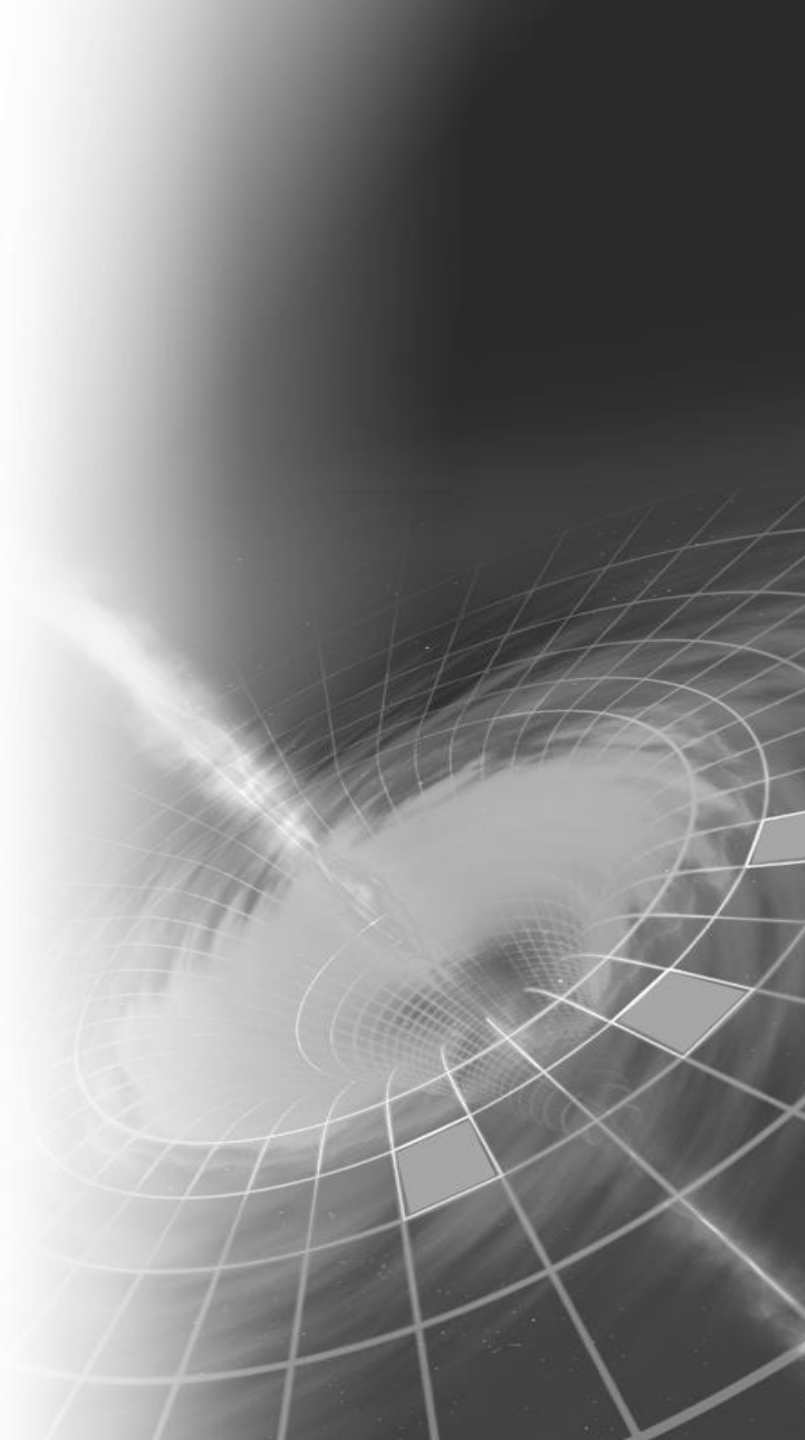
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
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
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

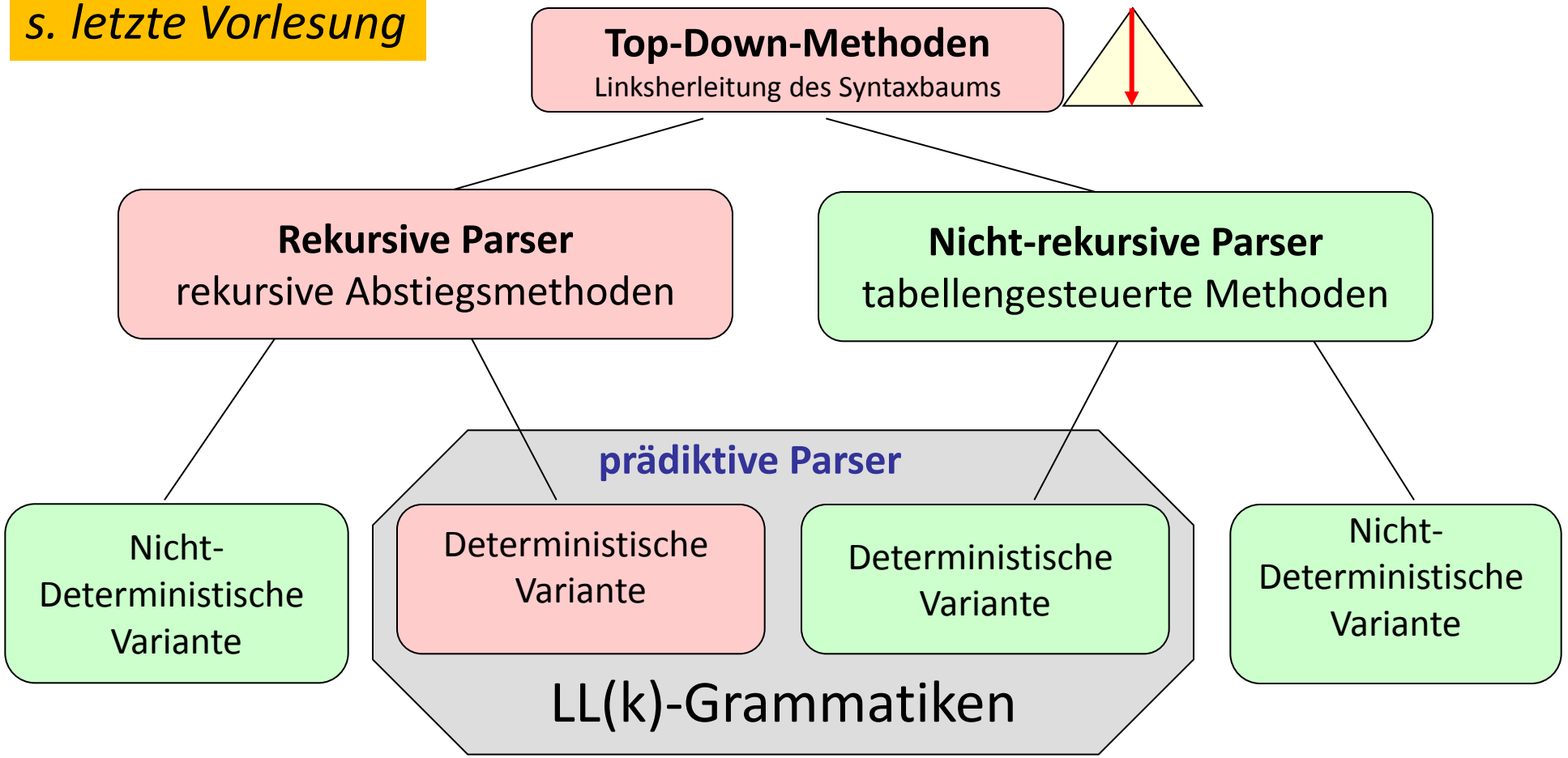
- 4.1
  Einführung in die Syntaxanalyse
- 4.2
  Restrukturierung von Grammatiken
- 4.3
  LL-Parser
- 4.4
  Beispiel: Ein-Pass-Compiler (Parser, Übersetzer)
- 4.5
  Tabellengesteuerter LL-Parser

