Dies ist nun also die freundlicherweise von mir mitgetippte Fassung der Vorlesung **Informatik** (3. Semester) bei **Hr. Prof. Dr. Poller** an der **BA-Mannheim**.

Ich hoffe ihr könnt damit was anfangen.

Fehler, Kritik, Anregungen und alles was euch sonst noch dazu einfällt mailt bitte an
st_tost@hotmail.com

Literatur

Aho, Sethi, Ullmann: Compilerbau (~200,-) [Standard wenn man Compiler bauen will]
Ullmann, Hopcroft: Formale Sprachen und Automatentheorie (Formal Languages and Automata Theorie)
Becker, Walter: Formale Sprachen
Zima: Compilerbau (WI-Verlag)
Nikolaus Wirth: Compilerbau (~50,-)

Inhalte / Gliederung

22 Seiten

3. Semester - Programmiersprachen und deren Übersetzer *(früher Compilerbau)*

- Theorie
  - formale Sprachen und Automaten
    - einseitig lineare Sprachen ↔ endlicher Automat
    - kontextfreie Sprachen ↔ Kellerautomat
- Programmiersprachen
  - lexikalischer Aufbau
  - syntaktischer Aufbau
  - semantischer Aufbau
  - eigene Sprache entwickeln
- Übersetzer
  - Frontend
    - lexikalische Analyse
    - Syntaxanalyse
    - semantische Analyse
  - Backend
    - Objektcodegenerierung
    - Maschinencodegenerierung

Stephan Tost
st_tost@hotmail.com

Sprache: Warum – Wie – Was – Wieso ?

- automatische Sprachübersetzung ist sehr kompliziert
- Kontextumfang ist nicht sicher abgrenzbar (Der Müller ma(h)lt, der Maler ma(h)lt, beide ma(h)len.)
- Wortschatz sehr umfangreich
  (bei hochgebildeten Menschen: 10.000 aktiv und 50.000 passiv, Compiler ~20)

Der lexikalische Aufbau beschreibt aus welchen Buchstaben eine Sprache aufgebaut ist.
Lexikalische Regeln legen fest nach welchen Regeln Buchstaben in den Worten auftauchen (Beispiel: Großbuchstaben nicht mitten im Wort, ...).
Die Syntax (Grammatik) beschreibt welche Worte in welcher Form wie hintereinander stehen (müssen).
Semantik
- Zusammenhang zwischen einzelnen Worten (Beispiel: nach IF muß ein Boolscher Ausdruck folgen)

Compiler und Interpreter führen beide dazu, das programmierter Code in Maschinensprache umgewandelt wird, Unterschied:
Compiler erzeugt ausführbaren Code $\leftrightarrow$ Interpreter erzeugt keinen solchen.

Formale Sprachen

L(G) = Language of Grammar
L(G) = (T(G), Z(G), S(G), P(G))

T(G)	Menge der Terminalsysmbole
Z(G)	Menge der Nichtterminalsysmbole
S(G)	Startsymbole
P(G)	Regeln

$A(G) = T(G) \cup Z(G)$ Gesamtalphabet
$T(G) \cap Z(G) = \emptyset$
$S(G) \subseteq Z(G)$
$P(G) \subset A^*(G) \times A^*(G)$

Beispiel:
$T(G) = \{a, b\}$
$Z(G) = \{A_0, A_1, A_2, A_3, A_4$
$S(G) = \{A_0\}$
$P(G) = \{ \; A_0 \rightarrow a \mid b \mid aA_1 \mid bA_2$
$A_1 \rightarrow aA_3 \mid a$
$A_2 \rightarrow bA_4 \mid b$
$A_3 \rightarrow a$
$A_4 \rightarrow b \; \}$

gesucht: Menge aller korrekten Worte $\rightarrow$ reguläre Menge RM
$A_0 \rightarrow \; aA_1$
aaA_3
$A_0 \rightarrow \; bA_2$
bbA_4
RM = {a, b, aa, aaa, bb, bbb}

Chromsky Hierarchie

Ch(1) + Ch(0) = einseitig begrenzte + natürliche Sprache

Beispiel: Ch(1)
L(G) = (T(G), Z(G), S(G), P(G)) wenn für alle $(u, v) \in P(G)$ gilt:
$u = u_1 \, x \, v_1$ $\quad v = u_1 \, y \, v_1$ $\quad$ mit $u_1, v_1 \, x \in A^*(G) \setminus \emptyset, \; y \in Z(G)$
$u = u_2 \, x \, v_1$ $\quad v = u_2 \, z \, v_1$

Stephan Tost
st_tost@hotmail.com

Ch(2) – Kontextfreie Sprachen

L(G) = (T(G), Z(G), S(G), P(G)), wenn für alle (u, v) $\in$ P(G) gilt: u $\in$ Z(G)
u $\rightarrow$ v $A_0 \rightarrow A_1aA_2bA_3$
$A_1 \rightarrow xA_1b \mid y$
$A_2 \rightarrow vA_2 \mid zA_1A_2 \mid b$
$A_3 \rightarrow p$

- Syntaxsprache

T(G) = {id, *, +, (,)}
Z(G) = {E, T, F}
S(G) = {E}
P(G) = {E $\rightarrow$ T | E + T
T $\rightarrow$ F | T * F
F $\rightarrow$ id | (E)}

Ch(3)-Sprachen

- einseitig lineare Sprachen
- wachsen immer nur um ein Element

L(G) = (T(G), Z(G), S(G), P(G)), wenn für alle (u, v) $\in$ P(G) gilt: u $\in$ Z(G) und v $\in$ T(G) oder v = $v_1 * \sigma$ mit $v_1 \in$ T(G) und $\sigma \in$ Z(G)

Beispiel:

reguläre Menge: RM : { begin, else, end, if, goto, program, then }

T(G) = {a, b, d, e, f, g, h, i, l, m, n, o, p, r, s, t}
Z(G) = {A_0 … A_{21}}
S(G) = {A_0}
P(G) = {$A_0 \rightarrow bA_1 \mid eA_2 \mid gA_3 \mid iA_4 \mid pA_5 \mid tA_6$
$A_1 \rightarrow eA_7 \mid$
$A_2 \rightarrow lA_8 \mid nA_9$
$A_3 \rightarrow oA_{10}$,
$A_4 \rightarrow f$
$A_5 \rightarrow rA_{11}$
$A_6 \rightarrow hA_{12}$
$A_7 \rightarrow gA_{13}$
$A_8 \rightarrow sA_{14}$
$A_9 \rightarrow d$
$A_{10} \rightarrow tA_{15}$
$A_{11} \rightarrow oA_{16}$
$A_{12} \rightarrow eA_{17}$
$A_{13} \rightarrow iA_{18}$
$A_{14} \rightarrow e$
$A_{15} \rightarrow o$
$A_{16} \rightarrow gA_{19}$
$A_{17} \rightarrow n$
$A_{18} \rightarrow n$
$A_{19} \rightarrow rA_{10}$
$A_{20} \rightarrow aA_{21}$
$A_{21} \rightarrow m$}

