Compilerbau

Literatur

1) Aho, A.V., Sethi, R., Ullman, J.D. (1988).
Compilerbau. Teil 1 und 2. ("Drachenbuch")
Addison-Wesley

2) Appel, A.W. (2002).
Modern Compiler Implementation in Java.
2nd Edition. Cambridge University Press

3) Grune, D., Bal, H.E., Jacobs, J.H., Langendoen, K.G.
(2001). Modern Compiler Design.
John Wiley & Sons

4) Jäger, M. (2011).
Compilerbau – eine Einführung.
Skript zur Vorlesung an der THM,
FB MNI. Im WWW

Überblick

1.5 1. Einführung: Aufgaben, Phasen, Läufe, Schnittstellen

1.5 2. Lexikalische Analyse: Reguläre Sprachen, endliche Automaten, Scanner-Generatoren

3 3. Syntaktische Analyse: Kontextfreie Sprachen, Ableitungen, LL- und LR-Parser, Parser-Generatoren

1 4. Abstrakte Syntax und Attributierung

2 5. Semantische Analyse: Typen und Typkonstruktoren, Typüberprüfung, Symboltabelle

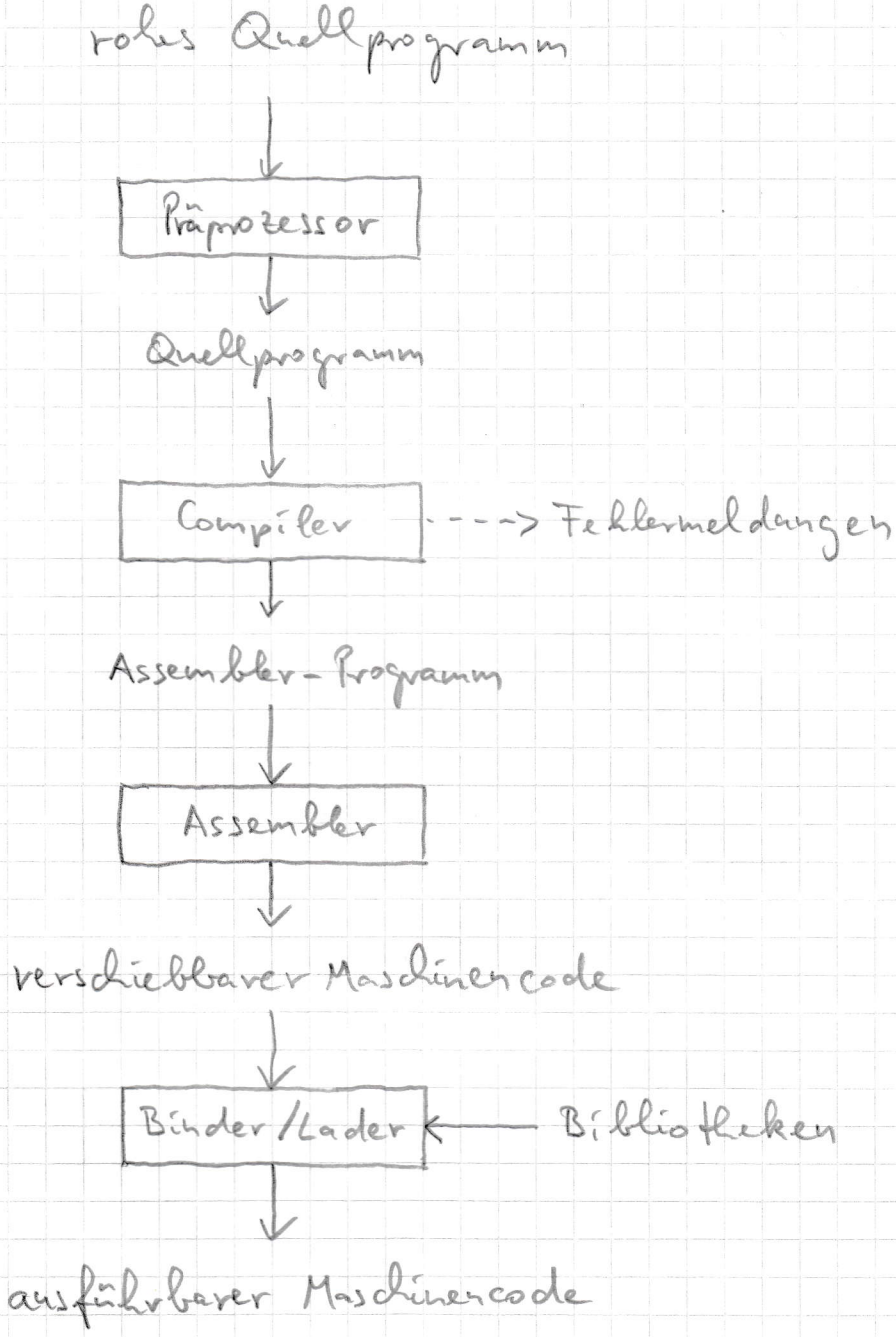
1 6. Laufzeit-Organisation: Stack und Stack-Frames, Prozedur-Eintritt/Austritt, Parameter-Übergabe

2 7. Codegenerierung: Stackmaschinen, Registermaschinen, Übersetzung von Ausdrücken, Übersetzung von Kontrollanweisungen, Übersetzung von Prozeduraufrufen

1 [8. Registerallokation]

1. Einführung

1.1. Der Compiler in der Werkzeugkette



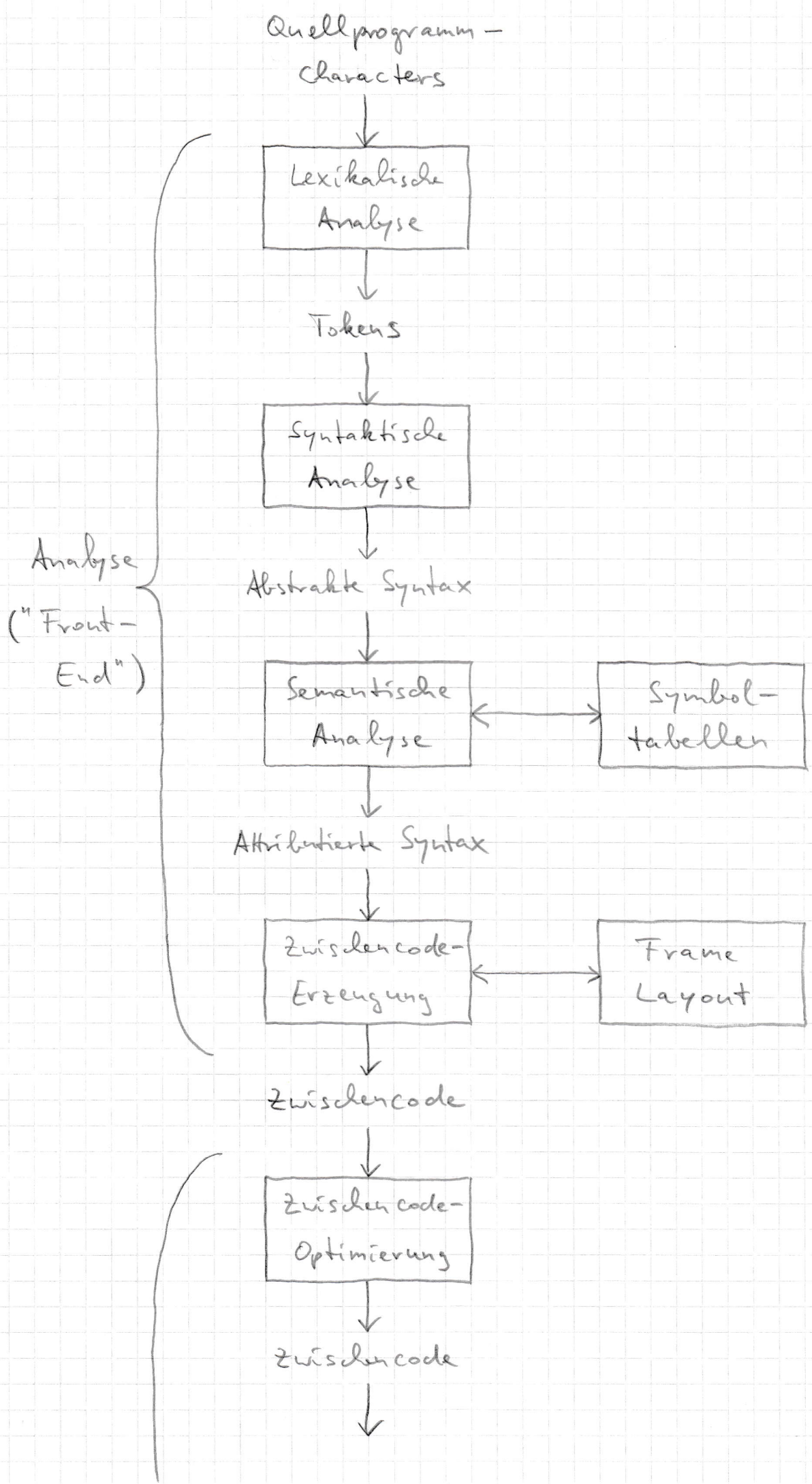
1.2. Phasen, Läufe, Schnittstellen

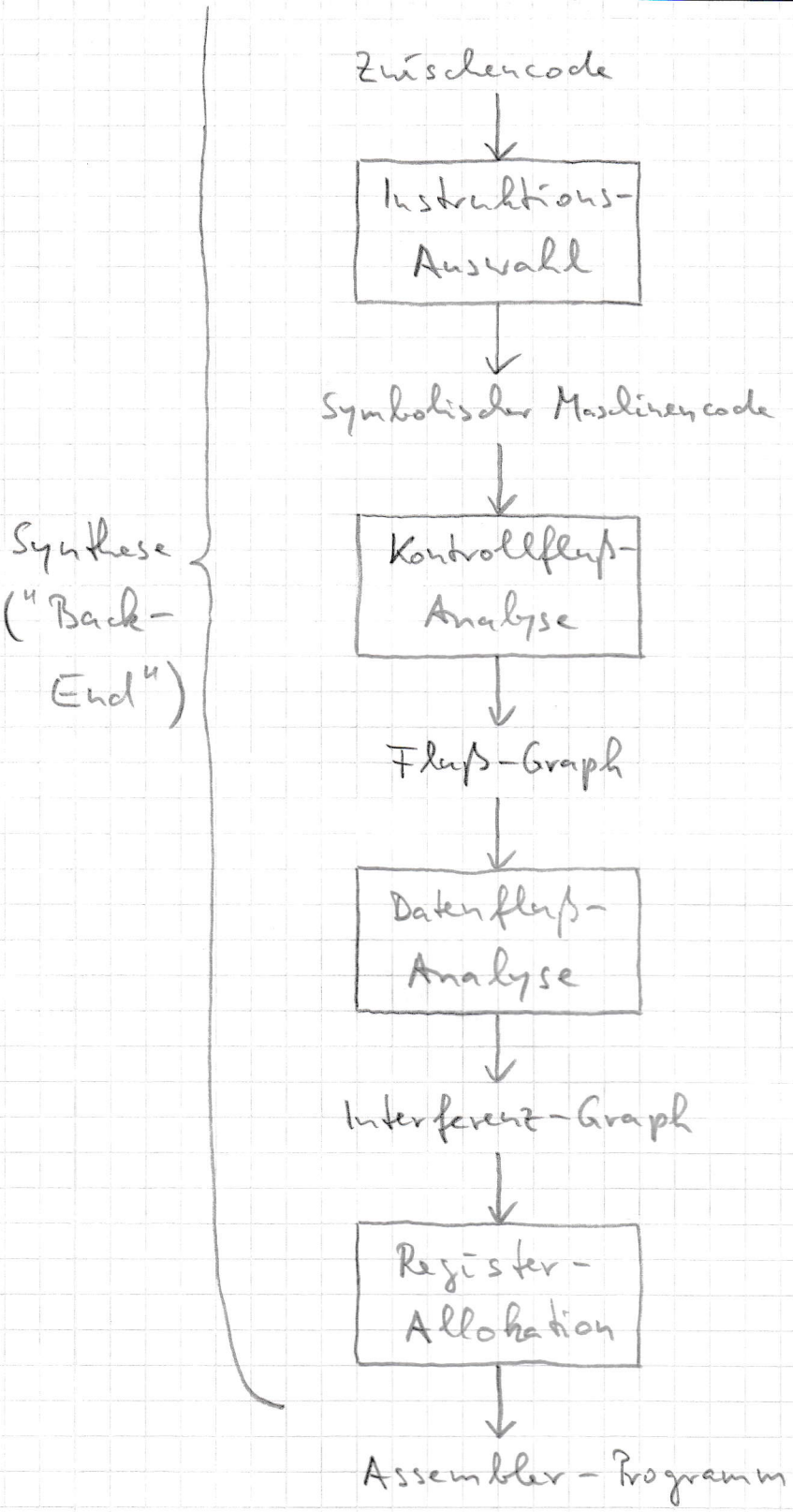
Verteilung der Aufgaben auf Phasen.

