

Skript zur Vorlesung

# Compilerkonstruktion

Wintersemester 2015/16

© Prof. Dr. R. Parchmann  
Institut für Praktische Informatik  
FG Programmiersprachen und Übersetzer  
Universität Hannover

## Vorwort

Dieses Skript entstand aus Notizen zu der Vorlesung „Compilerkonstruktion II“, die ich im Sommersemester 2003 und 2005 gehalten habe.

Ziel der Vorlesung ist es, gemeinsam mit der Vorlesung „Compilerkonstruktion I“ einen Überblick über die wichtigsten Konzepte und Techniken des Compilerbaus zu geben.

In dieser Vorlesung werden zunächst attributierte Grammatiken als formales Modell der Übersetzung kontextfreier Sprachen eingeführt. Danach wird als letzter Analyse-Teil eines Compilers die semantische Analyse behandelt. Vor der eigentlichen Zwischencode-Erzeugung wird das Laufzeitsystem sowie die Implementation von Blockstruktur und Parameterübergabe bei höheren Programmiersprachen diskutiert.

Die eigentliche Zwischencode-Erzeugung zeigt, wie wichtige Strukturen höherer Programmiersprachen in einen einfachen Zwischencode übersetzt werden können. Es schließt sich ein Abschnitt über Code-Optimierung an, in dem eine Analyse der Ablaufstruktur und der Datenabhängigkeiten behandelt wird. Das abschließende Kapitel beschäftigt sich mit den Problemen der Maschinencode-Erzeugung und gibt am Ende einen kurzen Überblick über Konzepte zur automatischen Generierung derartiger Programmteile aus formalen Beschreibungen der Maschinsprache des Zielrechners.

An dieser Stelle möchte ich Herrn cand. math. Thomas Schwanck danken, der einen Teil meiner etwas kryptischen und unvollständigen Vorlesungsnotizen in dieses Manuskript übertrug und dazu beigetragen hat, dass dieses Skript jetzt den Studierenden zur Verfügung steht.

Hannover, August 2005

Rainer Parchmann

Für die Vorlesung im Sommersemester 2007 wurde das Skript überarbeitet und teilweise umgestellt und erweitert. Der Abschnitt über die Datenfluss-Gleichungen wurde um die Berechnung verfügbarer Ausdrücke ergänzt. Speziell wurde das Konzept einer Transfer-Funktion für Drei-Adress-Befehle eingeführt und die Berechnung der Transferfunktion für einfache Blöcke hergeleitet. Das Zielmaschinen-Modell wurde im letzten Kapitel auf eine RISC-Architektur umgestellt und die Beispiele entsprechend angepasst.

Für die Vorlesung im Sommersemester 2009 wurde im Kapitel 4 die Implementation einer Laufzeitumgebungen für objektorientierte Programmiersprachen hinzugefügt.

Hannover, Juli 2009

Rainer Parchmann

Da im Wintersemester 2013/2014 diese Vorlesung ohne die früher gehaltene vorangehende Vorlesung „Compilerkonstruktion I“ gelesen wird, sind ein paar Änderungen notwendig geworden, um mit den Voraussetzungen der Vorlesung „Programmiersprachen und Übersetzer“ dieser Vorlesung folgen zu können. Einige Teile der semantischen Analyse und der Diskussion des Laufzeitsystems wurden weggelassen, dafür wurden Teile über das Speichermanagement und über globale Registerzuordnung hinzugefügt.

Hannover, Dezember 2013

Rainer Parchmann

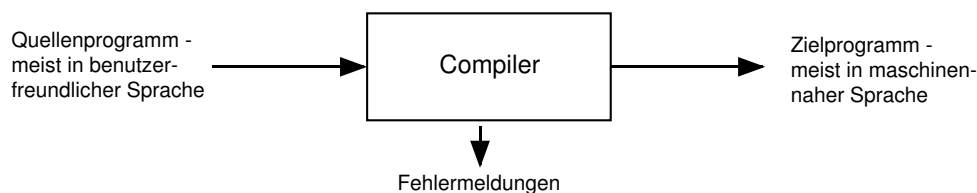
## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Prinzipieller Aufbau eines Compilers . . . . .	1
<b>2</b>	<b>Attributierte Grammatiken</b>	<b>5</b>
2.1	Definitionen . . . . .	5
2.1.1	Konstruktion eines Abhängigkeitsgraphen für eine Produktion . . . . .	11
2.1.2	Konstruktion eines Abhängigkeitsgraphen für einen Ableitungsbaum . . . . .	11
2.2	Übersetzung eines Wortes $w$ mit einer attribuierten Grammatik . . . . .	12
2.3	Auswertung L-attributierter Grammatiken beim Top-Down Parsing . . . . .	15
2.4	Umformungen der Grammatiken . . . . .	17
<b>3</b>	<b>Semantische Analyse</b>	<b>21</b>
3.1	Typ-Ausdrücke . . . . .	22
3.2	Einfache Typ-Synthese . . . . .	24
3.3	Äquivalenz von Typ-Ausdrücken . . . . .	26
3.4	Typ-Umwandlungen . . . . .	29
3.5	Operator-Identifikation (Überladene Funktionen) . . . . .	30
3.6	Untertypen . . . . .	32
3.7	Polymorphe Funktionen . . . . .	33
<b>4</b>	<b>Das Laufzeitsystem</b>	<b>38</b>
4.1	Gültigkeitsbereich und Lebensdauer . . . . .	38
4.2	Aktivierungs-Record, statische und dynamische Verkettung . . . . .	39
4.3	Prozeduren als Parameter oder als Rückgabewerte . . . . .	42
4.4	Speicherorganisation . . . . .	43
4.5	Parameterübergabe . . . . .	45
4.6	Speicherverwaltung in objektorientierten Systemen . . . . .	46
4.7	Heap Management . . . . .	51
4.7.1	Die Speicherhierarchie eines Rechners . . . . .	51
4.7.2	Speicherverwaltung . . . . .	51
4.8	Garbage Collection . . . . .	53
4.8.1	Erreichbarkeit von Objekten . . . . .	53
4.8.2	Garbage Collection über Reference Counting . . . . .	54
4.8.3	Mark-and-Sweep Garbage Collection . . . . .	54
<b>5</b>	<b>Zwischencode-Erzeugung</b>	<b>56</b>
5.1	Form des Zwischencodes . . . . .	56
5.2	Drei-Adress-Befehle . . . . .	58
5.2.1	Darstellung von Drei-Adress-Befehlen . . . . .	61
5.3	Syntax-gesteuerte Übersetzung von Deklarationen . . . . .	62
5.4	Übersetzung von Feldzugriffen . . . . .	63
5.4.1	Eindimensionale Felder . . . . .	63
5.4.2	Zweidimensionale Felder . . . . .	63
5.4.3	Mehrdimensionale Felder . . . . .	64
5.5	Übersetzung Boolescher Ausdrücke . . . . .	68
5.5.1	Numerische Kodierung . . . . .	69
5.5.2	Steuerung des Programmablaufs . . . . .	69
5.6	Übersetzung Boolescher Ausdrücke mit Backpatching . . . . .	73

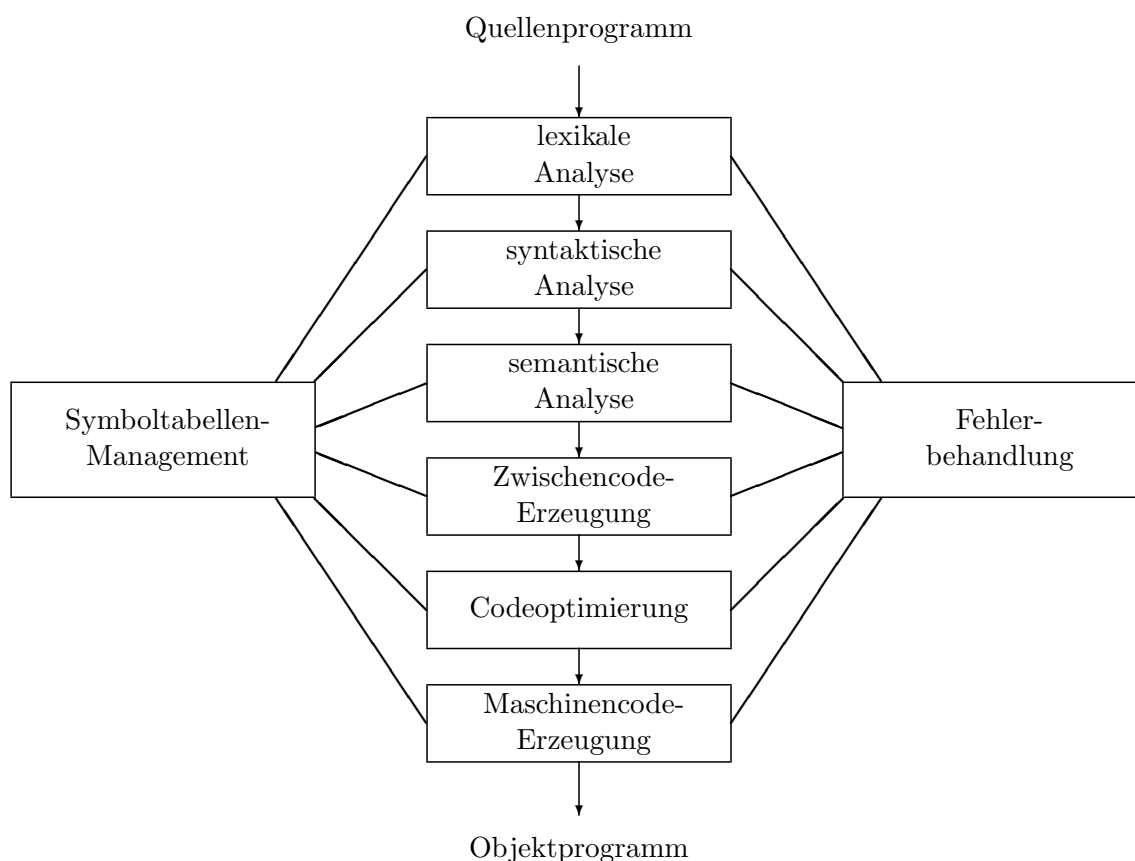
5.7	Übersetzung von Programmsteuerbefehlen (mit Backpatching)	75
<b>6</b>	<b>Codeoptimierung</b>	<b>78</b>
6.1	Einführung	78
6.2	Einfache Blöcke und Flussgraphen	80
6.2.1	Algorithmus zur Partitionierung in einfache Blöcke:	80
6.2.2	Nächster Gebrauch oder der Lebendigkeit von Variablen	80
6.2.3	Konstruktion des Flussgraphen	81
6.3	Möglichkeiten der Codeoptimierung	82
6.4	Lokale Optimierungen	87
6.4.1	Konstruktion eines DAGs für einen einfachen Block	87
6.4.2	Darstellung von Feldzugriffen im DAG	89
6.5	Datenfluss Analyse	91
6.5.1	Transferfunktionen	92
6.5.2	Verfügbare Definitionen (Reaching Definitions)	93
6.5.3	ud - Ketten	97
6.5.4	Lebendigkeit von Variablen	97
6.5.5	Analyse der verfügbaren Ausdrücke (Available Expressions)	100
6.6	Analyse der Ablaufstruktur	103
6.6.1	Algorithmus zur Berechnung der $\succeq$ -Relation	103
6.6.2	Natürliche Schleifen	104
<b>7</b>	<b>Maschinencode-Erzeugung</b>	<b>106</b>
7.1	Einführung	106
7.2	Die Zielmaschine	107
7.3	Ein einfacher Maschinencode-Erzeuger	110
7.3.1	Maschinencode-Erzeugung für einfache Blöcke	112
7.3.2	Mögliche Implementation der Funktion <code>getreg</code>	113
7.4	Maschinencode-Erzeugung für Syntax-Bäume	117
7.4.1	Die Markierungsphase	117
7.4.2	Die Code-Erzeugungsphase	119
7.5	Code-Erzeugung mit dynamischer Programmierung	122
7.5.1	Der Algorithmus	123
7.6	Globale Register Allokation	126
7.6.1	Usage Counts	126
7.6.2	Register Allokation durch Graph-Färbung	128
7.7	Generatoren für die Maschinencode-Erzeugung	133
7.7.1	Verwendung von Baum-Grammatiken bzw. Baum-Übersetzungs-Schemata	133
7.7.2	Verwendung eines üblichen LR-Parsers	137
7.7.3	Der Ansatz von Aho, Ganapathi und Tjiang	138
	<b>Literatur</b>	<b>141</b>
	<b>Index</b>	<b>143</b>

# 1 Einleitung

Als Compiler bezeichnet ein Programm, das einen Text - meist einen Programmtext - geschrieben in einer Sprache A einliest und in einen äquivalenten Text in einer anderen Sprache B übersetzt.



## 1.1 Prinzipieller Aufbau eines Compilers



Die Aufgaben der einzelnen Komponenten kann man kurz wie folgt charakterisieren:

**lexikale Analyse:** Aufgabe der lexikalischen Analyse ist das zeichenweise Lesen der Eingabe und das Zusammenfassen der Zeichen zu größeren Einheiten. Eine derartige Einheit wird auch **Lexem** genannt. Als interne Darstellung wird für jedes Lexem ein **Token** erzeugt. Token repräsentieren eine Folge von Eingabezeichen, die als logische Einheit zu betrachten sind; z. B. ist die Zeichenfolge `1234` als eine Zahl und `nicht` als Folge von vier Zahlen aufzufassen. An diesem Beispiel erkennt man auch, dass ein Token aus zwei Komponenten besteht,

nämlich der **Tokenklasse** oder dem **Tokennamen** (in diesem Fall **Zahl**) und dem **Tokenwert** (in diesem Fall der Zahlwert 1234). Übliche Tokenklassen bei Programmiersprachen sind z. B. Schlüsselworte wie etwa **do**, **begin**, **end** usw. oder **Zahl** oder **Identifizier** (Namen). Was als Token zu klassifizieren ist, hängt von der beabsichtigten Anwendung ab und ist in das Ermessen des Compilerkonstruktors gestellt. Häufig werden Schlüsselworte und Namen von Variablen (Identifizier) aus Effizienzgründen vom Scanner in eine Tabelle aller auftretenden Worte eingetragen. Man erreicht dadurch, dass auftretende Identifizier-Tokens eindeutig sind; zum Beispiel sollen zwei unterschiedliche Auftreten eines Identifiziers **abc** zu einem eindeutigen Token führen. Außerdem werden Leerzeichen und Kommentare von dieser Komponente des Compilers, die auch als **Lexical Scanner** oder einfach **Scanner** bezeichnet wird, überlesen.

Als theoretisches Modell zur Beschreibung der Tokenklassen dienen reguläre Sprachen bzw. reguläre Ausdrücke und endliche, erkennende Automaten.

**syntaktische Analyse:** Diese Komponente des Compilers wird auch als **Parser** bezeichnet. Ihre Aufgabe ist:

- 1) Eine Prüfung, ob die von der lexikalischen Analyse gelieferten Token in einer Reihenfolge auftreten, die der (syntaktischen) Definition der Programmiersprache entspricht. (Z. B. gibt **identifizier + \* identifizier** in Java einen Fehler, in C jedoch nicht!)
- 2) Die Gruppieren der Token zu größeren syntaktischen Strukturen, etwa in Form eines Baumes (**Syntaxbaum**). Ein interner Knoten des Syntaxbaums stellt eine Operationen dar, die auf Werte angewendet wird, die durch die Teilbäume unter dem internen Knoten repräsentiert werden.
- 3) Aufbau und Verwaltung einer Symboltabelle, in der die Token mit entsprechenden Typ-Informationen abgelegt werden.

Als theoretisches Modell zur Beschreibung der korrekten Syntax einer Programmiersprache werden kontextfreie Grammatiken benutzt. Dementsprechend bilden verschiedene Arten von Kellerautomaten ein Modell für den Parser.

**semantische Analyse:** Wichtigste Aufgabe der semantischen Analyse ist die Überprüfung der Typen von Konstanten und Variablen bezüglich der auf sie angewendeten Operationen. (Z. B. sollte ein Feldname mit einem Index vom Typ **real** einen Fehler liefern.) Falls es die Programmiersprache erlaubt, kann in dieser Phase des Compilers auch eine automatische Typanpassung von Variablen eingefügt werden. Weiterhin muss, sofern nicht schon in der syntaktischen Analyse geschehen, überprüft werden, ob verwendete Identifizier auch im Deklarationsteil des Programms definiert worden sind.

Bei objektorientierten Sprachen muss die semantische Analyse anhand der Klassenhierarchie prüfen, ob eine Methode auf einen Identifizier anwendbar ist und gegebenenfalls, falls zur Übersetzungszeit bereits möglich, die „passende“ Methode ausgewählt werden. Schwieriger wird es bei generischen Klassen und speziell bei komplizierteren Typ-Systemen, insbesondere bei Typ-Inferenzsystemen, muss man mit allgemeinen Typ-Ausdrücken und Unifikationalgorithmen arbeiten.

Bei einfacheren Programmiersprachen kann die semantische Analyse parallel zur syntaktischen Analyse erfolgen, bei komplexeren Typs-Systemen ist die semantische Analyse in mehrere Teilphasen zerlegt, die nacheinander abgearbeitet werden.

**Zwischencode-Erzeugung:** Aufgabe der Zwischencode-Erzeugung ist die Übersetzung in einen Zwischencode, d. h. in die Maschinensprache einer abstrakten Maschine, die möglichst einfache Struktur hat.

Dieser Zwischenschritt ist besonders bei optimierenden Compilern wichtig. Dieser Zwischencode (Syntax-Bäume, Bytecode, 3-Adress-Befehle usw.) sollte so aufgebaut sein, dass er leicht von der höheren Programmiersprache aus produzierbar, aber auch leicht in Maschinensprache übersetzbar oder aber auch interpretierbar ist, wie etwa der Bytecode der Java Virtual Machine.

Häufig ist diese Phase der Übersetzung in mehrere einzelne Schritte aufgeteilt, in denen verschiedene Zwischencodes erzeugt werden.

Ein theoretisches Modell für einen allgemeinen Übersetzungsprozess bilden zum Beispiel attributierte Grammatiken oder Syntax-gesteuerte Übersetzungsschemata.

**Codeoptimierung:** In dieser Phase des Compilers wird versucht, den Zwischencodes in Hinblick auf Speicherplatzbedarf und Laufzeitverhalten zu verbessern. Dazu dienen Algorithmen zum Entfernen überflüssiger Berechnungen, Ersetzung von Berechnungen zur Laufzeit durch Berechnungen zur Übersetzungszeit, Erkennen von schleifeninvarianten Berechnungen und Verschieben dieser Teile vor den Schleifenbeginn usw. Hier geht meist schon die Struktur der Zielmaschine entscheidend ein.

Um diese Veränderungen am Zwischencode ohne Veränderung der Bedeutung des Programms durchführen zu können, werden notwendige Informationen in dieser Phase durch das Lösen komplexer Datenfluss- und Kontrollfluss-Gleichungen gewonnen.

Auch diese Phase des Compilers läuft meist in mehreren nacheinander folgenden Schritten ab, die teilweise mit der Zwischencode-Erzeugung verzahnt sind.

**Code-Erzeugung:** Erzeugung von verschieblichen Maschinencode (d. h. eines Objektprogramms bzw. eines Assemblerprogramms).

Die größte Schwierigkeit besteht dabei darin, die Zuordnung der wenigen freien Register so zu wählen, dass möglichst wenig Speicherzugriffe nötig sind. (Speziell wichtig und kompliziert bei modernen RISC-CPU's oder Parallelrechnern.) sowie die Auswahl der „passenden“ Maschinenbefehle.

Als Modell verwendet man in dieser Compilerstufe häufig Baumgrammatiken in Verbindung mit dynamischer Programmierung mit einer speziellen Form des Template-Matchings oder des Pattern-Matchings.

**Symboltabelle:** Die Aufgabe der Symboltabelle ist das Sammeln aller Informationen über die im Programm auftretenden Variablen. Dazu gehört

- Typ-Informationen
- Speicherbedarf und Speicherort
- Gültigkeitsbereich
- Bei Prozeduren Anzahl und Typ der Parameter und der Rückgabewerte

Die Symboltabelle muss eine einfache Möglichkeit bieten, neue Informationen aufzunehmen und nach Informationen zu suchen. Außerdem muss die Symboltabelle Sichtbarkeitsregeln in der Programmiersprache, etwa eine Blockstruktur, berücksichtigen.

Die notwendigen Informationen werden von verschiedenen Teilen des Compilers geliefert und eingetragen.

**Hinweis::**

Die ersten Stufen eines Compilers sind relativ gut formalisierbar und die zugehörigen theoretischen Modelle sind intensiv studiert worden. Dies hat zur Folge, dass es eine Vielzahl von relativ guten Programmen zur automatischen Erzeugung von lexikalen Scannern und Parsern gibt. (etwa `yacc`, `lex`, `flex`, `bison`, `ANTLR` usw.)

Bei den restlichen Stufen sind die Trennungslinien nicht so ausgeprägt. Die Abhängigkeit der einzelnen Stufen voneinander und auch die Abhängigkeit von der Zielmaschine erschweren die Formalisierbarkeit.

Es hat in der Vergangenheit viele Versuche gegeben, auch diese Stufen eines Compilers automatisch auf einer Spezifikation zu generieren, jedoch hat sich kein Ansatz so richtig durchgesetzt. Die Schwierigkeiten und die Komplexität dieser Stufen hat sich in der Vergangenheit eher erhöht, denn

- 1) Die modernen Rechner haben hierarchisch geordnete Speicherstrukturen (verschiedene Cache-Stufen), so dass man nicht mehr von konstanten „Kosten“ für eine Lade- bzw. Speicheroperation ausgehen kann.
- 2) Die in modernen Prozessoren vorhandenen Einheiten zur Bearbeitung von Vektoroperationen oder die effiziente Ausnutzung mehrfach vorhandener Funktionseinheiten stellen zusätzliche schwierige Probleme an die letzten Stufen des Compilers.
- 3) Die „Kosten“ eines Maschinenbefehls sind bei modernen RISC-Prozessoren *nicht* mehr kontext-unabhängig; man denke an Pipelining und voneinander unabhängige Funktionseinheiten.

Als Beispiel soll folgendes Programmfragment<sup>1</sup> betrachtet werden. Man erkennt sofort, dass gleiche Operationen in unterschiedlichem Kontext unterschiedliche „Kosten“ haben können.

	step	rel. Geschw.
do i = 1,1024*step,step	1	100%
a[i] := a[i] + c;	2	83%
	4	63%
end do;	8	40%
	16	23%
	64	19%
	256	12%

---

<sup>1</sup>aus D.F.Bacon, Compiler Transformations for High-Performance Computing, ACM Comp. Surveys 26, 1994



## 2 Attributierte Grammatiken

Ein wichtiges formales Modell für die Übersetzung einer formalen Sprache (und damit natürlich auch der Festlegung einer Semantik für diese Sprache) ist die attributierte Grammatik.

Erste Ideen führen zurück auf eine Arbeit von Irons [33], die Originalarbeit stammt von Knuth [35] aus dem Jahr 1968.

### 2.1 Definitionen

Die grundlegende Idee der attributierten Grammatik ist es, den Symbolen der Grammatik „Attribute“ (Eigenschaften oder Werte) und den Produktionen Regeln zur Berechnung der Werte der Attribute zuzuordnen.

**Definition:** Sei  $G = (N, T, P, S)$  eine kontextfreie Grammatik. Jedem Symbol  $X \in (N \cup T)$  ist eine endliche Menge von **Attributen**  $\mathcal{A}(X)$  zugeordnet. Das Attribut  $a \in \mathcal{A}(X)$  wird auch mit  $X.a$  bezeichnet. Jedem Attribut  $a \in \mathcal{A}(X)$  ist eine Wertemenge  $\mathcal{W}(X.a)$  zugeordnet. Das Attribut  $a \in \mathcal{A}(X)$  kann nur Werte aus seiner Wertemenge  $\mathcal{W}(X.a)$  annehmen.

Die Menge der Attribute  $\mathcal{A}(X)$  ist disjunkt zerlegt in

- die Menge der **inheriten Attribute**  $\mathcal{A}_I(X)$  und in
- die Menge der **synthetischen Attribute**  $\mathcal{A}_S(X)$ ,

d.h. es gilt  $\mathcal{A}(X) = \mathcal{A}_I(X) \cup \mathcal{A}_S(X)$  und  $\mathcal{A}_I(X) \cap \mathcal{A}_S(X) = \emptyset$ .

Die Werte der Attribute eines Symbols  $X$  repräsentieren später Eigenschaften dieses Symbols  $X$  bezüglich des Ableitungsbaumes, in dem dieses Symbol auftritt.

- Inherite Attribute repräsentieren Aspekte der Umgebung von  $X$  im Ableitungsbaum
- Synthetische Attribute repräsentieren Aspekte der weiteren Ableitung von  $X$ , d.h. des Ableitungsbaumes unter  $X$ .

**Bemerkung:** Häufig wird in der Theorie außerdem gefordert, dass das Startsymbol  $S$  kein inherites Attribut und jedes  $X \in T$  kein synthetisches Attribut besitzt. Es hat sich jedoch als vorteilhaft erwiesen, für praktische Probleme auf diese Einschränkung zu verzichten und statt dessen eine Initialisierung von Attributwerten (etwa durch den Scanner) anzunehmen!

In [1] wird eine attributierte Grammatik auch **Syntax-Directed Definition (SSD)** genannt.

Jeder Produktion  $X \rightarrow Y_1 \dots Y_k$  in  $P$ ,  $X \in N$ ,  $Y_i \in (N \cup T)$ ,  $1 \leq i \leq k$ ,  $k \geq 0$ , ist eine endliche Menge von Funktionen (**Regeln**) zugeordnet, die die Berechnung der Werte der synthetischen Attribute von  $X$  und der inheriten Attribute aller  $Y_i$  festlegen (**Semantische Funktionen, semantische Regeln**).

Dabei gilt:

- Für jedes Attribut  $a \in \mathcal{A}_S(X)$  gibt es eine Regel, die den Wert von  $a$  in Abhängigkeit von Werten anderer Attribute  $D(X.a) \subseteq \mathcal{A}(X) \cup \bigcup_{i=1}^k \mathcal{A}(Y_i)$  berechnet.
- Für jedes Attribut  $a \in \mathcal{A}_I(Y_j)$ ,  $1 \leq j \leq k$ , gibt es eine Regel, die den Wert von  $a$  in Abhängigkeit der Werte anderer Attribute  $D(Y_j.a) \subseteq \mathcal{A}(X) \cup \bigcup_{i=1}^k \mathcal{A}(Y_i)$  berechnet.

**Sprechweise:** Das Attribut  $a$  ist *abhängig* vom Attribut  $b$ , falls  $b \in D(X.a)$  bzw.  $b \in D(Y_i.a)$ .

Eine kontextfreie Grammatik mit einer Attributierung der terminalen und nichtterminalen Symbole und mit einer Zuordnung semantischer Regeln zu allen Produktionen heißt **attributierte Grammatik**.

**Bemerkung:** In der formalen Definition treten nur Funktionen im mathematischen Sinne auf, d.h. Funktionen, bei deren Auswertung keine Nebeneffekte auftreten.

Es werden später jedoch als semantische Regeln auch Prozeduraufrufe bzw. Code-Fragmente verwendet, die durchaus Nebeneffekte haben können, etwa Teile des Zwischencodes erzeugen und ausgeben.

Man kann durch Einführen von zusätzlichen „Dummy“-Attributen und Festlegung einer bestimmten Reihenfolge der Auswertung dies jedoch wieder auf die ursprüngliche Definition zurückführen.

**Definition:** Ein **attributierter Ableitungsbaum** eines Wortes  $w \in L(G)$  ist ein Ableitungsbaum für  $w$  bezüglich der kontextfreien Grammatik  $G$ , bei dem in jedem Knoten für ein  $X \in N \cup T$  auch die Attribute  $\mathcal{A}(X)$  notiert sind.

**Definition:** Ein **ausgewerteter attributierter Ableitungsbaum** eines Wortes  $w \in L(G)$  ist ein attributierter Ableitungsbaum für  $w$ , in dem jedes Attribut  $a$  eines mit  $X \in (N \cup T)$  markierten Knotens einen Wert aus  $\mathcal{W}(X.a)$  hat und die zugehörige semantische Regel für  $X.a$  erfüllt (korrekt) ist (**annotierter Ableitungsbaum**).

Der Prozess des Zuordnens von Werten zu Attributen eines attributierten Ableitungsbaums heißt **Auswertung des attributierten Ableitungsbaums**.

**Bem.:**

- 1) Nicht jeder attributierte Ableitungsbaum lässt sich auswerten!
- 2) Man beachte, dass bei der Verwendung von Funktionen mit Seiteneffekten, z.B. bei der Einführung globaler Variablen, Schwierigkeiten auftreten können und die Auswertung nicht mehr eindeutig ist. In diesem Fall ist die Auswertereihenfolge entscheidend!
- 3) Häufig wird ein synthetisches Attribut des Startsymbols der Grammatik ausgezeichnet. Dieses Attribut des Wurzelknotens des Ableitungsbaumes enthält dann nach der Auswertung „die Übersetzung des Wortes  $w$ “.

### Beispiel 2.1:

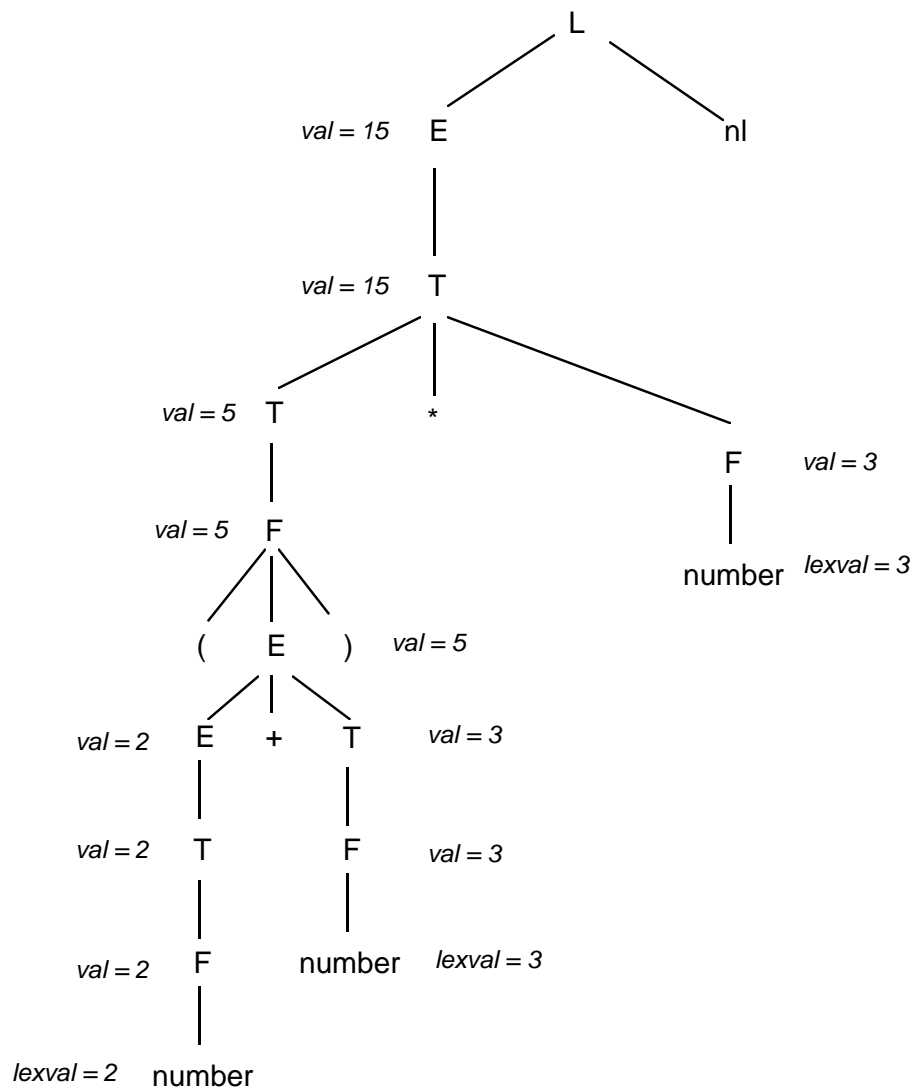
Gegeben ist eine kontextfreie Grammatik  $G = (N, T', P, L)$  mit  $N = \{L, F, E, T\}$ ,  $T' = \{\mathbf{nl}, +, *, (, ), \mathbf{number}\}$  und  $\mathcal{A}_S(E) = \mathcal{A}_S(F) = \mathcal{A}_S(T) = \{val\}$ ,  $\mathcal{A}_S(\mathbf{number}) = lexval$ .

Weiterhin sei  $\mathcal{A}_S(L) = \mathcal{A}_I(L) = \mathcal{A}_I(E) = \mathcal{A}_I(F) = \mathcal{A}_I(T) = \mathcal{A}_I(\mathbf{number}) = \emptyset$ .

Die folgende Attributierung bewirkt, dass ein Wort aus  $L(G)$  als arithmetischer Ausdruck interpretiert und ausgewertet wird.

Die der Produktion  $L \rightarrow E \mathbf{nl}$  zugeordnete semantische Regel hat als Seiteneffekt das Ausdrucken des Wertes des arithmetischen Ausdrucks.

Produktionen	Semantische Regeln
$L \rightarrow E \mathbf{nl}$	$\text{print}(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathbf{number}$	$F.val := \mathbf{number}.lexval$



Ausgewerteter attributierter Ableitungsbaum für  $w = (2 + 3) * 3 \text{ nl}$ .

Man beachte, dass unsere Beispielgrammatik nur synthetische Attribute enthält. Grammatiken dieses Typs haben einige Vorteile, so kann man zum Beispiel die Auswertung eines attributierten Ableitungsbaums bei diesen Grammatiken in einem Durchlauf durch den Baum „bottom-up“ durchführen.

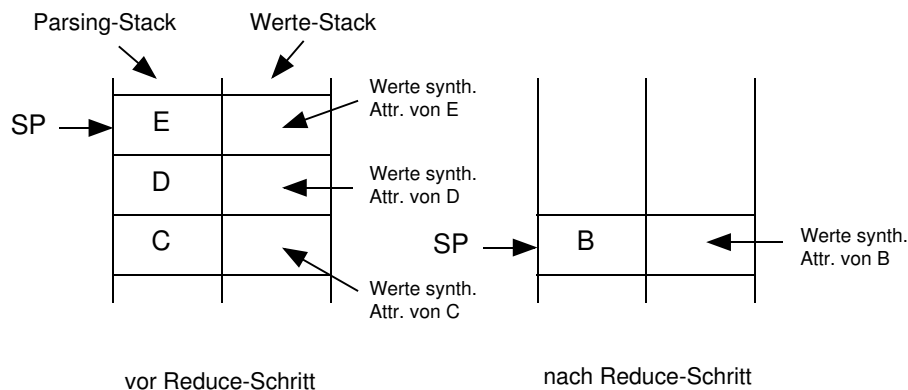
**Definition:** Eine attributierte Grammatik, in der nur synthetische Attribute definiert sind, heißt **S-attributierte Grammatik**.

Die Auswertung attributierter Ableitungsbäume von S-attribuierten Grammatiken kann parallel zum Parsing erfolgen, wenn ein bottom-up Parsingverfahren benutzt wird. Dazu müssen nur die Stack-Einträge im Parsing-Stack so groß gewählt werden, dass ausreichend Platz für die Werte der zugeordneten synthetischen Attribute vorhanden ist.

Vor einem Reduce-Schritt des Parsers befinden sich alle Symbole auf der rechten Seite einer Produktion samt den Werten ihrer Attribute auf dem Stack. Man kann also jetzt die der Produktion zugeordneten semantischen Regeln auswerten und die Werte der synthetischen Attribute des Symbols auf der linken Seite der Produktion berechnen. Die so berechneten Werte werden dann *nach* dem Reduce-Schritt dem obersten Stacksymbol zugeordnet.

**Beispiel 2.2:**

Sei  $B \rightarrow CDE$  eine Produktion und seien für alle  $a \in \mathcal{A}_S(B)$  Regeln zugeordnet. Die beiden Abbildungen symbolisieren die Situation vor und nach einem Reduktionsschritt des Parsers.



Für unsere Beispielgrammatik aus Beispiel 2.1 könnte man daher vielleicht wie folgt vorgehen: Da nur ein synthetisches Attribut für jedes nichtterminale Symbol der Grammatik existiert, ist der Zugriff auf die Werte des synthetischen Attributs recht einfach.  $val(SP)$  soll z. B. den obersten Wert im Werte-Stack bezeichnen.

Der lexikale Scanner legt bei Erkennen einer Zahl den Tokenwert direkt in den Werte-Stack ab.  $SP'$  bezeichne die Position von  $SP$  nach einem Reduce-Schritt. Dann kann man die semantischen Regeln durch kurze Code-Fragmente angeben, die die semantischen Regeln realisieren.

Produktionen	Code-Fragmente
$L \rightarrow E \text{ nl}$	<code>print(val[SP - 1])</code>
$E \rightarrow E_1 + T$	<code>val[SP'] := val[SP - 2] + val[SP]</code>
$E \rightarrow T$	
$T \rightarrow T_1 * F$	<code>val[SP'] := val[SP - 2] * val[SP]</code>
$T \rightarrow F$	
$F \rightarrow (E)$	<code>val[SP'] := val[SP - 1]</code>
$F \rightarrow \text{number}$	

Abarbeitung des Wortes  $(2 + 3) * 3 \text{ nl}$ :

Eingabe	Pars.-Stack	Werte-Stack	angew. Prod
$(2 + 3) * 3 \text{ nl}$			
$2 + 3) * 3 \text{ nl}$	(	—	
$+3) * 3 \text{ nl}$	( <b>number</b>	—2	
$+3) * 3 \text{ nl}$	( <i>F</i>	—2	$F \rightarrow \text{number}$
$+3) * 3 \text{ nl}$	( <i>T</i>	—2	$T \rightarrow F$
$+3) * 3 \text{ nl}$	( <i>E</i>	—2	$E \rightarrow T$
$3) * 3 \text{ nl}$	( <i>E</i> +	—2—	
$) * 3 \text{ nl}$	( <i>E</i> + <b>number</b>	—2 — 3	
$) * 3 \text{ nl}$	( <i>E</i> + <i>F</i>	—2 — 3	$F \rightarrow \text{number}$
$) * 3 \text{ nl}$	( <i>E</i> + <i>T</i>	—2 — 3	$T \rightarrow F$
$) * 3 \text{ nl}$	( <i>E</i>	—5	$E \rightarrow E_1 + T$
$*3 \text{ nl}$	( <i>E</i> )	—5—	
$*3 \text{ nl}$	<i>F</i>	5	$F \rightarrow (E)$
$*3 \text{ nl}$	<i>T</i>	5	$T \rightarrow F$
$3 \text{ nl}$	<i>T</i> *	5—	

usw.

Nun ein weiteres Beispiel, um zu zeigen, dass dieser Prozess des gleichzeitigen Parsens und Auswertens der Attribute keinesfalls immer so einfach funktioniert. Bei attribuierten Grammatiken mit inheriten und synthetischen Attributen kann der Auswertungsprozess eventuell viel komplexer werden:

### Beispiel 2.3:

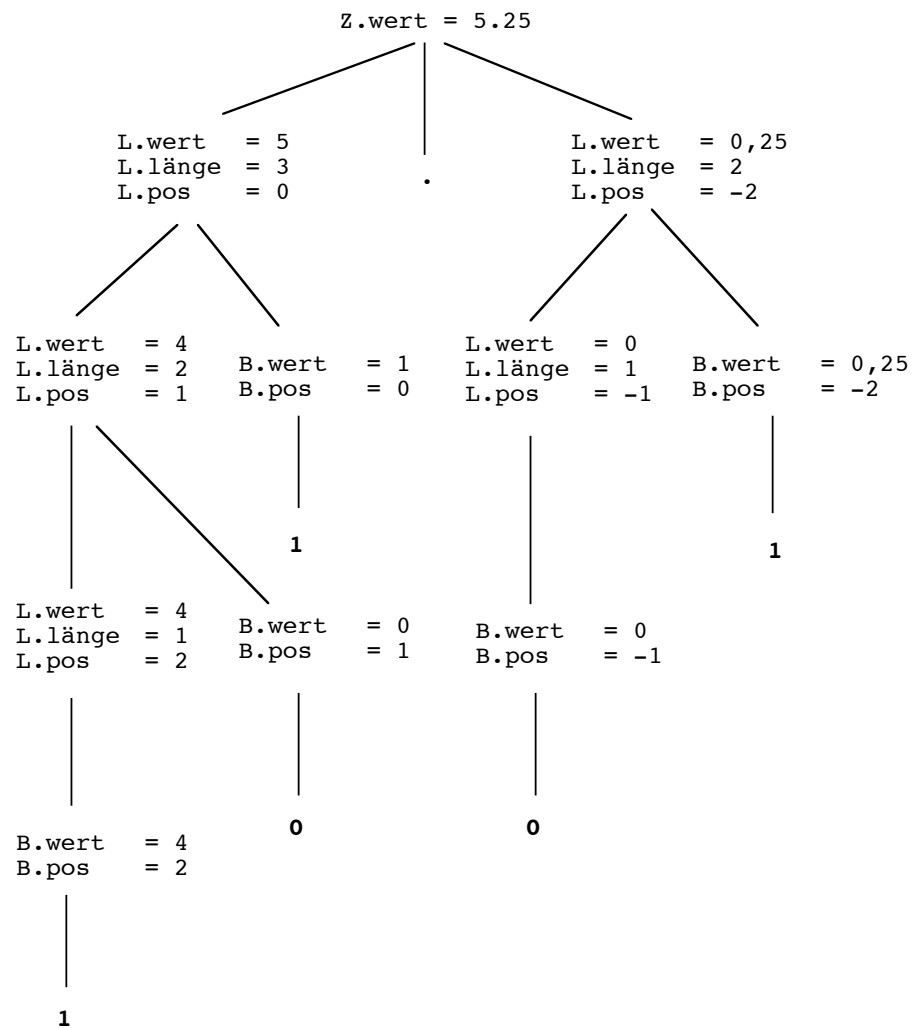
Sei  $G = (N, T, P, Z)$  eine kontextfreie Grammatik mit  $N = \{Z, L, B\}$ ,  $T = \{0, 1, .\}$  und

$$\begin{aligned} \mathcal{A}_S(L) &= \{\text{länge}, \text{wert}\} & \mathcal{A}_I(L) &= \{\text{pos}\} \\ \mathcal{A}_S(Z) &= \{\text{wert}\} & \mathcal{A}_I(Z) &= \emptyset \\ \mathcal{A}_S(B) &= \{\text{wert}\} & \mathcal{A}_I(B) &= \{\text{pos}\} \end{aligned}$$

Diese Grammatik erzeugt Wörter, die als Darstellung von Festkommazahlen im Dualsystem interpretiert werden können. Dabei steht das Symbol  $Z$  für eine Festkommazahl,  $L$  für eine Liste von 0 und 1 und  $B$  für ein Bit, also für 0 oder 1. Als Übersetzung eines Wortes soll der Wert der dargestellten Zahl berechnet und im synthetischen Attribut *wert* von  $Z$  abgelegt werden.

Produktion	Semantische Regeln
$Z \rightarrow L_1 . L_2$	$Z.\text{wert} := L_1.\text{wert} + L_2.\text{wert}$ $L_1.\text{pos} := 0$ $L_2.\text{pos} := -L_2.\text{länge}$
$L \rightarrow L_1 B$	$L.\text{länge} := L_1.\text{länge} + 1$ $L.\text{wert} := L_1.\text{wert} + B.\text{wert}$ $L_1.\text{pos} := L.\text{pos} + 1$ $B.\text{pos} := L.\text{pos}$
$L \rightarrow B$	$L.\text{länge} := 1$ $L.\text{wert} := B.\text{wert}$ $B.\text{pos} := L.\text{pos}$
$B \rightarrow 0$	$B.\text{wert} := 0$
$B \rightarrow 1$	$B.\text{wert} := 2^{B.\text{pos}}$

Das Attribut *länge* gibt die Länge der Liste *L* an und *pos* die Position der Liste oder des Bits (entspricht dem Abstand vom „.“). Das Attribut *wert* repräsentiert den Beitrag zum Gesamtwert der Zahl.



Ausgewerteter attributierter Ableitungsbaum für  $w = 101.01$ .

### 2.1.1 Konstruktion eines Abhängigkeitsgraphen für eine Produktion

Wie man am Beispiel 2.3 deutlich sehen kann, ist die Reihenfolge, in der die Attribute im Ableitungsbaum mit Werten versehen werden können, nicht ohne weiteres zu erkennen.

Um das Problem der Suche nach einer Auswertereihenfolge näher zu untersuchen, benötigt man das Konzept eines **Abhängigkeitsgraphen**.

