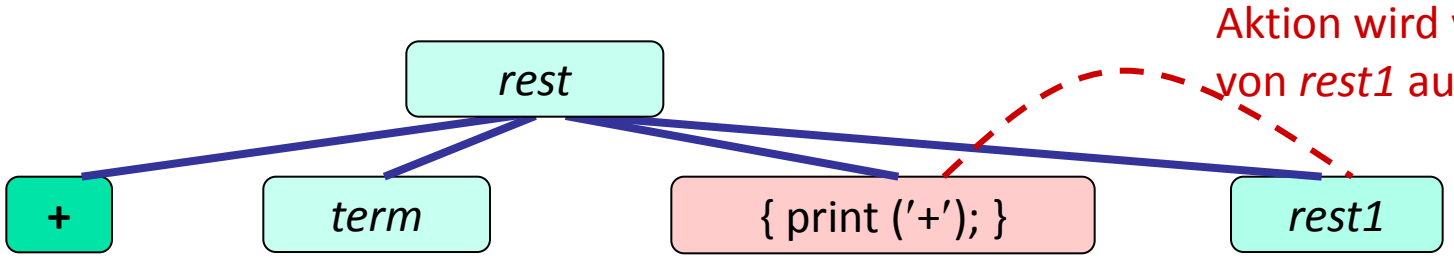
Beispiel: Übersetzungsschemata

- … **erzeugt** (durch Ausführung der Aktionen) für die jeweils generierte Satzform x eines beliebigen Eingabewortes **eine Ausgabe** (z.B. für Ausgabe-Datei)

  rest ⟶ **+** *term* { print ('+'); } *rest1*

- Reihenfolge der Aktionen entspricht einer festgelegten semantischen Interpretation:

  „Tiefe-Zuerst"-Durchlauf des Parse-Baumes für x



- Knoten mit semantischen Aktionen **hat nie Nachfolger**
- die Auswertung erfolgt bereits beim Erstanlauf

Beispiel: Übersetzungsschemata-Implementation

Ausdrücke in Infix-Notation

(9 – 5) + 2 9 – (5 + 2)

Compilation= Transformation

Ausdrücke in Postfix-Notation *Klammern werden überflüssig*

9 5 – 2 + 9 5 2 + -

kf Grammatik

1.	*s*	→ *expr*
2.	*expr*	→ *expr* + *term*
3.	*expr*	→ *expr* - *term*
4.	*expr*	→ *term*
5.	*term*	→ **0**
6.	*term*	→ **1**
	…	
14.	*term*	→ **9**

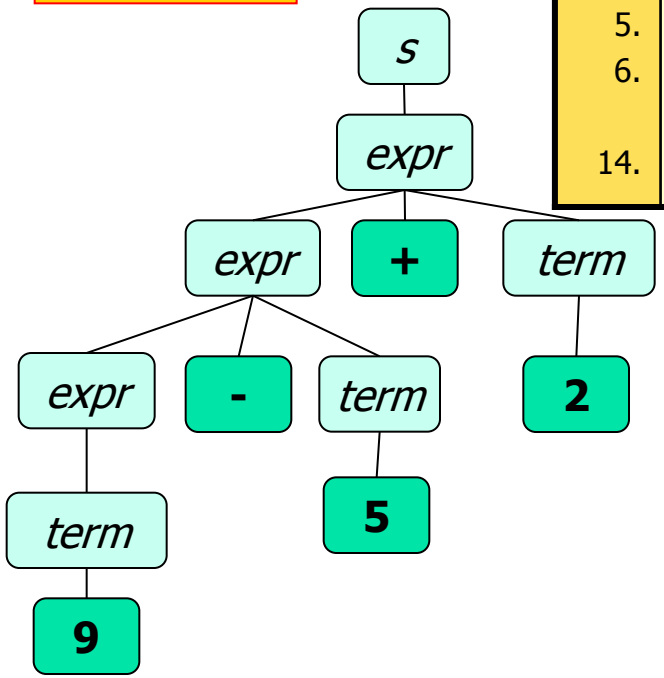
SDT

1.	*s*	→ *expr*
2.	*expr*	→ *expr* + *term* {print('+')}
3.	*expr*	→ *expr* - *term* {print('-')}
4.	*expr*	→ *term*
5.	*term*	→ **0** {print('0')}
6.	*term*	→ **1** {print('1')}
	…	
14.	*term*	→ **9** {print('9')}