4.5 Tabellengesteuerte (deterministische) Top-Down-Verfahren

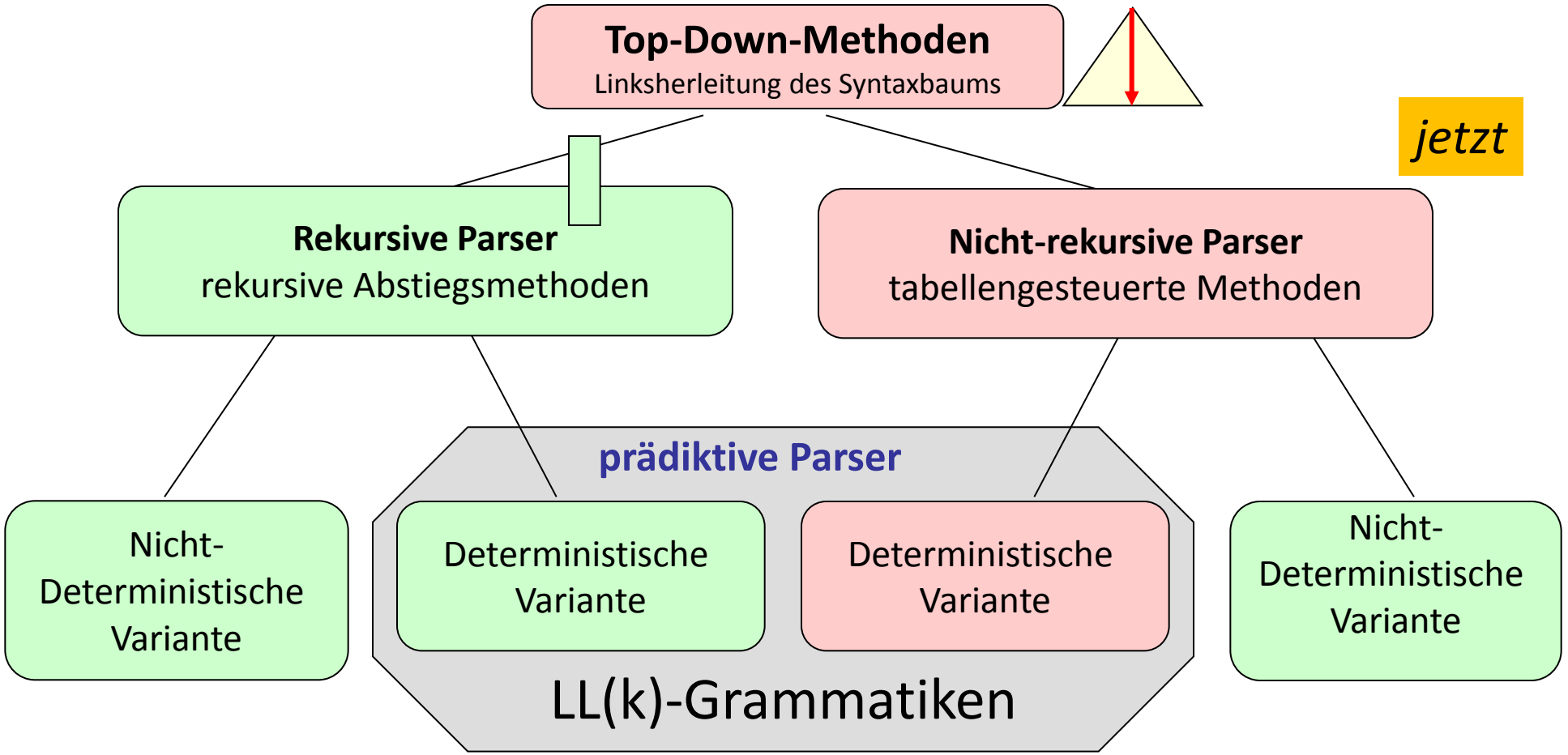
- Allg. Arbeitsweise eines tabellengesteuerten Top-Down-Parsers
- Konstruktion von Parse-Tabellen für nicht-rekursive prädiktive Parser
- Bestimmung synchronisierender Terminalsymbole zur Fehlerstabilisierung
- Parser-Generatoren für LL(1)-Sprachen

Klassen von Top-Down-Methoden

s. letzte Vorlesung



Klassen von Top-Down-Methoden



Vergleich: *rekursiver Abstieg – tabellengesteuertes Verfahren*

es ist möglich,
prädiktive Parser **nicht-rekursiv** zu implementieren:

- implizite Stapelverwaltung ←→ explizite Stapelverwaltung

Prozedur-Stack (durch verschachtelten Aufruf)	zusätzlicher Keller-speicher

rekursiver Abstieg

- ist leichter zu implementieren und
- die Fehlerbehandlung (Information und Stabilisierung) ist präziser möglich

tabellengesteuertes Verfahren

- effizienter im Platz- und Zeitverbrauch
- Tabelle zur Programmsteuerung lässt sich von Hand nur schwer entwerfen (fehleranfällig)
- jedoch gibt es Werkzeuge zur Tabellengenerierung

Vorzug der Tabellensteuerung
generischer Parser für beliebige LL(1)-Sprachen

es gibt auch Tricks zur Behandlung von einzelnen Nicht-LL(1)-Konstrukten !