Zusammenfassung von Phasen zu Läufen

(ein Lauf ≙ ein Durchgang durch das Programm).

Interne Darstellung des Programms durch Schnittstellen.


Quellprogramm-
characters
Lexikalische
Analyse
Tokens
Syntaktische
Analyse
Abstrakte Syntax
Analyse
("Front-
End")
Semantische
Analyse
Symbol-
tabellen
Attributierte Syntax
Zwischencode-
Erzeugung
Frame
Layout
Zwischencode
Zwischencode-
Optimierung
Zwischencode



Im Folgenden etwas vereinfacht:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Abstrakte Syntax
4. Semantische Analyse
5. Frame-Layout
6. Codegenerierung

Bsp.:

1) Quellcode

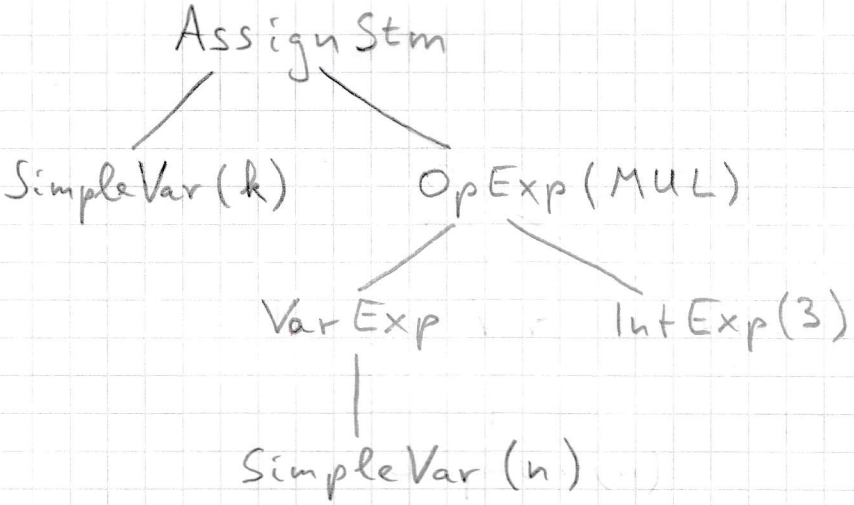
```
proc demo (n: int, ref k: int) {
    k := n * 3;
}
```

2) Tokens (für die Zuweisung)

IDENT("k") ASGN IDENT("n")

STAR INTLIT(3) SEMIC

3) Abstrakte Syntax (für die Zuweisung)



4) Symboltabelle (nach Analyse von "demo")

Level 0: n → var: int
k → var: ref int

Level 1: demo → proc: (int, ref int)
int → type: int
main → proc: ()
⋮
(Bibliotheksfunktionen)

5) Variablenallokation (für "demo")

param 'n': fp + 0

param 'k': fp + 4

local vars: 0

proc calls: none

6) Assemblercode (für die Zuweisung)

```
add   $8, $25, 4     ; fp+4 = Adresse von k
ldw   $8, $8, 0      ; Inhalt von k = Referenz
add   $9, $25, 0     ; fp+0 = Adresse von n
ldw   $9, $9, 0      ; Wert holen
add   $10, $0, 3     ; Konstante 3 nach $10
mul   $9, $9, $10    ; $9 ← $9 * $10
stw   $9, $8, 0      ; mem[$8+0] ← $9
```

2. Lexikalische Analyse

Aufgabe: Strom von Zeichen → Strom von Token

Token: kleinste Bedeutung tragende Einheiten einer Prog. Spr.
("Worte" einer Prog. Spr.)

Bsp.:

Token-Typ	Beispiele aus einem konkreten Programm
IDENT	k ab123 das_Letzte
NUM	123 ϕ ϕxAffe
IF	if
COMMA	,
LE	<=
RPAREN	)

2.1. Reguläre Ausdrücke

Def.:

c) Sprache — Menge von Strings

b) String — endliche Folge von Zeichen

a) Alphabet — alle zur Verfügung stehenden Zeichen (endlich viele)

Bsp.: Alphabet = {a, b}

Sprache = {b, ab, aab, aaab, ...}

Beschreibung einer (regulären) Sprache durch "reguläre Ausdrücke".
Jeder reguläre Ausdruck steht für eine Menge von Strings.

Zeichen: a steht für $L(a) = \{a\}$

Alternative: $\beta|\gamma$ steht für $L(\beta|\gamma) = L(\beta) \cup L(\gamma)$

Bsp: $L(a|b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$

Konkatenation: $\beta\gamma$ steht für $L(\beta\gamma) =$

$$\{st \mid s \in L(\beta) \wedge t \in L(\gamma)\}$$

Bsp: $L(a(b|c)) = \{st \mid s \in L(a) \wedge t \in L(b|c)\}$

$= \{st \mid s \in \{a\} \wedge t \in \{b, c\}\} = \{ab, ac\}$

Epsilon: ε steht für $L(\varepsilon) = \{\varepsilon\}$ (leerer String)

Kleene'scher Abschluß: $\beta*$ steht für $L(\beta*) =$

$L(\varepsilon) \cup L(\beta) \cup L(\beta\beta) \cup L(\beta\beta\beta) \cup \ldots$

Bsp.: $L((ab)*) = L(\varepsilon) \cup L(ab) \cup L(abab) \cup \ldots$

$= \{\varepsilon, ab, abab, ababab, \ldots\}$

Ein paar Beispiele:

a) Binärzahlen, die Vielfache von 2 sind: $(0|1)*0$

c) Strings aus a und b, die keine 2 a's hintereinander enthalten:

$b*(abb*)*(a|\epsilon)$

b) Strings aus a und b, die mind. einmal 2 a's hintereinander enthalten: $(a|b)*aa(a|b)*$

Weitere Abkürzungen:

[abcd] bedeutet $(a|b|c|d)$

[b-e] " $(b|c|d|e)$

$\beta?$ " $(\beta|\varepsilon)$

$\beta+$ " $(\beta\beta*)$

Bsp: Token einer Programmiersprache:

if → 17

. bedeutet irgend ein Zeichen (außer Zeilenumbruch)

[a-z][a-zø-9]* → IDENT

[ø-9]+ → NUM

øx[ø-9a-fA-F]+ → NUM

\) → RPAREN

Auflösen von Mehrdeutigkeiten:

a) Was ist "if8"? Ein IDENT oder IF NUM?

b) Was ist "if"? IDENT oder IF?

Zwei Regeln:

"Längste Übereinstimmung": "if8" ist IDENT

"Zuerst genannte Regel hat Vorrang": "if" ist IF

2.2. Deterministische Endliche Automaten (DEAs)

Bestandteile:

a) Endliche Menge von Zuständen (bezeichnet mit Zahlen)

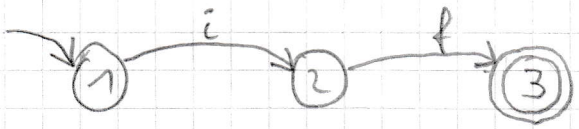
b) " " " Zustandsübergängen (markiert mit Eingabezeichen)

c) Ein ausgezeichneter Zustand, der "Startzustand"

d) Eine Teilmenge der Zustände sind "Endzustände"

"deterministisch": verschiedene Zustandsübergänge aus dem gleichen Zustand heraus sind mit wschiedenen Eingabezeichen markiert; kein Übergang ist mit dem leeren Eingabestring ε markiert.

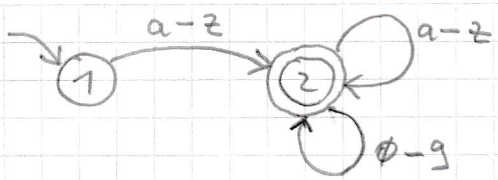
Bsp:



IF

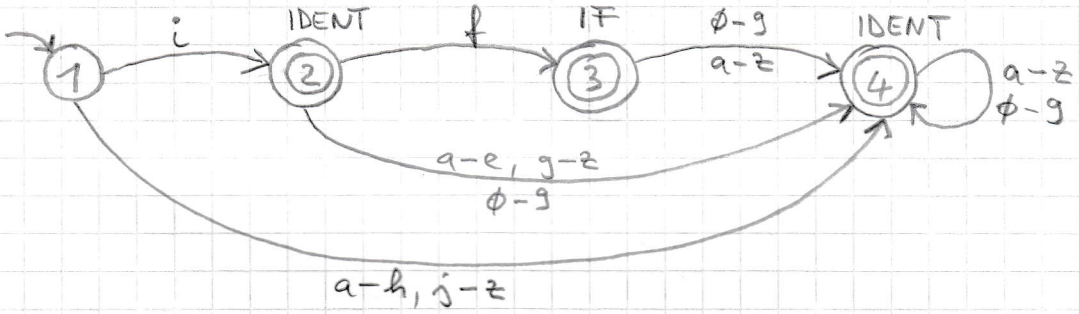
1: Startzustand, 3: Endzustand

Automat erkennt das Token IF.



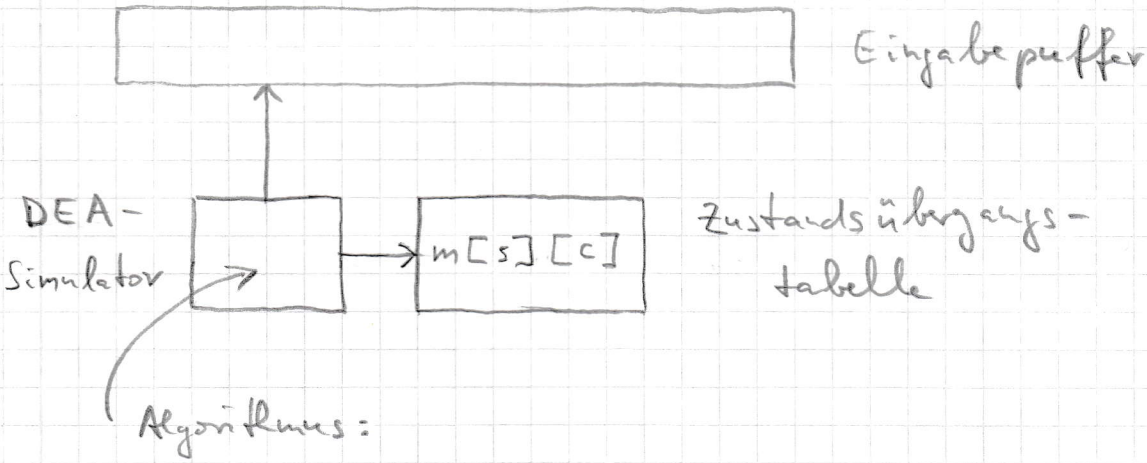
Automat erkennt das Token IDENT.

Kombinierter Automat zum Erkennen von IF und IDENT:



2 Fragen: Wie wird der Automat implementiert?
Wie wird der Automat konstruiert?

2.3. Tabellengesteuerter Scanner




```
s := s0  //Startzustand
c := nextchar();  // nächstes Eingabezeichen
while (c ≠ EOF) {
    s := m[s][c];
    c := nextchar();
}
```

Die erkannte Sprache wird durch m festgelegt!