Beispiel: Übersetzungsschemata-Ausführung

9 – 5 + 2

1.	*S*	→ *expr*
2.	*expr*	→ *expr* **+** *term* {print('+')}
3.	*expr*	→ *expr* **-** *term* {print('-')}
4.	*expr*	→ *term*
5.	*term*	→ **0** {print('0')}
6.	*term*	→ **1** {print('1')}
	...	
14.	*term*	→ **9** {print('9')}



Ausgabe erfolgt schrittweise

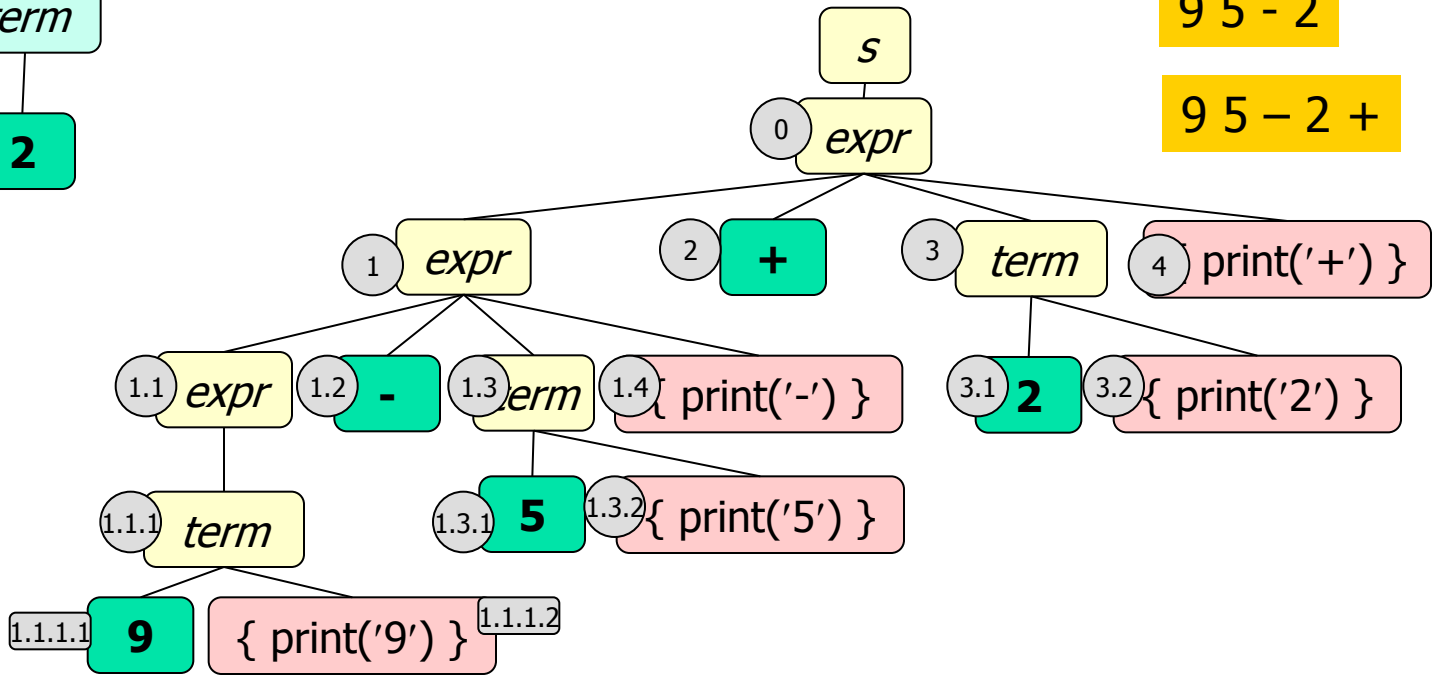
9

9 5

9 5 -

9 5 - 2

9 5 – 2 +



4.2 Restrukturierung von Grammatiken

- Ursache von Mehrdeutigkeiten (kontextfreie Mehrdeutigkeiten)
- Eliminierung von Mehrdeutigkeiten
  als Voraussetzung für die Konstruktion von LL- und LR-Parsern
- erste Betrachtung von Übersetzungstechniken
  (am Beispiel von Übersetzungsschemata)
- Problem der Linksrekursion
  rekursiver Parser und Übersetzungsschemata

