Syntaktische Analyse

(Dienstag, den 4. Mai 1999)

(Dienstag, den 11. Mai 1999)

(Dienstag, den 18. Mai 1999)

(Dienstag, den 1. Juni 1999)

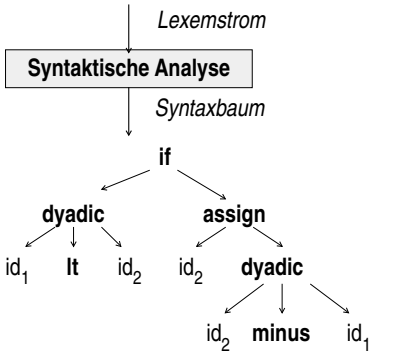
Aufgabe der Syntaxanalyse

hierarchische Analyse

- Erkennen der (kontextfreien) Syntax (*parsing*)
- Fehlerbehandlung
- Baumaufbau (oder Syntaxgesteuerte Übersetzung)

Beispiel PL0

if id_1 **lt** id_2 **then** id_2 **becomes** id_2 **minus** id_1 **;**



Spezifikation von Syntax: EBNF

erweiterte Backus-Naur-Form

Regeln der Form n ::= E,

wobei$E$ regulären Ausdrücken ähnelt:

E =	ε		leere Zeichenfolge
|	*'Z'*		*Lexem*
|	N		*Nichtterminal*
|	$E\ F$	Verkettung	(bis hier *BNF*)
|	'[' E ']'		Option
|	'{' E '}'		Wiederholung
|	'(' E ')'		Klammerung
|	E '|' F		Alternative

Beispiel

bedingte Anweisung in Modula-2

```
Statements ::= Statement { ';' Statement }
Statement ::= if    Expression
              then  Statements
             {elsif Expression
              then  Statements }
             [else  Statements ]
              end
```


Transformation in BNF

Einführen von Hilfsregeln

'[' E ']'	⇒	E_Option	::=
		E_Option	::=E
'{' E '}'	⇒	E_Sequence	::=
		E_Sequence	::= E_Sequence E
'(' E ')'	⇒	E_Group	::= E
E '|' F	⇒	E_or_F	::= E
		E_or_F	::= F

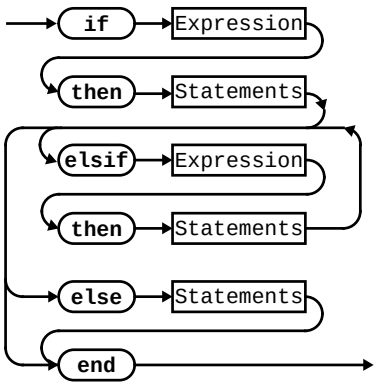
Beispiel

bedingte Anweisung in Modula-2

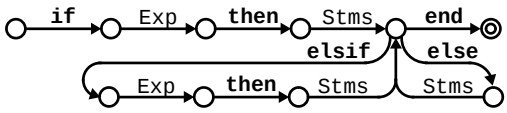
```
Statements ::= Statement
Statements ::= Statement { ';' Statement }
Statement ::=if    Expression
     then  Statements
     Elsepart
     end
Elsepart ::=
Elsepart ::= else  Statements
Elsepart ::= elsif Expression
             then  Statements
             Elsepart
```

Syntaxdiagramme

Beispiel



eine etwas andere Darstellung



Erinnerung an endliche Automaten sind *nicht* zufällig

(es gibt aber Übergänge unter Nichtterminalen)

darauf kommen wir beim absteigenden Parsieren zurück

kontextfreie Grammatiken

Sprache

Gegeben ein Vokabular V (bei uns sind das die *Lexeme*)

Wörter über V sind beliebige V-Ketten, geschrieben V^*

Sprachen über V sind beliebige Teilmengen von V^*

kontextfreie Grammatik

(für eine Sprache über V^*)

ist ein Tupel $G = (V, N, R, s)$ mit

- *Vokabular* V
- *Nichtterminalen* $N \subset V$, geschrieben A, B, C, ...
  induziert *Terminale* $T = V \setminus T$
- *Regeln* (Produktionen) $R \subset N \times V^*$, geschrieben $A \to \alpha$
- *Startsymbol* $s \in N$

Konvention: $\alpha \in V^*$, $w \in T^*$

Beispiel

kontextfreie Syntax einfacher Ausdrücke

$G = (V, N, R, s)$ mit

- $V = \{E, id, +, *, (,)\}$
- $N = \{E\}$ und $T = \{id, +, *\}$
- $R = \{E \to id, E \to E + E, E \to E * E, E \to (E)\}$
- $s = E$

Ableitungen

Ableitung

nimm ein Wort $\omega \in V^*$

ersetze ein beliebiges Nichtterminal

in einem Wort $\omega \in V^*$ gemäß den Regeln

$\omega = \beta n \gamma \Rightarrow \beta\alpha\gamma = \omega'$ wenn $n \to \alpha \in R$

wiederholtes Ableiten: $\Rightarrow^+$ bzw. $\Rightarrow^*$

Sprache einer kfG

Sprache eines Nichtterminals: $L(n) = \{ w \in T^* \mid n \Rightarrow^* w\}$

Sprache der Grammatik: $L(G) = L(s)$

eine Ableitung für $w = id * id + id \in L(G)$

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow id * id + id$

Definition (Links/Rechts-Ableitung)

$w n \gamma \Rightarrow w\alpha\gamma$ (ersetze linkes Nichtterminal)

$\beta n w \Rightarrow \beta\alpha w$ (ersetze rechtes Nichtterminal)

eine Ableitung $\omega \Rightarrow \omega'$ heißt

- *vollständig*, wenn $\omega = s$
- *terminal*, wenn $\omega' \in T^*$

Satz

Für jedes Wort einer kontextfreien Grammatik gibt es auch eine Links-Ableitung
(bzw. Rechts-Ableitung)

Ableitungsbaum

Definition

die Knoten eines Ableitungsbaum
sind mit $V \cup \{\varepsilon\}$ markiert, so daß

- innere Knoten mit N markiert sind,
- die Kinder eines n-Knoten mit $v_1 .. v_k$ markiert sind,
  wenn $n \to v_1 .. v_k \in R$ ist (wenn k=1 und $v_1 = \varepsilon$, ist $n \to \varepsilon \in R$)

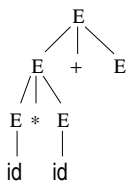
ein Ableitungsbaum heißt

- *vollständig*, wenn die Wurzel mit s markiert ist
- *terminal*, wenn alle Blätter mit $T \cup \{\varepsilon\}$ markiert sind

