Compilerbau

v1.2

Mitschrift der Vorlesung im WS 2003 von Dr. Hellwig Geisse
von Boris Budweg

...dieses Script ist mein Dank an alle, von denen ich Unterlagen geschnorrt habe, insbesondere:

Sascha Nitschke
fürs Aufspüren von 24 Fehlern – ohne ihn könnte euch die vorliegende Mitschrift an manchen Stellen aufs Glatteis führen!

Matthias Ansorg
fürs Erleichtern meines Grundstudiums in zahllosen Fächern durch seine guten Mitschriften von z.B. Mathe 1-3

Überblick

1. Einführung
   Aufgaben, Phasen, Läufe, Schnittstellen
2. Lexikalische Analyse
   reguläre Sprachen, endliche Automaten, Scanner-Generatoren
3. Syntaktischen Analyse
   Kontextfreie Sprachen, Ableitungen, LL- und LR-Parser, Parser-Generatoren
4. Abstrakte Syntax und Attributierung
5. Semantische Analyse
   Typen, Typ-Konstruktor, Typ-Überprüfung, Symboltabellen
6. Laufzeitorganisation
   Stack, Stackframes, Prozedurein- und -austritt Parameterübergabe
7. Code-Generierung
   Stack-Maschine, Registermaschinen, Übersetzen von Ausdrücken, Kontrollanweisungen und Prozeduraufrufen

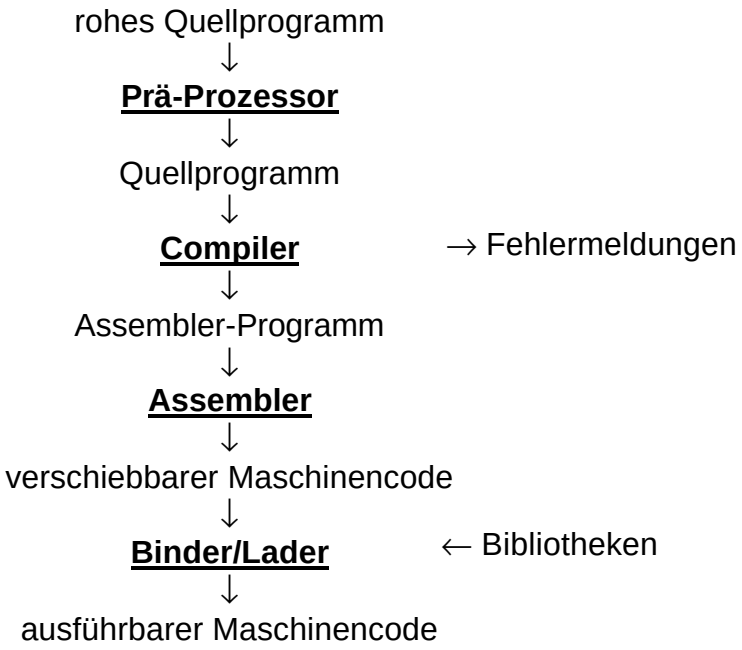
Literaturempfehlung: "Modern Compiler Design", John Wiley & Sons

Praktikum:

Aufgabe:	Konstruktion eines kompletten Compilers für die kleine Programmiersprache SPL
Organisation:	2er-Gruppen (oder einzeln)
Bewertung:	1. Hausübung 25 P (akzeptiert ab 12 P), 2. Hausübung 25 P (akzeptiert ab 12 P) Klausur 50 P (2 Hausübungen Voraussetzung)
Umfang:	~ 1400 Zeilen vorgegeben ~ 2000 Zeilen selbst schreiben
Unterlagen:	hera.mni.fh-giessen.de/~hg53/compilerbau.html

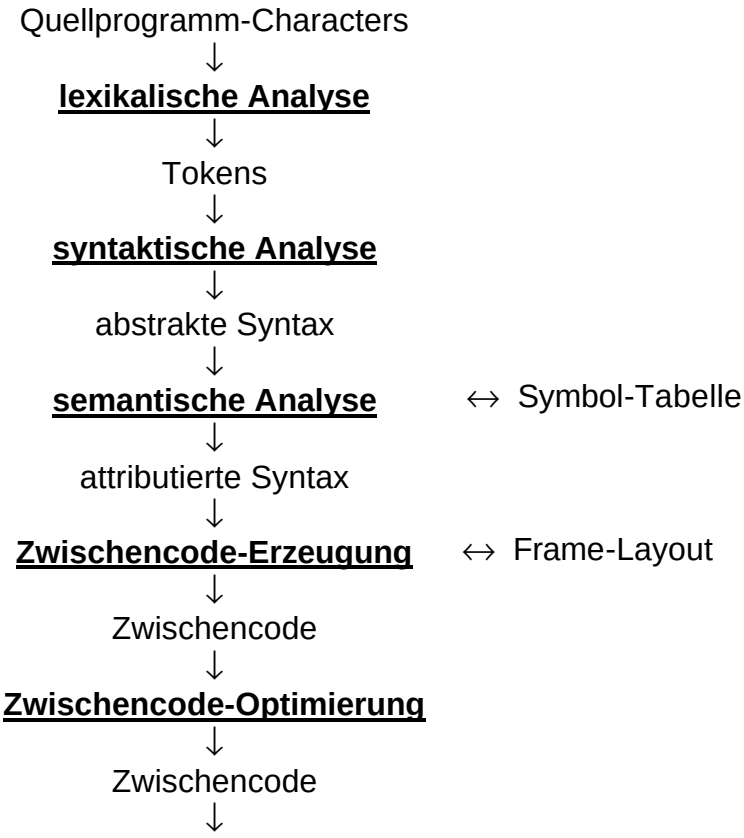
1. Einführung

1.1. Der Compiler in der Werkzeugkette



1.2. Phasen, Läufe, Schnittstellen

Verteilung der Aufgaben auf Phasen.
Zusammenfassung von Phasen zu Läufen (1 Lauf = 1 Durchgang durch das Programm).
Interne Darstellung des Programms durch Schnittstellen



Instruktionsauswahl

↓

symbolischer Maschinencode

↓

Kontrollflußanalyse

↓

Fluß-Graph

↓

Datenflußanalyse

↓

Interferenz-Graph

↓

Register-Allokation

↓

Assembler-Programm

im Folgenden etwas vereinfacht:

1. lexikalische Analyse
2. syntaktische Analyse
3. abstrakte Syntax
4. semantische Analyse
5. Frame-Layout
6. Codegenerierung

Bsp:

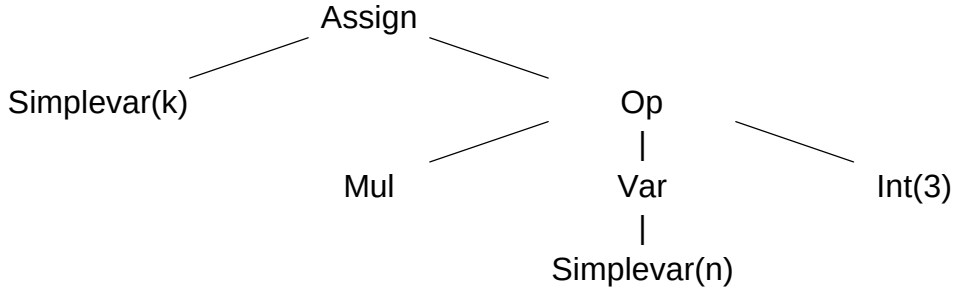
1) Quellcode

```
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: ()

.... (Bibliotheksprozeduren) ....

5) Variabelenallokation (für "demo")

param 'n':	fp+0
param 'k':	fp+4
proc calls:	none

6) Assemblercode (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 Programmiersprache

Bsp:

Token-Typ	konkretes Programm		
IDENT	k	ab123	Das_letzte
NUM	123	0	0xAffe
IF	if		
COMMA	,		
LE	<=		
RPAREN	)		

2.1. reguläre Ausdrücke

Def: Sprache - Menge von Strings
Strings - endliche Folge von Zeichen
Alphabet - alle verfügbaren Zeichen

Bsp: Alphabet = {a,b}
Sprache = {b, ab, aab, aaab, ...}

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

Zeichen: a steht für L(a) = {a} (L = language)
Alternative: $\beta \mid \gamma$ steht für $L(\beta|\gamma) = L(\beta) \cup L(\gamma)$ (β, γ = beliebige reguläre Ausdrücke)

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

Konkartenation: $\beta\ \gamma$ steht für $L(\beta\ \gamma) = \{ st \mid s \in L(\beta) \wedge t \in L(\gamma) \}$

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

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

Bsp: abcε → abc

Kleene'scher Abschluß: β^* steht für $L(\beta^*) = L(\varepsilon) \cup L(\beta) \cup L(\beta\beta) \cup L(\beta\beta\beta) \cup ...$

Bsp: $(ab)^*$ steht für $L((ab)^*) = L(\varepsilon) \cup L(ab) \cup L(abab) \cup ... = \{\varepsilon, ab, abab, ...\}$

Weitere Beispiele:

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

b) Strings aus a und b, die keine 2 a's hintereinander enthalten:
(b*(abb*)*(a|ε)

c) Strings aus a und b, die mindestens einmal 2 a's hintereinander enthalten:
(a|b)*aa(a|b)*

Abkürzungen:

[abcd]	bedeutet	(a | b | c | d)
[b-e]	bedeutet	(b | c | d | e)
β?	bedeutet	(β | ε)
β+	bedeutet	(ββ*)
.	bedeutet	beliebiges Zeichen außer Zeilenumbruch

Bsp: Token einer Programmiersprache:

if	→	IF
[a-z][a-z0-9]	→	IDENT
[0-9]+	→	NUM
0x[0-9a-fA-F]+	→	NUM
\)	→	RPAREN

Auflösen von Mehrdeutigkeiten:
a) Was ist "if8" ? 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

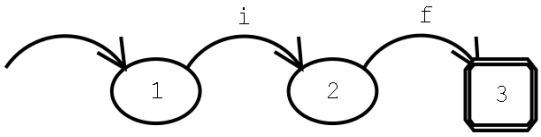
2.2. Deterministische endliche Automaten (DEA)

Bestandteile:
a) endliche Menge v. Zuständen (bezeichnet mit Zahlen)
b) endliche Menge von Zustandsübergängen (markiert mir Eingabezeichen)
c) ein ausgezeichneter Startzustand
d) eine Teilmenge der Zustände sind Endzustände

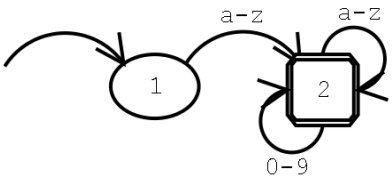
"deterministisch": verschiedene Zustandsübergänge aus dem gleichen Zustand heraus sind mit verschiedenen Eingabezeichen markiert, kein Übergang ist mit ε markiert


Start-Zustand

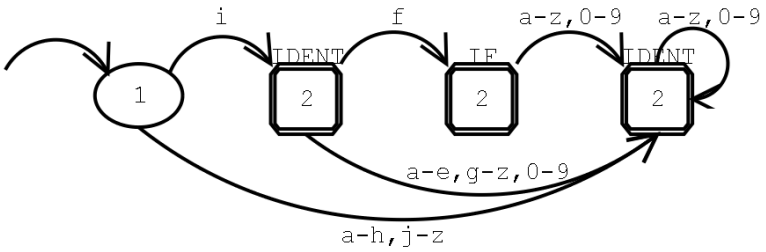

End-Zustand



Automat erkennt Token IF



Automat erkennt Token IDENT

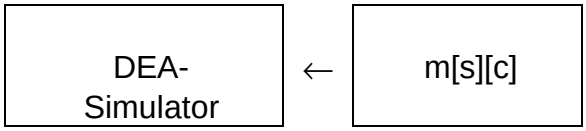


Automat unterscheidet Tokens IF und IDENT

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

2.3. Tabellengesteuerter Scanner

Eingabepuffer



s: momentaner Zustand
c: momentanes Eingabezeichen

Algorithmus:

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

Tabelle m:

	a	b	c
1			
2			
3			

Die erkannte Sprache wird demnach nur durch m festgelegt!

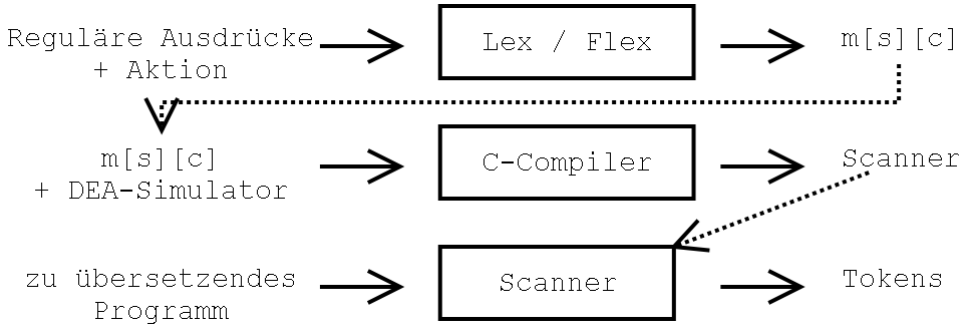
Ergänzungen:

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

b) Es muß eine weitere Tabelle geben, die für jeden Zustand sagt, ob es 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 möglich".

2.4. Scanner-Generatoren



Bsp:

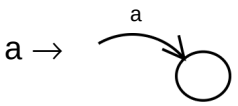
```
[\ \t]+         { /* kein Token zurück */ }
if              { return IF; }
[a-z][a-z0-9]* { return IDENT; }
.               { error("nicht erkanntes Zeichen"); }
```

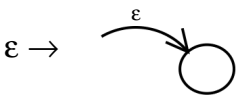
Bemerkung: vernünftige Fehlerausgaben in späteren Phasen des Compilers verlangen das Speicher der "Koordinate" (mind. Zeilennummer) je Token!

2.5. Konstruktion des DEA

zwei Schritte:
a) reguläre Ausdrücke → "Nicht-Deterministischer endlicher Automat" (NEA)
b) NEA → DEA

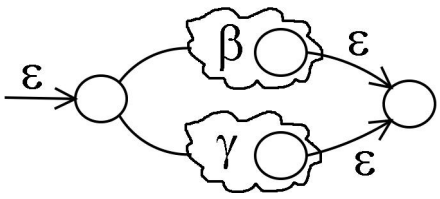
zu a): "Thompson's Konstruktion":



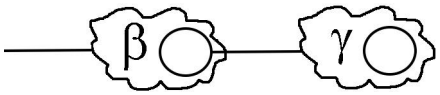


(z.B. u? = (u | ε))

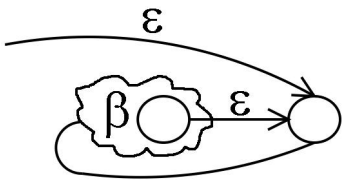
$\beta \mid \gamma \rightarrow$



$\beta\,\gamma \rightarrow$

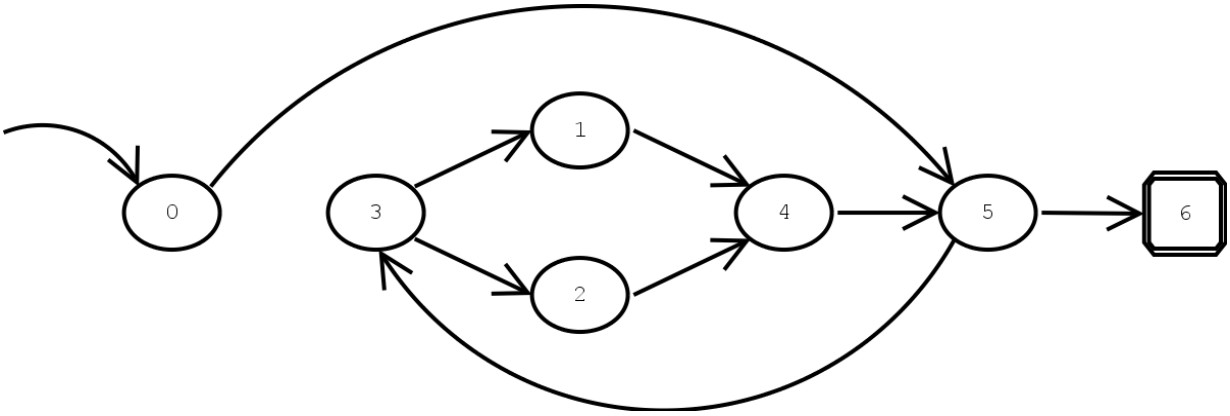


$\beta^* \rightarrow$



Zum Schluß: Startzustand hinzufügen und Endzustand markieren.

Bsp: (a|b)*b



Bem: "nichtdeterministisch": der Automat muß bei gegebenem Zustand und Eingabezeichen raten, wo's langgeht, und ermuß richtig raten!!
(5 + b $\rightarrow$ 6 oder 2 ?)

zu b): "Teilmengenkonstruktion":

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

Definitionen:

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 ∈ T allein über ε - Übergänge erreicht werden

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

S: Menge der Zustände der DEA

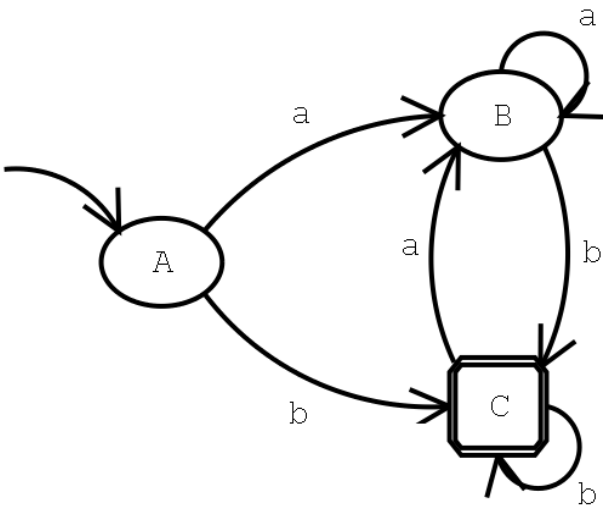
m[T][c]: Übergangstabelle des DEA

Teilmengenkonstruktion:

```
Start:      S := { ε - Abschluß (S0) } // unmarkiert
Iteration:  while ( S enthält 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 (a|b)*b

	NEA-Zustände	DEA-Zustände	Übergänge
	{ 0, 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



Startzustand ist derjenige Zustand, der den alten Startzustand enthält (A)

Endzustand sind alle Zustände, die den alten Endzustand enthalten (evtl. auch mehrere, hier nur C).

3. Syntaktische Analyse

Reguläre Ausdrücke können nicht alle relevanten Konstrukte einer Programmiersprache beschreiben, z.B. korrekte Klammerung.

Alphabet = { a, l, r } (Ausdruck, Klammer links, Klammer rechts)
Sprache = { a, lar, llarr, ... }

Ursache: Keine Informationen über frühere Zustände des Automaten.

Abhilfe: Automat wird mit Stack ausgestattet. Dann können sogenannte "Kontextfreie Sprachen" erkannt werden.

3.1. Kontextfreie Grammatiken

Bestandteile:

a) Menge von Terminalsymbolen T (das sind die Tokens aus Kapitel 2)
b) Menge von Nicht-Terminalsymbolen N ("Variablen")
c) Menge von Produktionen der Form $l \rightarrow r$, 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: $S \rightarrow a$
$S \rightarrow lSr$

Abkürzung: $S \rightarrow a \mid lSr$

Bem: Jede reguläre Sprache ist auch eine kontextfreie Sprache:

regulär	kontextfrei	
a	$S \rightarrow a$	
ε	$S \rightarrow \varepsilon$	
$\beta \mid \gamma$	$S \rightarrow \beta \mid \gamma$	
$\beta \gamma$	$S \rightarrow \beta \gamma$	
β^*	$S \rightarrow \varepsilon \mid \beta S$	(rechts-rekursiv)
β^*	$S \rightarrow \varepsilon \mid S \beta$	(links-rekursiv)

Also: reguläre Sprachen $\subseteq$ kontextfreie Sprachen
Warum dann überhaupt Scanner? $\rightarrow$ Effizienz, reduzierte Komplexität

3.2 Ableitungen, Parse-Bäume, Mehrdeutigkeiten

Grammatik: für arithmetische Ausdrücke:

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

Ist -(id+id) ein Satz der Grammatik?
Ja, denn es gibt eine Ableitung:

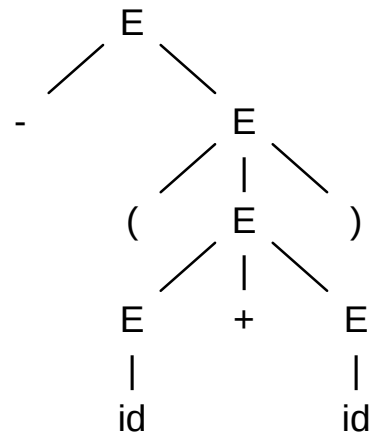
E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)

In jedem Herleitungsschritt gibt es zwei Entscheidungen:
a) Welches Nicht-Terminalsymbol wird ersetzt?
b) Welche Alternative für dieses N wird gewählt?

Linksableitungen ersetzen das jeweils am weitesten links stehende N, Rechtsableitungen analog.

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

Bsp: -(id+id)

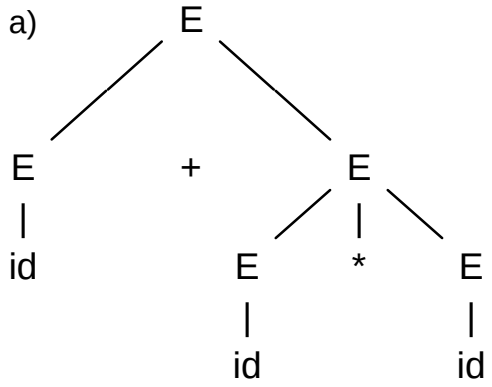


Syntaxanalyse = Finde eine Ableitung = Konstruktion des Parse-Baumes bei gegebenem Satz

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

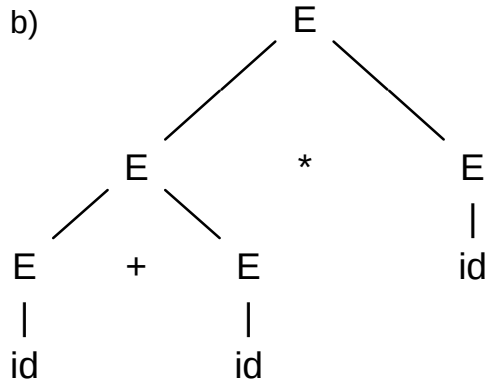
Bsp: Grammatik wie oben, Satz: id + id * id

a)



id + (id * id)

b)

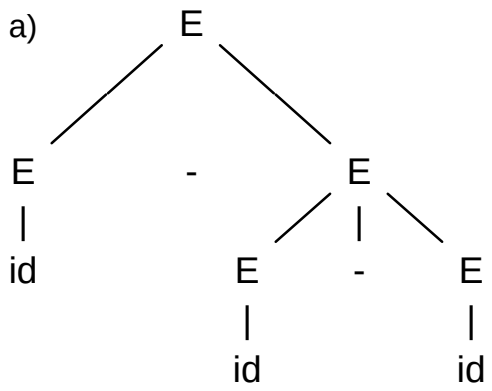


(id + id) * id

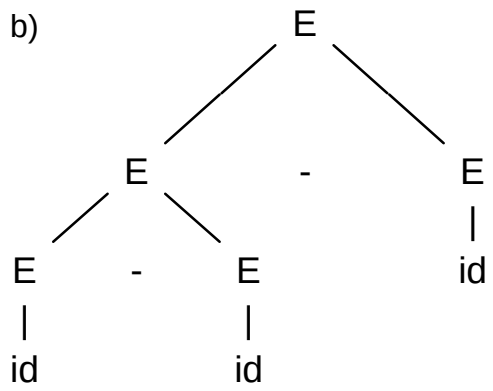
Punkt vor Strich ↑

Bsp: Grammatik wie oben, Satz: id - id - id

a)



b)



(id - id) - id

id - (id - id)

Assoziativität ↑

Grammatik wird eindeutig durch neue Nicht-Terminale:

↓ Priorität

E → E+T | E-T | T	(E = Expression)
T → T*F | T/F | F	(T = Term)
F → -F | id | (E)	(F = Faktor)

Bem: Priorität durch E / T / F
Assoziativität durch Linksrekursion

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

Verfahren: "Rekursiver Abstieg" - jedes N wird durch eine rekursive Prozedur dargestellt, jede Produktion / Alternative ist ein Zweig (switch) in der betreffenden Prozedur.

Eigenschaften: leicht zu programmieren, leider wenig mächtig.

Bsp: S → if E then S (S = Statement, E = Expression)
S → while E do S

Annahme: in t steht das nächste Token

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

Aber: $E \rightarrow E+T \mid E-T \mid T$ läßt sich nicht parsen, denn erstes Terminal muß genug Informationen liefern, um richtige Produktion auszuwählen!

