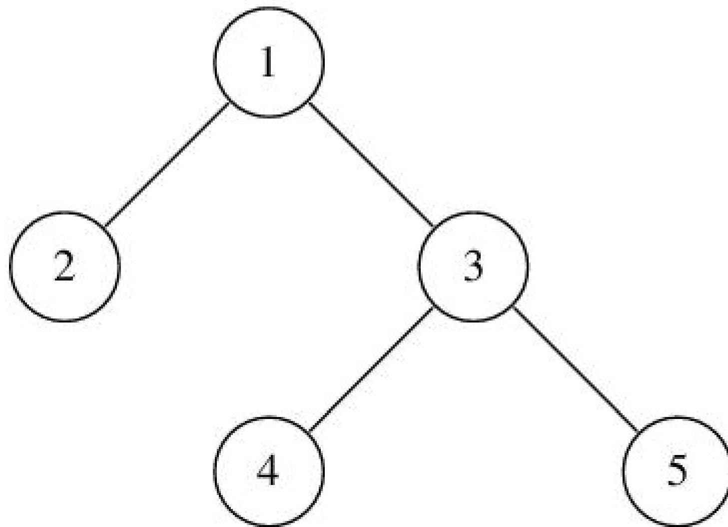
Syntaxanalyse (Parsing)

- Eingabe: Tokensequenz
- Ausgabe: Parsebaum (in der Regel)
  - innere Knoten: Nichtterminale
  - Blätter: Terminale (von links nach rechts: Eingabe)
- analysierbare Grammatiken:
  - kontextfrei (linke Seite: ein Nichtterminal)
  - eindeutig (nur ein Parsebaum pro Eingabe)
- Erkenner: Stackautomaten
  - Rekursion (Laufzeitstack)
  - Tabellen-gesteuert (separater Stack)

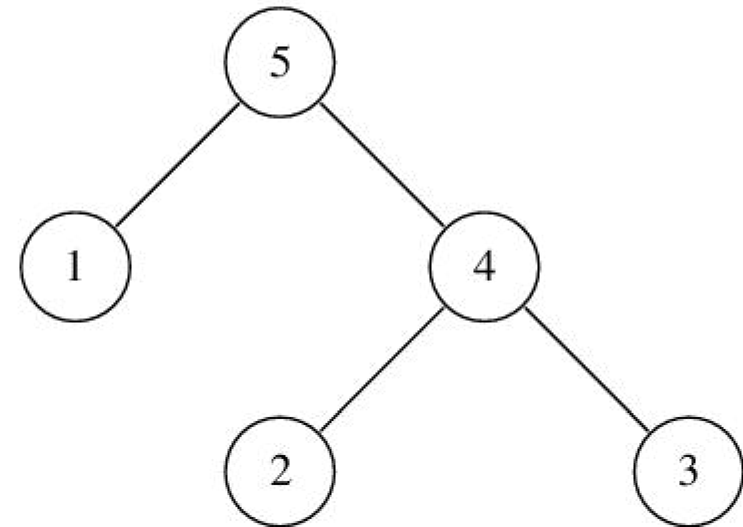
Zwei Strategien

	LL(k)	LR(k)
Eingabe lesen	links nach rechts (L_)	links nach rechts (L_)
Produktion generieren	links nach rechts (_L)	rechts nach links (_R)
Vorausschau	k Token	k Token
Zeitkomplexität	linear	linear
linksrekursive Produktionen	nicht möglich	möglich und effizient
Baumrekonstruktion	top-down	bottom-up
Bild	Abb. 2.51	Abb. 2.52

Preorder (LL), Postorder (LR)



(a) Pre-order



(b) Post-order

Figure 2.50 A tree with its nodes numbered in pre-order and post-order.