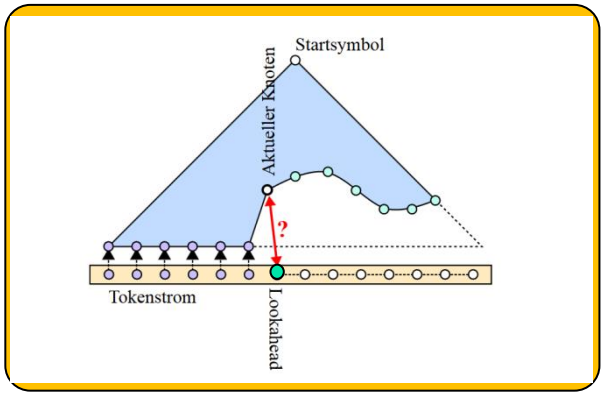
Tabellengesteuerter Parser

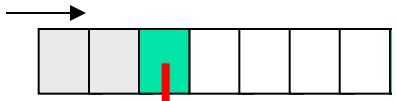
austauschbare Parsertabelle
2-dimensionales Feld

Grammatikregel oder Fehlereintrag

Variable	Terminalsymbol		
	id	...	+
S			
E			

Grammatik, FIRST-, FOLLOW-Mengen

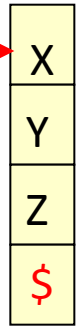




Eingabe
Folge von Token

Programm zum prädiktiven Parsen
- Kellerautomat -

Ausgabe
Syntaxbaum



Kellerspeicher
Folge von Terminalsymbolen oder/und Variablen

Anfangsbelegung
- $
- darüber Startsymbol S

Prädiktive Parsertabellen

(bekanntes) Grundproblem prädiktiver Parser

- für einige Nichtterminale A war beim Aufbau des Syntaxbaumes zu entscheiden,
  welche Regel (von mehreren Alternativen) aufgrund
  des nächsten Eingabe-Token t zur Anwendung gebracht werden soll

- **wenn** die richtige Produktion für jedes erlaubte Paar (Variable, Eingabesymbol)
  gefunden werden kann,
  lässt sich ein prädiktiver Parser mit Hilfe einer Tabelle **M**

  Regel= **M** [A, t]

  bauen

  Beispiel: Pascal (≈ 64 Terminalsymbole, 135 Variablen)

Beispiel: Prädiktive Parsertabelle

Grammatik:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Variable	Eingabesymbol					
	id	+	*	(	)	$
E						
E´						
T						
T´						
F						

Konstruktion der Einträge über FIRST und FOLLOW-Mengen (später!)

Beispiel: Prädiktive Parsertabelle

Grammatik:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Variable	Eingabesymbol					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E´		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T´		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Bem.:
Leerfelder symbolisieren **Fehlersituationen**

Programm zur Steuerung des Parsers

holt sich zyklisch Werte:

oberstes Kellerelement X ($X \in V \cup \Sigma$) und aktuelles Eingabesymbol t ($t \in \Sigma$)

mit drei **Fortsetzungsmöglichkeiten**

(1) $X = t$ und $t = \$$

der Parser **stoppt** und meldet erfolgreichen Abschluss der Syntaxanalyse

(2) $X = t$ und $t \neq \$$

Entfernung des obersten Kellerelementes

Umsetzung des Zeigers auf das nächste Eingabesymbol (≙ **match**)

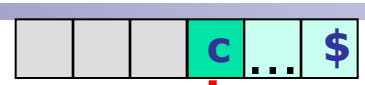
(3) $X = A$ ($A \in V$)

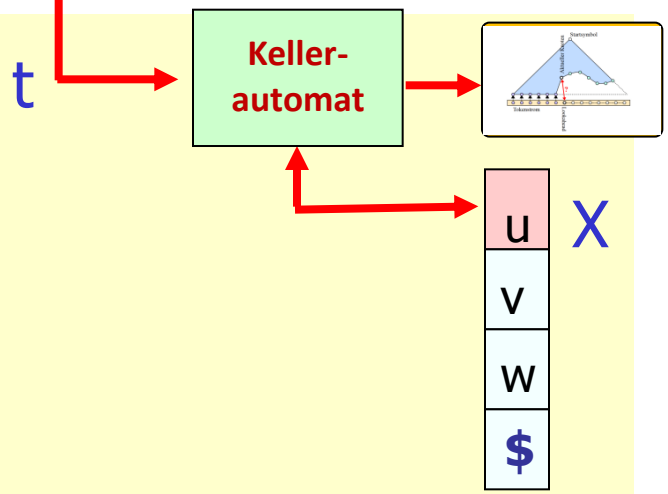
Parser wertet Eintrag **M** [A, t] aus:

- **falls Regeleintrag**: (z.B.) A → UVW
  oberstes Kellerelement A wird **ersetzt** durch *RS* UVW (U liegt oben) und
  die angewendete Produktion wird ausgegeben oder der Syntaxbaum konstruiert
- **falls error-Eintrag**: Aufruf einer Fehlerbehandlungsroutine

Programm in Pseudo-Code

Variable	Eingabesymbol	
	a	b
S		
T		






```
repeat
    sei X oberstes Kellerelement und
      t das aktuelle Eingabesymbol (lookAhead);
    if X ist Terminalsymbol oder $ then
        if X = t then
                entferne X vom Keller und aktualisiere lookAhead;
        else error()
    else /* X ist Nichtterminal */
        if M[X, t]= X →Y1Y2…Yk  then begin
                entferne X vom Keller;
                lege YkYk-1…Y1 in den Keller, so dass Y1 oberstes Element ist;
                gib die Produktion X →Y1Y2…Yk aus;
        end
        else error();
until X = $ /* Stapel leer */
```

Parser-Programms: Zustand und Verhalten

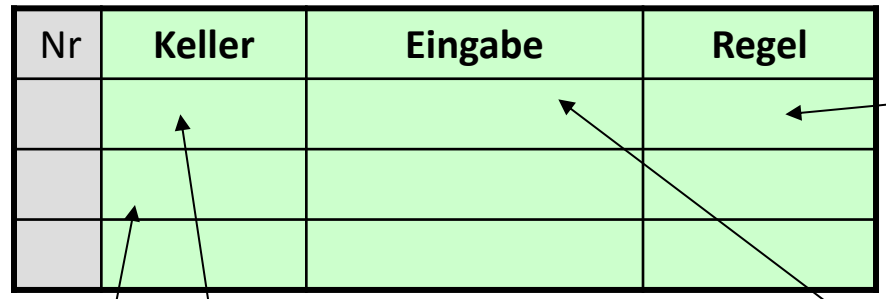
- **Konfiguration**= Programmzustand als

  Tripel von Informationen

  (1) aktueller Stapelinhalt,
  (2) noch zu verarbeitende Eingabe
  (3) Ausgabe nach Beendigung des aktuellen Arbeitsschrittes

- Folge von aufeinander folgenden Konfigurationen beschreibt das **Programmverhalten** (*Trace*) bei der Verarbeitung eines gegebenen Eingabewortes

Konfigurationsfolge: *Initiale Belegung, erster Schritt*



1b gefundene Regel in der Tabelle **M**

0 $ (gleiches Zeichen wie Eingabeende) zur Kennzeichnung des unteren Endes des Kellers

1a aktuelles Eingabesymbol (lookAhead)

1c i. Abh. der Regel:
Ersetzung des obersten Kellerelementes bzw. Abgleich mit aktuellem Eingabesymbol

1	$E	id + id * id $	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			

Beispiel-1: Konfigurationsfolge

Startsymbol

Grammatik:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Aktionen eines tabellengesteuerten prädiktiven Parsers

bei Eingabe von

id	+	id	*	id	$

NT	Eingabesymbole					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Beispiel-1: Konfigurationsfolge