Eine Ch(3) Sprache wächst beim Aufbau nur ein eine Richtung und nur um ein Element!
ohne Beweis:
Für eine Ch(3) Sprache existiert genau ein endlicher Automat EA(I, F, S, S_0, ϑ) der diese Sprache erkennt und zu jedem endlichen Automaten EA(I, F, S, S_0, ϑ) existiert genau eine Ch(3) Sprache die von EA erkannt werden kann.

Stephan Tost
st_tost@hotmail.com

<u>Beispiel einer Ch(3) Sprache</u>
T(G) = {a, b, _ }
Z(G) = {A_0, A_1}
S(G) = {A_0}
P(G) = {$A_0 \rightarrow aA_1 \mid bA_1$
$A_1 \rightarrow aA_1 \mid bA_1 \mid \_$ }
→ RM {a_, b_, aa_, ab_, ba_, bb_, aaa_, aab_, … }

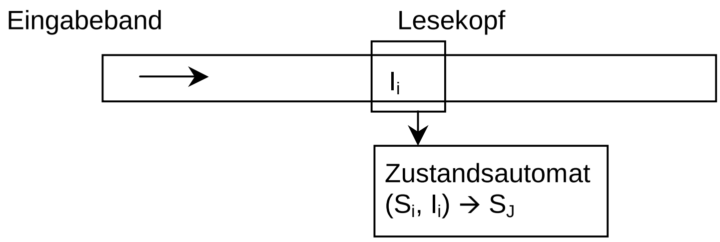
<u>geg:</u> RM : {abc, acb, bac, bca, cab, cba}   <u>ges:</u> Ch(3) Sprache

T(G) = {a, b, c}
Z(G) = {A_0 ... A_9}
S(G) = {A_0}
P(G) = {$A_0 \rightarrow aA_1 \mid bA_2 \mid cA_3$
$A_1 \rightarrow bA_4 \mid cA_5$
$A_2 \rightarrow aA_6 \mid cA_7$
$A_3 \rightarrow cA_8 \mid bA_9$
$A_4 \rightarrow c$
$A_5 \rightarrow b$
$A_6 \rightarrow c$
$A_7 \rightarrow a$
$A_8 \rightarrow b$
$A_9 \rightarrow a$}

Kurzform:
{$A_0 \rightarrow aA_1 \mid bA_2 \mid cA_3$
$A_1 \rightarrow bA_4 \mid cA_5$
$A_2 \rightarrow aA_4 \mid cA_6$
$A_3 \rightarrow aA_5 \mid bA_6$
$A_4 \rightarrow c$
$A_5 \rightarrow b$
$A_6 \rightarrow a$}

<u>EA – endlicher Automat</u>

I : Eingabesymbole



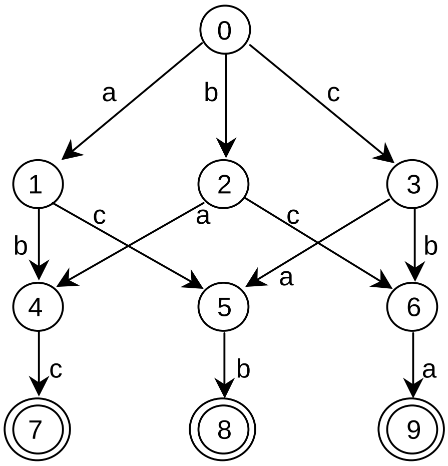
F : Endzustände F c S
S_0 : Anfangszustände S_0 c S S_0 n F = ∅ S : alle Zustände
σ : Menge der Übergangsfunktion
$\sigma \subset S \times I \mid (S_i, I_i) \rightarrow S_J$

EA für Kurzform (s.o.)

Stephan Tost
st_tost@hotmail.com

I = {a, b, c}
F = {7, 8, 9}
S_0 = {0}
S = {0 … 9}
σ = { (0, a) → 1,
(0, b) → 1,
(0, c) → 1,
(1, b) → 4,
(1, c) → 5,
(2, a) → 4,
(2, c) → 6,
(3, a) → 5,
(3, b) → 6,
(4, c) → 7,
(5, b) → 8,
(6, a) → 9}

Graphisch:



<u>Generierung eines EA aus der Ch(3) Sprache</u>

1) I : Eingabecontroller (Die Elemente von I entsprechen den Elementen von T(G))
2) S_0 : Startzustände: Die Elemente von S_0 entsprechen den Elementen von S(G)
3) S/F : Zustände ohne Endzustände: Die Elemente von S \ F entsprechen den Elementen von Z(G)
4) Regeln der Sprache (u, v) € P(G) mit v = v_0 roh v_1 € T(G) roh € Z(G)
A_J → xA_J daraus wird Übergangsfunktion (S_i, x) → S_J
5) Regeln der Ch(3) : (u, v) € P(G) mit v € T(G)
A_i → y daraus wird Übergangsfunktion (S_i, y) → S_k mit S_k = Endzustand
6) Alle S_k aus Nr. 5 bilden die Menge der Endzustände F

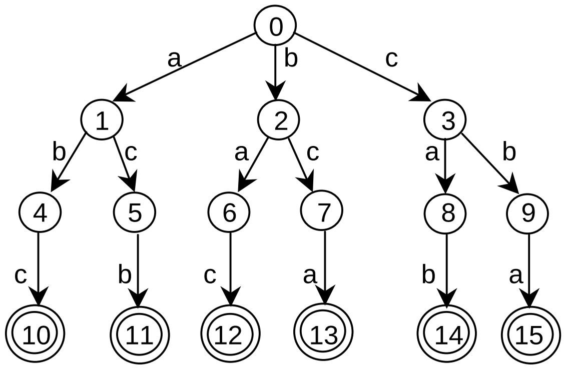
<u>Beispiel:</u>

RM : {abc, acb, bac, bca, cab, cba}
Ch(3)
T(G) = {a, b, c}
Z(G) = {A_0 ... A_9}
S(G) = {A_0}
P(G) = {A_0 → aA_1 | bA_2 | cA_3
A_1 → bA_4 | cA_5
A_2 → aA_6 | cA_7
A_3 → cA_8 | bA_9
A_4 → c
A_5 → b
A_6 → c
A_7 → a
A_8 → b
A_9 → a}