Rekursiver Abstieg bei LL

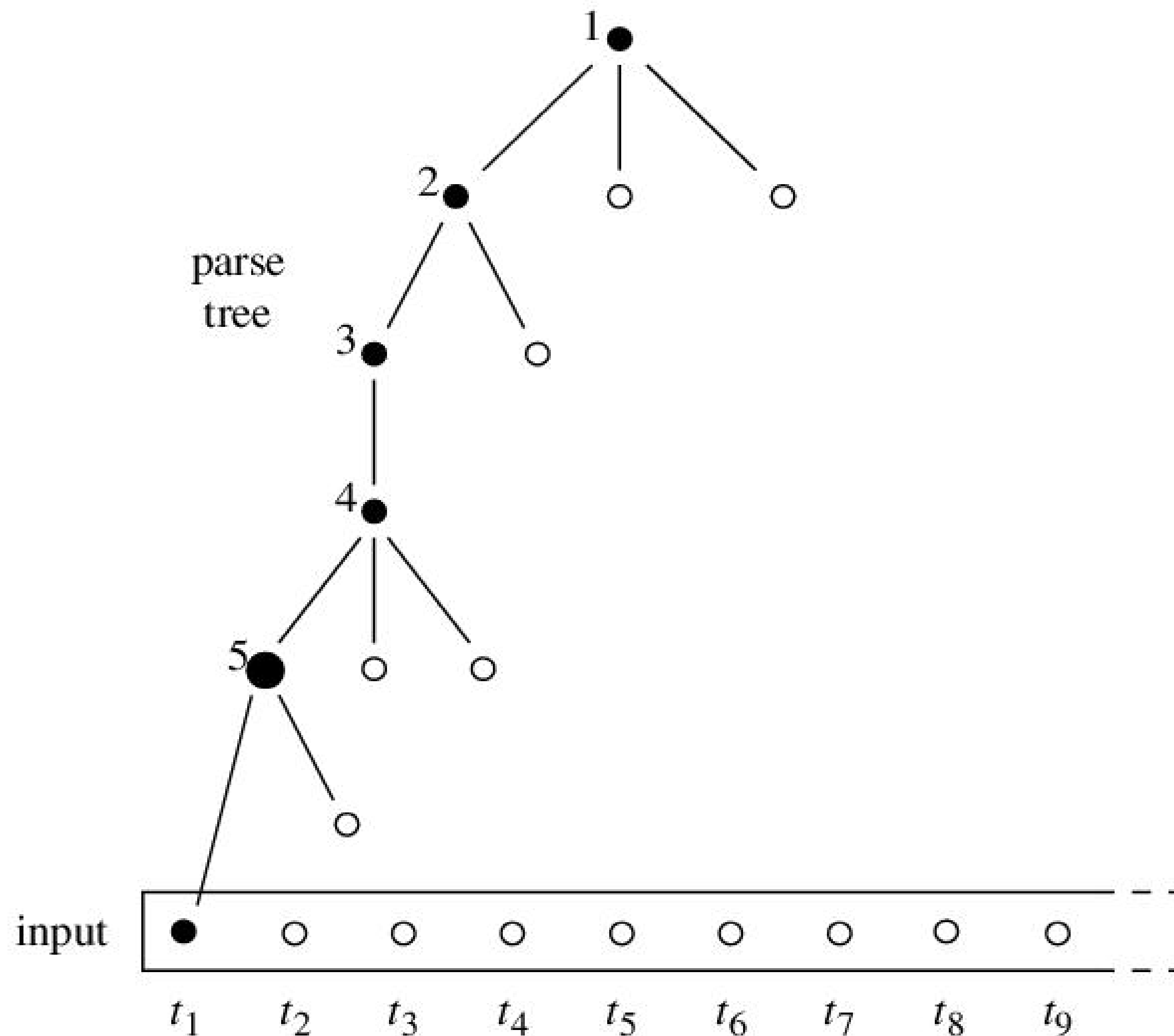


Figure 2.51 A top-down parser recognizing the first token in the input.

