Technische Universität
Kaiserslautern

o1o1seda o1o1oo

software engineering dependability

Software Entwicklung 2

Übersetzerbau 2

Inhalt

- Reguläre Grammatiken und lexikalische Analyse
  - Eigenschaften und Aufgaben
  - Scanner
  - First- und Follow-Menge
  - *Einschub: Kurzeinführung Nassi-Shneiderman-Diagramme*
  - Beispiele für Scannerroutinen
- Kontextfreie Grammatiken und Syntaxanalyse
  - Bedingungen für Grammatiken
  - Charakteristiken kontextfreier Grammatiken
  - Syntaxanalyse durch rekursiven Abstieg: Parser
  - Beispiel zur Implementierung
  - LL(1)-Grammatik
- Attributierte Grammatiken: Codegenerierung
- Ausblick
  - Weitere Compilerbautechniken
  - Compiler-Compiler

Lernziele

- Eigenschaften regulärer bzw. kontextfreier Grammatiken angeben können
- Eine gegebene Grammatik darauf überprüfen können, ob sie regulär oder kontextfrei ist.
- Eine reguläre Grammatik als Zustandsautomat darstellen können sowie zu einem Zustandsautomaten die reguläre Grammatik angeben können
- Die Syntaxanalyse durch rekursiven Abstieg anwenden können

- Die lexikalische Analyse wird durch den Scanner durchgeführt
- Die zugrundeliegende Grammatik ist regulär (Chomsky-Typ 3)
- Reguläre Grammatiken können durch endliche Automaten dargestellt werden
- Lexikalische Analyse:
  - Das Quellprogramm wird Zeichen für Zeichen gelesen
  - Die individuellen Zeichen werden zu Symbolen *(tokens)* zusammengefasst, die die Eingabe für den Parser (syntaktische Analyse) darstellen
  - Beispiele für Symbole sind Bezeichner, Zahlen, Operatoren, Begrenzer wie Klammern und Semikolon
  - Der Scanner unterscheidet aus Sicht des Parsers unterschiedliche Symbole (z.B. Bezeichner vs. Schlüsselworte) und übermittel ggf. deren Inhalte (z.B. das Symbol *number* mit dem Inhalt *35*)

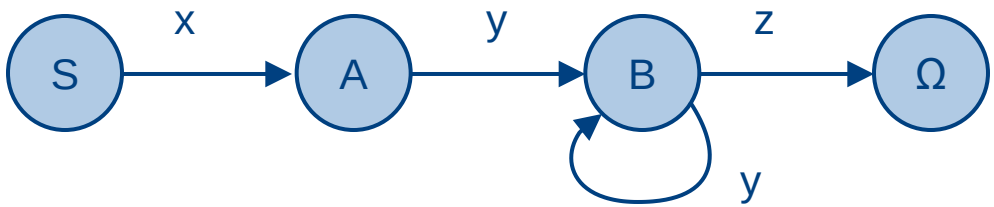
- Die Erkennung von Sätzen regulärer Grammatiken ist ausgesprochen einfach, da sie mit dem zugrundeliegenden Zustandsautomaten durchgeführt werden kann:
- Beispiel: G = (N,T,P,S)

  T = {x, y, z}

  N = {S, A, B}

  P = {S → x A, A → y B, B → y B, B → z}

  Erzeugt die Sprache L = { x y^+ z }



- Beispiel: Reguläre Grammatik für Bezeichner und Zahlen:

G = (N, T, P, Ident_Num)

T = {0, 1,..., 9, A, B, ..., Z} *Anmerkung: 0..9: digit; A..Z: letter*

N = {Ident_Num, Num, Ident}

P = {

Ident_Num →	"A" | "B" | ... | "Z" | "A" Ident | "B" Ident | ... | "Z" Ident |
	"0" | "1" | ... | "9" | "0" Num | "1" Num | ... | "9" Num,
Num →	"0" | "1" | ... | "9" | "0" Num | "1" Num | ... | "9" Num,
Ident →	"A" | "B" | ... | "Z" | "A" Ident | "B" Ident | ... | "Z" Ident |
	"0" | "1" | ... | "9" | "0" Ident | "1" Ident | ... | "9" Ident

}

- Beispiel: Reguläre Grammatik für Bezeichner und Zahlen:

G = (N, T, P, Ident_Num)

T = {0, 1,..., 9, A, B, ..., Z} *Anmerkung: 0..9: digit; A..Z: letter*

N = {Ident_Num, Num, Ident}

P = {

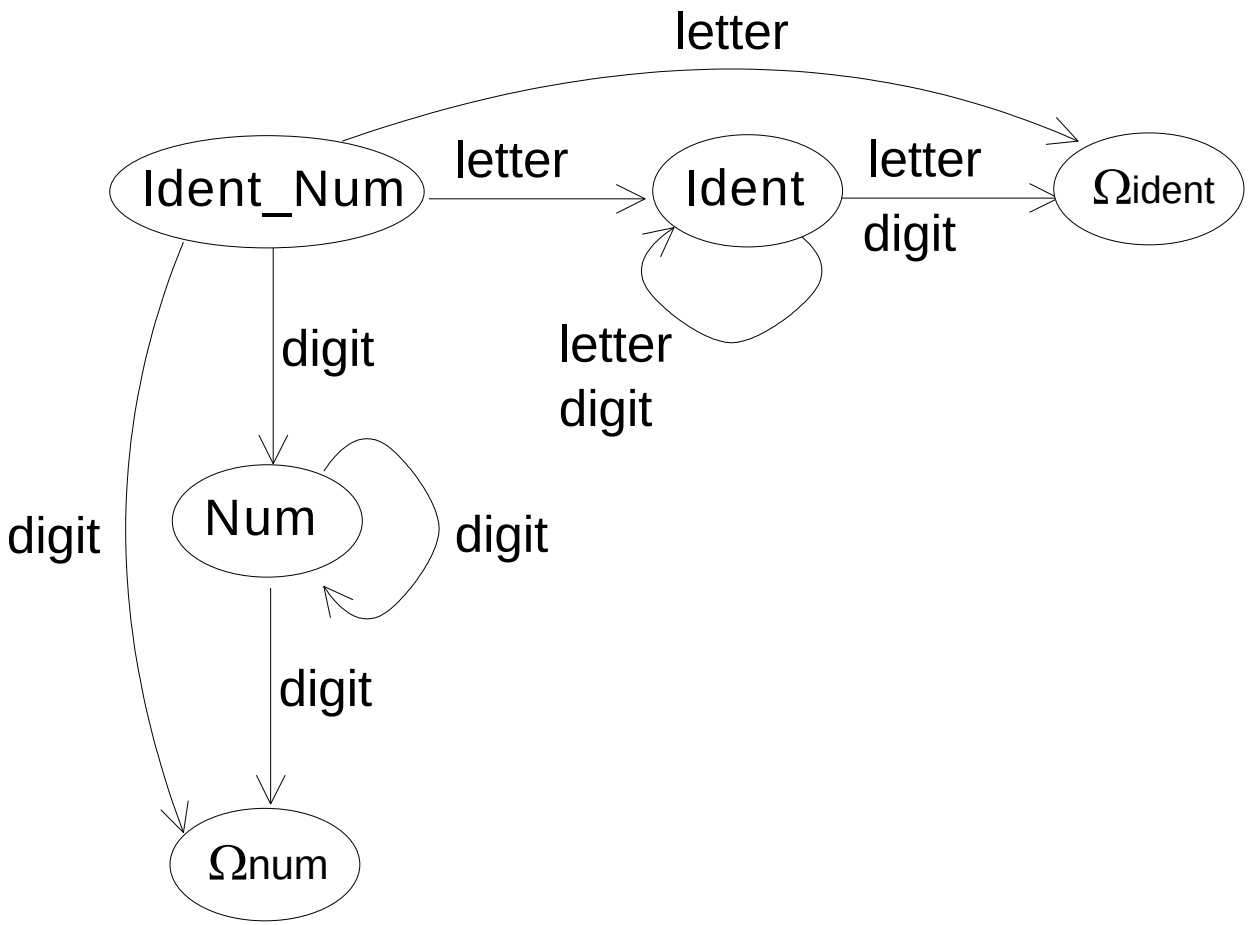
Ident_Num → letter | letter Ident | digit | digit Num,

Num → digit | digit Num,

Ident → letter| letter Ident | digit | digit Ident

}

- Beispiel: Reguläre Grammatik für Bezeichner und Zahlen:
-



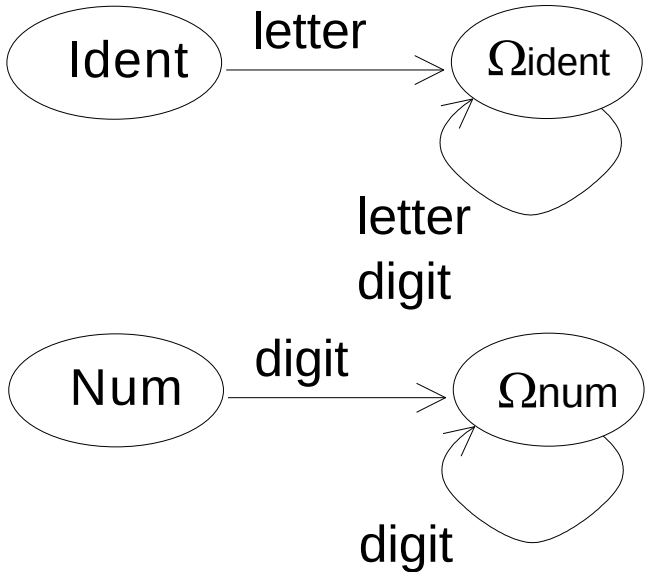
Sackgassen

- Ableitungsalternativen für die Zahl 95
  - Ident_Num => digit Num => 9 Num => 9 digit Num => 95 Num => ?
  - Ident_Num => digit => 9 ?
  - Ident_Num => digit Num => 9 Num => 9 digit => 95 (OK!)

- Beispiel: Reguläre Grammatik für Bezeichner und Zahlen:

  Ident ::= letter {letter | digit}.

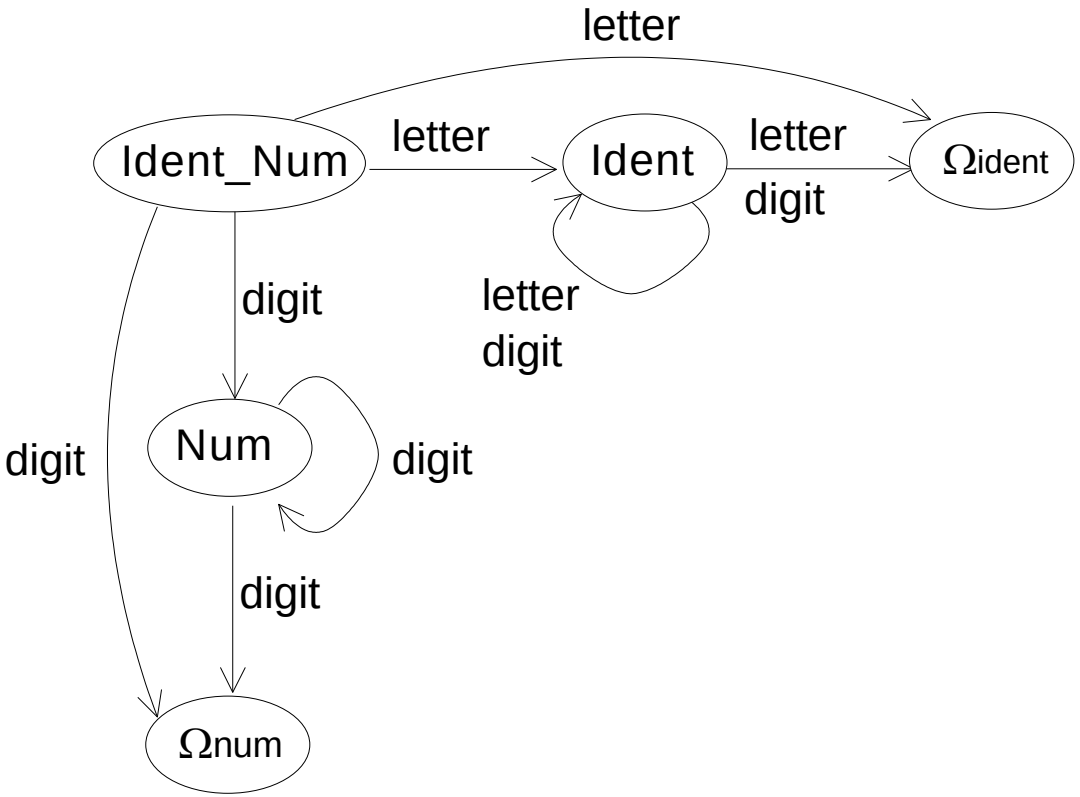
  Num ::= digit {digit}.