1	$E	id + id * id $	E → TE'
2	$E'T	id + id * id $	T → FT'
3	$E'T'F	id + id * id $	F → id
4	$E'T'id	id + id * id $	Reduktion
5	$E'T'	+ id * id $	T' → ε
6	$E'	+ id * id $	E' → +TE'
7	$E'T+	+ id * id $	Reduktion
8	$E'T	id * id $	T → FT'
9	$E'T'F	id * id $	F → id
10	$E'T'id	id * id $	Reduktion
11	$E'T'	* id $	T' → *FT'
12	$E'T'F*	* id $	Reduktion
13	$E'T'F	id $	F → id
14	$E'T'id	id $	Reduktion
15	$E'T'	$	T' → ε
16	$E'	$	E' → ε
17	$	$	Reduktion

Gesamtdarstellung

bei Eingabe von

	+	id	*	id	$

NT	Eingabesymbole					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

⇒ erfolgreiche Ableitung

4.5 Tabellengesteuerte (deterministische) Top-Down-Verfahren

- Allg. Arbeitsweise eines tabellengesteuerten Top-Down-Parsers
- Konstruktion von Parser-Tabellen für nicht-rekursive prädiktive Parser
- Bestimmung synchronisierender Terminalsymbole zur Fehlerstabilisierung
- Parser-Generatoren für LL(1)-Sprachen

Konstruktion prädiktiver Parser-Tabellen (1)

Idee:

- **Ann.**: A → α sei Produktion mit **a** $\in$ FIRST(α),

  dann expandiert der Parser A zu α, **wenn** **a** aktuelles Eingabesymbol ist.

- Zu Komplikationen kommt es nur, wenn $\alpha = \varepsilon$ oder $\alpha \Rightarrow^* \varepsilon$ ist.

  A muss in diesem Fall auch hier immer dann zu α expandiert werden,
  wenn
  - das aktuelle Symbol **a** in FOLLOW(A) liegt
    oder
  - in der Eingabe die Endemarkierung **$** erreicht wurde und
    $ in FOLLOW(A) enthalten ist.

Konstruktion prädiktiver Parser-Tabellen (2)

Algorithmus: Parser-Tabellenkonstruktion

Eingabe: Grammatik G

Ausgabe: Parser-Tabelle M

Methode:

1. Führe für jede Produktion $A \rightarrow \alpha$ die Schritte a) und b) durch.
   a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion $A \rightarrow \alpha$ in M[A, **a**] ein.
   b) Sollte ε in FIRST(α) enthalten ein,
   trage $A \rightarrow \alpha$ für jedes Terminal **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

   Sollte ε in FIRST(α) und **$** in FOLLOW(A) enthalten sein,
   so trage $A \rightarrow \alpha$ in M[A, **$**] ein.
2. Trage in jede undefinierte Position von M error ein (üblich: Leerstelle).

Bsp-1: Parser-Tabellenkonstruktion (1)

zunächst:
Bestimmung der FIRST- und FOLLOW-Mengen

$\alpha \in V$	FIRST(α)	FOLLOW(α)
E	id, (	), $
E'	+, ε	), $
T	id, (	), +, $
T'	*, ε	+,), $
F	id, (	+, *,), $

Erweiterung

Grammatik:

E → TE'$	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

Grammatik ist vom LL(1)-Typ

Bsp-1: Parser-Tabellenkonstruktion (2)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

Anwendung des Algorithmus auf eine LL(1)-Grammatik
(d.h. auf jede ihrer Regeln)

1a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion A → α in M[A, **a**] ein.

1b) Wenn ε in FIRST(α) enthalten ist, trage A → α für jedes Terminalsymbol **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

Ist ε in FIRST(α) und **$** in FOLLOW(A) enthalten, so trage A → α in M[A, **$**] ein.

Variable A	**Eingabesymbol**					
	id	**+**	*****	**(**	**)**	**$**
E						
E'						
T						
T'						
F						

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id**, **(**	), $
E'	**+**, ε	**)**, **$**
T	**id**, **(**	), +, $
T'	*, ε	**+**, **)**, **$**
F	**id**, **(**	+, *,), $

Bsp-1: Parser-Tabellenkonstruktion (3)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

1a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion A → α in M[A, **a**] ein.

1b) Wenn ε in FIRST(α) enthalten ist, trage A → α für jedes Terminalsymbol **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

Ist ε in FIRST(α) und **$** in FOLLOW(A) enthalten, so trage A → α in M[A, **$**] ein.

FIRST(TE')= FIRST(T)

Variable A	**Eingabesymbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'						
T						
T'						
F						

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id**, **(**	), $
E'	**+**, ε	**)**, **$**
T	**id**, **(**	), +, $
T'	*, ε	**+**, **)**, **$**
F	**id**, **(**	+, *,), $

Bsp-1: Parser-Tabellenkonstruktion (4)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

1a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion A → α in M[A, **a**] ein.

1b) Wenn ε in FIRST(α) enthalten ist, trage A → α für jedes Terminalsymbol **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

Ist ε in FIRST(α) und **$** in FOLLOW(A) enthalten, so trage A → α in M[A, **$**] ein.

nicht der Fall

Variable A	**Eingabesymbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'						
T						
T'						
F						

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id**, **(**	), $
E'	**+**, ε	**)**, **$**
T	**id**, **(**	), +, $
T'	*, ε	**+**, **)**, **$**
F	**id**, **(**	+, *,), $

Bsp-1: Parser-Tabellenkonstruktion (5)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

1a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion A → α in M[A, **a**] ein.

1b) Wenn ε in FIRST(α) enthalten ist, trage A → α für jedes Terminalsymbol **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

Ist ε in FIRST(α) und **$** in FOLLOW(A) enthalten, so trage A → α in M[A, **$**] ein.

FIRST(+TE')={+}

Variable A	Eingabesymbol					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'				
T						
T'						
F						

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id, (**	), $
E'	**+, ε**	**), $**
T	**id, (**	), +, $
T'	***, ε**	**+,), $**
F	**id, (**	+, *,), $

Bsp-1: Parser-Tabellenkonstruktion (6)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

1a) Trage für jedes Terminalsymbol **a** aus FIRST(α) die Produktion A → α in M[A, **a**] ein.

1b) Wenn ε in FIRST(α) enthalten ist, trage A → α für jedes Terminalsymbol **b** aus FOLLOW(A) an der Stelle M[A, **b**] ein.

Ist ε in FIRST(α) und **$** in FOLLOW(A) enthalten, so trage A → α in M[A, **$**] ein.