Stephan Tost
st_tost@hotmail.com

I : {a, b, c}
S_0 : {0}
S \ F : {0 … 9}
F : {10 ... 15}
σ = {   (0, a) → 1,
(0, b) → 2,
(0, c) → 3,
(1, b) → 4,
(1, c) → 5,
(2, a) → 6,
(2, c) → 7,
(3, a) → 8,
(3, b) → 9,
(4, c) → 10,
(5, b) → 11,
(6, c) → 12,
(7, a) → 13,
(8, b) → 14,
(9, a) → 15}

graphisch



Die Ch(3) Sprache kann mittels eines Programmes (LEX) in C-Quellcode umgewandelt werden, sodaß nach dem Compilieren der Scanner zur Verfügung steht (Dieser erkennt dann die Endzustände der ‚gegangenen' Wege.).

Kontextfreie Sprachen / Ch(2) / Syntax

L(G) = (T(G), Z(G), S(G), P(G)) wenn für alle (u, v) € P(G) gilt: u € Z(G)
Die linke Seite einer Regel besteht aus <u>einem</u> Nichtterminal; rechte Seite beliebig.

T(G) = {id, :=, if, then, else, begin, end, goto, zahl, +, -, *, (,), /, :}
Z(G) = {S, E, T, F, SE} [Statement, Expression, Term, Faktor, Simple Expression]
S(G) = {S}
P(G) = {S → if E then S else S | begin S ... S end | goto zahl | id := E | : zahl S
E → -SE
SE → SE + T | SE – T | T
T → F | T * F | T / F
F → zahl | id | (E)}

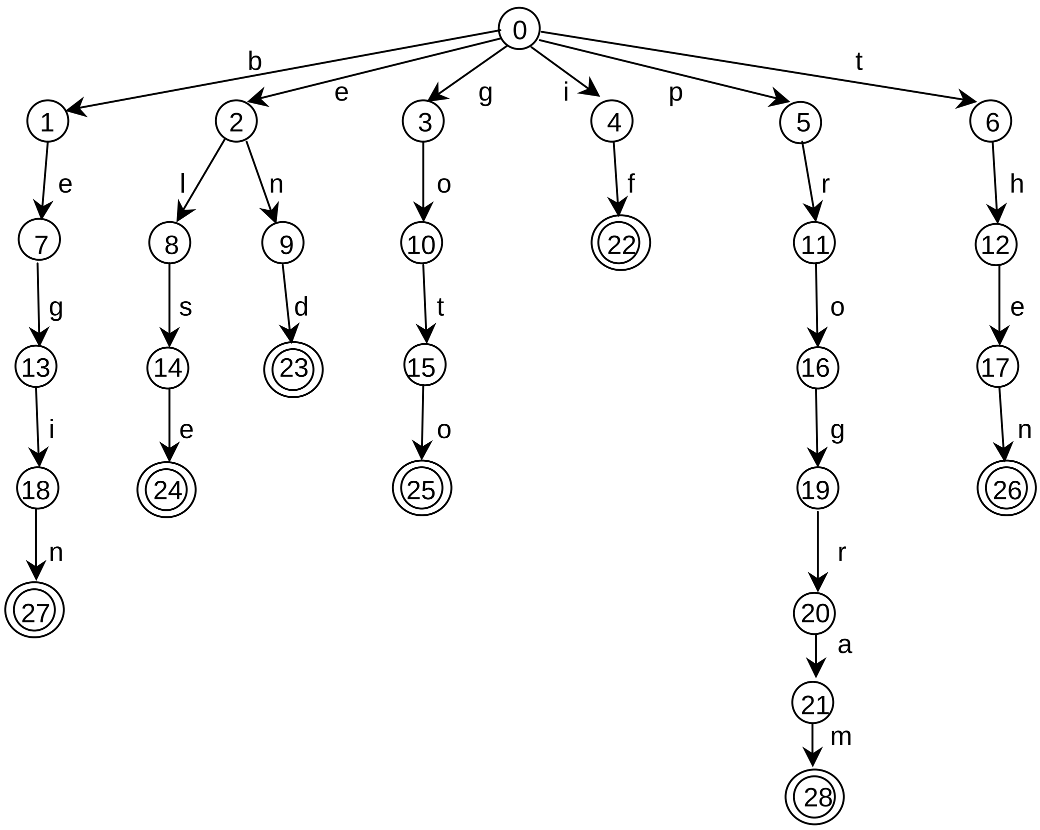
Stephan Tost
st_tost@hotmail.com

Aufgabe

reguläre Menge: RM : {begin, else, end, if, goto, program, then}

T(G) = {a, b, d, e, f, g, h, i, l, m, n, o, p, r, s, t}
Z(G) = {A_0 … A_{21}}
S(G) = {A_0}
P(G) = {$A_0 \rightarrow bA_1 \mid eA_2 \mid gA_3 \mid iA_4 \mid pA_5 \mid tA_6$
$A_1 \rightarrow eA_7 \mid$
$A_2 \rightarrow lA_8 \mid nA_9$
$A_3 \rightarrow oA_{10}$,
$A_4 \rightarrow f$
$A_5 \rightarrow rA_{11}$
$A_6 \rightarrow hA_{12}$
$A_7 \rightarrow gA_{13}$
$A_8 \rightarrow sA_{14}$
$A_9 \rightarrow d$
$A_{10} \rightarrow tA_{15}$
$A_{11} \rightarrow oA_{16}$
$A_{12} \rightarrow eA_{17}$
$A_{13} \rightarrow iA_{18}$
$A_{14} \rightarrow e$
$A_{15} \rightarrow o$
$A_{16} \rightarrow gA_{19}$
$A_{17} \rightarrow n$
$A_{18} \rightarrow n$
$A_{19} \rightarrow rA_{10}$
$A_{20} \rightarrow aA_{21}$
$A_{21} \rightarrow m$}