Problem der Linksrekursion

Notwendigkeit der Auflösung von Linksrekursionen bei prädiktiven Top-Down-Parsern

Beispiel einer links-rekursiven Produktionsregel:

expr → *expr* **+** *term*

führt bei Durchführung linksseitiger Ableitungen (in Top-Down-Verfahren) zu Endlosschleifen

Grund für Endlosschleife:
weil sich durch rekursiven Aufruf der Prozedur für das Nichterminal *expr* allein **nicht** das LookAhead-Symbol ändert
(nur wenn sich ein Terminal-Symbol nach Anwendung der Regel ergäbe)

Problemlösung: Überführung in äquivalente Rechtsrekursionen (als Grammatikregeln)

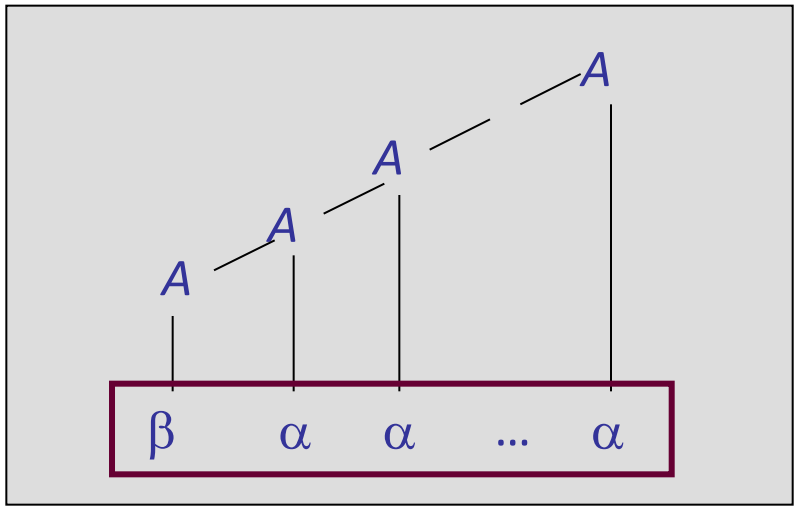
Eliminierung einer Linksrekursion

geg.: $A \rightarrow A\alpha \mid \beta$

Einführung *R* als neue Meta-Variable

α, β seien Folgen von Terminalsymbolen und Meta-Variablen, die **nicht** mit *A* beginnen

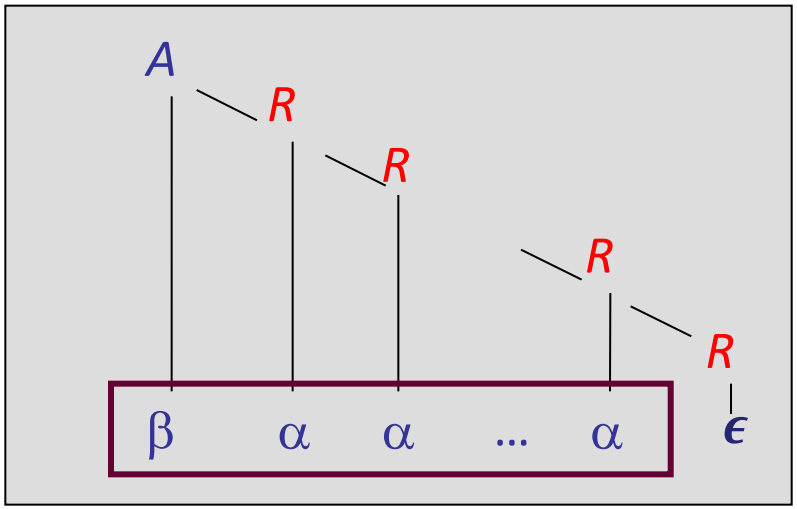
A ist links-rekursiv



$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$

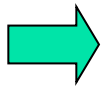
R ist rechts-rekursiv



Allgemeines Prinzip der Eliminierung

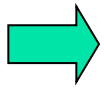
A = *expr*
α = **+** *term*
β = **-** *term*
γ = *term*