FOLLOW(E') = { **)**, **$** }

Variable A	**Eingabesymbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T						
T'						
F						

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id**, **(**	), $
E'	**+**, ε	**)**, **$**
T	**id**, **(**	), +, $
T'	*, ε	**+**, **)**, **$**
F	**id**, **(**	+, *,), $

Bsp-1: Parser-Tabellenkonstruktion (7)

Grammatik:

E → TE'	[1]
E' → +TE'	[2]
| ε	[3]
T → FT'	[4]
T' → *FT'	[5]
| ε	[6]
F → (E)	[7]
| id	[8]

Nach Anwendung des Algorithmus auf alle Regeln
der Eingangsgrammatik
ergibt sich die komplette Tabelle mit Leerstellen (Fehler)

Variable	Eingabesymbole					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → **id**			F → **(**E**)**		

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id**, **(**	), $
E'	**+**, ε	**)**, **$**
T	**id**, **(**	), +, $
T'	*, ε	**+**, **)**, **$**
F	**id**, **(**	+, *,), $

Parser-Tabelle mit Mehrfacheinträgen

- mit dem Algorithmus lässt sich prinzipiell **für jede kfG** eine Parser-Tabelle konstruieren

- bei manchen Grammatiken werden jedoch **mehrere** Einträge für eine Tabellenposition zu erzeugen sein

- eine Grammatik G,
  deren Parser-Tabelle **Mehrfacheinträge** besitzt, ist **keine** LL(1)-Grammatik.

  z.B. wenn G
  - mehrdeutig oder
  - linksrekursiv

  ist

Bsp-2: Parser-Tabellenkonstruktion (1)

Anwendung des Algorithmus auf NICHT-LL(1)-Grammatik

Grammatik:

Z → **d**	[1]
Z → XYZ	[2]
Y → **ε**	[3]
Y → **c**	[4]
X → Y	[5]
X → **a**	[6]

α ∈ V	FIRST(α)	FOLLOW(α)
Z	**d**, **a**, **c**	$
Y	**c**, ε	**a**, **c**, **d**
X	**a**, **c**, ε	**a**, **c**, **d**

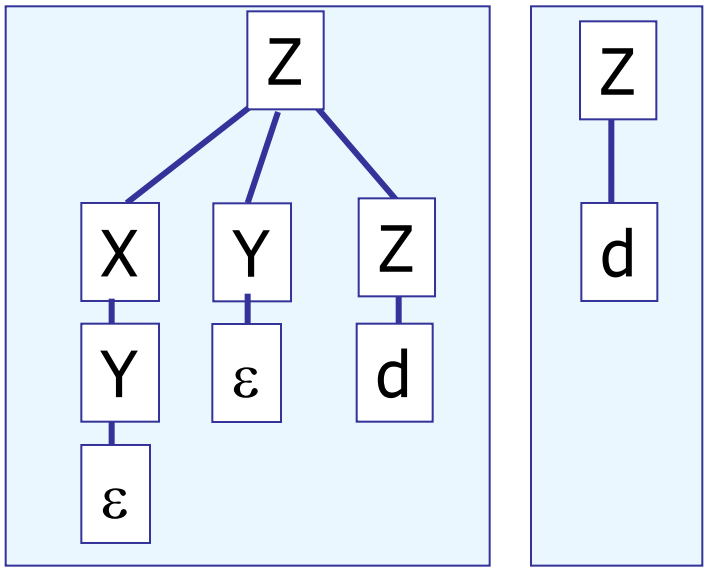
	Eingabesymbole		
	a	**c**	**d**
X	X → a X → Y	X → Y	X → Y
Y	Y → ε	Y → c Y → ε	Y → ε
Z	Z → XYZ	Z → XYZ	Z → XYZ Z → d

Mehrfacheinträge

Regel ist
ohne zusätzliche Information
nicht mehr eindeutig bestimmbar

Bsp-2: Parser-Tabellenkonstruktion (2)

Eingabe: **d**



zwei verschiedene Syntaxbäume

Grammatik ist mehrdeutig

Grammatik:

Z → **d**
Z → XYZ
Y → ε
Y → **c**
X → Y
X → **a**

	Eingabesymbole		
	a	**c**	**d**
X	X → a X → Y	X → Y	X → Y
Y	Y → ε	Y → c Y → ε	Y → ε
Z	Z → XYZ	Z → XYZ	Z → XYZ Z → d

Bsp-3: Parser-Tabellenkonstruktion (1)

Grammatik:
E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → **id**
F → **num**
F → **(E)**

Beispiel: NICHT-LL(1)-Grammatik

Grammatik ist linksrekursiv

α ∈ V	FIRST(α)	FOLLOW(α)
E	**id** , **num, (**	
T	**id** , **num, (**	
F	**id** , **num, (**	

	Eingabesymbol								
	id	**num**	**+**	**-**	*****	**/**	**(**	**)**	**$**
E	E → E+T E → E-T E → T	E → E+T E → E-T E → T					E → E+T E → E-T E → T		
T	T → T*F T → T/F T → F	T → T*F T → T/F T → F					T → T*F T → T/F T → F		
F	F → id	F → num					F → (E)		

Bsp-3: Parser-Tabellenkonstruktion (2)

Ausgangsgrammatik

Grammatik:
E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → **id**
F → **num**
F → **(** E **)**

links-rekursiv
➔ keine LL(1)-Grammatik

Eliminierung der Linksrekursion
neues Nichtterminal E'

E → T E'
E' → + T E'
E' → ε

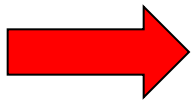
Grammatik:
E → T E'
E' → + T E'
E' → - T E'
E' → ε
T → F T'
T' → * F T'
T' → / F T'
T' → ε
F → **id**
F → **num**
F → **(** E **)**

rechts-rekursiv

wie Beispiel-1
Grammatik:
E → TE'
E' → **+**TE' | ε
T → FT'
T' → ***FT' | ε
F → **(**E**)** | **id**

Bsp-3: Parser-Tabellenkonstruktion (3)

Grammatik:
E → T E'
E' → + T E'
E' → - T E'
E' → ε
T → F T'
T' → * F T'
T' → / F T'
T' → ε
F → **id**
F → **num**
F → **(E)**



α ∈ V	FIRST(α)	FOLLOW(α)
E	**(, id, num**	), $
E'	**+, -,** ε	**), $**
T	**(, id, num**	), +, -, $
T'	***, /,** ε	**), +, -, $**
F	**(, id, num**	), *, /, +, -, $

	Eingabesymbol								
	id	**num**	**+**	**-**	*****	**/**	**(**	**)**	**$**
E	E → TE'	E → TE'					E → TE'		
E'			E' → **+**TE'	E' → **-**TE'				E' → ε	E' → ε
T	T → FT'	T → FT'					T → FT'		
T'			T' → ε	T' → ε	T' → *****FT'	T' → **/**FT'		T' → ε	T' → ε
F	F → **id**	F → **num**					F → **(**E**)**		

Einsatz der Parser-Tabellen

… auch als mögliche Basis für rekursive Abstiegsverfahren

Erkannte Ähnlichkeiten in der Tabelle helfen bei Zusammenfassung von Look-Ahead-Fällen

	Eingabesymbol								
	id	**num**	**+**	**-**	*****	**/**	**(**	**)**	**$**
E	E → TE'	E → TE'					E → TE'		
E'			E' → **+**TE'	E' → **-**TE'				E' → ε	E' → ε
T	T → FT'	T → FT'					T → FT'		
T'			T' → ε	T' → ε	T' → *FT'	T' → **/**FT'		T' → ε	T' → ε
F	F → **id**	F → **num**					F → **(**E**)**		

ähnlich

Bsp-4: Parser-Tabellenkonstruktion (1)

mehrdeutige Ausgangsgrammatik

Links-Faktorisierung

Grammatik:
S → **if** E **then** S
S → **if** E **then** S **else** S
S → **a**
E → **b**

Problem: für ein Nichtterminal A stehen mindestens zwei alternative Produktionen bereit:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Idee der Linksfaktorisierung

A-Produktionen so transformieren, dass die anzuwendende Produktion eindeutig wird.