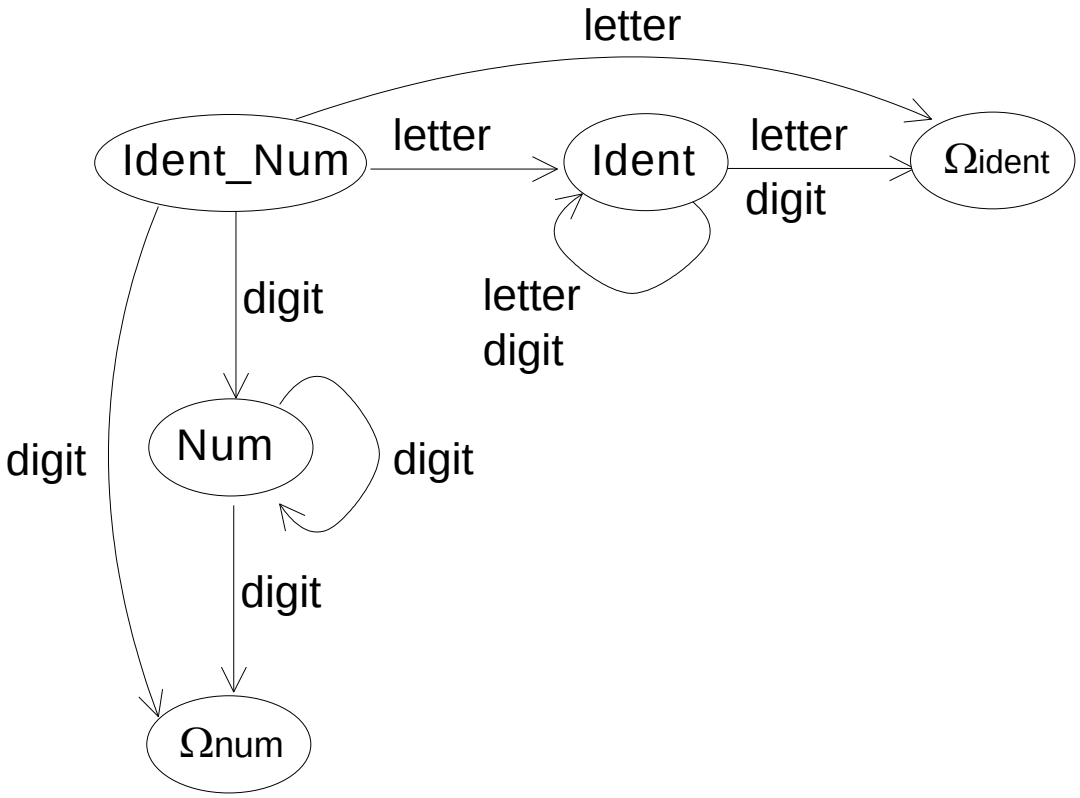
- Reguläre Grammatiken und endliche Automaten sind äquivalente Darstellungen:
- Reguläre Grammatiken können mit endlichen Automaten analysiert werden => lexikalische Analyse
- Der Analyseautomat liest in jedem Zustand das nächste Eingabezeichen und führt einen Schaltschritt durch:
  - Der Folgezustand hängt allein vom Startzustand und Eingabezeichen ab
  - Ist der Folgezustand eindeutig bestimmt, so ist der Automat deterministisch, sonst ist er nicht-deterministisch (Für lexikalische Analyse ist ein deterministischer Automat erforderlich)
  - Ist das Eingabezeichen nicht erlaubt, so ist ein Fehler erkannt (d.h. die Sequenz der Eingabezeichen bildet kein gültiges Symbol)

Beispiel: Der angegebene Automat ist nicht-deterministisch:

- Im Zustand *Num* kann aufgrund des Eingabezeichens *digit* nicht entschieden werden, ob der Folgezustand *Num* oder Ω_{num} sein muß.



- Beispiel: Die Zeichenfolge *95%* ist weder ein erlaubter Bezeichner noch eine erlaubte Zahl => Fehler



- Regel des längsten Musters
  - Solange ein gültiges Symbol vorliegt, werden weitere Zeichen hinzugefügt
  - Erst wenn ein Zeichen kommt, dass nicht mehr zu einem gültigen Symbol führt, ist das Symbol erkannt
- Beispiel
  - *beginner*
    - Ist Bezeichner (*beginner*)
    - Ist nicht das Symbol **beginSym** (*begin*) gefolgt vom Bezeichner *ner*

First- und Follow-Menge

- Menge first(A)
  - Menge aller terminalen Symbole, mit denen eine aus A abgeleitete Symbolfolge anfangen kann
    - First(A) = {a: A=>* a ω; für a $\in$ T und $\omega \in$ V*}
    - Insbesondere: First(ε) = $\varnothing$
- Beispiel: Reguläre Grammatik für Bezeichner und Zahlen:

  Ident ::= letter {letter | digit}.

  Num ::= digit {digit}
  - First(Ident) = {A, B, C, ..., Z}
  - First(Num) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Die First-Mengen von Alternativen müssen disjunkt sein:
  Gegenbeispiel *H95*: Bezeichner *H95* oder hexadezimale Zahl *95*
  Bessere Schreibweise für Hex-Zahlen: *95H*

First- und Follow-Menge

- Menge follow(A)
  - Menge aller terminalen Symbole, die in irgendeiner Satzform auf A folgen können
    - Follow(A) = {a: S=>* ω_1 A a ω_2; für A $\in$ N, a $\in$ T und ω_1, ω_2 $\in$ V*}

- Beispiel:
  - SATZ ::= SUBJEKT PRÄDIKAT.
  - SUBJEKT ::= "Peter" | "Siegfried".
  - PRÄDIKAT ::= "arbeitet" | "schläft".

  - Follow(SUBJEKT) = {arbeitet, schläft}

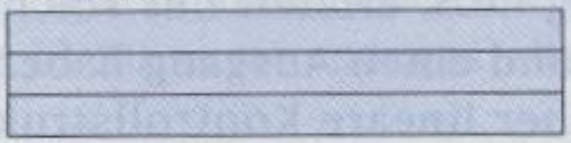
- Regeln für die Realisierung eines Scanners auf Basis einer EBNF
- Voraussetzungen:
  - Zu jeder Produktionsregel R existiert ein gleichnamiges Programmfragment P(R) (Prozedur oder Funktion) das die Regel R implementiert
  - Die Funktion *next* liefert das nächste Eingabezeichen als Inhalt der globalen Variable *sym* zurück
  - Die Funktion *error* meldet einen Fehler
  - A_x ist ein nichtterminales Symbol
  - t ist ein Terminalsymbol

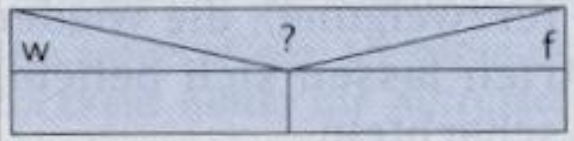
t	if (sym = t) {next} else {error}
A	P(A)
[A]	if (sym $\in$ first(A)) {P(A)}
{A}	while (sym $\in$ first(A)) {P(A)}
$A_1\ A_2\ ...\ A_n$	$P(A_1); P(A_2); ... P(A_n);$

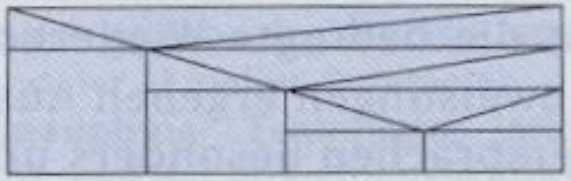
$A_1 \mid A_2 \mid ... \mid A_n$	switch (sym $\in$ ...)
	{ case ... first(A_1): $P(A_1)$; break;
	case ... first(A_2): $P(A_2)$; break;
	...
	case ... first(A_n): $P(A_n)$; break;
	default: error; break; }

Regeln für die Realisierung eines Scanners auf Basis einer EBNF

- Bedingungen:
  - Die first-Mengen von Alternativen (A_1 | A_2) müssen disjunkt sein,
    d.h. first (A_1) $\cap$ first(A_2) = $\varnothing$, weil sonst die vorliegende Alternative nicht bestimmt werden kann
  - Die first-Mengen von Sequenzen (A_1 A_2) müssen disjunkt sein,
    d.h. first (A_1) $\cap$ first(A_2) = $\varnothing$, falls A_1 in den leeren Satz ε abgeleitet werden kann (A_1 =>* ε), weil sonst nicht entschieden werden kann, ob erst A_1 oder gleich A_2 bearbeitet werden muß.
  - Bei Optionen [A_1] und Wiederholungen {A_1} müssen die die first- und follow-Mengen disjunkt sein,
    d.h. first (A_1) $\cap$ follow(A_1) = $\varnothing$, weil sonst nicht entschieden werden kann, ob die Option vorliegt bzw. ob noch eine Wiederholung bearbeitet werden muß.

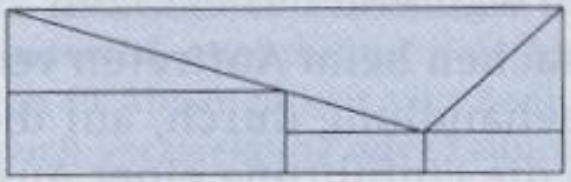
Sprachkonzept	Java
Sequenz 	Anweisung1; Anweisung2; Anweisung3;

Ein- und zweiseitige Auswahl 	if(expression)
	{...}
	[else {...}]
	//end if

Auswahlketten 	if(expression)
	{...}
	else if(expression)
	{...}
	//end if
	//kein eigenes Konstrukt

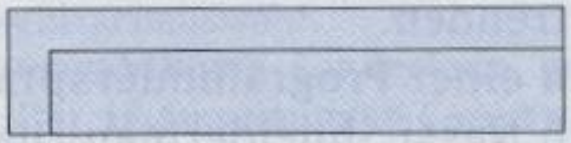
Kurzeinführung: Nassi-Shneiderman-Diagramme

Mehrfachauswahl



```
switch(expression)
{    case expression: ...; break;
     case...:...; break;
     default:...; break;
}
//end switch
```

while-Schleife

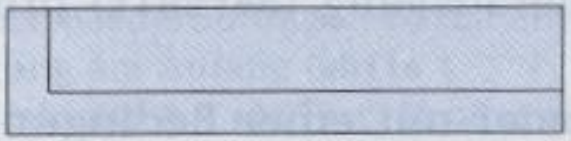


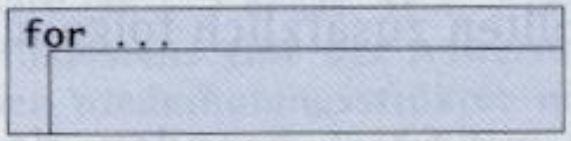
```
while(expression)
{...
}
//end while
```

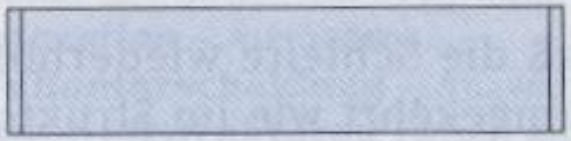
n+1/2-Schleife


```
while(true)
{...
   if(expression) break;
}
//end while
//kein eigenes Konstrukt
```

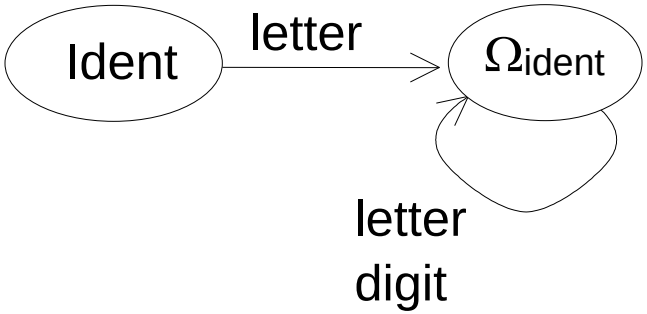
Kurzeinführung: Nassi-Shneiderman-Diagramme