Der Abhängigkeitsgraph  $g(\pi)$  für die Produktion  $\pi$  wird wie folgt bestimmt:

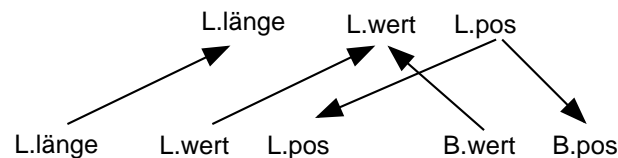
```

for jedes Symbol  $X$  in  $\pi$  do
  for jedes Attribut  $a$  von  $X$  do
    erzeuge einen neuen Knoten in  $g(\pi)$  mit Markierung  $X.a$ 
for jede Regel  $\rho$  zu  $\pi$  do
  bestimmt  $\rho$  das Attribut  $X.a$  in Abhängigkeit der Attribute
   $X_1.b_1, \dots, X_k.b_k$ , so erzeuge eine Kante in  $g(\pi)$  von jedem der
  Knoten  $X_i.b_i$  nach  $X.a$ 

```

#### Beispiel 2.4:

Betrachtet man die Produktion  $L \rightarrow LB$  aus dem Beispiel 2.3 oben, so erhält man den folgenden Abhängigkeitsgraph, wobei die Konvention so ist, dass die Knoten für die Attribute des Symbols auf der linken Seite „oben“ und die der Symbole auf der rechten Seite der Produktion in dieser Reihenfolge „unten“ angeordnet werden:



### 2.1.2 Konstruktion eines Abhängigkeitsgraphen für einen Ableitungsbaum

Der Abhängigkeitsgraph  $G(\tau)$  für den Ableitungsbaum  $\tau$  wird durch Aneinanderfügen von Abhängigkeitsgraphen der Produktionen schrittweise aufgebaut:

```

Erzeuge zunächst die Knoten  $S.a$  für jedes Attribut  $a$  des Startsymbols  $S$ .
Dann durchlaufe  $\tau$  in Präordnung
  Sei  $n$  der nächste Knoten in Präordnung mit Markierung  $X$  in  $\tau$ .
  Ist  $X$  ein nichtterminales Symbol, dann existieren in  $\tau$  die
  Knoten  $X.b$ ,  $b$  Attribut von  $X$ .
  Wurde auf  $X$  in  $\tau$  die Produktion  $\pi$  angewendet,
  so füge  $g(\pi)$  zu  $G(\tau)$  hinzu, wobei die Knoten  $X.b$  in  $G(\tau)$  und  $g(\pi)$ 
  identifiziert werden

```

Der Abhängigkeitsgraph eines Ableitungsbaumes kann nun benutzt werden, um eine **Auswertereihenfolge** für die Attribute im attributierten Ableitungsbaum festzulegen.

Es gilt:

Gibt es eine Kante von  $X.a$  nach  $Y.b$  im Abhängigkeitsgraphen, so muss das Attribut  $a$  von  $X$  vor dem Attribut  $b$  von  $Y$  mit einem Wert versehen werden.

Damit folgt sofort:

- Es existiert genau dann eine Auswertereihenfolge, wenn der Abhängigkeitsgraph des Ableitungsbaums azyklisch ist. (Stichwort: Topologisches Sortieren)





Den Fall 2 kann man noch unterteilen in

- **die Anzahl der Durchläufe ist unbegrenzt.** Jede nicht-zirkuläre attribuierte Grammatik kann auf diese Weise ausgewertet werden. Man durchläuft den Baum wiederholt in einer vorbestimmten Ordnung, etwa in Präordnung. In jedem Knoten berechnet man die Attributwerte, die sich momentan aufgrund der gegebenen Abhängigkeiten und der bereits berechneten Werte bestimmen lassen.
- **die Anzahl der benötigten Durchläufe ist durch eine Konstante begrenzt.** (Diese Eigenschaft ist entscheidbar! [26])

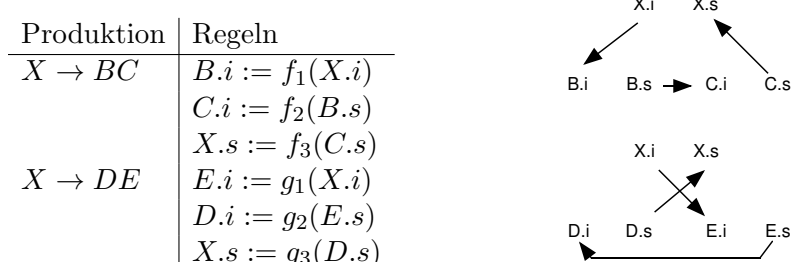
Weiterhin kann man noch bzgl. der Reihenfolge differenzieren, in der die Nachfolgerknoten eines Knotens im Ableitungsbaum ausgewertet werden, z.B.

- immer links-nach-rechts
- immer rechts-nach-links
- alternierend bei jedem Durchlauf
- in einer durch eine Permutation für jede Produktion festgelegten Reihenfolge.

Weiterhin kann man unterscheiden, ob diese Reihenfolge statisch, also für alle Ableitungsbäume der Grammatik, oder dynamisch, je nach vorliegendem Ableitungsbaum, gewählt wird.

### Beispiel 2.6:

Man betrachte die beiden Produktionen  $X \rightarrow BC$  und  $X \rightarrow DE$ . Jedes Symbol habe ein inherites Attribut  $i$  und ein synthetisches Attribut  $s$



Bei der ersten Produktion wäre eine links-nach-rechts Auswertung, bei der zweiten wäre eine rechts-nach-links Auswertung günstig. Können beide Produktionen in einem Ableitungsbaum beliebig oft untereinander auftreten, so ist die Zahl der notwendigen Durchläufe unbegrenzt, sofern eine feste Durchlaufordnung vorgegeben ist.

Andererseits könnte man mit einer durch eine Permutation für jede Produktion festgelegten Reihenfolge den Ableitungsbaum eventuell sogar in einem Durchlauf auswerten!

Eine weitere wichtige Teilklasse von attribuierten Grammatiken, deren Ableitungsbäume sich in einem Durchlauf auswerten lassen, sind die L-attribuierten Grammatiken [36].

**Definition:** Eine attribuierte Grammatik  $G = (N, T, P, S)$  heißt **L-attribuiert**, wenn für jede Produktion  $X \rightarrow Y_1 \dots Y_k$  in  $P$ ,  $X \in N$ ,  $Y_i \in (N \cup T)$ ,  $1 \leq i \leq k$ ,  $k \geq 0$  und für jedes  $j$ ,  $1 \leq j \leq k$  gilt:

Ist  $a \in \mathcal{A}_I(Y_j)$ , dann ist  $a$  nur abhängig von Attributen aus  $\mathcal{A}_I(X) \cup \bigcup_{i=1}^{j-1} \mathcal{A}(Y_i)$

**Bem.:** Jede S-attribuierte Grammatik ist auch L-attribuiert.

Die attribuierte Grammatik für die Dualzahlen aus dem Beispiel 2.3 ist nicht L-attribuiert, da die Attributierung der Produktion  $Z \rightarrow L_1 \cdot L_2$  die Bedingungen verletzt.

**Beispiel 2.7:**

Die folgende L-attributierte Grammatik beschreibt beispielhaft Typ-Deklarationen in einem Programm.

Es sei  $G = (N, T', P, D)$  mit  $N = \{D, L, T\}$ ,  $T = \{\text{int}, \text{float}, \text{id}, ', '\}$ . Weiterhin gelte, dass  $L$  ein inherites Attribut *inh*,  $T$  ein vom Scanner gesetztes synthetisches Attribut *type* und  $\text{id}$  ein synthetisches Attribut *entry* besitzt. Alle anderen Symbole haben keine Attribute. Die hier verwendete Funktion *addType* fügt einen neuen Eintrag für eine Variable mit zugehörigem Typ in die Symboltabelle ein. Die Produktionen mit den semantischen Regeln sind:

Produktion	Semantische Regeln
$D \rightarrow T L$	$L.inh := T.type$
$T \rightarrow \text{int}$	$T.type = \text{integer}$
$T \rightarrow \text{float}$	$T.type = \text{float}$
$L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh$ $addType(\text{id}.entry, L.inh)$
$L \rightarrow \text{id}$	$addType(\text{id}.entry, L.inh)$

Die Funktion *addType* ist natürlich eine Funktion mit Seiteneffekten. Da das Einsetzen eines Eintrags die anderen Einträge in der Symboltabelle nicht verändert, kann auf das Festlegen einer speziellen Auswertereihenfolge verzichtet werden.

Wir werden sehen, dass sich L-attributierte Grammatiken besonders in Verbindung mit Top-Down Parsern für die Übersetzung eignen, da in diesem Fall ebenfalls kein expliziter Ableitungsbaum benötigt wird.

Die Vorgehensweise wird klarer ersichtlich, wenn man den Zeitpunkt für die Auswertung der semantischen Regeln relativ zum Parsingprozess in einer gewissen Form festlegen kann. Dies ist besonders wichtig, wenn man semantische Regeln mit Seiteneffekten verwenden möchte.

**Definition:** Ein **syntax-gesteuertes Übersetzungsschema (SDTS)** ist eine andere Darstellung einer attribuierten Grammatik  $G = (N, T, P, S)$ . Die kontextfreien Produktionen  $X \rightarrow Y_1 \dots Y_k$ ,  $X \in N$ ,  $Y_i \in N \cup T$ ,  $1 \leq i \leq k$ ,  $k \geq 0$  werden um die semantischen Regeln erweitert. Die Produktionen haben dann die Form  $X \rightarrow \alpha_0 Y_1 \alpha_1 \dots \alpha_{k-1} Y_k \alpha_k$ , wobei die  $\alpha_i$  die Form  $\alpha_i = \varepsilon$  oder  $\alpha_i = \{ \text{semantische Regeln} \}$  haben. Die  $\alpha_i$  heißen **Aktionen**.

**Bedeutung:** Durch die Einbettung der semantischen Regeln in die Produktion kann der Zeitpunkt der Ausführung der Regeln implizit festgelegt werden. Die Bedeutung dieser Notation sei dabei so, dass die semantischen Regeln in  $\alpha_i$  ausgeführt werden, nachdem die Ableitung für  $Y_i$  erstellt wurde und bevor die Ableitung von  $Y_{i+1}$  beginnt. Da Aktionen auch Seiteneffekte, wie etwa eine **print**-Anweisung, enthalten können, ist die Reihenfolge der Ausführung der Aktionen wichtig.

Offensichtlich müssen die folgenden Regeln erfüllt sein, damit ein SDTS korrekt definiert ist: Für alle  $1 \leq i \leq k$  muß gelten:

- 1) Der Wert eines inheriten Attributs  $a$  von  $Y_i$  muß in einer der Aktionen  $\alpha_0, \dots, \alpha_{i-1}$  berechnet werden. Üblicherweise geschieht dies zum spätesten Zeitpunkt, also in  $\alpha_{i-1}$ .
- 2) Der Wert eines synthetischen Attributs  $b$  von  $X$  kann erst berechnet werden, wenn alle zur Berechnung benötigten Werte bekannt sind. Üblicherweise geschieht dies auch zum spätesten Zeitpunkt, also in  $\alpha_k$ .

Eine L-attributierte Grammatik erfüllt diese Bedingungen in jedem Fall.

## 2.3 Auswertung L-attributierter Grammatiken beim Top-Down Parsing

Wenn man eine LL-Grammatik mit einer L-Attributierung gegeben hat, kann man den Parsing- und den Auswerteprozess parallel durchführen.

Will man einen tabellengesteuerten Top-Down-Parser benutzen, reicht es im Gegensatz zu den S-attributierten Grammatiken nicht, einfach den Parsing-Stack zu erweitern.

Man benötigt zwar auch in diesem Fall einen zusätzlichen Attributwerte-Stack, der jedoch leider nicht genau synchron zum Parsing-Stack wächst und schrumpft. Man benötigt also zusätzlichen Aufwand zur Organisation des Werte-Stacks.

### Beispiel 2.8:

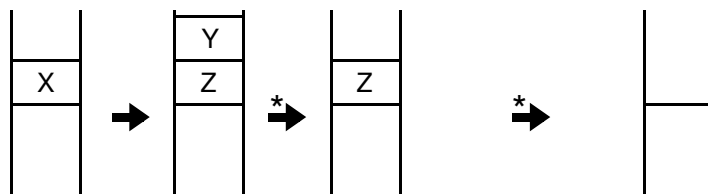
Am Beispiel der Regel  $X \rightarrow \alpha_0 Y \alpha_1 Z \alpha_2$  soll die Arbeitsweise erläutert werden.

In der Anfangssituation steht auf dem Parsing-Stack  $X$  und auf dem Werte-Stack steht ein Record, der die Werte der inheriten Attribute von  $X$  enthält. Die nächste anzuwendende Produktion sei  $X \rightarrow YZ$ .

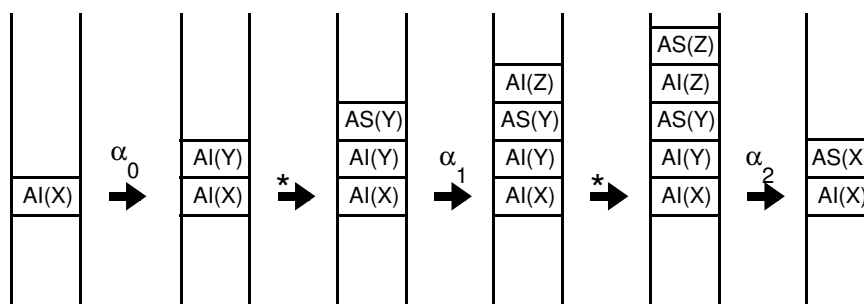
Zunächst werden die inheriten Attribute von  $Y$  berechnet. Da man dazu die inheriten Attribute von  $X$  benötigt, müssen diese sich noch auf dem Werte-Stack befinden. Ist die Ableitung von  $Y$  beendet und auf dem Parsing-Stack wird  $Z$  das oberste Symbol, so müssen die berechneten synthetischen Attribute von  $Y$  ebenfalls auf dem Werte-Stack gelegt werden, da alle diese Werte eventuell zur Berechnung der inheriten Attribute von  $Z$  benötigt werden.

Wenn also ein Symbol vom Stack gelöscht wird, sollen auf dem Werte-Stack sowohl die inheriten als auch die synthetischen Attribute dieses Symbols stehen. Erst nach Berechnung der synthetischen Attribute von  $X$  können die Attributwerte von  $Y$  und  $Z$  vom Werte-Stack gelöscht werden.

Parsing-Stack:



Werte-Stack:

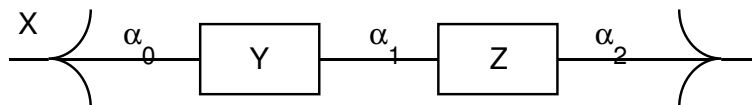


Einfacher und natürlicher ist die Verbindung eines Parsers, der nach der Methode des rekursiven Abstiegs arbeitet, mit einer L-Attributierung. Als Beispielproduktion sei  $X \rightarrow \alpha_0 Y \alpha_1 Z \alpha_2$  gewählt, wobei  $X, Y, Z \in N$  gelte und  $\alpha_0, \alpha_1, \alpha_2$  semantische Aktionen sind.

Es soll nun ein Parser konstruiert werden, der nach der Methode des rekursiven Abstiegs arbeitet. (**recursive descent parser**)

Wie üblich wird zunächst für jedes nichtterminale Symbol ein Syntaxgraph konstruiert. Man markiert dabei zusätzlich die Kanten des Graphen mit den Aktionen. Dann vereinfacht man die Syntaxgraphen.

Ausschnitt aus dem Syntaxgraph für  $X$ :



Für jeden Syntaxgraphen eines nichtterminalen Symbols  $X$  erzeugt man dann wie üblich eine Prozedur mit Namen  $X$ . Diese Prozedur bekommt für jedes inherite Attribut in  $\mathcal{A}_I(X)$  einen Formalparameter und gibt die Werte der synthetischen Attribute in  $\mathcal{A}_S(X)$  (z.B. als Record) zurück. Weiterhin hat die Prozedur  $X$  lokale Variablen für jedes synthetische Attribut von  $X$  und jeweils eine lokale Variable für jedes Attribut eines jeden im Syntaxgraphen auftretenden Symbols. Tritt ein Symbol mehrfach im Syntaxgraphen auf, so müssen entsprechend viele Exemplare der lokalen Variablen erzeugt werden.

Neben dem üblichen Code zur Steuerung der Ableitung beim rekursiven Abstieg werden

- bei terminalen Symbolen  $a$  die Werte der synthetischen Attribute von  $a$  in die lokalen Variablen gespeichert (als Initialisierung dieser Variablen)
- bei nichtterminalen Symbolen  $Y$  bekommen Prozeduraufrufe die Form  
 $\text{syn}_1, \dots, \text{syn}_r := Y(\text{inh}_1, \dots, \text{inh}_k)$ ,  
wobei  $\text{inh}_1, \dots, \text{inh}_k$  die lokalen Variablen für die inheriten und  $\text{syn}_1, \dots, \text{syn}_r$  für die synthetischen Attribute von  $Y$  sind.  
(Hier ist der Einfachheit halber eine Syntax angenommen worden, die es einer Prozedur erlaubt, mehrere Werte zurückzugeben.)
- die Aktionen durch entsprechende Code-Fragmente in die Parsing-Prozedur übertragen. In den semantischen Regeln auftretende Attribute werden dabei durch die korrespondierenden lokalen Variablen ersetzt.
- beim Verlassen der Prozedur die Werte der synthetischen Attribute zurückgeben.

### Beispiel 2.9:

Für die oben angegebene Beispielproduktion  $X \rightarrow \alpha_0 Y \alpha_1 Z \alpha_2$  wird das folgende Code-Fragment in der Parsingprozedur für das Nichtterminale  $X$  erzeugt.  $\text{AI}(X)$  bezeichne dabei die inheriten,  $\text{AS}(X)$  die synthetischen und  $\text{W}(\text{AS}(X))$  die Wertebereiche der synthetischen Attribute von  $X$ .

```

procedure X (AI(X)) : W(AS(X))
var AS(X), AI(Y), AS(Y), AI(Z), AS(Z)
begin
  ...

   $\alpha'_0$   /* Codefragment zur Berechnung der AI(Y) gemäß  $\alpha_0$  */
  AS(Y) := Y(AI(Y));    /* Aufruf von Y liefert Werte der AS(Y) */

   $\alpha'_1$   /* Codefragment zur Berechnung der AI(Z) gemäß  $\alpha_1$  */
  AS(Z) := Z(AI(Z));    /* Aufruf von Z liefert Werte der AS(Z) */

   $\alpha'_2$   /* Codefragment zur Berechnung der AS(X) gemäß  $\alpha_2$  */
  return AS(X)
  ...
end
```

## 2.4 Umformungen der Grammatiken

Manchmal ist es notwendig, Transformationen an den Produktionen der Grammatik vorzunehmen, damit ein deterministisches Parsen mit gleichzeitiger Auswertung der Attribute möglich ist oder aber damit gewisse Regeln (mit Seiteneffekten) zur „richtigen“ Zeit durchgeführt werden. Welche Auswirkungen haben derartige Transformationen auf die Attributierung, oder wie muss man die Attributierung ändern, damit der Wert der Übersetzung gleich bleibt?

Hier sollen exemplarisch nur zwei Fälle betrachtet werden:

- 1) Das Entfernen von Aktionen, die *in* der rechten Seite von Produktionen eines bottom-up parsebaren SDTS auftreten, um ein deterministisches Bottom-Up-Parsing mit gleichzeitiger Attributauswertung zu ermöglichen.
- 2) Das Entfernen linksrekursiver Produktionen bei S-attribuierten Grammatiken, damit ein deterministisches Top-Down-Parsing möglich wird.

Betrachten wir zunächst den Fall 1).

Man ersetzt eine Aktion  $a$ , die nicht am Ende, sondern innerhalb einer Produktion ausgeführt werden muss, durch ein neues nichtterminales Symbol (Marker) und führt eine  $\varepsilon$ -Produktion für dieses Symbol mit Aktion  $a$  ein.

### Beispiel 2.10:

Das folgende SDTS definiert eine Übersetzung einfacher arithmetischer Ausdrücke in Postfix-Notation, wobei die Übersetzung als Ausgabe der `print`-Anweisungen erscheint.

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +T\{\text{print}(' + ')\}R \mid -T\{\text{print}(' - ')\}R \mid \varepsilon \\ T &\rightarrow \mathbf{number}\{\text{print}(\mathbf{number.val})\} \end{aligned}$$

Zum Beispiel führt die Eingabe  $3 - 2 + 1$  zur Ausgabe  $3 2 - 1 +$ . Wegen der Seiteneffekte der `print`-Anweisung müssen die semantischen Aktionen aber auch genau an diesen Stellen ausgeführt werden!

Eine Übertragung des obigen SDTS in eine S-attribuierte Grammatik, die eine entsprechende Übersetzung parallel zum Parsingprozess ermöglicht, erfordert die Einführung der beiden neuen nichtterminalen Symbole  $M$  und  $N$  mit jeweiligen  $\varepsilon$ -Produktionen:

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +TMR \mid -TNR \mid \varepsilon \\ T &\rightarrow \mathbf{number}\{\text{print}(\mathbf{number.val})\} \\ M &\rightarrow \varepsilon\{\text{print}(' + ')\} \\ N &\rightarrow \varepsilon\{\text{print}(' - ')\} \end{aligned}$$

Die zusätzlich eingeführten Symbole und Produktionen erlauben weiterhin ein deterministisches bottom-up Parsen, wobei die semantischen Aktionen jetzt wie üblich bei jedem Reduktionsschritt ausgeführt werden.

Im Fall 2) kann man die Linksrekursionen mit dem „üblichen“ Verfahren entfernen, muss aber die Attributierung ändern:

Seien  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m$  alle A-Produktionen, deren rechte Seite mit  $A$  beginnt und seien  $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$  alle restlichen A-Produktionen. Weiterhin gelte  $\alpha_i \neq \varepsilon$  für  $1 \leq i \leq m$ . Man ersetzt diese Produktionen durch

$$A \rightarrow \beta_1 A', \dots, A \rightarrow \beta_n A' \text{ und } A' \rightarrow \alpha_1 A', \dots, A' \rightarrow \alpha_m A', A' \rightarrow \varepsilon$$

wobei  $A'$  ein neues nichtterminales Symbol ist.

Man beachte, dass durch diese Änderung an der Grammatik andere Ableitungsbäume entstehen, in denen die synthetischen Attribute nicht mehr so wie in der ursprünglichen Grammatik berechnet werden können, da die benötigten Werte „an falscher Stelle im Baum“ stehen.

In der transformierten Grammatik bekommt das neu eingeführte Symbol  $A'$  deshalb ein inherites und ein synthetisches Attribut. Das neu eingeführte inherite Attribut benötigt man, um die Werte synthetischer Attribute in dem jetzt veränderten Ableitungsbaum nach „unten“ reichen zu können, damit sie als Argumente für die entsprechenden semantischen Regeln zur Verfügung stehen. Das synthetische Attribut wird benötigt, um den berechneten Wert wieder nach oben zum nichtterminalen Symbol  $A$  zu transportieren. Die semantischen Aktionen müssen natürlich ebenfalls angepasst werden.

Diese Transformation soll hier nur beispielhaft an einer S-attribuierten Grammatik vorgestellt werden:

### Beispiel 2.11:

Die folgende links-rekursive, S-attribuierte Grammatik sei gegeben. Die auftretenden nichtterminalen Symbole haben wie das Token **number** alle nur das synthetische Attribut *val*.

Produktion	sem. Regel
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val := E_1.val - T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow (E)$	$T.val := E.val$
$T \rightarrow \mathbf{number}$	$T.val := \mathbf{number}.val$

Entfernt man die Links-Rekursionen und ändert die Attributierung entsprechend, so erhält man eine rechts-rekursive Grammatik. Das neu hinzugekommene nichtterminale Symbol  $R$  bekommt ein inherites Attribut  $i$  und ein synthetisches Attribut  $s$ . Insgesamt erhält man eine äquivalente L-attribuierte Grammatik, die in Form eines SDTS notiert wird.

$$\begin{aligned}
E &\rightarrow T \{R.i := T.val\} R \{E.val := R.s\} \\
R &\rightarrow +T \{R_1.i := R.i + T.val\} R_1 \{R.s := R_1.s\} \\
R &\rightarrow -T \{R_1.i := R.i - T.val\} R_1 \{R.s := R_1.s\} \\
R &\rightarrow \varepsilon \{R.s := R.i\} \\
T &\rightarrow (E \{T.val := E.val\} ) \\
T &\rightarrow \mathbf{number} \{T.val := \mathbf{number}.val\}
\end{aligned}$$

Konstruiert man jetzt für die obige L-attribuierte Grammatik einen recursive-descent-Parser wie im Abschnitt 2.3 beschrieben, so erhält man die folgenden Prozeduren:

```
procedure T(): integer;
Tval, Eval, numberval: integer;
begin
    if token = "(" then
        begin
            nextToken();
            Eval := E();
            Tval := Eval;
            if not (token = ")") then
                error();
            else
                begin
                    nextToken();
                    return(Tval);
                end;
            end;
        else if token = "number" then
            begin
                numberval := lexval;
                nextToken();
                Tval := numberval;
                return(Tval);
            end;
        else error();
```

```
procedure E(): integer;
Eval, Tval, Rs, Ri : integer;
begin
    Tval := T();
    Ri := Tval;
    Rs := R(Ri);
    Eval := Rs;
    return(Eval);
end;
```

```
procedure R(Ri: integer) : integer;
Rs, Tval, R1s, R1i : integer;
begin
  if token = "+" then
    begin
      /* Aufruf des lexikalischen Scanners, */
      nextToken(); /* setzt die globale Variable token */
      Tval := T();
      R1i := Ri + Tval;
      R1s:= R(R1i);
      Rs := R1s;
      return(Rs);
    end;
  else if token = "-" then
    begin
      /* Aufruf des lexikalischen Scanners */
      nextToken(); /* setzt die globale Variable token */
      Tval := T();
      R1i := Ri - Tval;
      R1s:= R(R1i);
      Rs := R1s;
      return(Rs);
    end;
  else if token = ")" or token = "$" then /* Symbole in FOLLOW(R) */
    begin
      Rs := Ri;
      return(Rs);
    end;
  else error();
end;
```



### 3 Semantische Analyse

Ein Variablenname, etwa „`listenElement`“, repräsentiert in einer Programmiersprache ein Objekt, das mehrere Eigenschaften besitzt:

- 1) Eine **Speicheradresse**, ab der dieses Objekt im Speicher abgelegt ist. Diese Speicheradresse wird vom Compiler, manchmal in Verbindung mit dem Linker, festgelegt und kann sich während des Programmlaufs durchaus ändern (etwa bei lokalen Variablen, die in rekursiven Prozeduren auftreten).
- 2) Eine Codierung des Objektes selbst, d.h. die zugeordneten Speichereinheiten und das darin abgelegte Bitmuster, das den **Wert** des Objektes darstellt.

Um den Wert eines Objektes manipulieren zu können, muß man wissen, wie lang diese Codierung ist und wie man sie zu interpretieren hat. Diese Information zu dem Objekt wird durch eine **Typ-Information** festgelegt und dem Objekt als *tag* oder der Variablen per Deklaration zugeordnet. In den meisten Programmiersprachen wird dies durch eine Typ-Angabe bei der Variablendeklaration eingeführt. In objektorientierten Sprachen wird der Begriff „Typ“ durch „Klasse“ ersetzt.

Jeder Teilausdruck eines arithmetischen Ausdrucks hat natürlich ebenfalls einen Typ (oder sogar bei gewissen Programmiersprachen mehrere Typen, z.B. sind nicht negative Integer in Modula-2 vom Typ `INTEGER` und `CARDINAL`). Die in der Programmiersprache verfügbaren Operatoren verlangen üblicherweise Operanden eines speziellen Typs. Ein Compiler kann in gewissem Rahmen prüfen, ob die durch die Sprachdefinition der Programmiersprache gegebenen Beschränkungen bzgl. der Typen im vorgelegten Programm eingehalten werden (**Type-Checking**) und ob eventuell Typen automatisch umgewandelt werden müssen.

Bei einer Programmiersprache mit **starker Typ-Prüfung** (strongly typed language) kann vom Compiler im gewissen Umfang garantiert werden, dass kein Typ-Fehler zur Laufzeit auftritt, also die Operanden einer Operation den „richtigen“ Typ haben.

Diese Typ-Prüfung kann, speziell bei einfachen Typregeln, während des Parsings bzw. bei der Zwischencode-Erzeugung geschehen (etwa Pascal) oder aber auch bei entsprechend komplexen Regeln (etwa Ada, C++ oder Java) in einem zusätzlichen Pass des Compilers geschehen (**statische Typ-Prüfung**).

Andererseits kann diese Überprüfung auch ganz oder teilweise erst zur Laufzeit durchgeführt werden (**dynamische Typ-Prüfung**). Man denke etwa an vom Compiler generierten Code zum Testen auf Bereichsüberschreitungen bei Arrayzugriffen auf dynamische Arrays oder die dynamische Auswahl virtueller Funktionen in C++ (dynamisches Binden).

Auch gibt es Sprachen (etwa Smalltalk oder Scheme), in denen keine Typdeklaration vorgesehen ist. In diesen Fällen ist jedem Datenobjekt auch eine Kodierung des Typs (bzw. in Smalltalk der Klasse) zugeordnet und diese Kodierung muß ebenfalls gespeichert werden (**tag-System**).

Eine andere Klasse von Sprachen, wie etwa ML, Caml oder Haskell oder neuerdings Swift, erlaubt die Deklaration von Variablen ohne Angabe eines Typs, führt aber trotzdem eine starke Typ-Prüfung durch. Dies geschieht durch ein komplexes System, bei dem durch logisches Schließen durch den Gebrauch der Variablen auf deren Typ geschlossen wird und damit eine konsistente Verwendung der Variablen gewährleistet wird.

Den Prozeß des Ableitens von Typen für Konstrukte der Programmiersprache aus dem Gebrauch der Konstrukte nennt man **Typ-Inferenz**. Zum Beispiel kann aus der Anweisung `x := 1.0` geschlossen werden, dass `x` vom Typ `real` sein muss und dass daher der nachfolgende Teilausdruck `a[x] := 0` zu einer Fehlermeldung wegen Typ-Unverträglichkeit führt.

Wird der Typ eines Ausdrucks dagegen aus den bekannten Typen der Operanden bestimmt, spricht man von **Typ-Synthese**. Diese Vorgehensweise erfordert die Deklaration von Namen vor ihrer Verwendung.

### 3.1 Typ-Ausdrücke

Es soll im folgenden jedem Objekt des Programms ein **Typ-Ausdruck** zugeordnet werden. Typ-Ausdrücke sind Terme, die mittels Konstruktoren aus elementaren Typen gebildet werden. Dazu werden rekursiv ausgehend von sogenannten **Basistypen** immer komplexere Typ-Ausdrücke aufgebaut.

Was ein Basistyp ist, hängt von der speziellen Programmiersprache ab; so besitzt z.B. FORTRAN 90 den Basistyp *complex*, Pascal jedoch nicht. Das **Typ-System** einer Programmiersprache ist ein Satz von Regeln, mit denen allen Objekten eines Programms ein Typ-Ausdruck zugeordnet werden kann.

#### Definition einfacher Typ-Ausdrücke

- 1) Ein Basistyp ist ein Typ-Ausdruck. Zu den Basistypen gehören etwa: *boolean*, *char*, *integer*, *real*, *void*.

**Bemerkung:** Auch Aufzählungen wie etwa *1..7* oder (*mo*, *di*, *mi*, *do*, *fr*) und Unterbereiche von einigen Basistypen sowie der spezielle (Fehler)-Typ *type-error* können als Basistypen angesehen werden. Aufzählungs- und Unterbereichstypen bringen jedoch viele Probleme in das Typ-System und werden daher in modernen Sprachen meist nicht mehr zu den Basistypen gezählt.

- 2) Sind Namen als Abkürzungen für Typen erlaubt, dann sind diese Namen ebenfalls Typ-Ausdrücke.

**Beispiel:** Durch `type Zeichen = char` wird *Zeichen* ein Typ-Ausdruck.

- 3) Ist  $T$  ein Typ-Ausdruck, so ist  $\text{array}(T)$  ebenfalls ein Typ-Ausdruck, der einen Feld-Typ beschreibt.

**Bemerkung:** *array* ist ein Beispiel eines **Typ-Konstruktors**, der einen neuen Typ erzeugt. In manchen Programmiersprachen betrachtet man den Indexbereich des Feldes als mit zum Typ gehörig. In diesem Fall hätte der Typ-Ausdruck die Form  $\text{array}(I, T)$ , wobei  $I$  ein Typ-Ausdruck ist, der einen Unterbereich von *integer* oder eine Aufzählung bezeichnet. Ist der untere Index des Feldes fest, so reicht es aus, die Länge des Feldes mit in den Typ-Ausdruck aufzunehmen.

**Beispiel:** Durch die Deklaration `var A : array[1..10] of integer` ist  $\text{array}(\text{integer})$  oder auch  $\text{array}(10, \text{integer})$  Typ-Ausdruck für  $A$ .

- 4) Ist  $T$  ein Typ-Ausdruck, so ist  $\text{list}(T)$  ebenfalls ein Typ-Ausdruck, der einen Listen-Typ beschreibt.

Dieser Typ-Konstruktor wird natürlich nur dann benötigt, wenn Listen elementare Datenstrukturen der Programmiersprache sind.

- 5) Sind  $T_1, \dots, T_r$  Typ-Ausdrücke und sind  $\text{name}_1, \dots, \text{name}_r$  unterschiedliche Namen, dann ist  $\text{record}(\text{name}_1:T_1 \times \dots \times \text{name}_r:T_r)$  ein Typ-Ausdruck, der eine Record-Konstruktion beschreibt.

**Beispiel:**

```

type t=record
  x:integer;
  y:array[1..15] of char
end

```

ordnet  $t$  den Typ-Ausdruck  $record(x: integer \times y: array(15, char))$  zu.

**Bemerkung** Analog würde man in objektorientierten Sprachen einen Typ-Konstruktor für neue Klassen definieren.

- 6) Sind  $T_1, \dots, T_r$  Typ-Ausdrücke mit  $r \geq 1$ , dann ist  $T_1 \times \dots \times T_r$  ein Typ-Ausdruck, der ein Tupel mit  $r$  Komponenten von Typ  $T_1, \dots, T_r$  beschreibt.

**Bemerkung:** Mit diesem Konstrukt kann man etwa die Typen der Parameter einer Funktion zu einem Typ zusammenfassen oder aber auch bei Funktionen mit mehreren Rückgabewerten die Typen festlegen.

- 7) Ist  $T$  ein Typ-Ausdruck, dann ist  $pointer(T)$  ein Typ-Ausdruck, der einen Zeigertyp auf ein Objekt vom Typ  $T$  beschreibt.

**Beispiel:**  $\text{var } p: \uparrow integer$  ordnet der Variablen  $p$  den Typ-Ausdruck  $pointer(integer)$  zu. Dieser Typ-Konstruktor wird natürlich nur benötigt, wenn die Programmiersprache die Verwendung von Pointer-Variablen zulässt.

- 8) Sind  $T_1$  und  $T_2$  Typ-Ausdrücke, dann ist  $T_1 \rightarrow T_2$  ein Typ-Ausdruck, der eine Funktion mit einem Parameter vom Typ  $T_1$  beschreibt, die ein Objekt vom Typ  $T_2$  zurückgibt.

**Bemerkung:** Mit der Definition von Tupeln kann man natürlich auch Prozeduren mit mehreren Parametern definieren,

Man beachte, dass der Operator  $\times$  linksassoziativ ist und eine größere Priorität als der rechtsassoziative Operator  $\rightarrow$  hat. Daher ist der Typ-Ausdruck

$char \times char \rightarrow integer \rightarrow pointer(integer \rightarrow integer)$  als  
 $(char \times char) \rightarrow (integer \rightarrow pointer(integer \rightarrow integer))$  zu interpretieren.

### Beispiel 3.1:

Die Deklaration  $\text{function } f(a, b: char): \uparrow integer$  in Pascal oder die äquivalente Deklaration  $\text{int } *f(char a, char b)$  in C liefert als Typ-Ausdruck

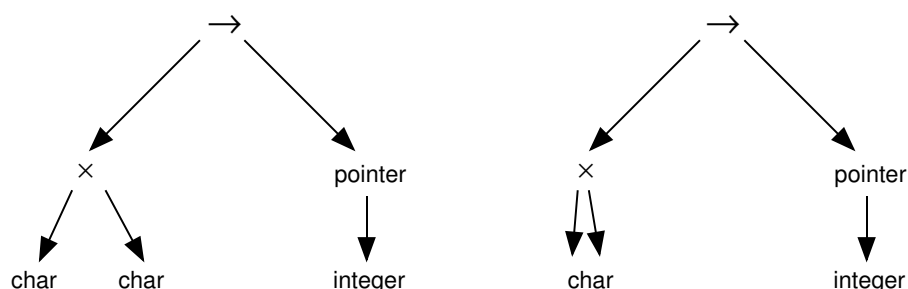
$char \times char \rightarrow pointer(integer)$ .

während in C die Deklaration  $\text{int } (*f)(char a, char b)$  den Typausdruck

$pointer(char \times char \rightarrow integer)$

liefern würde.

Typ-Ausdrücke werden auch häufig als Bäume oder Graphen dargestellt. Man erhält Graphen, wenn man Teilausdrücke eines Typausdrucks nicht doppelt aufführt, sondern mehrere Kanten auf diesen Teilausdruck zulässt wie etwa in der rechten Abbildung:



**Beispiel 3.2:**

Als Beispiel soll hier die Kodierung von Typ-Ausdrücken im Java class File Format vorgestellt werden. Typ-Ausdrücke werden dort „Deskriptoren“ genannt. Typ-Ausdrücke werden als Zeichenketten im UTF-8 Format kodiert. Es gibt eine Reihe von Basistypen und nur einen Typ-Konstruktor:

Typ	Kodierung	Bemerkung
byte	B	signed byte
char	C	Unicode Zeichen
double	D	Floating-Point-double
float	F	Floating-Point single
int	I	integer
long	J	long integer
reference	L<classname>	Objekt der Klasse <classname>
short	S	short
boolean	Z	boolean
reference	[	array

Der <classname> repräsentiert dabei einen vollen qualifizierten Klassen- oder Interfacenamen. Der kodierte Typ-Ausdruck für eine Instanzenvariable vom Typ Integer ist also einfach I. Der Typ-Ausdruck für eine Instanzenvariable vom Typ Object ist Ljava/lang/Object; . Hat man ein 2-dimensionales Feld d durch float d[] [] deklariert, ist der zugehörige Typ-Ausdruck [[F.

Die Kodierung von Typ-Ausdrücken von Methoden haben die Form:

( <Typ-Ausdrücke der Parameter > ) <Typ-Ausdruck des Rückgabewerts>

Der Rückgabebetyp void wird dabei mit V kodiert.

Der Deklaration Object myMethod (int i, double d, Thread t) würde also die Kodierung (IDLjava/lang/Thread;)Ljava/lang/Object; zugeordnet.

### 3.2 Einfache Typ-Synthese

Den Prozess des Ableitens von Typ-Ausdrücken für Konstrukte der Programmiersprache aus verfügbaren Typinformationen nennt man **Typ-Synthese**. Man diesen Vorgang bei einfachen Programmiersprachen während des Parsens durchführen.

Als Beispiel soll ein SDTS angegeben werden, das diese Aufgabe erfüllt. Als einziges Attribut tritt hier das synthetische Attribut *type* auf. Im Deklarationsteil wird das Attribut benutzt, um den Typ-Ausdruck der Deklaration zu konstruieren. Im beispielhaften Ausführungsteil enthält das Attribut *type* den Typ-Ausdruck des Teilausdrucks. Bei der Anwendung von Operationen wird geprüft, ob die Operanden den „richtigen“ Typ haben. Andernfalls wird eine Fehlermeldung ausgegeben.

Ein einfaches Übersetzungsschema zur Typ-Synthese und Typ-Prüfung

$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow \text{id} : T$	$\{ \text{addtype}(\text{id.entry}, T.type) \}$
$T \rightarrow \text{char}$	$\{ T.type := \text{char} \}$
$T \rightarrow \text{integer}$	$\{ T.type := \text{integer} \}$
$T \rightarrow \uparrow T_1$	$\{ T.type := \text{pointer}(T_1.type) \}$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$\{ T.type := \text{array}(\text{num.val}, T_1.type) \}$
$E \rightarrow \text{literal}$	$\{ E.type := \text{char} \}$
$E \rightarrow \text{num}$	$\{ E.type := \text{integer} \}$
$E \rightarrow \text{id}$	$\{ E.type := \text{lookup}(\text{id.entry}) \}$
$E \rightarrow E_1 \text{ op } E_2$	$\{ E.type := \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \\ \text{then integer else type-error} \}$
$E \rightarrow E_1[E_2]$	$\{ E.type := \text{if } E_2.type = \text{integer} \text{ and } E_1.type = \text{array}(s, t) \\ \text{then } t \text{ else type-error} \}$
$E \rightarrow E_1 \uparrow$	$\{ E.type := \text{if } E_1.type = \text{pointer}(t) \\ \text{then } t \text{ else type-error} \}$

Bemerkungen zum Übersetzungsschema:

- Hier wird angenommen, dass Felder als unteren Index immer die 1 haben und die Feldlänge Teil des Typs sind.
- `addtype(st, T)` speichert den Typ-Ausdruck  $T$  in dem Eintrag der Symboltabelle, auf den `st` zeigt. Bei Programmiersprachen, in denen eine Vielzahl von Typen definiert werden kann, wird der Compiler alle definierten Typen in einer **Typ-Tabelle** festhalten und in der Symboltabelle nur einen Verweis auf den entsprechenden Eintrag in der Typ-Tabelle machen.
- `lookup(st)` liefert den Typ-Ausdruck des Eintrags in der Symboltabelle, auf den `st` weist.
- Als Folge der Seiteneffekte befinden sich für dieses Übersetzungsschema die Informationen in der Symboltabelle bevor sie abgefragt werden.
- Durch Produktionen wie etwa  $E \rightarrow E < E$  könnte man Ausdrücke vom Typ *boolean* erzeugen.

Erweiterung des vorangehenden Beispiels auf Wertzuweisungen (Wertzuweisungen haben hier angenommen den zugeordneten Basis-Typ `void`):

$$\begin{array}{ll}
P \rightarrow D; S & \\
S \rightarrow \text{id} := E & \{ S.type := \text{if lookup(id.entry) = } E.type \\
& \quad \text{then void else type-error} \} \\
S \rightarrow \text{if } E \text{ then } S_1 & \{ S.type := \text{if } E.type = \text{boolean} \\
& \quad \text{then } S_1.type \text{ else type-error} \} \\
S \rightarrow \text{while } E \text{ do } S_1 & \{ S.type := \text{if } E.type = \text{boolean} \\
& \quad \text{then } S_1.type \text{ else type-error} \} \\
S \rightarrow \text{begin } L \text{ end} & \{ S.type := L.type \} \\
L \rightarrow S; L_1 & \{ L.type := \text{if } S.type = \text{void} \\
& \quad \text{then } L_1.type \text{ else type-error} \} \\
L \rightarrow \varepsilon & \{ L.type := \text{void} \}
\end{array}$$

Hier folgt jetzt eine Erweiterung des vorigen Beispiels auf einfache Funktionen mit einem Parameter. Die erste Produktion soll eine stark vereinfachte Form der Deklaration einer Funktion repräsentieren.

$$\begin{array}{ll}
T \rightarrow T_1 ' \rightarrow' T_2 & \{ T.type := T_1.type \rightarrow T_2.type \} \\
E \rightarrow E_1(E_2) & \{ E.type := \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \\
& \quad \text{then } t \text{ else type-error} \}
\end{array}$$

### 3.3 Äquivalenz von Typ-Ausdrücken

Bei der Typ-Prüfung muss immer wieder getestet werden, ob zum Beispiel der Typ-Ausdruck *typ1* eines Operanden mit dem Typ-Ausdruck *typ2* eines Operanden einer Operation zusammenpasst. Man sagt in diesem Fall, dass die Typen *typ1* und *typ2* **äquivalent** sind.

Solange keine Namen als Abkürzungen für Typ-Ausdrücke auftreten, ist die Sache relativ einfach. Die offensichtliche Definition der Typ-Äquivalenz wäre in diesem Fall:

*Zwei Typ-Ausdrücke sind äquivalent, wenn sie identisch sind.*

Der Typ-Prüfer muß also nur beide Typ-Ausdrücke bzgl. ihres Aufbaus miteinander vergleichen. Dies geschieht am besten rekursiv. (Das funktioniert, falls keine rekursiv definierten Typ-Ausdrücke möglich sind, d.h. der zugeordnete Graph azyklisch ist!)