Die Entscheidung einer Alternative wird so weit auf einen späteren Zeitpunkt verlagert, bis genug von der Eingabe gelesen wurde

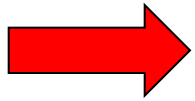
(bekannte) Methode:

A expandiert zunächst zu $\alpha A'$

man verarbeitet erst einmal das α und schaut dann weiter

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$



neue Grammatik:
S → **if** E **then** S S'
S → **a**
S' → **else** S
S' → ε
E → **b**

bleibt aber mehrdeutig
➔ keine LL(1)-Grammatik

Bsp-4: Parser-Tabellenkonstruktion (2)

$\alpha \in V$	FIRST(α)	FOLLOW(α)
S	**if**, **a**	**else**, **$**
S'	**else**, ε	**else**, **$**
E	**b**	

FOLLOW (S) = FIRST(S') \ {ε} $\cup$ FOLLOW(S')
FOLLOW (S') = FOLLOW(S)

Tabellenkonstruktion für geänderte (aber immer noch) mehrdeutige Grammatik

Variable	Eingabesymbole					
	a	**b**	**if**	**then**	**else**	**$**
S	S → **a**		S → **if** E **then** S S'			
S'					S' → ε S' → **else** S	S' → ε
E		E → **b**				

geänderte Grammatik:
S → **if** E **then** S S'
S → **a**
S' → **else** S
S' → ε
E → **b**

Auflösung der vorhandenen Mehrdeutigkeit
Anordnung einer generellen Entscheidung zugunsten der Regel
S' → **else** S
falls **else** in dieser Situation
das aktuelle Eingabesymbol ist

Die Linksfaktorisierung hat
sich demnach doch ausgezahlt,
Trick lässt sich aber nicht verallgemeinern !!!

4.5 Tabellengesteuerte (deterministische) Top-Down-Verfahren

- Allg. Arbeitsweise eines tabellengesteuerten Top-Down-Parsers
- Konstruktion von Parser-Tabellen für nicht-rekursive prädiktive Parser
- Bestimmung synchronisierender Terminalsymbole zur Fehlerstabilisierung
- Parser-Generatoren für LL(1)-Sprachen

Fehlerbehandlung beim Parsen: Allgemein

Kriterien der Fehlerbehandlung

- zuverlässig Feststellung mit verständliche Fehlermeldungen
- Fortsetzung der Analyse nach einem Fehler, um weitere Fehler entdecken zu können (Vermeidung von Scheinfehlern)
- Fehlerbehandlung darf die schnelle Übersetzung korrekter Programme nicht behindern.

Typische Syntaxfehler

- Zeichensetzungsfehler:

  zuviele oder zuwenige ';',

  ';' statt ',', usw.,
- Operatorenfehler:

  z.B. '=' statt ':=',
- Schlüsselwortfehler:

  fehlende oder falsch geschriebene Schlüsselwörter,
- Fehler in der Klammerstruktur
- ...

Vorgehen

(1) Möglichst schnelles Wiederaufsetzen (*recovery*) eines Parsers durch Überlesen möglichst weniger Symbole

(2) heuristische Verfahren für **lokale Korrekturen** durch den Parser, um Analyse fortsetzen zu können

Basis der Fehlererkennung eines Top-Down-Parsers

Keller eines tabellengesteuerten prädiktiven Parsers enthält

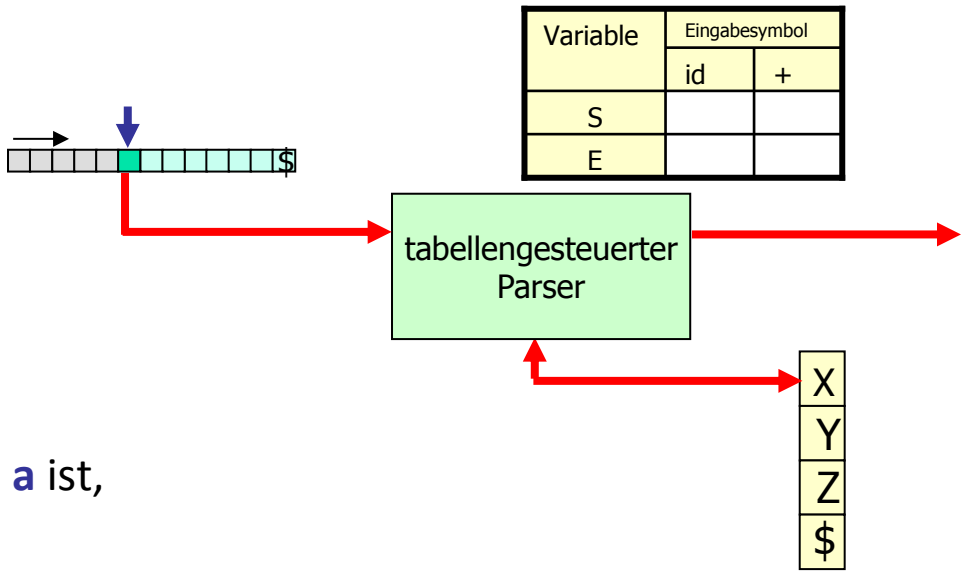
- Terminalsymbole und
- Variablen (als Strukturprognosen),

die noch mit der verbleibenden Eingabe zu vergleichen sind

Fehlersituationen:

1) wenn das oberste Kellerelement ein **Terminalsymbol** ist und dieses ungleich dem aktuellen Eingabesymbol ist

2) wenn das oberste Kellerelement eine **Variable** A und das Eingabesymbol **a** ist, der Eintrag von M[A, **a**] aber leer ist



Fehlerbehandlungsmethoden

leere Tabelleneinträge

Methoden

1. panische Behandlung

Basisstrategie:
Einsatz einer **Synchronisationsmenge** (besondere Token)

Vorgehen:

- ein Teil der Eingabe wird nach Auftreten eines Fehlers überlesen
- Ende solcher Teile werden durch *synchronisierende Symbole* angegeben
- Synchronisierende Symbole sind z.B. Symbole, die Anweisungen abschließen: ';' oder end

2. Konstrukt-orientierte Behandlung

→ Basisstrategie:

lokale Korrekturen:

z.B.: Ersetzung wenn Eingabe nur '=' statt ':=' liefert

leere Tabelleneinträge werden zu Verweisen auf Fehlerbehandlungsroutinen

- Änderung, Eingabe, Löschung von Eingabesymbolen
- explizite Stack-Änderung bleibt jedoch fragwürdig

Panische Fehlerstabilisierung (1)

5 bewährte heuristische Vorgehensweisen (sprachabhängig, konstruktabhängig)