I : {a, b, d, e, f, g, h, i, l, m, n, o, p, r, s, t}
S_0 : {0}
S \ F : {1 … 21}
F : {22 ... 28}
σ = { (0, b) → 1,
(0, e) → 2,
(0, g) → 3,
(0, i) → 4,
(0, p) → 5,
(0, t) → 6,
(1, e) → 7,
(2, l) → 8,
(2, n) → 9,
(3, o) → 10,
(4, f) → 22,
(5, r) → 11,
(6, h) → 12,
(7, g) → 13,
(8, s) → 14,
(9, d) → 23,
(10, t) → 15,
(11, o) → 16,
(12, e) → 17,
(13, i) → 18,
(14, e) → 24,
(15, o) → 25,
(16, g) → 19,
(17, n) → 26,
(18, n) → 27,
(19, r) → 20,
(20, a) → 21,
(21, m) → 28}



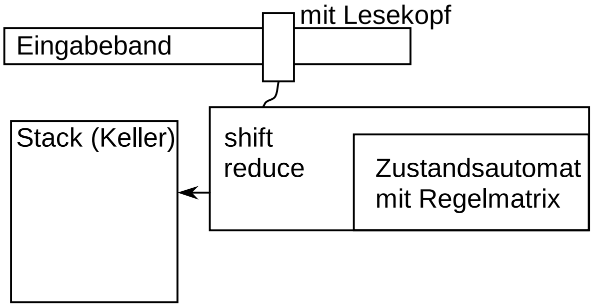
Stephan Tost
st_tost@hotmail.com

kontextfreie Sprachen

- für jedes Nichtterminal eindeutige Übersetzung
  (x → y, x → z   frei wählbar)
- T(G) = {id, +, -, *, /}
  Z(G) = {E, T, F}
  S(G) = {E}
  P(G) = { E → E+T | T
  T → T * F | F
  F → (E) | id}
  kontextfreier Baum
  - enthält die Terminale am Ende der Äste
    (im Baum nur Nichtterminale)

Welcher Automat kann diese kontextfreie Sprache übersetzen?
→ Ein Automat der eine Ch(2)-Sprache übersetzt ist ein **Kellerautomat**.

Kellerautomat



Der Kellerautomat führt zwei Operationen durch (shift / reduce)

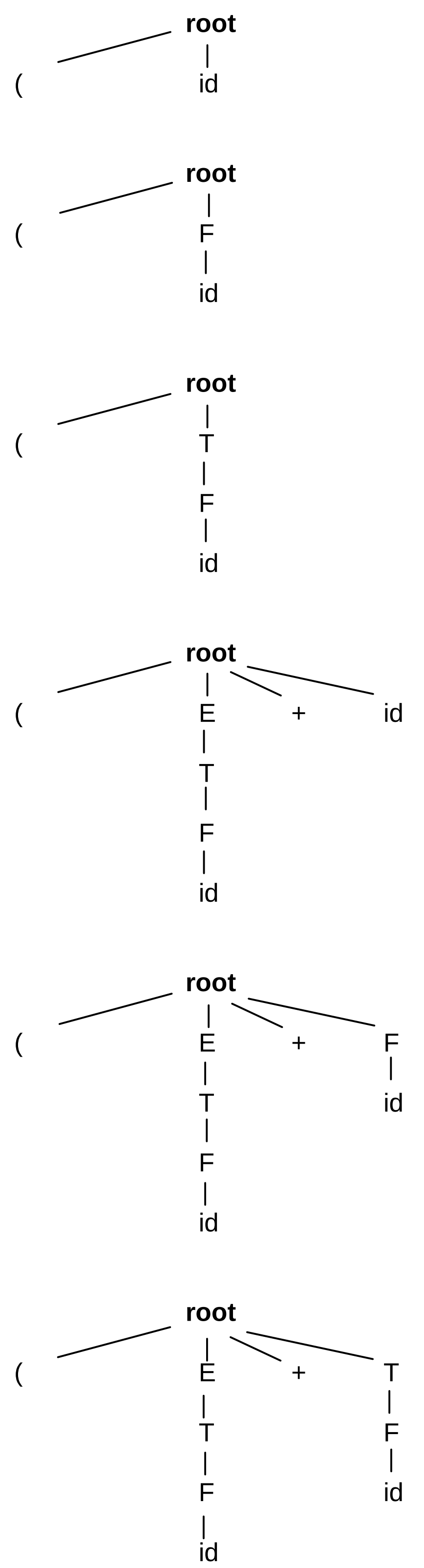
a) Analyse
   a. shift → Zeichen unter Lesekopf wird in den Stack eingetragen
   b. reduce → Elemente der rechten Seite eine Regel werden von Stack genommen und durch die linke Seite ersetzt
b) Aufbau des kontextfreien Baumes
   a. shift → Zeichen wird an die Wurzel gehangen
   b. reduce → rechten Teil der Regel von der Wurzel nehmen und linken Teil dranhängen, dann rechten Teil an linken Teil hängen

(id + id) * id $ $ = Abschlußzeichen
→ steht auf Eingabeband
→ soll analysiert werden

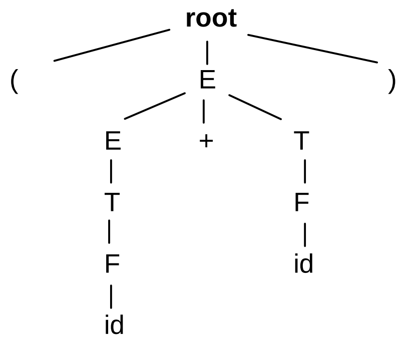
Input	***Stack***	***Operation***	
(id + id) * id $	$	shift	
id + id) * id $	$(	shift	
+ id) * id $	$(id	reduce	
+ id) * id $	$(F	reduce	
+ id) * id $	$(T	reduce	
+ id) * id $	$(E	shift	
id) * id $	$(E +	shift	
) * id $	$(E + id	reduce	
) * id $	$(E + F	reduce	
) * id $	$(E + T	reduce	
) * id $	$(E	shift	
* id $	$(E)	reduce	
* id $	$F	reduce	
* id $	$T	shift	// nicht zu E reducen, da sonst E * id → keine Regel
id $	$T *	shift	
$	$T * id	reduce	
$	$T * F	reduce	
$	$T	reduce	
$	$E	accept!	