$A \rightarrow A\alpha \mid \beta$



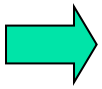
$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \epsilon$

$A \rightarrow A\alpha \mid A\beta \mid \gamma$



$A \rightarrow \gamma R$
$R \rightarrow \alpha R \mid \beta R \mid \epsilon$

1.	*S*	→ *expr*
2.	*expr*	→ *expr* **+** *term*
3.	*expr*	→ *expr* **-** *term*
4.	*expr*	→ *term*
5.	*term*	→ **0**
6.	*term*	→ **1**
	...	
14.	*term*	→ **9**



1.	*S*	→ *expr*
2.	*expr*	→ *term rest*
3.	*rest*	→ **+** *term rest*
4.	*rest*	→ **-** *term rest*
5.	*rest*	→ ϵ
6.	*term*	→ **0**
	term	→ **1**
	...	
15.	*term*	→ **9**

Eliminierung einer Linksrekursion in einem Übersetzungsschema

Technik lässt sich problemlos übertragen

$A \rightarrow Aa \mid b$ ⇒ $A \rightarrow bR$, $R \rightarrow aR \mid \epsilon$

$A \rightarrow Aa \mid Ab \mid g$ ⇒ $A \rightarrow gR$, $R \rightarrow aR \mid bR \mid \epsilon$

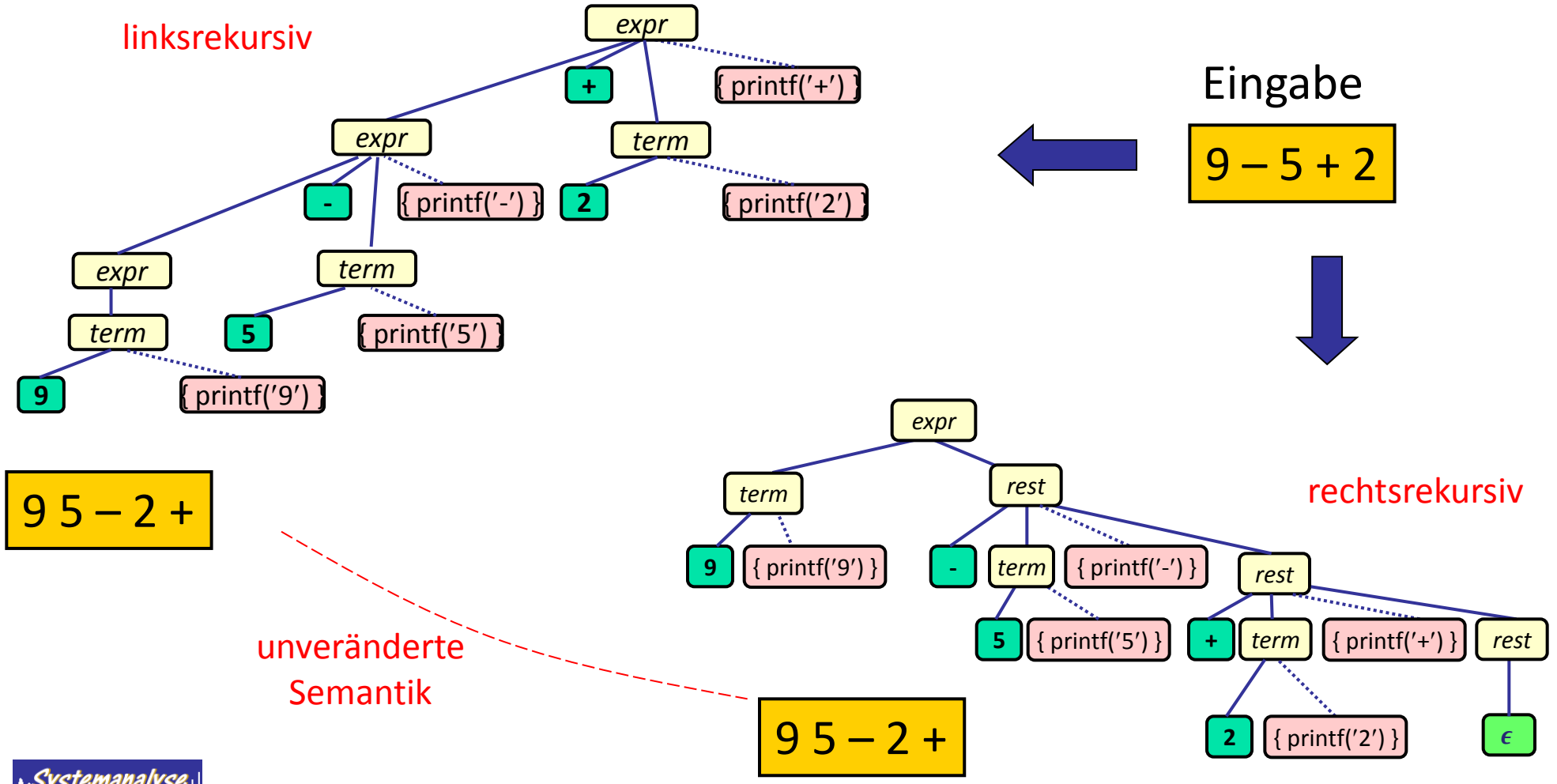
A = *expr*
a = **+** *term* {print('+')}
β = **-** *term* {print('-')}
g = *term*