3.3.1. First- und Follow-Mengen

Sei $\beta \in (T \cup N)^*$ eine beliebige Folge von Grammatiksymbolen.
Def: FIRST(β) = { $t \in T \mid \beta \Rightarrow ... \Rightarrow t\,\gamma$ }
d.h. diejenigen T, mit denen ein aus β abgelieferter Satz beginnen kann.
Gilt $\beta \Rightarrow ... \Rightarrow \varepsilon$, dann ist auch $\varepsilon \in$ FIRST(β);

1. Forderung für Top-Down-Parser:

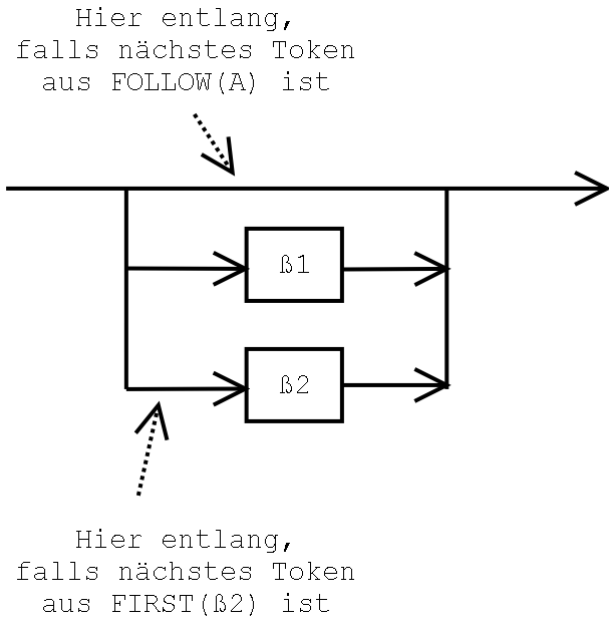
Fasse alle Produktionen für Nichtterminal A zusammen zu $A \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$.
Dann muß FIRST(β_i) $\cap$ FIRST(β_j) = { } für alle $i \neq j$ sein.

Das reicht nicht, falls $\beta \Rightarrow ... \Rightarrow \varepsilon$. Es kommt darauf an, was dann β folgt!

Sei $A \in N$ und S = Startsymbol
Def.: FOLLOW(A) = { $t \in T \mid S \Rightarrow ... \Rightarrow \beta\, A\, t\, \gamma$ }
d.h. diejenigen Terminalsymbole, die in einer Satzform direkt rechts neben A stehen können.

2. Forderung für Top-Down-Parser
Für jedes Nicht-Terminalsymbol A mit $A \Rightarrow ... \Rightarrow \varepsilon$, muß FIRST(A) $\cap$ FOLLOW(A) = { } sein.

Anschaulich in Syntax-Diagrammen:



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

L - Eingabe von links (nach rechts) lesen
L - beim Parsen wird Linksableitung erzeugt
1 - ein lookahead Token

Eine mehrdeutige oder linksrekursive Grammatik ist nicht LL1.

3.3.2 Elemination von Linksrekursion, Linksfaktorisierung

a) Transformation einer linksrekursiven Funktion $A \rightarrow A\ \beta|\gamma$ zu:

$A \rightarrow \gamma\ A'$
$A' \rightarrow \beta\ A' \mid \varepsilon$

Bsp: $E \rightarrow E + T \mid T$ umformen zu:

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

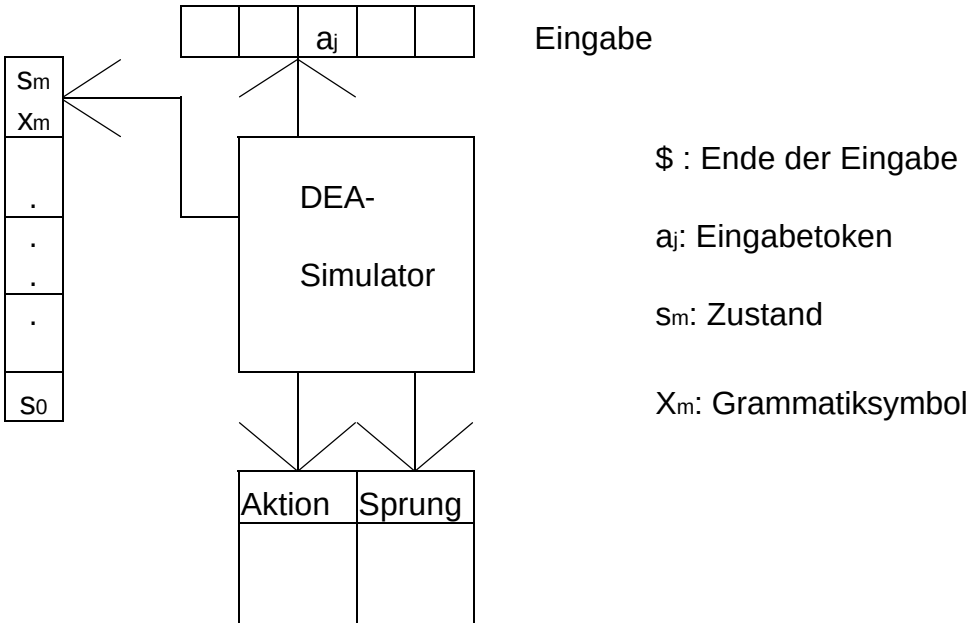
b) Transformation von $A \rightarrow \beta\gamma_1 \mid \beta\gamma_2$ zu:
$A \rightarrow \beta\ A'$
$A' \rightarrow \gamma_1 \mid \gamma_2$

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 gesamte rechte Seite der Produktion gelesen wurde (+ ein Token mehr)

3.4.1. Tabellengesteuerter LR-Parser



DEA-Simulator:

```
push(s0);
accept:=false;
while (!accept) {
      sei s der Zustand oben auf dem Stapel
      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-Nr. m
                  sei k die Anzahl der Grammatiksymbole von β
                  pop ( k Elemente );
                  sei t der Zustand, der jetzt oben auf dem
                                                Stack liegt
                  push ( A, sprung[t][A] );
                  break;
            case accept:
                  accept := true;
                  break;
            case error:
                  error(...);
                  break;
      }
}
```

Bsp: r0: S → E $
r1: E → T + E
r2: E → T
r3: T → a

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

($ = Eingabeende, acc = Accept = erfolgreicher Abschluß)

Bem:

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

b) LR(1) bedeutet: Eingabe v.l.n.r. lesen, Rechtsableitung konstruieren mit 1 Token Vorausschau
Stack + Eingabe = Rechtsableitung in umgekehrter Reihenfolge
E $ => T + E $ => T + T $ => T + a $ => a + a $

c) Tabelleneinträge bestimmen sowohl erkannte Sprache als auch Parsing-Methode (LR(0), SLR(1) (s=simple), LR(1), LALR(1) (la=lookahead))

3.4.2. Parser-Generatoren

Weiteres Vorgehen wie bei Scanner-Generatoren

Bsp:

```
%token        PLUS IDENT
%start        expr
%%
expr          : term PLUS expr
              | term
              ;
term          : IDENT
              ;
%%
// zusätzliche C-Funktionen
```

3.4.3. Konstruktion von LR(0)-Tabellen

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

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

Bsp: Die Produktion $A \to aBc$ bringt 4 Elemente hervor:
$A \to .aBc$ $A \to a.Bc$ $A \to aB.c$ $A \to aBc.$

Def.: Menge von Elementen heißt Zustand des DEA

Der Algorithmus zur Tabellenkonstruktion benutzt zwei Hilfsfunktionen.
Sei I eine Menge von Elementen und X ein Grafiksymbol.

a)
```
Hülle( I ) {
        do {
            for ( jedes Element A → a.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' → 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 ein dieser Iteration geändert );
```

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

Berechnung der Menge der Reduzier-Aktionen R:

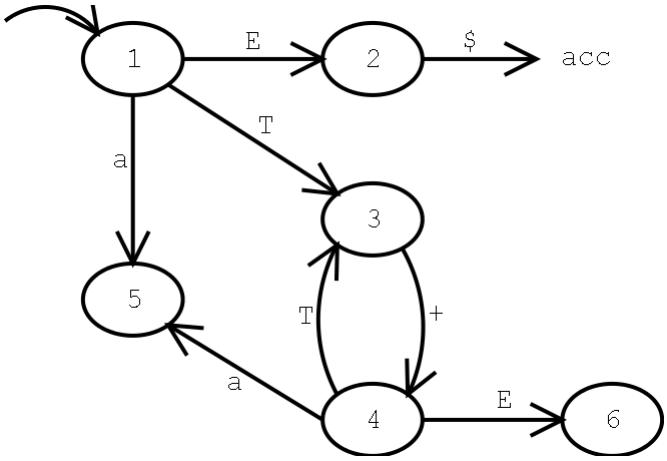
```
R := { };
for ( jeden Zustand I in T ) {
      for ( jedes Element A → α. in I ) {
          R := R ∪ { ( I, A → α ) };     //  I = Zustand
      }          //  A = Produktion, mit der reduziert wird
}
```