Beispiel

eine Ableitung ihr Ableitungsbaum

E
⇓
E +E
⇓
E * E + E
⇓
id * E + E
⇓
id * id + E



Satz

für jeden vollständigen terminalen Ableitungsbaum
gibt es *genau eine* vollständige terminale Linksableitung

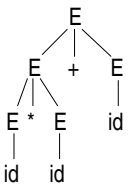
Mehrdeutigkeit

Definition

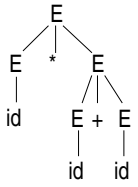
eine Grammatik ist *mehrdeutig*, wenn mindestens
ein Wort ihrer Sprache mehrere Linksableitungen hat
(bzw. mehrere Rechtsableitungen / Ableitungsbäume)

Beispiel

E
⇓
E +E
⇓
E * E + E
⇓
id * E + E
⇓
id * id + E
⇓
id * id + id



E
⇓
E * E
⇓
id * E
⇓
id * E + E
⇓
id * id + E
⇓
id * id + id



Probleme mit Mehrdeutigkeit

Mehrdeutigkeit ist semantisch unerwünscht

10 = (2 * 3) + 4 ≠ 2 * (3 + 4) = 14

Erkennung mehrdeutiger Sprachen ist aufwendiger

$O(n^3)$ statt $O(n^2)$

Mehrdeutigkeit ist nicht entscheidbar

es gibt keinen ALgorithmus, der Mehrdeutigkeit
für *beliebige* kontextfreie Grammatiken entscheidet

es gibt mehrdeutige Sprachen

manche kontextfreie Sprachen haben keine eindeutige Grammatik

Beispiel

$L = \{ a^k b^m c^n \mid k=m \text{ oder } m=n,\ k, m, n > 0 \}$

Grammatik

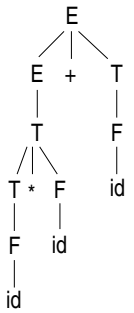
S → AB C
S → A BC
AB → ε
AB → a AB b
C → ε
C → c C
A → ε
A → a A
BC → ε
BC → b BC c

Mehrdeutigkeiten beheben

Ausdrücke

Einführen einer Hierarchie von Nichtterminalen

S → E
E → E
E → E + T
T → F
T → T * F
F → id
F → (E)



diese Grammatik berücksichtigt

- Prioritäten "* vor +"

$x * y + z = (x * y) + z$
$x + y * z = x + (y * z)$

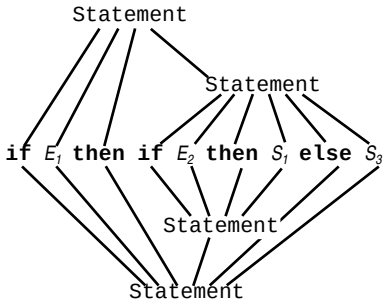
- Assoziativität (links bindet stärker)

$x * y * z = (x * y) * z$
$x + y + z = (x + y) + z$

Mehrdeutigkeit in Algol-60

dangling else

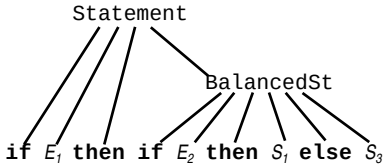
Statement ::= **if** Expression **then** Statement
| **if** Expression **then** Statement **else** Statement
| Other



Behebung

Statement ::= **if** Expression **then** Statement
| BalancedSt

BalancedSt ::=
if Expression **then** BalancedSt **else** Statement
| Other



Parsing

Erkennung

entscheide das *Wort-Problem*

$w \in L(G)$?

$rec_G : V^* \to \text{Bool}$

Parsieren (Zerteilen)

liefere auch die gefundenen Ableitungen

z.B. die Menge aller Ableitungsbäume des Wortes

$parse_G : V^* \to \mathrm{P}(T_G)$

($w \in L(G)$ genau dann wenn $parse_G(w) \neq \emptyset$)

generelle Annahme

das Wort w wird beim Parsieren von vorne nach hinten

bzw. von *rechts nach links* gelesen

dabei ist es nützlich, sich mit einem Punkt zu merken,
wie weit man bereits gelesen hat:

$w = u\,.\,v$

aufsteigende Analyse (*bottom-up*)

Aktionen des Parsers

schieben (*shift*): lies ein terminales Symbol:

$\alpha.tw \Rightarrow \alpha t.w$

reduzieren: ersetze die rechte Seite einer Regel durch die linke

$\alpha\beta.w \Rightarrow \alpha n.w$ wenn $n \to \alpha \in R$

Beispiel

```
S → E       0
E → T       1
E → E + T   2
T → F       3
T → T * F   4
F → id      5
F → ( E )   6
```

.a*b+c

id.*b+c	s
F.*b+c	r5
T.*b+c	r3
T*.b+c	s
T*id.+c	s
T*F.+c	r5
T.+c	r4
E.+c	r1
E+.c	s
E+id.	s
E+F.	r5
E+T.	r3
E.	r2
S.	r0

absteigende Analyse (*top-down*)

Aktionen des Parsers

expandieren: linke Seite einer Regel durch die rechte Seite ersetzen

$\alpha.n, w \Rightarrow \alpha.\beta, w$ wenn $n \to \beta \in R$

vergleichen (*matching*): Zeichen lesen, wenn sie
mit den expandierten rechten Seiten übereinstimmen

$\alpha.t\,\beta, tw \Rightarrow \alpha t.\beta, w$

Beispiel

```
S → E       0
E → T       1
E → E + T   2
T → F       3
T → T * F   4
F → id      5
F → ( E )   6
```

.S, a*b+c

.E, id*b+c	e0
.E+T, a*b+c	e2
.T+T, a*b+c	e1
.T*F+T, a*b+c	e4
.F*F+T, a*b+c	e3
.id*F+T, a*b+c	e5
id.*F+T, *b+c	m
id*.F+T, b+c	m
id*.id+T, b+c	e5
id*id.+T, +c	m
id*id+.T, c	m
id*id+.F, c	e3
id*id+.id, c	e5
id*id+id.,	m

(nicht-) deterministische Analyse

offen

wie "rät" man, welche Aktion des Parsers zum Erfolg führt?