1.	*S*	→	*expr*
2.	*expr*	→	*expr* **+** *term* {print('+')}
3.	*expr*	→	*expr* **-** *term* {print('-')}
4.	*expr*	→	*term*
5.	*term*	→	**0** {print('0')}
6.	*term*	→	**1** {print('1')}
	...		
14.	*term*	→	**9** {print('9')}

⇒

1.	*S*	→	*expr*
2.	*expr*	→	*term rest*
3.	*rest*	→	**+** *term* {print('+')} *rest*
4.	*rest*	→	**-** *term* {print('-')} *rest*
5.	*rest*	→	ϵ
6.	*term*	→	**0** {print('0')}
	term	→	**1** {print('1')}
	...		
15.	*term*	→	**9** {print('9')}

Eliminierungsprinzip lässt Sprache unverändert



Grammatik-Transformation: Linksfaktorisierung

Existieren für ein Nichtterminal einer Grammatik zwei oder mehr Produktionen mit einem **gemeinsamen Präfix** (ε ausgenommen) auf der **rechten Seite**,
muss ein prädiktiver Parser mit einem **Lookahead** von **einem Token** fehlschlagen:

kann **nicht** entscheiden,
welche der Alternativen gewählt werden muss.

1.	A	$\rightarrow \alpha\, \beta_1$
2.		$\mid \alpha\, \beta_2$

1.	*stmt*	→ **if** *expr* **then** *stmt* **else** *stmt*
2.		**|** **if** *expr* **then** *stmt*

nach Lesen von **if**
kann noch nicht die richtige Regel bestimmt werden

Durch **Linksfaktorisierung** wird die Entscheidung,
welche Alternative zu wählen ist, aufgeschoben, bis das gemeinsame Präfix erkannt wurde.