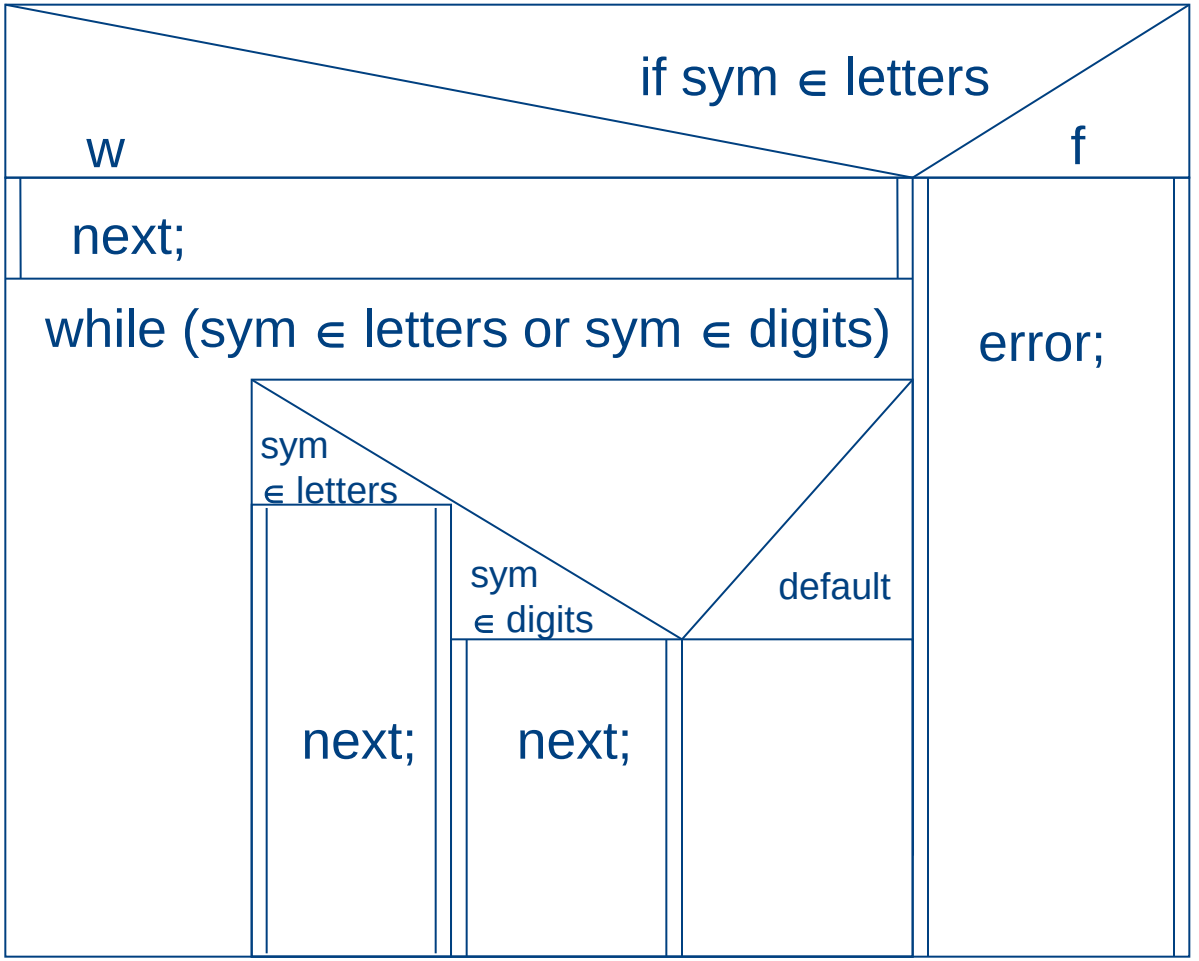
Schleife mit Bedingung am Ende 	`do`
	`{...`
	`}`
	`while(expression);`

Zählschleife 	`for(i=1; i<=10; i=i+1)`
	`{...`
	`}`
	`//end for`

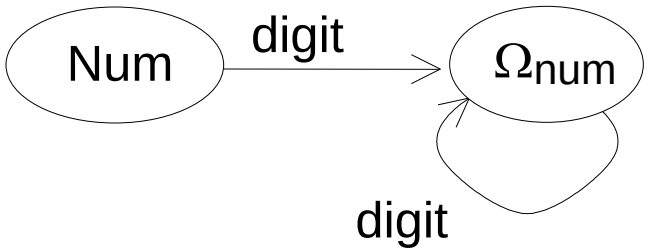
Aufruf 	`Operationsname(Parameter1,` `Parameter2, ...);`

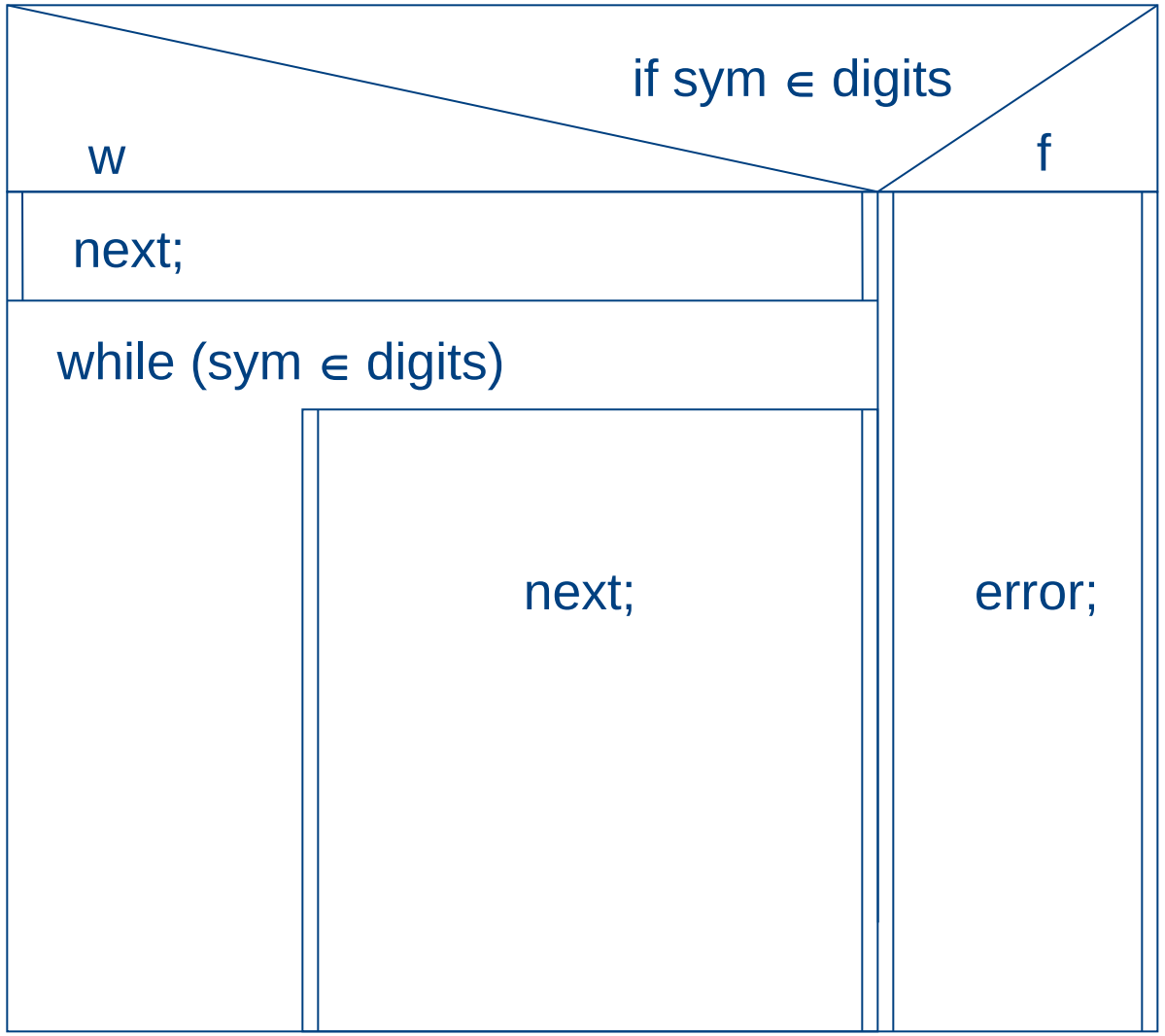
- Ident ::= letter {letter | digit}.





- Num ::= digit {digit}.





- Ziel einer Sprache
  - Einwandfreie Analysierbarkeit
- Notwendige Eigenschaften
  - Reduzierung
    - Terminalisierbarkeit
    - Ableitbarkeit
  - Zyklenfreiheit
  - Eindeutigkeit
  - Determinierbarkeit

Reduzierte Grammatik

- Grammatik ohne überflüssige nicht-terminale Symbole
  - Beitrag jedes nicht-terminalen Symbols N zur Erzeugung von Sätzen
- Terminalisierbarkeit
  - Ableitbarkeit in eine terminale Kette
  - $N \Rightarrow^+ t$ mit $t \in T^*$
- Ableitbarkeit
  - Vorkommen als Satzform
  - $S \Rightarrow^* \omega_1 N \omega_2$

- Nach der Beseitigung überflüssiger Symbole kann die Grammatik immer noch überflüssige Alternativen enthalten!
  - zirkuläre Ableitungen der Form $A =>^{+} A$
  - entsprechende Grammatik heißt zirkulär oder zyklenbehaftet
- Eine kontextfreie Grammatik heißt eindeutig
  - wenn sie für jeden Satz genau einen Syntaxbaum besitzt
  - Gegenbeispiel:

    EXPRESSION ::= EXPRESSION – EXPRESSION

    EXPRESSION ::= Id

    Ableitung von Id-Id-Id?: Syntaxbaum?

- Grammatiken, die so gebaut sind, dass
  - man durch Heranziehen eines Teils der zu erkennenden Symbolkette
  - die richtige Produktionsalternative mit Sicherheit feststellen kann,
  - ohne in eine Sackgasse zu laufen!

- Im Compilerbau nur deterministische Grammatiken!

- Mit regulären Grammatiken können aufgrund der Struktur der Produktionsregeln keine Schachtelstrukturen dargestellt werden:
  - Reguläre Sprachen dürfen nur Produktionen der folgenden Form enthalten: A $\rightarrow$ a oder A $\rightarrow$ a B (darstellbar als endlicher Automat)
- In der Programmierung benötigt man aber Schachtelstrukturen:
  - Beispiel: In eine Prozedur ist eine Prozedur eingeschachtelt, in die eine weitere Prozedur eingeschachtelt ist.
  - Erfordert Produktionsregeln der Form: A $\rightarrow$ a A b
    (kontextfreie Grammatik: Darstellung durch Kellerautomaten)
  - Werden durch Parser analysiert

- Beispiel: »Klammergebirge« beschreibbar
  - G = (N,T,P,S)
  - N = { S }
  - T = { (,) }
  - P = { S → (S) S, S → ε }
  - S = { S }
- Diese Grammatik beschreibt eine Sprache mit korrekter Klammerbilanz, z.B. (((( ))))(()))
- Es gibt keine reguläre Grammatik, die die gleiche Sprache erzeugt

- Zu jedem nichtterminalen Symbol (und damit auch zu jeder Produktionsregel) existiert eine Analyseroutine, die jede terminale Symbolfolge erkennt, die durch das nichtterminale Symbol erzeugt werden kann:
  - Hinweis: Bei kontextfreien Grammatiken stehen auf der linken Seite von Produktionen stets einzelne Nichtterminalsymbole, so daß immer eine Umsetzung einer Produktionsregel in genau eine Analyseroutine erreicht werden kann