Strategie-1: Terminalsymbole aus FOLLOW(A)
werden als Synchronisationselemente für A eingesetzt:

- „Schlucken“ von Eingabesymbolen
  bis jeweiliges Synchronisationselement tatsächlich als aktuelle Eingabe auftritt
- bei Entfernung von A aus dem Keller

Erweiterung der Synchronisationselemente um gewisse Schlüsselwort-Token,
denn Strategie-1 allein reicht **nicht** bei Sprachen mit hierarchischen Strukturen:

Beispiel: fehlendes Semikolon nach einer Anweisung (in C) bewirkt:
das nächstfolgende Schlüsselwort wird übersprungen
denn: Schlüsselworte gehören nicht zur FOLLOW-Menge von Anweisungsregeln

for ...; while ...; switch ...;

Wir würden zu viel Eingabetext ignorieren

Beispiel: Anwendung von Heuristik-1

… bewährt bei Ausdrücken

	Eingabesymbole					
	id	**+**	*	**(**	**)**	**$**
E	E → TE'			E → TE'		synch
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → **id**	synch	synch	F → **(**E**)**	synch	synch

***besser**: bleibt dennoch frei*

Grammatik:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → **(**E**)** | **id**

α ∈ V	FIRST(α)	FOLLOW(α)
E	**(** , **id**	**)**, **$**
E'	**+**, ε	**)**, **$**
T	**(** , **id**	**)**, **+**, **$**
T'	*, ε	**)**, **+**, **$**
F	**(** , **id**	**)**, *, **+**, **$**

Eintrag M[A, **a**] = **leer**: Eingabesymbol **a** wird übersprungen

Eintrag M[A, **a**] = synch: (d.h. ein Synchronisationselement **a** wurde in der Eingabe gefunden)
wenn oberstes Kellersymbol A ist,
Entfernung von A und Fortsetzung
wenn oberstes Kellersymbol Terminalsymbol (≠ Eingabe)
Anwendung von Heuristik-1

„Schlucken“ von Eingabesymbolen bis jeweiliges Synchronisationselement als aktuelle Eingabe auftritt bei Entfernung von A aus dem Keller

Synchronisationselemente

1	$E	) id * + id $	M[E,)] = leer überspringe)
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			

Fehler am Anfang:
Ausnahme, überspringen!!!

Beispiel: Konfigurationsfolge

Aktionen eines tabellengesteuerten prädiktiven Parsers

bei fehlerhafter Eingabe von

)	id	*	+	id	$

NT	Eingabesymbole					
	id	+	*	(	)	$
E	E → TE'			E → TE'		synch
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id	synch	synch	F → (E)	synch	synch

α ∈ V	FIRST(α)	FOLLOW(α)
E	(, id	), $
E'	+, ε	), $
T	(, id	), +, $
T'	*, ε	), +, $
F	(, id	), *, +, $

Beispiel: Konfigurationsfolge

1	$E	) id * + id $	M[E,)] = leer überspringe)
2	$E	id * + id $	**id** ist in FIRST(E)
3	$E'T	id * + id $	
4	$E'T'F	id * + id $	
5	$E'T'id	id * + id $	
6	$E'T'	* + id $	
7	$E'T'F*	* + id $	
8	$E'T'F	+ id $	M[F,+] = synch Entferne F
9			
10			
11			
12			
13			
14			
15			
16			
17			

)	id	*	+	id	$

NT	Eingabesymbole					
	id	+	*	(	)	$
E	E → TE'			E → TE'		synch
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id	synch	synch	F → (E)	synch	synch

α ∈ V	FIRST(α)	FOLLOW(α)
E	(, id	), $
E'	+, ε	), $
T	(, id	), +, $
T'	*, ε	), +, $
F	(, id	), *, +, $

Beispiel: Konfigurationsfolge

1	$E	) id * + id $	M[E,)] = leer !!! d.h. überspringe)
2	$E	id * + id $	
3	$E'T	id * + id $	
4	$E'T'F	id * + id $	
5	$E'T'id	id * + id $	
6	$E'T'	* + id $	
7	$E'T'F*	* + id $	
8	$E'T'F	+ id $	M[F,+] = synch entferne F
9	$E'T'	+ id $	
10	$E'	+ id $	
11	$E'T+	+ id $	
12	$E'T	id $	
13	$E'T'F	id $	
14	$E'T'id	id $	
15	$E'T'	$	
16	$E'	$	
17	$	$	

)	id	*	+	id	$

NT	Eingabesymbole					
	id	+	*	(	)	$
E	E → TE'			E → TE'		synch
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id	synch	synch	F → (E)	synch	synch

α ∈ V	FIRST(α)	FOLLOW(α)
E	(, id	), $
E'	+, ε	), $
T	(, id	), +, $
T'	*, ε	), +, $
F	(, id	), *, +, $

Panische Fehlerstabilisierung (2)

Strategie-2: Anfangssymbole höher stehender Konstrukte als Kandidaten für die Synchronisationsmenge tiefer angesiedelter Konstrukte

Beispiel: einleitende Schlüsselworte von Anweisungen werden Synchronisationsmengen von Ausdrücken zugeordnet

Strategie-3: Aufnahme von Symbolen aus FIRST(A) für Synchronisationsmenge für A (als aktuelle Variable des Stacks) an Tabellenpositionen ungültiger Eingaben

Ziel.: mögliche Fortsetzung, sobald ein Element aus FIRST(A) in der Eingabe auftritt

d.h.: alle Symbole der Eingabe werden solange „geschluckt“, bis in der Eingabe ein Element aus **FIRST**(A) auftritt, so dass sich der Compiler wieder durch Anwendung einer passenden Regel für A fangen kann

Panische Fehlerstabilisierung (3)

Strategie-4: Komplementäre Token als Synchronisationselemente

Situation: das oberste Kellersymbol (wenn es ein Terminalsymbol ist) **stimmt nicht** mit dem aktuellen Eingabesymbol überein,

Synchronisationsmenge besteht jeweils immer aus Token, die fälschlicher Weise in der Eingabe auftreten können

Aktion: Kellersymbol wird entfernt (Match mit virtuell eingefügten Token) und eine Fehlerausschrift wird erzeugt, die besagt, dass ein Token vermisst wurde

Effekt: neues Symbol auf dem Stack wird aufgedeckt und ermöglicht eine Fortsetzung

Strategie-5: Reduktion durch erzwungene ε-Ableitung

Situation: für das oberste Kellersymbol (wenn es eine Variable ist) gibt es **keine** Übereinstimmung
aber: es gibt eine Grammatik-Regel für diese Variable, die nach ε ableitet,