1.	A	$\rightarrow \alpha\, A'$
2.	A'	$\rightarrow \beta_1 \mid \beta_2$

Entscheidung zurückstellen
und lassen A zu $\alpha A'$ expandieren

1.	*stmt*	→ **if** *expr* **then** *stmt* *else-stmt*
2.	*else-stmt*	→ **else** *stmt* **|** ϵ

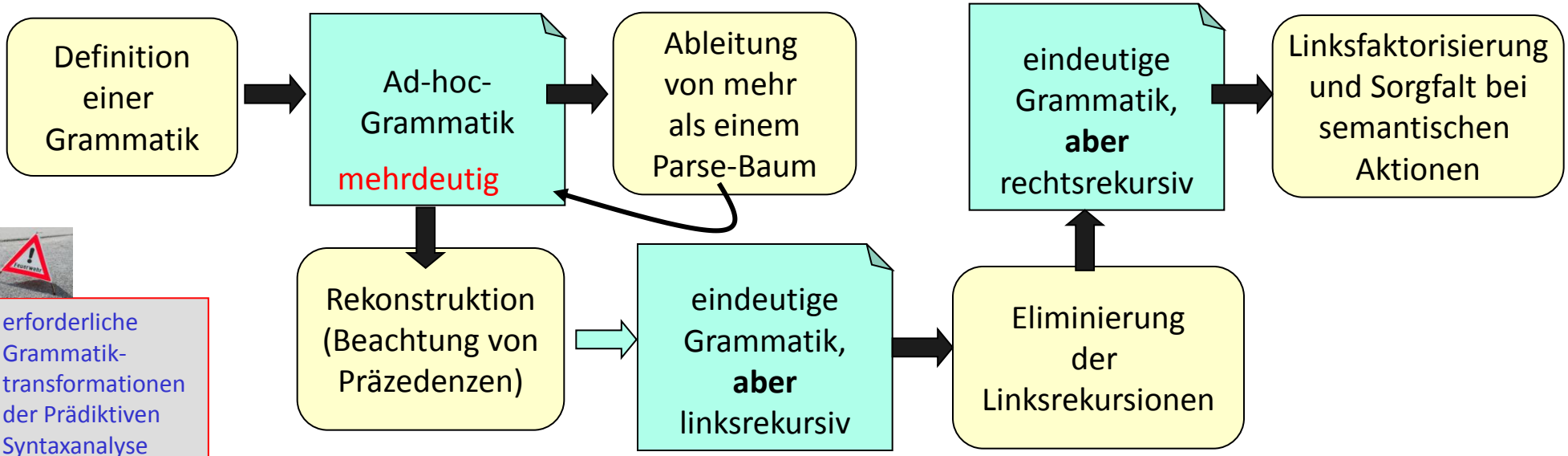
Beispiel-Grammatik bleibt aber noch mehrdeutig !

Grammatikanforderungen: Zusammenfassung

- **Ziel**: Compiler für einfache Ausdrücke
  - als Prädiktiver Analysator (rekursiver Abstieg) mit Übersetzungsschemata

für Analyse gut geeignet,
aber nicht für Compilierung
links-assoziativer Operatoren
(+, -, *, /)

- **Schritte**:



erforderliche Grammatik-transformationen der Prädiktiven Syntaxanalyse

prädiktive Analysatoren
laufen in Endlos-Schleifen

Position

- Teil I
  Die Programmie
- Teil II
  Methodische Gru
- Teil III
  Entwicklung ein

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Lexikalische 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
- 4.2
  Restrukturierung von Grammatiken
- 4.3
  LL-Parser
- 4.4
  . . .

4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- First-Mengen-Bildung
- Follow-Mengen-Bildung
- LL-Grammatik-Eigenschaften

Ableitungen (linksseitig, top-down)

1.	*s*	→ *stmt*
2.	*stmt*	→ *expr* **semi**
3.	*expr*	→ *factor* **plus** *expr*
4.		| *factor*
5.	*factor*	→ **num**