    Beispiel: $A \rightarrow a\ A\ c$

    $A \rightarrow b$

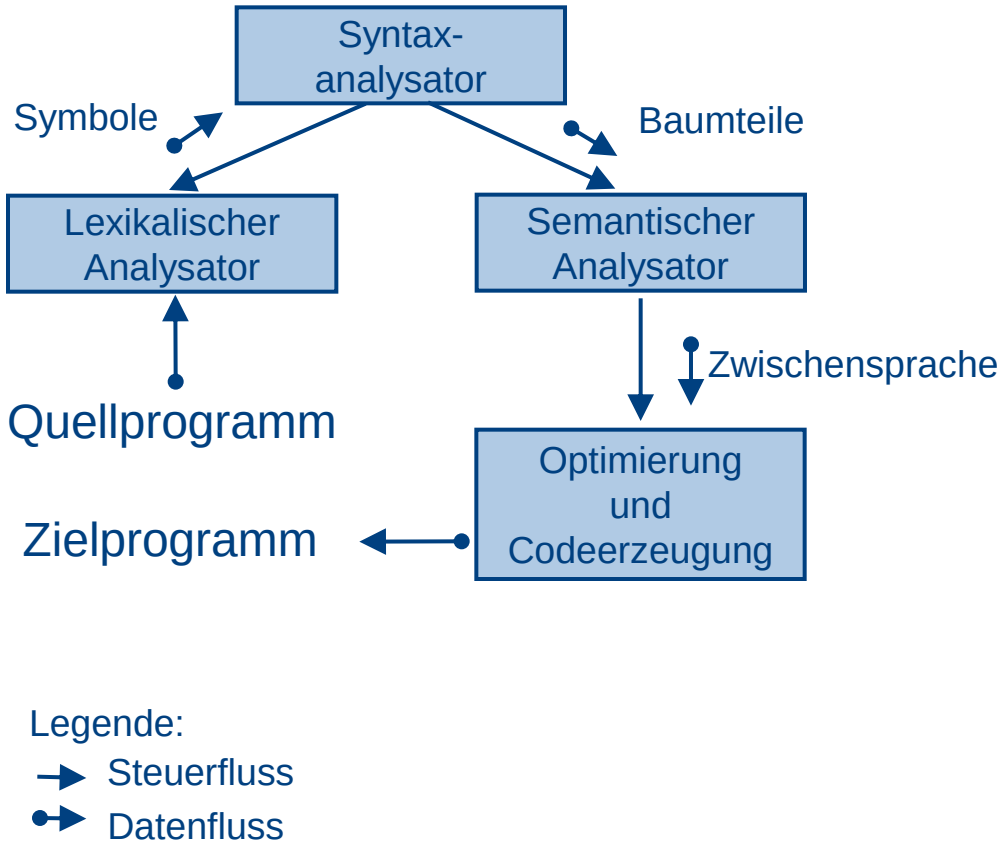
    kann zusammengefaßt werden: $A \rightarrow a\ A\ c \mid b$

    und anschließend in eine Analyseroutine umgesetzt werden
- In der Analyseroutine werden sowohl nichtterminale als auch terminale Symbole bearbeitet

- Beginn mit dem Syntaxdiagramm Start bzw. der Produktionsregel, auf deren linker Seite das Startsymbol steht
- Alle Syntaxdiagramme bzw. Produktionsregeln solange anwenden, bis nur noch terminale Symbole vorhanden sind
- Jede Sequenz terminaler Symbole, die so erzeugt werden kann, ist gültig
- Umgekehrt gilt, dass jede Sequenz terminaler Symbole, die nicht durch eine Sequenz von Ersetzungen nicht-terminaler Symbole erzeugt werden kann, illegal ist
- Entspricht der (rekursiven) Anwendung der Analyseroutinen, die durch den Syntaxbaum absteigen, den sie bei der Programmbearbeitung erkennen
  - Einfachste Syntaxanalyse-Technik
  - wird in vielen Compilern verwendet

Ablauf der Syntaxanalyse durch rekursiven Abstieg: Parser

- Der Parser beginnt mit der Ausführung der Analyseroutine des Startsymbols
- Er fordert den Scanner entsprechend des Fortschritts der Syntaxanalyse auf, das nächste Symbol zu liefern
- Er generiert entsprechend des Fortschritts der Syntaxanalyse Code (Aufruf der Codegenerierungsroutinen)



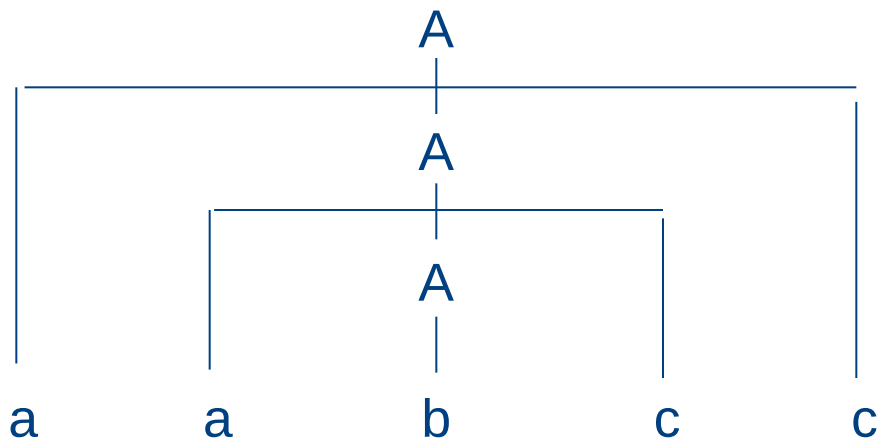
Syntaxanalyse durch rekursiven Abstieg: Beispiel

- G=(N, T, P, A); N = { A }; T = { a, b, c }; P = { A → a A c | b }
- Analyse des Satzes: aabcc
- Initialisierung:
  - Der Parser fordert den Scanner auf, das erste Symbol zu liefern: *a*
  - Die Analyseroutine für das Startsymbol *A* wird gestartet
- Analyse:
  - Das aktuelle Symbol *a* ist erlaubt (Beginn der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *a*; aktuelles Symbol: *a*)
    - Die erste Alternative der Produktionsregel muß weiter abgearbeitet werden, d.h. die Analyseroutine *A* wird rekursiv aufgerufen
    - Das aktuelle Symbol a ist erlaubt (Beginn der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *aa*; aktuelles Symbol: *b*)

- Fortsetzung Analyse:
  - Die erste Alternative der Produktionsregel muß weiter abgearbeitet werden, d.h. die Analyseroutine *A* wird rekursiv aufgerufen
  - Das aktuelle Symbol *a* ist erlaubt (Beginn der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *aa*; aktuelles Symbol: *b*)
    - Die erste Alternative der Produktionsregel muß weiter abgearbeitet werden, d.h. die Analyseroutine *A* wird rekursiv aufgerufen
    - Das aktuelle Symbol *b* ist erlaubt (Beginn der zweiten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *aab*; aktuelles Symbol: *c*)
    - Diese Rekursionsebene ist damit abgeschlossen, und es wird in die vorhergehende Ebene zurückgesprungen
  - Das aktuelle Symbol *c* ist erlaubt (Ende der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *aabc*; aktuelles Symbol: *c*)
  - Diese Rekursionsebene ist damit abgeschlossen, und es wird in die vorhergehende Ebene zurückgesprungen

- Fortsetzung Analyse:
  - Das aktuelle Symbol *c* ist erlaubt (Ende der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *aabcc*; aktuelles Symbol: *Endesymbol*)
- Der Satz *aabcc* ist aus dem Startsymbol *A* hergeleitet und die Eingabe ist beendet. Daher ist die Syntaxanalyse abgeschlossen.

- Syntaxbaum:

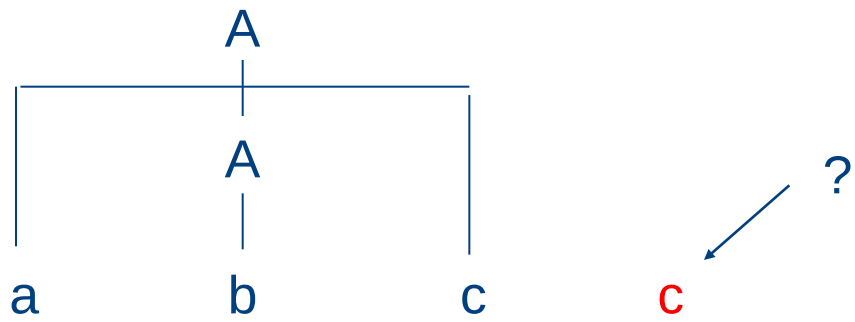


- G=(N, T, P, A); N = { A }; T = { a, b, c }; P = { A → a A c | b }
- Analyse des Satzes: *abcc*
- Initialisierung:
  - Der Parser fordert den Scanner auf, das erste Symbol zu liefern: *a*
  - Die Analyseroutine für das Startsymbol *A* wird gestartet
- Analyse:
  - Das aktuelle Symbol *a* ist erlaubt (Beginn der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *a*; aktuelles Symbol: *b*)
    - Die erste Alternative der Produktionsregel muß weiter abgearbeitet werden, d.h. die Analyseroutine *A* wird rekursiv aufgerufen
    - Das aktuelle Symbol *b* ist erlaubt (Beginn der zweiten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *ab*; aktuelles Symbol: *c*)

- Fortsetzung Analyse:
  - Diese Rekursionsebene ist damit abgeschlossen, und es wird in die vorhergehende Ebene zurückgesprungen
  - Das aktuelle Symbol *c* ist erlaubt (Ende der ersten Alternative); also ist der Satz bis hierher als gültig erkannt und der Scanner wird aufgefordert, das nächste Symbol zu liefern: (bisher erkannter Satz: *abc*; aktuelles Symbol: *c*)
- Die Syntaxanalyse ist ausgehend vom Startsymbol *A* vollständig abgearbeitet; statt des Endesymbols lieferte der Scanner aber ein weiteres Terminalsymbol

  => Fehler

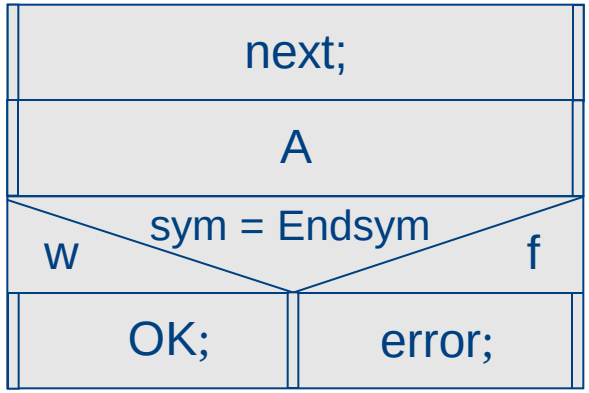
- Syntaxbaum:



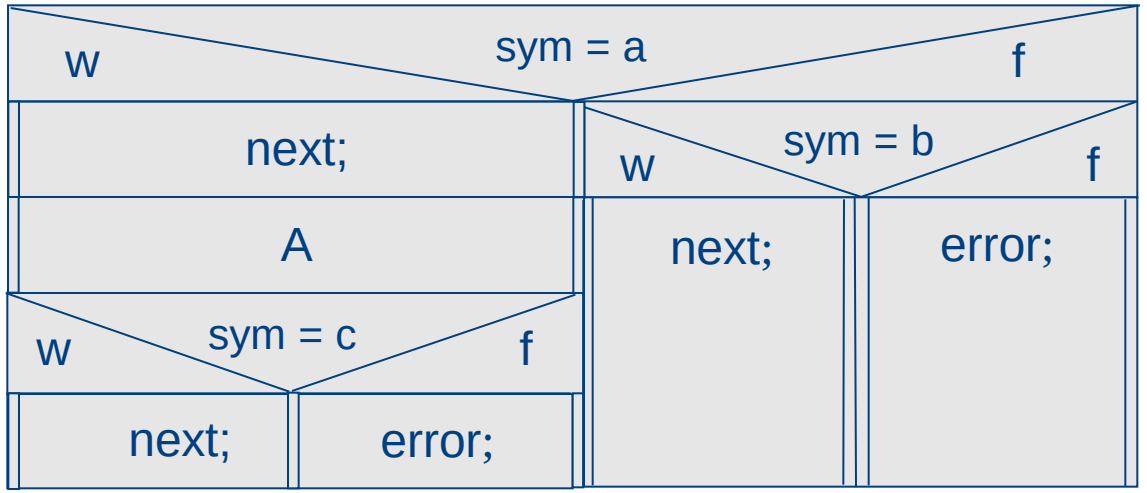
- G = (N, T, P, A); N = { A }; T = { a, b, c }; P = { A → a A c | b }
- Analyse des Satzes: *cba*
- Initialisierung:
  - Der Parser fordert den Scanner auf, das erste Symbol zu liefern: *c*
  - Die Analyseroutine für das Startsymbol *A* wird gestartet
- Analyse:
  - Das aktuelle Symbol *c* ist nicht erlaubt, da keine der beiden Alternativen mit dem Symbol *c* beginnt => Fehler
- Syntaxbaum:

$A \rightarrow a\ A\ c \mid b$

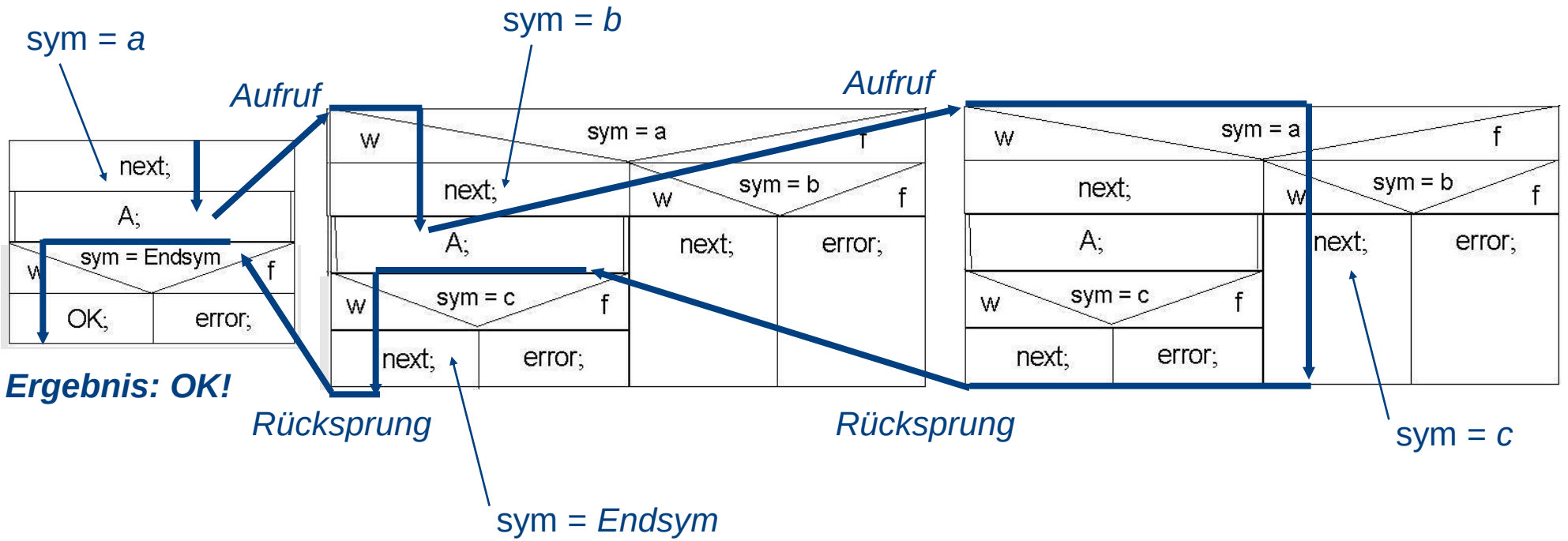
- Initialisierung:



- Analyseroutine *A*:



- Anlauf des Syntaxanalyse des Satzes abc entsprechend der Syntax:
- A → a A c | b



Voraussetzung für das beschriebene Verfahren:

- Bei der Top down-Analyse (Vom Startsymbol beginnend in Richtung terminaler Symbole)
- von links nach rechts
- ist in jeder Situation, in der man zwischen mehreren Alternativen wählen muss,
- durch Vorgriff um 1 Symbol entscheidbar, welche Alternative die richtige ist

=> sogenannte LL(1)-Grammatik

Erweiterung des Grammatik um sogenannte Attributregeln:

- Beispiel: Grammatik für arithmetische Ausdrücke (mit Klammern):

G= (N, T, P, Exp)

N = {Exp, Term, Factor}

T = { number, +, -, *, /, (,) }

P = { Exp $\rightarrow$ Term | Exp + Term | Exp – Term,
Term $\rightarrow$ Factor | Term * Factor | Term / Factor,
Factor $\rightarrow$ number | (Exp) }

Sätze der Sprache, z. B.:

15 – 3

3 * (45 – 15)

(4)

Ableitungen:

- 15 – 3:
  Exp → Exp – Term → Term – Term → Factor – Term → number – Term → number – Factor → number – number

- 3 * (45 – 15):
  Exp → Term → Term * Factor → Factor * Factor → number * Factor → number * (Exp) → number * (Exp – Term) → number * (Term – Term) → number * (Factor – Term) → number * (number – Term) → number * (number – Factor) → number * (number – number)

- (4):
  Exp → Term → Factor → (Exp) → (Term) → (Factor) → (number)

Attributierte Grammatiken: Codegenerierung

- Feststellung:
- Will man nur erkennen, ob ein Ausdruck syntaktisch korrekt ist, so reicht es aus, festzustellen, daß der Ausdruck *15 – 3* die Form
  number – number besitzt, und daher syntaktisch korrekt ist
- Will man den Ausdruck inhaltlich (semantisch) auswerten, so reicht es offensichtlich nicht aus, zu erkennen, daß der Ausdruck *15 – 3* die Form *number – number* besitzt, sondern man muß z.B. den "Wert" des jeweiligen Symbols beachten
- Erforderliche Erweiterungen:
  - Es ist erforderlich, daß der Scanner den entsprechenden "Wert" eines Symbols an den Parser übergibt
  - Die Produktionen müssen um sogenannte Attribute erweitert werden, die bei der Syntaxanalyse Informationen weiterreichen
  - Attributregeln beschreiben, wie die Informationen verarbeitet werden

Beispiel: Semantisch korrekte Auswertung eines arithmetischen Ausdrucks:

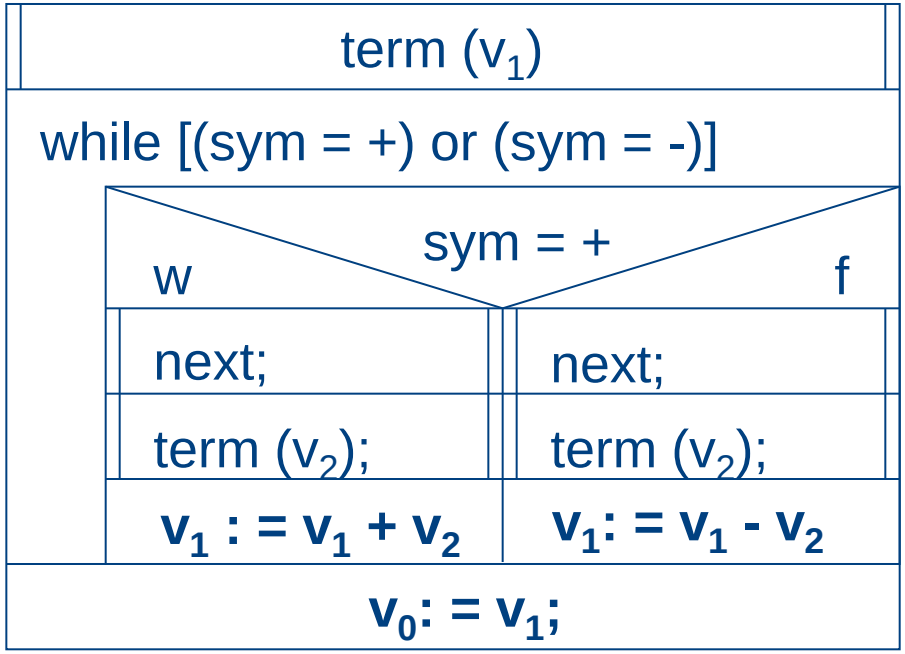
	Attribute	Attributregeln (Semantikroutinen)	
Exp $[v_0] \rightarrow$	Term $[v_1]$	<< *sem:* $v_0 := v_1$ >>	|
	Exp $[v_1]$ + Term $[v_2]$	<< *sem:* $v_0 := v_1 + v_2$ >>	|
	Exp $[v_1]$ – Term $[v_2]$	<< *sem:* $v_0 := v_1 - v_2$ >>	.
Term $[v_0] \rightarrow$	Factor $[v_1]$	<< *sem:* $v_0 := v_1$ >>	|
	Term $[v_1]$ * Factor $[v_2]$	<< *sem:* $v_0 := v_1 * v_2$ >>	|
	Term $[v_1]$ / Factor $[v_2]$	<< *sem:* $v_0 := v_1 / v_2$ >>	.
Factor $[v_0] \rightarrow$	number $[v_1]$	<< *sem:* $v_0 := v_1$ >>	|
	(Exp $[v_1]$)	<< *sem:* $v_0 := v_1$ >>	.

Problem: Die vorgestellte Grammatik ist nicht vom Typ LL(1).

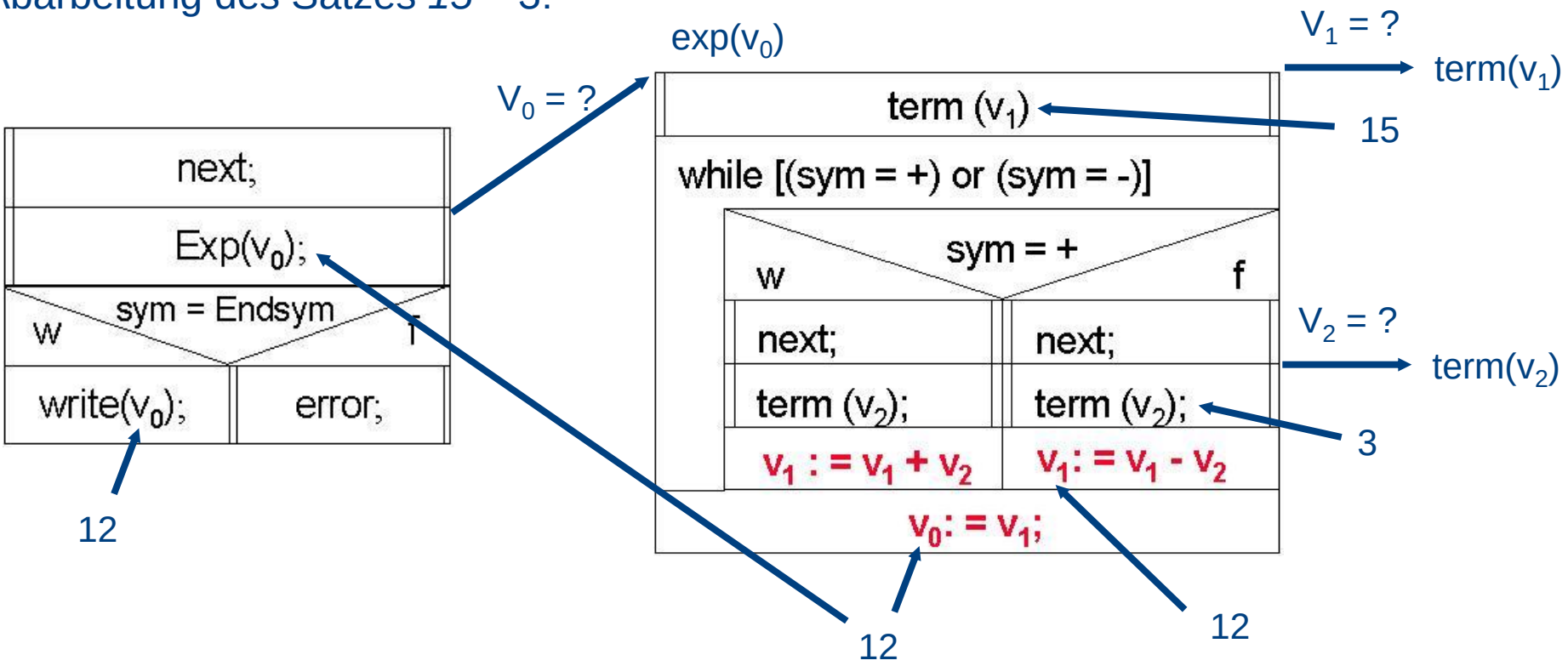
Äquivalente LL(1)-Grammatik:

Exp $[v_0] \rightarrow$	Term $[v_1]$	
	{ + Term $[v_2]$	<< *sem:* $v_1 := v_1 + v_2$ >> |
	– Term $[v_2]$	<< *sem:* $v_1 := v_1 - v_2$ >> }
		<< *sem:* $v_0 := v_1$ >>.
Term $[v_0] \rightarrow$	Factor $[v_1]$	
	{ * Factor $[v_2]$	<< *sem:* $v_1 := v_1 * v_2$ >> |
	/ Factor $[v_2]$	<< *sem:* $v_1 := v_1 / v_2$ >> }
		<< *sem:* $v_0 := v_1$ >>.
Factor $[v_0] \rightarrow$	number $[v_1]$	<< *sem:* $v_0 := v_1$ >> |
	(Exp $[v_1]$)	<< *sem:* $v_0 := v_1$ >>.

Umsetzung der Produktion für **Exp (v_0)**:



Abarbeitung des Satzes *15 – 3*:



Übersetzung geklammerter Infix-Notation in klammerfreie Postfix-Notation:

Exp → Term
{<< *sem: op := sym* >> (+ | -) Term
<< *sem: write (op)* >> }.

Term → Factor
{<< *sem: op := sym* >> (* | /) Factor
<< *sem: write (op)* >> }.

Factor → number
<< *sem: write (number)* >> |
(Exp) .