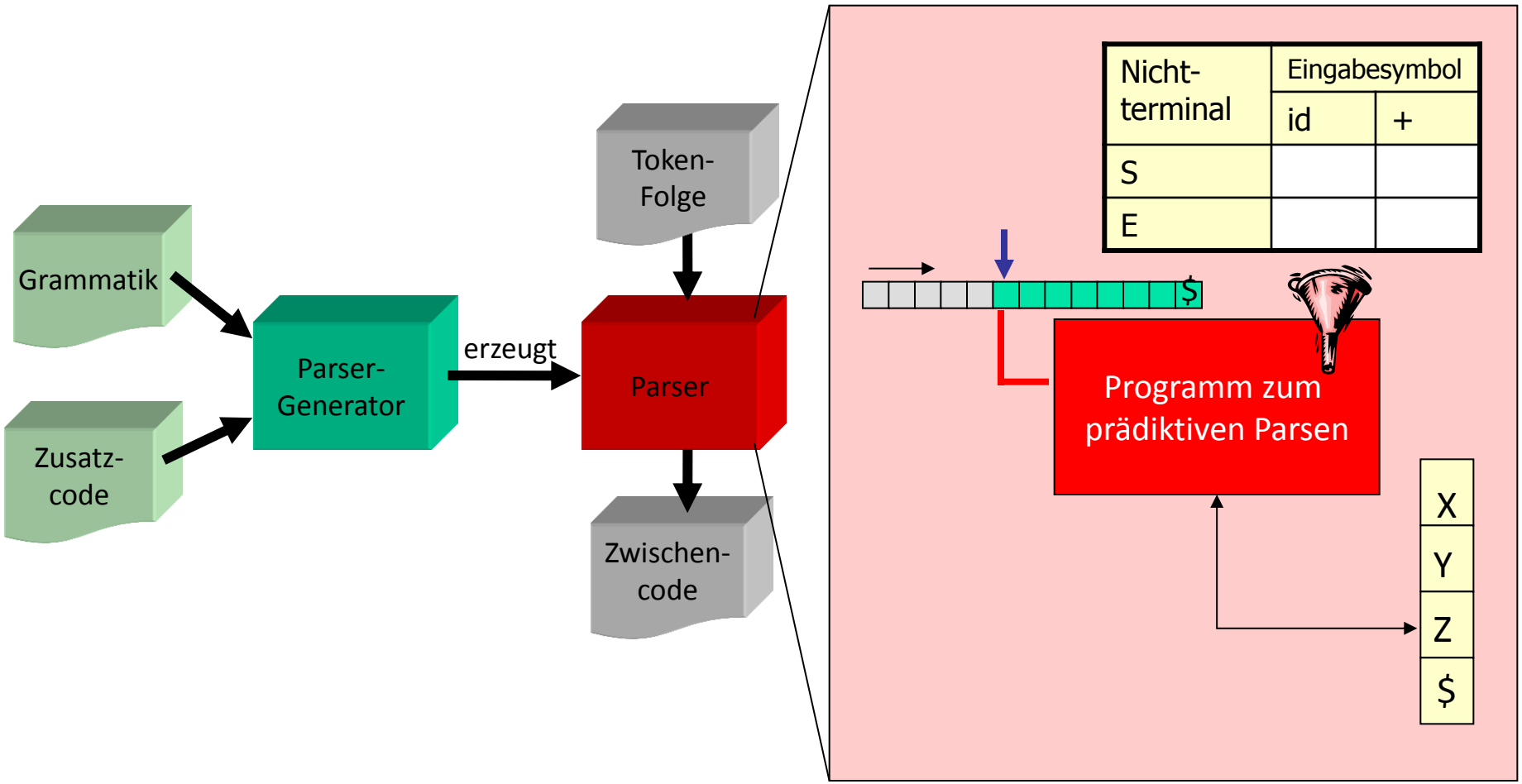
Aktion: diese Ableitung wird nun auch in der Fehler-Konstellation zur Anwendung gebracht

Effekt: neues Symbol auf dem Stack wird aufgedeckt und ermöglicht eine Fortsetzung

4.5 Tabellengesteuerte (deterministische) Top-Down-Verfahren

- Allg. Arbeitsweise eines tabellengesteuerten Top-Down-Parsers
- Konstruktion von Parser-Tabellen für nicht-rekursive prädiktive Parser
- Bestimmung synchronisierender Terminalsymbole zur Fehlerstabilisierung
- Parser-Generatoren für LL(1)-Sprachen (kurzer Einblick)

Konstruktion eines tabellengesteuerten Parsers

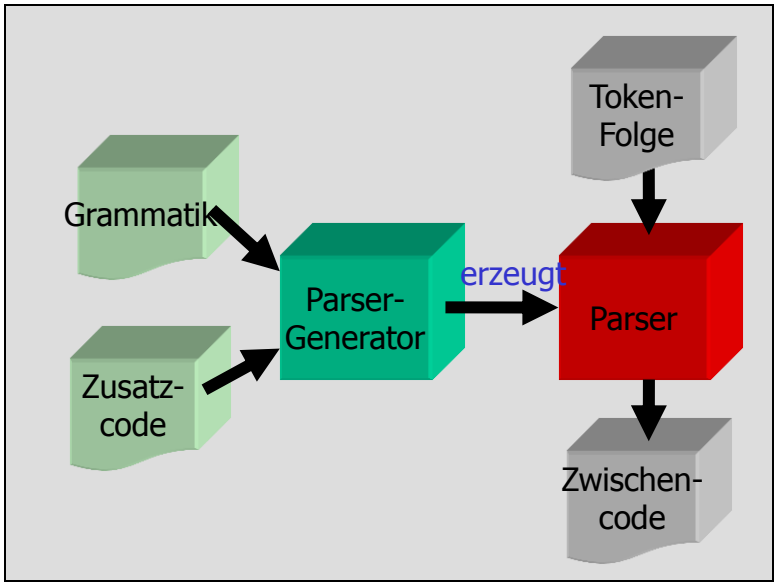


Ziel: Automatische Konstruktion eines flexiblen Parsers

Parser-Generatoren für LL(1)-Sprachen

Arbeitsweise

- Eingabe: beliebige kontextfreie Grammatik G
- Ausgabe:
  - Parser für LL(1)-Grammatik (tabellengesteuert)
  - negativer Fall: Infos über LL(1)-Verletzung
    z.B.: Linksrekursivität



1. Berechnung von M(ε): M(ε)= {A| A→*e}
2. Bestimmung von FIRST(a_i) für alle Alternativen
3. Bestimmung von FOLLOW(m) für alle m ∈ M (ε)
4. Entscheidung, ob G eine LL(1)-Grammatik
5. aus Informationen oberer Schritte wird eine Parser-Tabelle erstellt
6. der generierte Parser ist unabhängig von der Grammatik
   und arbeitet auf Grundlage der Parser-Tabelle

LL(1)-Parser-Generatoren

- **JavaCC** (Java Compiler Compiler)
  - in Java implementiert und erzeugt Java-Code
  - erzeugt einen LL(k)-Parser
  - als freie Software verfügbar

- **ANTLR** (**ANother Tool for Language Recognition**)
  - in Java implementiert, als freie Software verfügbar
  - entwickelt 1989 von Terence Parr an der Universität von San Francisco
  - unterstützt Erzeugung von Parsern, Lexern und TreeParsern für LL(k)-Grammatiken mit **beliebigen** k
  - verfügbar für Java-Plattform sowie .NET und weitere
  - Zielsprachen C, C++, C#, Java, Objective-C und Python