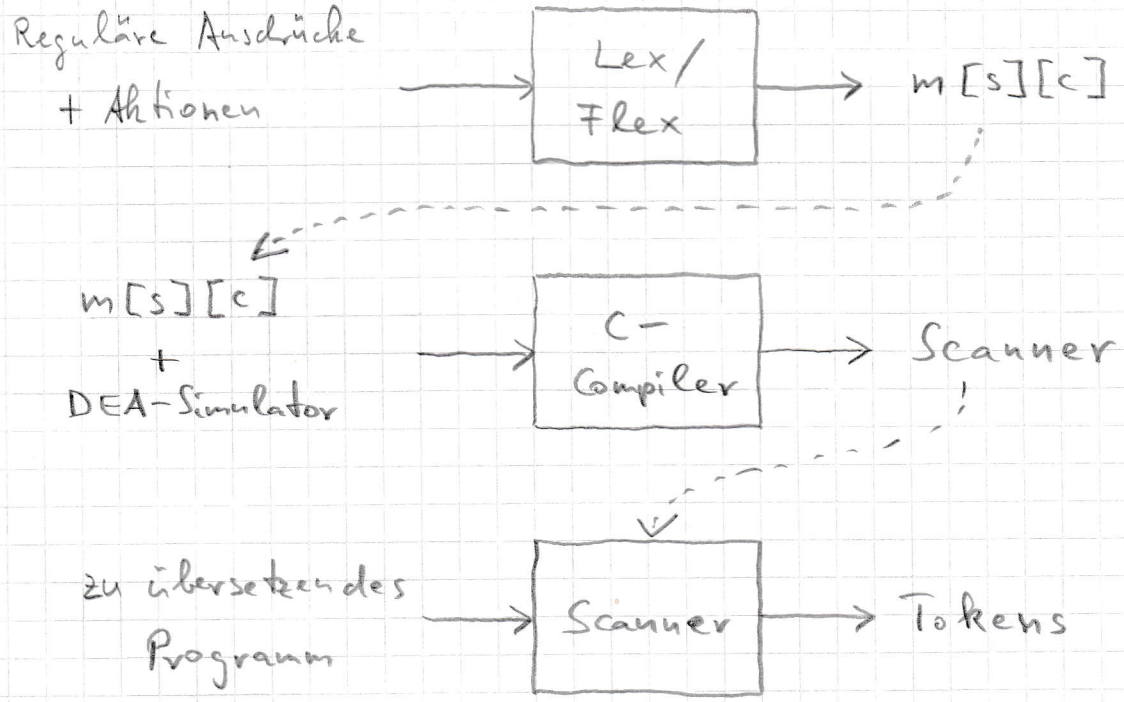
3 Ergänzungen:

a) Wenn ein Zustand s für ein Eingabezeichen c keinen Übergang besitzt, muß m[s][c] eine spezielle Markierung bekommen ("Weiterlaufen nicht möglich")

b) Es muß eine weitere Tabelle geben, die für jeden Zustand sagt, ob's ein Endzustand ist, und ggf. welches Token abgeliefert werden soll.

c) Realisierung der Regel "längste Übereinstimmung": Merken des letzten erreichten Endzustandes und Weiterlaufen des Automaten bis "Weiterlaufen nicht mögl."

2.4. Scanner-Generatoren



Bsp: `[ \ \t]+` `{ /* kein Token zurück */ }`

`if` `{ return IF; }`

```
[a-z][a-zφ-9]*   { return IDENT; }
.                { error("..."); }
```

Bem.: Vernünftige Fehlerausgaben in späteren Phasen verlangen das Speichern einer "Koordinate" (mindestens der Zeilennummer im Quelltext) pro Token!

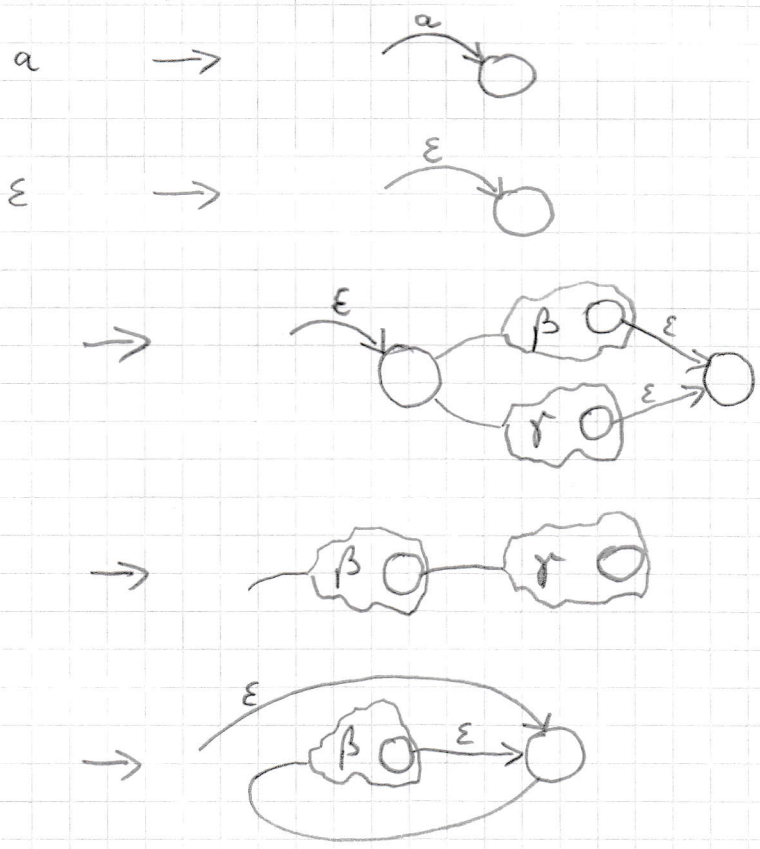
2.5. Konstruktion des DEA

Zwei Schritte:

a) reguläre Ausdrücke → nichtdeterminist. endl. Automat ("NEA")

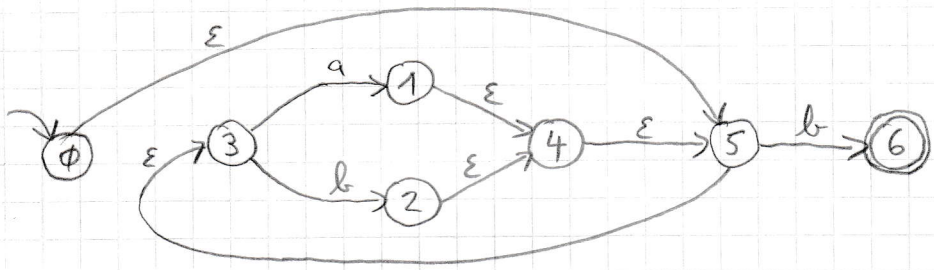
b) NEA → DEA

zu a) "Thompson's Konstruktion":



Am Ende: Startzustand s_0 hinzufügen.

Bsp.: $(a|b)^*b$



Bem.: "nichtdeterministisch": der Automat muß bei gegeb. Zustand und Eingabezeichen raten, wo's langgeht, und er muß richtig raten!

zu b) "Teilmengenkonstruktion"

Idee: Menge aller NEA-Zustände, die für einen gegeb. Eingabestring erreicht werden können = ein DEA-Zustand

Def.:

ε-Abschluß (t): Menge der NEA-Zustände, die vom NEA-Zustand t allein über ε-Übergänge erreicht werden

ε-Abschluß (T): Menge der NEA-Zustände, die von irgendwelchen NEA-Zuständen $t \in T$ allein über ε-Übergänge erreicht werden

move (T, c): Menge der NEA-Zustände, die von irgendwelchen NEA-Zuständen $t \in T$ durch das Eingabezeichen c erreicht werden

S: Menge der Zustände des DEA

m[T][c]: Übergangstabelle des DEA

Teilmengenkonstruktion:

Start: $S := \{\varepsilon\text{-Abschluß}(s_0)\}$; // unmarkiert

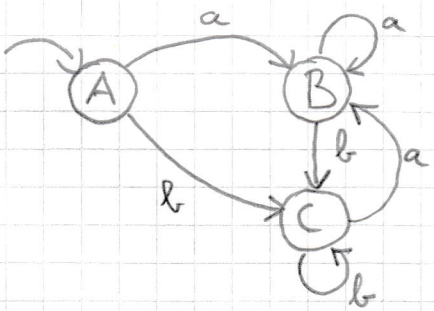
Iteration: while (S enthält einen unmarkierten Zustand T) {

```
markiere T;
for (jedes Eingabezeichen c) {
    X := ε-Abschluß(move(T, c));
    if (X ∉ S) S := S ∪ {X}; //unmarkiert
    m[T][c] := X;
}
}
```

Bsp.: NEA wie oben

NEA-Zustände	DEA-Zustand	Übergänge
$\{\phi, 5, 3\}$	A	
(A, a): $\{1, 4, 5, 3\}$	B	m[A][a] := B
(A, b): $\{6, 2, 4, 5, 3\}$	C	m[A][b] := C
(B, a): $\{1, 4, 5, 3\}$	B	m[B][a] := B
(B, b): $\{6, 2, 4, 5, 3\}$	C	m[B][b] := C
(C, a): $\{1, 4, 5, 3\}$	B	m[C][a] := B
(C, b): $\{6, 2, 4, 5, 3\}$	C	m[C][b] := C

DEA:



Endzustand: C, weil $6 \in C$

3. Syntaktische Analyse

Reguläre Ausdrücke können nicht alle relevanten Konstrukte einer Progr. Sprache beschreiben, Bsp.: Korrekte Klammerung

Alphabet = $\{a, l, r\}$, Sprache = $\{a, lar, llarr, \ldots\}$

Ursache: Keine Information über frühere Zustände des Automaten

Abhilfe: Automat wird mit Stack ausgestattet. Dann können die "kontextfreien Sprachen" erkannt werden.

3.1. Kontextfreie Grammatiken

Bestandteile:

a) Menge von Terminalsymbolen T (das sind die Tokens aus Kap2)

b) Menge von Nichtterminalsymbolen N ("Variablen")

c) Menge von Produktionen der Form $l \to r$, wobei $l \in N$ und $r \in (T \cup N)^*$ ("String von Grammatiksymbolen")

d) Ein ausgezeichnetes $S \in N$, das "Startsymbol".

Bsp.: Korrekte Klammerung

$T = \{a, l, r\}$, $N = \{S\}$

Produktionen: $S \to a$

$S \to lSr$

Abkürzung dafür: $S \to a \mid lSr$

Bem.: Jede reguläre Sprache ist eine kontextfreie Sprache.

a	$S \to a$
ε	$S \to \varepsilon$

$\beta \mid \gamma$	$S \to \beta \mid \gamma$
$\beta\gamma$	$S \to \beta\gamma$
$\beta*$	$S \to \varepsilon \mid \beta S$ ("rechtsrekursiv")
	alternativ: $S \to \varepsilon \mid S\beta$ ("linksrekursiv")

Also: Reguläre Sprachen $\subsetneq$ Kontextfreie Sprachen

Warum dann überhaupt Scanner? Effizienz, Einfachheit!

3.2. Ableitungen, Parse-Bäume, Mehrdeutigkeiten

Grammatik für arithmet. Ausdrücke:

$E \to E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid id$

Ist $-(id + id)$ ein Satz der Grammatik?

Ja, denn es gibt eine "Ableitung": $E \Rightarrow -E \Rightarrow -(E)$

$\Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$

In jedem Herleitungsschritt zwei Entscheidungen:

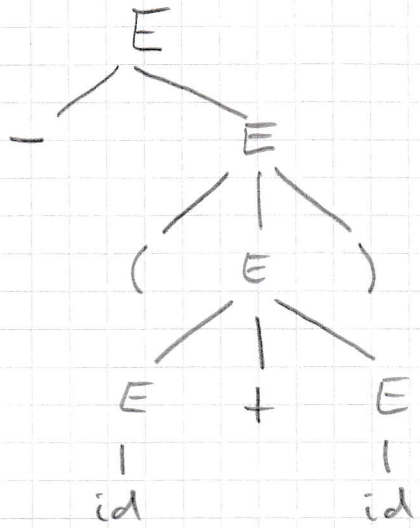
a) Welches Nichtterminal wird ersetzt?

b) Welche Alternative für dieses Nichtterminal wird gewählt?

Linksableitungen ersetzen das jeweils am weitesten links stehende Nichtterminalsymbol, Rechtsableitungen das am weitesten rechts stehende Nichtterminalsymbol.

Parse-Baum: grafische Darstellung einer Ableitung ohne Berücksichtigung der Reihenfolge.

Bsp.: $-(id + id)$

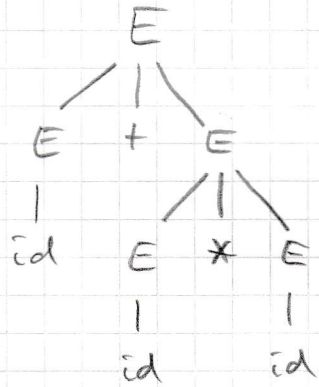


Syntaxanalyse = Finden einer Ableitung = Konstr. d. Parse-Baums bei gegeb. Satz

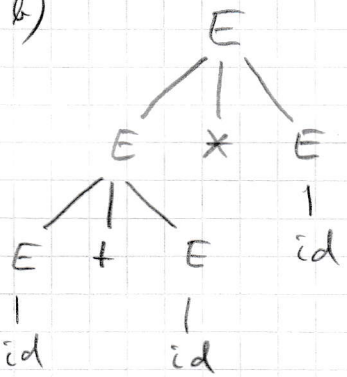
Eine Grammatik, bei der es mind. einen Satz gibt, der durch verschiedene Parse-Bäume repräsentiert wird, heißt mehrdeutig.