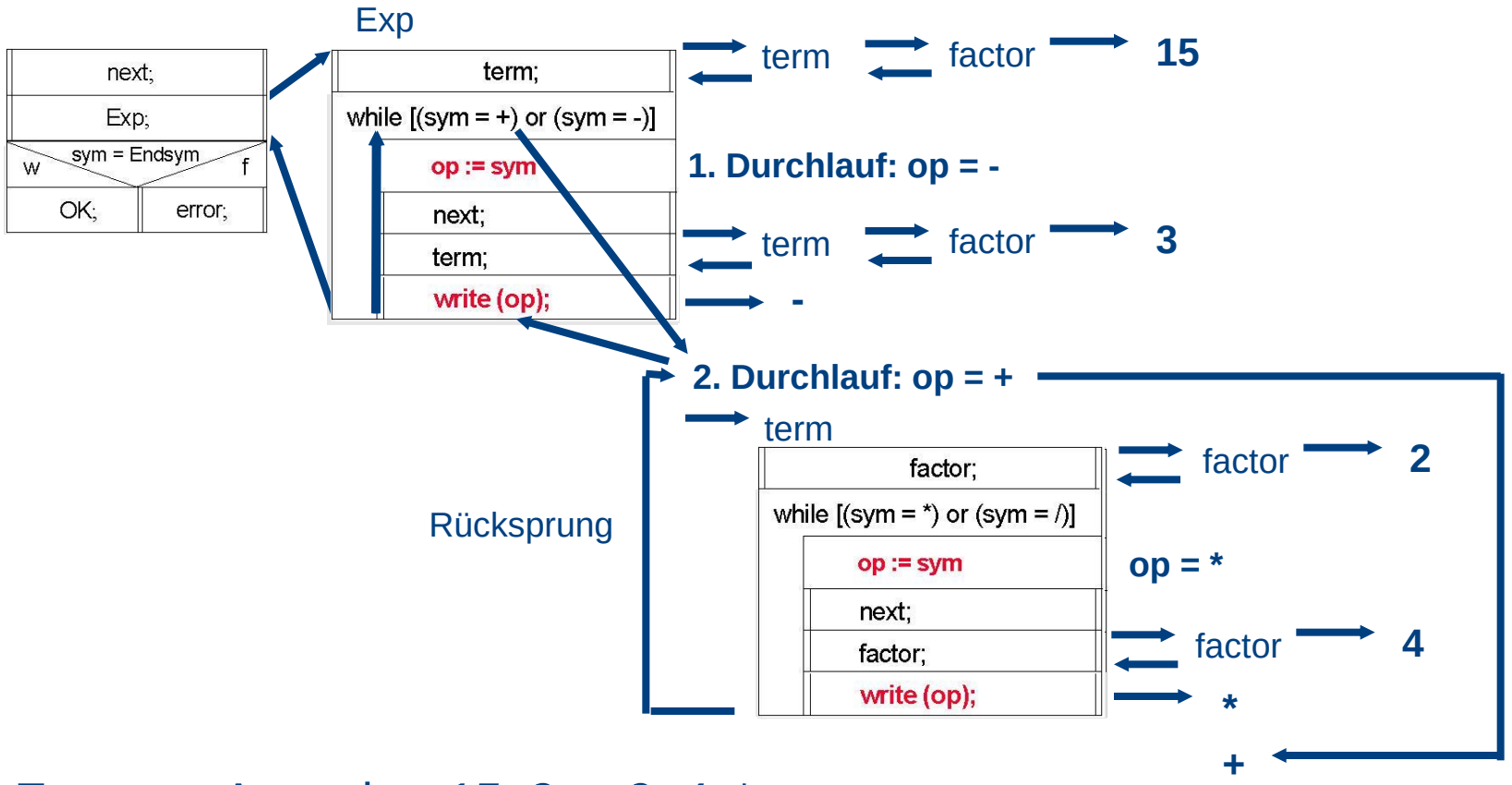
Attributierte Grammatiken: Codegenerierung

- Umsetzung der Produktion für Exp:

```
term;
while [(sym = +) or (sym = -)]
    op := sym
    next;
    term;
    write (op);
```

- **Vollständiger Übersetzer von Infix- nach Postfix-Notation:**
- Steuerung durch den Parser:
  - Aufruf des Scanners, um das nächste Symbol zu erhalten
  - Aufruf der Codegenerierung, um den bis dahin erzeugbaren Zielcode zu generieren

Abarbeitung des Satzes *15 – 3 + 2 * 4*:



Erzeugte Ausgabe: *15 3 - 2 4 * +*

- Parser
  - Die vorgestellte Technik des rekursiven Abstiegs geht von dem Startsymbol aus – also Top-Down von oben nach unten. Der Satz wird von links (L) nach rechts gelesen und es wird das jeweils am weitesten links (L) stehende Symbol expandiert (weiter abgeleitet), wobei stets genau 1 Symbol (1) vorausgeschaut wird => LL(1)-Parser:
    - Man kann mehr als 1 Symbol vorausschauen: **LL(k)-Parser**
    - Man muß nicht notwendig vom Startsymbol ausgehen, sondern kann auch versuchen, ausgehend vom terminalen Satz das Startsymbol durch Reduzieren zu erzeugen (Bottom-Up-Parser). Es wird dann stets vom rechten Ende (R) des Satzes ausgehend reduziert: **LR(x)-Parser**
      - (spezielle Varianten: SLR: Simple LR; LALR: Look Ahead LR)

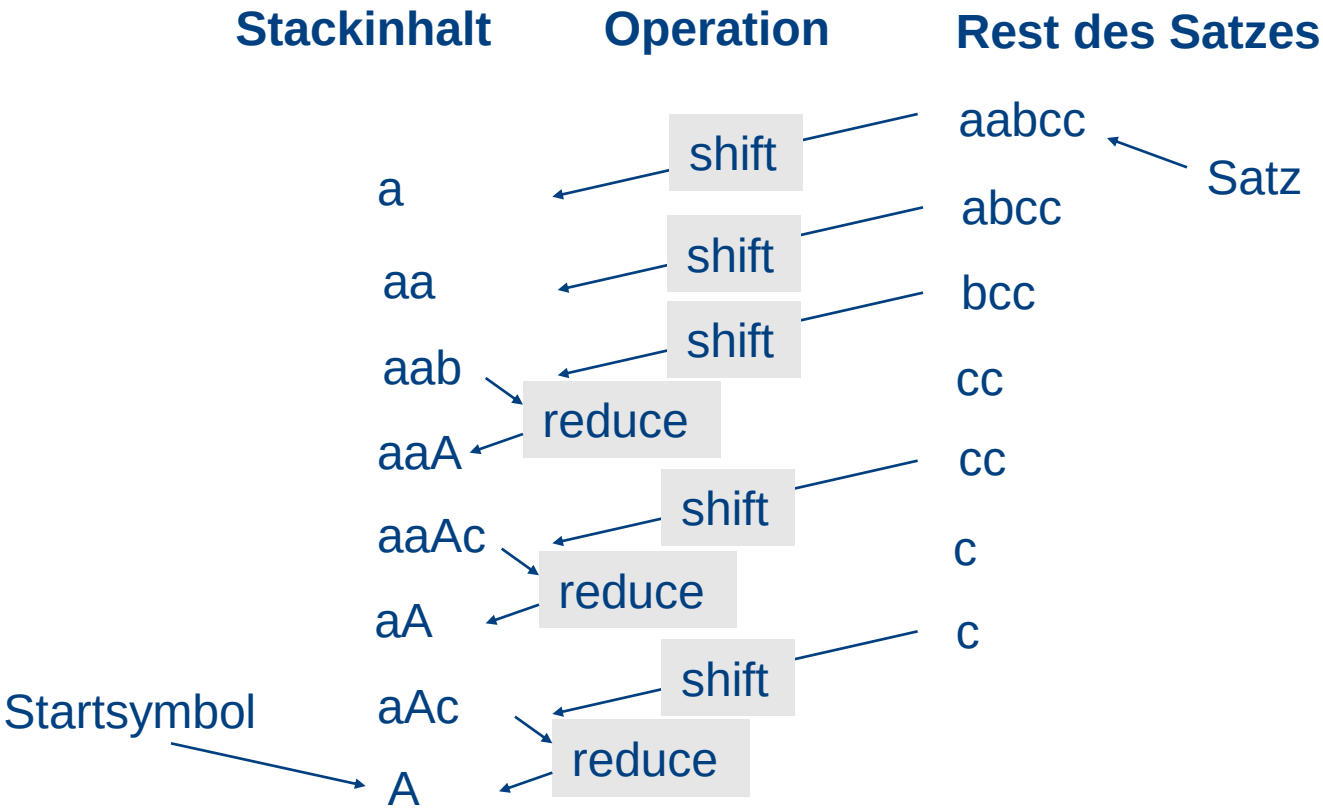
- Beispiel: LR-Parser
  - LR-Parser arbeiten tabellengesteuert, d.h. eine Tabelle gibt an
    - welche Aktion durchzuführen ist
    - und was der nächste Schritt ist
  - Der Satz wird von links gelesen.
  - Die Symbole werden in einem Stapel (*Stack*) gespeichert (Operation *shift*) bis die rechte Seite einer Produktionsregel auf eine im *Stack* oben befindliche Symbolfolge angewendet werden kann. Diese Symbolfolge wird vom *Stack* entfernt und gegen die linke Seite der Produktionsregel ersetzt (Operation *reduce*). Die Symbolfolge im *Stack* wird also von rechts reduziert.
  - Wenn nicht eindeutig erkannt werden kann, ob eine *shift*- oder eine *reduce*-Operation durchgeführt werden muß, so spricht man von einem *shift-reduce*-Konflikt
  - Wenn nicht eindeutig erkannt werden kann, welche *reduce*-Operation durchgeführt werden muß, so spricht man von einem *reduce-reduce*-Konflikt

Beispiel: LR-Parser

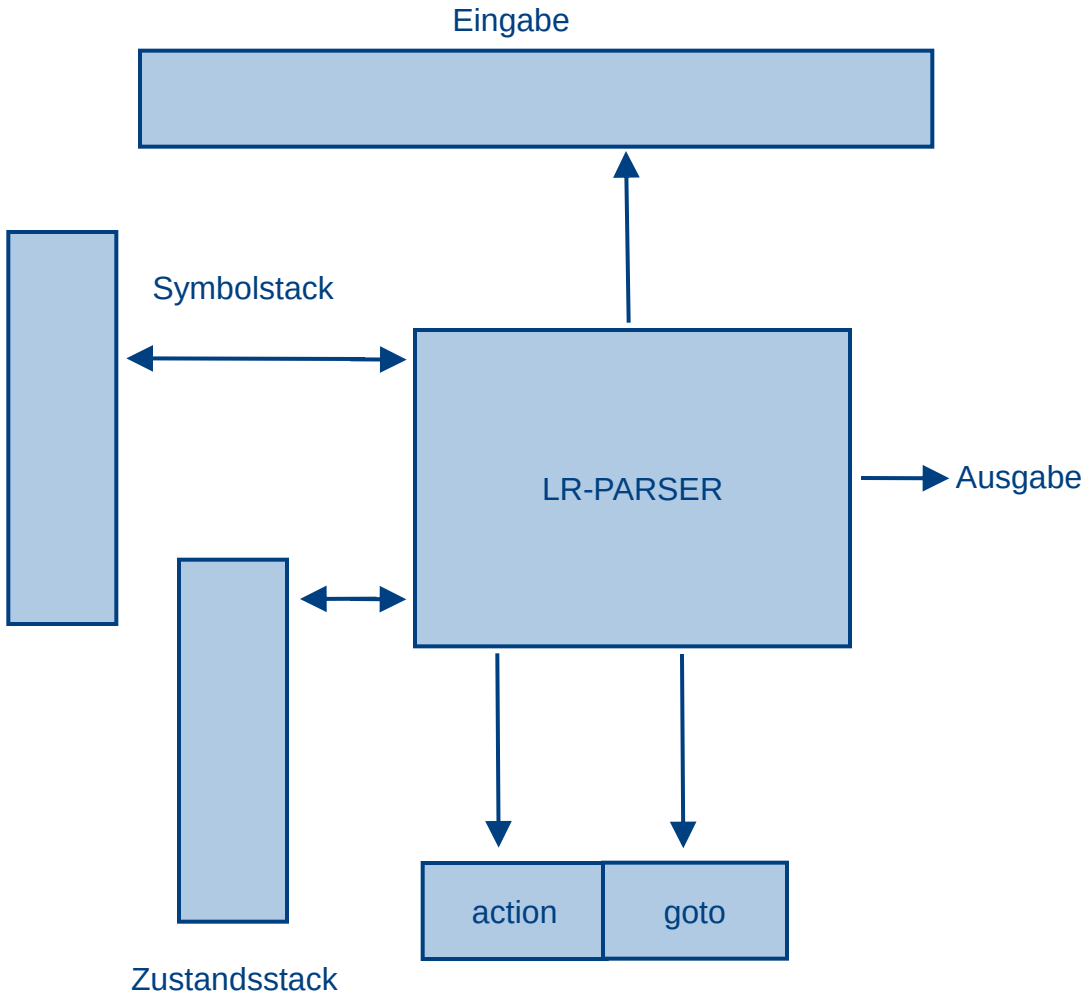
G=(N, T, P, A); N = { A }; T = { a, b, c }; P = { A → a A c | b }