Bsp:
r0: S → E $
r1: E → T + E
r2: E → T
r3: T → a

(1) (Startzustand) = Hülle ({ S → .E$ })
Elemente: S → .E $
E → .T + E
E → .T
T → .a

(1) —E→ Sprung ((1), E): S → E. $ | (2)
(1) —T→ Sprung ((1), T): E → T. + E | (3)
E → T.
(3) —+→ Sprung ((3), +): E → T +. E
E → .T + E
E → .T
E → .a | (4)
(1) —a→ Sprung ((1), a): T → a. | (5)
(4) —E→ Sprung ((4), E): E → T + E. | (6)
(4) —T→ Sprung ((4), T): E → T. + E | (3)
E → T.
(4) —a→ Sprung ((4), a): T → a. | (5)

Automat:



Reduzier-Aktionen:

(3, E → T) r2

(5, T → a) r3

(6, E → T + E) r1

LR(0) - Tabelle

Zustand	Aktion			Sprung	
	a	+	$	E	T
1	s5			2	3
2			acc		
3	r2	s4r2	r2		
4	s5			6	3
5	r3	r3	r3		
6	r1	r1	r1		

3 + : "Shift-Reduce-Konflikt"
→ Grammatik ist nicht LR(0)

Bem: Parser-Generatoren lösen solche Konflikte nach 2 Regeln auf:

a) Bei Shift-Reduce-Konflikten wird immer geschoben. Das ist in seltenen Fällen akzeptabel (IF ... IF ... ELSE -> zu welchem IF gehört das ELSE?)

b) Bei Reduce-Reduce-Konflikten hat die zuerst geschriebene Produktion Vorrang. Das ist praktisch 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 $\rightarrow \gamma$ nur dann, wenn nächstes Token $\in$ FOLLOW(A) ist. Winzige Änderung gegenüber LR(0):

Konstruktion der Reduzier-Aktionenen:

```
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 ) };
          }
     }
}
```

unser Bsp: FOLLOW(E) = { $ }
FOLLOW(T) = { +, $ }

Zustand	Aktion			Sprung	
	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-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 (o. Datenstruktur) erzeugen: Verknüpfung von Aktionen mit Grammatik-Produktionen ist notwendig.

4.1. Semantische Aktionen im LL-Parser

Einbau von Aktionen in die Parser-Routinen.
Bsp:

```
E  →  T E'
E' →  + T E' | ε
T  →  n
```

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

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

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

Eingabe: 1+2+3

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

4.2. Semantische Aktionen in LR-Parser

Einbau von Aktionen in die Produktionen.

Bsp:

E → E + T | T
T → n

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

Eingabe / Ausgabe wie oben.

Warum ist das so?
Nachvollziehen mit Rechtsableitung:

E => E + T => E + 3 => E + T + 3 => E + 2 + 3 => T + 2 + 3 => 1 + 2 + 3

In umgekehrter Reihenfolge:

T → 1	push 1
E → T	
T → 2	push 2
E → E + T	add
T → 3	push 3
E → E + T	add

Möglichkeit: ganzer Compiler eingebettet in semantischen Aktionen
Nachteile: schwer zu lesen und zu warten (syntaktische Analyse, semantische Analyse (Typ-Prüfung u.a.) und Code-Erzeugung miteinander verwoben), Informationsfluß ist auf Parse-Reihenfolge beschränkt.
Besser: Erzeugen einer Datenstruktur ("abstrakte Syntax")

4.3. Semantische Werte in LL-Parser

Jede Parser-Routine gibt sogenannten semantischen Wert (z.B. Baumknoten) des von ihr erkannten Nicht-Terminals zurück. Bsp:

ifstm → IF cond THEN stm

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

4.4. semantische Werte in LR-Parser

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

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

($$ = semantischer Wert der linken Seite, $2 + $4 = sem. Wert von cond und stm)

Technische Realisierung: "Semantik-Stack" parallel zum Zustands-Stack

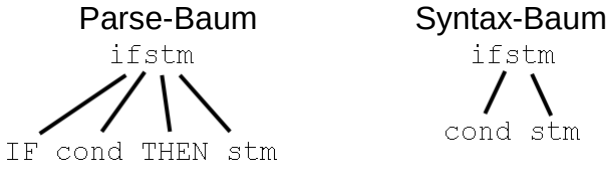
S_m	V_m
S_{m-1}	V_{m-1}
...	...

4.5. Abstrakte Syntax

Zweck: sauberes Interface zwischen Parser und späteren Phasen

Entsteht aus dem Parse-Baum duch weglassen unwichtiger Details.

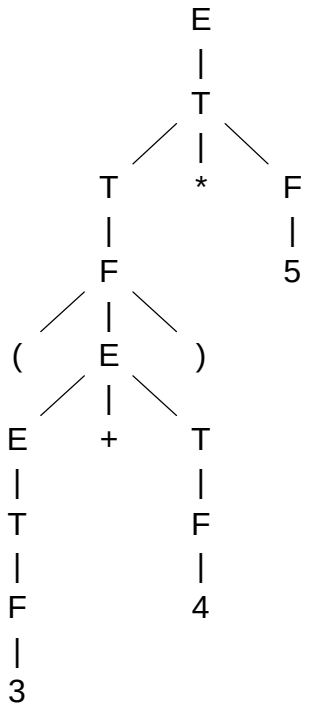
Bsp: a)



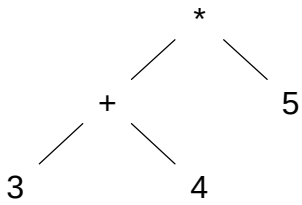
b) $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow n \mid (E)$

Eingabe: (3 + 4) * 5

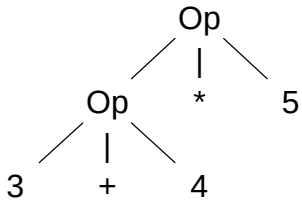
Parse-Baum:



Syntax-Baum:



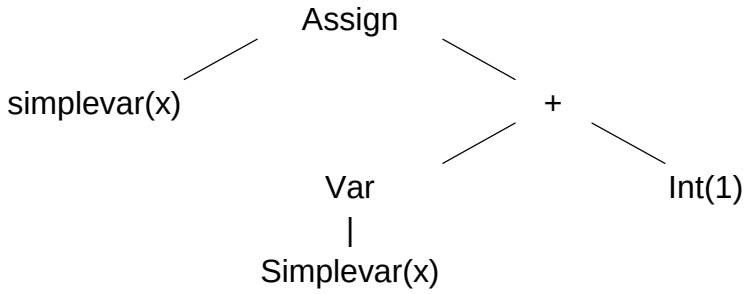
(verkürzt)



(ausführlich)

T : F
{$$ = $1; } (← meist Default-Aktion, kann weggelassen werden)
;

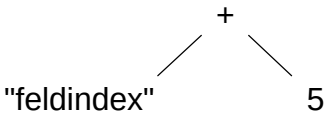
c) x := x + 1



Bem: Simplevar(x) = Adresse von x
Var(x) = Inhalt von x

4.5.1 Effiziente Handhabung von Bezeichnern

feldindex + 5 →



Ineffizient, da bei jedem Durchgang durch den Knoten ein Stringvergleich stattfinden muß!

Besser: Darstellung des Bezeichners durch "Symbol" (= eindeutige Zahl bzw. eindeutiger Zeiger)

Funktionen der Symbol-Verwaltung

Erzeugen:	String → Symbol	
externe Darstellung:	Symbol → String	(für Fehlermeldungen)
Anordnung:	Symbol1 < Symbol2 ?	

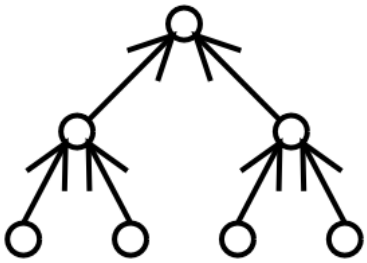
Technische Realisierung durch Hash-Tabelle.