Variante 1: *gar nicht* !

die Aktionen werden ausprobiert,

wenn es nicht weiter geht, werden die Alternativen ausprobiert

backtracking

unvermeidlich bei mehrdeutigen Grammatiken

ineffizient: $O(n^3)$

Variante 1: durch Vorberechnung

es werden Informationen berechnet,
die an jedem Punkt eine eindeutige Auswahl erlauben

das geht nur für (Unterklassen von) *eindeutigen* Grammatiken

die Vorberechnung ist aufwendig

das Pariseren ist aber effizient: $O(n)$

unsere Wahl für Übersetzer

deterministische absteigende Analyse

SLL(1)-Parsieren, auch bekannt als

rekursiver Abstieg (*recursive descent*)

- leichte Vorberechnung
- nicht sehr mächtig
- leicht von Hand zu implemenrtieren

Grammatikanalyse und -Transformation

Wozu?

wichtige Eigenschaften von kontextfreien Grammatiken feststellen

Vorbereiting auf das Parsieren

- Wohlgeformtheit
- hinreichende Kriterien für Eindeutigkeit
- Hilfen zur *Transformation* in "einfachere" Grammatiken
- *letztendlich*: Vorbereitung zum Parsieren

Inferenz

logische Folgerung

wenn $P_1 \ldots P_k$, *dann* Q

(aus den *Vorbedingungen* (*premises*) $P_1 \ldots P_k$ folgt das *Prädikat* Q)

Schreibweise: Implikation

$P_1 \ldots P_k, \Rightarrow Q$

logische Programmierung (Prolog)

Q :- P_1, ..., P_k.

Q heißt *Ziel* (*goal*)

wenn es keine Vorbedingung gibt,
definiert Q . ein *Faktum*

Inferenzregeln

$$\frac{P_1 \wedge \ldots \wedge P_k}{Q}$$

eine Notation von logischen Regeln, die z.B. bei der Typinferenz geläufig ist
(siehe *Kontextanalyse*)

Reduzieren von Grammatiken

erreichte Symbole: Gilt $s \Rightarrow^* \alpha n \beta$?

Regeln

$$\frac{true}{reach(s))} \quad \frac{reach(n) \wedge n \to v_1 \ldots v_k \in R}{reach(v_1) \wedge \ldots \wedge reach(v_k)}$$

produktive Symbole: Gilt $n \Rightarrow^* w$? (oder $L(n) = \varnothing$?)

Regeln

$$\frac{t \in T}{prod(t)} \quad \frac{(prod(v_1) \wedge \ldots \wedge prod(v_k)) \wedge n \to v_1 \ldots v_k \in R}{prod(n)}$$

nützliche Symbole: Gilt$s \Rightarrow^* \alpha n \beta \Rightarrow^* w$?

$$useful(n) = prod(n) \wedge reach(n)$$

zyklische Regeln: Gilt $n \Rightarrow^* n$?

Regeln

$$\frac{n \to m \in R}{loop(n, m)} \quad \frac{loop(k, l) \wedge loop(l, m)}{loop(k, m)} \qquad \frac{loop(n, n)}{cyclic(n)}$$

reduzieren

entferen alle nutzlosen Nichtterminale und deren Regeln

Solange es zyklische Regeln gibt, entferne nach und nach
alle Regeln $n \to m \in R$ mit *loop*(*m*,*n*)

Eigenschaften von Grammatiken

leere Produktionen

Gilt $n \Rightarrow^* \varepsilon$? (oder $\varepsilon \in L(n)$)

Regeln

$$\frac{n \to \varepsilon \in T}{null(n)} \quad \frac{(null(v_1) \wedge \ldots \wedge null(v_k)) \wedge n \to v_1 \ldots v_k \in R}{null(n)}$$

Linksrekursion

Gilt $n \Rightarrow^* n\alpha$?

Regeln

$$\frac{n \to m\alpha \in R}{lprod(n, m)} \quad \frac{lprod(k, l) \wedge lprod(l, m)}{lprod(k, m)} \qquad \frac{lprod(n, n)}{lrec(n)}$$

(Rechtrekursion ginge analog,
ist aber nicht problematisch)

Anfänge und Nachfolger

Anfänge

Bestimme $First(n) = \{t \in T \mid n \Rightarrow^* t\,\alpha\}$

Regeln

$$\frac{t \in T}{t \in First(t)} \qquad \frac{n \to \alpha \in R}{First(n) \subseteq First(\alpha)}$$

$$\frac{true}{First(\varepsilon) = \{\bot\}} \qquad \frac{t \in First(v)}{t \in First(v\alpha)} \qquad \frac{\bot \in First(v) \wedge t \in First(\alpha)}{t \in First(v\alpha)}$$

Nachfolger

Bestimme $Follow(n) = \{t \in T \mid s \Rightarrow^* \alpha n\,\beta \Rightarrow^* \alpha t\,\beta\}$

Regeln

$$\frac{true}{\bot \in Follow(s)} \qquad \frac{n \to \alpha m \beta \in R}{First(\beta) \subseteq Follow(m)}$$

$$\frac{n \to \alpha m \beta \in R \wedge \bot \in First(\beta)}{Follow(n) \subseteq Follow(m)}$$

Beispiel Grammatik für Ausdrücke

Nichtterminal	First	Follow
S	(id	^
E	(id	⊥ +)
T	(id	⊥ + *)
F	(id	⊥ + *)

SLL(1)-Bedingung

Definition

die Anfänge aller Alternativen eines Nichtterminals
müssen wechselseitig disjunkt sein

(SLL(1) = *strong leftmost left to right with 1 symbol lookahead*)

Regeln

$$\frac{n \to \alpha \in R \wedge n \to \beta \in R}{(First(\alpha) \cap First(\beta)) = \varnothing}$$

$$\frac{n \to \alpha \in R \wedge n \to \beta \in R \wedge \bot \in First(\beta)}{(First(\alpha) \cap Follow(n)) = \varnothing}$$

Konsequenz

linksrekursive Grammatiken
können diese Bedingung *nie* erfüllen

Beispiel

```
E → T1
E → E + T2
```

$First(T) \subseteq First(E)$

Erweiterung auf SLL(k)

betrachte die ersten *k* Symbole

Erweiterung auf LL(k)

betrachte die First-Mengen im Kontext der jeweiligen Parsier-Situation

Entfernen von Linksrekursion

Einschränkung: direkte Linksrekursion

$n \to \alpha$
$n \to n\,\beta$ (n taucht in α und β nicht auf)

$$\begin{aligned} L(n) &= L(\alpha) \cup L(n\,\beta) \\ &= L(\alpha)\,L(\beta)^* \end{aligned}$$

$n \to \alpha n'$
$n' \to \varepsilon$
$n' \to \beta\,n'$

Linksrekursion entfernen am Beispiel

Grammatik für Ausdrücke

```
S → E
E → T
E → E + T
T → F
T → T * F
F → id
F → ( E )
```

	First	Follow
S → E		
E → T E′		
E′ → ε		⊥)
E′ → + T E′	+	
T → F T′		
T′ → ε		⊥) +
T′ → * F T′	*	
F → id	id	
F → (E)	(	

die umgeformte Grammatik ist SLL(1)