Bsp.: Grammatik von oben, Satz id + id * id

a)



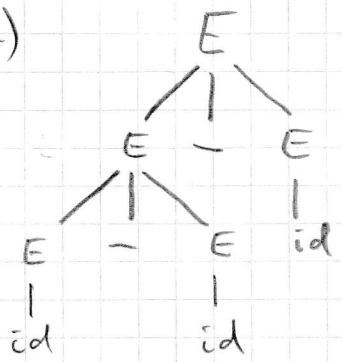
b)



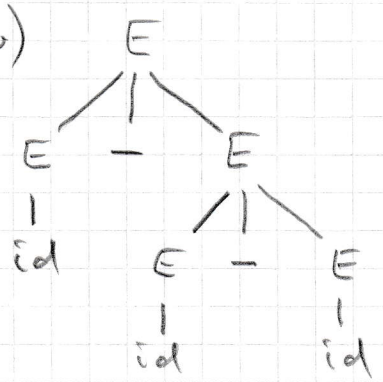
Wir wollen haben: a), wg. "Priorität"

Bsp.: Grammatik von oben, Satz id - id - id

a)



b)



Wir wollen haben: a), wg. "Assoziativität"

Grammatik wird eindeutig durch neue Nichtterminale:

$E \to E + T \mid E - T \mid T$ — E: "Expression"

$T \to T * F \mid T / F \mid F$ — T: "Term"

$F \to -F \mid id \mid (E)$ — F: "Faktor"

Bem.: Priorität durch E/T/F, Assoziativität durch (Links-) Rekursion (Links-)

Die beiden Beispiele behandeln!

3.3. Prädiktive Parser (Top-Down-Parser, LL(1)-Parser)

Verfahren: "Rekursiver Abstieg" – jedes Nichtterminalsymbol wird durch eine rekursive Prozedur dargestellt, jede Produktion ist ein Zweig in der betr. Prozedur

Eigenschaften: leicht zu programmieren, wenig mächtig

Bsp.: $S \to$ if E then S

$S \to$ while E do S

Annahme: in t steht das jeweils nächste Token

```
void S(void) {
  switch (t) {
    case IF:  t = getToken();
              E();
              if (t ≠ THEN) error(...);
              t = getToken();
              S();
              break;
    case WHILE: :
  }
}
```

Aber: $E \to E+T \mid E-T \mid T$ läßt sich so nicht parsen! Warum? Erstes Terminalsymbol muß genug Information liefern, um die richtige Produktion auszuwählen!

3.3.1. First- und Follow-Mengen

Sei $\beta \in (T \cup N)^*$ eine bel. Folge von Grammatiksymbolen.

Def.: $\text{FIRST}(\beta) = \{t \in T \mid \beta \Rightarrow \ldots \Rightarrow t\gamma\}$

d.h. diejenigen Terminalsymbole, mit denen ein aus β abgeleiteter Satz beginnen kann. Gilt $\beta \Rightarrow \ldots \Rightarrow \varepsilon$, dann ist (auch) $\varepsilon \in \text{FIRST}(\beta)$.

1. Forderung für Top-Down-Parser:

Fasse alle Produktionen für ein Nichtterminalsymbol A zusammen zu $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$. Dann muß

$\text{FIRST}(\beta_i) \cap \text{FIRST}(\beta_j) = \{\}$ f. alle $i \neq j$ sein.

Das reicht nicht: Falls $\beta \Rightarrow \ldots \Rightarrow \varepsilon$, dann kommt's darauf an, was dem β nachfolgt!

Sei $A \in N$ ein Nichtterminalsymbol und S das Startsymbol.

Def.: $\text{FOLLOW}(A) = \{t \in T \mid S \Rightarrow \ldots \Rightarrow \beta A t \gamma\}$

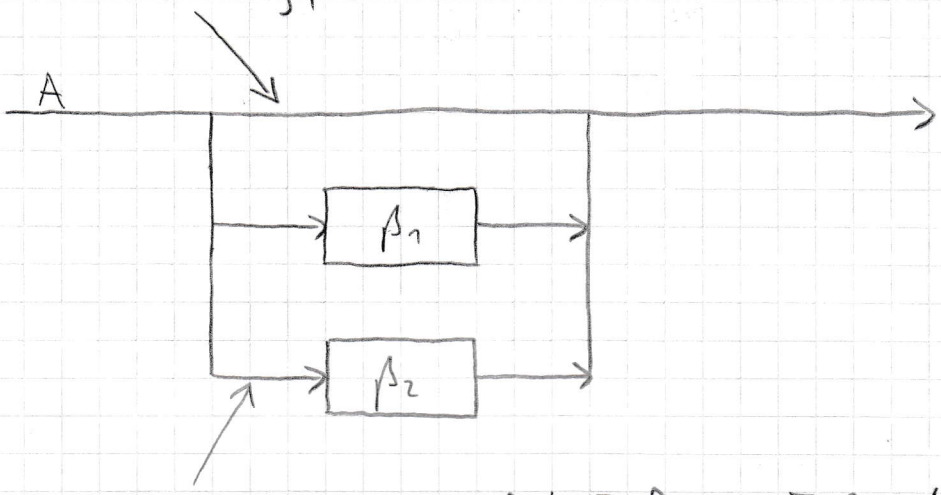
d.h. diejenigen Terminalsymbole, die in einer Satzform direkt rechts neben A stehen können.

2. Forderung für Top-Down-Parser:

Für jedes Nichtterminalsymbol A mit $A \Rightarrow \ldots \Rightarrow \varepsilon$ muß

$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \{\}$ sein.

Anschaulich in Syntax-Diagrammen:



Grammatiken, die beide Forderungen erfüllen, heißen LL(1).

1. "L" ≙ Eingabe wird von links nach rechts gelesen

2. "L" ≙ Beim Parsen wird eine Links-Ableitung erzeugt

"(1)" ≙ 1 Token Vorausschau ("Lookahead")

Eine mehrdeutige oder linksrekursive Grammatik ist nicht LL(1).

3.3.2. Elimination von Links-Rekursion, Links-Faktorisierung

a) Transformation von $A \to A\beta \mid \gamma$ zu:

$$A \to \gamma A'$$

$$A' \to \beta A' \mid \varepsilon$$

Bsp.: $E \to E+T \mid T$ umformen zu:

$$E \to T E'$$

$$E' \to +T E' \mid \varepsilon$$

b) Transformation von $A \to \beta\gamma_1 \mid \beta\gamma_2$ zu:

$$A \to \beta A'$$

$$A' \to \gamma_1 \mid \gamma_2$$

3.3.3. Wo ist beim Top-Down-Parser mit rek. Abstieg der Stack? →

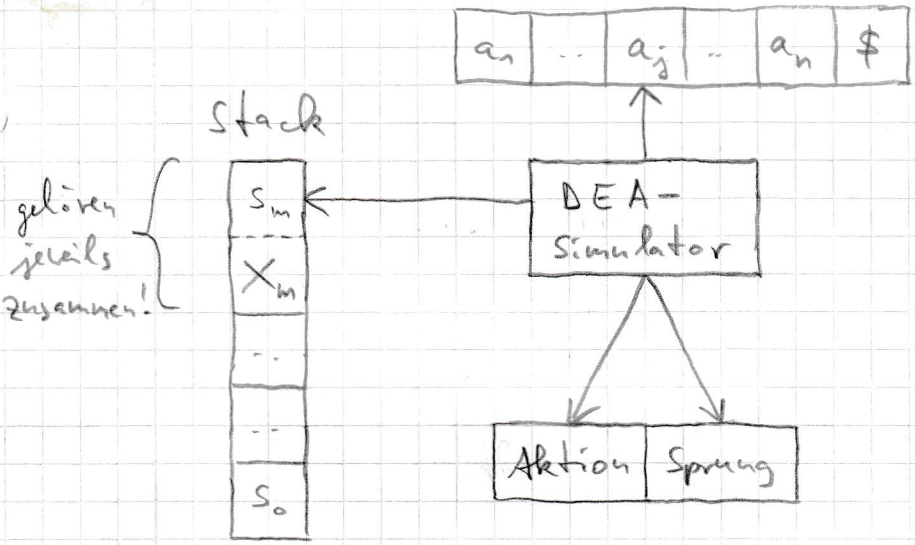
→ Das ist der Laufzeit-Stack der Programmiersprache, in der der Compiler geschrieben ist!

3.4. Shift-Reduce-Parser (Bottom-Up-Parser, LR(1)-Parser)

Schwäche beim LL(1)-Parser: Entscheidung, welche Produktion benutzt wird, fällt aufgrund eines Tokens!

Idee: Zurückstellung dieser Entscheidung, bis die gesamte rechte Seite der Produktion gelesen wurde (und noch ein Token mehr).

3.4.1. Tabellengesteuerter Parser



($: Ende der Eingabe,
a_j: Eingabe-Token,
s_m: Zustand,
X_m: Grammatiksymbol)

DEA-Simulator:

```
push(s0);
accept := false;
while (!accept) {
    sei s der Zustand oben auf dem Stack;
    sei a das nächste Token;
    switch (aktion[s][a]) {
        case shift(n):
            push(a,n);
            getToken();
            break;
```

```
case reduce(m):
    sei A → β die Produktion mit der Nummer m;
    sei k die Anzahl der Grammatiksymbole von β;
    pop(k Elemente);
    sei t der Zustand, der jetzt oben auf dem Stack ist;
    push(A, sprung[t][A]);
    break;
case accept:
    accept := true;
    break;
case error:
    error();
    break;
}
}
```

Bsp.:

r0: S → E $ r2: E → T

r1: E → T + E r3: T → a

	Aktion			Sprung	
Zustand	a	+	$	E	T
1	s5			2	3
2			acc		
3		s4	r2		
4	s5			6	3
5		r3	r3		
6			r1		

leer ≙ error

Stack	Eingabe	Aktion / Sprung
1	a + a $	s 5
1 a 5	+ a $	r 3 → 3
1 T 3	+ a $	s 4
1 T 3 + 4	a $	s 5
1 T 3 + 4 a 5	$	r 3 → 3
1 T 3 + 4 T 3	$	r 2 → 6
1 T 3 + 4 E 6	$	r 1 → 2
1 E 2	$	acc

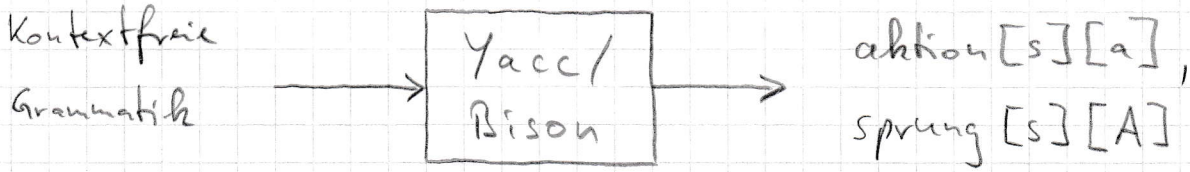
Bem.:

a) Die Grammatiksymbole (a, T, +, E) müssen nicht wirklich auf dem Stack stehen; der Zustand genügt!

b) LR(1) bedeutet: Eingabe von links nach rechts lesen, Rechtsableitung konstruieren, mit einem Token Vorausschau.
Stack + Eingabe = Rechtsableitung in umgekehrter Reihenfolge:
E $ ⇒ T + E $ ⇒ T + T $ ⇒ T + a $ ⇒ a + a $

c) Die Tabelleneinträge bestimmen sowohl die erkannte Sprache als auch die Parsing-Methode (LR(0), SLR(1), LR(1), LALR(1))

3.4.2. Parser-Generatoren



Weiteres Vorgehen wie bei Scanner-Generatoren.

Bsp.:

```
%token      PLUS  IDENT
%start      expr
%%
expr        :  term PLUS expr
            |  term
            ;
term        ;  IDENT
            ;
%%
```

3.4.3. Konstruktion von LR(0)-Tabellen

LR(0): Schiebe/Reduziere-Entscheidung ohne Voraussicht.