Bottom-Up Konstruktion bei LR

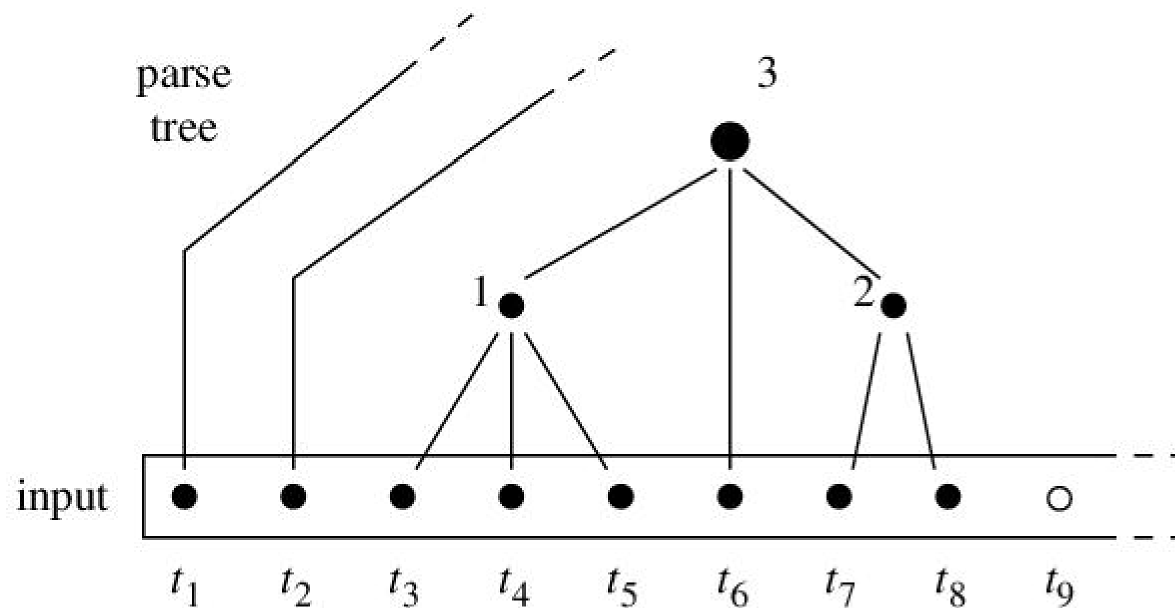


Figure 2.52 A bottom-up parser constructing its first, second, and third nodes.

LL(1) mit rekursivem Abstieg

- Beispielgrammatik:

```
input                       → expression EOF
expression                  → term rest_expression
term                        → IDENTIFIER
                            | parenthesized_expression
parenthesized_expression    → '(' expression ')'
rest_expression             → '+' expression | ε
```

- Beispiel-"Erkenner" (erzeugt keinen Parsebaum) mit rekursivem Abstieg:
  - Beispielwort:

    `IDENTIFIER + (IDENTIFIER + IDENTIFIER) EOF`

  - Implementierung (Abb. 2.54-2.55)
    - pro Nichtterminal eine Funktion
    - gdw. aus diesem Nichtterminal am Anfang das Eingabesymbol erzeugt werden kann (Produktion muss eindeutig sein)
      (a) wird das Eingabetoken verbraucht
      (b) gibt die Funktion den Wert true (≠0 in C) zurück

```
int input(void) {
    return expression() && require(token(EOF));
}

int expression(void) {
    return term() && require(rest_expression());
}

int term(void) {
    return token(IDENTIFIER) || parenthesized_expression();
}

int parenthesized_expression(void) {
    return token('(') && require(expression()) && require(token(')'));
}

int rest_expression(void) {
    return token('+') && require(expression()) || 1;
}

int token(int tk) {
    if (tk != Token.class) return 0;
    get_next_token(); return 1;
}

int require(int found) {
    if (!found) error();
    return 1;
}
```

Figure 2.54 A recursive descent parser/recognizer for the grammar of Figure 2.53.

```
int main(void) {
    start_lex(); get_next_token();
    require(input());
    return 0;
}

void error(void) {
    printf("Error in expression\n"); exit(1);
}
```

Figure 2.55 Driver for the recursive descent parser/recognizer.

Probleme beim LL(1)-Parsing

... der Parser ist nicht immer korrekt ...

- Ausdruckssprache mit Arrays:

```
term            → IDENTIFIER | Indexed_element
                | parenthesized_expression
indexed_element → IDENTIFIER '[' expression ']'
```

nicht links-faktorisiert, Auswahl der Produktion unklar

- Sprache {ab, aab}:

```
S → A 'a' 'b'
A → 'a' | ε
```

Auswahl auch unklar

Lösung des Problems: Substitution von A und Links-Faktorisierung:

```
S → 'a' R
R → 'a' 'b' | 'b'
```

jetzt ist das zweite **'a'** optional, nicht das erste

- Ausdruckssprache mit links-assoziativer Subtraktion

```
expression → expression '-' term
```

links-rekursiv, Parser gerät in Endlosschleife

Konflikte bei der Auswahl

```
S → A 'a' 'b'
A → 'a' | ε
```

Figure 2.56 A simple grammar with a FIRST/FOLLOW conflict.

```
int S(void) {
    return A() && require(token('a')) && require(token('b'));
}

int A(void) {
    return token('a') || 1;    /* erkennt nicht "ab"  */
           1 || token('a');    /* erkennt nicht "aab" */
```

Figure 2.57 A faulty recursive recognizer for grammar of Figure 2.56.