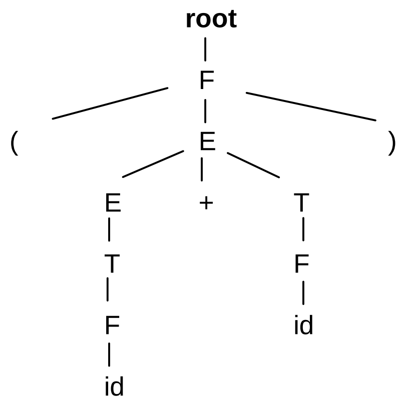
Stephan Tost
st_tost@hotmail.com

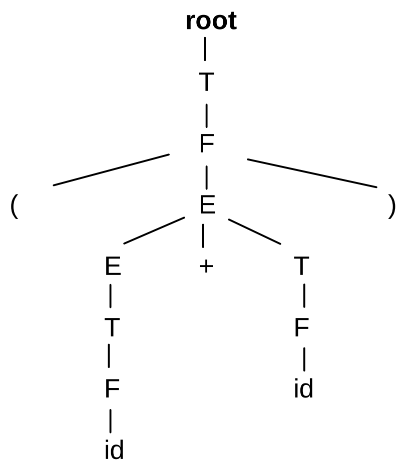
Baumaufbau

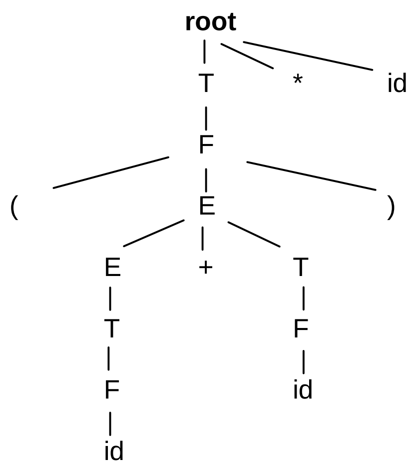


Stephan Tost
st_tost@hotmail.com

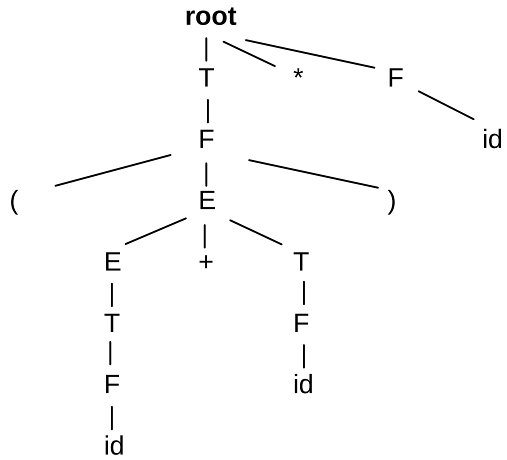


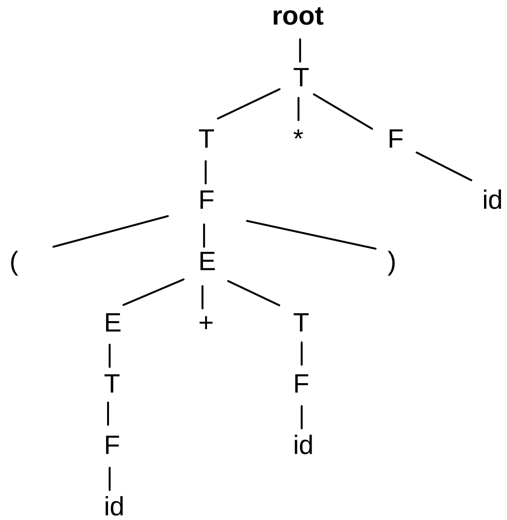


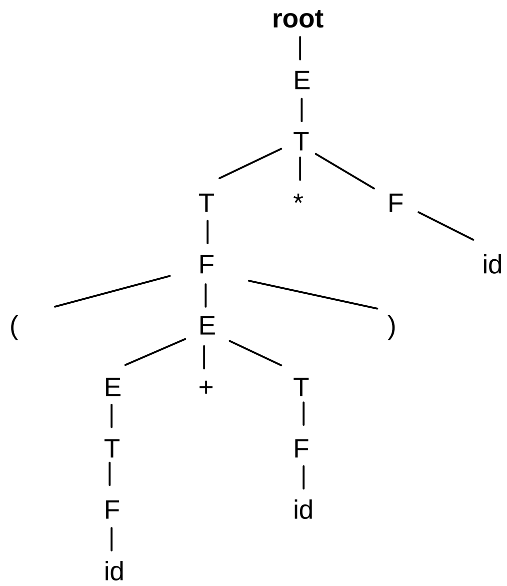




Stephan Tost
st_tost@hotmail.com







Stephan Tost
st_tost@hotmail.com

Baumaufbau direkt aus dem Stack

Stack
$
$(
$(id
$(F
$(T
$(E
$(E +
$(E + id
$(E + F
$(E + T
$(E
$(E)
$F
$T
$T *
$T * id
$T * F
$T
$E

Baum

```
            root
             |
             E
             |
             T
            / \
          T  *  F
         /       \
        F         id
      / | \
     (  E  )
       / \
      E + T
     /     \
    T       F
   /         \
  F          id
 /
id
```

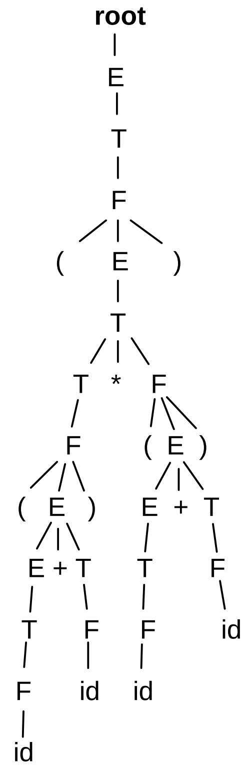

Stephan Tost
st_tost@hotmail.com

<u>weitere Beispiele</u>

P(G) = { E → E+T | T
T → T * F | F
F → (E) | id}

((id + id) * (id + id)) analysieren und Baum erzeugen

Input	***Stack***	***Operation***
((id + id) * (id + id)) $	$	shift
(id + id) * (id + id)) $	$(	shift
id + id) * (id + id)) $	$((	shift
+ id) * (id + id)) $	$((id	reduce
+ id) * (id + id)) $	$((F	reduce
+ id) * (id + id)) $	$((T	reduce
+ id) * (id + id)) $	$((E	shift
id) * (id + id)) $	$((E +	shift
) * (id + id)) $	$((E + id	reduce
) * (id + id)) $	$((E + F	reduce
) * (id + id)) $	$((E + T	reduce
) * (id + id)) $	$((E	shift
* (id + id)) $	$((E)	reduce
* (id + id)) $	$(F	reduce
* (id + id)) $	$(T	shift
(id + id)) $	$(T *	shift
id + id)) $	$(T * (	shift
+ id)) $	$(T * (id	reduce
+ id)) $	$(T * (F	reduce
+ id)) $	$(T * (T	reduce
+ id)) $	$(T * (E	shift
id)) $	$(T * (E +	shift
)) $	$(T * (E + id	reduce
)) $	$(T * (E + F	reduce
)) $	$(T * (E + T	reduce
)) $	$(T * (E	shift
) $	$(T * (E)	reduce
) $	$(T * F	reduce
) $	$(T	reduce
) $	$(E	shift
$	$(E)	reduce
$	$F	reduce
$	$T	reduce
$	$E	accept!