Beispiel einer Prüfung auf Typ-Äquivalenz:

```
function typequiv(s, t) : boolean
begin
  if s und t sind vom gleichen Basistyp then
    return (true);
  if s = pointer(s1) and t = pointer(t1) then
    return (typequiv(s1, t1));
    .
    .
    .
  if s = array(s1, s2) and t = array(t1, t2) then
    return (s1 = t1) and typequiv(s2, t2));
  else
    return (false);
end
```

Es gibt leider viele Konstruktionen in höheren Programmiersprachen, die implementationsabhängig sind. In Pascal betrachte man etwa

```
var x: integer;
    y: 1..20;
    z: 10..50;
    a: array[1..10] of integer;
    b: array[0..9] of integer;
```

Sollte der Term  $y + z$  erlaubt sein? Und welchen Typ hat das Ergebnis? Gehört der Indexbereich oder die Länge eines Feldes zum Typ oder nicht? Ist im Beispiel eine Wertzuweisung  $a := b$  erlaubt? Auch ein Überschreiten einer Feldgrenze beim Zugriff auf ein Feld kann als Typ-Fehler interpretiert werden, speziell wenn der Indexbereich Teil des Typs des Feldes ist. So ein Fehler ist jedoch nicht in jedem Fall bereits zur Übersetzungszeit lokalisierbar und leider wird bei einigen Programmiersprachen eine Überprüfung zur Laufzeit aus Effizienzgründen nicht durchgeführt.

Läßt man zu, dass Namen für Typ-Ausdrücke benutzt werden können, dann gibt es zusätzliche Probleme.

Man betrachte das folgende Pascal-Programmfragment:

```
type link = ↑cell;
      slk = link;
var next : link;
    last : link;
    s : slk;
    p : ↑cell;
    q,r : ↑cell;
```

Es stellt sich die Frage, welche Variablen vom selben Typ sind? Bemerkenswert ist, dass im ersten veröffentlichten Referenz-Handbuch zu Pascal (Jensen, Wirth (1976)) zwar häufig „Typ-gleichheit“ von Konstrukten der Programmiersprache gefordert wird, der Begriff aber nirgends definiert wird, mit der Folge, dass die obige Frage implementationsabhängig war!

Es gibt in diesem Fall zwei prinzipielle Möglichkeiten der Definition von Typ-Äquivalenz für Typ-Ausdrücke:

**Namensäquivalenz:** Jeder Typ-Name legt einen neuen Typ fest. Daraus folgt, dass zwei Typ-Ausdrücke äquivalent sind, wenn sie identisch sind. (Ada, abgeschwächt in Pascal und Modula 2) Zum Beispiel wären in Ada `type sec is range 0..60;` und `type min is range 0..60;` sinnvollerweise unterschiedlich. Objektorientierte Sprachen mit einem strengen Klassenkonzept (Smalltalk, Eiffel, Java) fallen ebenfalls unter diesen Punkt.

**Strukturäquivalenz:** Jeder Typ-Name ist nur eine Abkürzung für den definierten Typ-Ausdruck. Daraus folgert, dass bis auf rekursive Definitionen zwei Typ-Ausdrücke äquivalent sind, wenn man jeden auftretenden Namen durch den zugeordneten Typ-Ausdruck ersetzt und die resultierenden Typ-Ausdrücke identisch sind (Algol 60, Algol 68, FORTRAN, COBOL). Diese Form der Äquivalenz wird in modernen Sprachen nicht mehr verwendet.

### Beispiel 3.3:

Im obigen Beispiel sind bzgl. der Namensäquivalenz `next` und `last` sowie `p`, `q` und `r` äquivalent, aber z.B. nicht `p` und `last`. Andererseits sind natürlich alle Variablen strukturäquivalent.

In Pascal gab es eine noch verwirrendere Konvention, die besagte, dass man implizite Namen für jede direkte Typ-Deklaration verwendet. Dies führt dann dazu, dass im obigen Beispiel `q` und `r` typäquivalent sind, nicht jedoch `p` und `q`! Andererseits werden aber `last` und `s` als äquivalent betrachtet.

In Ada wären sogar `q` und `r` nicht vom gleichen Typ, da eine derartige Deklaration die gleiche Bedeutung wie zwei getrennte Deklarationen haben soll!

Es gibt aber noch weitere Problempunkte. Man betrachte das folgende Programmfragment:

```
t1 = array[-1..9] of integer;
t2 = array[0..10] of integer;
rec1 = record
    x: boolean;
    y: integer;
end;
rec2 = record
    a: boolean;
    b: integer;
end;
```

Sind `t1` und `t2` strukturell äquivalent? Falls der Indexbereich zum Typ gehört sind sie nicht äquivalent! Und wie sieht es mit der strukturellen Äquivalenz von `rec1` und `rec2` aus? Gehören die Namen der Komponenten des Records zum Typ, dann sind sie nicht äquivalent.

In C wird wie in vielen anderen Programmiersprachen eine Mischform zwischen Namens- und Strukturäquivalenz benutzt, denn es gilt die strukturelle Äquivalenz bis zur Record-Ebene hinunter. Jeder Record- (Aufzählungs-, Union-) Name wird dagegen als eigenständiger Typ interpretiert.

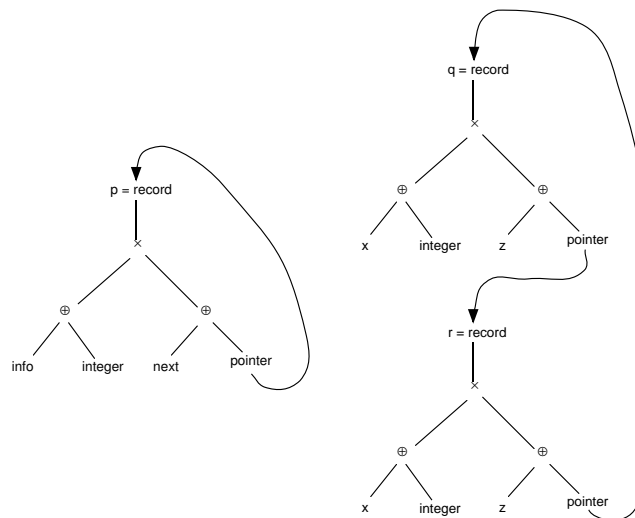
Weiterhin gibt es Probleme bei der Strukturäquivalenz falls rekursive Definitionen erlaubt sind. Man betrachte den Fall:



```

type p=record
    info:integer;
    next:^p;
end;
q=record
    x:integer;
    z:^r;
end;
r=record
    x:integer;
    z:^q;
end;

```



Man sieht an den Beispielen, dass die Namensäquivalenz viel leichter zu handhaben ist und im Gegensatz zur Strukturäquivalenz zu weniger unangenehmen „Überraschungen“ führt.

Bei objektorientierten Programmiersprachen sind kompliziertere Typkonstruktionen nur über die Klassenbildung möglich und die Klassenhierarchie erlaubt eine relativ einfache Überprüfung der Typverträglichkeiten.

### 3.4 Typ-Umwandlungen

Speziell für die Basistypen gibt es in vielen Programmiersprachen eine Reihe von Umwandlungsregeln, um eine automatische Anpassung der Typen von Operanden an zulässige Typen eines Operators zu ermöglichen.

Man betrachte etwa das folgende Beispiel:

$j := x + i$ ; mit  $x$  real und  $i, j$  integer

Da es i.A. keinen Operator für eine derartige Additionsoperation mit unterschiedlichen Operandentypen gibt, muß der Compiler oder aber auch der Programmierer etwas tun:

- Der Compiler führt eine automatische Typ-Anpassung durch Einfügen einer Typ-Umwandlung durch. Dies ist üblicherweise nur dann erlaubt, wenn damit keine Genauigkeitsverluste verbunden sind, z.B. in C bei der Umwandlung `char`  $\rightarrow$  `integer` (Typ-Erweiterung).
- Der Compiler signalisiert eine Fehlersituation, etwa "MIXED MODE" in Fortran. Dann
  - kann der Programmier eine explizite Typ-Umwandlung durch Aufruf einer Bibliotheksfunktion programmieren, etwa `x + float(i)` oder
  - der Programmierer kann eine explizite Typ-Umwandlung durch sogenanntes „casting“ vornehmen. Dies bewirkt eine Änderung des zugehörigen Typ-Ausdrucks, aber nicht notwendig eine Änderung der internen Darstellung des Wertes, etwa in `(char)20`.

Ein abschreckendes Beispiel für eine zu weitgehende automatische Typ-Umwandlung liefert PL/1. Numerische Werte sind entweder vom Typ `BINARY` oder `DECIMAL` und haben weitere Attribute wie etwa `FIXED`, `FLOAT` oder diverse andere Genauigkeitsangaben. Dazu gibt es eine Vielzahl von automatischen Typumwandlungen.

**Beispiel 3.4:**

Es seien die Standardwerte in PL/1 gesetzt.

Dann liefert  $1/3 + 25$  das Ergebnis 5.33333333333333. Laut Regel liefert  $1/3$  das Ergebnis mit maximaler Genauigkeit, d.h. 15-stellig mit 14 Stellen nach dem Komma. Bei einer Addition muß die Genauigkeit erhalten bleiben. Dies führt zwar zu einer OVERFLOW-Exception, die kann aber eventuell ignoriert werden!

Auch in Java gibt es derartige automatische Typumwandlungen, die eventuell zu Überraschungen führen, etwa in

```
byte  b1, b2, b3;
b1 = 1;
b2 = 2;
b3 = b1 + b2;  /* führt zur Fehlermeldung */
```

Das Ergebnis der Addition wird automatisch vom Compiler auf den Typ `integer` gesetzt. Eine Wertzuweisung zu einer Byte-Variablen könnte also zu einem Genauigkeitsverlust führen und ist deshalb nicht erlaubt!

**3.5 Operator-Identifikation (Überladene Funktionen)**

Manchmal haben verschiedene Operatoren eine identische lexikale Darstellung, z.B. kann in Java `+` eine Integer-Addition, eine Float-Addition oder auch eine String-Konkatenation repräsentieren. Man bezeichnet `+` daher als **überladenen Operator**. Aufgabe des Compiler ist es, aus dem Kontext den „richtigen“ Operator zu identifizieren.

Das Problem verschärft sich bei Programmiersprachen, die dem Programmierer mehrere Definitionen einer Funktion mit unterschiedlichen Parameter- oder Ergebnistypen erlaubt.

Beispiel: In Ada sind z.B folgende Definitionen gleichzeitig erlaubt:

```
function '*' (i,j:integer) return complex;
function '*' (x,y:complex) return complex;
```

Damit hat eine Funktion (genauer ein Funktionsname) eine Vielzahl von Typ-Ausdrücken und auch arithmetische Ausdrücke haben nicht notwendigerweise nur einen Typ!

**Beispiel 3.5:**

Nehmen wir an, dass `*` unter anderen die folgenden Typ-Ausdrücke hat:

- $integer \times integer \rightarrow integer$
- $integer \times integer \rightarrow complex$
- $complex \times complex \rightarrow complex$

Haben dann 2, 3 und 5 den einzig möglichen Typ `integer` und sei `z` eine Variable vom Typ `complex`, dann kann  $3 * 5$  den Typ `integer` oder `complex` haben. Im Ausdruck  $(3 * 5) * 2$  **muss**  $3 * 5$  den Typ `integer` haben, im Ausdruck  $(3 * 5) * z$  **muss**  $3 * 5$  den Typ `complex` haben.

Eine Regel in Ada ist, dass der Gesamtausdruck nur genau einen Typ haben darf. Weitere Komplikationen ergeben sich, wenn, wie in  $C^{++}$ , selbstdefinierte und automatische Typ-Konvertierung erlaubt ist.

Beispielhaft soll die Identifikation eines Operators oder einer einer Funktion an folgendem Beispiel einer Übersetzung in Postfix-Notation erläutert werden.

**Vorgehensweise bei der Operatoridentifikation:**

- 1) Bottom-Up die möglichen Typ-Mengen eines Teilausdrucks berechnen (mit einem synthetischen Attribut *types*)
- 2) Top-Down die identifizierten Typen für jeden Teilausdruck festlegen (mit einem inheriten Attribut *utype*)
- 3) Bottom-Up den Postfix-Code erzeugen (mit dem synthetischen Attribut *code*)

$$\begin{aligned}
 E' \rightarrow E & \quad \{ E'.types := E.types; \\
 & \quad E.utype := \text{if } E.types = \{t\} \text{ then } t \text{ else type-error}; \\
 & \quad E'.code := E.code; \} \\
 E \rightarrow \text{id} & \quad \{ E.types := \text{lookup}(\text{id.entry}); \\
 & \quad E.code := \text{concat}(\text{id.lexstring}, ":", E.utype); \} \\
 E \rightarrow E_1 \text{ op } E_2 & \quad \{ E.types := \{t \mid \text{es gibt Typ } r \in E_1.types \text{ und Typ } s \in E_2.types \text{ und} \\
 & \quad \text{Typ } r \times s \rightarrow t \in \text{op.types}; \\
 & \quad \text{Sei } \sigma = \{(r, s) \mid r \times s \rightarrow t \in \text{op.types}, \\
 & \quad r \in E_1.types, s \in E_2.types \text{ und } t = E.utype\}; \\
 & \quad E_1.utype := \text{if } \sigma = \{(r, s)\} \text{ then } r \text{ else type-error}; \\
 & \quad E_2.utype := \text{if } \sigma = \{(r, s)\} \text{ then } s \text{ else type-error}; \\
 & \quad E.code := \text{concat}(E_1.code, E_2.code, \text{op} : r \times s \rightarrow t, r, s, t \text{ wie oben}; \}
 \end{aligned}$$

Viel komplexer ist das Problem bei objekt-orientierten Sprachen mit Klassenhierarchie und statischer Typprüfung. Hier geht es um die Auswahl der korrekten Methode, die zum Teil auch erst zur Laufzeit getroffen werden kann. Folgendes Beispiel<sup>2</sup> zeigt die Problematik

```

class T {
    T n() {return new R(); }
}
class S extends T {
    T n() {return new S();}
}
class R extends S {
    T n() {return new R();}
}
main () {
    T a;
    if (...)
        a = new T();
    else
        a = new S();
    a = a.n();
}

```

Welchen Typ kann der Compiler **a** zuordnen und welche Methode mit Namen **n** wird benutzt?

<sup>2</sup>nach dem Buch von Aho, Lem, Sethi und Ullman

Da Java das dynamische Laden von Klassen erlaubt und viele größere Programme auch noch mit *reflection* arbeiten, ist eine genaue Auswahl der „passenden“ Methode vom Compiler nicht immer zu erreichen und man muss die Auswahl auf die Laufzeit des Programms verschieben (dynamische Methodenauswahl).

### 3.6 Untertypen

Das Bilden von Untertypen erzeugt eine Relation auf den Typen, die es erlaubt, Werte eines Typs anstelle von Werten eines anderen Typs zu benutzen. Betrachtet man in einer üblichen Programmiersprache mit Typ-Prüfung die Anwendung einer Funktion  $f$  auf ein Argument  $x$ , so bestimmt der Typ-Prüfer den Typ  $A \rightarrow B$  der Funktion  $f$  und den Typ  $C$  des Arguments  $x$  und prüft, ob  $C = A$  gilt.

In Programmiersprachen, die die Bildung von Untertypen erlauben, ist die Situation komplexer. Ist  $X$  Untertyp von  $Y$ , geschrieben  $X <: Y$ , dann kann jeder Ausdruck vom Typ  $X$  ohne Hervorrufen eines Typ-Fehlers in jedem Kontext benutzt werden, in dem ein Ausdruck vom Typ  $Y$  benötigt wird.

Für das obige Beispiel bedeutet dieses, dass der Typ-Prüfer nach Bestimmung der Typen  $A \rightarrow B$  für  $f$  und  $C$  für  $x$  prüfen muss, ob  $C <: A$  gilt.

Das Konzept der Untertypen erlaubt eine konsistente Behandlung heterogen zusammengesetzter Daten, die alle Untertyp eines gemeinsamen Typs sind. So kann man etwa eine Liste von unterschiedlichen Arten von Bankkonten bilden, die alle Untertypen eines Typs `bank_konto` sind. Ohne die Möglichkeit, Untertypen bilden zu können, müssten alle Listenelemente vom gleichen Typ sein, mit Untertypen müssen die Listenelemente nur Untertypen eines gemeinsamen Typs sein.

Dieses Konzept ist auch wichtig bei der Erweiterung der Funktionalität eines Programms. Wenn Objekte eines bestimmten Typs  $X$  nicht die gewünschte Funktionalität haben und man möchte sie durch Objekte eines Typs  $Y$  ersetzen, die die gewünschte Funktionalität besitzen, so wird man in vielen Fällen ohne größere Änderungen des gesamten Programms die neuen Objekte mit  $Y$  als Untertyp von  $X$  einführen können. Dies ist besonders hilfreich, wenn man sich über eine Reihe von Prototypen an das gewünschte Programm annähert.

Einer der großen Vorteile objektorientierter Sprachen ist die Möglichkeit der Definition und der Verwendung von Untertypen. Häufig werden Untertypen implizit über Vererbung definiert.

**Vererbung** ist ein Konzept, das es erlaubt, neue Objekte durch Erweiterung bereits existierender Objekte zu definieren. Durch Vererbung werden Attribute und Methoden einer Klasse  $X$  an eine andere Klasse  $Y$  weitergegeben. Die Klasse  $Y$  ist dann eine Unterklasse von  $X$  bzw. die Klasse  $X$  ist Oberklasse der Klasse  $Y$ . In einer Unterklasse können Attribute und Methoden zu den geerbten der Oberklasse hinzugefügt werden. Es können aber auch vererbte Methoden neu definiert und/oder implementiert werden.

Bei einer einfachen Vererbung hat jede Klasse höchstens eine Oberklasse. Dies führt zu einer hierarchischen Anordnung der Klassen. Ist eine mehrfache Vererbung erlaubt, so hat eine Klasse mehrere Oberklassen. In diesem Fall muss man darauf achten, dass in der Vererbungsfolge keine Zyklen und keine Namenskonflikte auftreten.

Vom Prinzip her könnte man Vererbung durch Duplizieren von Code realisieren. Durch das Vererben spart man sich aber das Kopieren. Man muss natürlich dabei beachten, dass eine Änderung in der Oberklasse auch eine Änderung in den Objekten der Unterklasse bewirkt (sofern die geänderten Teile in der Unterklasse nicht überschrieben werden).

Wichtig ist in diesem Zusammenhang der Unterschied zwischen Untertyp-Bildung und Vererbung. Untertypen bilden eine Relation auf den Typen, Vererbung dagegen bildet eine Relation auf den Implementationen. Dass diese Begriffe häufig durcheinander gebracht werden liegt an

den einzelnen Programmiersprachen, die in ihren Klassenkonzepten beides kombinieren. In C++ wird *A* als Untertyp von *B* nur dann akzeptiert, wenn *B* public base class von *A* ist.

Folgendes Beispiel<sup>3</sup> zeigt die Problematik:

#### Beispiel 3.6:

Es soll ein Programm geschrieben werden, das mit den Datenstrukturen **stack**, **queue** und **dequeue** arbeitet.

**stack:** Eine Datenstruktur mit Einsetz- und Löschoperation, so dass das zuerst eingesetzte Objekt als letztes entfernt wird (first-in, last-out).

**queue:** Eine Datenstruktur mit Einsetz- und Löschoperation, so dass das zuerst eingesetzte Objekt als erstes entfernt wird (first-in, first-out).

**dequeue:** Eine Datenstruktur mit zwei Einsetz- und zwei Löschoperationen. Eine dequeue (doubly ended queue) kann man sich als eine Liste vorstellen, bei der sowohl am Anfang als auch am Ende der Liste Objekte eingesetzt oder entfernt werden können. Daher benötigt man für Anfang und Ende jeweils eine Einsetz- und eine Löschoperation.

Die Datenstruktur **dequeue** kann sowohl die Aufgaben der Datenstrukturen **stack** als auch **queue** übernehmen. Also wäre es eine Möglichkeit, zunächst die Klasse **dequeue** zu implementieren und dann die Klassen **stack** und **queue** als Unterklassen von **dequeue** zu definieren, indem man die geerbten Methoden zum Einsetzen und Löschen umbenennt bzw. überschreibt. Für die Klasse **queue** würde man z.B. die Löschoperation für den Anfang und die Einsetzoperation für das Ende der **dequeue** umbenennen und die beiden anderen Operation durch Methoden überschreiben, die etwa eine Fehlermeldung produzieren.

Obwohl **stack** und **queue** Unterklassen von **dequeue** sind, bilden sie *keinen* Untertyp von **dequeue**. Angenommen eine Funktion *f* hat ein Objekt *d* der Klasse **dequeue** als Argument und fügt an jedem Ende ein Objekt ein. Wäre jetzt **stack** oder **queue** ein Untertyp von **dequeue**, so müsste die Funktion *f* auch mit einem Objekt der Klasse **stack** oder **queue** arbeiten. Dies führt aber zu einem Fehler, also sind **stack** und **queue** keine Untertypen von **dequeue**.

Dagegen kann man in jedem Kontext, in dem man etwa ein Objekt der Klasse **stack** bzw. **queue** benutzt, ohne Schwierigkeiten auch ein Objekt der Klasse **dequeue** benutzen. Folglich ist **dequeue** Untertyp sowohl von **stack** als auch von **queue**.

### 3.7 Polymorphe Funktionen

Weitere Probleme treten auf, wenn in der Programmiersprache sogenannte **polymorphe Funktionen** oder **polymorphe Operatoren** zugelassen sind. Dies sind Funktionen, bei denen der Typ der Parameter nicht eindeutig festgelegt werden muß. Dieses Konzept ist wichtig und hilfreich, z.B. möchte man allgemeine Sortierprogramme schreiben können, ohne Rücksicht auf den speziellen Typ der zu sortierenden Elemente. Ein anderes Beispiel ist eine Funktion zum Bestimmen der Länge einer Liste, etwa in Scheme:

```
(define (length liste)
  (if (null? liste)
      0
      (+ 1 (length (cdr liste)))))
```

---

<sup>3</sup>lt. J.C. Mitchell ist es von Alan Snyder

Ähnliche Konstruktionen findet man in anderen funktionalen Sprachen wie etwa ML oder Haskell oder in rein objektorientierten Sprachen wie Smalltalk, aber z.B. nicht in Pascal. Welchen Typ-Ausdruck kann man der Funktion `length` zuordnen?

Man benötigt also wieder ein Konzept ähnlich dem der Untertypen im vorigen Abschnitt, um eine Typ-Prüfung vornehmen zu können. Allerdings kann die Definition von Untertypen in diesen Sprachen aber nicht an eine Klassenhierarchie gebunden sein.

Um auch Typ-Ausdrücke für polymorphe Funktionen angeben zu können, muß man das Konzept einer **Typ-Variablen** einführen. Eine Typ-Variable steht dabei für einen beliebigen Typ. Für das obige Beispiel könnte man der Funktion `length` den Typ-Ausdruck  $\forall \alpha. list(\alpha) \rightarrow integer$  zuordnen, wobei  $\alpha$  die Typ-Variable ist.

Wir wollen im folgenden den Quantor weglassen und schreiben einfach nur  $list(\alpha) \rightarrow integer$

Die Typ-Prüfung von Programmen, in denen polymorphe Funktionen auftreten können, ist signifikant schwieriger als die Typ-Prüfung in den vorangegangenen Fällen.

- Verschiedene Auftreten einer polymorphen Funktion in einem Ausdruck können durchaus Argumente unterschiedlichen Typs haben.
- Da in Typ-Ausdrücken Variablen auftreten können, ist die Typ-Äquivalenz neu zu definieren. Um zwei Typ-Ausdrücke mit Variablen „anzupassen“, müssen die auftretenden Variablen durch Typ-Ausdrücke (eventuell wieder mit neuen Variablen) ersetzt werden. Diese Problematik ist unter dem Namen **Unifikation** bekannt und wird auch in der logischen Programmierung, etwa in PROLOG, benutzt.

**Beispiel:** Die Typ-Ausdrücke  $pointer(\alpha)$  und  $pointer(pointer(\beta))$  könnte man z.B. durch  $\alpha \mapsto pointer(\gamma)$  und  $\beta \mapsto \gamma$  unifizieren.

Es gibt nun mehrere Möglichkeiten, wie man die Typ-Prüfung in Programmiersprachen, die polymorphe Funktionen und Operatoren erlauben, durchführen kann.

- 1) Man verzichtet völlig auf die Typisierung von Variablen usw. in der Programmiersprache und auf eine Überprüfung zur Übersetzungszeit und benutzt eine dynamische Typ-Überprüfung zur Laufzeit (etwa in LISP, Scheme oder Smalltalk)
- 2) Man verzichtet auf die Typisierung von Variablen usw. in der der Programmiersprache, prüft aber in der Übersetzungsphase jeden Gebrauch eines Namens oder eines anderen Sprachkonstrukts auf einen konsistenten Gebrauch (etwa in ML, Caml oder Haskell)
- 3) Man erweitert die Syntax der Programmiersprache um Typ-Variable oder Typ-Parameter, damit man generische Typen definieren kann. Oder man „durchlöchert“ das übliche Typ-System durch die Verwendung von Zeigern oder durch beliebiges „casting“.
- 4) Man benutzt eine Kombination der obigen Möglichkeiten.

### Beispiel 3.7:

in einer Programmiersprache mit Typ-Variablen wären etwa die folgenden Deklarationen möglich:

```
deref : pointer( $\alpha$ )  $\rightarrow$   $\alpha$ ;
q : pointer(pointer(integer));
```

Tritt dann im Programm der Ausdruck `deref(deref(q))` auf, so kann durch Typ-Inferenz geschlossen werden, dass dieser Ausdruck den Typ *integer* hat.

Wie kann man nun zwei Typ-Ausdrücke unifizieren? Das Problem besteht darin, für alle in beiden Ausdrücken auftretenden Typ-Variablen Ausdrücke zu finden, so dass nach einer Substitution der gefundenen Ausdrücke die Typ-Ausdrücke identisch sind. So kann z.B. der Typ-Ausdruck  $\alpha \rightarrow \alpha$  zu  $integer \rightarrow integer$  oder aber auch zu  $pointer(\beta) \rightarrow pointer(\beta)$  werden. Man nennt das Ergebnis einer derartigen Substitution auch eine **Instanz** von  $\alpha \rightarrow \alpha$ .

Nun sucht man nicht irgendeine Substitution, die zwei Typ-Ausdrücke identisch macht, sondern nach einer möglichst allgemeinen Substitution mit dieser Eigenschaft, d.h. eine möglichst allgemeine Instanz der zwei Ausdrücke.

Man bezeichnet eine Substitution  $\sigma$ , die zwei Typ-Ausdrücke  $t_1$  und  $t_2$  identisch macht als **allgemeinste Substitution**, falls gilt:

- 1)  $\sigma(t_1) = \sigma(t_2)$ , d.h. die Substitution macht beide Ausdrücke identisch
- 2) für jede andere Substitution  $\sigma'$  mit  $\sigma'(t_1) = \sigma'(t_2)$  ist  $\sigma'(t_1)$  eine Instanz von  $\sigma(t_1)$ .

**Bemerkung:** Wenn man von einer Unifikation zweier Typ-Ausdrücke spricht, so wird darunter meist die Anwendung der allgemeinsten Substitution auf die Typ-Ausdrücke verstanden.

Beispiel: Gegeben seien die beiden Typ-Ausdrücke

$$\begin{aligned} t_1 &= ((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2) \quad \text{und} \\ t_2 &= ((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5 \end{aligned}$$

Die allgemeinste Substitution  $\sigma$  wäre dann

$$\begin{aligned} \alpha_1 &\mapsto \alpha_3 \\ \alpha_2 &\mapsto \alpha_2 \\ \alpha_3 &\mapsto \alpha_3 & \sigma(t_1) = \sigma(t_2) = ((\alpha_3 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2) \\ \alpha_4 &\mapsto \alpha_2 \\ \alpha_5 &\mapsto list(\alpha_2) \end{aligned}$$

Eine speziellere Substitution  $\sigma'$  wäre

$$\begin{aligned} \alpha_1 &\mapsto \alpha_1 \\ \alpha_2 &\mapsto \alpha_1 \\ \alpha_3 &\mapsto \alpha_1 & \sigma'(t_1) = \sigma'(t_2) = ((\alpha_1 \rightarrow \alpha_1) \times list(\alpha_1)) \rightarrow list(\alpha_1) \\ \alpha_4 &\mapsto \alpha_1 \\ \alpha_5 &\mapsto list(\alpha_1) \end{aligned}$$

und natürlich ist  $\sigma'(t_1)$  eine Instanz von  $\sigma(t_1)$ .

Eine interessante Anwendung erfährt dieses Konzept in der Programmiersprache ML, einer funktionalen Sprache, die polymorphe Funktionen erlaubt. Der Programmierer muß aber nicht selbst seine Funktionen deklarieren, sondern ein ausgefeiltes Typ-System in ML berechnet automatisch den Typ eines eingegebenen Ausdrucks.

Definiert man also z.B. eine Funktion, etwa

```
fun ac(oper, init, seq) =
  if null(seq) then init
  else oper(hd(seq), ac(oper, init, tl(seq)));
```

so erhält man als Antwort des Systems:

```
val ac = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
```

wobei der Typ-Ausdruck gemäß unserer Notation wie folgt wäre:

$$(\alpha \times \beta \rightarrow \beta) \times \beta \times \text{list}(\alpha) \rightarrow \beta.$$

Ein anderes Beispiel wäre etwa die ML-Version der `filter`-Prozedur

```
fun filter(pred, seq) =
  if null(seq) then nil
  else if pred(hd (seq)) then
    hd(seq) :: filter(pred, tl(seq))
  else filter(pred, tl(seq));
```

die zu der Ausgabe

```
val filter = fn : ('a -> bool) * 'a list -> 'a list
```

führt, die in unserer Notation

$$(\alpha \rightarrow \text{boolean}) \times \text{list}(\alpha) \rightarrow \text{list}(\alpha)$$

lauten würde.

### Beispiel 3.8:

Typ-Inferenz für die ML-Funktion

```
fun length(lptr) =
  if null(lptr) then 0
  else length(tl(lptr)) + 1;
```

**Bemerkung:** Das Schlüsselwort `fun` leitet eine Funktionsdefinition ein. Definiert wird hier eine Funktion `length` mit einem Parameter `lptr`. Das Prädikat `null` testet, ob das Argument eine leere Liste ist. Die Funktion `tl` angewendet auf eine Liste liefert die Ausgangsliste ohne das erste Listenelement zurück. Die Funktion `length` berechnet also die Länge einer Liste. Ihr ist damit der Typ-Ausdruck  $\text{list}(\alpha) \rightarrow \text{integer}$  zugeordnet.

Wie kann nun das Typ-System auf diesen Typ-Ausdruck schließen?

Betrachtet man zunächst den Ausdruck links vom „=-Zeichen, dann kann man den beiden Symbolen `length` und `lptr` zum Beispiel die Typ-Ausdrücke  $\gamma \rightarrow \delta$  und  $\gamma$  zuordnen, da es sich ja um eine Funktionsdefinition einer Funktion mit einem Parameter handelt. Der Ausdruck rechts vom „=-Zeichen muss also einen Wert ergeben, dem der Typ-Ausdruck  $\delta$  zugeordnet ist. Betrachtet man nun den `null` zugeordneten Typ-Ausdruck  $\text{list}(\alpha_n) \rightarrow \text{boolean}$ , wobei  $\alpha_n$  eine neue Typ-Variable ist, so kann man schließen, dass  $\gamma = \text{list}(\alpha_n)$  sein muss. Aus dem Typ-Ausdruck  $\text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$  für `tl` kann man wiederum schließen, dass  $\alpha_t = \alpha_n$  sein muss. Die Anwendung der `length`-Funktion auf `tl(lptr)` liefert ein Objekt vom Typ  $\delta$ .

Da die Konstante 1 vom Typ `integer` ist und der Addition ein Typ-Ausdruck  $\text{integer} \times \text{integer} \rightarrow \text{integer}$  zugeordnet ist, muss  $\delta = \text{integer}$  gelten. Damit ist aber auch der Typ-Ausdruck  $\text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$  für die `if`-Anweisung mit  $\alpha_i = \text{integer}$  erfüllt und man erhält so den Typ-Ausdruck  $\text{list}(\alpha_n) \rightarrow \text{integer}$  für die Funktion `length`.



Um das ganze etwas formaler darzustellen, werden die einzelnen Schritte jetzt noch tabellarisch aufgeführt:

Die folgende Funktionen und ihre Typ-Ausdrücke sind vorgegeben:

$$\begin{aligned}
 \text{if} & : \text{boolean} \times \alpha \times \alpha \rightarrow \alpha \\
 \text{null} & : \text{list}(\alpha) \rightarrow \text{boolean} \\
 \text{tl} & : \text{list}(\alpha) \rightarrow \text{list}(\alpha) \\
 0 & : \text{integer} \\
 1 & : \text{integer} \\
 + & : \text{integer} \times \text{integer} \rightarrow \text{integer}
 \end{aligned}$$

Hier nun die Tabelle der einzelnen Schlussfolgerungen:

ZEILE	AUSDRUCK : TYP-AUSDRUCK	SUBSTITUTION
(1)	lptr : $\gamma$	
(2)	length : $\beta$	
(3)	length(lptr) : $\delta$	$\beta = \gamma \rightarrow \delta$
(4)	lptr : $\gamma$	
(5)	null : $\text{list}(\alpha_n) \rightarrow \text{boolean}$	
(6)	null(lptr) : $\text{boolean}$	$\gamma = \text{list}(\alpha_n)$
(7)	0 : $\text{integer}$	
(8)	lptr : $\text{list}(\alpha_n)$	
(9)	tl : $\text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$	
(10)	tl(lptr) : $\text{list}(\alpha_n)$	$\alpha_t = \alpha_n$
(11)	length : $\text{list}(\alpha_n) \rightarrow \delta$	
(12)	length(tl(lptr)) : $\delta$	
(13)	1 : $\text{integer}$	
(14)	+ : $\text{integer} \times \text{integer} \rightarrow \text{integer}$	
(15)	length(tl(lptr)) + 1 : $\text{integer}$	$\delta = \text{integer}$
(16)	if : $\text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
(17)	if( .... ) : $\text{integer}$	$\alpha_i = \text{integer}$

## 4 Das Laufzeitsystem

Bevor die Übersetzung von der höheren Programmiersprache in einen Zwischencode behandelt werden kann, muss etwas genauer auf Fragen des Gültigkeitsbereichs von Deklarationen und der Speicherzuordnung von Variablen eingegangen werden. Die Realisierung dieser Konzepte geschieht im Laufzeitsystem, das zusammen mit den System-Schnittstellen, dem Speicher-Management bzw. dem darüber liegenden Software-Schichten den Zielrechner darstellt. Außerdem werden in diesem Abschnitt verschiedene Methoden der Parameterübergabe diskutiert.

### 4.1 Gültigkeitsbereich und Lebensdauer

Deklarationen sind Sprachkonstrukte in höheren Programmiersprachen, bei denen einem Namen ein Typ zugeordnet wird. Dies geschieht in fast allen modernen Programmiersprachen explizit, d.h. der Programmierer schreibt etwa `var abc : integer;`. Diese Zuordnung hat nun einen gewissen Gültigkeitsbereich innerhalb des Programms und das zugeordnete Objekt hat eine eventuell sogar davon abweichende Lebensdauer.

Der **Gültigkeitsbereich** (Scope) dieser Deklaration von `abc` umfasst den Teil des Programms, in dem sich eine Verwendung des Namens `abc` auf diese Deklaration bezieht.

Üblicherweise wird der Gültigkeitsbereich statisch und lexikographisch, d.h. unabhängig von einem Programmlauf, allein durch den Programmtext, festgelegt.

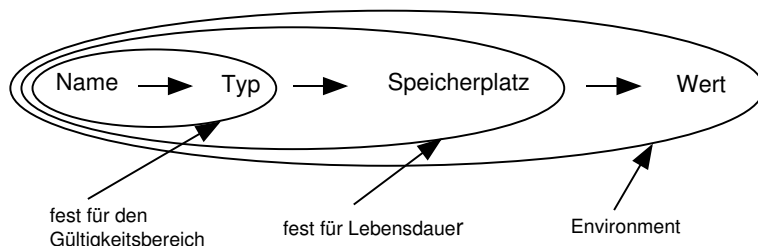
**Bem.:** Man könnte den Gültigkeitsbereich von `abc` auch dynamisch festlegen, d.h. die zeitlich zuletzt erfolgte Deklaration von `abc` hätte an dieser Stelle Gültigkeit! Diese Methode wurde in einigen frühen Lisp-Versionen benutzt.

**Bem.:** In anderen Programmiersprachen wie etwa ML, Smalltalk oder Python kann auf eine Deklaration der Variablen verzichtet werden, da entweder ein komplexes Typ-Inferenzsystem den zugehörigen Typ einer Variablen berechnet oder aber die an die Variablen gebundenen Objekte selbst Typ-Informationen enthalten.

FORTRAN ist ein Beispiel einer Programmiersprache, die eine implizite Deklaration von Variablen erlaubt. Jede Variable, die mit einem der Buchstaben I, J, K, L, M oder N beginnt, ist vom Typ `integer`, alle anderen vom Typ `real`.

Die **Lebensdauer** (Extend) des durch `abc` bezeichneten Objekts beginnt mit der Speicherzuordnung für das Objekt und endet mit der Freigabe des Speichers.

Meist geschieht die Speicherzuordnung beim Eintritt in eine Prozedur oder einen Block und beim Verlassen wird der Speicher wieder freigegeben. Es gibt aber auch viele andere Möglichkeiten, etwa die Speicherzuordnung durch expliziten Aufruf von Konstruktoren und die Freigabe des Speichers durch den Aufruf von Destrukturen oder aber auch durch automatische Garbage-Collector Verfahren.



## 4.2 Aktivierungs-Record, statische und dynamische Verkettung

Im folgenden Beispiel sollen Gültigkeitsbereich und Lebensdauer von Variablen betrachtet und die notwendigen Implikationen auf das Laufzeitsystem des Programms etwas genauer untersucht werden.

Wir wollen eine Programmiersprache betrachten, die Gültigkeitsbereiche von Variablen über eine Blockstruktur regelt. Ein Block ist ein Programmteil, der Deklarationen von Variablen enthält, deren Gültigkeitsbereich und Lebensdauer auf diesen Block beschränkt ist. Probleme treten dann auf, wenn Blöcke ineinander geschachtelt werden können und besonders dann, wenn innerhalb von Blöcken Prozeduren definiert werden können.

### Beispiel 4.1:

Beispiel für Schachtelung von Blöcken und Prozeduren

```

A:  begin
      real x,y;
      procedure P();
      begin
          real z;
          x:=x+z;
      end P;

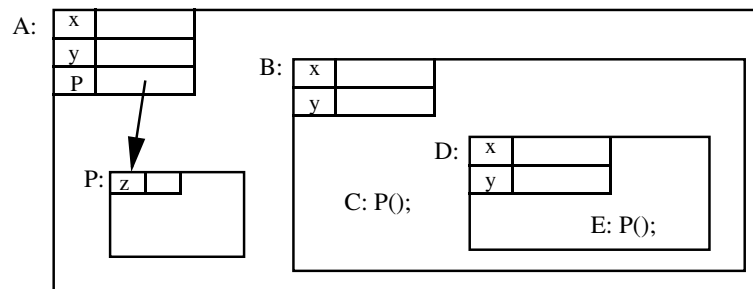
B:      begin
          boolean x,y;
          ...
C:      P();
          ...
D:      begin
          integer x,y;
          ...
E:      P();
          ...
      end D;
  end B;
end A;

```

Um die Schachtelung der einzelnen Blöcke bzw. Prozedurrümpfe formal zu beschreiben, verwendet man die sogenannte Stufenzahl. Alle Deklarationen im äußersten, umfassenden Block sind auf Stufe 1. Bei jedem Blockeintritt erhöht sich die Stufenzahl um 1, bei jedem Blockaustritt vermindert sie sich um 1. Bei Prozeduren gilt eine spezielle Regel - bei einer Prozedur auf Stufe  $i$  sind die Deklarationen im Rumpf sowie die Deklarationen der Formalparameter auf Stufe  $i$ , der Name der Prozedur ist jedoch noch auf Stufe  $i - 1$  deklariert.

Im Beispiel 4.1 sind offensichtlich die Deklarationen von  $x$  und  $y$  als **real** auf Stufe 1, die Deklarationen der beiden Variablen als **integer** dagegen auf Stufe 3. Die Prozedurname  $P$  ist auf Stufe 1, die lokale Variable  $z$  dagegen auf Stufe 2 deklariert.

Manchmal ist es nützlich, die Verschachtelungen in Form eines **Schachtelungsdiagramms** graphisch darzustellen.



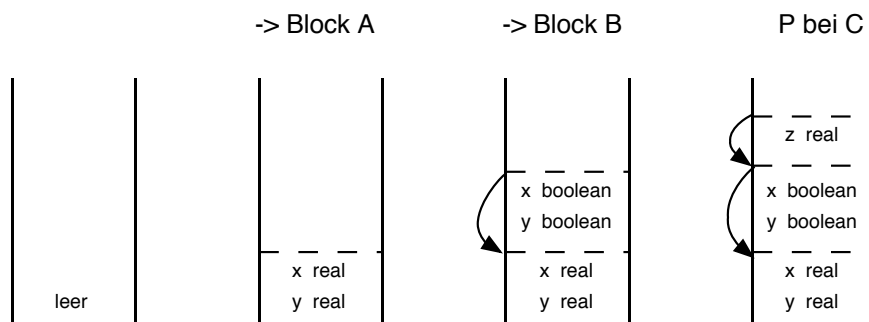
Die großen Rechtecke stellen dabei jeweils einen Block oder einen Prozedurrumpf dar. In der linken oberen Ecke eines derartigen Rechtecks sind die in diesem Block deklarierten Namen mit symbolischen Platzhaltern für die jeweiligen Werte vermerkt. Die Stufenzahl wird dann durch die Tiefe der Verschachtelung der Rechtecke repräsentiert.

Jeder Blockrand ist halb durchlässig - man kann innerhalb eines Blocks nach außen sehen, jedoch nicht von außen in einen Block hinein. Wird z. B. innerhalb des Blocks D bei E der Name P benutzt, so ist damit die Prozedur auf Stufe 2 gemeint; wird der Name z benutzt, so ist an dieser Stelle keine Deklaration von z gültig! Die Benutzung der Namen x und y bezieht sich auf die in D deklarierten Variablen. Die in B und A deklarierten Variablen x und y sind von dieser Stelle aus nicht sichtbar (werden durch die Deklaration in D überdeckt).

Der Speicher für die deklarierten Variablen eines Blocks wird in einem **Aktivierungs-Record** (stack-frame) bereitgestellt.

Dieser enthält alle für **eine** Aktivierung einer ausführbaren Programmeinheit notwendigen Daten. Die Aktivierungs-Records werden üblicherweise auf einem Stack, dem sogenannten **Laufzeitstack** (run-time stack) abgelegt.

Man betrachte das Beispiel 4.1. Der Laufzeitstack würde sich dann wie folgt entwickeln:



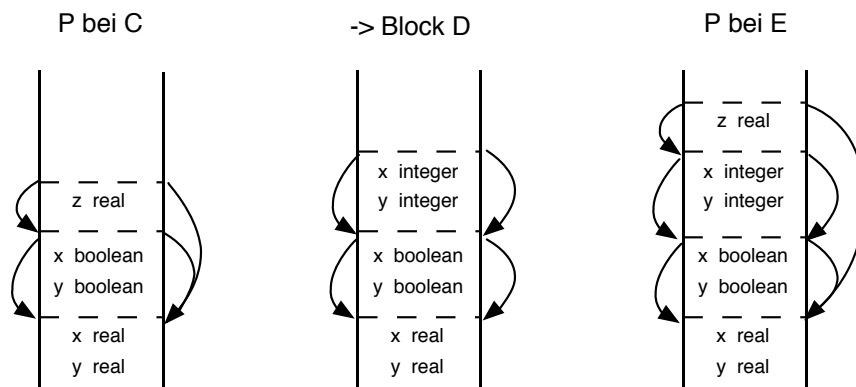
wobei jeder einzelne Aktivierungs-Record einen zusätzlichen Zeiger hat (**dynamischer Link**, durch Zeiger auf der linken Seite des Stacks angedeutet), der auf den Anfang des darunter liegenden Records auf dem Stack verweist. Damit kann nach Verlassen eines Blocks der zugehörige Aktivierungs-Record wieder gelöscht werden.

Betrachtet man die Situation bei Abarbeitung des Prozedurrumpfes von P (die Aktivierungs-records sind dabei wie im Bild rechts), so hat man bei der Abarbeitung des Befehls  $x := x + z$  in P zwei Möglichkeiten der Interpretation:

- 1) Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record zu finden, in dem ein x auftritt. Dies entspricht einem dynamischen Gültigkeitsbereich für die Deklarationen. Man folgt also der dynamischen Verkettung der Aktivierungs-Records (dem dynamischen Link) und würde die Deklaration von x bei B: als erste finden.