Def.: Eine Produktion $A \to \beta$ mit einem Punkt irgendwo auf der rechten Seite heißt "LR(0)-Element" (kurz: Element).

Bsp.: Die Produktion $A \to aBc$ bringt 4 Elemente hervor:

$A \to .aBc$ $\quad$ $A \to a.Bc$ $\quad$ $A \to aB.c$ $\quad$ $A \to aBc.$

Def.: Eine Menge von Elementen heißt "Zustand" des DEA.

Algorithmus zur Tabellenkonstruktion benutzt zwei Hilfsfunktionen.

Sei I eine Menge von Elementen und X ein Grammatiksymbol.

a) Hülle (I) {

do {

Der Punkt kennzeichnet den Stand der Analyse.

```
        for (jedes Element A → α.Xβ in I) {
            for (jede Produktion X → γ) {
                I := I ∪ {X →.γ}
            }
        }
    } while (I hat sich in dieser Iteration geändert);
    return I;
}
```

b)
```
Sprung(I, X) {
    J := {};
    for (jedes Element A → α.Xβ in I) {
        J := J ∪ {A → αX.β};
    }
    return Hülle(J);
}
```

Tabellenkonstruktion:

Ergänze die Grammatik um die neue Startproduktion $S' \to S\,\$$. Sei T die Menge der Zustände des DEA und E die Menge seiner Kanten.

```
T := {Hülle({S' →. S$})};
E := {};
do {
    for (jeden Zustand I in T) {
        for (jedes Element A → α.Xβ in I) {
            J := Sprung(I, X);
```

```
        T := T ∪ { J } ;
        E := E ∪ { I -X-> J } ;
    }
}
} while (T oder E haben sich in dieser Iteration geändert);
```

Aber: Für das Symbol $ wird kein Sprung berechnet;
jeder Zustand mit Element S' → S. $ akzeptiert bei Token = $!

Berechnung der Menge der Reduzier-Aktionen R:

```
R := {} ;
for (jeder Zustand I in T) {
    for (jedes Element A → α. in I) {
        R := R ∪ { (I, A → α) } ;
    }
}
```

(I: Zustand; A → α: Reduktion mit dieser Produktion)

Bsp.:

r0: S → E $ r2: E → T

r1: E → T + E r3: T → a

① (Startzustand) = Hülle ({ S → . E $ })

Elemente:
- S → . E $
- E → . T + E
- E → . T
- T → . a

} ①

Füllen der Tabellen: ⟶

a) Für jede Kante $I \xrightarrow{X} J$:

Falls X ein Terminalsymbol ist,

setze aktion[I][X] := shift J

Falls X ein Nichtterminalsymbol ist,

setze sprung[I][X] := J

b) Für jeden Zustand I, der das Element
$S' \to S.\$$ enthält, setze aktion[I][$] := accept.

c) Für jeden Zustand I, der ein Element
$A \to \alpha.$ enthält, setze aktion[I][y] := reduce n
für alle Token y ($\in R(\Phi)$, keine Voraussicht!),
wobei n die Produktion ist, die $A \to \alpha.$ hervorgebracht
hat.

d) Setze alle anderen Einträge auf error.

① $\xrightarrow{E}$ Sprung(①, E): $S \to E.\$$ } ②

① $\xrightarrow{T}$ Sprung(①, T): $E \to T.+E$, $E \to T.$ } ③

③ $\xrightarrow{+}$ Sprung(③, +): $E \to T+.E$, $E \to .T+E$, $E \to .T$, $T \to .a$ } ④

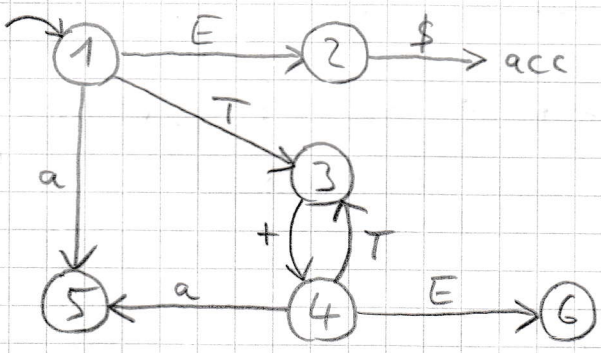
① $\xrightarrow{a}$ Sprung(①, a): $T \to a.$ } ⑤

④ $\xrightarrow{E}$ Sprung(④, E): $E \to T+E.$ } ⑥

④ $\xrightarrow{T}$ Sprung(④, T): $E \to T.+E$, $E \to T.$ } ③

④ $\xrightarrow{a}$ Sprung(④, a): $T \to a.$ } ⑤

Automat:



Reduzier-Aktionen:

(③, $E \to T$)

(⑤, $T \to a$)

(⑥, $E \to T+E$)

LR(0) - Tabelle:

	Aktion			Sprung	
Zustand	a	+	$	E	T
1	s5			2	3
2			acc		
3	r2	s4, r2	r2		
4	s5			6	3
5	r3	r3	r3		
6	r1	r1	r1		

(3), + : "Schiebe-Reduziere-Konflikt"

Grammatik ist nicht LR(0)!

Bem.: Parsergeneratoren lösen solche Konflikte nach zwei Regeln auf:

a) Bei Schiebe-Reduziere-Konflikten wird immer geschoben. Das ist in seltenen Fällen akzeptabel ~~Reduktion~~.

b) Bei Reduziere-Reduziere-Konflikten hat die zuerst hingeschriebene Produktion Vorrang. Das ist praktisch immer unbrauchbar!

Konsequenz: Bei Konflikten ist der Automat zu analysieren und die Grammatik umzuformen!

3.4.4. Konstruktion von SLR(1)-Tabellen

Der Konflikt in der LR(0)-Tabelle kann leicht aufgelöst werden: Reduktion mit $A \to \gamma$ nur dann, wenn nächstes Token $\in$ FOLLOW(A) ist!

Bsp.: Das "hängende else" ist eine Mehrdeutigkeit, die sich beim if-Statement ergeben kann:

```
if_stm  : IF LPAREN exp RPAREN stm
        | IF LPAREN exp RPAREN stm ELSE stm
        ;
```

Der Parsergenerator meldet einen Schiebe/Reduziere-Konflikt in einem der Zustände des Parsers, wenn das Lookahead-Token ELSE ist:

```
if_stm → IF LPAREN exp RPAREN stm .
if_stm → IF LPAREN exp RPAREN stm . ELSE stm
```

Warum schiebt er bei ELSE nicht einfach?

```
if (x < 5)
    if (y > 10)
        :
    else
        :
```

Wozu gehört das else ...?

Schieben → else gehört zum inneren if

reduzieren → " " " äußeren "

```
R := {};
for (jeden Zustand I in T) {
    for (jedes Element A → α. in I) {
        for (jedes Token X in FOLLOW(A) {
            R := R ∪ {(I, X, A → α)};
        }
    }
}
```

Zustand | Lookahead | Reduktion mit dieser Produktion

Unser Bsp.: Automat bleibt derselbe, aber Tabelle enthält weniger Reduktionen:

FOLLOW(E) = {$}

FOLLOW(T) = {+, $}

	Aktion			Sprung	
Zustand	a	+	$	E	T
1	s5			2	3
2			acc		
3		s4	r2		
4	s5			6	3
5		r3	r3		
6			r1		

3.4.5. Konstruktion von LR(1)- bzw. LALR(1)-Tabellen

LR(1)-Elemente sind Paare: (LR(0)-Element, Token)

Mehr Information in den Zuständen des DEA ⇒ viel mehr Zustände; weniger Konflikte in der Tabelle.

LALR(1): Wie LR(1), aber Zusammenfassen der Zustände, die sich nur im Token unterscheiden: ~ Faktor 10 weniger Zustände als LR(1)!

4. Abstrakte Syntax

Compiler soll Code (oder Datenstruktur) erzeugen: Verknüpfung von Aktionen mit Grammatik-Produktionen notwendig.

4.1. Semantische Aktionen in LL-Parsern

Einbau von Aktionen in die Parser-Routinen. Bsp.:

$E \to TE'$ $\quad E' \to +TE' \mid \varepsilon$ $\quad T \to n$

```
void E(void) {
    T();
    E'();
}

void E'(void) {
    if (token == PLUS) {
        getToken();
        T();
        printf("add");
        E'();
    }
}

void T(void) {
    if (token == NUM) {
        printf("push %d", token.value);
        getToken();
    } else {
        error("Zahl erwartet");
    }
}
```

Eingabe: 1 + 2 + 3

Ausgabe:
```
push 1
push 2
add
push 3
add
```

4.2. Semantische Aktionen in LR-Parsern

Einbau von Aktionen in die Reduktionen. Bsp.:

$E \to E+T \mid T \qquad T \to n$

```
E : E + T
    { printf("add"); }
  | T
  ;
T : n
    { printf("push %d", n.value); }
  ;
```

Eingabe / Ausgabe wie oben! Warum?

Rechtsableitung: $E \Rightarrow E+T \Rightarrow E+3 \Rightarrow E+T+3 \Rightarrow E+2+3 \Rightarrow T+2+3 \Rightarrow 1+2+3$. Im umgekehrter Reihenfolge:

$T \to 1$	push 1
$E \to T$	
$T \to 2$	push 2
$E \to E+T$	add
$T \to 3$	push 3
$E \to E+T$	add

Möglichkeit: ganzer Compiler eingebettet in semantischen Aktionen. Nachteile: schwer zu lesen und zu warten (syntakt. Analyse, semant. Analyse und Codeerzeugung miteinander verwoben), Informationsfluß ist auf die Parse-Reihenfolge beschränkt. Besser: Erzeugen einer Datenstruktur ("Abstrakte Syntax").

4.3. Semantische Werte in LL-Parsern

Jede Parser-Routine gibt den "semantischen Wert" (z.B. Baumknoten) des von ihr erkannten Nichtterminals zurück. Bsp.:

```
ifstm -> IF cond THEN stm

Node * ifstm(void) {
    Node *c, *s;
    if (token == IF) getToken(); else error();
    c = cond();
    if (token == THEN) getToken(); else error();
    s = stm();
    return newNode(IFSTM, c, s);
}
```

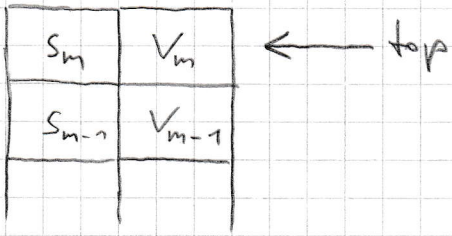
4.4. Semantische Werte in LR-Parsern

Jede Reduktion produziert den semantischen Wert ihrer linken Seite. Bsp.:

```
ifstm : IF cond THEN stm
        { $$ = newNode(IFSTM, $2, $4); }
      ;
```

Technische Realisierung: "Semantik-Stack"

parallel zum Zustands-Stack.

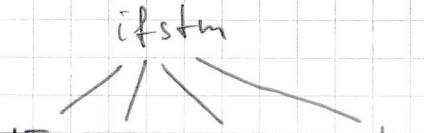


4.5. Abstrakte Syntax

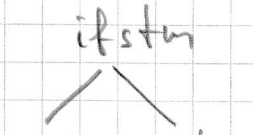
Zweck: sauberes Interface zwischen Parser und späteren Phasen.

Entsteht aus Parse-Baum durch Weglassen unwichtiger Details.