Linksseitige Ableitung

das **am weitesten links stehende** Metasymbol einer RS wird abgeleitet

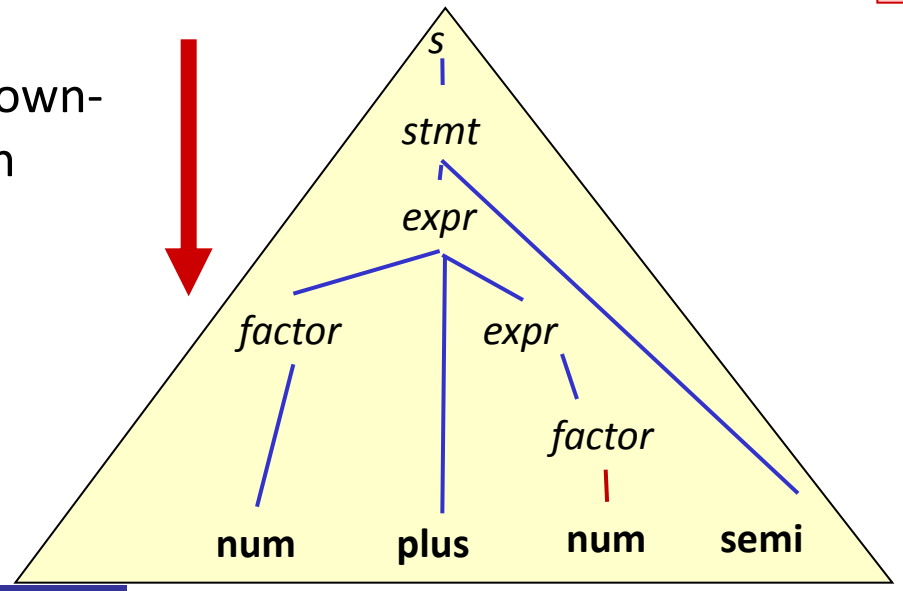
Eingabe

1 + 2;

Token-Folge

num plus num semi

Top-down-Parsen



s ⇒ *stmt*	[nach 1. Regel]
⇒ *expr* **semi**	[nach 2. Regel]
⇒ *factor* **plus** *expr* **semi**	[nach 3. Regel]
⇒ **num plus** *expr* **semi**	[nach 5. Regel]
⇒ **num plus** *factor* **semi**	[nach 4. Regel]
⇒ **num plus num semi**	[nach 5. Regel]

Prädiktive Syntaxanalyse

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

Fazit:

- … stützt sich auf die Kenntnis,
  welche Anfangssymbole von der RS
  einer Produktion in der jeweiligen Situation tatsächlich angewendet werden können

  bisherige Beispiel-Grammatiken boten keinerlei Analyseprobleme
  bei rekursiven Abstieg

- Konstruktion von allgemeinen Bottom-Up und Top-Down-Parsern für prädiktive Analysen
  benötigt Hilfsfunktionen, FIRST, FOLLOW
  mit denen ausgehend vom nächsten Eingabesymbol
  die anzuwendende Regel sicher gewählt werden kann.

4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- FIRST-Mengen-Bildung
- FOLLOW-Mengen-Bildung
- LL-Grammatik-Eigenschaften

FIRST- Mengen

... sind hilfreich bei der Konstruktion **prädiktiver Parser**

Motivation

- für jeweils zwei alternative Produktionen

$A \rightarrow \alpha \mid \beta$

$\alpha, \beta \in (\Sigma \cup V)^*$ setzen sich aus Terminalsymbolen und Metasymbolen zusammen

brauchen wir Unterscheidungsmerkmale,
um die richtige Regel beim Expandieren des Syntaxbaumes zu bestimmen

Bedingung für Grammatik für prädiktive Syntaxanalyse

- sind $A \rightarrow \alpha$ und $A \rightarrow \beta$ Regeln, dann

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$

d.h. das **LookAhead-Symbol** gibt den alleinigen Ausschlag, welche Regel anzuwenden ist

- bilden alle möglichen Ableitungen von α (bzw. β)
- schauen uns dabei nur die **jeweils ersten** Terminalsymbole an (können aber erst über mehrere Ableitungsstufen sichtbar werden)
- fassen diese zur Menge FIRST zusammen

Definition einer FIRST-Menge

Def.: FIRST-Menge (informale Definition)

sei $\alpha \in V$ die rechte Seite einer Produktion

dann ist FIRST(α) die Menge aller Terminalsymbole,
die bei mindestens einem aus α hergeleiteten Wörter als **Anfangssymbol** auftreten
(bei Betrachtung aller möglichen Ableitungen)