- 2) Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record eines P umfassenden Blocks zu finden, der eine Deklaration von  $x$  enthält. In diesem Fall würde also die Deklaration von  $x$  bei A: gemeint sein. Dies entspricht einem lexikalen Gültigkeitsbereich für die Deklarationen. Um diesen Record zu finden benötigt man eine weitere Verkettung der Aktivierungs-Records (**statischer Link**, durch Zeiger auf der rechten Seite des Stacks angedeutet)

Die weitere Entwicklung des Laufzeit-Stacks mit Angabe der statischen Verkettung der Aktivierungs-Records wäre dann wie folgt:



Befindet man sich zur Laufzeit des Programms in einem Block oder einer Prozedur, die auf Stufe  $i$  definiert wurde, so hat die Kette der statischen Links die Länge  $i - 1$ . Eine nicht lokale Größe, die  $j$  Stufen zurück definiert wurde, ist in einem Aktivierungs-Record zu finden, der  $j$  Schritte in der Kette der statischen Links zurück liegt.

Also werden Variablen 2-dimensional adressiert:

- die *Stufenzahl* der Deklaration wird benötigt, um den richtigen Aktivierungs-Record zu finden, und
- ein vom Compiler festgelegter Abstand vom Anfang des Aktivierungs-Records (*Offset*) liefert die endgültige Adresse.

*Dies bedeutet aber auch, dass die Variablenamen **nicht** in dem Aktivierungs-Records abgespeichert werden müssen!*

#### Beispiel 4.2:

Angenommen im Programm aus Beispiel 4.1 werde im Block A eine neue Variable  $i$  deklariert, die im Block D benutzt wird. Die Deklaration von  $i$  ist auf Stufe 1, benutzt wird  $i$  im Block D, der sich auf Stufe 3 befindet. Die Stufendifferenz ist 2, also muss man, um auf  $i$  zuzugreifen, zweimal dem statischen Link folgen um zum Aktivierungs-Record zu kommen, in dem  $i$  abgelegt ist. *Diese Differenz ist aber zur Übersetzungszeit bekannt.*

Um die Zugriffsgeschwindigkeit zu erhöhen, verwendet man häufig ein Feld von Zeigern (**Display** genannt) auf die jeweils aktuellen Aktivierungs-Records.

Die Länge des Feldes entspricht genau der Stufenzahl des aktiven Blocks. Eine Variable  $a$ , die auf Stufe  $i$  deklariert wurde, ist in dem Aktivierungs-Record zu finden, auf den der  $i$ -te Eintrag im Display zeigt.

Erlaubt die Programmiersprache nur Prozedurdefinitionen auf Stufe 1, wie etwa in C, kann man auf den statischen Link völlig verzichten, denn es gibt nur Variablen auf Stufe 1 (globale Variablen) oder Variablen auf Stufe 2 (lokale Variablen).

### 4.3 Prozeduren als Parameter oder als Rückgabewerte

Wenn die Programmiersprache Prozeduren als Parameter zulässt, ergeben sich weitere Probleme beim Laufzeitsystem.

#### Beispiel 4.3:

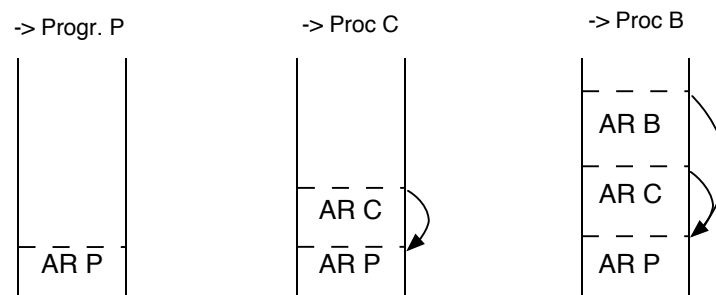
Man betrachte das folgende Programmfragment:

```

program P
begin
  procedure B (function h(n:integer):integer);
  begin
    writeln(h(2))
  end;
  procedure C
  var m:integer;
  function f(n:integer):integer);
  begin
    f:=m+n;
  end;
  begin
    m:=0;
    B(f);
  end;
begin
  C;
end;

```

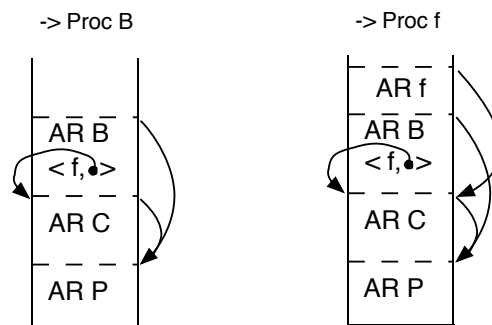
Nach dem Aufruf von C und danach von B ergibt sich folgendes Bild:



In B wird jetzt die als Parameter übergebene Funktion **f** aufgerufen. Allerdings tritt im Gültigkeitsbereich von B die Variable **m** *nicht* auf. Es ist also wichtig, dass die Funktion **f** ihr Environment mitbringt, d.h. als Parameter für den Aufruf der Prozedur B wird sowohl die Startadresse der Prozedur als auch ein Zeiger auf das Environment des Aufrufs, also einen Zeiger auf den Aktivierungs-Record von C übergeben.

Man nennt ein derartiges Paar auch **Closure**.

Damit ergibt sich folgendes Bild des Laufzeitstacks:



Ein ähnliches Problem ergibt sich, wenn die Programmiersprache nicht-lokale Sprünge, also Sprünge mit einem Sprungziel in einem umfassenden Block, zulässt.

Diese Sprünge müssen das Environment ändern. Man benötigt dazu die Stufendifferenz zwischen der Definition der Marke, also des Sprungziels, und dem entsprechenden `goto`-Befehl.

**Hinweis:** Sieht die Programmiersprache vor, dass Prozeduren auch als Rückgabewerte von Prozeduren auftreten, gibt es zusätzliche Probleme. Liefert z.B. eine Prozedur F eine Prozedur P zurück, die lokal in F ist, dann verschwindet nach dem Verlassen von F der Aktivierungs-Record von F und damit ein Teil des Environments von P. Damit dürfte klar sein, dass bei derartigen Sprachen, so etwa bei Scheme, keine einfache Stack-Implementation des Laufzeitsystems möglich ist.

## 4.4 Speicherorganisation

Wie wird man nun den virtuellen oder auch realen Adressraum eines Prozesses aufteilen? Üblicherweise ordnet das Betriebssystem einem Prozess zumindest die folgenden drei Speichersegmente zu, wobei jedes Speichersegment einen zusammenhängenden, sequentiellen Adressraum darstellt:

- Das **Code-Segment**, das das ausführbare Programm enthält und üblicherweise schreibgeschützt ist.
- Das **Stack-Segment**, das den Laufzeit-Stack für die Aktivierungs-Records enthält.
- Das **Daten-Segment**, das den statischen Speicher und den dynamischen Speicher (auch **Heap** genannt) enthält.

Der Laufzeit-Stack enthält den Speicher für Datenobjekte, deren Lebensdauer an die Aktivierung von Prozeduren bzw. Blöcken gebunden ist. Für jede Aktivierung wird im allgemeinen ein Speicherplatz einer festen Größe zugeordnet (Teil des Aktivierungs-Records). Die Adressierung der Objekte erfolgt meist indirekt über eine für alle Aktivierungs-Records gleiche Startposition, die z.B. über einen sogenannten **Frame-Pointer** festgelegt wird, und einen Offset, der bereits zur Übersetzungszeit bekannt ist.

Der statische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer sich vom Start des Programms bis zu dessen Ende erstreckt. Diese Speicherzuordnung kann bereits während der Übersetzungsphase erfolgen und die Adressierung kann damit direkt erfolgen.

Der dynamische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer nicht durch die Ablaufstruktur der Programmiersprache vorhersagbar ist. Typischerweise sind dies

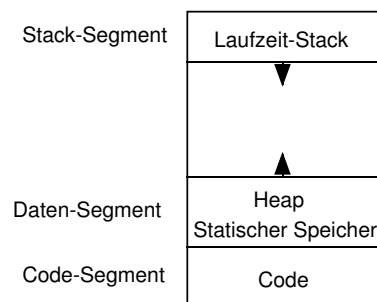
Datenobjekte, die vom Programmierer selbst erzeugt werden können und deren Speicher, je nach Programmiersprache, auch explizit wieder freigegeben werden muss. In C gibt es dazu die Standardaufrufe `malloc` und `free`. Auch Konstruktoren in objektorientierten Programmiersprachen erzeugen Objekte, die in diesem Speicherbereich abgelegt werden.

In manchen dieser Sprachen, wie etwa Smalltalk oder auch Java, muss sich der Programmierer nicht mehr um die Rückgabe des frei gewordenen Speichers kümmern, diese Aufgabe erledigt ein sogenannter **Garbage Collector** im Hintergrund.

In anderen Sprachen wie etwa auch in Scheme werden die benötigten Datenobjekte sowohl automatisch erzeugt als auch wieder zurückgegeben. Auch hier arbeitet ein Garbage Collector, um den nicht mehr benötigten Speicherplatz dem freien Speicher zuzuordnen.

Die in der Verwaltung des Heaps verwendeten Verfahren sind meist recht aufwendig und werden später behandelt.

Die folgende Abbildung zeigt eine typische Verteilung der drei Segmente in dem hier angenommenen linearen und zusammenhängenden Adressraum eines Prozesses.



Das Stack-Segment wächst dabei von oben, d.h. zu den kleineren Adressen hin, während der Heap bei Bedarf nach oben, also zu den größeren Adressen hin wächst.

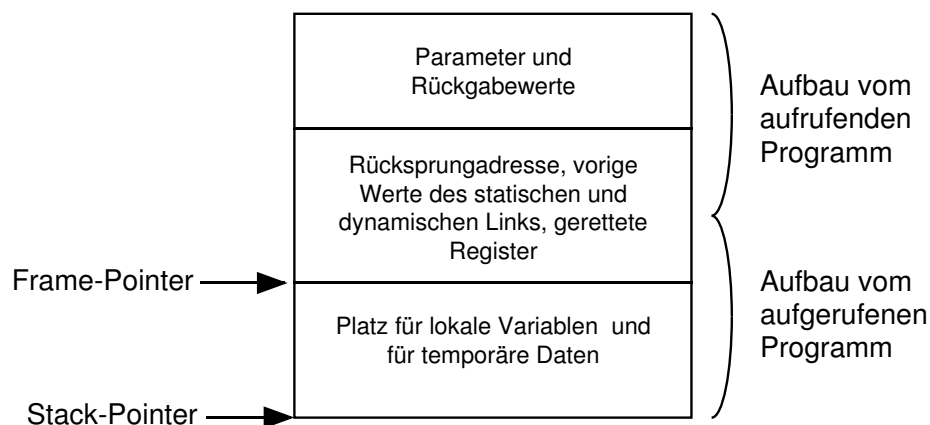
Wichtig zu bemerken ist, dass zugeordnete Datenobjekte eventuell im Daten-Segment verschoben werden müssen, um wieder größere, zusammenhängende Speicherbereiche zu erhalten.

Das bedeutet aber, dass z.B. in lokalen Variablen keine direkten Pointer oder Referenzen auf diese Datenobjekte gespeichert werden dürfen, da diese sonst nach einer Verschiebung auf unsinnige Speicherbereiche zeigen würden.

Eine Lösung ist die Verwendung sogenannter **Handles**, die einen Zeiger auf einen Zeiger auf das Datenobjekt darstellen und deren zweiter Zeiger bei einer Verschiebung vom System aktualisiert werden kann.

Abschließend soll noch beispielhaft der Aufbau eines Aktivierungs-Records gezeigt werden. Beim Aufruf einer Prozedur werden zunächst die Aktualparameter auf den Stack geschrieben und Platz für eventuelle Rückgabewerte geschaffen. Danach werden die momentanen Werte des dynamischen und statischen Links gerettet. Beim Sprung zur Prozedur wird dann die Rücksprungadresse ebenfalls auf den Stack geschrieben. In der angesprochenen Prozedur werden zunächst die Registerinhalte gerettet und dann, durch Setzen des Stack- und Frame-Pointers, der Aufbau des Aktivierungs-Records beendet.





Möglicher Aufbau eines Aktivierungs-Records

## 4.5 Parameterübergabe

Es gibt mehrere Methoden der Parameterübergabe, da es meist mehrere Möglichkeiten gibt, die Bedeutung eines Aktualparameters zu interpretieren.

Zum Beispiel kann ein Aktualparameter  $a[i]$  interpretiert werden als

- 1) der Wert des  $i$ -ten Elementes eines Feldes  $a$  (**r-Wert**), oder
- 2) der Ort, an dem das  $i$ -te Element eines Feldes  $a$  abgespeichert ist (**l-Wert**), oder aber auch
- 3) eine Zeichenkette, die immer dann einzusetzen ist, wenn der zugeordnete Formalparameter in der Prozedur auftritt (Makro-Substitution).

**Bemerkung:** Die Begriffe r-Wert und l-Wert beziehen sich auf eine übliche Anweisung etwa der Form  $x := y + z$  in einer Programmiersprache. Um diese Anweisung abzarbeiten benötigt man die Werte von  $y$  und  $z$  und die Adresse von  $x$ . Die unterschiedliche Behandlung der auftretenden Namen  $x$ ,  $y$  und  $z$  hängt also davon ab, ob sie links (l-Wert) oder rechts (r-Wert) vom Wertzuweisungszeichen  $:=$  auftreten. Natürlich gibt es einige Ausdrücke in Programmiersprachen, die etwa nur einen r-Wert haben, etwa eine Zahl oder ein Ausdruck wie  $y + z$ .

Es gibt die folgenden Möglichkeiten der Parameterübergabe:

**call-by-value** Formalparameter sind wie initialisierte lokale Variablen zu betrachten - die Aktualparameter werden im Environment des Aufrufs ausgewertet und die r-Werte werden zur Initialisierung der Formalparameter benutzt.

Dies ist die Standardmethode in C oder auch in Java.

Probleme gibt es nur bei „großen“ Parametern, etwa Feldern, die ja in die entsprechende lokale Version kopiert werden müssten.

**call-by-reference** Hat der Aktualparameter einen l-Wert, so wird dieser übergeben. Ansonsten wird der Aktualparameter ausgewertet und der Wert in einem temporären Speicherplatz, dessen Adresse übergeben wird, abgelegt.

Dies war die Standardmethode in Fortran.

**call-by-copy-restore** Die Aktualparameter werden ausgewertet und die r-Werte werden übergeben. Beim Verlassen der Prozedur werden die r-Werte der Formalparameter in die l-Werte der Aktualparameter (soweit vorhanden) zurückgespeichert.

Sie ist die Standardmethode in Ada und auch in einigen Fortran-Versionen.

**call-by-name** Jedes Auftreten eines Formalparameters wird im entsprechenden Aufruf der Prozedur durch den Aktualparameter textuell substituiert. Das entspricht bei einer Variablen als Aktualparameter einem call-by-reference. Bei komplizierteren Ausdrücken als Aktualparameter wird dieser in eine parameterlose Prozedur (**thunk**) umgewandelt und jedesmal aufgerufen, wenn der Formalparameter benötigt wird. Dies war die Standardmethode in Algol 60 und ist wegen vieler überraschender Seiteneffekte nicht weiter verwendet worden.

#### Beispiel 4.4:

Man betrachte das folgende, in einer Pseudo-Sprache geschriebene Programm:

```

procedure sub (var a, b, c, d: integer);
begin
    b:=a+a;
    d:=a+c;
end;

program parameter-test;
begin
    var x,y,z:integer;
    x:=1;
    y:=2;
    z:=7;
    sub(x,x,x+y,z);
    print z;
end

```

Der vom Programm ausgegebene Wert von **z** ist:

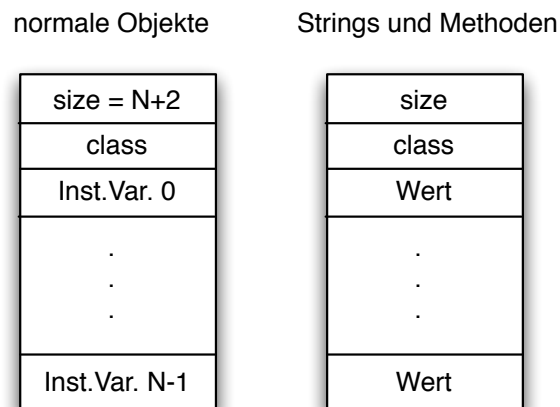
Art der Parameterübergabe	Wert
call-by-value	7
call-by-reference	5
call-by-copy-restore	4
call-by-name	6

## 4.6 Speicherverwaltung in objektorientierten Systemen

Für objektorientierte Programmiersprachen tritt der Laufzeit-Stack mit seinen Aktivierungs-Records gegenüber der Heapverwaltung in den Hintergrund, da die Lebensdauer von Objekten nicht an das Eintreten bzw. Verlassen eines Blocks gekoppelt ist.

Als Beispiel soll hier die Laufzeitumgebung für Smalltalk-80 beschrieben werden [31], die charakteristisch für ein auf einer virtuellen Maschine ablaufendes objektorientiertes System ist.

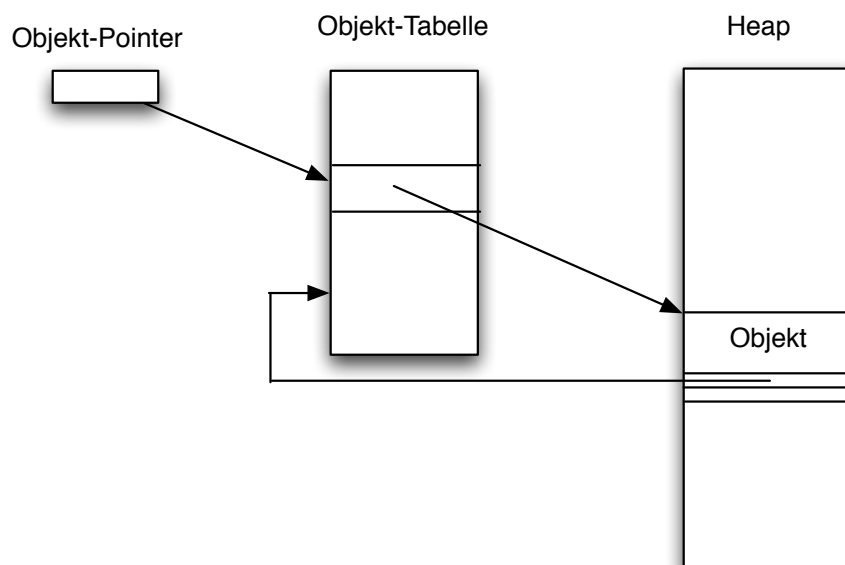
Fast alle im Smalltalk-System existierenden Objekte sind im Heap gespeichert. Der Standardaufbau eines Objekts im Heap ist der folgende:



Abweichend davon werden Zeichenketten (Strings) und Methoden wie rechts dargestellt. Die Werte sind dabei entweder die Zeichen, aus denen der String zusammengesetzt ist oder aber Literale und der Bytecodes der übersetzten Methode.

Integerzahlen werden ebenfalls anders dargestellt.

Die im Heap befindlichen Objekte werden nicht direkt, sondern über eine Objekt-Tabelle indirekt adressiert. Der Grund für diese Vorgehensweise ist einfach einzusehen. Bei einer eventuell notwendigen Kompaktifizierung des freien Speichers im Heap müssen lebendige Objekte verschoben werden. Damit müssten aber auch in allen anderen Objekten, die dieses Objekt referenzieren, die entsprechende Adresse geändert werden. Dies ist natürlich viel zu aufwendig. Für jedes im Heap befindliche Objekt gibt es einen Eintrag in der Objekt-Tabelle und nur dort ist, neben anderen Informationen, die Adresse verzeichnet, ab der sich das Objekt im Speicher befindet. Eine Referenz auf dieses Objekt (ein *Objekt-Pointer*), etwa in einer Variablen oder einer Instanzenvariablen, ist nur ein Index in die Objekt-Tabelle. Muss das Objekt im Speicher verschoben werden, so muss danach nur in der Objekt-Tabelle die aktuelle Adresse nachgetragen werden.



Damit ist es aber auch möglich, die Darstellung von Integer-Objekten zu beschreiben. Ist das unterste Bit eines Objekt-Pointers eine 1, so werden die restlichen Bits als vorzeichenbehaftete

Zahl, also als ein Objekt der Klasse Integer, interpretiert. Ist das unterste Bit eines Objekt-Pointers dagegen eine 0, so werden die restlichen Bits als Index in die Objekt-Tabelle betrachtet

vorz. behaftete Zahl	1
Index zur Objekt-Tabelle	0

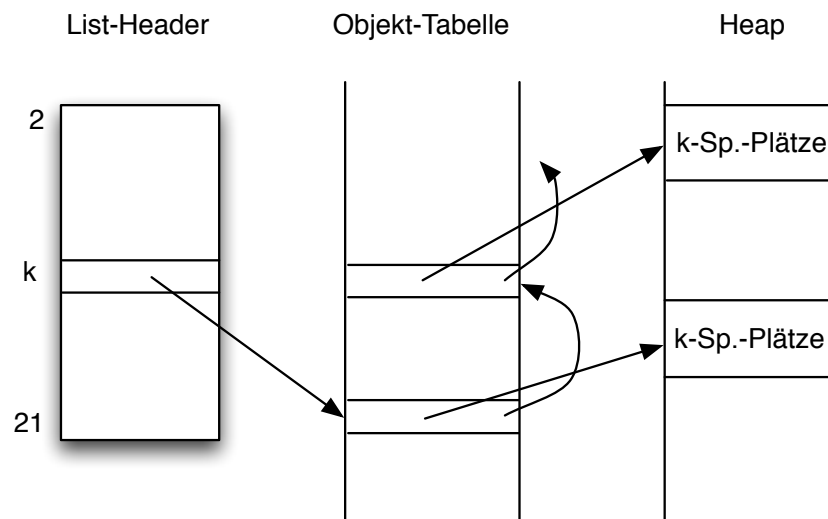
Objekt-Tabellen Einträge bestehen aus zwei Speicherplätzen, die Verwaltungsinformationen und den Ort (gegeben durch Segmentnummer und Adresse im Segment) enthalten, wo sich das Objekt im Heap befindet. Smalltalk-Systeme arbeiten üblicherweise sowohl mit Reference-Counting als auch mit einem „mark-and-sweep“ Verfahren zum Garbage-Collection. Der Reference-Count für jedes Objekt befindet sich ebenfalls in den Verwaltungsinformationen in der Objekt-Tabelle. Freie Einträge in der Objekt-Tabelle werden durch das F-Bit markiert. Diese Einträge sind verkettet und über eine globale Variable erreichbar.

Alle Objekte im System werden auf diese Art gespeichert. Allerdings gibt es zwei Ausnahmen - Zeichenketten und übersetzte Methoden werden aus Effizienzgründen anders gespeichert. Um diese Art Objekte von den anderen unterscheiden zu können, gibt es das P-Bit. Da Objekte dieses Typ im Gegensatz zu allen anderen auch eine ungerade Zahl von Bytes im Speicher belegen können, gibt es weiterhin das O-Bit.

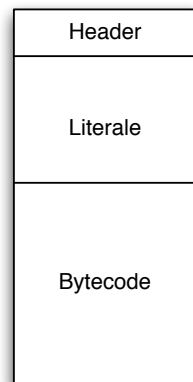
Ref-count	O	P	F	Segment
Adresse				

Um auf Speicheranforderungen bei Objekterzeugung schnell reagieren zu können, sind die freien Einträge in der Objekt-Tabelle wie auch die zugehörigen freien Bereiche im Heap verkettet. Es gibt dabei mehrere Ketten - für kleine Objekte, die insgesamt  $k$  ( $\leq 20$ ) Speicherplätze benötigen, gibt es einen List-Header  $k$ , der auf eine verkettete Liste von Einträgen in der Objekt-Tabelle verweist, die jeweils auf frei Speicherplätze der Größe  $k$  im Heap verweisen.

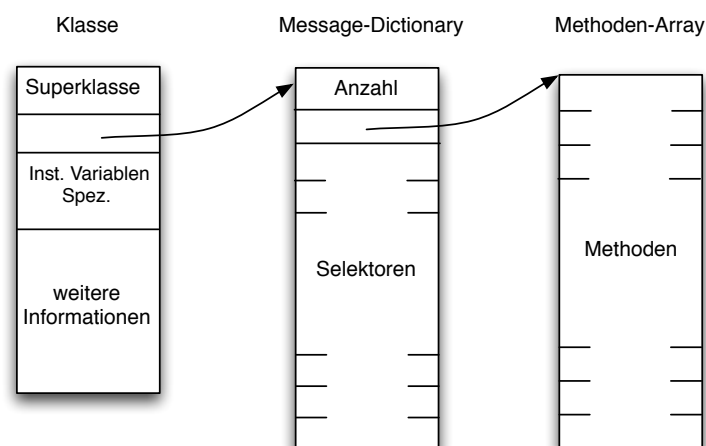
Alle größeren freien Speicherbereiche (im Original  $> 20$  Worte) sind ebenfalls auf diese Weise verkettet.



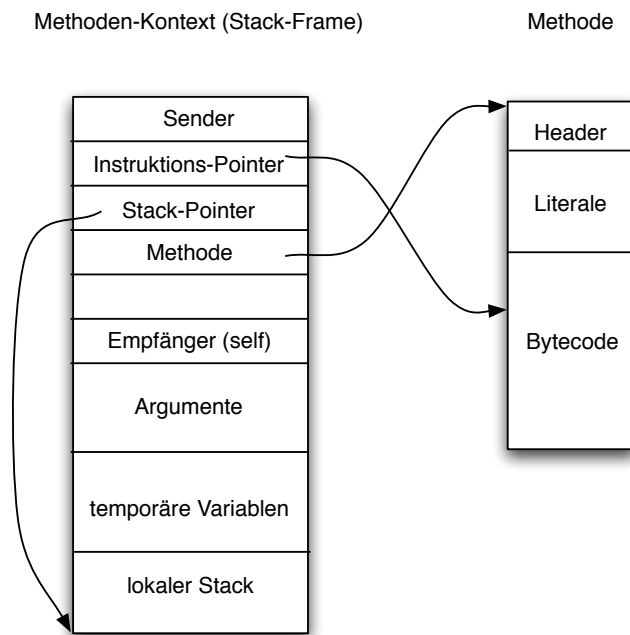
Übersetzte Methoden sind natürlich ebenfalls Objekte und befinden sich daher im Heap. Allerdings werden sie etwas anders gespeichert. Der Header enthält Informationen über die Anzahl der Argumente und die Anzahl der Literale der Methode. Ferner ist die Zahl der benötigten temporären Werte inklusive der Argumente vermerkt.



Da Klassen ebenfalls Objekte sind und die virtuelle Maschine auf die Methoden zugreifen muss, werden die Methoden in einem „Dictionary“ gespeichert. Der Zugriff erfolgt mittels einer Hash-Funktion über den Selektor der Nachricht.



Zur Ausführung von Methoden wird aber nun wieder ein Laufzeit-Stack benötigt. Im Smalltalk-System spricht man statt von Aktivierungs-Records von sogenannten „Kontexten“. In der Abbildung sei wieder angenommen, dass der Stack nach unten wächst.



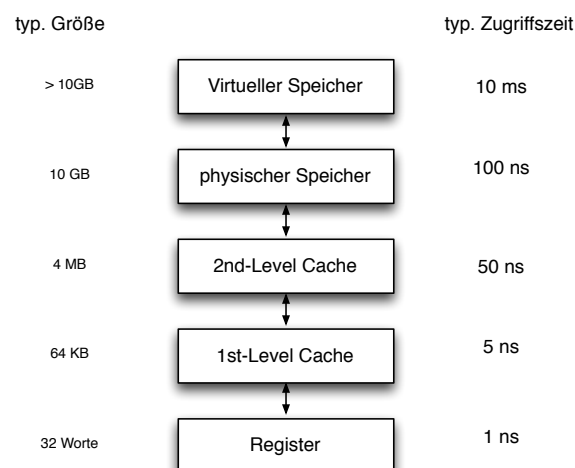
## 4.7 Heap Management

Während der statische Speicherbereich alle Datenobjekte enthält, deren Lebensdauer sich vom Start des Programms bis zum Ende erstreckt und deren Speicherzuordnung somit vom Compiler vor dem eigentlichen Programmlauf festgelegt werden kann, wird der Heap-Speicher hauptsächlich zum Speichern von Daten und Objekten benutzt, die unabhängig von Blockeintritten oder Prozeduraufrufen zur Laufzeit des Programms erzeugt und später wieder vom Programm freigegeben werden.

Die Speicheranforderung geschieht etwa über eine **new**-Anweisung in einer Prozedur, um Speicherplatz für ein neues Datenobjekt anzufordern (Konstruktor). Dieses Datenobjekt existiert aber auch noch, wenn die Prozedur beendet wird. In einigen Sprachen muss der Programmierer selbst dafür Sorge tragen, den Speicherplatz wieder frei zu geben (etwa in in C oder C++), in anderen Sprachen gibt es ein Garbage-Collector Verfahren, das den Heap nach nicht mehr benötigten Datenobjekten durchsucht und diese wieder dem freien Heap-Speicher zuordnet.

### 4.7.1 Die Speicherhierarchie eines Rechners

An dieser Stelle ist es wichtig, sich etwas genauer mit dem Aufbau des Speichers zu beschäftigen.



Wie man sieht, befindet sich eventuell nur ein Teil des gesamten Programms und der Daten im Hauptspeicher. Von diesem Teil befindet sich wiederum nur ein Teil im schnelleren Cache usw. Der Transfer zwischen den verschiedenen Speicherstufen findet immer in größeren Einheiten statt, beim virtuellem Speicher vom Betriebssystem gesteuert in etwa 32 KB großen Blöcken, bei den Cache-Speichern von der Hardware gesteuert in kleineren Blöcken von etwa 256 Bytes. Die Speicher-Hardware arbeitet üblicherweise mit einer Varianten des *most recently used*-Algorithmus, das bedeutet, dass versucht wird, die zuletzt benutzen Speicherbereiche im Cachespeicher zu behalten.

Hieran erkennt man einen wichtigen Hinweis für den Programmierer und den Compilerbauer. Es ist wichtig, das Programm so zu schreiben (und zu übersetzen!), dass man größtmögliche Lokalität erreicht. Es gilt immer noch die 80-20 Regel, die besagt, dass 80% der Rechenzeit in 20% des Maschinencodes verbraucht wird. Wichtigste Kandidaten für diese 20 Prozent Maschinencode sind die Rümpfe innerer Schleifen.

### 4.7.2 Speicherverwaltung

Die Speicherverwaltung für den Heap-Speicher muss zwei wichtige Aufgaben erfüllen:

**Allokation** Wenn ein Programm Speicherplatz für ein Objekt anfordert, muss die Speicherverwaltung ein genügend großes zusammenhängendes Stück aus dem Heap-Speicher finden und einen Referenz auf diesen Speicherbereich zurückgeben. Sollte nicht ausreichend Speicher zur Verfügung stehen, kann die Speicherverwaltung eventuell zusätzlichen Speicher vom Betriebssystem anfordern und so die Speicheranforderung erfüllen.

**Deallokation** Die Speicherverwaltung muss Speicherbereiche, die vom Anwendungsprogramm nicht mehr benötigt werden, wieder dem Bereich des freien Speichers zuordnen.

Die Speicherverwaltung sollte diese Aufgaben schnell und effizient erfüllen und dabei selbst nicht zu viele Ressourcen verbrauchen.

Um eine Fragmentierung des Speichers zu vermeiden, werden häufig Varianten des bekannten Best-Fit oder First-Fit Verfahrens bei der Allokation von Speicher angewendet. Fordert die Programmiersprache eine manuelle Deallokation des Speichers, ist die Aufgabe der Speicherverwaltung nicht schwer. Man muss nur darauf achten, dass nebeneinander liegende freie Bereiche des Speichers wieder zu einem größeren Bereich verschmolzen werden. Allerdings hat die manuelle Deallokation einen schweren Nachteil. Bei ungenauer Programmierung kann man eventuell vergessen, nicht mehr benötigte Speicherbereiche zurückzugeben (*memory leak*) oder man referenziert bereits zurückgegebene Speicherbereiche (*dangling-pointer*). Während der erste Teil zu erhöhtem Speicherbedarf und so eventuell nach einiger Laufzeit zu einem Abbruch des Programms wegen Speichermangels führen kann, ist der zweite Fehler viel schwerer zu lokalisieren, da das Programm eventuell noch einige Zeit weiterläuft, um dann entweder fehlerhafte Daten zu liefern oder aber mit einer Exception zu enden. Ausserdem ist diese Art Fehler häufig nicht einfach reproduzierbar und somit schwer zu lokalisieren.

Eine weitere wichtige Methode der Speicherverwaltung ist das Reference Counting. Jedes dynamisch allokierte Objekt wird um einen Zähler (**reference count**) erweitert. Jedes mal, wenn eine neue Referenz zu einem Objekt erzeugt wird, inkrementiert man den Zähler; wird eine Referenz gelöscht, wird der Zähler dekrementiert. Erreicht der Reference Count eines Objektes den Wert 0, kann das Objekt deallokiert werden. Enthält das Objekt Referenzen zu anderen Objekten, so müssen die Reference Counter der anderen Objekte entsprechend dekrementiert werden. Diese Methode hat Probleme bei zirkularen Strukturen, ausserdem kostet sie bei jeder Operation, die eine Referenz verändert.

Auch bei dieser Methode muss sehr sorgfältig programmiert werden, da ein vergessenes Inkrementieren bzw. Dekrementieren zu den gleichen Problemen wie bei der manuellen Speicherverwaltung führt. Diese Methode wurde lange Zeit von Apple in Objective C verwendet. In Xcode 4.2 wurde ein automatisches Reference Counting eingeführt, dass im Wesentlichen ohne Unterstützung des Programmierers funktioniert. Die entsprechenden Befehle zum Manipulieren der Reference Counts werden dabei vom Compiler eingefügt.

Für den Programmierer ist dieses Verfahren vergleichbar mit einem Garbage Collector Verfahren. Die Grundidee ist dabei, dass Objekte, die zu einem Zeitpunkt nicht über Referenzen erreichbar sind, endgültig nicht mehr erreichbar sind und daher freigegeben werden können. Eine wichtige Voraussetzung ist dabei, dass die Programmiersprache Typ-sicher ist. Das bedeutet, dass für jede Variable der Typ des Objekts und damit die Größe des Objekts auf das sie verweist, zumindest zur Laufzeit bekannt ist. Ausserdem muss man natürlich wissen, welche Teile des Objekts wiederum Referenzen auf andere Objekte enthalten, um diese gegebenenfalls auch freizugeben. Das ist z.B für C und C++ nicht erfüllt, da in diesen Sprachen Pointer-Arithmetik zulässig ist und somit Pointer auch mitten in ein Objekt zeigen können.



## 4.8 Garbage Collection

Was sind die Qualitätsmerkmale eines Garbage-Collector Verfahrens?

- Das Verfahren sollte die Laufzeit des Anwendungsprogramms (in diesem Kontext spricht vom **Mutator**, da das Programm den Heap-Speicher verändert) nicht zu sehr verlangsamen. Da der Garbage Collector viele Datenobjekte untersuchen muss, sollte er gut mit dem Speichersystem zusammenarbeiten.
- Das Verfahren sollte selbst nicht zu viel Speicherplatz benötigen und die Fragmentierung des freien Speichers klein halten.
- Da das Verfahren quasi parallel zum Anwendungsprogramm läuft, sollte es keine langen Pausen in der Abarbeitung des Anwendungsprogramms produzieren. Das ist besonders wichtig für real-time Anwendungen, wo man in zeitkritischen Bereichen häufig den Garbage-Collector abschalten muss, um eine gewisse Antwortzeit zu garantieren.
- Das Verfahren sollte die Lokalität des Anwendungsprogramms verbessern oder zumindest nicht verschlechtern.

Es ist klar, dass einige dieser Merkmale in Widerspruch zueinander stehen, so dass man Kompromisse eingehen muss. Werden in einer Programmiersprache hauptsächlich viele kleine Objekte erzeugt, sollte die Allokation des Speichers schnell gehen. Die Umspeicherung (**Reallokation**) der Objekte kostet dann nicht viel. Anders sieht es aus, wenn hauptsächlich große Objekte erzeugt werden. Hier ist der Aufwand für die Reallokation beträchtlich.

### 4.8.1 Erreichbarkeit von Objekten

Die Basismenge der erreichbaren Objekte sind alle, die direkt, ohne Dereferenzierung eines Zeigers, vom Programm aus erreicht werden können. In Java wären dies die Klassenvariablen (static) und die Variablen auf dem Laufzeit-Stack. Alle Objekte in der Basismenge sind direkt erreichbar. Rekursiv ist dann jedes Objekt, dessen Referenz in einer Instanzvariablen eines erreichbaren Objekts gespeichert ist, ebenfalls erreichbar.

Es dürfte klar sein, dass speziell bei optimierenden Compilern, Probleme hinzukommen, da Variablenwerte zeitweise nur in einem Register gehalten werden oder erst durch Addition eines Offsets zu einer korrekten Referenz werden (*versteckte Referenzen*).

In diesem Fall muss der Compiler dem Garbage Collector helfen. Er kann

- Garbage Collection nur zu gewissen Zeitpunkten erlauben, wenn diese „versteckten“ Referenzen nicht existieren.
- dem Garbage Collector zusätzliche Informationen mitgeben oder
- er kann garantieren, dass sich alle erreichbaren Objekte immer aus der Basismenge rekursiv bestimmen lassen.

Es gibt vier Situationen, in denen der Mutator die Menge der erreichbaren Objekte verändert:

- Allokation von neuen Objekten. Die Menge erreichbarer Objekte wächst dadurch an.
- Übergabe von Parameter und Rückgabewerten. Die Menge erreichbarer Objekte bleibt in diesem Fall gleich.

- Zuweisen von Referenzen. Bei einer Zuweisung  $a = b$ , bei der die Typen von  $a$  und  $b$  Referenzen zu Objekten sind, bleibt das von  $b$  referenzierte Objekt erreichbar, sofern  $a$  erreichbar ist. Gleichzeitig wird das ursprünglich von  $a$  referenzierte Objekt jetzt einmal weniger referenziert. War die Referenzierung von  $a$  die letzte für das Objekt, so wird dieses Objekt und alle Objekte, die nur über dieses Objekt erreichbar waren, unerreichbar. Die Menge erreichbarer Objekte kann also kleiner werden.
- Beenden von Prozeduren. Der zugehörige Aktivierungsrecord wird frei gegeben. Gibt es in ihm Referenzen zu Objekten, die nur so erreichbar sind, werden diese Objekte ebenfalls unerreichbar und natürlich auch alle weiteren Objekte, die nur so erreichbar waren.

#### 4.8.2 Garbage Collection über Reference Counting

Jedes Objekt hat ein zusätzliches Feld, das den Reference Count enthält. Der Zähler wird wie folgt benutzt:

- Bei Erzeugung eines Objekts wird dieser Zähler auf 1 gesetzt, da der Rückgabewert des Konstruktors eine Referenz ist.
- Wird ein Objekt als Parameter an eine Prozedur übergeben oder als Rückgabewert zurückgegeben, wird der Zähler inkrementiert.
- Bei einer Zuweisung  $a = b$  von Referenzen wird der Zähler des Objekts, auf das  $b$  verweist, inkrementiert und der Zähler des Objekts, auf das  $a$  verweist, dekrementiert.
- Beim Verlassen einer Prozedur müssen die Zähler aller Objekte, die über lokale Variable referenziert werden, dekrementiert werden. Verweisen mehrere lokale Variable auf das selbe Objekt, so muss der Zähler für jede Referenz dekrementiert werden.
- Wird der Reference Count eines Objekts auf 0 gesetzt, so müssen die Reference Counter alle Objekte, die über Instanzvariablen dieses Objekts direkt erreichbar sind, ebenfalls dekrementiert werden.

Diese Methode hat den Nachteil, dass der Overhead durch einzuführende extra Operationen relativ groß ist und nicht direkt von der Zahl der Objekte abhängt. Von Vorteil ist allerdings, dass dieses Garbage Collector Verfahren inkrementell stattfindet, so dass dieses Verfahren besonders bei zeitkritischen Anwendungen Vorteile hat.

Es dürfte klar sein, dass diese Methode bei zirkulären Strukturen, bei denen kein Mitglied in der Basismenge liegt, versagt. Hier muss man besondere Vorkehrungen treffen.

#### 4.8.3 Mark-and-Sweep Garbage Collection

Das Prinzip dieser Klasse von Garbage Collector Verfahren ist relativ simpel. Das Anwendungsprogramm (der Mutator) allokiert Speicher für neue Objekte. Nach einer gewissen Zeit, eventuell abhängig von der Größe des freien Speichers, beginnt der Garbage Collector die erreichbaren Objekte zu markieren. Dazu muss jedes Objekt Platz für ein Markierungsbit bereitstellen. Anschließend durchläuft der Garbage Collector den gesamten Heap und sammelt die nicht markierten Bereiche in einer Liste der freien Speicherbereiche, die anschließend vom Allokierungsalgorithmus wieder benutzt werden kann.

Eine Variante dieses Verfahren führt in der Sweep-Phase eine Kompaktifizierung des Speichers durch, indem es alle erreichbaren Objekte an ein Ende des Heaps verschiebt und damit einen großen zusammenhängenden freien Speicherbereich erzeugt.

Eine andere Strategie verfolgen die sogenannten Copying Garbage Collector Verfahren. Der gesamte Heap ist dabei in zwei Hälften aufgeteilt. Der Mutator arbeitet zunächst nur in einer Hälfte und allokiert dort Objekte, bis der freie Speicher in dieser Hälfte knapp wird. Der Garbage Collector kopiert dann die erreichbaren Objekte in die andere Hälfte des Heaps. Danach kann der Mutator weiter laufen wobei die Rollen der beiden Hälften vertauscht sind.

Es dürfte klar sein, dass bei all diesen Verfahren das Anwendungsprogramm für einen längeren Zeitraum unterbrochen wird.

Es gibt Verfahren, die die erste oder die zweite Phase des Garbage Collecting in kleinere Prozesse zerlegen, um die Pausen für das Anwendungsprogramm nicht zu lang werden zu lassen. Da es hier aber zu Interaktionen mit dem Mutator kommen kann, müssen diese Verfahren beim Bestimmen der erreichbaren Objekte einen beträchtlichen Extra-Aufwand treiben, um zu verhindern, dass erreichbare Objekte fälschlicherweise dem freien Speicher zugeteilt werden. Werden andererseits nicht erreichbare Objekte nicht dem freien Speicher zugeteilt und geschieht dies bei nicht zu vielen Objekten, macht das Verfahren keine Probleme, da bei der nächsten Aktivierung des Garbage Collectors diese Objekte garantiert nicht erreicht werden können (Man bedenke, dass deallokierte Objekte nie wieder vom Anwendungsprogramm referenziert werden können!).

## 5 Zwischencode-Erzeugung

Warum verwendet man einen Zwischencode und erzeugt nicht direkt ein Programm in der Zielsprache? Da gibt es mehrere Gründe:

- 1) Leichteres Ändern des Compilers auf eine neue Zielsprache – im Idealfall müsste nur der Maschinencode-Erzeuger geändert werden (das sogenannte **Back-End** des Compilers).
- 2) Die Code-Optimierung ist weitestgehend unabhängig von der Zielsprache und kann somit für mehrere Compiler benutzt werden.
- 3) Es ist einfacher, eine Übersetzung in zwei (oder mehr) kleineren Schritten als in einem großen Schritt durchzuführen.

### 5.1 Form des Zwischencodes

Man unterscheidet die folgenden Hauptkategorien von Formen des Zwischencodes:

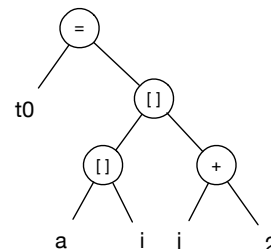
- HIL - High-level Intermediate Languages werden nur in den ersten Stufen des Übersetzungsprozesses benutzt.
- MIL - Medium-level Intermediate Languages spiegeln die Möglichkeiten der Quellsprache in sehr vereinfachter Form.
- LIL - Low-level Intermediate Languages sind nahe an der Zielsprache.

#### Beispiel 5.1:

Ein Feld `a` sei deklariert als `float a[20][10]`, wobei die untere Feldgrenze jeweils 0 sei. Der Ausdruck `a[i][j+2]` würde etwa übersetzt werden

HIL: in einen Ausdruck wie etwa

`t0 = a[i,j+2]` oder in einen Syntaxbaum wie



oder auch in einen gerichteten, azyklischen Graphen (DAG), wenn man gemeinsame Teilausdrücke berücksichtigen will.

MIL: in ein sehr einfach strukturiertes Programm, in dem jeder Befehl nur eine elementare Operation ausführt:

```

t1 = j + 2
t2 = i * 10
t3 = t1 + t2
t4 = 4 * t3      /* Größe einer Float-Zahl sei 4 Bytes */
t0 = a[t4]       /* a[t4] entspricht Anfangsadresse von a + t4 */

```

wobei die `ti` temporäre Namen sind.

Es wird also der Ausdruck  $(\text{addr } a) + 4 * (i * 10 + j + 2)$  berechnet. Jeder Befehl der Zwischensprache enthält nur eine Operation. Über die Speicherung des Feldes `a` muss nur bekannt sein, dass sie zeilenweise erfolgt und dass jedes Feldelement 4 Bytes belegt.