Nichtterminal	First	Follow
S	(id	^
E	(id	⊥)
E′	⊥ +	⊥)
T	(id	⊥) +
T′	⊥ *	⊥) +
F	(id	⊥) + *

rekursiver Abstieg

Prinzip

jedes Nichtterminal *N* wird in eine Prozedur *parseN* umgesetzt

lex enthält das gerade zu erkennende Lexem

match(*t*) liest das Symbol *t* oder meldet einen Fehler,
wenn ein anderes Lexem vorliegt

nextlex liest das nächste Symbol
(wenn das laufende schon angesehen wurde)

parseN erfüllt die Invariante:
- vor Aufruf enthält *lex* einen Anfang von *N*
- nach Aufruf enthält *lex* einen Nachfolger von *N*

(außer im Fehlerfall)

reumütige Rückbesinnung

die Umsetzung wird für EBNF definiert
- das geht fast so einfach wie für kf Grammatiken
- dann spart man Hilfs-Nichtterminale

Transformiere EBNF in Modula-2

Regeln zu Prozeduren

Annahme: jedes Nichtterminal hat *genau ein*e Regel

n ::= *E* ⇒ `procedure parse_n; begin` *parse_E* `end`

reguläre Ausdrücke zu Prozedurrümpfen

parse_ε	⇒	`skip`
parse_t	⇒	`nextlex` wenn *t* das erste Lexem einer Alternative oder Wiederholung ist
parse_t	⇒	`match(`*t*`)` sonst
parse_n	⇒	`parse_n`
parse_E F	⇒	*parse_E*; *parse_F*
parse_E | F	⇒	`if lex.kind in` *First(E)* `then` *parse_E* `elsif lex.kind in` *First(F)* `then` *parse_E* `else` *Syntaxfehler* `end`
parse_(E)	⇒	*parse_E*
parse_{E |}	⇒	`while lex.kind in` *First(E)* `do` *parse_E* `end`

EBNF-Transformation am Beispiel

EBNF von Ausdrücken

```
S  → E eot          E  → T {+ T}
T  → F {* F}        F  → id | ( E )
```

reguläre Ausdrücke zu Prozedurrümpfen

```
procedure parse_S;
   begin parse_E; accept(eot) end
procedure parse_E;
   begin
   parse_T;
   while lex.kind in {plusop}
   do nextlex; parse_T
   end
   end
procedure parse_T;
   begin
   parse_F;
   while lex.kind in {mulop}
   do nextlex; parse_F
   end
   end
procedure parse_F;
   begin
      if lex.kind in {id}
      then nextlex
      elsif lex.kind in {lpar}
      then nextlex; parse_E; match(rpar)
      else Syntaxfehler
      end
   end
```


Fehlerbehandlung

analog zur lexikalischen Fehlerbehandlung

einfügen, tauschen, überlesen
- ein Lexem einfügen

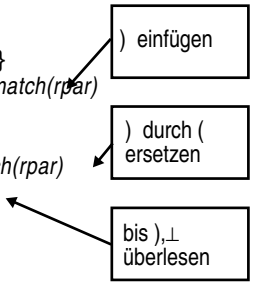
a*(b+c*d• ⇒ a*(b+c*d)•
- ein Lexem gegen ein anderes austauschen

a*•)b+c)*d ⇒ a*•(b+c)*d
- ein Lexem überlesen (oder auch mehrere)

a*(•+12)*d ⇒ a*(•)*d

Beispiel

```
procedure parse_F;
   begin
      if lex.kind in {id}
      then nextlex
      elsif lex.kind in {lpar}
      then nextlex; parse_E; match(rpar)
      else
         if lex.kind = rpar
         then error; parse_E; match(rpar)
         else skip_over({rpar})
         end
      end
   end
```



allgemein (theoretisch)

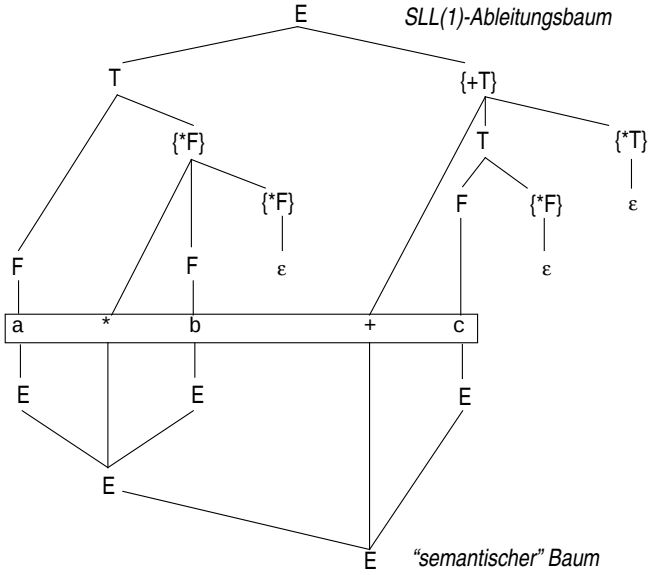
bestimme die kleinstmöglichen Veränderungen der Eingabe,
für die das Parsieren gelingt

das geht, ist aber ineffizient

Ableitungsbäume für rekursiven Abstieg

rekursiver Abstieg liefert eine Zerteilung

als Aufrufstruktur des Programms



aber nicht die, die wir für die Übersetzung brauchen!

- viele semantisch gleichwertige Nichtterminale (E,T,F)
  14 statt 5 Knoten
- überflüssige Symbole (z.B. Klammern)
- falsche Prioritäten und Assoziativitäten

Aspekte der Syntax

Referenzsyntax

Definition in der Sprach-Beschreibung

```
S → E eot          E → T | E + T
T → F |T * F       F → id | ( E )
```

Implementierungssyntax

zum Parsieren umgeformte Syntax

```
S  → E eot
E  → T {+ T}
T  → F {* F}
F  → id | ( E )
```

Abstrakte Syntax

semantisch orientierte Struktur, von Bäumen

(Mehrdeutigkeit ist dann kein Problem mehr!)