ϵ gehört auch zur FIRST-Menge, falls

- $\alpha = \epsilon$ oder
- $\alpha \Rightarrow^* \epsilon$

Konstruktion von FIRST-Mengen

sei $\alpha \in V$ die rechte Seite einer Produktion

folgende Regeln

werden solange zur Berechnung von FIRST(α) angewendet,
bis zu keiner FIRST-Menge mehr ein neues **Terminalsymbol** oder ϵ hinzukommt:

1. ist α **Terminalsymbol**, dann ist FIRST(α)= { α };

2. gibt es eine Produktion $\alpha \rightarrow Y_1 Y_2 ... Y_k$, dann ist
   - zunächst FIRST(Y_1) \ {ϵ} der Menge FIRST(α) zuzuführen;
   - sollte ϵ in FIRST(Y_1) enthalten sein, so ist auch FIRST(Y_2) \ {ϵ} der Menge FIRST(α) hinzuzufügen;
   - Fortsetzung bis zu einem $i \leq n$, wo Y_i **nicht** ϵ -ableitbar ist;
   - sollte ϵ jedoch in **allen** FIRST(Y_i) enthalten sein, ist auch ϵ der Menge FIRST(α) hinzuzufügen;

3. gibt es eine Produktion $\alpha \rightarrow \epsilon$, dann ist ϵ der Menge FIRST(α) hinzuzufügen.

Beispiel-1: FIRST-Mengen

G-1
S → A B **c**
A → **a**
A → ε
B → **b**
B → ε

$\alpha \in V$	FIRST(α)	Y_1	Y_2	Y_3	FIRST(Y_1)	FIRST(Y_2)	FIRST(Y_3)	epsilon	FIRST(α)
S	ø	A	B	**c**	{ **a**, ε}	{ **b**, ε}	{ **c** }	NEIN	{ **a**, **b**, **c** }
A	ø	**a**			{ **a**, ε}			JA	{ **a**, ε}
A		ε							
B	ø	**b**			{ **b**, ε }			JA	{ **b**, ε}
B		ε							

Beispiel-2: FIRST-Mengen

G-2:
s → **if** *E* **then** *S* **else** *S*
s → **begin** *S L*
s → **print** *E*
s → **end**
L → **;** *S L*
E → **num = num**

$\alpha \rightarrow Y_1Y_2...Y_k$				FIRST(Y_1)	FIRST(Y_2)	ε	FIRST(α)
$\alpha \in V$	Y_1	Y_2	Y_3				
s	**if**	-		{ **if** }		NEIN	{ **if** , **begin**, **print**, **end** }
s	**begin**	-		{ **begin** }			
s	**print**	-		{ **print** }			
s	**end**	-		{ **end** }			
L	**;**	-		{ **;** }		NEIN	{ **;** }
E	**num**	-		{ **num** }		NEIN	{ **num** }

trivial (keine Eintragungen notwendig),
weil **jeweilige Y_1** ausschließlich Terminalsymbole
sind und sich damit eine Untersuchung folgender Y_i (i>1) erübrigt

Beispiel-3: FIRST-Mengen

G-3
S → E **$**
E → E **+** T
E → E **-** T
E → T
T → T ***** F
T → T **/** F
T → F
F → **id**
F → **num**
F → **(E)**

$\alpha \in V$	Y_1	Y_2	FIRST(Y_1)	FIRST^2(Y_1)	FIRST(Y_2)	ε	FIRST(α)
S	E	**$**		{ **id**, **num**, **(** }		nein	{ **id**, **num**, **(** }
E	E	**+**		{ **id**, **num**, **(** }		nein	
E	T						{ **id**, **num**, **(** }
T	T	*****		{ **id**, **num**, **(** }		nein	{ **id**, **num**, **(** }
T	T	**/**					
T	F						
F	**id**		{ **id** }	{ **id**, **num**, **(** }		nein	{ **id**, **num**, **(** }
F	**num**		{ **num** }				
F	**(**		{ **(** }				

FIRST(Y_2) nicht zu betrachten, da sich kein Y_1 nach ϵ ableiten lässt

FIRST(T) = FIRST(F)
FIRST(E) = FIRST(T)