LIL: in eine einfache Art von Maschinensprachen eines hypothetischen Rechners unter Berücksichtigung der Speicherzuordnung.

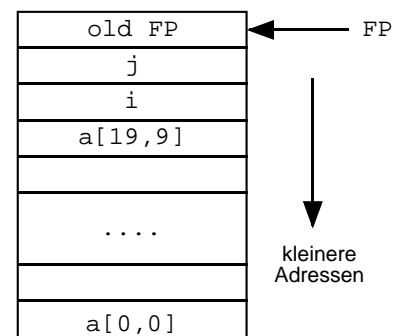
Hier wird angenommen, dass das Feld *a* lokal definiert ist und der Speicherplatz für *a* und die beiden Variablen *i* und *j* im Aktivierungs-Record reserviert wurde. Ein Frame-Pointer zeigt auf den Beginn des Aktivierungs-Records, dort sei zunächst Platz für die Variablen *j* und *i* reserviert und das Feld *a* schließt sich direkt an. Weiterhin sei angenommen, dass jede Integergröße 2 Bytes lang ist. Der Laufzeitstack wachse außerdem zu kleineren Adressen hin und FP bezeichne den Framepointer. Ein Ausdruck der Form *[c]* bezeichnet den Inhalt der Speicherzelle mit Adresse *c*. Die *ri* sind (fiktive) Register, die *fi* (fiktive) spezielle Floating-Point Register.

```

r1 <- [FP-2]      /* r1 enthält j */
r2 <- r1 + 2
r3 <- [FP-4]      /* r3 enthält i */
r4 <- r3 * 10
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- FP - 804    /* Startadresse */
                  /* des Feldes a */

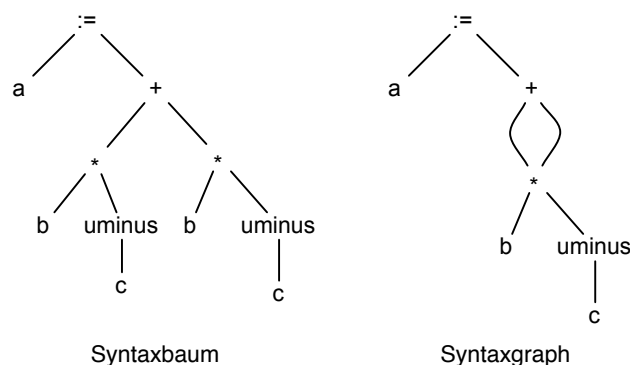
f1 <- [r7+r6]

```



### Beispiel 5.2:

Es soll beispielhaft die Übersetzung von Wertzuweisungen in Syntaxbäume betrachtet werden. Gegeben sei der Ausdruck *a := b\*-c+b\*-c*, dann wäre der zugehörige Syntaxbaum bzw. Syntaxgraph etwa



Wie kann man derartige Übersetzungen nun mit Hilfe einer attributierten Grammatik erzeugen? Betrachten wir zunächst das Problem für Syntaxbäume.

Gegeben seien zwei Funktionen:

- **mkleaf**(*id*, *id.entry*) erzeugt ein Blatt mit Markierung *id* und einem Zeiger in die Symboltabelle auf den korrespondierenden Eintrag.
- **mknode**(*op*, *left*, *right*) erzeugt einen internen Knoten mit Markierung *op* und zwei Zeigern mit Werten *left* und *right*. Die Variante **mknode**(*op*, *link*) erzeugt einen internen Knoten mit nur einem Zeiger.

Beide Funktionen geben einen Zeiger auf den erzeugten Knoten zurück.

Mit Hilfe dieser Funktionen ist es nun leicht, eine attributierte Übersetzung von Wertzuweisungen festzulegen. `nptr` ist ein synthetisches Attribut für jedes nichtterminale Symbol der Grammatik, das auf den Wurzelknoten des zugehörigen Syntaxbaums verweist.

Ein SDTS für diese Übersetzung wäre etwa:

Produktion	Semantische Regel
$S \rightarrow \text{id} := E$	$S.\text{nptr} := \text{mknode}(\text{'assign'}, \text{mkleaf}(\text{id}, \text{id.entry}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} := \text{mknode}(\text{'+'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr} := \text{mknode}(\text{'*'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow - E_1$	$E.\text{nptr} := \text{mknode}(\text{'uminus'}, E_1.\text{nptr})$
$E \rightarrow ( E_1 )$	$E.\text{nptr} := E_1.\text{nptr}$
$E \rightarrow \text{id}$	$E.\text{nptr} := \text{mkleaf}(\text{id}, \text{id.entry})$

Will man Syntaxgraphen erzeugen, so müssen die Funktion `mknode` und `mkleaf` vor dem Erzeugen eines neuen Knotens prüfen, ob ein entsprechender Knoten mit entsprechenden Nachfolgern bereits existiert. Falls dies der Fall ist, erzeugt man keinen neuen Knoten, sondern gibt den gefundenen Knoten zurück.

## 5.2 Drei-Adress-Befehle

Wir werden im Folgenden als Zwischencode einen sogenannten Drei-Adress-Code benutzen. Der Drei-Adress-Code ist eine MIL-Zwischensprache.

### Beispiele für Drei-Adress-Befehle:

- Wertzuweisungen der Form:  $x := y \text{ op } z$ ,  $\text{op} \in \{ +, -, *, /, <, =, \dots \}$
- Wertzuweisungen der Form:  $x := \text{op } y$ ,  $\text{op} \in \{ -, \text{not}, \text{intToReal}, \dots \}$
- Wertzuweisungen der Form:  $x := y$
- Wertzuweisungen der Form  $x := y[i]$  und  $x[i] := y$ ,  
dabei bedeutet  $y[i]$ : Adresse von  $y + i$  Speichereinheiten
- Wertzuweisungen der Form:  $x := \&y$  (Adresse von  $y$ ),  
 $x := *y$  (Dereferenzierung) oder  $*x := y$
- Unbedingte Sprünge der Form: `goto L`, dabei ist  $L$  eine Marke.  
(Marken können vor jedem Drei-Adress-Befehl auftreten.)
- Bedingte Sprünge der Form: `if x relOp y goto L`,  $\text{relOp} \in \{ <, =, \dots \}$ .

Bei den Sprungbefehlen kann man zwei Arten von Sprungzielen verwenden. Eine erste Möglichkeit ist es, einzelnen Drei-Adress-Befehlen eine symbolische Marke zuzuordnen. Eine zweite Möglichkeit besteht darin, die Drei-Adress-Befehle in ein Feld zu speichern und den Index des Feldes als Sprungziel zu verwenden.

### Beispiel 5.3:

Betrachtet man die Anweisung `do i = i+1; while (a[i] < v);` so wären die folgenden Übersetzungen in Drei-Adress-Code möglich:

L:	t1 = i + 1	100:	t1 = i + 1
	i = t1	101:	i = t1
	t2 = i * 8	102:	t2 = i * 8
	t3 = a [ t2 ]	103:	t3 = a [ t2 ]
	if t3 < v goto L	104:	if t3 < v goto 100

Betrachten wir nun die Übersetzung für Wertzuweisungen mit einfachen arithmetischen Ausdrücken in Drei-Adress-Code. In der folgenden S-attribuierten Grammatik sind der Variablen  $E$ , die für einen arithmetischen Ausdruck steht, zwei Attribute zugeordnet:

- 1)  $E.place$ : Namen der Variablen im Drei-Adress-Code, die den Wert des zu  $E$  korrespondierenden Ausdrucks enthält
- 2)  $E.code$ : Folge von Drei-Adress-Befehlen, die den zu  $E$  korrespondierenden Ausdruck auswertet.

`newtemp()` erzeugt bei jedem Aufruf einen neuen, temporären Variablennamen und `gen( ... )` erzeugt einen entsprechenden Drei-Adress-Befehl.

Produktion	Semantische Regel
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.lexeme \text{ ':=' } E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp()$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place \text{ ':=' } E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp()$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place \text{ ':=' } E_1.place * E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp()$ $E.code := E_1.code \parallel gen(E.place \text{ ':=' } 'uminus' E_1.place)$
$E \rightarrow ( E_1 )$	$E.place := E_1.place$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.lexeme$ $E.code := ''$

Wenn man sich die Regeln zur Code-Erzeugung für die obigen Produktionen ansieht, stellt man fest, dass alle nach dem gleichen Schema aufgebaut sind:

Einer Produktion  $A \rightarrow BC$  ist eine Regel der Form

$$A.code := string_B \parallel B.code \parallel string_C \parallel C.code \parallel string_D$$

zugeordnet.

Man nennt derartige Regeln, in denen Zeichenkettenattribute der einzelnen nichtterminalen Symbole in der Reihenfolge ihres Auftretens in der rechten Seite der Produktion mit eventuell weiteren konstanten Zeichenketten konkateniert werden **einfach**.

Derartige Regeln können ersetzt werden durch

$$A \rightarrow \{emit(string_B)\} B \{emit(string_C)\} C \{emit(string_D)\}$$

wobei `emit( .... )` die als Parameter übergebene Zeichenkette ausgibt.

Die Funktion `emit` ist eine Funktion mit Seiteneffekten, d.h. man muss auf die Reihenfolge der Auswertung achten!

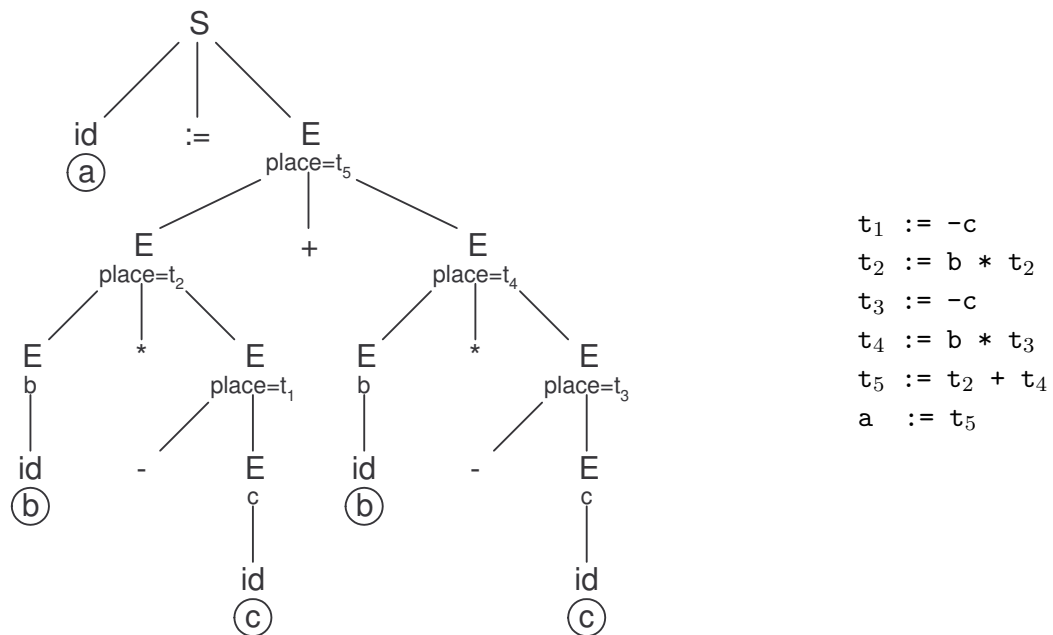
Mit diesen Änderungen erhalten wir das folgende SDTS zur Übersetzung in Drei-Adress-Code:

Produktion	Semantische Regel
$S \rightarrow \text{id} := E$	<code>emit(id.lexeme ':=' E.place)</code>
$E \rightarrow E_1 + E_2$	<code>E.place := newtemp();</code> <code>emit(E.place ':=' E<sub>1</sub>.place + E<sub>2</sub>.place)</code>
$E \rightarrow E_1 * E_2$	<code>E.place := newtemp();</code> <code>emit(E.place ':=' E<sub>1</sub>.place * E<sub>2</sub>.place)</code>
$E \rightarrow - E_1$	<code>E.place := newtemp();</code> <code>emit(E.place ':=' 'uminus' E<sub>1</sub>.place)</code>
$E \rightarrow ( E_1 )$	<code>E.place := E<sub>1</sub>.place</code>
$E \rightarrow \text{id}$	<code>E.place := id.lexeme</code>

Man kann sich vorstellen, dass die Funktion `emit` die so erzeugten Drei-Adress-Befehle etwa in ein größeres Feld oder in eine Datei schreibt, damit die nachfolgenden Stufen des Compilers das gesamte Drei-Adress-Programm weiterverarbeiten können.

#### Beispiel 5.4:

Für den Ausdruck aus Beispiel 5.2 erhält man damit den ausgewerteten Ableitungsbaum und Drei-Adress-Befehle





### 5.2.1 Darstellung von Drei-Adress-Befehlen

Es gibt zwei wesentliche Möglichkeiten der Darstellung von Drei-Adress-Befehlen:

#### Quadrupel-Darstellung:

Ein Befehl wird durch einen Record mit 4 Feldern dargestellt, etwa

Operator	1. Operand	2. Operand	Ergebnis
----------	------------	------------	----------

also z.B. für den Befehl  $x := y * z$  den Record

*	y	z	x
---	---	---	---

wobei  $x$ ,  $y$  und  $z$  Zeiger zum korrespondierenden Eintrag in der Symboltabelle sind. *Temporäre Namen müssen also bei dieser Methode in die Symboltabelle eingetragen werden.*

#### Tripel-Darstellung: (das Ergebnis wird durch den Befehl selbst repräsentiert).

Ein Befehl wird durch einen Record mit 3 Feldern dargestellt, etwa

Operator	1. Operand	2. Operand
----------	------------	------------

also z.B. für den Befehl  $x := y * z$  den Record

*	y	z
---	---	---

wobei  $y$  und  $z$  Zeiger zum korrespondierenden Eintrag in der Symboltabelle sind oder aber auf einen Drei-Adress-Befehl verweisen.

#### Beispiel 5.5:

Betrachten wir wieder den Ausdruck  $a := b * -c + b * -c$ . Äquivalente Folgen von Drei-Adress-Befehlen wären:

Quadrupel-Darstellung					und	Tripel-Darstellung			
Nr.	Op	1. Op	2. Op	Erg.		Nr.	Op	1. Op	2. Op
0	uminus	c		t1		0	uminus	c	
1	*	b	t1	t2		1	*	b	(0)
2	uminus	c		t3		2	uminus	c	
3	*	b	t3	t4		3	*	b	(2)
4	+	t2	t4	t5		4	+	(1)	(3)
5	:=	t5		a		5	:=	a	(4)

wobei (i) den Index des Drei-Adress-Befehls darstellt, mit dem der entsprechende Operand berechnet wurde.

Die Tripel-Darstellung hat den Vorteil, dass sie weniger Speicherplatz benötigt und die Symboltabelle nicht mit temporären Namen überflutet wird. Nachteilig ist, dass eine Umordnung der Befehle, etwa in der Optimierungsphase, nur schwierig zu realisieren ist und dass es bei einigen Drei-Adress-Befehlen (z.B. beim Befehl  $x[i] := y$  oder  $x := y[i]$ ) keine vernünftige, einfache Darstellung in Tripelform gibt.

Es ist in der Quadrupel-Darstellung möglich, die Anzahl der temporären Namen durch Wiederbenutzung signifikant zu verringern, allerdings handelt man sich bei der späteren Code-Optimierung dadurch zusätzliche Probleme ein.

### 5.3 Syntax-gesteuerte Übersetzung von Deklarationen

Als erstes soll jetzt ein SDTS zur Berechnung des Typs und des Offsets im Aktivierungs-Record für lokale Variablen vorgestellt werden. An dieser Stelle muss bereits etwas über die Zielmaschine bekannt sein – für jeden elementaren Datentyp benötigt man die Anzahl Bytes zur Darstellung dieser Größen. Hinzu kommen noch Informationen über Wortgrenzen und Alignment, denn z.B. kann eine `float`-Zahl bei einigen Zielmaschinen nicht auf jeder Adresse abgelegt werden. Dies soll hier jedoch nicht berücksichtigt werden.

Produktion	Semantische Regel
$P \rightarrow D$	<code>offset := 0;</code>
$D \rightarrow D ; D$	
$D \rightarrow \text{id} : T$	<code>enter (id.name, T.type, offset);</code> <code>offset := offset+T.width;</code>
$T \rightarrow \text{integer}$	<code>T.type := integer;</code> <code>T.width := 4;</code>
$T \rightarrow \text{real}$	<code>T.type := real;</code> <code>T.width := 8;</code>
$T \rightarrow \text{array} [\text{num}] \text{ of } T_1$	<code>T.type := array (num.val, T<sub>1</sub>.type);</code> <code>T.width := num.val * T<sub>1</sub>.width;</code>
$T \rightarrow \uparrow T_1$	<code>T.type := pointer (T<sub>1</sub>.type);</code> <code>T.width := 4;</code>

Bemerkungen zum obigen SDTS zur Übersetzung von Deklarationen:

T hat zwei synthetische Attribute:

- 1) **type** enthält den korrespondierenden Typausdruck
- 2) **width** enthält die zur Speicherung eines korrespondierenden Objektes benötigte Anzahl Bytes.

Es wird angenommen, dass der untere Index eines Feldes 0 ist. **offset** ist eine globale Größe, die zu Beginn auf 0 gesetzt wird. Damit dies geschieht, muss **offset** vor dem Ableiten von D in der ersten Produktion initialisiert werden. Dies erreicht man entweder mit einer Aktion, die innerhalb der rechten Seite der Produktion ausgeführt werden muss oder aber alternativ mit einer Umformung der Grammatik.

Die erste Produktion kann umgeschrieben werden zu

$$\begin{array}{l}
 P \rightarrow M D \\
 M \rightarrow \varepsilon \quad \{ \text{offset} := 0 \}
 \end{array}$$

`enter(name,type,offset)` erzeugt einen Eintrag in der Symboltabelle für **name** mit dem Typ **typ** und der relativen Adresse **offset** im Aktivierungs-Record.

Damit haben wir eine S-attributierte Grammatik erzielt, die die Deklaration abarbeitet und die notwendigen Einträge in die Symboltabelle vornimmt.

Bei einer Programmiersprache mit Blockstruktur können innere Blöcke natürlich so nicht abgearbeitet werden. Man benötigt eine anders strukturierte Symboltabelle, in der die Gültigkeitsbereiche der deklarierten Variablen sowie die Stufenzahl berücksichtigt werden muss. Der eigentliche Ablauf ist aber natürlich sehr ähnlich dem hier gezeigten und soll nicht weiter behandelt werden.

**Beispiel 5.6:**

Die Deklarationen

```
a: array [10] of real;
b: ↑ array [10] of integer;
c: real;
```

würden in der Symboltabelle zu den folgenden Einträgen führen:

a	array(10,real)	0
b	pointer(array(10,integer))	80
c	real	84

**5.4 Übersetzung von Feldzugriffen**

Es wird angenommen, dass die Feldelemente fortlaufend abgespeichert werden und dass jedes Element des Feldes eine feste Größe von  $w$  Bytes hat.

**5.4.1 Eindimensionale Felder**

Die Deklaration des Feldes sei **a**: array[low..high] of ....

Dann ergibt sich die Adresse des Feldelements **a**[*i*] zu

$$\text{base} + (i - \text{low}) * w,$$

wobei **base** die Anfangsadresse des Speicherbereichs für das Feld **a** und **low** der Index des ersten Elementes in **a** ist, d.h. **a**[**low**] beginnt auf Adresse **base**.

Durch Umformung erhält man die Summe

$$i * w + \text{base} - \text{low} * w,$$

wobei der zweite Summand eventuell bereits zur Übersetzungszeit bestimmt werden kann.

**5.4.2 Zweidimensionale Felder**

Die Deklaration des Feldes sei

```
a: array[low1..high1, low2..high2] of ....
```

Es gibt zwei prinzipielle Möglichkeiten:

- zeilenweise Speicherung (row-major), wird z.B. in Pascal angewendet
- spaltenweise Speicherung (column-major), wird z.B. in Fortran angewendet.

Wir wollen im folgenden eine zeilenweise Speicherung annehmen.

Dann ergibt sich die Adresse des Feldelements **a**[*i*<sub>1</sub>, *i*<sub>2</sub>] zu

$$\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

wobei  $n_2 = \text{high}_2 - \text{low}_2 + 1$ , d.h. die Anzahl der Spalten angibt.

Auch hier kann man die Formel umformen in einen Summanden, der zur Laufzeit berechnet werden muss und einen zweiten, der eventuell bereits zur Übersetzungszeit bekannt ist.

Man erhält:

$$((i_1 * n_2) + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w).$$

### 5.4.3 Mehrdimensionale Felder

Die Deklaration des Feldes sei nun

```
a: array[low1..high1, low2..high2, ..., lowk..highk] of ....
```

Verallgemeinert man die zeilenweise Speicherung auf mehr als zwei Dimensionen, dann erhält man eine sequentielle Speicherung der Feldelemente in einer Form, dass der letzte Index am schnellsten variiert.

Dann ergibt sich die Adresse des Feldelements  $a[i_1, i_2, \dots, i_k]$  zu

$$\text{base} + ((\dots (i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * n_3 + \dots + i_k - \text{low}_k) * w$$

wobei  $n_i = \text{high}_i - \text{low}_i + 1$  für  $1 < i \leq k$  gilt.

Durch Umformung erhält man wie oben einen Summanden, der zur Laufzeit berechnet werden muss und einen zweiten, der eventuell bereits zur Übersetzungszeit bekannt ist:

$$\begin{aligned} & ((\dots (i_1 * n_2) + i_2) * n_3 + \dots) * n_k + i_k) * w \\ & + \text{base} - ((\dots (\text{low}_1 * n_2) + \text{low}_2) * n_3 + \dots) * n_k + \text{low}_k) * w. \end{aligned}$$

Der zur Laufzeit zu berechnende erste Summand der Formel lässt sich wie folgt rekursiv bestimmen:

$$\begin{aligned} e_1 &:= i_1 \text{ und} \\ e_j &:= e_{j-1} * n_j + i_j \text{ für } 1 < j \leq k \end{aligned}$$

und  $e_k * w$  ergibt dann den Wert des ersten Summanden.

Wie übersetzt man nun Zugriffe auf Feldelemente?

Als erster Ansatz könnte folgende Grammatik dienen:

$$\begin{aligned} F &\rightarrow \text{id } [L] \\ L &\rightarrow L, E \mid E \end{aligned}$$

F steht dabei für ein Feldelement, L steht für eine Liste von Ausdrücken und E für einen Ausdruck. F und L haben je zwei synthetische Attribute:

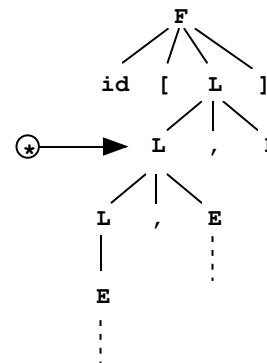
- **F.place** enthält den Namen einer Variablen, die den Wert des zweiten, hier als zur Übersetzungszeit bekannt angenommenen Summanden der obigen Zugriffsformel enthält,
- **F.offset** enthält den Namen einer Variablen, die den Wert des ersten Summanden der obigen Zugriffsformel enthält.
- **L.ndim** enthält die Anzahl der arithmetischen Ausdrücke in L
- **L.place** gibt den Namen der Variablen an, die den Wert der Zugriffsformel für die Ausdrücke in L, also den Wert  $e_j$  enthält.

Weiterhin soll es eine Funktion `limit(array, j)` geben, die für das Feld `array` den korrespondierenden Wert  $n_j = \text{high}_j - \text{low}_j + 1$  zurückgibt.

### Beispiel 5.7:

Zugriff auf `a[1,2,3]`. Wir betrachten den Ableitungsbaum und versuchen, ob eine Auswertung der Attribute in dieser Form möglich ist.

An der mit einem Stern markierten Stelle kann man keine korrekte Drei-Adress-Befehle für den Ausdruck  $e_2 = e_1 * n_2 + i_2$  erzeugen. Der Wert von  $n_2$  muss mit der Funktion `limit` aus der Symboltabelle extrahiert werden, aber der Name des Feldes ist an dieser Stelle nicht verfügbar!



Es gibt nun zwei Möglichkeiten, dieses Problem zu umgehen:

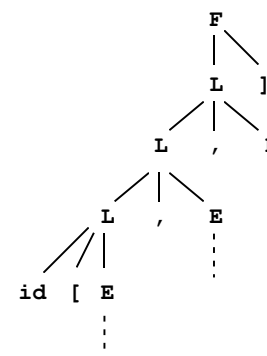
- 1) Einführung eines inheriten Attributs für L und Herunterreichen des Feldnamens im Ableitungsbaum
- 2) Umformung der Grammatik, um bei einer S-attributierten Grammatik bleiben zu können.

Eine Möglichkeit wäre:

$$\begin{array}{lcl} F & \rightarrow & L \ ] \\ L & \rightarrow & L, E \mid \text{id} \ [ \ E \end{array}$$

Damit erhalte man folgenden Ableitungsbaum für den Ausdruck aus Beispiel 5.7:

L benötigt noch ein weiteres synthetische Attribut `L.array`, das den Namen des Feldes bzw. einen Zeiger auf den entsprechenden Eintrag in der Symboltabelle enthält. Damit ist eine syntax-gesteuerte Übersetzung von Feldzugriffen möglich.



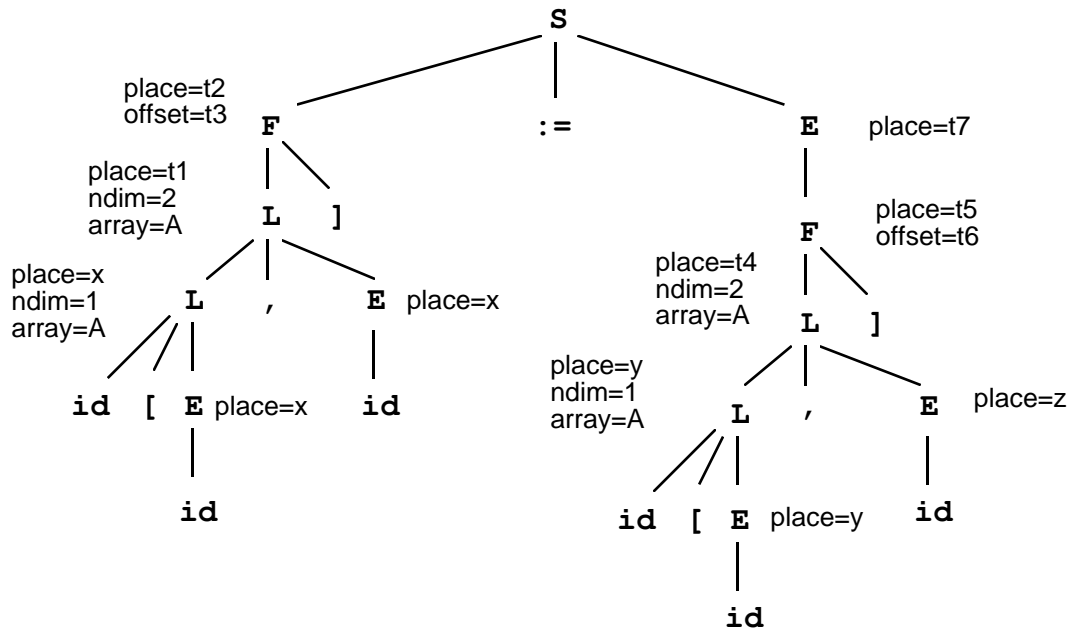
Mit dieser Idee kann man das folgende SDTS zur Übersetzung von Feldzugriffen definieren:

Produktion	Semantische Regel
$S \rightarrow \text{id} := E$	<code>emit (id.lexeme ':' E.place);</code>
$S \rightarrow F := E$	<code>emit (F.place '[' F.offset ']' ':' E.place);</code>
$E \rightarrow E_1 + E_2$	<code>E.place := newtemp(); emit (E.place ':' E<sub>1</sub>.place + E<sub>2</sub>.place);</code>
$E \rightarrow ( E_1 )$	<code>E.place := E<sub>1</sub>.place;</code>
$E \rightarrow \text{id}$	<code>E.place := id.lexeme;</code>
$E \rightarrow F$	<code>E.place := newtemp(); emit (E.place ':' F.place '[' F.offset ']);</code>
$F \rightarrow L ]$	<code>F.place := newtemp(); emit (F.place ':' L.array '-' <math>\sigma</math>(L.array); F.offset := newtemp(); emit ( F.offset ':' w '*' L.place);</code>
$L \rightarrow L_1 , E$	<code>t := newtemp(); j := L<sub>1</sub>.ndim + 1; emit ( t ':' L<sub>1</sub>.place '*' limit (L<sub>1</sub>.array, j )); emit (t ':' t '+' E.place ); L.array := L<sub>1</sub>.array; L.place := t;</code>
$L \rightarrow \text{id} [ E$	<code>L.ndim := j; L.place := E.place; L.ndim := 1; L.array := id.lexeme;</code>

**Bemerkung:** Der durch `emit(F.place ':' L.array '-'  $\sigma$ (L.array))` ausgegebene Drei-Adress-Befehl soll den festen Teil der Zugriffsformel darstellen. `L.array` entspricht dem Wert `base` und  `$\sigma$ (L.array)` dem geschachtelten Ausdruck.

**Beispiel 5.8:**

Sei  $A$  ein  $10 \times 20$ -Feld, d.h.  $A : \text{array}[1..10, 1..20]$  of  $\dots$ . Es ist also  $n_1 = 10$  und  $n_2 = 20$ . und es gelte  $w = 4$ . Übersetzt werden soll die Wertzuweisung  $A[x, x] := A[y, z]$ . Es ergibt sich folgender attributierter Ableitungsbaum:



Die erzeugten Drei-Adress-Befehle wären dann:

```

t1 := x * 20
t1 := t1 + x
t2 := A - 84
t3 := 4 * t1
t4 := y * 20
t4 := t4 + z
t5 := A - 84
t6 := 4 * t4
t7 := t5[t6]
t2[t3] := t7

```

wobei  $\sigma(A) = ((low_1 * n_2) + low_2) * w$  den Wert 84 ergibt.

### 5.5 Übersetzung Boolescher Ausdrücke

Es sollen im folgenden Boolesche Ausdrücke betrachtet werden, die gemäß der folgenden Grammatik festgelegt seien:

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relOp id} \mid \text{true} \mid \text{false}$$

Dabei steht **relOp** für einen der üblichen arithmetischen Vergleichsoperatoren und es wird angenommen, dass die Operationen **or** und **and** linksassoziativ sind und dass **not** eine höhere Priorität als **and**, und **and** wiederum eine höhere als **or** hat.

Es gibt nun zwei prinzipielle Möglichkeiten, den Wert eines Booleschen Ausdrucks darzustellen:

- 1) Durch numerische Kodierung, etwa  $1 \hat{=} \text{true}$  und  $0 \hat{=} \text{false}$  oder aber auch jeder Wert ungleich 0  $\hat{=} \text{true}$  und 0  $\hat{=} \text{false}$ .

Die Kodierung sollte so gewählt werden, dass für die Booleschen Operationen möglichst die in der Hardware der Zielmaschine vorhandenen Befehle verwendet werden können.

Diese Methode ist besonders geeignet, wenn Boolesche Werte etwa für Boolesche Variablen gespeichert werden müssen.

- 2) Durch die Steuerung des Programmablaufs, d.h. der Wert eines Booleschen Ausdrucks wird durch eine im Programm erreichte Position dargestellt. Diese Methode hat Vorteile bei der Implementation von Ablaufstrukturen.

Wichtig ist speziell bei dieser Methode die Frage, ob z.B. bei der Abarbeitung eines Ausdrucks  $E_1 \text{ or } E_2$  immer beide Ausdrücke abgearbeitet werden müssen, oder ob auf die Abarbeitung von  $E_2$  verzichtet werden kann, falls  $E_1$  bereits den Wert **true** hat. Diese „abgekürzte Auswertung“ wird in einigen Programmiersprachen gefordert (etwa in C), in anderen verboten (in Standard-Pascal von N. Wirth) und in einigen hat der Programmierer es in der Hand, welche Methode er wählt. (etwa in Java).

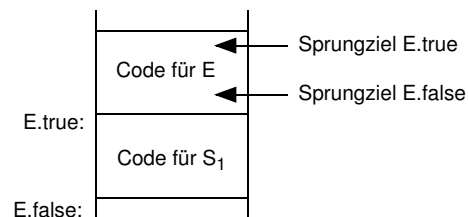




$$\begin{aligned}
S \rightarrow & \text{ if } E \text{ then } S_1 \mid \\
& \text{ if } E \text{ then } S_1 \text{ else } S_2 \mid \\
& \text{ while } E \text{ do } S_1
\end{aligned}$$

Betrachtet man die erste Produktion, so ist klar, dass zunächst der Drei-Adress-Code für  $E$  und danach der für  $S_1$  erzeugt wird.

Die Sprungziele der bedingten und unbedingten Drei-Adress-Sprungbefehle im Code für  $E$  müssen also *vor* Abarbeitung von  $S_1$  bekannt sein. Das ist das bekannte Problem der „forward references“, wie es auch z.B. in Assemblern auftritt. Die erste Lösungsmöglichkeit, die hier zunächst besprochen werden soll, besteht darin, Marken für die Sprungziele als Werte inheriter Attribute von  $E$  vorzugeben und diese Marken erst später als Sprungziel vor einem Drei-Adressbefehl zu schreiben.



$E$  erhält also drei Attribute:

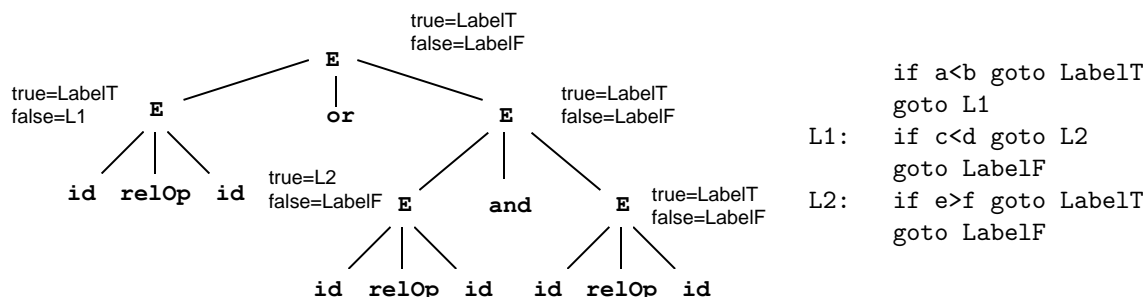
- 1) Das inherite Attribut **true** gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn der Ausdruck den Wert **true** ergibt,
- 2) das inherite Attribut **false** gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn der Ausdruck den Wert **false** ergibt und
- 3) das synthetische Attribut **code** enthält die Folge der Drei-Adress-Befehle zur Auswertung von  $E$ .

Damit kann man jetzt eine attributierte Grammatik für die Übersetzung Boolescher Ausdrücke aufstellen:

Produktion	Semantische Regel
$E \rightarrow E_1 \text{ or } E_2$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := \text{newlabel}();$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code};$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.\text{true} := \text{newlabel}();$ $E_1.\text{false} := E.\text{false};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code};$
$E \rightarrow \text{not } E_1$	$E_1.\text{true} := E.\text{false};$ $E_1.\text{false} := E.\text{true};$ $E.\text{code} := E_1.\text{code};$
$E \rightarrow ( E_1 )$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code};$
$E \rightarrow \text{id}_1 \text{ relOp } \text{id}_2$	$E.\text{code} := \text{gen}('if' \text{ id}_1.\text{place relOp.op id}_2.\text{place}$ $\text{'goto' } E.\text{true}) \parallel \text{gen}('goto' E.\text{false});$
$E \rightarrow \text{true}$	$E.\text{code} := \text{gen}('goto' E.\text{true});$
$E \rightarrow \text{false}$	$E.\text{code} := \text{gen}('goto' E.\text{false});$

**Beispiel 5.10:**

Der Ausdruck  $a < b$  or  $c < d$  and  $e > f$  würde zu dem folgenden attribuierten Ableitungsbaum führen und den angegebenen Drei-Adress-Code liefern. Hier wird angenommen, dass die Werte der inheriten Attribute des Startsymbols  $E$  an der Wurzel des Ableitungsbaumes „von außen“ zu  $labelT$  bzw.  $labelF$  gesetzt werden.



Nun ist es nicht mehr schwer, für Steueranweisungen in der Programmiersprache Zwischencode zu erzeugen. Betrachtet man etwa die Produktion  $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ , so ist es natürlich möglich, dass z.B.  $S_1$  oder  $S_2$  wiederum bedingte oder unbedingte Sprünge enthalten, die als Sprungziel den nach dieser Anweisung folgenden Befehl haben. Also muss man auch dem Symbol  $S$  ein inherites Attribut zuordnen, das dieses Sprungziel angibt.

$S$  hat also die folgenden Attribute:

- 1) das inherite Attribut **next** gibt das Sprungziel an, das in dem Fall angesprungen werden soll, wenn zu der  $S$  folgenden Anweisung gesprungen werden soll und
- 2) das synthetische Attribut **code** enthält die Folge der Drei-Adress-Befehle zur Auswertung von  $S$ .

Damit ergeben sich die folgenden Erweiterungen der obigen attribuierten Grammatik:

Produktion	Semantische Regel
$S \rightarrow \text{if } E \text{ then } S_1$	$E.true := \text{newlabel}();$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel \text{gen } (E.true \text{ ':' }) \parallel S_1.code;$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.true := \text{newlabel}();$ $E.false := \text{newlabel}();$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel \text{gen } (E.true \text{ ':' }) \parallel S_1.code \parallel$ $\text{gen } ('goto' S.next) \parallel \text{gen } (E.false \text{ ':' }) \parallel$ $S_2.code;$
$S \rightarrow \text{while } E \text{ do } S_1$	$E.true := \text{newlabel}();$ $E.false := S.next;$ $\text{beginlabel} := \text{newlabel}();$ $S_1.next := \text{beginlabel};$ $S.code := \text{gen } (\text{beginlabel} \text{ ':' }) \parallel E.code \parallel$ $\text{gen } (E.true \text{ ':' }) \parallel S_1.code \parallel$ $\text{gen } ('goto' \text{beginlabel});$

**Beispiel 5.11:**

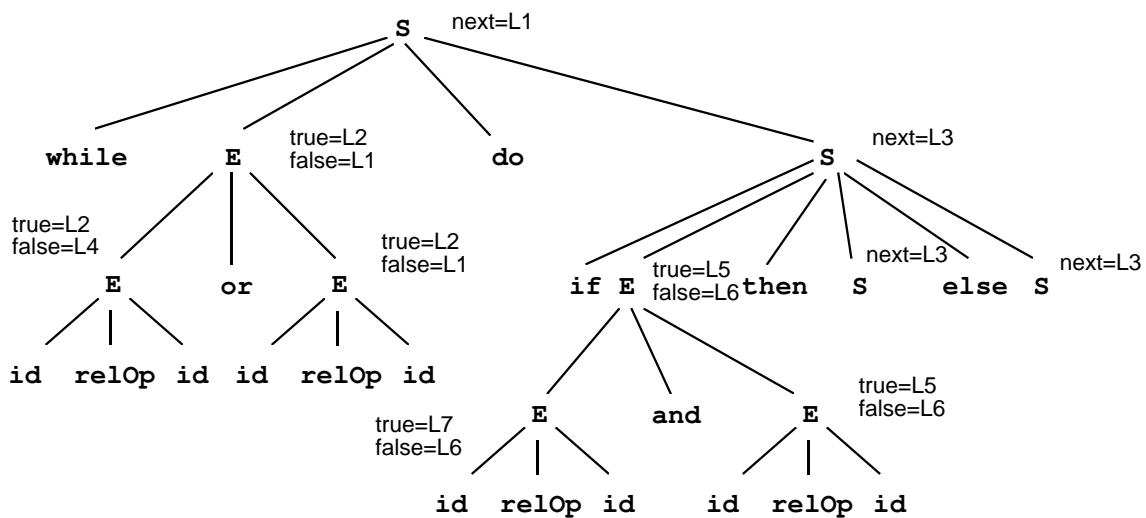
Es soll folgendes Programmfragment mit der obigen attribuierten Grammatik übersetzt werden:

```

while a<b or e>f do
  if c<d and g<h then
    x := y+z
  else
    x := y-z

```

Zunächst wird der zugehörige attribuierte Ableitungsbaum betrachtet:



Man erhält durch die Auswertung den folgenden Drei-Adress-Code:

```

L3:  if a<b goto L2
      goto L4
L4:  if e>f goto L2
      goto L1
L2:  if c<d goto L7
      goto L6
L7:  if g<h goto L5
      goto L6
L5:  t1 := y+z
      x := t1
      goto L3
L6:  t2 := y-z
      x := t2
      goto L3

```

## 5.6 Übersetzung Boolescher Ausdrücke mit Backpatching

Die Übersetzung Boolescher Ausdrücke mit der bisherigen Attributierung lässt sich nicht mit einer einfachen Top-Down Analyse koppeln, falls man als Sprungziele Indizes im Feld der abgespeicherten Drei-Adress-Befehle benutzen möchte. Das ist natürlich das übliche Problem bei Vorwärts-Referenzen, d.h. bei Sprüngen, deren Ziel ein nachfolgender Befehl ist. Es sind zwei Standardmethoden bekannt, wie man ein solches Problem lösen kann. Zum einen gibt es die übliche 2 Pass-Methode, bei der der Programmcode zweimal gelesen wird und es gibt die **Backpatch**-Methode, bei der unvollständige Sprungbefehle mit leerem Sprungziel erzeugt werden. Kommt man im Verlauf der weiteren Übersetzung an das eigentliche Sprungziel, so werden alle korrespondierenden unvollständigen Sprungbefehle vervollständigt, indem das jetzt bekannte Sprungziel nachgetragen wird.

Wir wollen uns vorstellen, dass die generierten Drei-Adress-Befehle in einem Feld abgelegt werden und wir wollen den Index des Feldes zur Lokalisierung von Befehlen benutzen. Das bedeutet, dass Marken jetzt Feldindizes sind. Bei der Übersetzung müssen also Listen mit Indizes von unvollständigen Sprungbefehlen manipuliert werden.

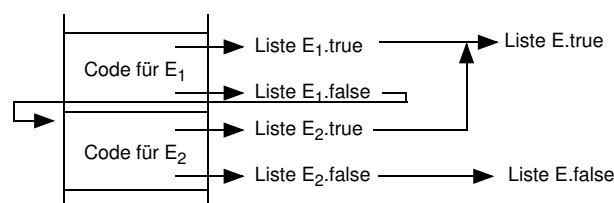
Zunächst werden drei Prozeduren zur Verwaltung derartiger Listen definiert:

- 1) **makelist(i)** erzeugt eine neue Liste mit einem Eintrag i und gibt einen Zeiger auf diese Liste zurück.
- 2) **merge(p<sub>1</sub>, p<sub>2</sub>)** erzeugt eine Liste, die aus den Elementen der beiden Listen besteht, auf die p<sub>1</sub> und p<sub>2</sub> verweisen. Es wird ein Zeiger auf die Ergebnisliste zurückgegeben.
- 3) **backpatch(p, j)** ist die eigentliche Backpatch-Prozedur. p zeigt auf eine Liste mit Indizes unvollständiger Sprungbefehle. In jedem dieser Befehle wird durch diese Prozedur das Sprungziel j eingetragen.

Die Variable E erhält zwei synthetische Attribute, nämlich

- 1) **truelist** - ein Zeiger auf eine Liste der Indizes im Feld der Drei-Adress-Befehle. An diesen Indizes befinden sich die unvollständigen Sprungbefehle, über die aus dem Drei-Adress-Code von E gesprungen werden soll, falls der Ausdruck E wahr ist.
- 2) **falselist** - wie **truelist**, nur muss der Ausdruck E falsch sein.

Man kann nun im wesentlichen die Grammatik aus dem vorigen Abschnitt verwenden. Zur Verdeutlichung betrachten wir die Produktion  $E \rightarrow E_1 \text{ or } E_2$ .



Wie man sieht, muss man die unvollständigen Sprünge in der Liste **E<sub>1</sub>.false** mit dem Index des ersten Drei-Adress-Befehls von **E<sub>2</sub>** als Sprungziel vervollständigen. Um diesen Wert festzuhalten, muss nach der Ableitung von **E<sub>1</sub>** und vor der Ableitung von **E<sub>2</sub>** der nächste freie Index im Feld der Drei-Adress-Befehle gespeichert werden. Zu diesem Zweck muss an einigen Stellen in den rechten Seiten der Produktionen eine neue, auf das leere Wort ableitbare Variable eingeführt werden, deren zugeordnete semantische Regel nur daraus besteht, die momentane Position im Feld der Drei-Adress-Befehle in einem synthetischen Attribut zu speichern.