LL(1)-Parsergenerierung: FIRST-Mengen

... geben an, mit welchen Zeichen eine Satzform beginnen kann ...
(falls Satzform das leere Wort herleiten kann, zusätzlich ε)

Inferenzregeln eines Hüllenalgorithmus (vgl. Abb. 2.58):

- $N \to \alpha$

  $\mathrm{FIRST}(N) \supseteq \mathrm{FIRST}(\alpha)$

  "füge alle Zeichen aus $\mathrm{FIRST}(\alpha)$ zu $\mathrm{FIRST}(N)$ hinzu"

- $\alpha = A\beta$
  - $\neg(A \rightarrowtail \varepsilon)$ "aus A kann nicht der leere String erzeugt werden"

    $\mathrm{FIRST}(\alpha) \supseteq \mathrm{FIRST}(A)$

  - $A \rightarrowtail \varepsilon$

    $\mathrm{FIRST}(\alpha) \supseteq (\mathrm{FIRST}(A) \setminus \{\varepsilon\}) \cup \mathrm{FIRST}(\beta)$

Data definitions:

1. Token sets called FIRST sets for all terminals, non-terminals and alternatives of non-terminals in G.
2. A token set called FIRST for each alternative tail in G; an alternative tail is a sequence of zero or more grammar symbols α if $A\alpha$ is an alternative or alternative tail in G.

Initializations:

1. For all terminals T, set FIRST(T) to $\{T\}$.
2. For all non-terminals N, set FIRST(N) to the empty set.
3. For all non-empty alternatives and alternative tails α, set FIRST(α) to the empty set.
4. Set the FIRST set of all empty alternatives and alternative tails to $\{\varepsilon\}$.

Inference rules:

1. For each rule $N \rightarrow \alpha$ in G, FIRST(N) must contain all tokens in FIRST(α), including ε if FIRST(α) contains it.
2. For each alternative or alternative tail α of the form $A\beta$, FIRST(α) must contain all tokens in FIRST(A), excluding ε, should FIRST(A) contain it.
3. For each alternative or alternative tail α of the form $A\beta$ and FIRST(A) contains ε, FIRST(α) must contain all tokens in FIRST(β), including ε if FIRST(β) contains it.

Figure 2.58 Closure algorithm for computing the FIRST sets in a grammar G.

Rule/alternative (tail)	FIRST set
`input`	`{ }`
`    expression EOF`	`{ }`
`        EOF`	`{ EOF }`
`expression`	`{ }`
`    term rest_expression`	`{ }`
`        rest_expression`	`{ }`
`term`	`{ }`
`    IDENTIFIER`	`{ IDENTIFIER }`
`|`	
`    parenthesized_expression`	`{ }`
`parenthesized_expression`	`{ }`
`    '(' expression ')'`	`{ '(' }`
`        expression ')'`	`{ }`
`            ')'`	`{ ')' }`
`rest_expression`	`{ }`
`    '+' expression`	`{ '+' }`
`        expression`	`{ }`
`|    ε`	`{ ε }`

Figure 2.59 The initial FIRST sets.

Rule/alternative (tail)	FIRST set
`input`	{ IDENTIFIER '(' }
`    expression EOF`	{ IDENTIFIER '(' }
`        EOF`	{ EOF }
`expression`	{ IDENTIFIER '(' }
`    term rest_expression`	{ IDENTIFIER '(' }
`        rest_expression`	{ '+' ε }
`term`	{ IDENTIFIER '(' }
`    IDENTIFIER`	{ IDENTIFIER }
`|`	
`    parenthesized_expression`	{ '(' }
`parenthesized_expression`	{ '(' }
`    '(' expression ')'`	{ '(' }
`        expression ')'`	{ IDENTIFIER '(' }
`            ')'`	{ ')' }
`rest_expression`	{ '+' ε }
`    '+' expression`	{ '+' }
`        expression`	{ IDENTIFIER '(' }
`|   ε`	{ ε }

Figure 2.60 The final FIRST sets.

LL(1)-Parsergenerierung: FOLLOW-Mengen

... geben an, welche Zeichen auf ein Nichtterminal folgen können ...

Inferenzregeln eines Hüllenalgorithmus (vgl. Abb. 2.62):

- $M \rightarrow \alpha N \beta$
  - $\neg\left(\beta \rightarrowtail \varepsilon\right)$

    $\mathrm{FOLLOW}(N) \supseteq \mathrm{FIRST}(\beta)$
  - $\beta \rightarrowtail \varepsilon$ (beinhaltet, dass β nicht existent ist)

    $\mathrm{FOLLOW}(N) \supseteq (\mathrm{FIRST}(\beta) \setminus \{\varepsilon\}) \cup \mathrm{FOLLOW}(M)$

LL(1)-Parsergenerierung: FOLLOW-Mengen

Data definitions:

1. Token sets called FOLLOW sets for all non-terminals in G.
2. Token sets called FIRST sets for all alternatives and alternative tails in G.