Bsp.: a) Parse-Baum



Syntax-Baum



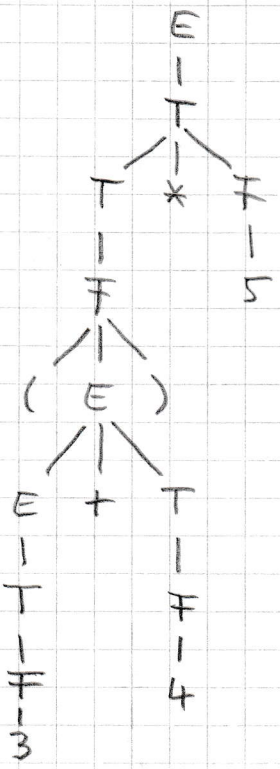
b) $E \to E + T \mid T$

$T \to T * F \mid F$

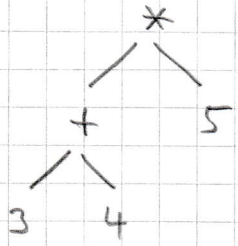
$F \to n \mid (E)$

Eingabe $(3+4)*5$

Parse-Baum



Syntax-Baum

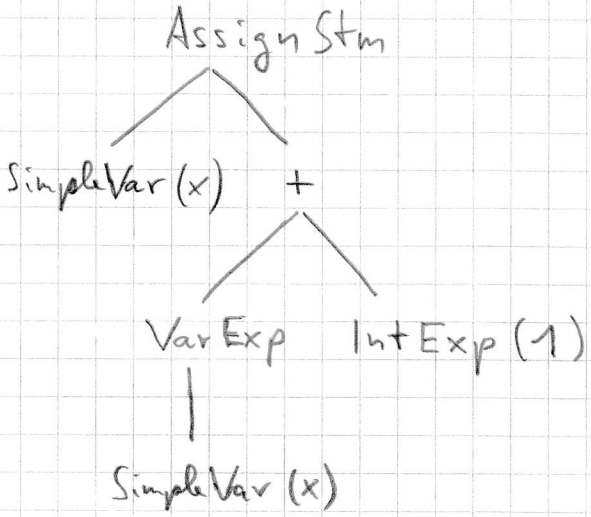


Erreicht wird das durch z.B.:

```
T : F
    { $$ = $1; }
  ;
```

(ist häufig die "Default-Aktion" im Parser-Generator)

c) x := x + 1

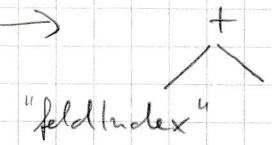


SimpleVar $\hat{=}$ Berechnung d. Adresse einer einfachen Var.

VarExp $\hat{=}$ Berechnung d. Inhalts aus einer Adresse

4.5.1. Effiziente Handhabung von Bezeichnern

feldIndex + 5 ⟶



Das ist ganz ineffizient, da bei jedem Durchgang durch diesen Knoten ein Stringvergleich stattfinden muß! Besser: Darstellung des Bezeichners durch "Symbol" (= eindeutige Zahl bzw. eindeutiger Zeiger). Funktionen d. Symbolverwaltung:

Erzeugen: String → Symbol

ext. Darstellung: Symbol → String

Anordnung: Symbol1 < Symbol2 ?

Technische Realisierung durch Hashtabelle.

4.6. Attributierung

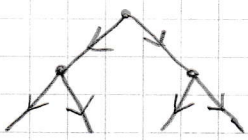
Bedeutet: Anheften von Eigenschaften (Typ, Größe, Speicherort, etc.) an die Knoten des Syntax-Baumes.

Realisierung: durch rekursive Attribut-Auswerter

"Synthetisierte Attribute" (Blätter → Wurzel):

Bsp.: Auswertung konstanter Ausdrücke

"Vererbte Attribute" (Wurzel → Blätter):



Bsp.: char *p1, *p2;

Im Allgemeinen: Attribute beider Sorten, berechnet in mögl. mehreren Durchläufen durch den Baum

5. Semantische Analyse

Aufgabe: Sammeln von Informationen über Bezeichner, Überprüfen von Einschränkungen bei der Benutzung von Bezeichnern ("Typprüfungen"). ← {, Literalen und Ausdrücken

5.1. Typen

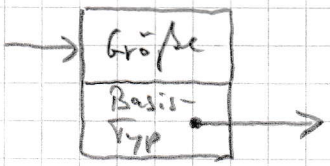
Typ: Information (zur Übersetzungszeit), wie Bitmuster zu interpretieren sind (zur Laufzeit)

5.1.1. Darstellung von Typen

Typen können Bezug auf andere Typen nehmen (Bsp.: Array)
⇒ Darstellung durch "Typgraph"

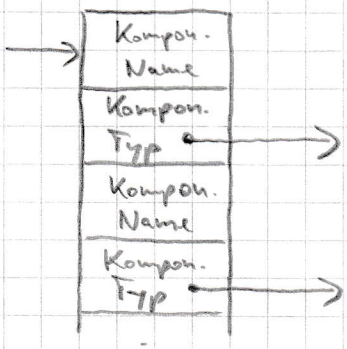
a) Primitive ("eingebaute") Typen:

b) Arrays:

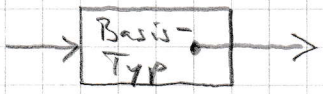


Bem.: Manche Sprachen betrachten die Größe als nicht zum Typ gehörend!

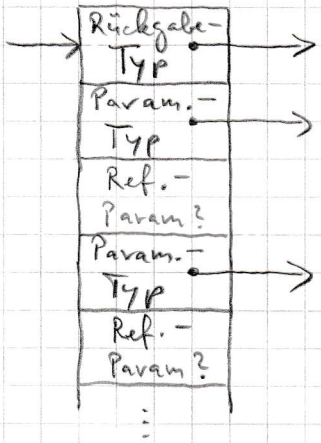
c) Records:



d) Pointer

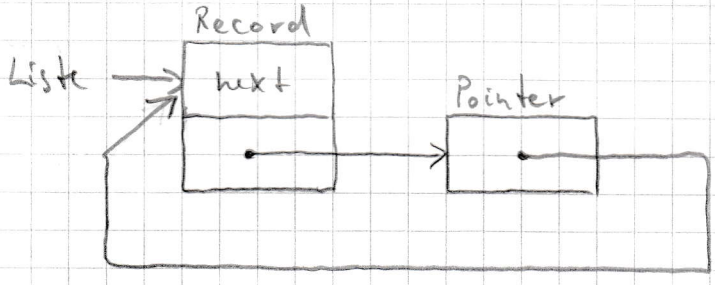


e) Funktionen



Achtung: Typgraphen können Zyklen enthalten:

```
typedef struct liste {
        struct liste *next;
} Liste;
```



5.1.2. Gleichheit von Typen

a) Namensgleichheit: Zwei Typen sind genau dann gleich, wenn sie denselben Namen haben.

Bsp:
```
type a = array [3] of int;
type b = a;
type c = array [3] of int;
```

Alle 3 Typen sind verschieden.

b) Ausdrucksgleichheit: Zwei Typen sind genau dann gleich, wenn sie durch denselben Typausdruck konstruiert wurden (d.h. durch denselben Typgraphen repräsentiert werden).

Bsp.: Typdefinitionen wie oben. Die Typen a und b sind gleich, aber verschieden von c. (So macht es SPL.)

c) Strukturgleichheit: Zwei Typen sind genau dann gleich, wenn ihre Typgraphen (rekursiv) die gleiche Struktur aufweisen.

Bsp.: Typdefinitionen wie oben. Alle 3 Typen sind gleich.

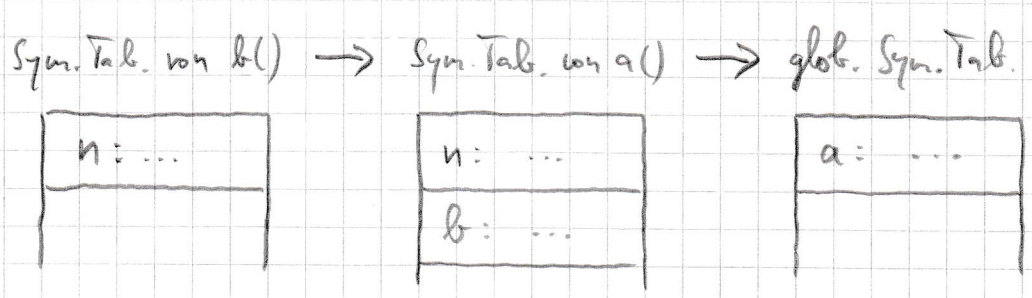
5.2. Symboltabellen

Realisieren Abbildungen: Bezeichner ↦ Attribute
(bzw. wegen Effizienz: Symbol ↦ Attribute)

5.2.1. Mehrstufige Symboltabellen

Geschachtelte Gültigkeitsbereiche von Bezeichnern ⇒ verschiedene Attributsätze für den gleichen Bezeichner ⇒ mehrstufige Symboltabelle.

Bsp.:
```
proc a () {
    var n: int;
    proc b () {
        var n: array [10] of int;
        :
    }
}
```

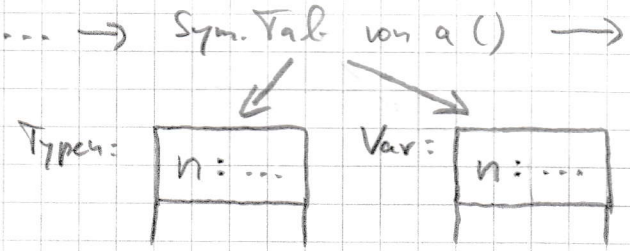


(SPL benutzt zweistufige Symboltabellen.)

5.2.2. Parallele Symboltabellen

Mehrere Namensräume ⇒ verschiedene Attributsätze für den gleichen Bezeichner (auf derselben Stufe!) ⇒ parallele Symboltabellen (heißen zusammen: "Environment")

Bsp.:
```
proc a() {
    type n = array [10] of int;
    var n: n;
}
```



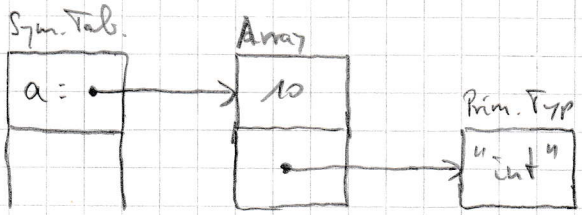
einer Stufe

(SPL hat einen gemeinsamen Namensraum f. alle Bezeichner.)

5.2.3. Attribute für Bezeichner in SPL

a) Typ-Bezeichner ⟼ Typ

Bsp.: `type a = array [10] of int;`



b) Variablen-Bezeichner ⟼ (Typ, ist Referenz)

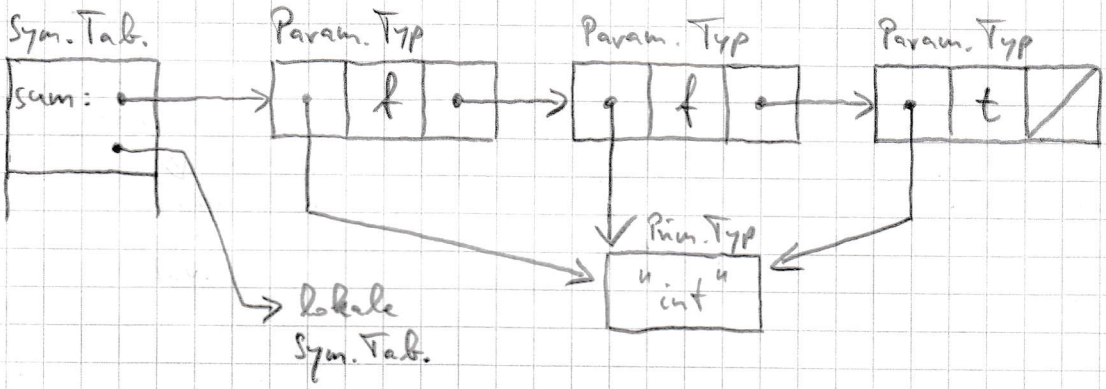
lokale Variablen : ist Ref. = false

Wertparameter : ist Ref. = false

Referentparameter : ist Ref. = true

c) Prozedurbezeichner ⟼ (Parameter-Typen, lokale Sym. Tab.)

Bsp.: proc sum (i: int, j: int, ref k: int) { ... }



5.3. Type-Checking

Der Type-Checker ist eine rekursive Funktion, die die Abstrakte Syntax inspiziert und einen Typ zurückgibt:

```
Type *checkNode (Absyn *node, Table *symTable) {
    switch (node->type) {
        case ABSYN_NAMETYP:
            return checkNametype (node, symTable);
        case ABSYN_PROCDECL:
            return checkProcdecl (node, symTable);
        case ABSYN_OP:
            return checkOp (node, symTable);
            :
    }
}
```

5.3.1. Type-checking von Ausdrücken

a) Binäre Operatoren

```
Type *checkOp(Absyn *node, Table *symTable) {
    Type *leftType, *rightType, *type;
    leftType = checkNode(node->u.opNode.left, symTable);
    rightType =  "      "      "    .right,  "      );
    if (leftType != rightType) error(...);
    switch(node->u.opNode.op) {
        :
      case ABSYN_OP_ADD:
        if(leftType != intType) error(...);
        type = intType;
        break;
        :
    }
    return type;
}
```

b) Einfache Variable

```
Type *checkSimpleVar(Absyn *node, Table *symTable) {
    Entry *entry;
    entry = lookup(symTable, node->u.simplevarNode.name);
    if (entry == NULL) error(...);
    if (entry->kind != ENTRY_KIND_VAR) error(...);
    return entry->u.varEntry.type;
}
```

c) Array-Variable

- überprüfen, daß wirklich ein Array indiziert wird
- überprüfen, daß Index von Typ Integer ist
- Basistyp zurückgeben! Bsp.: type A = array [5] of int; var a: A; ⇒ Typ von a ist A, Typ von a[2] ist int

Bem.: Es ist eine gute Idee, bei Ausdrücken den Typ in der Abstrakten Syntax als Attribut festzuhalten.