4.6. Attributierung

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

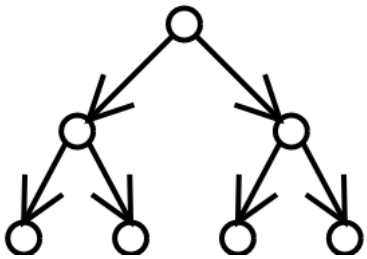
Realisierung: Rekursive Attribut-Auswertung

"Synthetisierte Attribute": (Blätter → Wurzel)



z.B. Konstante Ausdrücke

"vererbte Attribute": (Wurzel → Blätter)



z.B. char *p1, *p2;

→ Typinformationen für *p2 (char) steht weiter oben im Baum

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

5. Semantische Analyse

Aufgabe: Sammeln von Informationen über Bezeichner, überprüfen von Einschränkungen bei der Benutzung von Bezeichnern ("Typprüfungen")

5.1. Typen

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

5.1.1. Darstellung von Typen

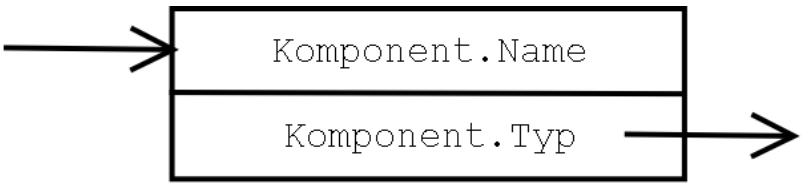
Typen können Bezug auf andere Typen nehmen (Bsp: Array) => Darstellung durch "Typgraph"

a) primitive ("eingebaute") Typen:

b) Arrays:

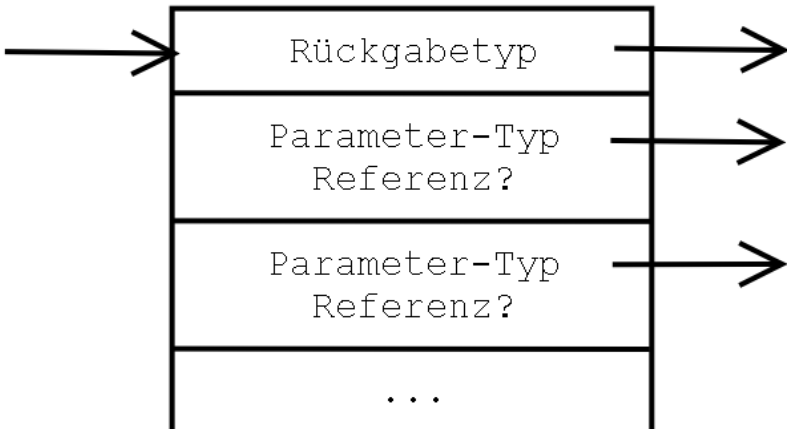
Bem: Manche Sprachen betrachten die Größe als nicht zum Typ gehörend (z.B. C++)

c) Records:



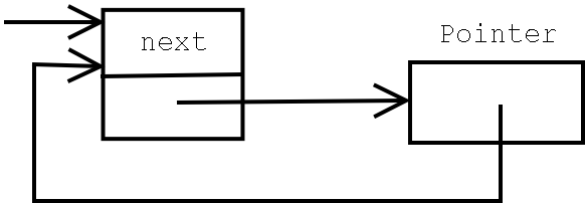
d) Zeiger:

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 drei sind verschieden(!)

b) Ausdrucksgleichheit: Zwei Typen sind genau dann gleich, wenn sie durch denselben Typenausdruck konstruiert werden (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 die gleiche Struktur aufweisen

Bsp: Alle 3 Typen sind gleich.

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

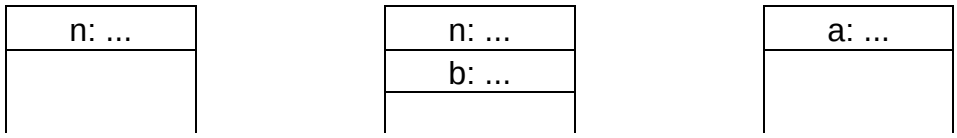
5.2.1. Mehrstufige Symboltabellen

Geschachtelte Gültigkeitsbereiche von Bezeichnern => verschiedene Attributsätze für den gleichen Bezeichner sind möglich => mehrstufige Symboltabellen

Bsp:

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

Symboltabelle von b() → Symboltabelle von a() → globale Symboltabelle



SPL benutzt zweistufige Symboltabellen.

5.2.2. Parallele Symboltabellen

Mehrere Namensräume => verschiedene Attributsätze für gleichen Bezeichner (aus der selben Stufe!) sind möglich => parallele Symboltabellen

Bsp:

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

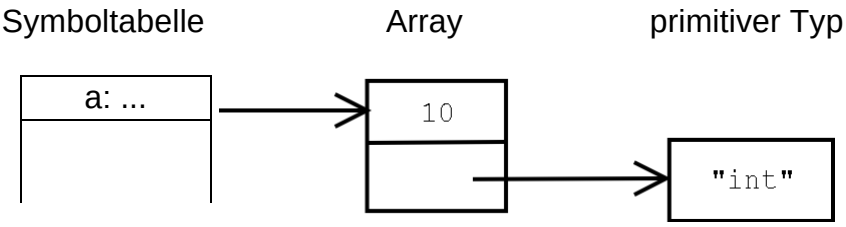
... → Symboltabelle von a():

SPL hat einen gemeinsamen Namensraum für alle Bezeichner.

5.2.3. Attribute für Bezeichner in SPL

a) Typ-Bezeichner benötigt (Typ)

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

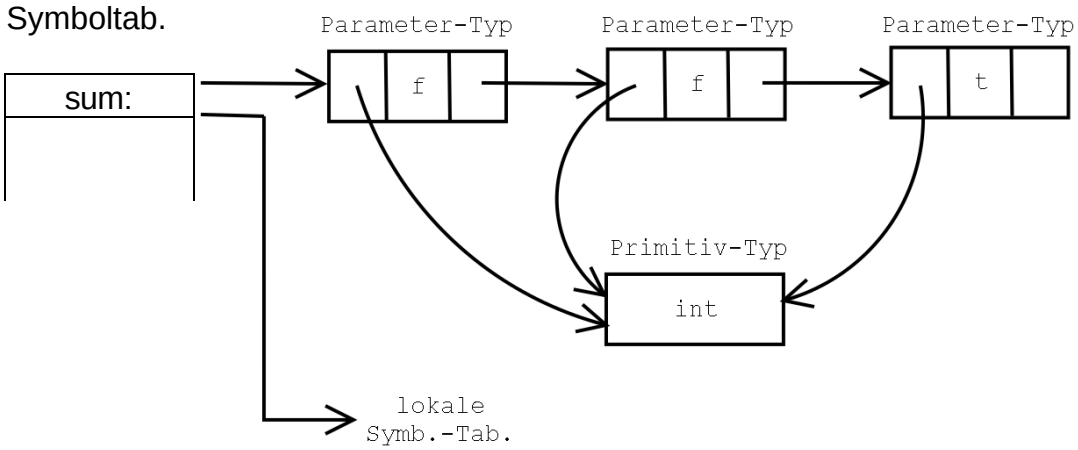


b) Variablen-Bezeichner benötigt (Typ, IstReferenz?)

lokale variable: isRef = false
Wertparameter: isRef = false
Referenzparameter: isRef = true

c) Prozedurbezeichner benötigt (Parameter-Typen, lokale Symboltabelle)

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



Anm: f = false, t = true

5.3. Type-Checking

Der Typ-Checker ist eine rekursive Funktion, die die Abstrakte Syntax inspiziert und einen Typ zurückliefert.

```
Type *checkNode(Absyn *node, Tabelle *symtab) {
     switch (node->type) {
          case ABSYN_NAMETYP:
          return checknode(node,symtab);
          ...
     }
}
```

5.3.1. Type-Checking von Ausdrücken

a) Binäre Operatoren:

```
Type *checkOp(Absyn *node, Tabelle *symTab) {
     Type *leftType, *rightType, *type;
     leftType = checkNode(node->u.opNode.left, symTab);
     rightType = checkNode (node->u.opNode.right, symTab);
     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 *symTab) {
     Entry *entry;
     entry = lookup(symTab, node->u.simplevarNode.name);
     if (entry == NULL) error(...);
     if (entry->kind != ENTRY_KIND_VAR) error(...);
     return entry->u.varEntry.type;
}
```

c) Array:

- überprüfen, daß wirklich ein Array indiziert wird (var a:int; a[15]....)
- überprüfen, daß Index vom Typ int (a[n+3])
- 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 von lhs (left-hand-side) und rhs (right-hand-side) "zuweisungskompatibel" (bei uns: gleich) sind

b) If- und While-Anweisungen

- überprüfen, daß Tests boolische Werte liefern
- rekursiven Abstieg in die Teilanweisungen nicht vergessen

c) Prozeduraufrufe

- Bezeichner existent? Ist Prozedur?
- Stimmen die aktuellen Argumente in Zahl und Typ mit den formalen Parametern überein?
- bei Referenzparameter: Ist Variable? (andere Ausdrücke verboten)