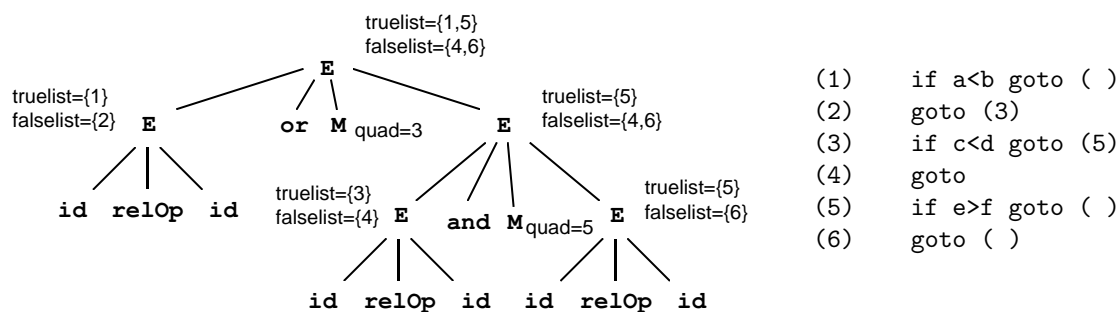
Zu diesem Zweck führt man die Variable  $M$  mit der Produktion  $M \rightarrow \varepsilon$  ein.  $M$  erhält ein synthetisches Attribut `quad`, dessen Wert den Index des nächsten freien Platzes im Feld der Drei-Adress-Befehle speichert.

Damit kann man dann die Übersetzung Boolescher Ausdrücke definieren:

Produktion	Semantische Regel
$E \rightarrow E_1 \text{ or } M E_2$	<code>backpatch (E<sub>1</sub>.falselist, M.quad);</code> <code>E.truelist := merge (E<sub>1</sub>.truelist, E<sub>2</sub>.truelist);</code> <code>E.falselist := E<sub>2</sub>.falselist;</code>
$E \rightarrow E_1 \text{ and } M E_2$	<code>backpatch (E<sub>1</sub>.truelist, M.quad);</code> <code>E.truelist := E<sub>2</sub>.truelist;</code> <code>E.falselist := merge (E<sub>1</sub>.falselist, E<sub>2</sub>.falselist);</code>
$E \rightarrow \text{not } E_1$	<code>E.truelist := E<sub>1</sub>.falselist;</code> <code>E.falselist := E<sub>1</sub>.truelist;</code>
$E \rightarrow ( E_1 )$	<code>E.truelist := E<sub>1</sub>.truelist;</code> <code>E.falselist := E<sub>1</sub>.falselist;</code>
$E \rightarrow \text{id}_1 \text{ relOp id}_2$	<code>E.truelist := makelist (nextquad());</code> <code>E.falselist := makelist (nextquad()+1);</code> <code>emit ('if' id<sub>1</sub>.place relOp.op id<sub>2</sub>.place 'goto ( )');</code> <code>emit ('goto ( )');</code>
$E \rightarrow \text{true}$	<code>E.truelist := makelist (nextquad());</code> <code>emit ('goto ( )');</code>
$E \rightarrow \text{false}$	<code>E.falselist := makelist (nextquad());</code> <code>emit ('goto ( )');</code>
$M \rightarrow \varepsilon$	<code>M.quad := nextquad();</code>

### Beispiel 5.12:

Der Ausdruck `a<b or c<d and e>f` würde mit diesem SDTS zu folgendem attribuierten Ableitungsbaum führen und den angegebenen Drei-Adress-Code liefern:



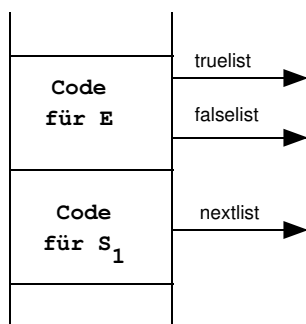
## 5.7 Übersetzung von Programmsteuerbefehlen (mit Backpatching)

Im folgenden soll gezeigt werden, wie man mit Hilfe der „Backpatch-Methode“ auch Steuerbefehle elegant übersetzen kann. Wir erweitern die Grammatik für Boolesche Ausdrücke um die folgenden Produktionen, wobei *S* für eine Anweisung, *L* für eine Liste von Anweisungen und *A* für eine Wertzuweisung steht.

$$\begin{aligned} S &\rightarrow \quad \text{if } E \text{ then } S_1 \\ &\quad | \text{if } E \text{ then } S_1 \text{ else } S_2 \\ &\quad | \text{while } E \text{ do } S_1 \\ &\quad | \text{begin } L \text{ end} \\ &\quad | A \\ L &\rightarrow \quad L ; S \\ &\quad | S \end{aligned}$$

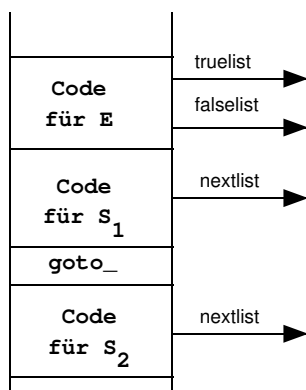
*S* und *L* bekommen jeweils ein synthetisches Attribut **nextlist** - ein Zeiger auf eine Liste der Indizes im Feld der Drei-Adress-Befehle, die die unvollständigen Sprungbefehle kennzeichnen, über die an die auf *S* bzw. *L* folgende Anweisung gesprungen werden soll.

Betrachten wir etwa die **while**-Anweisung. Es wird zunächst der Code für den Booleschen Ausdruck und danach der Code für den Rumpf erzeugt.



Alle Sprünge in **truelist** müssen auf den Anfang des Codes von *S*<sub>1</sub>, alle Sprünge in **nextlist** müssen dagegen auf den Anfang des Codes von *E* verweisen. Man muss also, um die Sprungziele festzuhalten, an zwei Stellen in der rechten Seite der Produktion das zusätzliche nichtterminale Symbol *M* einführen. Mit den Werten des Attributs **quad** kann dann die **backpatch**-Prozedur für die beiden Listen aufgerufen werden. Das Attribut **nextlist** bekommt dann die Liste **falselist** übergeben. Am Ende muss noch ein unbedingter Sprung an den Anfang des Codes von *E* eingefügt werden, damit man nicht aus dem Rumpf der Schleifen „herausfallen“ kann.

Ähnliche Überlegungen muss man etwa bei der **if-then-else** Anweisung anstellen.



Alle Sprünge in **truelist** müssen auf den Anfang des Codes von *S*<sub>1</sub>, alle Sprünge in **falselist** müssen dagegen auf den Anfang des Codes von *S*<sub>2</sub> verweisen. Beide Positionen müssen wieder durch Einfügen des zusätzlichen nichtterminalen Symbols *M* bereitgestellt werden. Damit nach Abarbeitung des Codes für *S*<sub>1</sub> nicht gleich in den Code für *S*<sub>2</sub> verzweigt wird, muss an dieser Stelle der unbedingte Sprung eingesetzt werden. Dies erreicht man wieder durch Einföhrung einer neuen Variablen *N* in die rechte Seite der Produktion und Hinzufügung einer Produktion  $N \rightarrow \varepsilon$  mit der entsprechenden Regel. Da das Sprungziel nicht bekannt ist, erhält *N* ein Attribut **nextlist** mit der gleichen Bedeutung wie bei *S*. Das Attribut **nextlist** bekommt dann als Wert die Vereinigung der drei **nextlist** Listen von *S*<sub>1</sub>, *S*<sub>2</sub> und *N* übergeben.





Man erhält den folgenden Drei-Adress-Code:

```
( 1)   if a<b goto (5)
( 2)   goto (3)
( 3)   if e>f goto (5)
( 4)   goto (.)
( 5)   if c<d goto (7)
( 6)   goto (12)
( 7)   if g<h goto (9)
( 8)   goto (12)
( 9)   t1 := y+z
(10)   x := t1
(11)   goto (1)
(12)   t2 := y-z
(13)   x:= t2
(14)   goto (1)
```

## 6 Codeoptimierung

Dieser Abschnitt ist eine kurze Einführung in die prinzipielle Vorgehensweise bei der Codeoptimierung. Ziel der Codeoptimierung ist die Umformung eines Zwischencode-Programms in ein kürzeres oder schnelleres, aber funktional äquivalentes Programm. Der Begriff „Codeoptimierung“ ist dabei eigentlich irreführend, denn Aho, Johnson und Ullman zeigten in [21], dass eine optimale Code-Erzeugung für eine hypothetische Maschine mit einer noch einfacheren Struktur als die einer Drei-Adress-Code verarbeitenden Maschine, bereits NP-vollständig ist.

### 6.1 Einführung

Ansatzpunkt für die Optimierung sind die Rümpfe innerer Schleifen, denn nach der bekannten 80-20 Regel werden etwa 80% der Zeit in 20% des Codes verbraucht. Es ist sicherlich nicht sinnvoll, auf den nur einmal durchlaufenen Initialisierungsteil eines Programms eine komplizierte und aufwändige Optimierung anzusetzen.

Forderungen an die zur Codeoptimierung benutzten Programmtransformationen:

- 1) Eine Programmtransformation darf die Funktion eines Programms nicht ändern. Das ist einleuchtend für das Input/Output Verhalten, schwieriger wird es schon bei Laufzeitfehlern. Ein zurückhaltendes Vorgehen bei der Codeoptimierung ist aber wichtig.
- 2) Eine angewendete Programmtransformation sollte, zumindest im Mittel, die Laufzeit des Programms messbar verringern.
- 3) Eine Programmtransformation sollte kosteneffektiv sein, d.h. der Kostenaufwand sollte in vernünftiger Relation zur Verbesserung der Laufzeit des Programms stehen. Es daher ist wenig sinnvoll, komplizierte Codeoptimierungen auf Programme anzuwenden, die nur ein paar mal aufgerufen werden.

Um ein wenig in die Problematik einzuführen betrachten wir zunächst das folgende Beispiel (nach [23]), das die Problematik von Programmtransformationen zeigt.

#### Beispiel 6.1:

Betrachten wir zunächst das folgende Ausgangsprogramm mit  $n \leq m$  und  $k \leq m$ .

```
subroutine tricky (a, b, n, m, k);
integer i, n, m, k;
int a[m], b[m];
for i := 1 to n do
    a[i] := b[k] + a[i] + 100000;
end;
```

Wie man sofort sieht, gibt es eine „schleifeninvariante Berechnung“, d.h. eine Berechnung, die bei jedem Durchlauf durch die Schleife den gleichen Wert liefert. Zieht man diese Berechnung vor die Schleife, so erhält man:

```
subroutine tricky (a, b, n, m, k);
integer i, n, m, k;
int a[m], b[m], C;
C := b[k] + 100000;
for i := 1 to n do
    a[i] := a[i] + C;
end;
```

und man kann vermuten, dass dieses Programm schneller läuft, da der Term  $b[k] + 100000$  nur einmal ausgewertet werden muss.

Diese harmlos aussehende Änderung hat nun aber folgende Konsequenzen:

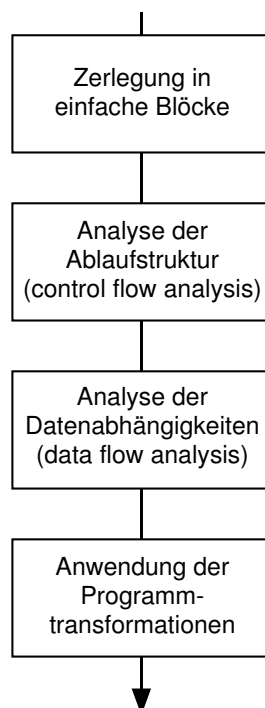
- 1) Es kann sein, dass im Originalprogramm alles richtig läuft, während im transformierten Programm ein Überlauf signalisiert wird! Der Term  $b[k] + 100000$  kann für die Integerdarstellung zu groß sein, während etwa die Berechnung von  $b[k] + a[i] + 100000$  durchaus Ergebnisse im Darstellungsbereich liefern kann. Erschwerend kommt hinzu, dass ein derartiger Überlauf *vor* Eintritt in die Schleife gemeldet würde. Wenn der Rumpf der Schleife etwa mit einer print-Anweisung beginnen würde, würde der Überlauf vor dem ersten Ausdruck gemeldet!
- 2) Ein Aufruf der Form

```
k := m+1;
n := 0
call tricky(a, b, n, m, k);
```

würde(hoffentlich) in der optimierten Version zu einer Fehlermeldung führen.  
Die Originalversion würde dagegen korrekt arbeiten.

Um überhaupt Programmtransformationen vernünftig ansetzen zu können, muss erst einmal das Drei-Adress-Programm analysiert werden.

Ein möglicher Aufbau eines einfachen Codeoptimierers wäre:



In der ersten Phase wird das Drei-Adress-Programm in sogenannte einfache Blöcke zerlegt. Ein einfacher Block ist dabei ein Teilstück, das garantiert vom ersten bis zum letzten Befehle sequentiell durchlaufen wird.

Die einfachen Blöcke bilden die Knoten eines gerichteten Flussgraphen. Dieser Graph wird in der nächsten Phase benutzt, um z.B. Informationen über die Ablaufstruktur zu sammeln. Es wird dabei auch versucht, Schleifen im Drei-Adress-Code zu identifizieren.

In der anschließenden Phase werden Definition und Verwendung von Variablenwerten in Beziehung gesetzt. Zum Beispiel interessiert vielleicht bei einem Drei-Adress-Befehl  $a := b \text{ op } c$ , an welchen Stellen im Programm die hier benötigten Variablenwerte von  $b$  und  $c$  definiert wurden und ob deren Werte nach diesem Befehl noch an anderer Stelle benötigt werden. Typischerweise muss man, um derartige Fragen beantworten zu können, größere Gleichungssysteme (Datenfluss-Gleichungen) lösen. Dieses Schema ist relativ maschinenunabhängig und eignet sich daher für einen allgemeinen Codeoptimierer. Die einzelnen Phasen sind aber meist nicht so strikt getrennt, sondern für eine bestimmte mögliche Programmtransformation werden die dazu benötigten Informationen gesammelt und daraufhin entschieden, ob die Transformation durchgeführt wird. Häufig beschränkt man sich darauf, den Drei-Adress-Code einer jeden Prozedur getrennt zu optimieren.

## 6.2 Einfache Blöcke und Flussgraphen

Idee: Partitionierung der Drei-Adress-Befehle des Zwischencode-Programms in maximale, zusammenhängende Teile, die bei Ablauf des Programms immer sequentiell vom ersten bis zum letzten Befehl abgearbeitet werden müssen. Diese Teile werden **einfache Blöcke** genannt.

### Beispiel 6.2:

Betrachte das folgende Fragment eines Drei-Adress-Programms:

```

        t1 := a + t1
L1:    t2 := a + c
        t3 := t1 * t2
        t4 := t3 + t1
        if t2 < t4 goto L1
        t1 := 2 * t4

```

Die Folge der Drei-Adress-Befehle vom 2. bis zum 5. Befehl bilden einen einfachen Block. Der 1. Befehl gehört nicht dazu, da andernfalls über die Marke L1: in den Block eingetreten werden könnte. Der 6. Befehl gehört ebenfalls nicht mit zu diesem Block, da nicht in jedem Fall nach dem 5. der 6. Befehl abgearbeitet wird.

### 6.2.1 Algorithmus zur Partitionierung in einfache Blöcke:

**Eingabe:** Eine Folge von Drei-Adress-Befehlen

**Ausgabe:** Eine Liste einfacher Blöcke; jede Anweisung befindet sich in genau einem einfachen Block.

**Verfahren:**

- 1) Zuerst wird die Menge der führenden Befehle in einem einfachen Block markiert.
  - i) Die erste Anweisung des Drei-Adress-Programms wird markiert.
  - ii) Jede Anweisung, die das Ziel eines bestimmten oder unbestimmten Sprunges ist, wird markiert.
  - iii) Jede Anweisung, die einem bestimmten oder unbestimmten Sprung oder einem Prozeduraufruf folgt, wird markiert.
- 2) Für jeden führenden Befehl besteht der zugeordnete einfache Block aus diesem Befehl und allen nachfolgenden Befehlen ausschließlich des nächsten markierten Befehls oder dem Ende des Drei-Adress-Programms.

### 6.2.2 Nächster Gebrauch oder der Lebendigkeit von Variablen

An dem obigen Beispiel kann man auch noch weitere wichtige Begriffe erläutern:

- Der 3. Befehl `t3 := t1 * t2` repräsentiert eine Position im Drei-Adress-Programm, an der die Variable **t3 definiert** wird und an der die Variablen **t1** und **t2 benötigt** (oder auch **gebraucht**) werden.
- Manchmal wird der Gebrauch einer speziellen Definition einer Variablen betrachtet. Im Beispiel wird im 3. Befehl die Definition der Variablen **t1** im 1. Befehl benötigt. Wichtig ist dabei, dass zwischen der Definition und dem Gebrauch keine Neudefinition von **t1** stattfindet.
- Ein Programm-Punkt im Drei-Adress-Programm charakterisiert eine Position vor oder nach einem Drei-Adress-Befehl.

- Am Programm-Punkt nach dem 4. Befehl ist die Variable **t1 lebendig**, weil der Wert von **t1** an anderer, erreichbarer Stelle (in diesem Fall im 3. Befehl) im Programm benötigt wird. Die Erreichbarkeit ist natürlich nur hypothetisch und kann auch in einem anderen einfachen Block liegen. Man nimmt dabei also an, dass jede Alternative eines bedingten Sprungs auch möglich ist.
- Die Variable **t3** ist an diesem Programm-Punkt **nicht lebendig** oder **tot**, da kein weiterer Gebrauch von **t3** erreichbar ist.
- Auch hier kann man genauer von der Lebendigkeit einer speziellen Definition der Variablen sprechen. Am Programm-Punkt nach dem 4. Befehl ist die Definition der Variable **t1** im 1. Befehl lebendig. Die Definition von **t1** im letzten Befehl ist dagegen an diesem Programm-Punkt nicht lebendig

### 6.2.3 Konstruktion des Flussgraphen

Aus der Liste der einfachen Blöcke wird ein gerichteter Graph erzeugt, der den Kontrollfluss innerhalb des Drei-Adress-Programms berücksichtigt. Die einfachen Blöcke bilden die Knoten des Flussgraphen. Der Knoten, der den ersten Drei-Adress-Befehl enthält, wird als **Startknoten** ausgezeichnet.

Eine gerichtete Kante führt vom Knoten  $B_1$  zum Knoten  $B_2$ , falls

- 1) es einen bedingten oder unbedingten Sprung vom letzten Befehl in  $B_1$  zum ersten Befehl in  $B_2$  gibt oder
- 2) der erste Befehl von  $B_2$  im Drei-Adress-Programm unmittelbar nach dem letzten Befehl von  $B_1$  folgt und der letzte Befehl in  $B_1$  kein unbedingter Sprung ist.

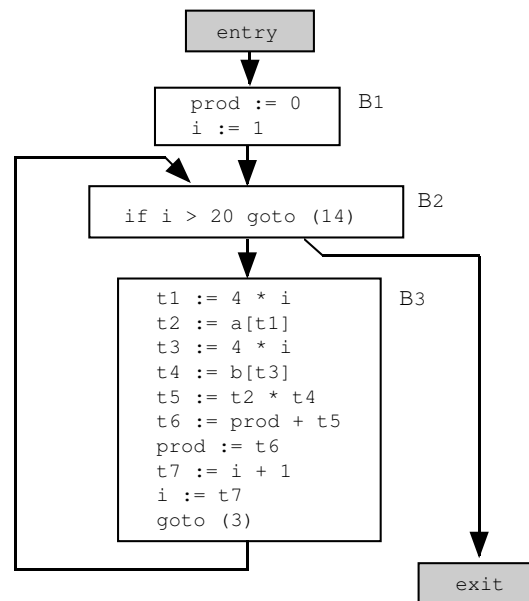
**Hinweis:** In manchen Situationen ist es vorteilhaft, zwei zusätzliche, künstliche einfache Blöcke **entry** und **exit** dem Flussgraphen hinzuzufügen (**erweiterter Flussgraph**). Vom Block **entry** gibt es genau eine Kante zum ausgezeichneten Startknoten. Ferner führt eine Kante von jedem Block, über die der Flussgraph verlassen werden kann, zum Block **exit**.

#### Beispiel 6.3:

Betrachten wir das folgende Programmfragment mit der möglichen Übersetzung in Drei-Adress-Code:

<pre> begin   prod := 0;   i := 1;   while i &lt;= 20 do     begin       prod := prod + a[i] * b[i];       i := i+1;     end   end; end; </pre>	<pre> (1)  prod := 0 (2)  i := 1 (3)  if i &gt; 20 goto (14) (4)  t1 := 4 * i (5)  t2 := a[t1] (6)  t3 := 4 * i (7)  t4 := b[t3] (8)  t5 := t2 * t4 (9)  t6 := prod + t5 (10) prod := t6 (11) t7 := i + 1 (12) i := t7 (13) goto (3) </pre>
---	---

Man erhält folgenden Flussgraphen:



An diesem Beispiel ist auch zu sehen, dass es in dieser Darstellung zweckmäßig ist, als Sprungziele bedingter und unbedingter Sprünge die entsprechenden einfachen Blöcke zu verwenden.

### 6.3 Möglichkeiten der Codeoptimierung

Bevor die Methoden zur Analyse der Datenflussabhängigkeiten und der Ablaufstruktur genauer vorgestellt werden, soll zunächst einmal ein umfangreiches Beispiel (aus [1]) vorgestellt werden, dass die Möglichkeiten zur Codeoptimierung zeigt.

#### Beispiel 6.4:

Betrachtet wird eine Implementation einer Quicksort-Variante. Das zu sortierende Feld sei  $a[0..max]$ . Für jedes Teilfeld  $a[m..n]$  sei  $a[n]$  das Pivotelement.

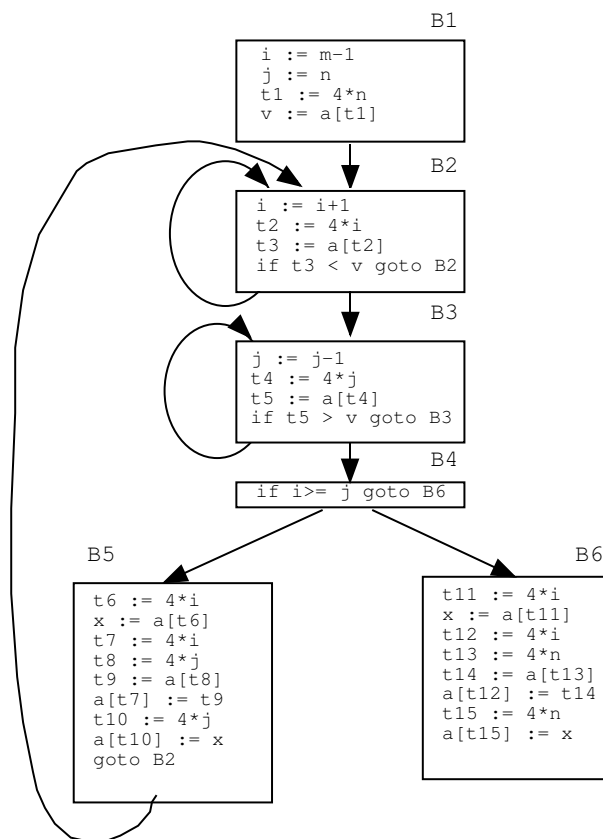
```

void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* ab hier beginnt das betrachtete Codefragment */
    i = m-1;  j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* hier endet das betrachtete Codefragment */
    quicksort(m,j); quicksort(i+1,n);
}
  
```

Für den oben bezeichneten Ausschnitt erhält man etwa das folgende Drei-Adress-Programm:

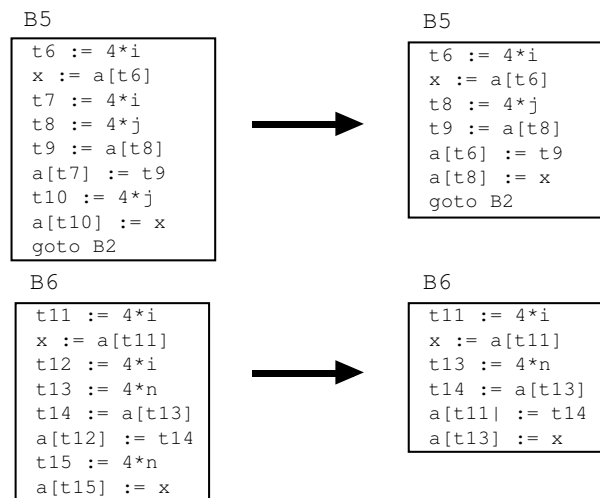
(1)     i := m-1	(16)    t7 := 4*i
(2)     j := n	(17)    t8 := 4*j
(3)     t1 := 4*n	(18)    t9 := a[t8]
(4)     v := a[t1]	(19)    a[t7] := t9
(5)     i := i+1	(20)    t10 := 4*j
(6)     t2 := 4*i	(21)    a[t10] := x
(7)     t3 := a[t2]	(22)    goto (5)
(8)     if t3 < v goto (5)	(23)    t11 := 4*i
(9)     j := j-1	(24)    x := a[t11]
(10)    t4 := 4*j	(25)    t12 := 4*i
(11)    t5 := a[t4]	(26)    t13 := 4*n
(12)    if t5 > v goto (9)	(27)    t14 := a[t13]
(13)    if i >= j goto (23)	(28)    a[t12] := t14
(14)    t6 := 4*i	(29)    t15 := 4*n
(15)    x := a[t6]	(30)    a[t15] := x

Der zugehörige Flussgraph wäre dann:



In B5 werden die Ausdrücke  $4*i$  und  $4*j$  zweimal berechnet, in B6 werden die Ausdrücke  $4*i$  und  $4*n$  ebenfalls zweimal berechnet. Wir haben also sogenannte **gemeinsame Teilausdrücke**, die natürlich nur einmal berechnet werden müssen. Da diese Optimierung sich nur auf jeweils einen einfachen Block bezieht, bezeichnet man diesen Vorgang auch als **lokale Entfernung gemeinsamer Teilausdrücke**.

Die beiden Blöcke werden also wie folgt umgeformt:



Nach der lokalen Entfernung gemeinsamer Teilausdrücke stellt man fest, dass die Berechnung von  $4*j$  in B5 überflüssig ist, da dieser Term bereits in B3 berechnet wurde und es keine andere Möglichkeit gibt, als über B3 nach B5 zu gelangen. Also kann man die drei Befehle  $t8 := 4*j$ ,  $t9 := a[t8]$  und  $a[t8] := x$  durch die beiden Befehle  $t9 := a[t4]$  und  $a[t4] := x$  ersetzen. Damit ergibt sich aber der gemeinsame Teilausdruck  $a[t4]$  in B3 und B5, also kann man  $t9 := a[t4]$  und  $a[t6] := t9$  ersetzen durch  $a[t6] := t5$ .

Der Block B5 enthält jetzt die Befehle:

```

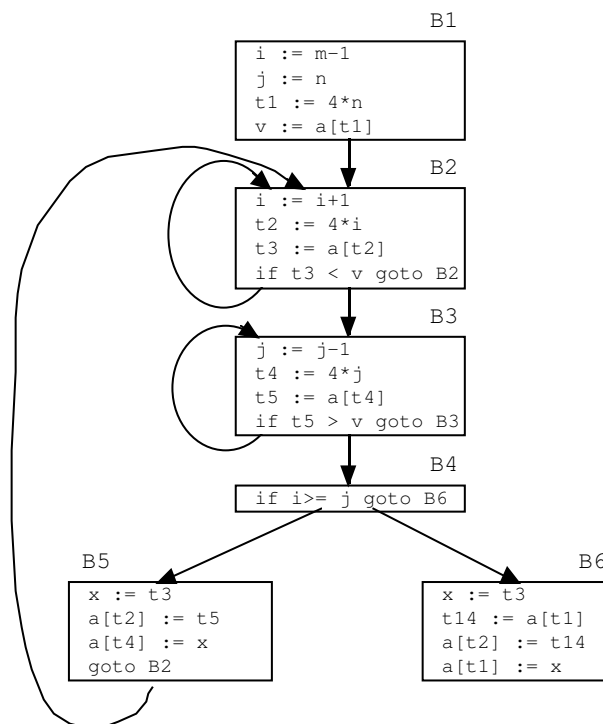
t6 := 4*i
x := a[t6]
a[t6] := t5
a[t4] := x
goto B2

```

Analog erkennt man jetzt den gemeinsamen Teilausdruck  $4*i$  in B5 und B2 und ersetzt die drei Befehle  $t6 := 4*i$ ,  $x := a[t6]$  und  $a[t6] := t5$  durch  $x := a[t2]$  und  $a[t2] := t5$ . Danach wird wiederum  $a[t2]$  als gemeinsamer Teilausdruck erkannt und  $x := a[t2]$  durch  $x := t3$  ersetzt.

Die gleichen Überlegungen können für den Block B6 angestellt werden und man erhält einen neuen Flussgraphen:





**Achtung:** Der Ausdruck `a[t1]` in B1 und B6 ist **kein** gemeinsamer Ausdruck! Auf dem Weg von B1 nach B6 kann der Block B5 durchlaufen werden, in dem eine Wertzuweisung an das Feld `a` steht. Sofern keine weiteren Informationen zur Verfügung stehen, muss angenommen werden, dass **jedes** Element von `a` dadurch verändert wurde, also auch `a[t1]`!

Man sieht an diesem Beispiel, dass es unbedingt notwendig ist, Informationen über den Gebrauch und die Definitionen von Variablen auch über Blockgrenzen hinweg zu sammeln und mögliche Wege des Ablaufs im Flussgraphen zu berücksichtigen. Diese Probleme werden in den nächsten Abschnitten behandelt.

Neben der Erkennung und Entfernung gemeinsamer Teilausdrücke gibt es noch eine ganze Reihe weiterer Möglichkeiten, den Zwischencode zu verbessern.

Durch die Entfernung gemeinsamer Teilausdrücke werden häufig Anweisungen der Form `x := y`, sogenannte „Kopien“, eingeführt. Ziel einer weiteren Programmtransformation ist es, nach einer derartigen Anweisung soweit wie möglich ein auftretendes `x` durch `y` zu ersetzen. Man nennt diese Transformation auch das „Weiterreichen von Kopien“ (Copy Propagation).

In unserem Beispiel kann man in B5 `a[t4] := x` durch `a[t4] := t3` und in B6 `a[t1] := x` durch `a[t1] := t3` ersetzen.

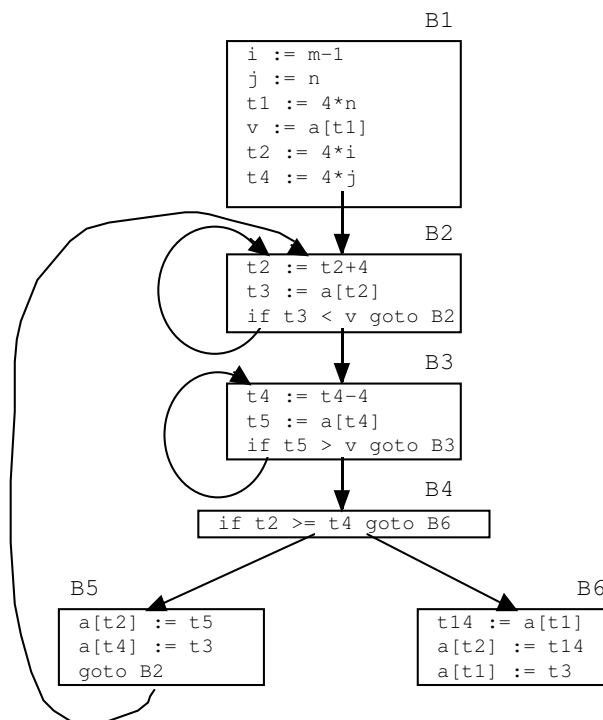
Eine weitere Möglichkeit zur Codeverbesserung besteht darin, Befehle mit konstanten Operanden bereits zu diesem Zeitpunkt auszuwerten und die Ergebnisse ähnlich wie oben weiterzureichen. Beide Transformationen hinterlassen meist überflüssige Befehle. Ein Befehl ist überflüssig, wenn der einfache Block, in dem er auftritt, nicht vom Startknoten erreichbar ist oder aber wenn der Befehl von der Form `a := ...` ist und `a` nach diesem Befehl nicht lebendig ist. Überflüssige Befehle können ohne weiteres entfernt werden.

Eine weitere wichtige Programmtransformation ist das Erkennen und Verschieben schleifeninvarianter Berechnungen an den Schleifeneintritt. Das erste Problem hierbei ist das Erkennen von Schleifen und deren Eintrittspunkt im Flussgraph. Als nächstes sind die Befehle im Rumpf der

Schleife zu identifizieren, die bei jedem Durchlauf durch die Schleife den selben Wert berechnen. Dann müssen diese Befehle in einen einfachen Block vor dem Schleifenanfang ausgelagert werden.

Ein anderer Ansatz zur Schleifenoptimierung führt über das Erkennen von sogenannten „Induktionsvariablen“, das sind Variable, die sich bei jedem Schleifendurchlauf um einen konstanten Wert verändern. Häufig kann man diese Induktionsvariablen bis auf eine einsparen und dabei eventuell sogar eine „teure“ Operation (etwa eine Multiplikation) durch eine „billige“ (etwa eine Addition) ersetzen.

In unserem Beispiel werden im Block B3 die Werte von  $j$  und  $t4$  synchron bei jedem Schleifendurchlauf verändert. Wird  $j$  um 1 vermindert, so wird  $t4$  um 4 vermindert. Entsprechendes gilt für die Werte von  $i$  und  $t2$  im Block B2. Letztlich ist es sogar möglich, im Block B4 die Abfrage `if i >= j goto B6` durch `if t2 >= t4 goto B6` zu ersetzen, da  $t2 = 4*i$  und  $t4 = 4*j$  an dieser Stelle gilt. Damit werden die Variablen  $i$  und  $j$  in den Blöcken B2 und B3 überflüssig und man erhält:



Es wären noch weitere Optimierungen in B1 möglich, jedoch ist es fraglich, ob sich der Aufwand lohnt, da B1 nur einmal durchlaufen wird.

## 6.4 Lokale Optimierungen

In diesem Abschnitt werden einfache Blöcke isoliert betrachtet. Der Drei-Adress-Code in den Blöcken wird optimiert, dabei werden gemeinsame Teilausdrücke im Block entfernt, zur Übersetzungszeit bereits mögliche Berechnungen und einfache algebraische Umformungen durchgeführt.

### 6.4.1 Konstruktion eines DAGs für einen einfachen Block

Ausgehend von einem einfachen Block wird ein gerichteter azyklischer Graph (*directed acyclic graph* - **DAG**) aufgebaut, der die Berechnungen im Block darstellt. Dieser DAG kann dann als Eingabe für eine Maschinencode- Erzeugung verwendet werden oder aber in einen Block zurücktransformiert werden. Bei der Konstruktion werden gemeinsame Teilausdrücke erkannt und berücksichtigt.

**Eingabe:** Ein einfacher Block

**Ausgabe:** Ein DAG für den einfachen Block.

Die Knoten des DAGs enthalten die folgende Information:

- interne Knoten enthalten einen Operator als Markierung
- Blätter enthalten einen Variablennamen oder eine Konstante als Markierung
- den Knoten ist eine Liste angehängt, in der Null oder mehr Variablennamen auftreten.

Jeder Knoten des DAGs korrespondiert mit einem berechneten Zwischenergebnis und die in der Liste aufgeführten Variablen „enthalten“ diesen Wert.

Einige Knoten sind als Ausgabeknoten markiert. Die Werte dieser Knoten werden an andere Stelle im Programm (nicht in diesem Block!) benötigt. Die Variablen, die diese Werte enthalten sind lebendig am Ausgang dieses einfachen Blocks. Welche Variablen lebendig sind, kann man mit einer Datenfluss-Analyse berechnen.

**Methode:** Man benötigt eine Tabelle, in der ein Verweis auf den jeweils „neuesten“ Knoten zu finden ist, der den Wert einer Variablen bzw. einer Konstanten repräsentiert.

`node(identifier)` gibt jeweils diesen Knoten zurück.

Durchlaufe den einfachen Block von Anfang bis Ende.

Für jeden Drei-Adress-Befehl der Form  $x := y \text{ op } z$  führe die folgenden Schritte aus:

- 1) Befindet sich  $y$  nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung  $y$  und vermerke dies in der Tabelle. (d.h. `node(y)` gibt diesen Knoten zurück!)  
Befindet sich  $z$  nicht in der Tabelle, so führe dieselben Schritte für  $z$  aus.
- 2) Bestimme, ob es im DAG einen Knoten  $\alpha$  gibt, der als linken Nachfolger `node(y)`, als rechten Nachfolger `node(z)` und als Markierung `op` hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten  $\alpha$  mit dieser Eigenschaft.
- 3) Ist die Variable  $x$  in der dem Knoten `node(x)` angehefteten Liste, so lösche  $x$  aus der Liste. Füge  $x$  in die Liste des Knotens  $\alpha$  ein und ändere die Tabelle so, dass `node(x)` den Wert  $\alpha$  hat.

Für jeden Drei-Adress-Befehl der Form  $x := op\ y$  führe die folgenden Schritte aus:

- 1) Befindet sich  $y$  nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung  $y$  und vermerke dies in der Tabelle.
- 2) Bestimme, ob es im DAG einen Knoten  $\alpha$  gibt, der als Nachfolger  $\mathbf{node}(y)$  und als Markierung  $op$  hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten  $\alpha$  mit dieser Eigenschaft.
- 3) Ist die Variable  $x$  in der dem Knoten  $\mathbf{node}(x)$  angehefteten Liste, so lösche  $x$  aus der Liste. Füge  $x$  in die Liste des Knotens  $\alpha$  ein und ändere die Tabelle so, dass  $\mathbf{node}(x)$  den Wert  $\alpha$  hat.

Für jeden Drei-Adress-Befehl der Form  $x := y$  führe die folgenden Schritte aus:

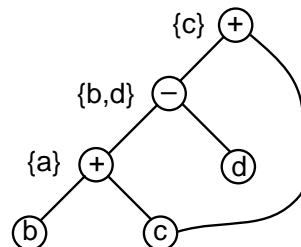
- 1) Befindet sich  $y$  nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung  $y$  und vermerke dies in der Tabelle.
- 2) Sei  $\alpha = \mathbf{node}(y)$ .
- 3) Ist die Variable  $x$  in der dem Knoten  $\mathbf{node}(x)$  angehefteten Liste, so lösche  $x$  aus der Liste. Füge  $x$  in die Liste des Knotens  $\alpha$  ein und ändere die Tabelle so, dass  $\mathbf{node}(x)$  den Wert  $\alpha$  hat.

### Beispiel 6.5:

Für den einfachen Block

```
a := b + c
b := a - d
c := b + c
d := a - d
```

erhielte man den DAG



Die Knotenmarkierungen sind in den Knoten, die Listen in den geschweiften Klammern daneben notiert.

Für die vierte Anweisung wird kein Knoten erzeugt, es wird nur die Ergebnisvariable  $d$  zu der Liste hinzugefügt. Diese Vorgehensweise entspricht dem Entfernen gemeinsamer Teilausdrücke. Ist die Variable  $b$  nicht lebendig am Ende des Blocks, könnte man den einfachen Block in

```
a := b + c
d := a - d
c := d + c
```

umschreiben. Sind  $b$  und  $d$  am Ende lebendig, müsste man noch den Befehl  $b := d$  anfügen.

In diese Konstruktion lassen sich leicht Optimierungen einbeziehen, die sich auf algebraische Identitäten beziehen. So wäre zum Beispiel bei der Konstruktion des DAGs bei kommutativen Operatoren die Reihenfolge der beiden Nachfolger irrelevant. Befehle der Art  $y := 0 + x$  lassen sich bei der Konstruktion leicht erkennen und die Identitäten können natürlich gleich berücksichtigt werden.

Außerdem kann man das Verfahren erweitern und in der Übersetzungsphase mögliche Berechnungen bereits hier durchführen.

So kann man etwa einen Befehl der Art  $y := 5 + 7$  durch  $y := 12$  ersetzen.

### 6.4.2 Darstellung von Feldzugriffen im DAG

Auf den ersten Blick sieht es so aus, als könne man Zugriffe auf Feldelemente wie die anderen Operatoren bearbeiten. Aber es gibt da ein Problem:

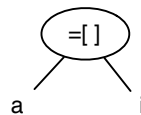
#### Beispiel 6.6:

Betrachte die Drei-Adress-Befehle

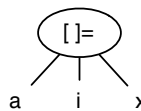
$$\begin{aligned} x &:= a[i] \\ a[j] &:= y \\ z &:= a[i] \end{aligned}$$

Interpretiert man den Feldzugriff  $a[i]$  als eine Operation mit Operanden  $a$  und  $i$ , dann wäre  $a[i]$  im Beispiel ein gemeinsamer Teilausdruck. Allerdings könnten  $i$  und  $j$  den gleichen Wert haben und damit repräsentieren die beiden Auftreten von  $a[i]$  eventuell unterschiedliche Werte!

Also muss man bei einer Zuweisung zu einem Feldelement eine zusätzliche Regel einführen. Man übersetzt einen Feldzugriff der Art  $x := a[i]$  in einen Knoten mit zwei Nachfolgern der Art

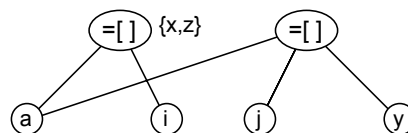


Eine Zuweisungen zu Feldelementen wie etwa  $a[i] := x$  wird durch einen Knoten mit einer dreifach-Verzweigung dargestellt:

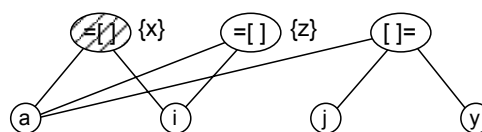


Beim Einsetzen eines Knotens dieser Art in den entstehenden DAG werden alle Knoten, deren Wert vom Knoten  $\text{node}(a)$  abhängen, eingefroren, da angenommen werden muss, dass *jedes* Element des Feldes verändert wurde. Eingefrorene Knoten werden in den Abbildungen schraffiert gezeichnet. Dies bedeutet, dass ihre Listen nicht mehr verändert werden dürfen!

Ohne diese Regel würde man aus dem Block aus Beispiel 6.6 den (falschen) DAG



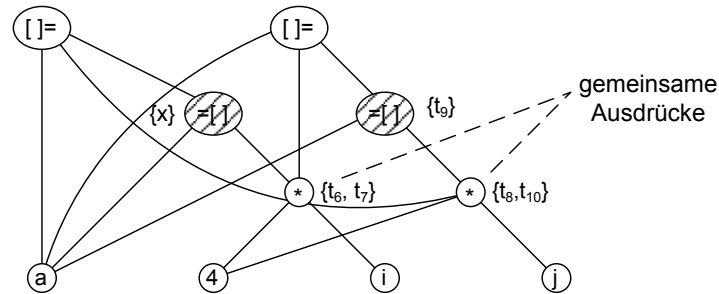
erhalten. Beachtet man jedoch die zusätzliche Regel, so erhält man:



Entsprechende Regeln müssen auch bei Zeiger-Operationen und bei Prozeduraufrufen beachtet werden. Im Zweifelsfall bleibt bei Sprachen, die etwa beliebige Zeigeroperationen ermöglichen nichts anderes über, als alle vorhandenen Knoten einzufrieren!

**Beispiel 6.7:**

Der ursprüngliche Block  $B_5$  im Quicksort-Beispiel 6.4 ergäbe den folgenden DAG:



Was kann man nun mit dem so gewonnenen DAG machen?

- 1) Erkennen, welche Variablen und Konstanten in einem einfachen Block benötigt werden (Blätter) und welchen Variablen neue Werte zugewiesen werden (use/def-Werte für die Datenfluss-Analyse).
- 2) Rückwandlung in einen einfachen Block, wobei gemeinsame Teilausdrücke entfernt werden.
- 3) Maschinencode-Erzeugung direkt vom DAG aus.

## 6.5 Datenfluss Analyse

Damit ein Programm sicher optimiert werden kann, müssen vor der Anwendung von Programmtransformationen Informationen über das gesamte Programm gesammelt werden. Dazu wird versucht, den „Zustand“ des Programms, d.h. die Werte der Variablen an einem Programm-Punkt zu bestimmen. Ein Programm-Punkt ist dabei eine Stelle im Drei-Adress-Programm vor oder nach einem Drei-Adress-Befehl. Jeder Drei-Adress-Befehl transformiert den Eingangszustand am Programm-Punkt *vor* dem Befehl in einen neuen Ausgangszustand am Programm-Punkt *nach* dem Befehl.

Häufig interessiert aber nur ein Teil der Variablen oder bestimmte Eigenschaften der Variablen, so dass man nur diese Datenfluss-Werte betrachtet und bestimmt. Man nennt die Menge der für eine Anwendung interessierenden Werte den *Datenfluss-Domain*.

So kann man zum Beispiel an einem Programm-Punkt die Frage stellen, an welchen Stellen der momentane Wert einer Variablen  $x$  im Programm definiert wurde. Dies ist das Problem der *reaching definitions*. Gibt es zum Beispiel an diesem Punkt nur genau eine Definition von  $x$  und wurde bei dieser Definition der Variablen  $x$  eine Konstante zugewiesen, so kann man diese Konstante statt der Variablen  $x$  verwenden und eventuell Berechnungen zur Übersetzungszeit statt zur Laufzeit durchführen (*constant propagation*).