5.3.2. Type-Checking von Anweisungen

Liefern grundsätzlich den Typ "void" zurück (kann bei uns durch NULL repräsentiert werden).

a) Zuweisungen

- überprüfen, daß Typ(lhs) und Typ(rhs) "zuweisungskompatibel" (bei uns: gleich) sind

b) If- und While-Statements

- überprüfen, daß die Tests Boole'sche Werte liefern
- Rekursiven Abstieg in die Teilanweisungen (then-Teil, else-Teil, Schleifenrumpf) nicht vergessen!

c) Prozeduraufrufe

- Gibt's den Bezeichner? Ist es eine Prozedur?
- Stimmen die aktuellen Argumente in Zahl und Typ mit den formalen Parametern überein?
- Ist das aktuelle Argument eines Referenzparameters eine Variable? (Andere Ausdrücke sind hier verboten!)

5.3.3. Type-Checking von Typen

Das ist einfach: entspr. Typgraphen zurückgeben!

a) Benannte Typen:
   - In Symboltabelle nachschauen (vorhanden?, Typ?)
   - Typ zurückgeben

b) Array-Typen:
   - Basistyp ermitteln (Rekursion!)
   - Array-Typ konstruieren und zurückgeben

5.3.4. Type-Checking von Deklarationen

Deklarationen bewirken Einträge in die Symboltabelle.

a) Typ-Deklarationen

```
Type *checkTypedecl (Absyn *node, Table *symTable) {
    Type *type;
    Entry *entry;
    type = checkNode (node->u.typedeclNode.type, symTable);
    entry = newTypeEntry (type);
    if (enter (symTable, node->u.typedeclNode.name,
               entry) == NULL) error (...);
    return NULL;
}
```

b) Variablen-Deklarationen

Wie a), aber mit newVarEntry (type, FALSE)
(≙ keine Referenz-Variable)

c) Prozedur-Deklarationen

- Parameter-Typen ermitteln
- Prozedur in globale Symboltabelle eintragen
- Parameter in lokale Symboltabelle eintragen
- lokale Deklarationen bearbeiten
- Prozedur-Rumpf bearbeiten
- ggf. Symboltabellen ausgeben (Debugging!)

5.3.5. Vorwärtsreferenzen und Rekursion

Vorwärtsreferenz: Benutzung eines Bezeichners vor seiner Deklaration. Das ist unvermeidlich der Fall bei wechselseitig rekursiven Typen oder Prozeduren.

Schwierigkeit: Information über den Bezeichner wird benötigt, ist aber (noch) nicht vorhanden.

Lösung: Type-Checking in zwei Durchläufen.

1. Durchlauf: Sammeln von Informationen über Bezeichner aus den "Köpfen"; keine Behandlung der Benutzung von Bezeichnern in den "Rümpfen".

2. Durchlauf: Jetzt sind alle Bezeichner in der Symboltabelle eingetragen → Type-Checking der "Rümpfe".

Bsp: Prozedur-Deklarationen

1. Durchlauf: Type-Checking der Parameter und Eintrag der Prozedur in die globale Symboltabelle (Rumpf der Prozedur wird ignoriert)

2. Durchlauf: alle anderen o.g. Aktionen

Bem.: Bei der Typprüfung von rekursiven Typdefinitionen geht man genauso vor. Meist wird aber zusätzlich das Aufdecken von illegalen Zyklen verlangt. Bsp.:

type a = b; type b = c; type c = a;

5.3.6. Weitere Aufgaben des Type-Checkers

- Erzeugen der Typgraphen für die primitiven und die "eingebauten" Typen (Unterschied?)
- Erzeugen der globalen Symboltabelle
- Eintragen der vordefinierten Bezeichner (Typen, Prozeduren, etc.) in die globale Symboltabelle
- Überprüfen globaler semantischer Bedingungen (z.B. "ist main() vorhanden?")
- ggf. Ausgabe der ~~globalen~~ Symboltabelle(n)

6. Laufzeit-Organisation

Jeder Funktions/Prozeduraufruf hat seinen eigenen Satz von lokalen Variablen (und Parametern). Viele solcher Aufrufe können zur gleichen Zeit existieren (Rekursion!).
In vielen Sprachen (einschließlich SPL) sollen die Speicherplätze für lokale Variablen (und Parameter) automatisch freigegeben werden, wenn die Ausführung die Funktion verläßt ⇒ Stack ist geeignete Datenstruktur hierfür.

Bem.: In Sprachen, die geschachtelte Funktionsdefinitionen und Funktionen als Rückgabewerte zulassen (sog. "Funktionen höherer Ordnung") gelt das nicht! Dort leben lokale Variablen u.U. länger, als der Funktionsaufruf, der sie erzeugt. Bsp.: Funktionsgenerator "Erhöhen um x"

```
int (*f(int x))(int) {
    int g(int y) { return x+y; }
    return g;
}
int (*h)(int) = f(3);   // h == "Erhöhen um 3"
int (*j)(int) = f(4);   // j == "Erhöhen um 4"
int z = h(5);           // z == 8
int w = j(7);           // w == 11
```

Offensichtlich müssen die Speicherplätze für 3 und 4 über den Aufruf von f hinaus erhalten bleiben! ⇒ Stack ungeeignet!

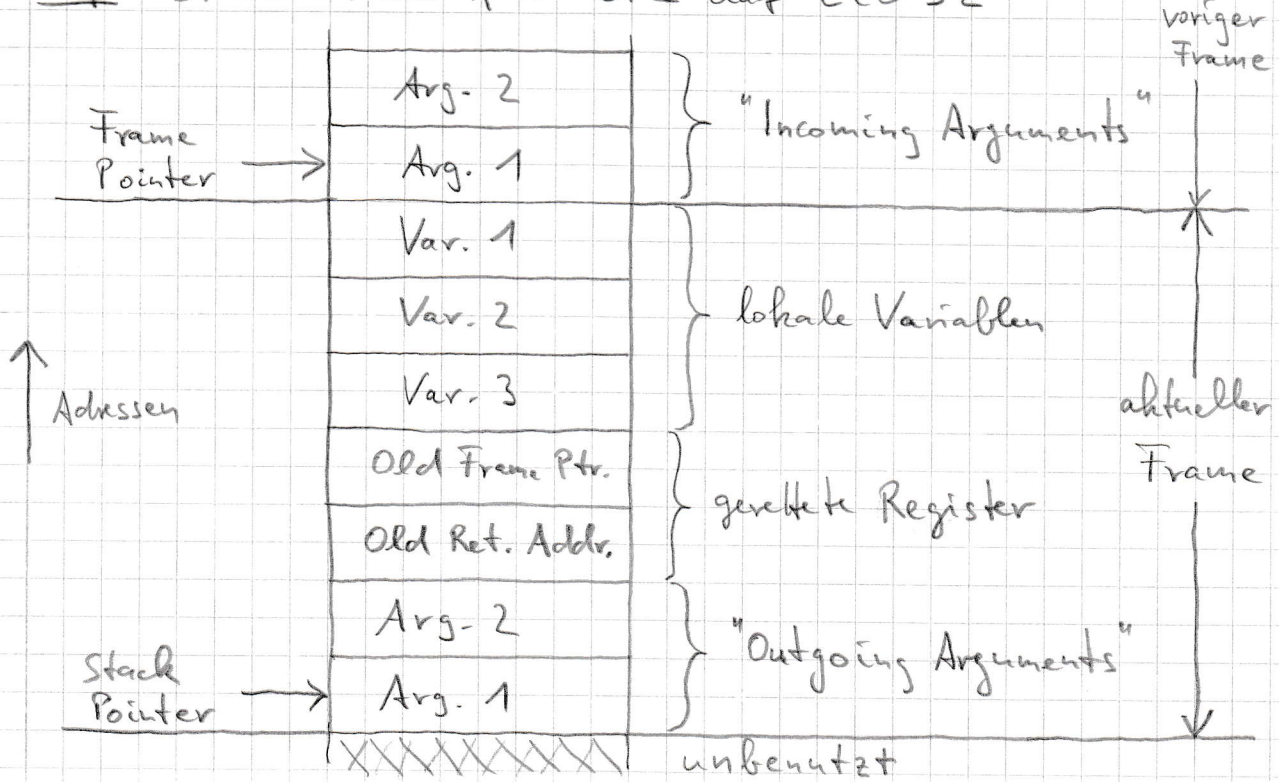
6.1. Stack-Frames

Variablen werden in größerer Zahl angelegt / freigegeben (nicht einzeln) ⇒ push / pop von ganzen Bereichen ("Stack-Frame", "Activation Record"). Die Struktur eines Stack-Frames wird typischerweise nicht von der Hardware vorgegeben, sondern durch Konventionen vereinbart (wichtig für das Binden von Programmen, die in verschiedenen Progr. Sprachen geschrieben sind).

Bestandteile:

- lokale Variablen
- Return-Adresse
- Hilfsvariablen
- gerettete Register
- Argumente f. aufgerufene Prozeduren

Bsp.: Stack-Frame für SPL auf ECO32



Bem.: a) Zur Vereinfachung werden wir alle lokalen Variablen im Stack-Frame speichern (keine in Registern) und alle Hilfsvariablen in Registern (keine im Frame).

b) Der Speicher unterhalb des Stack-Pointers darf nicht benutzt werden (Interrupts können dort hinein schreiben).

c) Die Speicherplätze für "Old Ret. Addr." und "Outgoing Args" werden nur angelegt, falls die aktuelle Prozedur eine andere aufrufen kann (kann – denn ob sie es wirklich tut, ist zur Übersetzungszeit nicht berechenbar).

6.2. Der Frame-Pointer, Prozedur-Eintritt/Austritt

Der Frame-Pointer dient zum Adressieren von Argumenten und lokalen Variablen. Er muß bei Eintritt in eine Prozedur gesetzt und bei Austritt restauriert werden:

```
proc: sp ← sp − framesize
      mem[sp + offset(old fp)] ← fp
      fp ← sp + framesize
        ⋮
(Hier können Argumente mit fp + offset(Argument)
und lokale Variablen mit fp − offset(Variable)
adressiert werden. Argumente für gerufene
Prozeduren werden an sp + offset(Argument) gelegt.)
        ⋮
      fp ← mem[sp + offset(old fp)]
      sp ← sp + framesize
      return
```

Zum Zeitpunkt der Codegenerierung werden also benötigt:

- framesize
- offset (old fp)
- offset (Argument) f. alle eingehenden Argumente
- offset (Variable) f. alle lokalen Variablen
- offset (Argument) f. alle ausgehenden Argumente

Bem.: Der Frame-Pointer muß nicht tatsächlich in einem Register gespeichert werden: fp ≡ sp + framesize. Alle Zugriffe auf Variablen, Parameter etc. können über den sp erfolgen. Warum gibt's dann überhaupt den fp? Der framesize ist normalerweise erst spät im Übersetzungsprozeß bekannt — nachdem man weiß, wieviel Hilfsvariablen gebraucht werden. Deshalb in jedem Falle zunächst: Zugriff auf Variablen etc. über (evtl. fiktiven) fp. Bei SPL auf ECO32: realer fp in einem Register.

6.3. Berechnungen zur Laufzeitorganisation

1. Jeder Typ (nicht nur benannte!) speichert in seinem Typgraphen seine Größe, d.h. die Anzahl Bytes, die ein Objekt von diesem Typ im Speicher belegen wird. Das ist notwendig, um die Offsets (siehe 2.) berechnen zu können. Diese Information entsteht bei der Typgraphen-Konstruktion.

2. Jedes Argument, jeder Parameter und jede lokale Variable speichert den zugehörigen Offset relativ zum Frame-Pointer. Diese Information wird bei einem Durchgang durch die Prozedurdeklarationen gewonnen.

3. Jede Prozedur speichert die Größe von 3 Bereichen:

   a) eingehende Argumente

   b) lokale Variable

   c) ausgehende Argumente

   a) und b) werden bei der Berechnung oben (siehe 2.) mit ausgerechnet.

   c) wird am besten in einem separaten Durchlauf ermittelt: Für jede Prozedur p bestimme das Maximum der eingehenden Argumente der Prozeduren, die p ruft. Dieser Wert ist die Größe des Bereichs der ausgehenden Argumente für p.