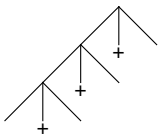
Regel	Baum-Konstruktor
E → id	useE(id)
E → E + E	dyadE(E, plus, E)
E → E * E	dyadE(E, times, E)

```
type Ekind =(use, dyad);
     Etree =pointer to record
     case Ekind of
        use: Ident;
      | dyad:  op: Op; left, right: Etree;
     end
   end
```


Baumaufbau

rekursives "Einfädeln" des Baumes

Die Assoziativität stimmt!




```
procedure parse_E(var T: Etree);
   var L: Etree;
   begin
   parse_T(L);
   while lex.kind in {plusop}
   do nextlex; parse_T(T); L:= dyadE(L,plus, T)
   end;
   T:= L
   end
procedure parse_T(var T: Etree);
   var L: Etree;
   begin
   parse_F(L);
   while lex.kind in {mulop}
   do nextlex; parse_F(T); L:= dyadE(L,times, T)
   end;
   T:= L
   end
procedure parse_F(var T: Etree);
   begin
      if lex.kind in {id}
      then T:= useE(lex); nextlex
      elsif lex.kind in {lpar}
      then nextlex; parse_E(T); match(rpar)
      else
         if lex.kind = rpar
         then error; parse_E(T); match(rpar)
         else skip_over({rpar, eot})
         end
      end
   end
```


Syntax-gesteuerte Transduktion

Parsieren + semantische Aktionen, z. B. für

- Baumaufbau (letzte Folie)
- Übersetzung (hier unten)

```
procedure parse_E;
   var L: Etree;
   begin
   parse_T;
   while lex.kind in {plusop}
   do nextlex; parse_T;
      emit(add)
   end
   end
procedure parse_T;
   var L: Etree;
   begin
   parse_F;
   while lex.kind in {mulop}
   do nextlex; parse_F;
      emit(mult)
   end;
   end
procedure parse_F;
   begin
      if lex.kind in {id}
      then emit(ldv, lex.spelling); nextlex
      elsif lex.kind in {lpar}
      then nextlex; parse_E; match(rpar)
      else error
      end
   end
```

Codeerzeugung aus dem Baum

Traversieren + semantische Aktionen

```
procedure code_E(T: Etree);
  begin
  case T^.Ekind of
    use:
       emit(ldv, lex.spelling);
   | dyad:
       code_E(T^.left);
       code_E(T^.right);
       if    T^.op = plus
       then  emit(add)
       elsif T^.op = times
       then  emit(mult)
       else  error
       end
  end
```

Vorteile gegenüber direkter Codeerzeugung

- kompaktere Definition
- flexibel bei Erweiterung der Syntax um Operatoren

Syntax von Ausdrücken, richtig abstrakt

```
E  → N       useE(N)
E  → N EE    apply(N, EE)
N  → id      (Variablen, Literale, Funktionen,
N  → op       Aggregat-Konstruktoren, Operatoren)
EE → E       voidE
EE → E EE    prodEE(E, EE)
```


aufsteigende Analyse

Unterschied zu absteigender Analyse

Entscheidung für eine Regel kann später getroffen werden

mehrere Regeln (Alternativen) werden parallel verfolgt

Zustände sind dann Merkmalsmengen (*item sets*)

Beispiel (Ausdrücke)

Startzustand I_0:	S →•E	Kern des Startsymbols
Möglichkeiten:	E →•T	Abschluß
	E →•E+T	(alle möglichen
	T →•F	Fortsetzungen
	T →•T*F	des Parsierens)
	F →•id	
	F →•(E)	

Übergänge in Folgezustände (*shift*)

unter Lesen eines Terminals oder Nichtterminals

I_1 = goto(I_0, E) = { S →E•, E →E•+T }
I_2 = goto(I_0, T) = { E →T•, T →T•+F }
I_3 = goto(I_0, F) = { T →F• }
I_4 = goto(I_0, id) = { F →id• }
I_5 = goto(I_0, () = { T →(•E) }

Reduzieren

in Zuständen $N \to \alpha\bullet$ wird N in die Eingabe eingefügt

Zustände: Stapel (*stacks*) von Merkmalsmengen

Kern, Abschluß und Sprung

Kerne (*cores*)

Kern des Startzustands: Die Startregel mit • am Anfang

(wir nehmen an, das Startsymbol habe nur eine Regel)

Abschluß (*closure*)

einer Merkmalsmenge

(1) $I \subseteq I^*$ (2) $\dfrac{n \to \alpha\bullet m\beta \in I^* \wedge m \to \gamma \in R}{m \to \bullet\gamma \in I^*}$

Sprung-Merkmale (*goto items*)

einer Merkmalsmenge

$$\frac{n \to \alpha\bullet v\beta \in I^*}{n \to \alpha v\bullet\beta \in Go(I^*, v)}$$

Konstruktion der LR(0)-Zustandsmengen:

$S_0 = \{\{ s \to \bullet\alpha, s \to \alpha \in R \}\}^*$;
repeat $i := i+1$;
$S_i = S_{i-1} \cup \bigcup_{v \in V, I \in S_{i-1}} Go(I, v)^*$
until $S_i = S_{i-1}$

der charakteristische endliche Automat

(bzw. dessen Zustandsübergangsdiagramm)

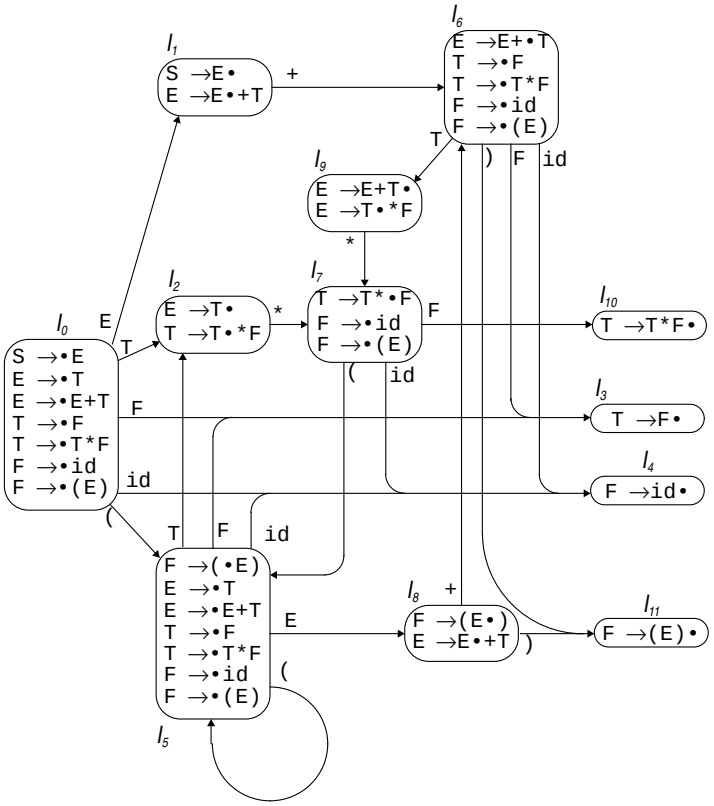
Knoten: Merkmalsmengen (I)