5.3.3. Type-Checking von Typen

(einfach entsprechenden Typgraphen zurückgeben)

a) benannte Typen

- in Symboltabelle nachschauen (vorhanden? Typ?)
- Typ zurückgeben

b) Array-Typen

- Basistyp ermitteln (rekursiv)
- Array-Typ konstruieren und zurückgeben

5.3.4. Type-Checking von Deklarationen

Deklarationen bewirken Einträge in der Symboltabelle

a) Typ-Deklaration

```
Typ *checkTypdecl(Absyn *node, Table *symTab) {
     Type *type;
     Entry *entry;
     type := checkNode(node->u.typedeclNode.type, symTab);
     entry = newTypeEntry(type);
     if (enter(symTab, node->u.typedeclNode.name, entry)
          == NULL) error(...);     // Typ schon vorhanden
     return NULL;
}
```

b) Variablen-Deklaration

Wie a), aber mit newVarEntry(typ, FALSE); // FALSE = keine Referenz-Variable

c) Prozedur-Deklaration

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

5.2.5. Vorwärtsreferenzen und Rekursion

Benutzung eines Bezeichners vor seiner Deklaration, unvermeidlich der Fall bei wechselseitig rekursiven Typen oder Prozeduren.

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

Lösung: Type-Checking in 2 Durchläufen
1. Durchlauf: Type-Checking der Parameter und Eintrag der Prozedur in globale Symboltabelle; Rumpf ignorieren
2. Durchlauf: Alle anderen o.g. Aktivitäten

Bem: Bei Typ-Prüfung 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 Typ-Checkers

- Erzeugen der Typgraphen für primitive und eingebaute Typen
- Erzeugen der globalen Symboltabelle
- Eintragen der vordefinierten Bezeichner: Typen, Prozeduren, etc. in die globale Syboltabelle
- überprüfen globaler semantischer Bedingungen (z.B. ist main() vorhanden?)
- ggf Ausgabe der globalen Symboltabelle (Debugging)

6. Laufzeitorganisation

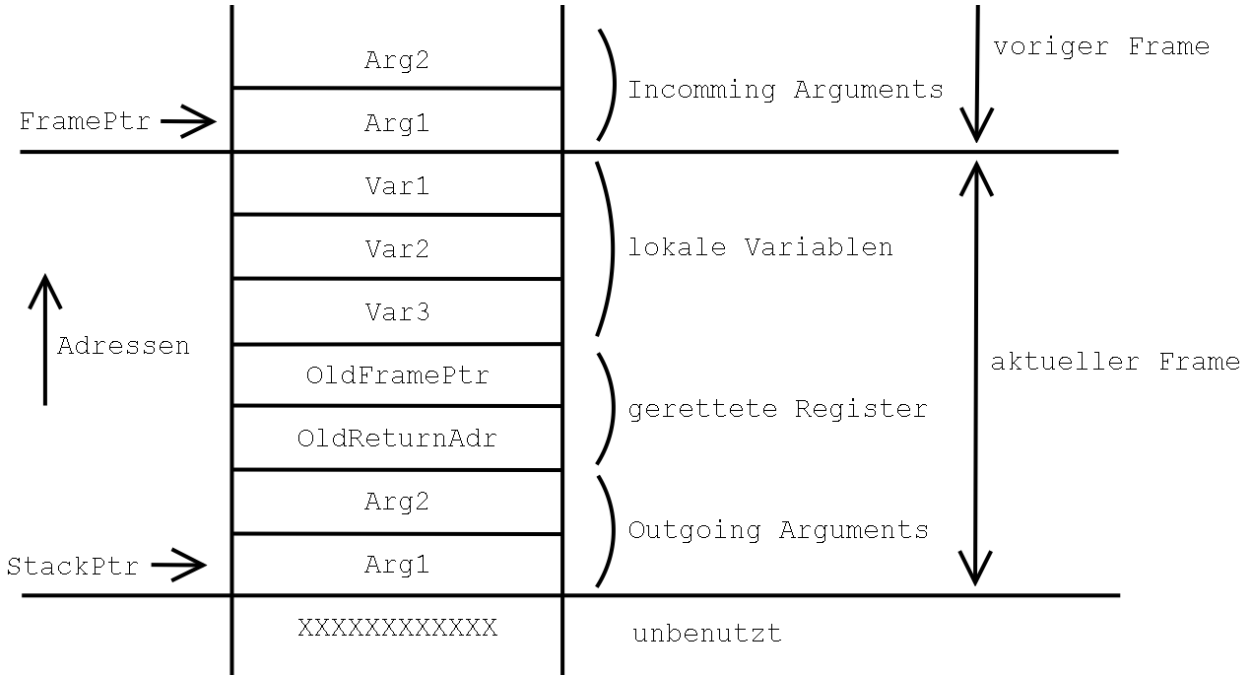
Jeder Funktions- / Prozedur-Aufruf hat seinen eigenen Satz von lokalen Variablen (und Parametern). Viele solcher Aufrufe kommen zur gleichen Zeit existieren (Rekursion!). In vielen Sprachen (einschließlich SPL) sollen die Speicherplätze für lokale Variablen automatisch freigegeben werden, wenn die Ausführung die Funktion verläßt => Stack

6.1. Stack-Frames

Variablen werden in größerer Zahl angelegt / freigegeben (nicht einzeln) => push / pop für ganze Bereiche ("Stack-Frame", "Activation Record"). Die Struktur eines Stack-Frames wird typischerweise nicht von der Hardware vorgegeben, sondern von Software-Ingenieuren vereinbart (wichtig für das Binden von Programmen, die in verschiedenen Programmiersprachen geschrieben sind).

Bestandteile:
- lokale Variablen
- Return-Adresse
- Hilfsvariablen
- gerettete Register
- Argumente für aufgerufene Prozedur

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 hineinschreiben)

c) Die Speicherplätze für OldReturnAdr und Outgoing Arguments werden nur angelegt, falls die aktuelle Prozedur eine andere aufrufen kann (ob sie's wirklich tut, ist zur Übersetzungszeit meist nicht berechenbar)

6.2. Der Frame-Pointer, Prozedurein- und -austritt

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

proc:

```
sp ← sp - framesize                 // sp = Stack-Pointer
mem[sp + offset(oldfp)] ← fp        // fp = Frame-Pointer
fp ← sp + framesize
...
(Hier können Argumente mit fp+offset(Arg) und lokale Variablen mit fp-offset(Var)
adressiert werden. Argumente für gerufene Funktion werden an sp+offset(Arg)
gelegt.)
...
fp ← mem[sp + offset(oldfp)]
sp ← sp + framesize
return
```

Zum Zeitpunkt der Codegenerierung werden benötigt:
- framesize
- offset(oldfp)
- offset(Argument) für alle eingehenden Argumente
- offset(Argument) für alle ausgehenden Argumente
- offset(Variable) für alle lokalen Variablen

6.3. Berechnung zur Laufzeitorganisation

1. Jeder Typ (nicht nur benannte!) speichert in seinem Typgraphen seine Größe, d.h. die Anzahl Bytes, die ein Objekt diesen Typs im Speicher belegen wird (Notwendig für Offsetberechnung). Diese Information entsteht bei der Typgraphen-Konstruktion

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

3. Jede Prozedur speichert die Größe von 3 Bereichen:
   a) Eingehende Argumente
   b) lokale Variablen
   c) Ausgehende Argumente

   a und b entstehen durch 2. c wird am besten in sepparatem Durchlauf ermittelt. Für jede Prozedur p bestimme das Maximum der eingehenden Argumente der von p gerufenen Prozeduren. Dieser Wert ist die Größe des Bereiches für ausgehende Argumente für p.