Analyse des Satzes:

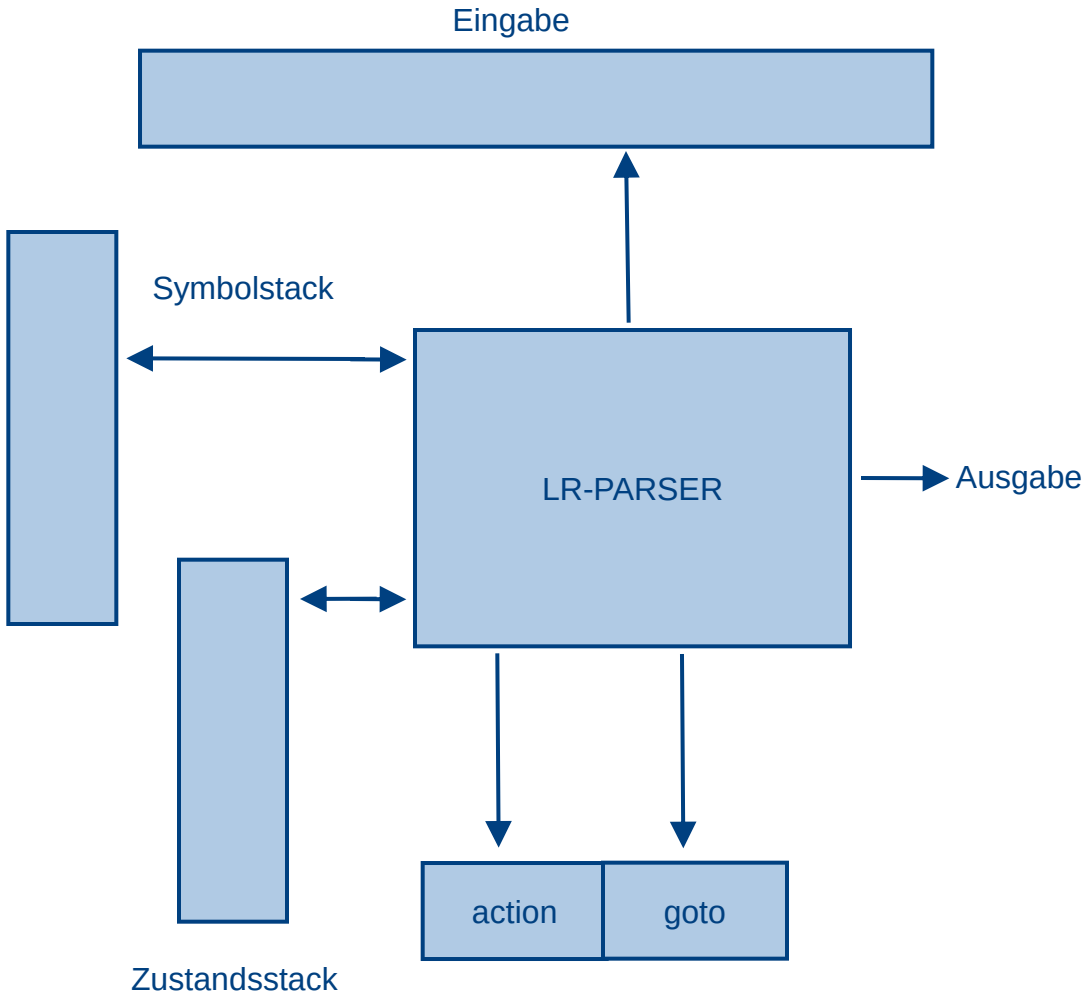
aabcc



- Anfangszustand *s0* wird auf den Zustandsstack geschoben, der Symbolstack ist leer.
- Der nächste Schritt des Parsers bestimmt sich aus dem lookahead-Symbol *l* und dem aktuellen Zustand *s* durch den Aktionstabelleneintrag *action(l, s)*



- Fall 1: *action(s, l)=shift s1*: Parser schiebt *l* in einer SHIFT-Aktion auf den Symbolstack, wechselt in Zustand *s1* und schiebt *s1* auf den Zustandsstack.
- Fall 2: *action(s, l)=reduce A* $\rightarrow$ B:
  - Sei *n* die Anzahl der Symbole auf der rechten Regelseite *B*, d.h. die Länge des auf dem Symbolstack liegenden zu reduzieren Satzteils
  - Sowohl vom Zustandsstack als auch vom Symbolstack werden *n* Elemente entfernt. *A* wird auf den Symbolstack geschoben. Der neue auf den Zustandsstack zu schiebende Zustand ergibt sich aus dem jetzt oben liegenden Zustand *s1* aus der goto-Tabelle: *goto(s1, A)*



- LL-Parser können das Prinzip des rekursiven Abstiegs verwenden oder tabellengesteuert arbeiten
- LR-Parser arbeiten stets tabellengesteuert
- Vorteile und Nachteile des rekursiven Abstiegs:
  - Einfach aus der Syntax herleitbar, Direkte Umsetzung des Rekursionen der Grammatik in rekursive Aufrufe
  - Unflexibel
- Vorteile und Nachteile der tabellengesteuerten Syntaxanalyse
  - Komplizierter
  - Flexibel: Bei einer neuen Grammatik bleibt der Code des Parsers unverändert; nur die Tabellen ändern sich.
  - Parsertabellen direkt aus der Grammatik erzeugbar

=> Es ist möglich, Compilerteile (Scanner, Parser) automatisch zu erzeugen:

=> Compiler-Compiler

Name	Quelle / Status	Grammatik	Sprache
JavaCC	SUN Microsystems Freeware	LL(1), falls erforderlich LL(k)	Java
Jaccie	Universität der Bundeswehr, München	LL(1), LR(0), SLR(0), LALR(1), LR(1)	Java
PCCTS bzw. ANTRL	Purdue University MageLang Institute	Sogen. pred.-LL(k)	C/C++, Java
lex / yacc (auch PClex / PCyacc, Aflex und Ayacc für Ada)	Unix	LALR(1)	C
Eli	Uni Paderborn, GNU Public License	LALR(1)	C

- Erstellung einer Eingabedatei (*.jj), die die lexikalische und syntaktische Struktur beschreibt und in Form von Javaroutinen semantische Schritte enthält
- Übersetzung der Eingabedatei in Java-Code (*.java)
- Übersetzung des Java-Codes in eine ausführbare Java-Klasse (*.class)
- Ausführung der Klasse

- Beispiel
- Umwandlung eines arithmetischen Integerausdrucks in geklammerter Infix-Notation in klammerfreie Postfix-Notation incl. Auswertung des Ausdrucks

Compiler-Compiler
JavaCC: Eingabedatei

```
PARSER_BEGIN(upn)
public class upn
   { public static void main (String args [])
                { upn parser = new upn(System.in);
                for (;;)
                try {
                                if (parser.exp() == -1)
                                System.exit(0);
                }
                catch (Exception e)
                {e.printStackTrace(); System.exit(1);}
                }
   }
PARSER_END(upn)

SKIP:    // defines input to be ignored
{ " " | "\r" | "\t"}

TOKEN:    // defines token names
{ < EOL: "\n" >
| < CONSTANT: ( <DIGIT> )+ > // re: 1 or more
| < #DIGIT: ["0" - "9"] > // private re
}
```

- Eingabedatei für die Auswertung arithmetischer Ausdrücke (Integer) und Ausgabe in Postfix-Notation
- Initiale Festlegungen

Compiler-Compiler
JavaCC: Eingabedatei

```
int exp() throws Exception: // exp: expr \n
{ int e; }   // prints result; -1 at eof, 0 at eol/error
{ try {
    ( e = expr() <EOL> { System.out.println("\t"+e); return 1; }
    | <EOL>  { return 0; }
    | <EOF>  { return -1; }
    )
    } catch (Exception err) {
    if (err instanceof ParseException || err instanceof
    ArithmeticException
    || err instanceof NumberFormatException)
                { System.err.println(err);
                  for (;;)
                              switch (getNextToken().kind)
                                          {case EOF: return -
    1;
                                           case EOL: return 0;
                                          }
                }
    throw err;
    }
}
```

- Startsymbol

Ergebnisausgabe

Start

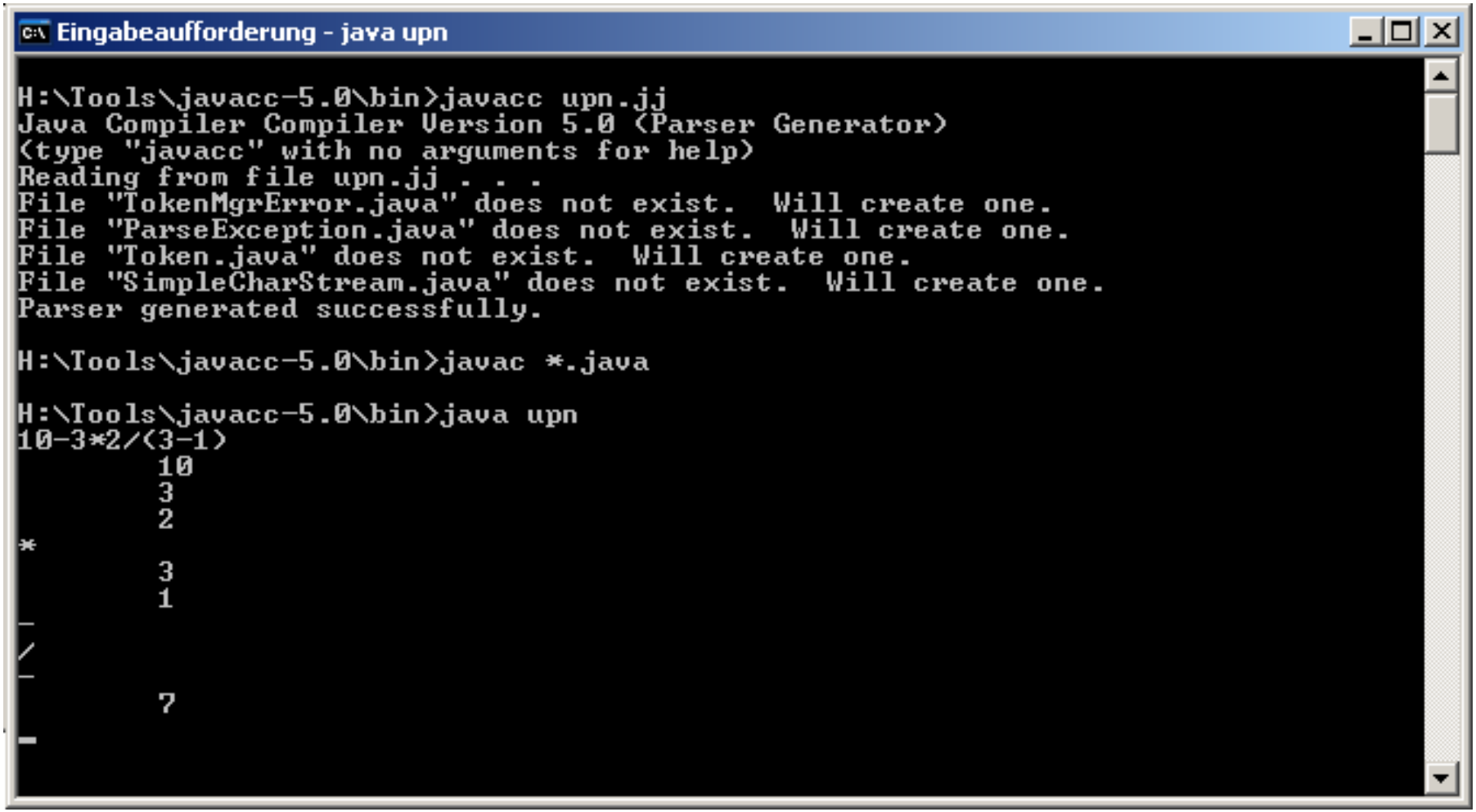
Compiler-Compiler
JavaCC: Eingabedatei

```
int expr() throws NumberFormatException:  // expr: term { +
    term | - term }
{ int s, r; }     // returns value
{ s = term()
    ( "+" r = term() { s += r; System.out.println("+"); }
    | "-" r = term() { s -= r; System.out.println("-"); }
    )*   { return s; }
}

int term() throws NumberFormatException: // term: factor { *
    factor | / factor }
{ int p, r;}     // returns value
{ p = factor()
    ( "*" r = factor() { p *= r; System.out.println("*"); }
    | "/" r = factor() { p /= r; System.out.println("/"); }
    )*   { return p; }
}

int factor() throws NumberFormatException: // factor: (expr) |
    number
{ int t;}   // returns value
{   "(" t = expr() ")" { return t; }
    | <CONSTANT>  { t = Integer.parseInt(token.image);
    System.out.println("\t"+t); return t; }
}
```