Kanten: Sprünge (I) —v→ (J) wenn $J = Go(I,v)^*$

LR(0)-Automat für eindeutige Ausdrücke

charakteristischer endlicher Automat

für Ausdrücke



Aktion- und Sprungtabelle

Aktionstabelle

Zustand × Terminal → Aktion

Aktion = "**s**" Zustand | "**r**" Regelnummer | **acc** | **err**

Regeln

$$\frac{n \to \alpha \bullet t\beta \in I_i \wedge Go(I_i, t) = I_j}{Aktion[i, t] = \mathbf{s}j}$$

$$\frac{n \to \alpha \bullet \in I_i \wedge t \in Follow(n))}{Aktion[i, t] = \mathbf{r}no(n \to \alpha)}$$

$$\frac{s \to \alpha \bullet \in I_i}{Aktion[i, \mathbf{eot}] = \mathbf{acc}}$$

Sprungtabelle

Zustand × Nichterminal → Zustand

Regel

$$\frac{Go(I_i, n) = I_j}{Sprung[i, n] = j}$$

Fehlererkennung

Alle anderen Einträge sind **err**

Konflikte bei den Einträgen (Grammatik nicht SLR(1))

schieben oder reduzieren

reduzieren oder reduzieren

SLR(1)-Tabellen für eindeutige Ausdrücke

Die Regeln

```
S → E ♦
E → E + T  1
E → E      2
T → T * F  3
T → F      4
F → ( E )  5
F → id     6
```

Zustände	Aktionen						Sprung		
	id	+	*	(	)	♦	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Der Parser

Stapel (*stack*)

Zustand {Symbol Zustand}*

Der Parser

```
begin
firstlex;
push(0); { Startzustand}
while Action[top,lex] <> "acc"
do case Action[top,lex] of
   | "s" i: push(lex); push(i); nextlex;
   | "r" n: pop(2*length(n)); z:= top;
            push(lhs(N)); push(Sprung[z, N]);
   else ... { Fehlerbehandlung }
end
```

Beispiel: Parsieren von x+y*z ♦

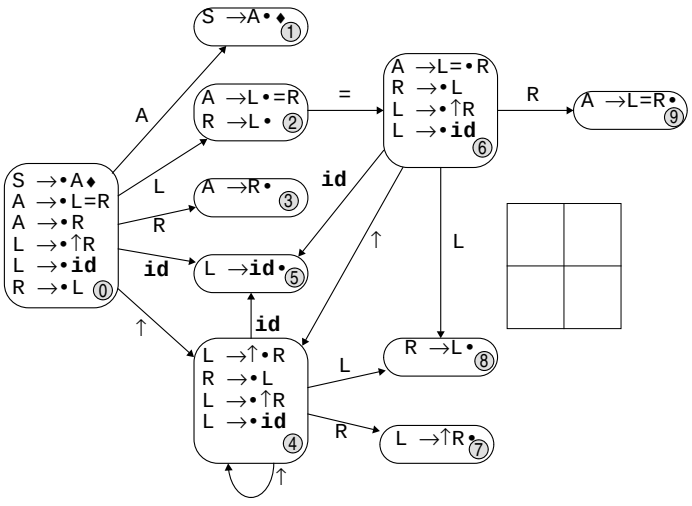
0 • x+y*z **eot**	
→ 0 id 5 • +y*z ♦	s5
→ 0 F 3 • +y*z ♦	r6
→ 0 T 2 • +y*z ♦	r4
→ 0 E 1 • +y*z ♦	r2
→ 0 E 1 + 6 • y*z ♦	s6
→ 0 E 1 + 6 id 5 • *z ♦	s5
→ 0 E 1 + 6 F 3 • *z ♦	r6
→ 0 E 1 + 6 T 9 • *z ♦	r4
→ 0 E 1 + 6 T 9 * 7 • z ♦	s7
→ 0 E 1 + 6 T 9 * 7 id 5 • ♦	s5
→ 0 E 1 + 6 T 9 * 7 F 10 • ♦	r6
→ 0 E 1 + 6 T 9 • ♦	r3
→ 0 E 1 • ♦	r1
→ 0 E 1 ♦ •	**acc**

SLR(1)-Konflikte (1)

Grammatik für Zuweisungen in C

S → A ♦	0
A → L = R	1
A → R	2
L → ↑ R	3
L → **id**	4
R → L	5

charakteristischer endlicher Automat für Zuweisungen



SLR(1)-Konflikte (2)

Die Parsiertabelle

Zustände	Aktionen				Sprung		
	♦	=	↑	**id**	A	L	R
0			s4	s5	1	2	3
1	**acc**						
2	r5	**s6** **r5**					
3	r2	r2					
4			s4	s5		8	7
5	r4	r4					
6			s4	s5		8	9
7	r3	r3					
8	r5	r5					
9		r1					

mögliche Konflikte

shift–reduce: Schieben und reduzieren ist möglich

$n \to \alpha\bullet t\beta \in I \wedge n \to \alpha\bullet \in I \wedge t \in Follow(n)$

reduce–reduce: Es kann nach zwei Regeln reduziert werden

$n \to \alpha\bullet \in I \wedge m \to \beta\bullet \in I \wedge t \in Follow(n) \cap Follow(m)$

Konflikt im Beispiel

in Zustand 2 kann geschoben oder reduziert werden

Ursache

Die Abschlußberechnung berücksicht den Kontext nicht genug

LR(1)-Merkmale

erweiterte Merkmale

Merkmale mit Vorschauzeichen

$N \to \alpha\bullet\beta: t \equiv$ von Regel $N \to \alpha.\beta$ wurde α erkannt
in einem Zustand, in dem t folgen kann

Abkürzung für Merkmale mit gleichem Kern

$N \to \alpha\bullet\beta; s \,|\, t \equiv \{N \to \alpha\bullet\beta; s, N \to \alpha\bullet\beta; t\}$

LR(1)-Abschluß

einer erweiterten Merkmalsmenge

$$(1)\ I \subseteq I^* \quad (2)\ \frac{n \to \alpha\bullet m\beta; t \in I^* \wedge m \to \gamma \in R \wedge s \in First(\beta t)}{m \to \bullet\gamma; s \in I^*}$$

Sprung-Merkmale

einer Merkmalsmenge

$$\frac{n \to \alpha\bullet v\beta; t \in I^*}{n \to \alpha v\bullet\beta; t \in Go(I^*, v)}$$

Konstruktion der LR(0)-Zustandsmengen:

$S_0 = \{\{s \to \bullet\alpha; \textbf{eot}\}^*\}$;
repeat $i := i+1$;
$S_i = S_{i-1} \cup \bigcup_{v \in V, I \in S_{i-1}} Go(I, v)^*$
until $S_i = S_{i-1}$