P(G) = { E → E+T | T
T → T * F | F
F → (E) | id}

Stephan Tost
st_tost@hotmail.com

(((id)) id) analysieren und Baum erzeugen

Input	***Stack***	***Operation***
(((id)) id) $	$	shift
((id)) id) $	$(	shift
(id)) id) $	$((	shift
id)) id) $	$(((	shift
)) id) $	$(((id	reduce
)) id) $	$(((F	reduce
)) id) $	$(((T	reduce
)) id) $	$(((E	shift
) id) $	$(((E)	reduce
) id) $	$((F	reduce
) id) $	$((T	reduce
) id) $	$((E	shift
id) $	$((E)	reduce
id) $	$(F	reduce
id) $	$(T	reduce
id) $	$(E	shift
) $	$(E id	error!, keine Regel definiert

Top-Down – Parsing

- wird kaum noch verwendet
- nur möglich, wenn die Sprache eine sogenannte LL(1)-Sprache ist
  (eine Sprache, die von links orientiert ist [man erkennt am linken Zeichen der Regel, welche Regel verwendet wird], Beispiel oben ist keine LL(1)-Sprache)
  LL(1): (S) → if E then S_1 **|** repeat S until E **|** while E do S **|** var E **|** begin S S_2 end
  S_1 → ∅ | else S
  S_2 → ∅ | else S
  E → +ET **|** T
  T → *TF **|** F
  F → id **|** (E)

MINI – eigene Sprache

MINI Identifier = Buchstaben
Zahlen = Ziffern

Datentypen
- if, then, else, goto
- if - then Zuweisung
- begin ... end
- program

Operatoren
- +, -, /, &, *, =, <, >, <>, =<, =>

α = a...z
β = 0...9

Stephan Tost
st_tost@hotmail.com

Ch(3) →
T(G) = {α, β, +, -, *, /, &, =, <, >, :, _, (,)}
Z(G) = {A_0 ... A_{45}}
S(G) = {A_0
P(G) = {A_0 → _A_0 | αA_1 | βA_2 | +A_3 | -A_4 | *A_5 | /A_6 | &A_7 | (A_8 |)A_9 | :A_{10} | =A_{11} | <A_{12} | >A_{13} | bA_{14} | eA_{15} | gA_{16} | iA_{17} | pA_8 | tA_{19}

A_1 →	αA_1 | _		
A_2 →	βA_2 | _		
A_3 →	_		
A_4 →	_		
A_5 →	_	A_{26} →	nA_{27}
A_6 →	_	A_{27} →	_
A_7 →	_	A_{28} →	dA_{29}
A_8 →	_	A_{29} →	_
A_9 →	_	A_{30} →	sA_{31}
A_{10} →	_ | =A_{20}	A_{31} →	eA_{32}
A_{11} →	_ | <A_{21} | >A_{22}	A_{32} →	_
A_{12} →	_ | >A_{23}	A_{33} →	tA_{34}
A_{13} →	_	A_{34} →	oA_{35}
A_{14} →	eA_{24}	A_{35} →	_
A_{15} →	nA_{28} | lA_{30}	A_{36} →	_
A_{16} →	oA_{33}	A_{37} →	oA_{38}
A_{17} →	fA_{36}	A_{38} →	gA_{39}
A_{18} →	rA_{37}	A_{39} →	rA_{40}
A_{19} →	hA_{43}	A_{40} →	aA_{41}
A_{20} →	_	A_{41} →	mA_{42}
A_{21} →	_	A_{42} →	_
A_{22} →	_	A_{43} →	eA_{44}
A_{23} →	_	A_{44} →	nA_{45}
A_{24} →	gA_{25}	A_{45} →	_}
A_{25} →	iA_{26}		

4 verschiedene Outputs
- (id, „name“)
- (zahl, „ziffer“)
- (op, „art“)
- (longword, „art“)

Ch(2) Sprache definieren
Syntax
<u>Terminalsymbole</u>
T(G) = {id, Zahl, +, -, *, /, (,), &, =, =<, =>, <>, <, >, :, :=, begin, end, else, if, goto, then, program, $}
<u>Nichtterminale</u>
Z(G) = {P, S, S_1, S_2, E, SE, T, F, V}   *// V = Variable*

S(G) = {P}
P(G) = {P → program id begin S S_1
S → : Zahl S | goto Zahl | V := E | begin S S_1 | if E then S S_2
S_1 → S S_1 | end
S_2 → else S | ∅
E → -SE | SE | E = E | E =< E | E => E | E > E | E < E | E <> E
SE → T | SE + T | SE – T
T → F | T * F | T / F | T & F
F → id | (E) | Zahl
V → id }

Stephan Tost
st_tost@hotmail.com

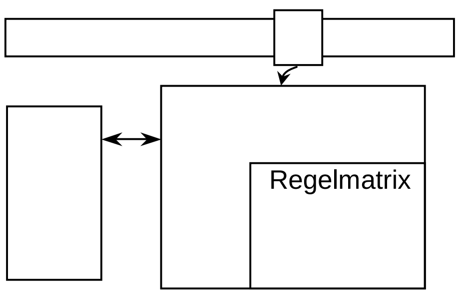
Beispielprogramm das mit dieser Sprache erstellbar wäre:

program mist
begin
   x := 100
   y := 200
   z := 300
:1001
   if x = y **then**
      z := 400 + z + x + y
   else
   begin
      x := x + 1
      goto 1001
   end
end

Compiler
Frontend

- lexikalische Analyse
- Syntaxanalyse
  <u>shift / reduce – Parser</u> $\rightarrow$ kommt jetzt

Regelmatrix



Regelmatrix
T(G) = {+, *, (,), id, $}
Z(G) = {E T, F}
S(G) = {E}
P(G) = { E $\rightarrow$ E + T r_1
E $\rightarrow$ T r_2
T $\rightarrow$ T * F r_3
T $\rightarrow$ F r_4
F $\rightarrow$ id r_5
F $\rightarrow$ (E)} r_6

<u>Regelmatrix</u>
Nach dieser Matrix ist der input-stream zu analysieren.