Initializations:

1. For all non-terminals N, set FOLLOW(N) to the empty set.
2. Set all FIRST sets to the values determined by the algorithm for FIRST sets.

Inference rules:

1. For each rule of the form $M \rightarrow \alpha N \beta$ in G, FOLLOW(N) must contain all tokens in FIRST(β), excluding ε, should FIRST(β) contain it.
2. For each rule of the form $M \rightarrow \alpha N \beta$ in G where FIRST(β) contains ε, FOLLOW(N) must contain all tokens in FOLLOW(M).

Figure 2.62 Closure algorithm for computing the FOLLOW sets in grammar G.

Beispiel: FIRST/FOLLOW-Konstruktion

	FIRST	FOLLOW
input → exp EOF	{ ID, '(' }	{ }
exp → term rest_exp	{ ID, '(' }	{ EOF, ')' }
term → ID | par_exp	{ ID, '(' }	{ '+', EOF, ')' }
par_exp → '(' exp ')'	{ '(' }	{ '+', EOF, ')' }
rest_exp → '+' exp | ε	{ '+', ε }	{ EOF, ')' }

Prädiktiver LL(1)-Parser

zusätzliche Herausforderungen:

- Behandlung des Kleene-Sterns

  (verkompliziert FIRST/FOLLOW-Berechnungen)
- aussagekräftige Fehlermeldungen
- Erzeugung des Syntaxbaums
- Behandlung von Syntaxfehlern
- Optimierungen des Parsens

FIRST/FIRST-Konflikte

Beispielgrammatik:

```
term              → IDENTIFIER | Indexed_element
                  | parenthesized_expression
indexed_element → IDENTIFIER '[' expression ']'
```

FIRST(indexed_element) = { IDENTIFIER }

Parser:

```
void term(void) {
    switch (Token.class) {
    case IDENTIFIER: token(IDENTIFIER); break;
    case IDENTIFIER: indexed_element(); break;
    case '(':        parenthesized_expression();
                      break;
    default:         error(); }
}
```

bei Links-Rekursion: auch FIRST/FIRST-Konflikte!

FIRST/FOLLOW-Konflikte

	FIRST	FOLLOW
S → A 'a 'b'	{ 'a' }	{ }
A → 'a' | ε	{ **'a'**, ε }	{ **'a'** }

Parser:

```
void S(void) {
     switch(Token.class) {
     case 'a': A(); token('a'); token('b'); break;
     default: error(); }
}
void A(void) {
     switch(Token.class) {
     case 'a': token('a'); break;
     case 'a': break;
     default: error(); }
}
```

LL(1)-Anforderungen

... eindeutige Auswahl jeder Alternative ...

- Voraussetzung für Auswahl von Alternative $N \rightarrow A_k$ für Nichtterminal N
  - $\neg(A_k \rightarrowtail \varepsilon)$: Lookahead-Token $\in$ $\mathrm{FIRST}(A_k)$
  - $A_k \rightarrowtail \varepsilon$: Lookahead-Token $\in$ $\mathrm{FIRST}(A_k) \cup \mathrm{FOLLOW}(N)$
- Bedingungen für eine eindeutige Bestimmung von A_k:
  - keine FIRST/FIRST-Konflikte
  - keine FIRST/FOLLOW-Konflikte
  - höchstens eine Alternative A_k, für die gilt: $A_k \rightarrowtail \varepsilon$

Grammatikanpassung an LL(1)

Problem

- gegeben: Grammatik $G \notin$ LL(1)
- gesucht: LL(1)-Grammatik G', so dass $L(G')=L(G)$
- Vorteil von G': automatische Parsererstellung möglich
- Nachteil von G': Struktur des Parsebaum passt nicht zu G

Tricks und Guidelines

- Substitution von Nichtterminalen durch ihre rechten Seiten
- Linksfaktorisierung
- Elimination von Linksrekursion

Tricks und Guidelines

Beispiel 1

```
S → A 'a' 'b'
A → 'a' | ε
```

FIRST/FOLLOW-Konflikt

1. Substitution ↓

```
S → 'a' 'a' 'b'
  | 'a' 'b'
```

FIRST/FIRST-Konflikt

2. Links-faktorisierung ↓

```
S → 'a' R
R → 'a' 'b | 'b'
```

LL(1)