   Bem.: Hierbei wird auch bemerkt, ob p überhaupt eine weitere Prozedur aufruft. Diese Information wird ebenfalls festgehalten (bei der Code-erzeugung: Sichern der Return-Adresse sollte ansonsten entfallen).

7. Codegenerierung

7.1. Die Zielmaschine ECO32

a) Allgemeines:

- 32-Bit RISC-Architektur, big-endian
- 32 32-Bit Register $0 .. $31, $0 ≡ 0 durch Hardware
- 64 Instruktionen (wir brauchen ca. 20)
- Kernel/User-Mode (wir benutzen nur Kernel-Mode)
- Speicher: Wort-organisiert, Byte-adressiert, max. 1 GByte, Zugriffe müssen ausgerichtet sein
- I/O: Timer, Terminal, Disk, Grafik-Controller, Memory-mapped, Interrupt- und DMA-fähig (Bedienung wird von den Bibliotheksprozeduren gekapselt)
- Virtuelle Adressierung, 4 KByte Seitengröße, max. 4 GB, TLB-Unterstützung (das benutzen wir nicht: alle Adressen ab 0x C0000000 werden durch die Hardware auf physikal. Adressen ab 0 abgebildet)

b) Instruktionen:

```
add   $4, $17, $8     ; $4 ← $17 + $8
add   $4, $17, 8      ; $4 ← $17 + 8
entspr. auch für sub, mul, div
ldw   $4, $17, 8      ; $4 ← mem[$17 + 8]
stw   $4, $17, 8      ; mem[$17 + 8] ← $4
                      ; $4 → mem[$17 + 8]
```

```
hier:                       ; das ist ein Label
        beq  $4, $17, hier  ; if $4 = $17 : weiter bei "hier"
```

entspr. auch für bne, bgt, bge, blt, ble

(sowie für vorzeichenlose Größen: bgtu, bgeu, bltu, bleu)

```
        j    hier           ; unbedingter Sprung zu "hier"
        jal  hier           ; Prozeduraufruf der Proz. "hier",
                            ; Sichern der Ret. Addr. in $31
        jr   $31            ; weiter bei Inhalt v. $31, "return"
```

c) Pseudo-Instruktionen (= Assembler-Direktiven):

```
.export  hier       ; lokales Label "hier" wird außerhalb
                    ; des Moduls verfügbar
.import  dort       ; externes Label "dort" wird innerhalb
                    ; des Moduls verfügbar
.code               ; die im folgenden erzeugten Bytes
                    ; kommen ins Code-Segment
.align   4          ; Einstellen der Adresse auf einen
                    ; durch 4 teilbaren Wert ("Alignment")
```

d) Register-Konventionen

$0	immer 0 (Hardware)
$31	Return-Adresse (Hardware)
$25	Frame-Pointer
$29	Stack-Pointer
$8 .. $23	Hilfsvariable

Alle anderen Register dürfen nicht benutzt werden!

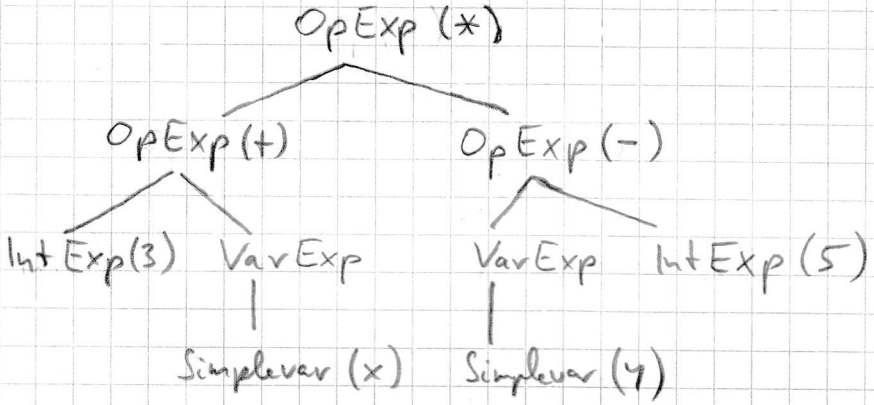
7.2. Code für Stack-Maschinen

Vorteil: einfach zu erzeugen

Nachteil: ineffizienter Code

Methode: Post-Order-Durchgang durch Abstrakte Syntax

Bsp.: (3+x)*(y-5) ← Depth-First-




```
push (3)                push (5)
push (Adr (x))          sub
load                    mul
add
push (Adr (y))
load
```

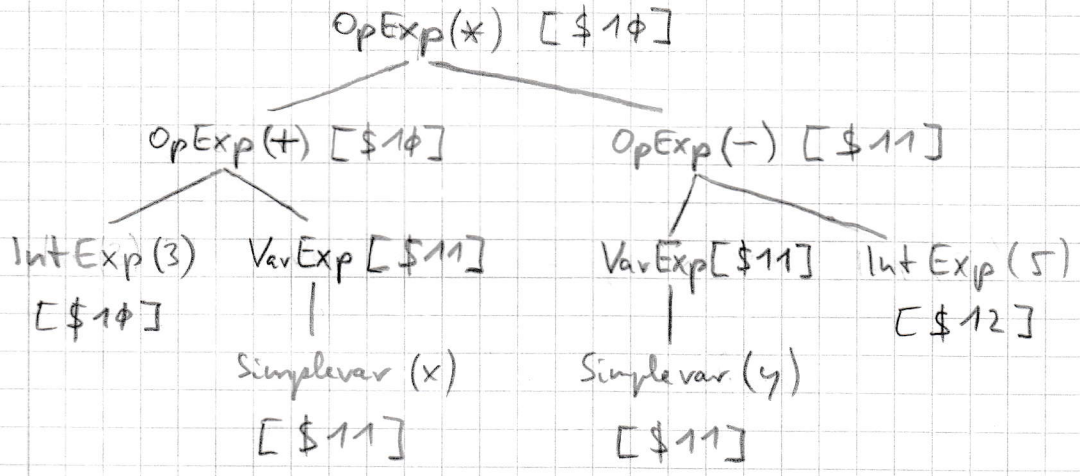
7.3. Code für Register-Maschinen

Wie oben, aber mit Registern anstelle des Stacks:
die Stack-Positionen werden jetzt zu Register-Nummern.

Bsp.: Codeerzeugung für binäre Operationen

```
/* erzeuge Code, so daß Wert von "node" in "target" erscheint */
void genCodeOp (Absyn *node, int target) {
```

```
genCode(node->left, target);
temp = target + 1;
if (temp > maxreg) error("Ausdruck zu Kompliziert");
genCode(node->right, temp);
output("add $%d, $%d, $%d \n",
       target, target, temp);
}
```




```
add   $10, $0, 3
add   $11, $25, offset(x)
ldw   $11, $11, 0
add   $10, $10, $11
add   $11, $25, offset(y)
ldw   $11, $11, 0
add   $12, $0, 5
sub   $11, $11, $12
mul   $10, $10, $11
```

kann zusammengefasst werden:
`ldw $11, $25, offset(x)`

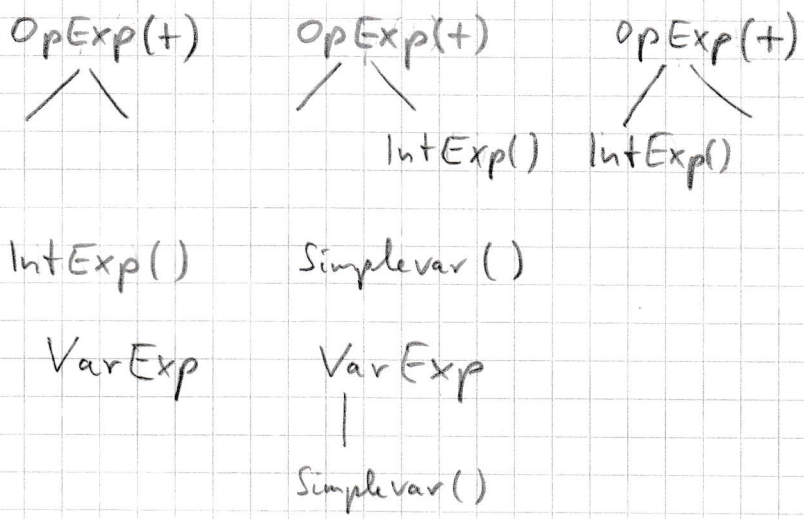
entspr.

`sub $11, $11, 5`

Bem.:

a) Das "Zusammenfassen" von Instruktionen kann folgendermaßen erreicht werden (Algorithmus heißt "Maximal Munch"):

Jede Instruktion der Zielmaschine erzeugt eine oder mehrere "Kacheln", je nach möglichen Argumenten:



Dann wird der ganze Baum der Abstrakten Syntax mit Kacheln belegt, ausgehend von der Wurzel, mit möglichst großen Kacheln. Das ist nicht immer das Optimum!

b) Eine Verbesserung läßt sich mit "Dynamischer Programmierung" erzielen.(*) Weiterer Vorteil: Algorithmus kann in ein Tool integriert werden ("Codegenerator-Generator"). ↑ das diese Optimierung beim Bau des Compilers durchführt

c) Der o.a. Algorithmus zur Allokation von Registern ist nicht immer optimal (→ "Sethi-Ullman-Algorithmus"; behandelt auch das Auslagern / Wiedereinlagern von Werten für temporäre Variable in den Stackframe, falls nicht genügend Register verfügbar sind), das sog. "Spilling"

(*) Das ist ein Algorithmus, der die optimale Lösung eines Problems aus den optimalen Lösungen der Subprobleme berechnet.

7.4. Einige Details zur Codeerzeugung

a) Array-Variable

- rekursive Berechnung der (Adresse der) Variablen
- rekursive Berechnung des Index
- Überprüfung, ob $0 \leq$ Index $<$ Feldgröße

  Hinweis: bei Zweierkomplement-Darstellung kommt man mit einem Vergleich vorzeichenloser Größen aus!
- Adresse der Array-Variablen = Adresse der Variablen + Index * Datentypgröße des Basistyps

b) While-Statement

Bsp.: while ($e_1 < e_2$) stm

Zwei mögliche Übersetzungen:

α)
```
      L123:
              <Code für e1 in r1>
              <code für e2 in r2>
              bge  r1, r2, L124
              <code für stm>
              j    L123
      L124:
```
} 2 Sprünge / Durchlauf

β)
```
              j    L123
      L124:
              <code für stm>
      L123:
              <code für e1 in r1>
              <code für e2 in r2>
              blt  r1, r2, L124
```
} 1 Sprung / Durchlauf

Bem. L123 und L124 sind "generierte Marken".

Markengenerator:

```
int newLabel (void) {
    static int numLabels = 0;
    return numLabels++;
}
```

c) If-Statement

α) einarmig, Bsp.: if ($e_1 < e_2$) stm

```
<code für e1 in r1>
<code für e2 in r2>
bge r1, r2, L123
<code für stm>
L123:
```

β) zweiarmig, Bsp.: if ($e_1 < e_2$) stm_1 else stm_2

```
<code für e1 in r1>
<code für e2 in r2>
bge r1, r2, L123
<code für stm1>
j L124
L123:
<code für stm2>
L124:
```

d) Prozedur-Aufruf

- Argumente rekursiv berechnen und abspeichern
- Prozedur aufrufen

e) Prozedur-Deklaration

- Framegröße berechnen:

  ruft diese Prozedur andere Prozeduren?

  nein: Framegröße = Platz f. lokale Variable + 4

  (wegen Framepointer-Speicherplatz)

  ja: Framegröße = Platz f. lokale Variable + 8 +

  Platz f. ausgehende Argumente

  (wegen Framepointer- und Returnadressen-Speicherplatz)

- Prozedur-Prolog ausgeben; siehe 6.2

  (Label, Stackframe allokieren, fp und ggf. Ret. Addr. sichern, neuen fp etablieren)

- Code für Prozedurkörper erzeugen

- Prozedur-Epilog ausgeben; siehe 6.2

  (fp und ggf. Ret. Addr. wiederherstellen, Stackframe freigeben)

- Prozedur verlassen

f) Referenzvariable

Gibt's bei uns nur in Form von Referenz-Parametern

Beim Prozeduraufruf: Übermittlung der Adresse anstatt des Wertes

Bei Verwendung: automatische Dereferenzierung