	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		r5	r5		r5	r5			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s2		r1	r1			
10		r3	r3		r3	r3			
11		r6	r6		r6	r6			

Stephan Tost
st_tost@hotmail.com

id+id$

Input	***Stack***	***Operation***
id+id$	$0	shift
+id$	$0id5	matrix: 5
+id$	$0F3	matrix: 3
+id$	$0T2	matrix: 2
+id$	$0E1	shift
id$	$0E1+6	shift
$	$0E1+6id5	matrix: 5
$	$0E1+6F3	matrix: 3
$	$0E1+6T9	matrix: 9
$	$0E1	accept

*id*id$*

Input	***Stack***	***Operation***
id*id$	$0	shift
*id$	$0id5	matrix: 5
*id$	$0F3	matrix: 3
*id$	$0T2	matrix: 2
id$	$0T2*7	matrix: 7
$	$0T2*7id5	matrix: 5
$	$0T2*7F10	matrix: 10
$	$0T2	matrix: 2
$	$0E1	accept

(id)$

Input	***Stack***	***Operation***
(id)$	$0	shift
id)$	$0(4	shift
)$	$0(4id5	matrix: 5
)$	$0(4F3	matrix: 3
)$	$0(4T2	matrix: 2
)$	$0(4E8	matrix: 8
$	$0(4E)11	matrix: 11
$	$0F3	matrix: 3
$	$0T2	matrix: 2
$	$0E1	accept

*(id+id)*id$*

Input	***Stack***	***Operation***
(id+id)*id$	$0	shift
id+id)*id$	$0(4	shift
+id)*id$	$0(4id5	matrix: 5
+id)*id$	$0(4F3	matrix: 3
+id)*id$	$0(4T2	matrix: 2
+id)*id$	$0(4E8	matrix: 8
id)*id$	$0(4E8+6	matrix: 6
id)*id$	$0(4E8+6	shift
)*id$	$0(4E8+6id5	matrix: 5
)*id$	$0(4E8+6F3	matrix: 3
)*id$	$0(4E8+6T9	matrix: 9
)*id$	$0(4E8	shift
*id$	$0(4E8)11	matrix: 11
*id$	$0F3	matrix: 3
*id$	$0T2	shift
id$	$0T2*7	shift
$	$0T2*7id5	matrix: 5
$	$0T2*7F10	matrix: 10
$	$0T2	matrix: 2
$	$0E1	accept

Stephan Tost
st_tost@hotmail.com

*(id+*id)$*

Input	***Stack***	***Operation***
(id+*id)$	$0	shift
id+*id)$	$0(4	shift
+*id)$	$0(4id5	matrix: 5
+*id)$	$0(4F3	matrix: 3
+*id)$	$0(4T2	matrix: 2
+*id)$	$0(4E8	shift
*id)$	$0(4E8+6	error (nach + kann nicht * folgen → keine Regel)

T(G)={+, *, (,), id, $}
Z(G)={E, T, F}
S(G)={E}

P(G)={	E → E + T	r1
	E → T	r2
	E → T * F	r3
	T → F	r4
	F → id	r5
	F → (E)}	r6

Regelmatrix erstellen

1) Kanonische LR(0) – Mengen aufbauen
2) Erzeugen eines Zustandsautomaten
3) Matrix ableiten

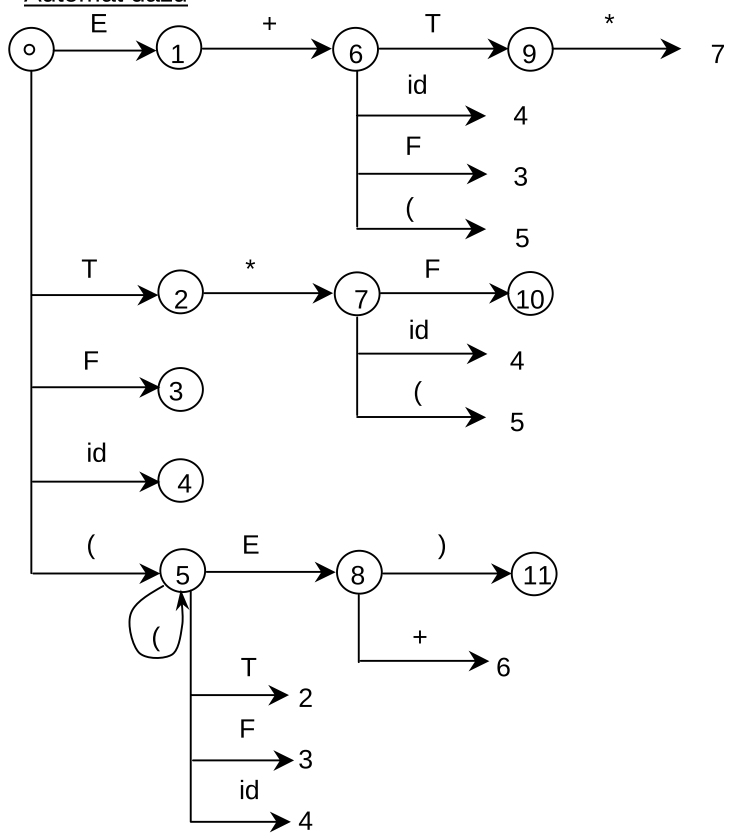
zu 1)
- Operator **.** (Punkt) der uns den aktuellen Stand bei der Abarbeitung einer Regel darstellt
  E → **.**E + T → vor der Abarbeitung
  E → E **.**+ T → E wurde erkannt (+ folgt)
  E → E + T**.** → eine Regel wurde abgearbeitet
- Metaregel: E’ → E

Dann beginn mit Menge I_0 mit E’ → **.**E wenn der Punkt vor einem Nichtterminal steht werden alle folgenden ableitbaren Regeln in die Menge eingetragen.