Beispiel 2

```
term   → ID | par_exp
            | ind_el
ind_el → ID '[' exp ']'
```

indirekter FIRST/FIRST-Konflikt

1. Substitution ↓

```
term → ID | par_exp
          | ID '[' exp ']'
```

FIRST/FIRST-Konflikt

2. Links-faktorisierung ↓

```
term → ID R | par_exp
R    → '[' exp ']' | ε
```

LL(1)

Tricks und Guidelines

3. Elimination von Linksrekursion

Beispiel:

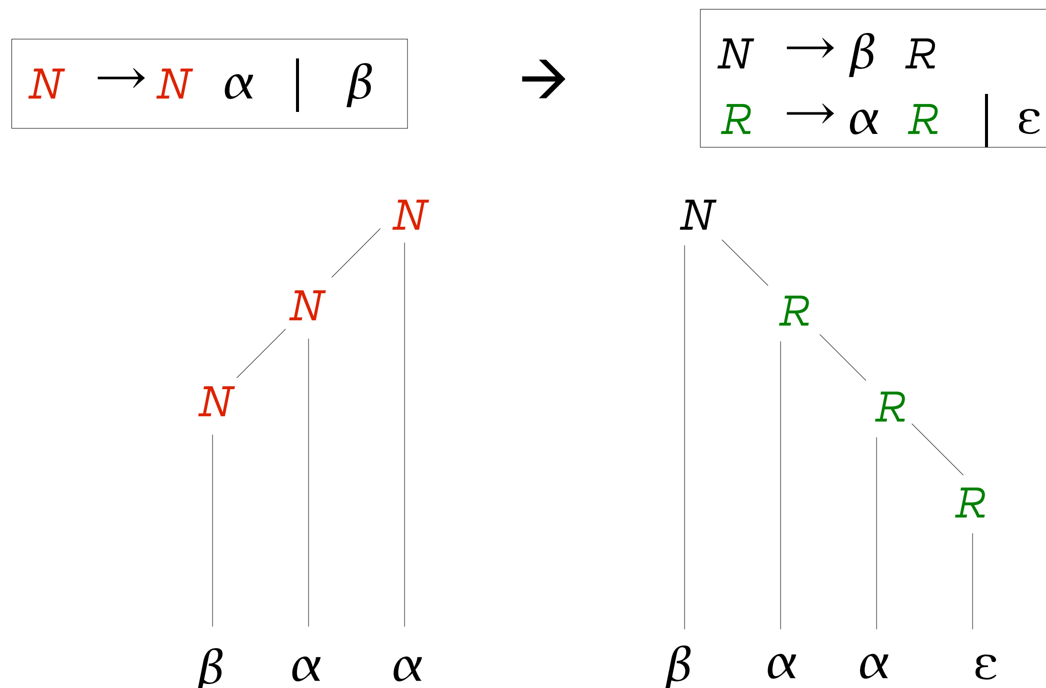
```
exp  → exp '-' term | term
```

↓

```
exp  → term rest
rest → '-' term rest | ε
```

Regel zur Elimination *direkter* Linksrekursion:

$N \to N\ \alpha \mid \beta$ → $N \to \beta\ R$; $R \to \alpha\ R \mid \varepsilon$



Tricks und Guidelines

Arten von Linksrekursion

- direkt: $N \to N \ \alpha \mid \beta$
- indirekt: $N \to M \ \alpha \mid \beta$
  $M \to N \ \gamma \mid \delta$
- versteckt: $N \to \alpha \ N \mid \beta$, wobei $\alpha \twoheadrightarrow \varepsilon$

Semantische Annotationen

... mit Markierungsregeln ...

- Frage: wie die semantischen Ergebnisse verwerten?
- Idee: füge der Grammatik Produktionen hinzu, die
  - alle nur ε generieren
  - die an der Stelle geforderten semantischen Aktionen ausführen
- Beispiel in C (Referenzen `(int *)` für Ein-/Ausgabeparameter)
  - linksrekursiv

```
exp(int *e) → exp(int *e) '-' term(int *t) {*e -= *t;}
            | term(int *t)                 {*e  = *t;}
```

  - nach Elimination von Linksrekursion

```
exp(int *e)  → term(int *t)     {*e  = *t;} rest(int *e)
rest(int *e) → '-' term(int *t) {*e -= *t;} rest(int *e)
             | ε
```