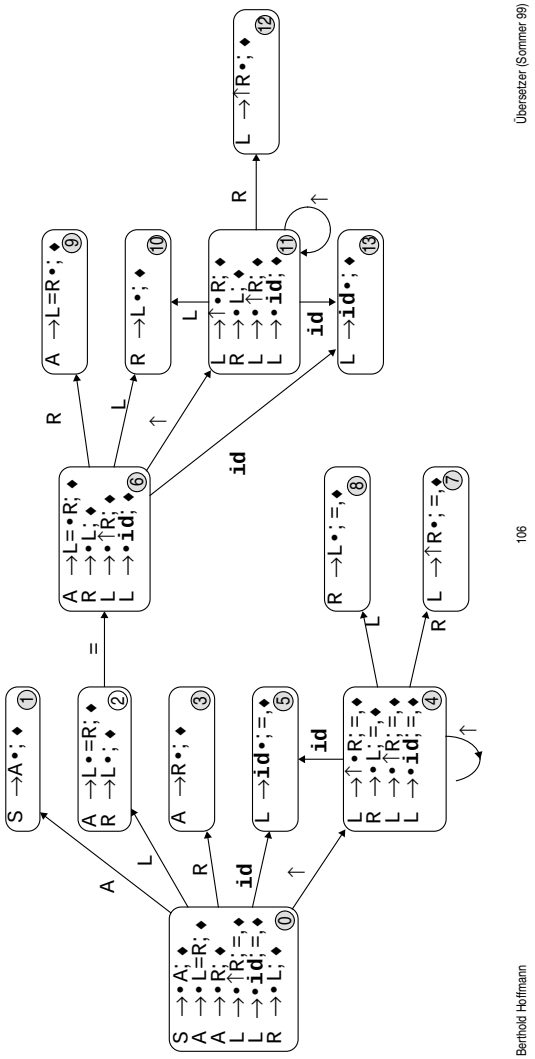
der charakteristische endliche Automat

analog

LR(1)-Parser

S → A ♦ 0 A → R 2 L → **id** 4
A → L = R 1 L → ↑R 3 R → L 5

charakteristischer endlicher Automat für Zuweisungen



LR(1)-Tabelle

Die Parsiertabelle

Zustände	Aktionen				Sprung		
	♦	=	↑	**id**	A	L	R
0			s4	s5	1	2	3
1	**acc**						
2	r5	**s6**					
3	r2						
4			s4	s5		8	7
5	r4	r4					
6			s11	s13		10	9
7	r3	r3					
8	r5	r5					
9	r1						
10	r5						
11			s11	s13		10	12
12	r3						
13	r4						

die Konflikte verschwinden

Zustände mit unterschiedlichem Lookahead werden getrennt

50% mehr Zustände und Speicherplatz

für Programmiersprachen mit ca. 100 Regeln im Durchschnitt 10 mal so groß

LALR(1)-Tabellen

Prinzip

identifiziere Zustände mit gleichem Kern

Die Parsiertabelle

Zustände	Aktionen				Sprung		
	♦	=	↑	**id**	A	L	R
0			s4	s5	1	2	3
1	**acc**						
2	r5	s6					
3	r2						
4=11			s4	s5		8	7
5=13	r4	r4					
6			s4=11	s5=13		8=10	9
7=12	r3	r3					
8=10	r5	r5					
9	r1						
10	r5						
11			s4=11	s5=13		8=10	7=12
12	r3						
13	r4						

die meisten Konflikte bleiben verschwinden

shift-reduce-Konflikte immer

Tabellengröße wie bei SLR

für Programmiersprachen mit in der Regel ausreichend

Präzedenz-gesteuerte LR(1)-Parser

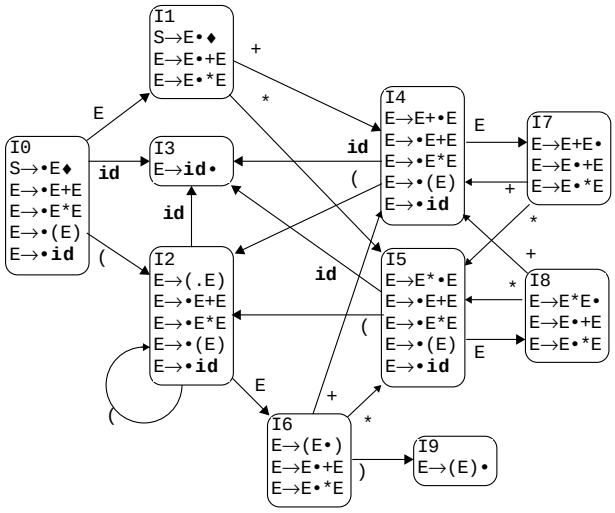
Idee

mehrdeutige Grammatiken werden benutzt

shift–reduce-Konflikte werden aufgelöst durch Operator-Präzedenzen

Beispiel

```
S → E ♦      0
E → E + E    1
E → E * E    2
E → ( E )    3
E → id       4
```



SLR-Tabellen mit Präzedenzen

die Tabelle enthält *shift–reduce*-Konflikte

Zustände	Aktionen						Sprung
	id	+	*	(	)	♦	E
0	s3			s2			1
1		s4	s5			**acc**	
2				s2			6
3		r5	r5		r5	r5	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4 r2	s5 r2		r2	r2	
8		s4 r3	s5 r3		r3	r3	
9		r4	r4			r4	

LR-Parsieren mit Präzedenzen

Operatorpräzedenzen

	+	*
+	>	<
*	>	>

definieren die mathematischen Konventionen

+ > + heißt:	$x + y + z = (x + y) + z$	Linksssoziativität
* > * heißt:	$x * y * z = (x * y) * + z$	
+ < * heißt:	$x + y * z = x + (y + z)$	Präzedenz
* > + heißt:	$x * y + z = (x * y) + z$	

Modifikation des Parsers

Bei *shift–reduce*-Konflikten der Form ... $n\ q_1 \otimes q_2\ m\ q_3 \bullet \oplus w$

wird die Tabelle wie folgt benutzt:

$\otimes > \oplus$ *reduziere* : ... $k\ q_0 \bullet \oplus w$

$\otimes > \oplus$ *schiebe* ... $n\ q_1 \otimes q_2\ m\ q_3 \oplus q_4 \bullet w$

Vorteile

kleinere und übersichtlichere Syntax

effizienterer Parser

LR(1)-Fehlerbehandlung

Prinzip

trifft der Parser auf einen leeren Tabellenplatz, geht er in den *Fehlermodus*

Veränderungen der Eingabe bei der Fehlerbehandlung

ein Zeichen *überlesen* bis zu einem Wiederaufsetzpunkt (*skip*)

ein Zeichen einfügen (*insert*)

ein Zeichen austauschen

Grade der Fehlerbehandlung

panic mode
Benede das Parsieren

lokal (YACC)
lösche im Stack und der *ungelesenen* Eingabe

global (Burke und Fisher, ML-YACC)
lösche im Stack und der Eingabe, auch bis zu *k* gelesene Symbole

minimal oder fast-minimal (Dean)
führe die (fast) geringste Veränderung aus,
mit der die Eingabe ein syntaktisch korrektes Programm wird