I_0	E’ → **.**E
	E → **.**E + T
	E → **.**T
	T → **.**T * F
	T → **.**F
	F → **.**id
	F → **.**(E)
I_1	E’ → E**.**
	E → E**.** + T
I_2	E → T**.**
	T → T**.** * F
I_3	T → F**.**
I_4	F → id**.**
I_5	F → (**.**E)
	E → **.**E + T
I_6	E → E + **.**T
	T → **.**T * F
	T → **.**F
	F → **.**id
	F → **.**(E)
I_7	T → T * **.**F
	F → **.**id
	F → **.**(E)
I_8	(E**.**)
	E → E**.** + T
I_9	E → E + T**.**
	T → T**.** * F
I_{10}	T → T * F**.**
I_{11}	F → (E)**.**

Stephan Tost
st_tost@hotmail.com

Automat dazu



Zustandsautomaten erzeugen

zu 3)
a) in Zeile i bei $ = (i, $) kommt acc mit i = I_i in der E' → E.
b) mit (i, a) → j und a = Terminal wird s_j in Zeile i Spalte a eingetragen (s - shift)
c) mit (i, a) → j mit a = Nichtterminal wird j in Zeile i und Spalte a eingetragen
d) in I_i ist eine Regel j abgeschlossen in der Form x → s_0 dann wird j in (i, Follow(x)) eingetragen
E → E + T. E = x
Follow(E) = +,), $
Follow(T) = +, *,), $
Follow(F) = +, *,), $

e) sonst Fehler

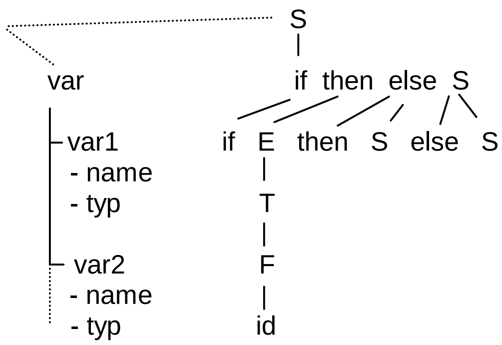
abgeschlossenen Regeln in I 2, 3, 4, 9 und 11

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

Stephan Tost
st_tost@hotmail.com

Semantische Analyse

Kontextfreier Baum ist Input der semantischen Analyse
Ziel:
- semantische Korrektheit überprüfen
  - Vollständigkeit der Deklaration
  - Typverträglichkeit
  - Operationsverträglichkeit
- Erzeugen eines kontextsensitiven Baumes verknüpft mit den Symboltabellen



Deklarationsteil
Definition von Konstanten, Typen, Variablen und Labels
→ Symboltabellen
type id
- Größe, Art
Konstanten
- id, Wert
Variablen
- id, type

3 Adresscode (Backend)

Frontend	
- lexikalische Analyse	Text
- syntaktische Analyse	Tokenstring
- semantische Analyse	
→ kontextabhängiger Baum, Symboltabelle	kontextfreier Baum

Backend Codegenerierung
Objektcodegenerierung
→ um Maschinen unabhängigen Code zu erhalten (3 Adresscode, DIN)
Maschinencodegenerierung

Befehl: Typunabhängig
3 Parameter (die nicht alle belegt sein müssen, meist ist mindestens ein Register mit dabei (temp))

Operationen

- integer	intadd(op1, op2, temp) →	op1 + op2 → temp
	intsub(op1, op2, temp) →	op1 – op2 → temp
	intmult(op1, op2, temp)	
	intdiv(op1, op2, temp) →	op1 / op2 → temp (ganzzahliger Wert ohne Rest)
	intmod(op1, op2, temp)	gibt Rest der Operation in temp zurück
- real	realadd(op1, op2, temp)	
	realsub(op1, op2, temp)	
	realdiv(op1, op2, temp)	
	realmult(op1, op2, temp)	
- boolean	boolnot(op1, , temp)	(2. Operator wird nicht verwendet)
	boolor(op1, op2, temp)	
	booland(op1, op2, temp)	
- char	charpre(op1, , temp)	(Vorgänger)
	charsuc(op1, , temp)	(Nachfolger)

Beispiel: a + b + c + d
intadd(a, b, temp)
intadd(temp, c, temp)
intadd(temp, d, temp)

Stephan Tost
st_tost@hotmail.com

Zuweisungen

load(op1, , temp)	weist op1 temp - Wert zu
assign(temp, , op1)	weist temp op1 - Wert zu

Vergleiche

compeq(op1, op2, temp)	=
compne(op1, op2, temp)	<>
compgt(op1, op2, temp)	>
compls(op1, op2, temp)	<
compge(op1, op2, temp)	>=
comple(op1, op2, temp)	<=

Label

Label(, , L1)

Sprünge

branch(, , L1)
branchtrue(temp, , L1)
branchfalse(temp, , L1)

<u>Beispiele:</u>

if x = y then
  a := a + 1
else
  a := a + 3

```
Label( , , L1)
Label( , , L2)
// Bedingung:
    compeq(x, y, temp)
    branchfalse(temp, , L1)
intadd(a, 1, temp)          // then-Fall: a +1 in Register schreiben
assign(temp, , a)           // und wieder zu a zuweisen
branch( , , L2)
L1: intadd(a, 3, temp)      // else - Fall
    assign(temp, , a)
L2:                         // hier kann auch der then-Fall hingeschrieben werden (sieht besser aus)
```

repeat
  n := n + 2
  i := i – 1
until i <= 0

```
Label( , , L1)
L1: intadd(n, 2, temp)      // repeat-Befehle ausführen
    assign(temp, , n)
    intsub(i, 1, temp)
    assign(temp, , i)
    comple(i, 0, temp)      // wenn Abbruchbedingung nicht erfüllt → Rücksprung zum Anfang
    branchfalse(temp, , L1)
```


Stephan Tost
st_tost@hotmail.com

while (i > 0) do
n := n + 2
i := i – 1

```
Label( , , L1)
Label( , , L1)
L2: compgt(i, 0, temp)          // wenn i nicht größer als 0 ist → Abbruch
    branchfalse(temp, , L1)
    intadd(n, 2, temp)
    assign(temp, , n)
    intsub(i, 1, temp)
    assign(temp, , i)
    branch( , , L2)
L1:
```

for (i = 1 to m) do
n := n + 2

```
Label( , , L1)
Label( , , L2
assign(1, , i)
L2: comple(i, m, temp)
    branchfalse(temp, , L1)     // Abbruch wenn m > i
    intadd(n, 2, temp)
    assign(temp, , n)
    intadd(i, 1, temp)          // i erhöhen
    assign(temp, , i)
    branch( , , L2)
L1:
```


Stephan Tost
st_tost@hotmail.com