- Produktionsregeln

Semantikroutine

JavaCC: Erzeugung des Java-Codes aus der Eingabedatei

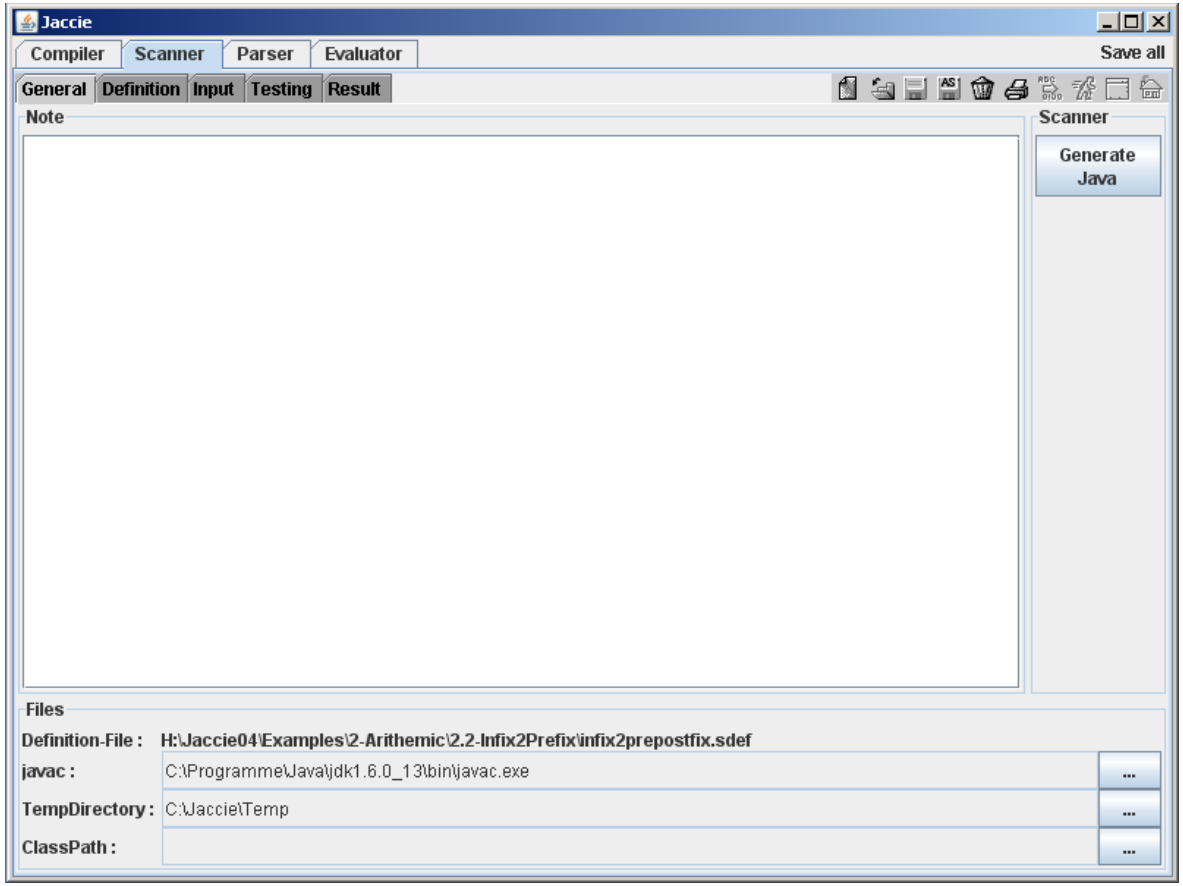


Compiler-Compiler
JavaCC: Hinweise zur Benutzung

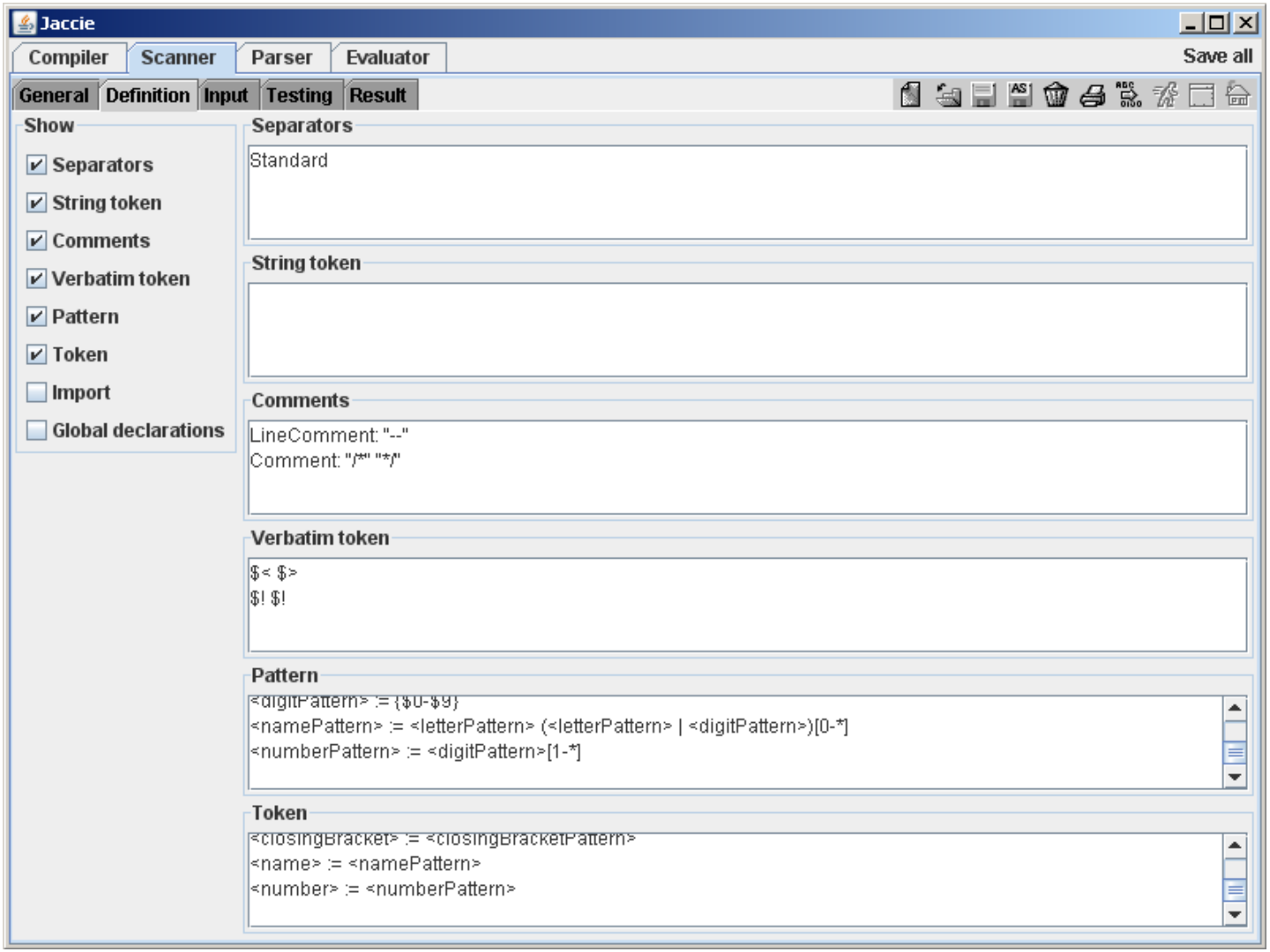
- Die JavaCC-Eingabedatei heißt *upn.jj*
- Übersetzen Sie die Datei mit JavaCC in Java-Code
- Übersetzen Sie den Java-Code: *javac *.java*
- Starten Sie die Klasse: *java upn*

- Versuchen Sie die JavaCC-Eingabedatei so zu erweitern, daß Vorzeichenbehaftete Integerwerte verarbeitet werden können. Es ist nur eine sehr kleine Änderung erforderlich (Die Eingabedatei heißt *upnx.jj*)!

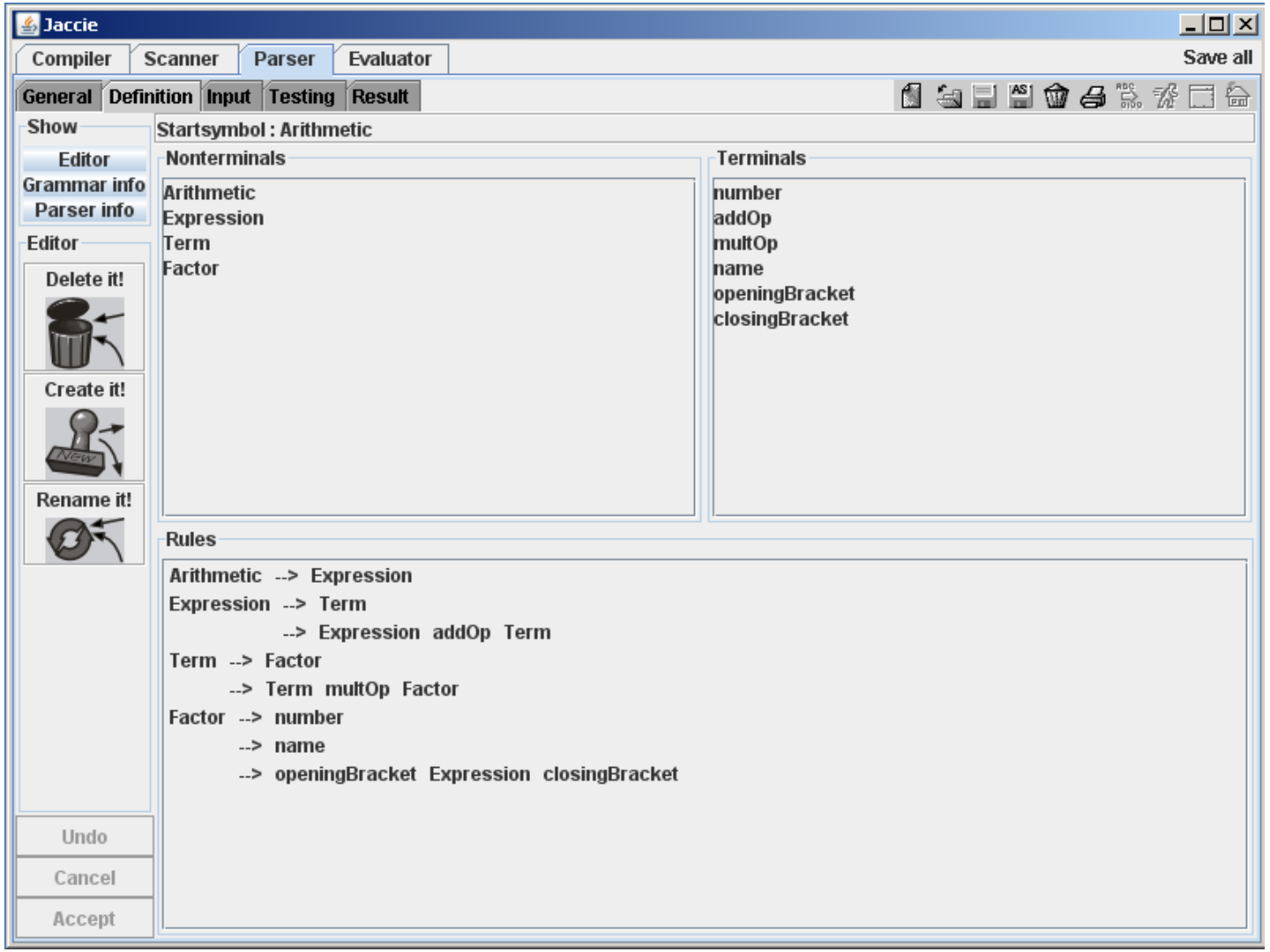
- Interaktives Compilerbauwerkzeug
- Start: *java –jar Jaccie04.jar*



Compiler-Compiler
Jaccie: Scanner



Compiler-Compiler
Jaccie: Parser



Compiler-Compiler
Jaccie: Interaktive Analyse des Ausdrucks: 10-3*2/(3-1)