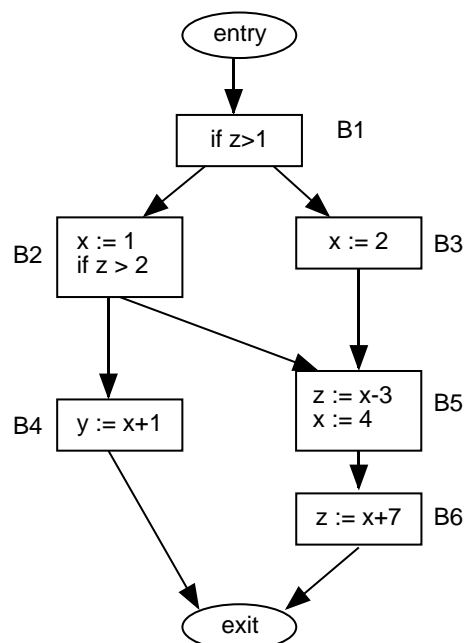
Allgemeiner spricht man in diesem Kontext von der *use-definition-Verkettung* (ud-Kette) und der *definition-use-Verkettung* (du-Kette) von Variablen.

**Definition:** Sei  $p$  eine Position im Drei-Adress-Code, an der eine Variable  $a$  definiert wird. Die **du-Kette** für  $p$  und  $a$  ist eine Liste von Positionen im Drei-Adress-Code, an denen die Variable  $a$  gebraucht wird und auf dem Weg dorthin der in  $p$  definierte Wert nicht überschrieben werden kann.

**Definition:** Sei  $p$  eine Position im Drei-Adress-Code, an der eine Variable  $a$  gebraucht wird. Die **ud-Kette** für  $p$  und  $a$  ist eine Liste von Positionen im Drei-Adress-Code, an denen die Variable  $a$  definiert wird und der zugewiesene Wert auf jedem Weg bis  $p$  nicht überschrieben werden kann.

### Beispiel 6.8:

Man betrachte den folgenden Flussgraphen:



Die du-Kette für die Definition von  $x$  im Block B2 enthält die beiden Positionen in den Blöcken B4 und B5, in denen  $x$  gebraucht wird. Der Gebrauch im Block B6 ist nicht in der du-Kette, da  $x$  in B5 neu definiert wird!

Die ud-Kette für den Gebrauch von  $x$  im Block B4 enthält nur die Definition im Block B2 während die ud-Kette für den Gebrauch von  $x$  in B5 die beiden Definitionen in B2 und B3 enthält.

Zur Lösung dieser Fragen und zur Bestimmung von ud- bzw. du-Ketten werden Datenfluss-Gleichungen aufgestellt, die geeignete Informationen am Anfang eines einfachen Blocks (**in**-Mengen) mit den entsprechenden Informationen am Ende des Blocks (**out**-Mengen) und dem Inhalt des Blocks in Beziehung setzen.

### 6.5.1 Transferfunktionen

Betrachten wir zunächst einen Drei-Adress Befehl  $s$ .  $\text{in}[s]$  bezeichnet die Datenfluss-Werte am Programm-Punkt vor  $s$ ,  $\text{out}[s]$  die Werte am Programm-Punkt nach  $s$ . Dem Drei-Adress-Befehl zugeordnet ist eine Transferfunktion  $f_s$ . Da die Datenfluss-Information entweder in Richtung der Programmausführung (Vorwärts) oder aber in entgegengesetzter Richtung (Rückwärts) fließen kann, gibt es auch zwei mögliche Transferfunktionen:

- 1) für die **Vorwärtsanalyse**:  $\text{out}[s] = f_s(\text{in}[s])$
- 2) für die **Rückwärtsanalyse**:  $\text{in}[s] = f_s(\text{out}[s])$ .

**Bemerkung:** Um die Notation nicht zu unübersichtlich werden zu lassen und da aus dem Kontext immer klar ist, ob es sich um eine Vorwärts- oder Rückwärtsanalyse handelt, werden die beiden Transferfunktionen gleich bezeichnet.

Innerhalb eines einfachen Blocks  $B$ , der aus den Befehlen  $s_1, s_2, \dots, s_k$  besteht, gilt immer  $\text{out}[s_i] = \text{in}[s_{i+1}]$  für  $1 \leq i < k$ . Es ist daher sinnvoll, die Transferfunktion auf den einfachen Block  $B$  auszuweiten. Auf diese Weise verringert man die Zahl der Datenfluss-Gleichungen, ohne allzu viel Information zu verschenken. Man definiert dazu  $\text{in}[B] = \text{in}[s_1]$  und  $\text{out}[B] = \text{out}[s_k]$ . Die Transferfunktion ist dann

- 1) für die Vorwärtsanalyse gilt  $\text{out}[B] = f_B(\text{in}[B])$
- 2) für die Rückwärtsanalyse gilt  $\text{in}[B] = f_B(\text{out}[B])$ .

wobei  $f_B = f_{s_k} \circ \dots \circ f_{s_2} \circ f_{s_1}$  (für die Vorwärtsanalyse) bzw.  $f_B = f_{s_1} \circ f_{s_2} \circ \dots \circ f_{s_k}$  (für die Rückwärtsanalyse) gilt.

Jetzt bleibt nur noch, die Datenfluss-Informationen im Fluss-Graphen zu betrachten. Auch hier muss man zwischen den beiden Möglichkeiten unterscheiden.

Bei der Vorwärtsanalyse gilt

$$\text{in}[B] = \cup_{P \in \text{pred}(B)} \text{out}[P]$$

bei der Rückwärtsanalyse gilt dagegen

$$\text{out}[B] = \cup_{S \in \text{succ}(B)} \text{in}[S]$$

wobei  $\text{pred}(B)$  die Vorgängerblöcke und  $\text{succ}(B)$  die Nachfolgerblöcke von  $B$  im Flussgraph bezeichnet. Man erhält so zwei Datenfluss-Gleichungen für jeden Block des Fluss-Graphen. Das so erhaltenen Gleichungssystem wird dann üblicherweise iterativ gelöst.



### 6.5.2 Verfügbare Definitionen (Reaching Definitions)

Dies ist eine Analyse, die für eine oder auch mehrere Variablen alle Positionen im Drei-Adress-Code bestimmt, an denen diese Variable definiert wird und diese Definitionen bis zu einer bestimmten Stelle (Blockanfang, Blockende) im Flussgraphen nicht überschrieben wurden, also noch verfügbar sind. Diese Analyse ist offensichtlich eine Vorwärtsanalyse.

Man benötigt diese Information für die ud-Ketten (**use-definition-chaining**), bei dem man für jeden Gebrauch einer Variablen wissen möchte, an welchen Stellen im Programm der momentane Wert dieser Variablen definiert worden sein könnte. Die Datenfluss-Domain ist in diesem Fall die Menge aller Positionen von Drei-Adress-Befehlen.

Man kann diese Information auch nutzen, um einen möglichen Gebrauch einer Variablen **a** vor einer Definition von **a** zu bestimmen. Man ordnet dazu jeder Variablen im **entry**-Block einen dummy-Wert zu. Erreicht eine derartige Definition für **a** einen Punkt im Flussgraphen, an dem **a** benutzt wird, gibt einen Weg im Flussgraphen von **entry** zu dieser Position, in dem der Variablen **a** kein Wert zugewiesen wird.

Zunächst soll die Transferfunktion für einen Drei-Adress-Befehl beschrieben werden. Man betrachte einen Befehl der Art **a := b op c** auf Position *p*. Dieser Befehl erzeugt (*generates*) eine Definition *p* der Variablen **a** und löscht (*kills*) alle anderen Definitionen von **a** im Programm. Sämtliche Definitionen von anderen Variablen werden nicht verändert.

Also könnte man die Transferfunktion für diesen Befehl in der Form  $f_p(x) = \text{gen}[p] \cup (x - \text{kill}[p])$  notieren, wobei  $\text{gen}[p] = \{p\}$  die von diesem Befehl erzeugte Definition und  $\text{kill}[p]$  die Menge aller *anderen* Definitionen von **a** im Programm ist, also  $\text{kill}[p] = \text{defs}(\mathbf{a}) - \{p\}$ , wobei  $\text{defs}(x)$  die Menge aller Positionen ist, an denen *x* definiert wird.

Bei einem Drei-Adress-Befehl der Form **a := b[c]** wird die Transferfunktion analog definiert. Für Befehle der Form **if a relop b** oder **a[b] := c** ist die Transferfunktion dagegen die Identität.

#### Beispiel 6.9:

Man betrachte folgendes Drei-Adress-Programm, wobei Gebrauch und Definition der beiden Variablen **a** und **c** interessieren:

```
(1)    a := 7
(2)    c := 2
(3) L:  if c > a goto L1
(4)    c := c + a
(5)    goto L
(6) L1: a := c - a
(7)    c := 0
```

Dann berechnen sich die **gen**- und **kill**-Mengen für die uns interessierenden Variablen **a** und **c** wie folgt:

s	gen[s]	kill[s]
(1)	1	6
(2)	2	4,7
(3)		
(4)	4	2,7
(5)		
(6)	6	1
(7)	7	2,4

Die Transferfunktion für einen einfachen Block kann man nun einfach durch Komposition der Transferfunktionen für die einzelnen Befehle bestimmen. Dabei gilt für zwei Funktionen  $f_1(x) = \text{gen}[1] \cup (x - \text{kill}[1])$  und  $f_2(x) = \text{gen}[2] \cup (x - \text{kill}[2])$ , dass die Komposition die Form

$$\begin{aligned} f_2(f_1(x)) &= \text{gen}[2] \cup (\text{gen}[1] \cup (x - \text{kill}[1]) - \text{kill}[2]) \\ &= (\text{gen}[2] \cup (\text{gen}[1] - \text{kill}[2])) \cup (x - (\text{kill}[1] \cup \text{kill}[2])) \\ &= \text{gen}[2] \cup (\text{gen}[1] - \text{kill}[2]) \cup (x - \text{kill}[1] - \text{kill}[2]) \end{aligned}$$

hat.

Also erhält man für einen einfachen Block  $B$  mit  $k$  Befehlen die Transferfunktion:

$$f_B(x) = \text{gen}[B] \cup (x - \text{kill}[B])$$

wobei

$$\text{kill}[B] = \text{kill}[1] \cup \text{kill}[2] \cup \dots \cup \text{kill}[k]$$

und

$$\begin{aligned} \text{gen}[B] &= \text{gen}[k] \cup (\text{gen}[k-1] - \text{kill}[k]) \cup (\text{gen}[k-2] - (\text{kill}[k-1] \cup \text{kill}[k])) \cup \\ &\quad \dots \cup (\text{gen}[1] - (\text{kill}[2] \cup \text{kill}[3] - \dots \cup \text{kill}[k])) \\ &= \text{gen}[k] \cup (\text{gen}[k-1] - \text{kill}[k]) \cup (\text{gen}[k-2] - \text{kill}[k-1] - \text{kill}[k]) \cup \\ &\quad \dots \cup (\text{gen}[1] - \text{kill}[2] - \text{kill}[3] - \dots - \text{kill}[k]) \end{aligned}$$

$\text{gen}[B]$  enthält also alle Definitionen im Block, die am Ende des Blocks noch gültig, d.h. nicht durch andere Definitionen überschrieben sind. Die Menge  $\text{kill}[B]$  ist einfach die Vereinigung der  $\text{kill}$ -Mengen aller in  $B$  vorhandenen Befehle.

### Algorithmus zur Bestimmung der $\text{gen}$ - und $\text{kill}$ -Menge eines Blocks

**Eingabe:** Ein einfacher Block  $B$

**Ausgabe:** Die  $\text{gen}$  und  $\text{kill}$ -Menge für den Block  $B$

**Verfahren :**

- 1) Setze  $\text{gen}[B] := \emptyset$  und  $\text{kill}[B] := \emptyset$ .
- 2) Durchlaufe die Instruktionen des Blocks  $B$  vom ersten bis zum letzten Befehl und führe für jeden Befehl  $s$  an Position  $p$ , der eine Variable definiert, die folgenden Schritte aus:
  - i)  $\text{gen}[B] := \text{gen}[s] \cup (\text{gen}[B] - \text{kill}[s])$
  - ii)  $\text{kill}[B] := \text{kill}[B] \cup \text{kill}[s]$

wobei  $\text{kill}[s]$  die Menge der Definitionen ist, die durch diesen Befehl überschrieben werden und  $\text{gen}[s] = \{p\}$  gilt, sofern auf Position  $p$  eine Variable definiert wird.

**Bemerkung:** Prozeduraufrufe und Zeigerzugriffe der Art  $*p := x$  müssen wegen ihrer eventuellen Seiteneffekte gesondert betrachtet werden!

#### Beispiel 6.10:

Für den einfachen Block  $B$ :

- (1)  $a := 3$
- (2)  $a := 4$

würde  $\text{gen}[B] = \{2\}$  und  $\{1, 2\} \subseteq \text{kill}[B]$  gelten.

Sei  $\text{pred}(B)$  die Menge der Vorgängerblöcke von  $B$  im Flussgraphen. Dann gelten für jeden Block  $B$  die Datenfluss-Gleichungen:

$$\begin{aligned}\text{out}[B] &= \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) \\ \text{in}[B] &= \bigcup_{B' \in \text{pred}(B)} \text{out}[B']\end{aligned}$$

Da sich der Wert von  $\text{in}[B]$  aus Werten der Vorgängerknoten berechnet, haben wir hier eine Vorwärtsanalyse.

Für den Eingangsblock „**entry**“ gilt offensichtlich die Randbedingung  $\text{out}[\text{entry}] = \emptyset$ .

Wie löst man nun so ein Gleichungssystem? Man setzt zunächst  $\text{out}[B] = \emptyset$  für alle Blöcke  $B$  und iteriert solange, bis es keine Änderungen an den Mengen mehr gibt! Man berechnet also einen kleinsten Fixpunkt dieses Gleichungssystems. Es lässt sich zeigen, dass die iterativ berechneten  $\text{in}$ - und  $\text{out}$ -Mengen monoton wachsen und das System somit einen Fixpunkt besitzt. Die Anzahl der Iterationen ist also kleiner als die Knotenzahl im Flussgraphen. Bei „günstiger“ Auswahl der Blöcke hat sich gezeigt, dass die Anzahl der Iterationen meist kleiner gleich 5 ist.

### Algorithmus zur Analyse der verfügbaren Definitionen

**Eingabe:** Ein Flussgraph, für dessen Blöcke die jeweiligen  $\text{gen}$ - und  $\text{kill}$ -Mengen bestimmt worden sind.

**Ausgabe:** Die  $\text{in}$  und  $\text{out}$ -Mengen für jeden Block im Flussgraphen.

**Verfahren :**

$\text{pred}(B)$  sei die Menge der Vorgängerblöcke von  $B$  im Flussgraphen.

- 1) Setze  $\text{out}[B] := \emptyset$ .
- 2) Solange sich eine der  $\text{out}$ -Mengen ändert, führe man die folgenden Schritte für jeden Block  $B$  aus:

- i)  $\text{in}[B] := \bigcup_{P \in \text{pred}(B)} \text{out}[P]$
- ii)  $\text{out}[B] := \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$

### Beispiel 6.11:

Betrachten wir zunächst das Beispiel 6.9 oben. In diesem Beispiel wird die Analyse nicht auf Blöcken, sondern auf einzelnen Befehlen durchgeführt. Man erhält somit:

s	in[s]	out[s]
(1)		1
(2)	1	1,2
(3)	1,2,4	1,2,4
(4)	1,2,4	1,4
(5)	1,4	1,4
(6)	1,2,4	2,4,6
(7)	2,4,6	6,7

Also könnte man z.B. das **a** im Befehl (3) und im Befehl (6) durch 7 ersetzen! (Copy Propagation)

**Beispiel 6.12:**

Betrachten wir den ursprünglichen, nicht optimierten Flussgraphen aus dem Quicksort-Beispiel auf Seite 83. Von Interesse seien nur die Definitionen der Variablen  $i$ ,  $j$ ,  $v$  und  $x$ .

Wie oben erwähnt sollen Definitionen durch Positionen des Drei-Adress-Befehls repräsentiert werden.

$i$  wird in Position 1 (B1) und 5 (B2) definiert,

$j$  wird in Position 2 (B1) und 9 (B3) definiert,

$v$  wird in Position 4 (B1) definiert,

$x$  wird in Position 15 (B5) und 24 (B6) definiert.

Damit ergeben sich die Mengen **gen** und **kill** für die einfachen Blöcke wie folgt:

	gen	kill
B1	1, 2, 4	5, 9
B2	5	1
B3	9	2
B4	–	–
B5	15	24
B6	24	15

Man setzt zunächst  $\text{in}[B] = \text{out}[B] = \emptyset$  für alle Blöcke  $B$  und iteriert solange, bis es keine Änderungen an den Mengen mehr gibt.

Nach einigen Iterationen bleiben die **in**- und **out**-Mengen konstant und man erhält das folgende Ergebnis:

	in	out
B1	–	1, 2, 4
B2	1, 2, 4, 5, 9, 15	2, 4, 5, 9, 15
B3	2, 4, 5, 9, 15	4, 5, 9, 15
B4	4, 5, 9, 15	4, 5, 9, 15
B5	4, 5, 9, 15	4, 5, 9, 15
B6	4, 5, 9, 15	4, 5, 9, 24

Man sieht zum Beispiel, dass am Anfang von Block B3 für  $i$  nur die Definition auf Position 5 aktuell ist, während für  $j$  sowohl die Definition in Position 2 als auch die in Position 9 aktuell sein kann.

**Bemerkung:** Bei einer realen Implementation dieses Algorithmus würde man mit entsprechenden Bit-Vektoren arbeiten und die Mengen-Operationen durch Boolesche Operationen ersetzen.

### 6.5.3 ud - Ketten

Zur Bestimmung von ud-Ketten benutzt man die Analyse der verfügbaren Definitionen aus dem vorigen Abschnitt. Es wird für eine Benutzung einer Variablen in einem Drei-Adress-Befehl eine Liste (**Kette**) von Positionen von Befehlen konstruiert, an denen diese Variable definiert wird und der dort definierte Wert unverändert bleibt.

#### Algorithmus zur Bestimmung der Verwendungs-Definitions-Ketten

**Eingabe:** Ein Flussgraph, für dessen Blöcke die jeweiligen *in*-Mengen der Analyse der verfügbaren Definitionen bestimmt worden sind.

**Ausgabe:** Für jede Verwendung einer Variablen eine Liste mit den Definitionen dieser Variablen, die den Verwendungspunkt erreichen

**Verfahren :**

Durchlaufe jeden Block *B* im Flussgraphen und führe für jede interessierende Verwendung einer Variablen *a* die folgenden Schritte aus.

- 1) Wenn vor der Verwendung von *a* keine Definition von *a* in *B* erfolgt, so ist die Definitionskette für diese Verwendung von *a* die Menge der Definitionen für *a*, die in *in*[*B*] enthalten sind.
- 2) Existieren Definitionen von *a* im Block *B*, die vor der Verwendung von *a* liegen, so wird nur die letzte dieser Definitionen in die Definitionskette aufgenommen (also *in*[*B*] wird nicht aufgenommen!)

### 6.5.4 Lebendigkeit von Variablen

Es soll für eine Position im Drei-Adress-Programm festgestellt werden, ob der aktuelle Wert einer Variable entlang eines Pfades durch den Flussgraphen nochmal genutzt wird. Ist dem so, wird die Variable als **lebendig** an dieser Stelle bezeichnet, im anderen Fall als **tot**. Bei der Generierung des Maschinencodes für ein Programm kann so z. B. entschieden werden, ob der berechnete Wert einer Variablen, der sich am Ende eines Blocks in einem Register befindet, abgespeichert werden muss, da er später noch gebraucht wird.

Bei der Registerzuordnung kann eine Situation auftreten, in der alle Register vergeben sind und noch ein Register benötigt wird. In diesem Fall sollten die Register, die Werte toter Variablen enthalten, zuerst wiederverwendet werden.

Die Datenfluss-Domain ist in diesem Fall die Menge der verwendeten Variablen.

**Bemerkung:** Man kann die folgenden Verfahren auch statt auf Blöcken auf einzelnen Befehlen arbeiten lassen. Dadurch erhöht sich die Anzahl der Datenfluß-Gleichungen, dagegen ist die Bestimmung der Transferfunktionen einfacher.

Zunächst wird wieder die Transferfunktion für einen Drei-Adress-Befehl *s* bestimmt. Man betrachte einen Befehl der Form *a* := *b* op *c*. Dieser Befehl benötigt die Variablen aus der Menge *use*[*s*] = {*b*, *c*} und definiert die Menge *def*[*s*] = {*a*}. Also sind am Programm-Punkt vor diesem Befehl die Variablen in *use*[*s*] sowie alle anderen Variablen lebendig, die am Programm-Punkt nach diesem Befehl lebendig sind abzüglich der Variablen in *def*[*s*]. Wir haben damit eine Rückwärtsanalyse vor uns. Die Transferfunktion wäre  $f_s(x) = \text{use}[s] \cup (x - \text{def}[s])$ .

Wie oben beschrieben kann man aus diesen Transferfunktionen die Transferfunktion für einen Block zusammensetzen. Man erhält für einen einfachen Block *B* mit *k* Befehlen die Transferfunktion:

$$f_B(x) = \text{use}[B] \cup (x - \text{def}[B])$$

wobei

$$\text{def}[B] = \text{def}[1] \cup \text{def}[2] \cup \dots \cup \text{def}[k]$$

und

$$\begin{aligned} \text{use}[B] = & \text{use}[1] \cup (\text{use}[2] - \text{def}[1]) \cup (\text{use}[3] - \text{def}[1] - \text{def}[2]) \cup \\ & \dots \cup (\text{use}[k] - \text{def}[1] - \text{def}[2] - \dots - \text{def}[k-1]) \end{aligned}$$

### Algorithmus zur Bestimmung der def- und use-Menge eines Blocks

**Eingabe:** Ein einfacher Block B

**Ausgabe:** Die def und use-Menge für den Block B.

**Verfahren :**

- 1) Setze  $\text{def}[B] := \emptyset$  und  $\text{use}[B] := \emptyset$ .
- 2) Durchlaufe die Drei-Adress-Befehle des Blocks B vom ersten bis zum letzten Befehl und führe für jeden Befehl  $s$  die folgenden Schritte nacheinander aus:
  - i)  $\text{use}[B] := \text{use}[B] \cup (\text{use}[s] - \text{def}[B])$
  - ii)  $\text{def}[B] := \text{def}[B] \cup \text{def}[s]$

Nachdem diese Mengen bestimmt worden sind, kann man nun die lebendigen Variablen am Eintritt in einen Block bzw. beim Austritt aus einem Block bestimmen. Sei  $\text{succ}(B)$  die Menge der Nachfolgerblöcke von B im Flussgraphen. Dann gelten für jeden Block B die Datenfluss-Gleichungen:

$$\begin{aligned} \text{in}[B] &= \text{use}[B] \cup (\text{out}[B] - \text{def}[B]) \\ \text{out}[B] &= \bigcup_{S \in \text{succ}(B)} \text{in}[S] \end{aligned}$$

Da sich der Wert von  $\text{out}[B]$  aus Werten der Nachfolgerknoten berechnet, haben wir hier eine Rückwärtsanalyse.

Für den Ausgangsblock „exit“ gilt offensichtlich die Randbedingung  $\text{in}[\text{exit}] = \emptyset$ .

Man setzt zunächst für jeden Block  $\text{in}[B] = \emptyset$ . Gelöst wird das Gleichungssystem wiederum mit fortgesetzten Iterationen bis sich die in-Mengen nicht mehr verändern.

### Algorithmus zur Analyse der lebendigen Variablen

**Eingabe:** Ein Flussgraph, für dessen Blöcke die jeweiligen def- und use-Mengen bestimmt worden sind.

**Ausgabe:** Die out und in-Mengen für jeden Block im Flussgraphen.

**Verfahren :**

$\text{succ}(B)$  sei die Menge der Nachfolgerblöcke von B im Flussgraphen.

- 1) Setze  $\text{in}[B] := \emptyset$ .
- 2) Solange sich eine der in-Mengen ändert, führe man die folgenden Schritte für jeden Block B aus:
  - i)  $\text{out}[B] := \bigcup_{S \in \text{succ}(B)} \text{in}[S]$
  - ii)  $\text{in}[B] := \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$

**Beispiel 6.13:**

Betrachtet man den optimierten Flussgraphen des Quicksort-Beispiels aus dem vorigem Abschnitt auf Seite 86, so erhält man:

	use	def
B1	m, n	i, j, t1, v, t2, t4
B2	t2, v	t3, t2
B3	t4, v	t5, t4
B4	t2, t4	-
B5	t2, t3, t4, t5	-
B6	t1, t2, t3	t14

Nach 5 Iterationen gibt es keine Änderungen an den *in*- und *out*-Mengen mehr (Die Anzahl der Iterationen hängt auch von der Reihenfolge ab, in der die Blöcke betrachtet werden!). Es ergibt sich:

	in	out
B1	m, n	t1, t2, t4, v
B2	t1, t2, t4, v	t1, t2, t3, t4, v
B3	t1, t2, t3, t4, v	t1, t2, t3, t4, t5, v
B4	t1, t2, t3, t4, t5, v	t1, t2, t3, t4, t5, v
B5	t1, t2, t3, t4, t5, v	t1, t2, t4, v
B6	t1, t2, t3	-

Man erkennt, dass zum Beispiel *i* und *j* nicht lebendig am Ende von Block B1 sind.

**Bemerkung:** Treten in einem Drei-Adress-Programm Prozeduraufrufe und Zeigerzugriffe auf, so muss man auch dafür *def*- und *use*-Mengen bestimmen. Ein konservativer Ansatz wäre es, anzunehmen, dass ein Prozeduraufruf keine Variablen definiert und dass alle nicht-temporären Variablen benutzt werden. Für Speicherzugriffe der Form *\*p := a* gilt, dass nur die Variablen *p* und *a* verwendet werden. Ein Zugriff der Form *a := \*p* definiert jedoch *a* und verwendet neben *p* auch jede Variable, auf die *p* verweisen könnte!

### 6.5.5 Analyse der verfügbaren Ausdrücke (Available Expressions)

Ein Ausdruck  $b \text{ op } c$  ist an einem Punkt  $p$  im Programm verfügbar, wenn auf jedem Weg von Eingangsknoten *entry* zum Punkt  $p$  dieser Ausdruck ausgewertet wird und danach auf dem Weg nach  $p$  keine Zuweisungen zu  $b$  oder  $c$  passieren. Diese Information ist wichtig, um gemeinsame Teilausdrücke über Blockgrenzen hinaus zu erkennen.

Sei  $U$  die Menge aller Ausdrücke, die in Drei-Adress-Befehlen des Programms auftreten.

Zunächst bestimmt man wie in den bisherigen Fällen die Transferfunktion für einen Drei-Adress-Befehl  $s$  der Form  $a := b \text{ op } c$ . Offensichtlich wird an dieser Stelle der Ausdruck  $b \text{ op } c$  erzeugt und jeder Ausdruck aus  $U$ , der  $a$  enthält, gelöscht. Für den Spezialfall, dass  $a = b$  oder  $a = c$  gilt, wird der Ausdruck  $b \text{ op } c$  also sofort wieder gelöscht!

Also hätten wir  $e\_gen[s] = \{b \text{ op } c\}$  und  $e\_kill[s] = \{\text{Ausdrücke aus } U, \text{ die } a \text{ enthalten}\}$  und die Transferfunktion wäre  $f_s(x) = (e\_gen[s] \cup x) - e\_kill[s]$ .

Analog folgt für andere Formen von Drei-Adress-Befehlen:

Befehlstyp $s$	$e\_gen[s]$	$e\_kill[s]$
$a := b+c$	$\{b+c\}$	alle Ausdrücke aus $U$ , die $a$ enthalten
$a := b[c]$	$\{b[c]\}$	alle Ausdrücke aus $U$ , die $a$ enthalten
if $a \text{ relOp } b$ goto ...	$\{\}$	$\{\}$
$a[b] := c$	$\{\}$	Ausdrücke aus $U$ der Form $a[x]$
usw.		

Wie üblich kann man jetzt aus den Transferfunktionen für die einzelnen Befehle  $s_1, \dots, s_k$  eines Blocks eine Transferfunktion für den gesamten Block konstruieren.

Man erhält

$$f_B(x) = e\_gen[B] \cup (x - e\_kill[B])$$

wobei

$$kill[B] = e\_kill[1] \cup e\_kill[2] \cup \dots \cup e\_kill[k]$$

und

$$\begin{aligned} e\_gen[B] = & (e\_gen[k] - e\_kill[k]) \cup (e\_gen[k-1] - e\_kill[k-1] - e\_kill[k]) \cup \\ & \dots \cup (e\_gen[1] - e\_kill[1] - e\_kill[2] - \dots - e\_kill[k]) \end{aligned}$$

#### Algorithmus zur Bestimmung der $e\_gen$ - und $e\_kill$ -Menge eines Blocks

**Eingabe:** Ein einfacher Block  $B$

**Ausgabe:** Die  $e\_gen$  und  $e\_kill$ -Menge für den Block  $B$

**Verfahren :**

- 1) Setze  $e\_gen[B] := \emptyset$  und  $e\_kill[B] := \emptyset$ .
- 2) Durchlaufe die Instruktionen des Blocks  $B$  vom ersten bis zum letzten Befehl und führe für jeden Befehl  $s$  die folgenden Schritte nacheinander aus:
  - i)  $e\_gen[B] := (e\_gen[B] \cup e\_gen[s]) - e\_kill[s]$
  - ii)  $e\_kill[B] := e\_kill[B] \cup e\_kill[s]$



**Beispiel 6.14:**

Man betrachte die den folgenden, aus 4 Befehlen bestehenden einfachen Block  $B$ . Es werden für jeden Programm-Punkt die verfügbaren Ausdrücke angegeben, wobei angenommen wird, dass vor Eintritt in diesen Block kein Ausdruck verfügbar ist.

Befehl	verfügbare Ausdrücke
	$\emptyset$
$a := b + c$	$\{b + c\}$
$b := a - d$	$\{a - d\}$
$c := b + c$	$\{a - d\}$
$d := a - d$	$\emptyset$

Es wäre also  $\mathbf{e\_gen}[B] = \emptyset$  und es gilt  $\{b + c, a - d\} \subseteq \mathbf{e\_kill}[B]$ .

Offensichtlich haben wir hier eine Vorwärtsanalyse. Es gelten die Datenfluss-Gleichungen:

$$\mathbf{out}[B] = \mathbf{e\_gen}[B] \cup (\mathbf{in}[B] - \mathbf{e\_kill}[B])$$

$$\mathbf{in}[B] = \bigcap_{P \in \text{pred}(B)} \mathbf{out}[P]$$

Für den Eingangsblock „entry“ gilt offensichtlich die Randbedingung  $\mathbf{out}[\text{entry}] = \emptyset$ .

**Bemerkung:** Man beachte, dass in der Bestimmungsgleichung für die  $\mathbf{in}$ -Menge ein Durchschnittsoperator statt eines Vereinigungsoperators auftritt. Dies ist korrekt, da ein Ausdruck nur dann am Anfang eines Blocks verfügbar ist, wenn er am Ende *aller* Vorgängerblöcke verfügbar ist.

Die Lösung wird wieder iterativ bestimmt. Beim Start der Iteration muss man jedoch etwas anders vorgehen: Man setzt die  $\mathbf{out}$ -Menge für den „entry“-Knoten auf die leere Menge und für alle anderen Knoten setzt man sie auf die Menge aller Ausdrücke  $U$ .

**Algorithmus zur Bestimmung verfügbarer Ausdrücke**

**Eingabe:** Ein Flussgraph, für dessen Blöcke die jeweiligen  $\mathbf{e\_gen}$ - und  $\mathbf{e\_kill}$ -Mengen bestimmt worden sind.

**Ausgabe:** Die  $\mathbf{out}$  und  $\mathbf{in}$ -Mengen für jeden Block im Flussgraphen.

**Verfahren :**

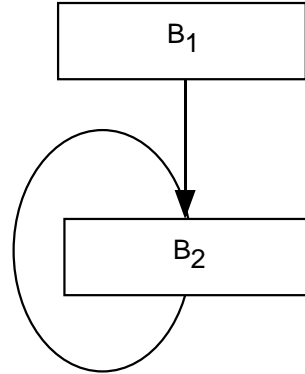
$\text{pred}(B)$  sei die Menge der Vorgängerblöcke von  $B$  im Flussgraphen.

- 1) Setze  $\mathbf{out}[B] := U$ ,  $\mathbf{out}[\text{entry}] = \emptyset$ .
- 2) Solange sich eine der  $\mathbf{out}$ -Mengen ändert, führe man die folgenden Schritte für jeden Block  $B$  aus:

- i)  $\mathbf{in}[B] := \bigcap_{P \in \text{pred}(B)} \mathbf{out}[P]$
- ii)  $\mathbf{out}[B] := \mathbf{e\_gen}[B] \cup (\mathbf{in}[B] - \mathbf{e\_kill}[B])$

**Beispiel 6.15:**

Sei folgendes Flussdiagramm gegeben:



Man betrachte den Block  $B_2$ . Man kann die zeitliche Entwicklung der **in**- und **out**-Werte durch die folgenden Rekursionen beschreiben, dabei seinen  $\text{out}[B_2]_j$  bzw.  $\text{in}[B_2]_j$  der Wert der **out**- bzw. **in**-Menge von  $B_2$  zum Zeitpunkt  $j$ . Man erhält:

$$\begin{aligned}\text{in}[B_2]_{j+1} &= \text{out}[B_1]_j \cap \text{out}[B_2]_j \\ \text{out}[B_2]_{j+1} &= \text{e\_gen}[B_2] \cup (\text{in}[B_2]_{j+1} - \text{e\_kill}[B_2])\end{aligned}$$

Würde man mit  $\text{out}[B_2]_0 = \emptyset$  starten, dann wäre  $\text{in}[B_2]_1 = \text{out}[B_1]_0 \cap \text{out}[B_2]_0 = \emptyset$ . Das bedeutet aber, dass ein in  $B_1$  generierter Ausdruck wie  $\mathbf{a} \text{ op } \mathbf{b}$ , der in  $B_2$  nicht auftritt und auch nicht in der Kill-Menge von  $B_2$  vorkommt, nie in der **out**-Menge von  $B_2$  auftaucht. Startet man dagegen mit  $\text{out}[B_2]_0 = U$ , erhält man korrekterweise  $\text{in}[B_2]_1 = \text{out}[B_1]_0 \cap \text{out}[B_2]_0 = \text{out}[B_1]$ .

Hier zum Abschluss eine Tabelle mit den wichtigsten Parametern für die Datenfluss-Analyse:

	verfügbare Definitionen	Lebendigkeit	verfügbare Ausdrücke
Domain	Menge von Definitionen	Menge von Variablen	Menge von Ausdrücken = $U$
Richtung	Vorwärts	Rückwärts	Vorwärts
Transfer-Fkt.	$\text{gen}[B] \cup (x - \text{kill}[B])$	$\text{use}[B] \cup (x - \text{def}[B])$	$\text{e\_gen}[B] \cup (x - \text{e\_kill}[B])$
Gleichungen	$\text{out}[B] = f_B(\text{in}[B])$ $\text{in}[B] = \bigcup_{P \in \text{pred}[B]} \text{out}[P]$	$\text{in}[B] = f_B(\text{out}[B])$ $\text{out}[B] = \bigcup_{S \in \text{succ}[B]} \text{in}[S]$	$\text{out}[B] = f_B(\text{in}[B])$ $\text{in}[B] = \bigcap_{P \in \text{pred}[B]} \text{out}[P]$
Initialisierung	$\text{out}[B] = \emptyset$	$\text{in}[B] = \emptyset$	$\text{out}[B] = U$
Randbedingung	$\text{out}[\text{entry}] = \emptyset$	$\text{in}[\text{exit}] = \emptyset$	$\text{out}[\text{entry}] = \emptyset$

## 6.6 Analyse der Ablaufstruktur

Eine wichtige Aufgabe im Rahmen der Codeoptimierung ist das Erkennen von Schleifen, damit spätere Optimierungen speziell auf die Schleifenrumpfe angesetzt werden können.

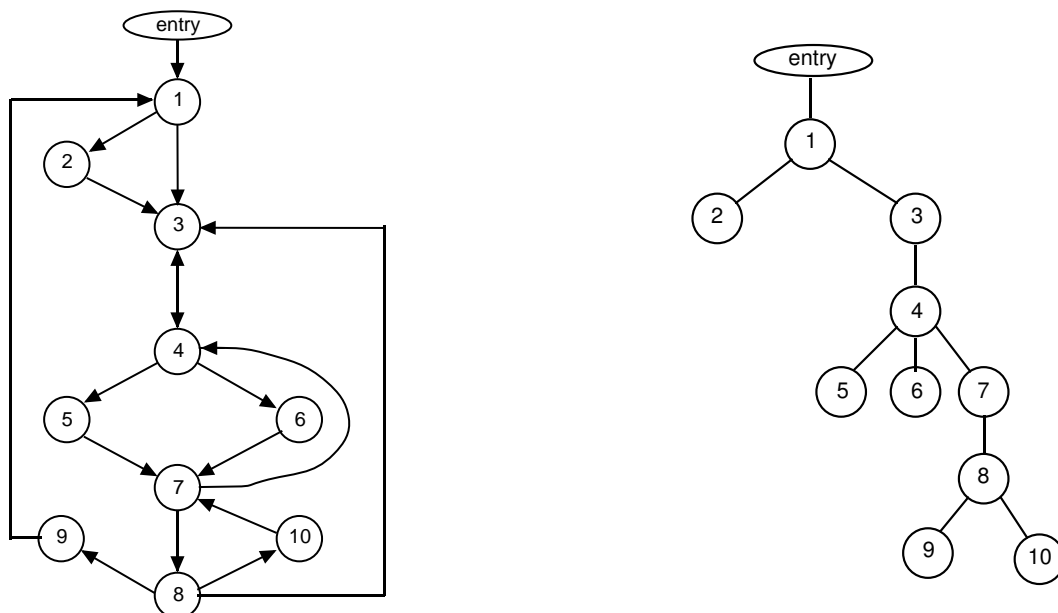
**Definition:** Gegeben sei ein erweiterter Flussgraph. Block  $B_a$  **dominiert** Block  $B$ , geschrieben  $B_a \succeq B$ , falls  $B_a$  auf jedem Weg vom Knoten **entry** nach  $B$  liegt. Die Relation  $\succeq$  ist offensichtlich eine Ordnung auf der Menge der Blöcke.

Block  $B_1$  ist **direkter Dominator-Block** vom Block  $B_2$ , geschrieben  $B_1 \succ B_2$ , falls für alle Blöcke  $B$  gilt: Aus  $B \succeq B_2$  und  $B \neq B_2$  folgt  $B \succeq B_1$

Jeder Block außer **entry** hat einen eindeutigen direkten Dominator-Block.

### Beispiel 6.16:

Gegeben sei der folgende erweiterte Flussgraph. Der rechts abgebildete Dominator-Baum gibt für jeden Block den direkten Dominator-Block als Vorgängerknoten an.



### 6.6.1 Algorithmus zur Berechnung der $\succeq$ -Relation

**Eingabe:** Ein erweiterter Flussgraph mit der Knotenmenge  $N$  und einem Startknoten **entry**  $\in N$ .

**Ausgabe:** Die  $\succeq$ -Relation. Für jeden Knoten  $B$  aus  $N$  gibt  $D(B)$  die Menge der Dominatoren von  $B$  an

**Verfahren:** Man löse die folgende Datenfluss-Analyse:

	Dominatoren
Domain	Potenzmenge von $N$
Richtung	Vorwärts
Transfer-Fkt.	$f_B(x) = x \cup \{B\}$
Gleichungen	$\text{out}[B] = f_B(\text{in}[B])$ $\text{in}[B] = \bigcap_{P \in \text{pred}[B]} \text{out}[P]$
Randbedingung	$\text{out}[\text{entry}] = \{\text{entry}\}$
Initialisierung	$\text{out}[B] = N$

Am Ende des Algorithmus gilt  $\text{out}[B] = D(B)$  für alle Blöcke  $B \in N$ . Es ist  $a \in D(b)$  genau dann wenn  $a \succeq b$ .

**Beispiel 6.17:**

Für den Flussgraphen aus Beispiel 6.16 ergibt sich im ersten Durchlauf, wenn man die Knoten in der Reihenfolge der Nummern betrachtet:

$$\begin{aligned}
 D(1) &:= \{1\} \\
 D(2) &:= \{2\} \cup D(1) = \{1, 2\} \\
 D(3) &:= \{3\} \cup (D(1) \cap D(2) \cap D(4) \cap D(8)) = \{1, 3\} \\
 D(4) &:= \{4\} \cup (D(3) \cap D(7)) = \{1, 3, 4\} \\
 D(5) &:= \{5\} \cup D(4) = \{1, 3, 4, 5\} \\
 D(6) &:= \{6\} \cup D(4) = \{1, 3, 4, 6\} \\
 D(7) &:= \{7\} \cup (D(5) \cap D(6) \cap D(10)) = \{1, 3, 4, 7\} \\
 D(8) &:= \{8\} \cup D(7) = \{1, 3, 4, 7, 8\} \\
 D(9) &:= \{9\} \cup D(8) = \{1, 3, 4, 7, 8, 9\} \\
 D(10) &:= \{10\} \cup D(8) = \{1, 3, 4, 7, 8, 10\}
 \end{aligned}$$

(Der Knoten *entry* wurde nicht aufgeführt, da er in allen  $D$ -Mengen enthalten ist.) Ein zweiter Durchlauf durch die Schleife bringt bei diesem Beispiel bereits keine weitere Veränderung.

### 6.6.2 Natürliche Schleifen

Mit dieser Information ist nun möglich, Schleifen im Programmablauf zu erkennen. Als erstes überlegt man sich, welche Eigenschaften eine Schleife charakterisiert:

- 1) Eine Schleife hat einen einzigen **Eintrittspunkt** oder **Kopf** (Header). Dieser Knoten dominiert alle anderen Knoten in der Schleife.
- 2) Es muss mindestens einen Knoten in der Schleife geben, vom dem aus eine Kante zum Eintrittspunkt zurückführt.

Also muss man Kanten  $a \rightarrow b$  im Flussgraphen suchen, für die  $b \succeq a$  gilt, d.h. der Zielknoten  $b$  dominiert den Ausgangsknoten  $a$ .

Die Kanten nennt man **Rückwärtskanten** (Back-Edges). In unserem Beispiel 6.16 wären dies die Kanten  $7 \rightarrow 4$ ,  $10 \rightarrow 7$ ,  $4 \rightarrow 3$ ,  $8 \rightarrow 3$  und  $9 \rightarrow 1$ .

**Definition:** Die **natürliche Schleife** einer Rückwärtskante  $n \rightarrow d$  mit  $d \neq n$  besteht aus  $d$  und den Knoten, über die man  $n$  erreichen kann ohne über  $d$  zu gehen. Ist  $n = d$ , so besteht die natürliche Schleife nur aus dem Block  $d$ .  $d$  ist der Eintrittspunkt der Schleife.

Natürliche Schleifen haben die Eigenschaft, dass zwei Schleifen entweder disjunkt sind oder, sofern beide Schleifen nicht den gleichen Kopf besitzen, die eine in der anderen enthalten ist. Für den Fall, dass zwei Schleifen den gleichen Kopf haben und keine in der anderen enthalten ist, scheint es sinnvoll zu sein, beide zusammen als eine Schleife aufzufassen.

Um die zu einer Rückwärtskante gehörende Schleife zu finden, kann man folgenden Algorithmus verwenden:

**Algorithmus zur Konstruktion einer natürlichen Schleife**

**Eingabe:** Ein Flussgraph und eine Rückwärtskante  $n \rightarrow d$

**Ausgabe:** Eine Menge *loop*, die alle Knoten enthält, die in der natürlichen Schleife von  $n \rightarrow d$  enthalten sind.

**Verfahren :**

```

stack := leer;
loop := {d};
if n = d return;
insert (n);
while stack  $\neq$  leer do begin
    m := pop(stack);
    for jedem Vorgänger p von m do
        insert(p);
    end;

procedure insert(m)
if m  $\notin$  loop then begin
    loop := loop  $\cup$  {m}
    push (m, stack)
end
end

```

**Beispiel 6.18:**

Man betrachte wieder den Flussgraphen aus Beispiel 6.16.

Zur Kante  $10 \rightarrow 7$  gehört die Schleife  $\{7, 8, 10\}$ .

Zur Kante  $7 \rightarrow 4$  gehört die Schleife  $\{4, 5, 6, 7, 8, 10\}$ .

Die Kanten  $4 \rightarrow 3$  und  $8 \rightarrow 3$  definieren die Schleife  $\{3, 4, 5, 6, 7, 8, 10\}$  und die Kante  $9 \rightarrow 1$  definiert eine Schleife, die alle Knoten von 1 bis 10 enthält.