Aktionen bei der Fehlerbehandlung

ein Zeichen *überlesen* bis zu einem Wiederaufsetzpunkt (*skip*)

ein Zeichen einfügen (*insert*)

ein Zeichen austauschen

lokale Fehlerbehandlung (wie in YACC)

Fehlersymbole und Fehlerregel

Die Syntax wird um Regeln ergänzt, die das Symbol **`error`** enthalten

Beispiel

```
S → E ♦          0
E → E + E        1
E → E * E        2
E → error + E    1e
E → error * E    2e
E → ( E )        3
E → ( error )    3e
E → id           4
```

Wirkung (z. B. in YACC)

`error` wird wie ein normales Terminalsymbol
in die Übergangstabellen und Aktins- und Sprungtabellen eingetragen

In einem Fehlerzustand (Akrionstabelle ist leer):

- Poppe so viele Symbol-Zustandspaare, bis ein Zustand erreicht ist, für den unter **`error`** ein *shift* eingetragen ist.
- Führe den *shift* aus
- Entferne Eingabesymbole, bis ein Zustand erreicht ist, in dem ein *shift* unter dem Vorschauzeichen möglich ist.
- Fahre mit dem normalen Parsieren fort.

Vorsicht

- Fehlerproduktionen sollten ein Synchronisationszeichen nach **`error`** haben sonst könnte es zu Folgefehlern kommen
- Semantische Aktionen können durch das Abräumen "außer Tritt" geraten insbesondere bei Aktionen mit Seiteneffekten (z. B. durch Ausgabe)

globale Fehlerbehandlung (Burke und Fisher, ML-YACC)

Vorteil

die Grammatik und die Tabellen werden nicht verändert
(nur der Tabellentreiber)

bis zu *k* schon gelesene Symbole werden berücksichtigt
(oft wird der Fehler erst entdeckt,
nachdem das den Fehler verursachende Symbol schon gelesen wurde)

Ziel

finde die geringste Veränderung der letzten *k* gelesenen Symbole,
mit der der Parser *am weitesten* fehlerfrei weiter parsieren kann

eine Veränderung ist "gut genug",
wenn *3-4 weitere Zeichen* fehlerfrei parsiert werden können

Verfahren

Es wird ein aktueller und ein alter Stack
und eine Schlange mit *k* Symbolen verwaltet

wird ein Zeichen geschoben,
wird es auf den aktuellen Stack gelegt und ans Ende der Schlange gelegt

gleichzeitig werden der Kopf der Schlange auf den alten Stack geschoben
und die nötigen Reduktionen auf dem alten Stack durchgeführt

die semantischen Aktionen werden erst und nur auf dem alten Stack ausgeführt
(damit sie im Fehlerfall nicht wiederholt ausgeführt werden)

im Fehlerfall
werden alle möglichen Veränderungen auf der Schlange ausprobiert
und mit dem alten Stack geparst
die gringste Veränderung, die gut genug ist, wird genommen

(Für eingefügte Symbole müssen semantische Werte definiert werden)

Baumaufbau

Implementierung

Ein *Attributstack* wird parallel zum Parse-Stack verwaltet

für jedes Nichtterminal wird ein Baum auf diesen Stack gelegt

Beim Reduzieren werden Knoten aufgebaut (wenn nötig)

S → E ♦	$S_0 = E_1$
E → T	$E_0 = T_1$
E → E + T	$E_0 = \text{plus}(E_1, T_2)$
T → F	$T_0 = F_1$
T → T * F	$T_0 = \text{times}(T_1, F_2)$
F → (E)	$F_0 = E_1$
F → id	$E_0 = \text{use\_id}(\text{lex.repr})$

YACC (1)

Eindeutige Ausdrücke

```
%% /* Deklaration für semantischen Aktionen */

%%

/* Deklaration von zusammengesetzten Lexemen */
%token id
S : E          { $$ = $1 }
  ;
E : T          { $$ = $1 }
  | E"+" T     { $$ = plus($1, $2 ) }
  ;
T : F          { $$ = $1}
  | T"*" F     { $$ = times($1, $2 ) }
F : "(" E ")"{ $$ = $1 }
  | id         { $$ = use_id(lex.repr) }
%% /* Definitionen für semantischen Aktionen */
%%
```


YACC (2)

Mehrdeutige Ausdrücke

```
%% /* Deklaration für semantischen Aktionen */

%%

/* Deklaration von Operatorpräzedenzen */
%left "+"
%left "*"
/* Deklaration von zusammengesetzten Lexemen */
%token id
S : E          { $$ = $1 }
  ;
E : E"+" E     { $$ = plus($1, $2 ) }
  | E"*" E     { $$ = times($1, $2 ) }
  | "(" E ")"{ $$ = $1 }
  | id         { $$ = use_id(yyvalue) }
%% /* Definitionen für semantischen Aktionen */
%%
```


interaktive Übersetzer

***batch*-Übersetzer**

Edieren des Quelltextes mit einem Texteditor

Übersetzen des vollständigen Quelltextes in den Zielcode

interaktives Übersetzersystem

Edieren und Übersetzen wird *verzahnt*

- der Editor ist sprachorientiert (er kennt die Syntax der Sprache)
- der Editor kann nach jeder Änderung das Programm *inkrementell* parsieren
- der Editor erlaubt *strukturelles Editieren* durch Auswahl von Schemata (z. B. die Schemata für Kontrollstrukturen)
- der Editor speichert gleich den abstrakten Syntaxbaum ab (er hält im wentlichen *nur* diesen Baum, nicht den Quelltext)

Vor-und Nachteile

batch-Übersetze

- flexibel (freie Editorwahl)
- einfach zu implementieren

interaktives Übersetzersystem

- benutzerfreundliche (integrierte) grafische Schnittstelle
- schnelle Diagnose von Fehlern
- vereinfachtes Erlernen durch strukturelles Edieren

Syntax-orientiertes Editieren

interaktives Erstellen von Syntaxbäumen

Edieren: quit, file, load, goto, cut, copy, paste

construct: strukturiertes Editieren

parse: Text-Eingabe

```
procedure edit(var T: Tree);
var buffer, cursor: Tree;
begin
T:= nil; cursor:= T; buffer:= nil;
  loop
    receive(cmd);
    case cmd.name of
      quit : return
    | file : write(T)
    | load : T:= read(cmd.arg); cursor:= T
    | goto : cursor:= cmd.arg;
    | cons : cursor:= construct(cmd.arg)
    | parse: cursor:= parse(cursor.type)
    | cut  : buffer:= cursor; cursor:= nil
    | copy : buffer:= cp(cursor)
    | paste: cursor:= buffer
    else error
    end;
    display(T,cursor, buffer)
  end
end
```