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

7. Codegenerierung

7.1. Die Zielmaschine ECO32

- 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, maximal 1 GB, Zugriffe müssen ausgerichtet werden

- I/O: Timer, Terminal, Disk, Grafik-Controller, Memory-mapped, Interrupt- und DMA-fähig (Bedienung wird von Bibliotheks-Prozeduren geleistet)

- virtueller Speicher 4KB Seitengröße, max. 4 GB, TLB-Unterstützung (das benutzen wir nicht: alle Adressen ab 0xC000 0000 werden durch die Hardware auf physikalische Adressen ab 0 abgebildet)

b) Instruktionen

```
add  $4,  $17, $8   ; $4 ← $17 + $8
add  $4,  $17, 8    ; $4 ← $17 + 8
```
entsprechend für sub, mul, div

Speicherzugriff

```
ldw  $4,  $17, 8    ; $4 ← mem[$17+8]
stw  $4,  $17, 8    ; mem[$17+8] ← $4
```

Label

```
hier:               ; das ist ein Label
```

bedingte Anweisung

```
beq  $4,  $17, hier ; if ($4 == $17) goto "hier"
```
entsprechend für bne (!=), bgt (>), bge (>=), blt, ble für vorzeichenbehaftete Werte
entsprechend bgtu, bgeu, bltu, bleu für vorzeichenlose Größen

```
j    hier           ; unbedingter Sprung
jal  hier           ; Prozeduraufruf der Prozedur "hier"
jr   $31            ; weiter beim Inhalt von $31
```

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 folgenden Bytes kommen ins Code-Segment
.align   4         ;  entstehen der Adresse auf einen durch 4
                      teilbaren Wert einmal am Anfang des Moduls
```

d) Register-Konventionen

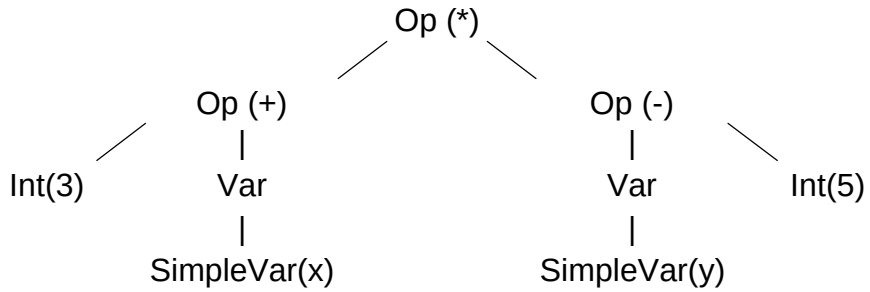
$0	immer 0 (HW)
$31	return-Adresse (HW)
$25	Frame-Pointer
$29	Stack-Pointer
$8 - $23	Hilfsvariablen

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 die Abstrakte Syntax

Bsp: (3 + x) * (y - 5)



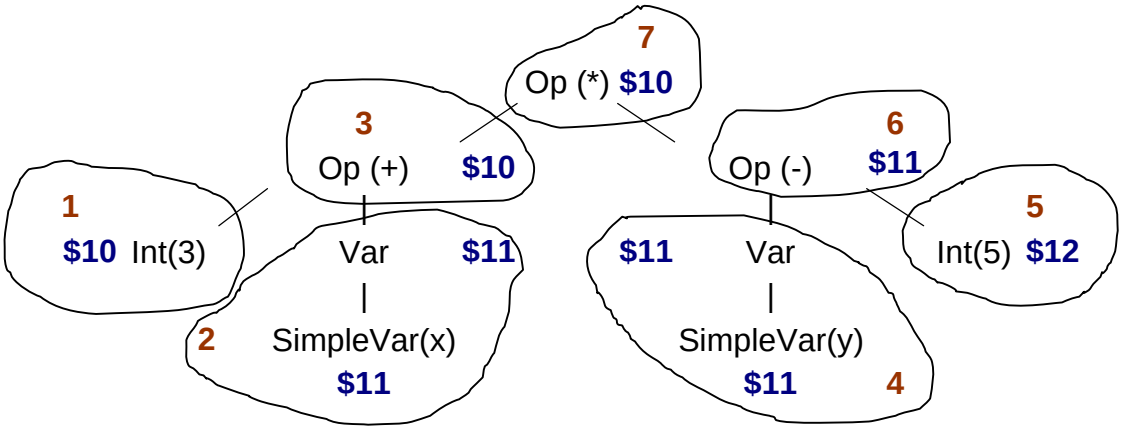

```
push(3);
push(Adr(x));
load;    ersetzt letzten Stackwert mit Inhalt seiner Adresse
add;
push(Adr(y));
load;
push(5)fv;
sub;
mul;
```

7.3. Code für Register-Maschinen

Wie oben, aber mit Registern anstelle des Stacks: Die Stackpositionen werden jetzt zu Registernummern

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);
     printf("add $%d, $%d, $%d\n",target,target,temp);
}
```



(linksrekursiver Abstieg)

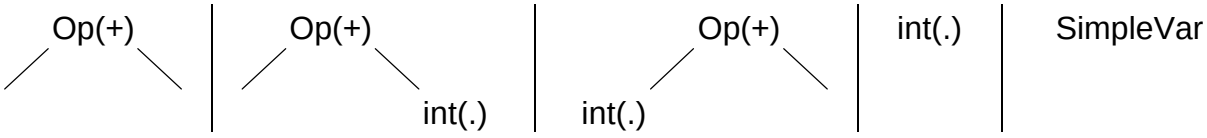
Zusammengefaßt:

```
1    add  $10, $0,  3              add  $10, $0,  3

2    add  $11, $25, offset(x)      ldw  $11, $25, offset(x)
     ldw  $11, $11, 0

3    add  $10, $10, $11            add  $10, $10, $11

4    add  $11, $25, offset(y)      ldw  $11, $25, offset(y)
     ldw  $11, $11, 0

5    add  $12, $0,  5              sub  $11, $11, 5
6    sub  $11, $11, $12

7    mul  $10, $10, $11            mul  $10, $10, $11
```

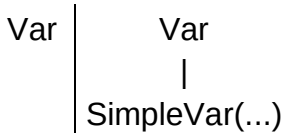
Bem:

a) Das Zusammenfassen von Instruktionen kann folgendermaßen erreicht werden ("Maximal Munch"):
Jede Instruktion der Zielmaschine erzeugt eine oder mehrere "Kacheln", je nach möglichen Argumenten:

add:



ldw:



Dann wird der ganze Baum der Abstrakten Syntax mit Kacheln belegt, ausgehend von der Wurzel mit möglichst großen Kacheln.

b) Eine Verbesserung läßt sich mit dynamische Programmierung erzielen. Weiterer Vorteil: Algorithmus kann in ein Tool integriert werden ("Codegenerator-Generator").

c) der o.g. Algorithmus zur Allokation von Registern ist nicht immer optimal (→ "Sethi-Ullmann-Algorithmus"; behandelt auch das Aus-/Wiedereinlagern von Werten für temporäre Variablen in den Stackframe, falls nicht genügend Register verfügbar sind).

7.4. Einige Details zur Codeerzeugung

a) Array-Variable
- rekursive Berechnung der Variablen (Adressen der Variablen)
- rekursive Berechnung des Index
- Überprüfung, ob 0 = Index < Feldgröße
  Hinweis: Bei Zweierkomplement-Darstellung kommt man hier mit einem Vergleich vorzeichenloser Größen aus
- Adresse der Array-Variablen = Variablen-Adresse + (Index * Datentypgröße)

b) While-Statement
Bsp: `while (e1 < e2) stm`
Zwei Varianten:

1)
```
L123:                          // Label
        <code für e1 in r1>
        <code für e2 in r2>
        bge r1, r2, L124       // branch on greater or equal
        <code für stm>
        j L123
  L124:
```

(2 Sprünge pro Durchlauf)

2)
```
j 123
L124:
        <code für stm>
L123:
        <code für e1 in r1>
        <code für e2 in r2>
        bge r1, r2, L124
```

(1 Sprung pro Durchlauf)

Bem: Markengenerator: (L123, L124 sind "generierte Marken")

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

c) If-Statement

Bsp: `if (e1 < e2) stm else stm2`

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

d) Prozeduraufruf
- Argumente rekursiv berechnen und abspeichern
- Prozedur aufrufen

e) Prozedurdeklarationen
- Framegröße berechnen
  ruft diese Prozedur andere Prozeduren?
  nein: Framegröße = Platz für lokale Variablen + 4 Bytes für Framepointer
  ja: Framegröße = Platz für lokale Variablen + 8 Bytes für Framepointer und Returnadresse + Platz für ausgehende Argumente

- Prozedur-Prolog ausgeben (→6.2) (Label (Procname), Stackframe allokieren, fp und ggf Return-Adresse sichern, neuen Framepointer etablieren)

- Code für Prozedur-Körper erzeugen

- Prozedur-Epilog ausgeben (→6.2) (Framepointer und ggf Returnadresse wiederherstellen, Stackframe freigeben)

- Prozedur verlassen

f) Referenz-Variable (bei SPL nur als Parameter beim Prozeduraufruf)
- Übermittlung der Adresse anstatt des Wertes
- Bei Verwendung: automatische Dereferenzierung