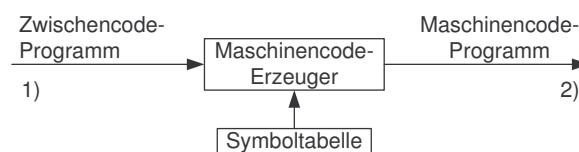
Die Knoten  $\{4, 5, 6, 7\}$  bilden keine Schleife, da in diese Knotenmenge sowohl über 4 als auch über 7 eingetreten werden kann und damit die Kopf-Bedingung verletzt ist.

## 7 Maschinencode-Erzeugung

Dieser Abschnitt beschäftigt sich mit der Maschinencode-Erzeugung aus einem vorher generierten und optimierten Zwischencode. Dieser Abschnitt ist natürlich in hohem Maße von der Architektur der Maschinenbefehle der Zielmaschine abhängig. Aus diesem Grund werden einige Verfahren auch nur ansatzweise erläutert, damit die prinzipielle Vorgehensweise nicht durch die Komplikationen der realen Maschinenstruktur überdeckt wird.

### 7.1 Einführung

Der prinzipielle Aufbau eines Maschinencode-Erzeugers wird durch folgende Zeichnung ersichtlich:



Zu 1) Übersetzung ist soweit durchgeführt, dass

- Werte in der Zwischensprache direkt auf der Zielmaschine dargestellt werden können (integer, real, bits, etc.)
- notwendige Typanpassungen eingefügt wurden.

Zu 2) Verschiedene Formen sind möglich:

- absolutes Maschinenprogramm (direkt ausführbar)
- Objektprogramm (als Eingabe für den Binder)
- Assemblerprogramm

Welche Probleme treten bei der Maschinencode Erzeugung auf?

- 1) Auswahl der Maschinenbefehle speziell bei Zielmaschinen mit einem Befehlscode, der viele Spezialfälle enthält.

#### Beispiel 7.1:

Bei den Motorola 68000-Prozessoren gibt es Varianten einfacher Befehle, bei denen der Operand im Maschinenbefehl kodiert werden kann, wenn er klein genug ist. Bei größeren Operanden muss der Befehl um ein sogenanntes Erweiterungswort verlängert werden:

Drei-Adress-Code		Assemblercode	Länge des Befehls
A := 4	→	MOVEQ #4,A	1 Wort
A := 100	→	MOVE #100,A	2 oder 3 Worte
A := A + 3	→	ADDQ #3,A	1 Wort
A := A + 10	→	ADDI #10,A	2 Worte

Die „richtige“ Antwort kann eigentlich nur getroffen werden, wenn möglichst viele Informationen über Zeit- und Platzbedarf einzelner Befehle vorhanden sind. Sicherlich ist die Übersetzung von Drei-Adress-Befehlen nicht kontextfrei – eine Übersetzung, die isoliert nur einen Befehl nach dem nächsten behandelt, ist nicht günstig.

Die Entscheidung, welche Werte über einen längeren Zeitraum in Registern gehalten werden sollen, kann nur nach genauer Datenfluß-Analyse getroffen werden.

Die Erzeugung von schnellen und platzsparenden Maschinencode macht bei RISC-Maschinen zwar weniger Probleme bei der Auswahl der Befehle, dafür ist die Reihenfolge der Maschinenbefehle wichtig (Pipelining).

## 2) Registerzuordnung

Befehle, deren Operanden in Hardwareregistern vorliegen, sind meist kürzer und schneller abzuarbeiten als äquivalente Befehle, deren Operanden im Speicher abgelegt sind. Daher ist es wichtig, häufig benutzte Variablen bzw. Werte in Registern zwischenspeichern. Erschwerend kommt hinzu, dass manche Befehle nur für spezielle Register oder auch Registerpaare erlaubt sind, z.B. bei der PDP 11 (einer 16-Bit-Maschine):

MUL A,R0	→	Das Ergebnis der Multiplikation der beiden Zahlen in Speicherzelle A und im Register R0 wird in R0 und R1 abgelegt.
DIV A,R0	→	R0 und R1 werden als 32 Bit Zahl interpretiert und durch die Zahl in Speicherzelle A dividiert, R0 enthält danach den Quotienten, R1 den Rest.

Leider sind die meisten in diesem Zusammenhang auftretenden Probleme NP-vollständig, d.h. es ist nicht zu erwarten, dass effiziente Algorithmen existieren, die optimale Registerzuordnungen oder optimale Maschinencodes liefern.

## 7.2 Die Zielmaschine

Im folgenden soll die grundlegende Vorgehensweise bei der Maschinencode-Erzeugung anhand einer einfachen, hypothetischen Maschine aufgezeigt werden. Unser hypothetischer Beispielrechner besitzt (im Gegensatz zur Realität) nur wenige Befehlstypen mit relativ klarer Semantik. Es sollen die folgenden Annahmen gelten:

- Die Hardware ist byte-orientiert,
- Die Maschine besitzt Drei-Adress-Befehle
- Es gibt  $n$  allgemeine Register R0, R1, ..., Rn
- Es gibt Lade-Operationen der Form LD Ri,src mit der Bedeutung, dass der Wert in Adresse src in das Zielregister Ri geladen wird. LD R0,A speichert also den Inhalt der Speicherzelle A im das Register R0.
- Es gibt Speicher-Operationen der Form ST dst,Ri mit der Bedeutung, dass der Wert im Register Ri in die Speicherzelle dst gespeichert wird.
- Es gibt übliche Operationen der Form OP Ri, src1, src2, wobei OP eine „übliche“ Operation wie MOV, ADD, SUB, ... ist und src1 und src2 (nicht notwendigerweise unterschiedliche) Speicherzellen sind. Bei einstelligen Operationen wie etwa NEG, INC, ... entfällt src2. Der Befehl SUB R0, R1, A subtrahiert zum Beispiel vom Wert im Register R1 den Wert in der Speicherzelle A und speichert das Ergebnis im Register R0. INC R1 erhöht den Inhalt des Registers R1.
- Es gibt unbedingte Sprünge der Form BR L, bei denen der Rechner zum Befehl mit Label L verzweigt.

- Es gibt bedingte Sprünge der Form  $Bcond\ Ri, L$ , dabei steht *cond* für eine der „üblichen“ Bedingungen, die an den Wert im Register  $Ri$  gestellt werden.  $BLTZ\ R1, L1$  verzweigt nur dann zum Label  $L1$ , wenn der Inhalt im Register  $R1$  kleiner oder gleich 0 ist.
- Der Rechner hat eine Vielzahl von Adressierungsarten,
  - Ein Variablenname bezeichnet den Inhalt der Speicheradresse für diese Variable. Ist die Variable  $a$  zum Beispiel ab Adresse 1000 im Speicher abgelegt, so bezeichnet diese Adressierung den Inhalt der Speicherzelle 1000, geschrieben  $contents(1000)$ .
  - Ein Konstrukt der Form  $num(Ri)$ , wobei  $num$  eine Integer-Zahl ist, bezeichnet den Inhalt einer Speicherzelle, deren Adresse man durch Addition der Zahl auf den Inhalt des Registers  $Ri$  erhält, also  $contents(num + contents(Ri))$ . Manchmal schreibt man dies auch in der Form  $a(Ri)$ , wobei der Variablenname  $a$  dann für die Adresse steht, ab der  $a$  gespeichert ist.
  - Ein Konstrukt der Form  $*Ri$  steht für eine indirekte Adressierung und bezeichnet den Inhalt einer Speicherzelle, deren Adresse in einer Speicherzelle steht, deren Adresse wiederum in Register  $Ri$  steht. Also bezeichnet  $*Ri$  den Wert  $contents(contents(contents(Ri)))$
  - Ein Konstrukt der Form  $*num(Ri)$  steht für eine indirekte Adressierung und bezeichnet den Inhalt einer Speicherzelle, deren Adresse in einer Speicherzelle steht, deren Adresse sich durch Addition der Zahl  $num$  auf den Wert im Register  $Ri$  ergibt. Wir haben also  $contents(contents(num + contents(Ri)))$
  - Ein Konstrukt der Form  $\#num$  beschreibt eine Konstante. Der Befehl  $LD\ R1, \#100$  lädt zum Beispiel die Konstante 100 in das Register  $R1$ .

Als Näherung für den Zeitbedarf zur Abarbeitung eines Befehls werden die Kosten eines Befehls definiert.

Die Kosten eines Befehls seien definiert als die Länge des Befehls (1 plus der Anzahl der zusätzlichen Speicherzugriffe zum Zugriff auf Konstanten, die nach dem Befehl gespeichert werden) plus der Zahl der Speicherzugriffe auf Operanden der Operation. Die Kosten entsprechen also der Gesamtzahl der Speicherzugriffe.

### Beispiel 7.2:

Befehl	Länge	Kosten
$LD\ R0, R1$	1	1
$LD\ R0, M$	2	3
$LD\ R0, \#100$	2	2
$LD\ R0, 100(R1)$	2	3
$LD\ R0, *R1$	1	2
$LD\ R0, *100(R1)$	2	4
$ADD\ R1, *50(R0), *(R2)$	2	6
$BLTZ\ R1, M$	2	2



**Beispiel 7.3:**

Beispielsweise sollen hier mögliche Übersetzungen von  $a := b + c$  betrachtet werden.

Annahme: Register R1 und R2 sind frei:

- |               |                    |
|---------------|--------------------|
| 1) LD R1,b    | Länge 7, Kosten 10 |
| LD R2,c       |                    |
| ADD R1,R1,R2  |                    |
| ST a,R1       |                    |
| 2) ADD R1,b,c | Länge 5, Kosten 8  |
| ST a,R1       |                    |

Wenn R3 und R4 die Adressen von b und c enthalten, dann:

- |                   |                   |
|-------------------|-------------------|
| 3) ADD R1,*R3,*R4 | Länge 3, Kosten 6 |
| ST a,R1           |                   |

Wenn R1 und R2 die Werte von b und c enthalten und der Wert von b ab dieser Stelle nicht weiter benötigt wird:

- |                 |                   |
|-----------------|-------------------|
| 4) ADD R1,R1,R2 | Länge 3, Kosten 4 |
| ST a,R1         |                   |

Man sieht hieran, dass eine der wesentlichen Aufgaben der Maschinencode-Erzeugung die Registerzuordnung ist! Außerdem erkennt man, wie in der vorangehenden Phase gewonnene Informationen zur Maschinencode-Erzeugung benutzt werden können und dass man Informationen über den momentan Speicherort der Werte von Variablen, in diesem Fall die Tatsache, dass sich im Register R1 der Wert von a befindet, für die Übersetzung des nächsten Drei-Adress-Befehls benutzen kann.

Betrachten wir mögliche Übersetzungen weiterer Drei-Adress-Befehle:

Annahme: a ist ein Feld von Objekten, die jeweils 8 Bytes belegen. Die untere Indexgrenze sei 0. Dann kann man den Befehl  $b = a[i]$  übersetzen in:

- |              |                    |
|--------------|--------------------|
| 5) LD R1,i   | Länge 8, Kosten 11 |
| MUL R1,R1,#8 |                    |
| LD R2,a(R1)  |                    |
| ST b,R2      |                    |

Einen Befehl der Form  $a[i] = b$  kann man übersetzen in:

- |              |                    |
|--------------|--------------------|
| 6) LD R1,b   | Länge 8, Kosten 11 |
| LD R2,i      |                    |
| MUL R2,R2,#8 |                    |
| ST a(R2),R1  |                    |

Einen Befehl der Form  $\text{if } x < y \text{ goto } L$  übersetzt man in:

- |              |                   |
|--------------|-------------------|
| 7) LD R1,x   | Länge 7, Kosten 9 |
| LD R2,y      |                   |
| SUB R1,R1,R2 |                   |
| BLTZ R1,M    |                   |

wobei M die dem Label L zugeordnete Speicheradresse ist. Sind die Variablen x und y nach diesem Befehl tot, wäre es günstiger, die ersten drei Befehle durch  $\text{SUB R1,x,y}$  zu ersetzen!

### 7.3 Ein einfacher Maschinencode-Erzeuger

Im Folgenden soll ein Maschinencode-Erzeuger vorgestellt werden, der Code für einen einfachen Block erzeugt. Es wird angenommen, dass bis auf eventuell speziell benutzte Register (für den Frame-Pointer, Stack-Pointer usw.) für jeden einfachen Block die gleiche Zahl von freien Registern verfügbar ist.

**Prinzip:** Man belässt berechnete Werte so lange wie möglich in Registern und speichert sie erst zurück, wenn

- das Register für andere Werte benötigt wird
- das Ende des einfachen Blocks erreicht wird.

#### Beispiel 7.4:

Um möglichst kurzen Code für den Befehl  $a := b + c$  zu erzeugen, muss man wissen, wo sich die momentanen Werte von  $b$  und  $c$  befinden (im Register, im Speicher, etc.), und man muss wissen, ob  $b$  oder  $c$  nach diesem Befehl im einfachen Block noch gebraucht werden oder ob sie am Ende des einfachen Blocks lebendig sind.

Gilt

- $b$  befindet sich in  $R_i$ ,  $b$  ist nicht lebendig,
- $c$  befindet sich in  $R_j$ ,  $c$  wird danach noch gebraucht:

dann erzeuge `ADD  $R_i, R_i, R_j$`  und notiere, dass  $a$  in  $R_i$  ist. (Länge 1, Kosten 1)

Gilt

- $c$  ist im Speicher,
- $R_j$  ist frei,
- sonst wie oben:

dann erzeuge `LD  $R_j, c$`  und notiere, dass  $a$  in  $R_i$  und  $c$  in  $R_j$  ist. (Länge 3, Kosten 4)  
`ADD  $R_i, R_i, R_j$`

Gilt

- $b$  ist im Speicher,
- $R_i$  ist frei,
- sonst wie oben:

dann erzeuge `ADD  $R_i, b, R_j$`  und notiere, dass  $a$  in  $R_i$  ist. (Länge 3, Kosten 4).

usw.

Also benötigt man zunächst einmal Informationen über den **nächsten Gebrauch** einer Variablen an einer Stelle im einfachen Block. Will man sich nicht auf einfache Blöcke beschränken, muss man Informationen aus der globalen Optimierung, in diesem Fall Informationen über die Lebendigkeit von Variablen, nutzen.

Für die Maschinencode-Erzeugung interessiert bei einem Befehl

$$x := y + z$$

der nächste Gebrauch von  $x$ ,  $y$  und  $z$  (im einfachen Block).

Zu diesem Zweck geht man zunächst an das Ende eines einfachen Blockes. Man benötigt eigentlich jetzt die Information, welche Variablen an dieser Stelle lebendig sind (d.h. ein Gebrauch in einem anderen oder, bei Schleifen möglich, auch im selben Block haben). Entweder muss man das mit Datenfluss-Analyse bestimmen, oder aber die (konservative) Annahme treffen, dass alle nicht-temporären Namen am Ende des Blocks lebendig sind. Alle diese Variablen haben einen „unbestimmten nächsten Gebrauch“.

Diese Information wird für jede Variable in die Symboltabelle eingetragen. Dann geht man rückwärts im einfachen Block voran.

Trifft man auf einen Befehl

$$(i) \ x := y \text{ op } z$$

dann sind die folgenden Schritte auszuführen:

- 1) füge an den Befehl auf Platz (i) die Information aus der Symboltabelle über den nächsten Gebrauch von  $x$ ,  $y$  und  $z$  an. Hat  $x$  keinen nächsten Gebrauch, d.h. ist  $x$  nicht lebendig, so kann dieser Befehl ersatzlos gestrichen werden!
- 2) Setze in der Symboltabelle  $x$  auf „keinen nächsten Gebrauch“.
- 3) Setze in der Symboltabelle den nächsten Gebrauch von  $y$  und  $z$  auf (i).

Dabei ist die Reihenfolge wichtig, man betrachte z.B. den Befehl  $x := x * x$ .

#### Beispiel 7.5:

Betrachtet werden soll der Drei-Adress-Code für das Programmfragment

```
a := (-c)*b;
b := b*c + a;
```

Am Blockende seien  $a$  und  $b$  lebendig, während  $c$  und die temporären Variablen  $t_1, \dots, t_4$  nicht lebendig sind.

An jeden Befehl wird jetzt die Information über den nächsten Gebrauch der im Befehl auftretenden Variablen angefügt. Dabei stehe

- (i) für einen nächsten Gebrauch beim Befehl (i),
- ? für unbestimmten nächsten Gebrauch und
- für keinen nächsten Gebrauch (nicht lebendig).

An die Drei-Adress-Befehle des Programms werden die folgenden Informationen angehängt:

Befehl	nächster Gebrauch		
(1) $t_1 := -c$	(2)	(4)	
(2) $t_2 := t_1 * b$	(3)	-	(4)
(3) $a := t_2$	(5)	-	
(4) $t_3 := b * c$	(5)	-	-
(5) $t_4 := t_3 + a$	(6)	-	?
(6) $b := t_4$	?	-	

Beim Befehl auf Platz (4) steht zum Beispiel, dass  $t_3$  nach diesem Befehl lebendig ist und einen nächsten Gebrauch im Befehl (5) hat, während  $b$  und  $c$  an dieser Stelle nicht lebendig sind.

Die Symboltabelle entwickelt sich dabei wie folgt:

Variable		nächster Gebrauch vor Befehl					
		(6)	(5)	(4)	(3)	(2)	(1)
$t_1$	-	-	-	-	-	(2)	-
$t_2$	-	-	-	-	(3)	-	-
$t_3$	-	-	(5)	-	-	-	-
$t_4$	-	(6)	-	-	-	-	-
$a$	?	?	(5)	(5)	-	-	-
$b$	?	-	-	(4)	(4)	(2)	(2)
$c$	-	-	-	(4)	(4)	(4)	(1)

Für die Maschinencode-Erzeugung benötigt man weiterhin zwei Tabellen, in denen festgehalten wird, welche Variablenwerte sich momentan in den Registern befinden und wo sich die Variablenwerte zur Zeit befinden:

1) Die Registertabelle:

Zu Beginn eines einfachen Blocks soll angenommen werden, dass jedes der verfügbaren Register leer ist. Später ist in dieser Tabelle verzeichnet, welche Variablenwerte sich in einem Register befinden.

Register	Variable
R0	$a, b$
R1	-
R2	$t_1, d$
$\vdots$	$\vdots$

2) Die Adresstabelle:

Diese Tabelle gibt an, an welchen Stellen sich der Wert einer Variablen zur Zeit befindet. Diese Tabelle könnte etwa in die Symboltabelle integriert werden.

Variable	Wert befindet sich zur Zeit in
$a$	R0, Speicher (Stack)
$b$	R0
$d$	R3, Speicher (statisch)
$\vdots$	$\vdots$

### 7.3.1 Maschinencode-Erzeugung für einfache Blöcke

**Eingabe:** Ein einfacher Block mit Informationen über den nächsten Gebrauch von Variablen

**Ausgabe:** Übersetzung des einfachen Blocks in Maschinencode

**Methode:**

- Durchlaufe den einfachen Block von Anfang bis Ende.
- Für jeden Drei-Adress-Befehl der Form  $x := y \text{ op } z$  führe die folgenden Schritte aus:

- 1) Bestimme mit Hilfe der Adresstabelle, wo sich momentan der Wert von  $y$  befindet. Ist der Wert in einem Register  $R_i$ , dann setze  $y' := R_i$ . Befindet sich  $y$  nicht in einem Register und hat  $y$  einen nächsten Gebrauch im einfachen Block und ist ein Register  $R_j$  frei, so erzeuge den Befehl `LD  $R, y$`  und setze  $y' := R_j$ . Andernfalls setze  $y' := y$ . Aktualisiere die Adress- und Registertabelle.
  - 2) wie unter 1) bestimme mit Hilfe der Adresstabelle einen Platz  $z'$ , wo sich momentan der Wert von  $z$  befindet.
  - 3) Bestimme mit Hilfe der Funktion `getreg` ein Register  $R$ , in dem das Ergebnis der Berechnung  $y \text{ op } z$  abgelegt werden soll. Erzeuge den Befehl `OP  $R, y', z'$` . Vermerke in den Tabellen, dass der Wert von  $x$  nur in  $R$  steht.
  - 4) Hat  $y$  und/oder  $z$  keinen nächsten Gebrauch im einfachen Block und ist nicht lebendig am Ende des Blocks und befindet sich in einem Register, dann lösche  $y$  und/oder  $z$  aus dem entsprechenden Eintrag der Registertabelle und setze die Adresstabelle entsprechend um.
- Für jeden Drei-Adress-Befehl der Form  $x := \text{op } y$  führe die folgenden Schritte aus:
    - 1) Bestimme mit Hilfe der Adresstabelle, wo sich momentan der Wert von  $y$  befindet. Ist der Wert in einem Register  $R_i$ , dann setze  $y' := R_i$ . Befindet sich  $y$  nicht in einem Register und hat  $y$  einen nächsten Gebrauch im einfachen Block und ist ein Register  $R_j$  frei, so erzeuge den Befehl `LD  $R_j, y$`  und setze  $y' := R_j$ . Andernfalls setze  $y' := y$ . Aktualisiere die Adress- und Registertabelle.
    - 2) Bestimme mit Hilfe eine der Funktion `getreg` ein Register  $R$ , in dem das Ergebnis der Berechnung  $\text{op } y$  abgelegt werden soll. Erzeuge den Befehl `OP  $R, y'$` . Vermerke in den Tabellen, dass der Wert von  $x$  nur in  $R$  steht.
    - 3) Hat  $y$  keinen nächsten Gebrauch im einfachen Block, ist nicht lebendig am Ende des Blocks und befindet sich in einem Register, dann lösche  $y$  aus dem entsprechenden Eintrag der Registertabelle und setze die Adresstabelle entsprechend um.
  - Für jeden Drei-Adress-Befehl der Form  $x := y$  führe die folgenden Schritte aus:
    - 1) Ist der Wert von  $y$  im Register, ändere die Eintragungen in der Register- und der Adresstabelle so, dass der Wert von  $x$  nur im Register auftritt, das  $y$  enthält. Hat  $y$  keinen nächsten Gebrauch und ist nicht lebendig am Ende des Blocks, lösche  $y$  aus der Registertabelle.
    - 2) Ist der Wert von  $y$  im Speicher, dann bestimme mit `getreg` ein Register  $R$  und erzeuge den Befehl `LD  $R, y$` . Danach müssen die beiden Tabellen wieder aktualisiert werden.
  - Am Ende des einfachen Blocks müssen für die Werte alle Variablen, die an dieser Stelle lebendig sind und sich noch nicht im Speicher befinden, `ST`-Befehle erzeugt werden, um die entsprechenden Werte zurück zu speichern.

### 7.3.2 Mögliche Implementation der Funktion `getreg`

**Eingabe:** Ein Drei-Adress-Befehl  $x := y \text{ op } z$  oder  $x := \text{op } y$  oder  $x := y$  mit angehefteten Informationen über den nächsten Gebrauch der Variablen.

**Ausgabe:** Ein Register  $R$ , in dem das Ergebnis der Operation abgelegt wird.

**Methode:**

- 1) Ist  $y$  oder  $z$  in einem Register, das nicht auch den Wert anderer Variablen enthält und hat  $y$  oder  $z$  keinen nächsten Gebrauch nach diesem Befehl, dann wähle dieses Register für  $R$ .
- 2) Trifft 1) nicht zu, wähle ein freies Register für  $R$ .
- 3) Trifft 2) nicht zu, hat  $x$  aber einen nächsten Gebrauch in diesem Block, dann suche ein besetztes Register  $R$ .  
 (Hier gibt es viele Möglichkeiten, mit Hilfe der Informationen über den nächsten Gebrauch von Variablen, der Registertabelle und der Speichertabelle ein Register zu suchen, das mit minimalem Aufwand „geleert“ werden kann.)  
 Erzeuge für jede Variable, deren Wert sich momentan in  $R$  und nicht im Speicher befindet, einen **ST**-Befehl, der diesen Wert in den Speicher befördert.

Diese Funktion könnte noch signifikant verbessert werden, wenn man z.B. die mögliche Kommutativität der Operation  $op$  bei der Wahl eines Registers ausnutzt .

### Beispiel 7.6:

Die Übersetzung von

```
a := (-c) * b
b := b * c + a
```

ergibt folgenden Drei-Adress Code (siehe Beispiel 7.5)

```
t1 := -c
t2 := t1 * b
a := t2
t3 := b * c
t4 := t3 + a
b := t4
```

Dabei seien  $t_1, \dots, t_4$  temporäre Dateien,  $a$  und  $b$  am Blockende lebendig und  $c$  nicht lebendig. Damit gelten die Informationen über den nächsten Gebrauch aus Beispiel 7.5.

Weiter sei angenommen, dass drei Register  $R0$ ,  $R1$  und  $R2$  frei sind. Zu Beginn befinden sich  $a$ ,  $b$  und  $c$  im Speicher,  $t_1, \dots, t_4$  befinden sich *nicht* im Speicher.

Unter diesen Voraussetzungen ergibt sich folgende Übersetzung in Maschinencode:

Befehl	Maschinencode	Registertabelle	Adresstabelle
-	-	Alle Register sind frei	s.o.
$t_1 := -c$	LD R0,c NEG R1,R0	R0 : c R1 : $t_1$	c in R0 und im Speicher $t_1$ in R1
$t_2 := t_1 * b$	LD R2,b MUL R1,R1,R2	R2 : b R1 : $t_2$	b in R2 und im Speicher $t_2$ in R1 $t_1$ -
$a := t_2$		R1 : a	a in R1, nicht im Speicher $t_2$ -
$t_3 := b * c$	MUL R2,R2,R0	R2 : $t_3$	$t_3$ in R2 b im Speicher
$t_4 := t_3 + a$	ADD R2,R2,R1	R2 : $t_4$	$t_4$ in R2 $t_3$ -
$b := t_4$		R2 : b	b in R2, nicht im Speicher
	ST a,R1 ST b,R2		a in R1 und im Speicher b in R2 und im Speicher

Damit erhält man ein Programm mit Kosten 16.

Unser Algorithmus zur Maschinencode-Erzeugung ist natürlich noch unvollständig, da nicht alle Typen von Drei-Adress-Befehlen behandelt werden. Um einen Eindruck von der zunehmenden Komplexität zu geben, soll nun beispielhaft die Behandlung eines Drei-Adress-Befehls der Form  $a := b[c]$  gezeigt werden.

Als Ziel sei ein Register  $R_j$  gewählt worden. Wieder gibt es eine große Fallunterscheidung:

- Ist  $c$  in  $R_i$ ,  $b$  eine Konstante, dann erzeuge
 

LD  $R_j, b(R_i)$       mit Kosten 3,
- ist  $c$  im Speicher, dann erzeuge
 

LD  $R_j, c$   
LD  $R_j, b(R_j)$       mit Kosten 6,
- ist  $c$  im Speicher,  
 $c$  wird im selben Block noch gebraucht und es sind noch  
 ein weiteres Register  $R_1$  frei, dann erzeuge
 

LD  $R_1, c$   
LD  $R_j, b(R_j)$       mit Kosten 6,
- ist  $c$  im Aktivierungs-Record (AR), FP der Frame-Pointer, dann erzeuge
 

LD  $R_j, c(FP)$   
LD  $R_j, b(R_j)$       mit Kosten 6,
- $c$  im AR und ist auch das Array  $b$  im AR und  
 ist  $\#b_{\text{offset}}$  der Abstand des Arrays  $b$  vom FP, dann erzeuge
 

ADD  $R_j, FP, \#b_{\text{offset}}$   
ADD  $R_j, c(FP), R_j$       mit Kosten 7  
LD  $R_j, *R_j$
- usw.

**Fazit:** Dieser Ansatz der Maschinencode-Erzeugung funktioniert zwar, wird aber bei der großen Zahl von Maschinenbefehlen und Adressierungsarten moderner Rechner zu einem Programmteil führen, das äußerst komplex und undurchsichtig ist! Die relevante Information ist gegen alle Regeln des Software Engineering zerstreut und teilweise in der Programmlogik eingearbeitet. Das führt zu einem fehleranfälligen, schwer zu modifizierenden und schwierig zu übertragenden Programmcode!



## 7.4 Maschinencode-Erzeugung für Syntax-Bäume

In diesem Abschnitt wird ausgehend von baumartigen DAGs oder Syntaxbäumen Maschinencode erzeugt. Ershov [28] und Sethi und Ullmann [37] zeigen, dass dieses Verfahren unter gewissen Annahmen an die Maschinenstruktur der Zielmaschine sogar den kürzesten Code liefert. Man beachte dabei, dass die Forderung nach baumartigen DAGs, das Erkennen gemeinsamer Teilausdrücke und damit die nur einmalige Auswertung des gemeinsamen Teilausdrucks in einer arithmetischen Anweisung verhindert!

Wir wollen im folgenden annehmen, dass nur 4 Typen von Maschinenbefehlen erlaubt sind:

- LD *reg, mem*
- ST *mem, reg*
- OP *reg, reg, reg*
- OP *reg, reg, mem*

Ziel des Verfahrens ist es, Maschinencode zu erzeugen, der bei einer vorgegebenen Zahl von verfügbaren Registern den Baum mit einer minimalen Zahl von Zwischenspeicherungen temporärer Ergebnisse auswertet. Die Idee besteht darin, zunächst den Baum einmal zu durchlaufen und Informationen über die Anzahl von Registern zu sammeln, die notwendig sind, um einen Teilbaum *ohne Zwischenspeicherungen* in den Speicher auszuwerten. In einer zweiten Phase werden die so gewonnenen Informationen genutzt, um bei einem Knoten zu entscheiden, ob zuerst der linke oder der rechte Teilbaum ausgewertet werden soll.

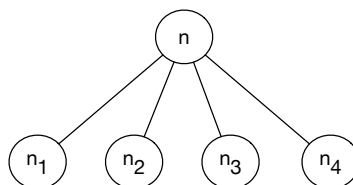
**Bem.:** Modifikationen dieses Verfahrens werden, teilweise mit viel komplexeren Bewertungsfunktionen, die dann mit Methoden der dynamischen Programmierung berechnet werden müssen, in existierenden Compilern angewendet (siehe auch Abschnitt 7.5)

Der Algorithmus läuft in zwei Phasen ab:

- 1) **Markierungsphase:** Für jeden Knoten  $n$  wird die minimale Anzahl  $\text{label}(n)$  von Registern (**Ershov-Zahlen**) bestimmt, die notwendig ist, um den entsprechenden Teilbaum ohne Speichern von Zwischenergebnissen auszuwerten. Diese Phase verläuft bottom-up.
- 2) **Code-Erzeugungsphase:** Jetzt werden die in 1) gefunden Werte benutzt, um einen möglichst effizienten Maschinencode zu erzeugen. Diese Phase verläuft top-down.

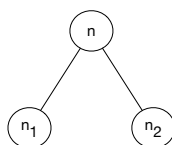
### 7.4.1 Die Markierungsphase

Betrachtet man z.B. einen Baum mit Wurzel  $n$  und Nachfolgerknoten  $n_1$  bis  $n_4$  und seien  $r_1$  bis  $r_4$  die jeweils minimal benötigten Register um den Teilbaum mit Wurzel  $n_1$  bis  $n_4$  auszuwerten.



Wenn man die Teilbäume von links nach rechts auswertet, dann benötigt man für den Teilbaum mit Wurzel  $n_1$  genau  $r_1$  Register. Da keine Zwischenspeicherungen erlaubt sind, muss dieses Ergebnis in einem Register belassen werden. Wird jetzt der Teilbaum mit Wurzel  $n_2$  ausgewertet, so benötigt man  $r_2 + 1$  Register. Dieses Zwischenergebnis muss wiederum in einem Register belassen werden, also benötigt man für den nächsten Teilbaum  $r_3 + 2$  Register usw. Also benötigt man für den gesamten Baum  $\max\{r_1, r_2 + 1, r_3 + 2, r_4 + 3\}$  Register! Da die Reihenfolge der Auswertung der einzelnen Teilbäume aber frei wählbar ist, kann man diesen Wert möglichst klein halten.

Für den Spezialfall binärer Operatoren ergibt sich folgendes Bild



und damit die folgende Formel zur Berechnung der label-Funktion:

$$\text{label}(n) = \begin{cases} \max(\text{label}(n_1), \text{label}(n_2)) & \text{falls } \text{label}(n_1) \neq \text{label}(n_2) \\ \text{label}(n_1) + 1 & \text{falls } \text{label}(n_1) = \text{label}(n_2) \end{cases}$$

Bei unären Operatoren erhält man  $\text{label}(n) = \text{label}(n_1)$ .

Für die Blätter des Baumes gilt:

$$\text{label}(n) = \begin{cases} 1 & \text{falls } n \text{ linkstes Blatt des Vorgängerknotens ist} \\ 0 & \text{sonst} \end{cases}$$

Diese Überlegungen führen zu folgendem Markierungsalgorithmus:

**Eingabe:** Ein Baum (Syntaxbaum)

**Ausgabe:** Derselbe Baum mit einer Markierung **label** an jedem Knoten. Die Markierung gibt die minimale Zahl von Registern an, die zur Auswertung des betreffenden Teilbaums ohne Zwischenspeicherung benötigt werden.

**Verfahren :**

Sei  $n$  der aktuelle Knoten.

- Ist  $n$  ein Blatt und linker Nachfolger des Vorgängerknotens, setze  $\text{label}(n) = 1$ .
- Ist  $n$  ein Blatt, aber nicht linker Nachfolger, setze  $\text{label}(n) = 0$ .
- Ist  $n$  ein interner Knoten und seien  $n_1, \dots, n_k$  Nachfolgerknoten von  $n$ , wobei die Reihenfolge so gewählt sei, dass

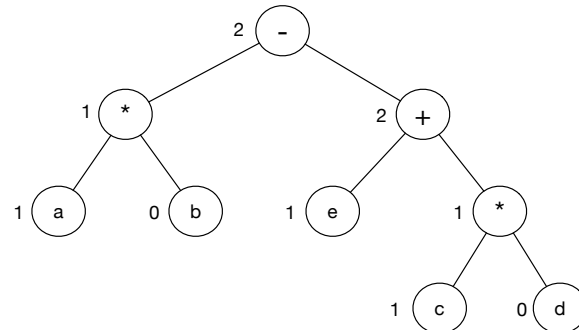
$$\text{label}(n_1) \geq \text{label}(n_2) \geq \dots \geq \text{label}(n_k)$$

gilt.

Dann setze  $\text{label}(n) = \max\{\text{label}(n_i) + i - 1 \mid 1 \leq i \leq k\}$

**Beispiel 7.7:**

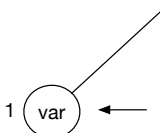
Man betrachte den arithmetischen Ausdruck  $a*b - e + c*d$ . Der zugehörige DAG ist der folgende Baum, wobei die Zahlen an den Knoten den Wert der label-Funktion angeben:

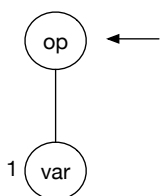
**7.4.2 Die Code-Erzeugungsphase**

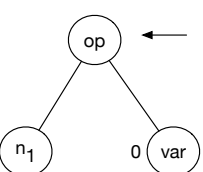
Der Baum wird jetzt mit einer rekursiven Methode **gencode** top-down durchlaufen. Der Algorithmus verwaltet einen Stack, in dem die Namen  $R_0, \dots, R_{r-1}$  der insgesamt  $r$  freien Register gespeichert sind. Bei einem Aufruf von **gencode(n)** erzeugt die Methode Maschinencode zur Auswertung des Teilbaums mit Wurzel  $n$ , wobei nur Register benutzt werden, deren Namen sich momentan im Stack befinden. Der Wert des Teilausdrucks befindet sich später im Register, das oben im Stack steht. Beim Verlassen der Methode befindet sich der Stack wieder im gleichen Zustand wie beim Eintritt. Mit der Methode **swap** werden die beiden obersten Einträge im Stack vertauscht.

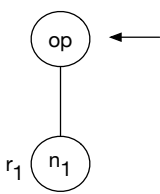
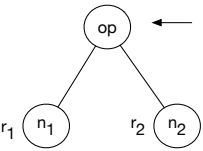
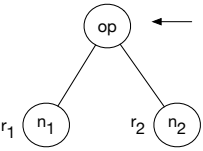
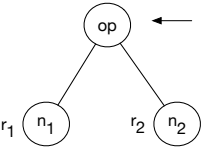
Im Algorithmus sind die folgenden Fälle zu unterscheiden:

(hier sei angenommen, dass der oberste Stackeintrag  $R_i$  ist und darunter  $R_j$  liegt. Der Pfeil zeigt immer auf den aktuellen Knoten.)

①  Das Blatt ist linker Nachfolger eines Knoten mit  $> 1$  Nachfolgern  
 $\rightarrow \text{LD } R_i, \text{var}$

②  Das Blatt ist einziger Nachfolger des Vorgängerknotens, dann  
 $\rightarrow \text{op } R_i, \text{var}$

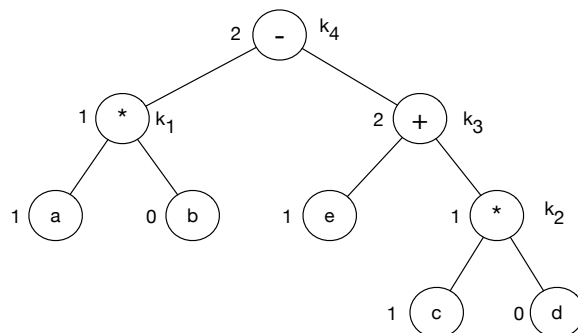
③   $n_1$  auswerten, das Ergebnis steht in  $R_i$ , dann  
 $\rightarrow \text{op } R_i, R_i, \text{var}$

- ④
- 
- $r_1 > 1$
- $n_1$  auswerten, das Ergebnis steht in  $R_i$ , dann  
 $\rightarrow \text{op } R_i, R_i$
- 
- ⑤
- 
- $1 \leq r_1 < r_2$   
 $r_1 < r$
- swap** ausführen,  $R_j$  ist oben auf dem Stack  
 $n_2$  auswerten  $\rightarrow$  Ergebnis steht in  $R_j$   
 $R_j$  aus Stack entfernen, dann  
 $n_1$  auswerten  $\rightarrow$  Ergebnis steht in  $R_i$   
 $\rightarrow \text{op } R_i, R_i, R_j$   
 $R_j$  auf den Stack und **swap** ausführen
- 
- ⑥
- 
- $r_1 \geq r_2 \geq 1$   
 $r_2 < r$
- $n_1$  auswerten  $\rightarrow$  Ergebnis steht in  $R_i$   
 $R_i$  aus dem Stack entfernen, dann  
 $n_2$  auswerten  $\rightarrow$  Ergebnis steht in  $R_j$   
 $\rightarrow \text{op } R_i, R_i, R_j$   
 $R_i$  auf den Stack legen.
- 
- ⑦
- 
- $r_1 \geq r$   
 $r_2 \geq r$
- $n_2$  auswerten  $\rightarrow$  Ergebnis steht in  $R_i$   
 $\rightarrow \text{ST Temp}, R_i$   
 $n_1$  auswerten  $\rightarrow$  Ergebnis steht in  $R_i$   
 $\rightarrow \text{op } R_i, R_i, \text{Temp}$

**Beispiel 7.8:**

Es soll der Syntaxbaum aus Beispiel 7.7 in Maschinencode übersetzt werden, wobei angenommen wird, dass die beiden Register R0 und R1 frei sind. Es ist also  $r = 2$

Zur besseren Übersicht sind hier die Knoten zusätzlich nummeriert

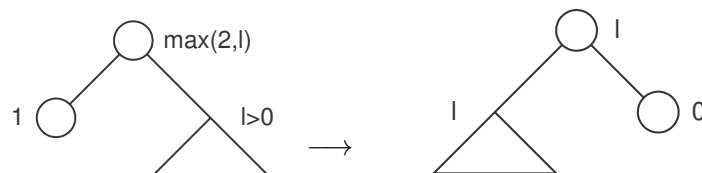


In der ersten Spalte stehen die Aufrufe der **gencode**-Methode, in der zweiten der momentane Stackinhalt (oberstes Stackelement rechts), in der dritten Spalte ist der angewendete Fall notiert und in der letzten Spalte steht der erzeugte Maschinencode.

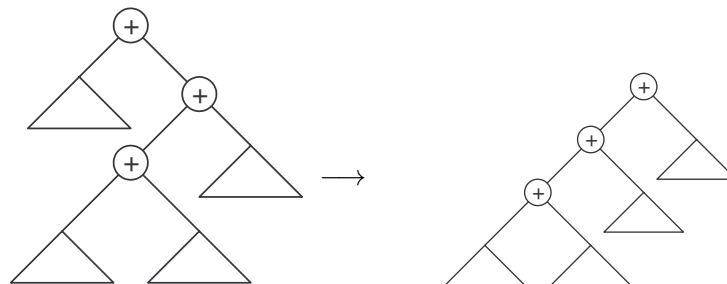
Aufrufe	Registerstack	Fall	Maschinencode
<b>gencode</b> (k <sub>4</sub> )	R1 R0	5	
<b>gencode</b> (k <sub>3</sub> )	R0 R1	6	
<b>gencode</b> (e)	R0 R1	1	
		1	LD R1,e
<b>gencode</b> (k <sub>2</sub> )	R0	3	
<b>gencode</b> (c)	R0	1	
		1	LD R0,c
		3	MUL R0,R0,d
	R0 R1	6	ADD R1,R1,R0
<b>gencode</b> (k <sub>1</sub> )	R0	3	
<b>gencode</b> (a)	R0	1	
		1	LD R0,a
		3	MUL R0,R0,b
		5	SUB R0,R0,R1
	R1 R0		

Man kann zeigen [37], dass der Algorithmus **gencode** unter den gemachten Voraussetzungen (keine algebraischen Umformungen, keine gemeinsamen Teilausdrücke, betrachtete Maschinenbefehle) optimalen Code, d.h. Code mit der kürzesten Befehlssequenz, erzeugt.

Viele Variationen dieses Verfahrens sind denkbar. So kann man z.B. algebraische Umformungen direkt auf den Baum anwenden, um eventuell die Zahl der benötigten Register zu verringern, etwa:



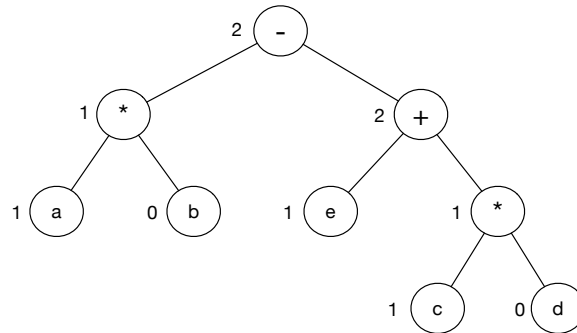
So wird bei kommutativen Operatoren ein LD- Befehl gespart. Bei assoziativen Operatoren kann man den Baum ebenfalls umordnen, um die Zahl der benötigten Register eventuell zu verringern:



Das Verfahren liefert natürlich nur für die dort eingeführte Maschine optimalen Code. Erweitert man z.B. die Maschinensprache, dann erhält man mit diesem Verfahren unter Umständen nicht mehr optimale Maschinencode-Sequenzen.

**Beispiel 7.9:**

Unser Beispiel  $a*b - e + c*d$  mit dem Baum



aus Beispiel 7.7 ergibt unter der Annahme, das nur ein Register R0 frei ist den folgenden Maschinencode mit Kosten = 27:

```

LD  R0,c
MUL R0,R0,d
ST  Temp,R0
LD  R0,e
ADD R0,R0,Temp
ST  Temp,R0
LD  R0,a
MUL R0,R0,b
SUB R0,R0,Temp

```

Hat die Maschine andererseits Befehle, bei denen nicht mindestens einer der beiden Operanden in einem Register liegen muss, wäre auch das folgende Programm mit Kosten = 21 möglich:

```

MUL R0,c,d
ADD R0,e,R0
ST  Temp,R0
MUL R0,a,b
Sub R0,R0,Temp

```

Im allgemeinen Fall wird man speziell bei *heterogenen* Befehlssätzen einen anderen Ansatz probieren müssen. Außerdem müssen natürlich auch die Kosten der evtl. vielen anwendbaren Befehle in Betracht gezogen werden. Eine Möglichkeit, unter diesen Nebenbedingungen optimalen oder fast optimalen Code zu erzeugen, besteht in der Anwendung des Prinzips der dynamischen Programmierung (siehe [20], verwendet in einigen Compilern, etwa in Johnsons PCC 2)

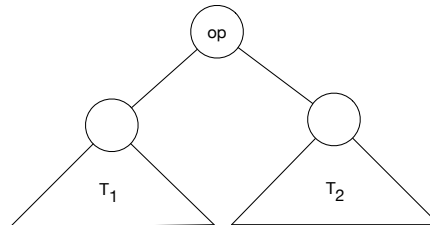
## 7.5 Code-Erzeugung mit dynamischer Programmierung

In diesem Abschnitt wird eine Verallgemeinerung des Verfahrens aus Abschnitt 7.4 vorgestellt, die unter gewissen Annahmen optimalen Code liefert, jedoch auch für komplexere und umfangreichere Sätze von Maschinenbefehlen gute Übersetzungen liefert. Das Verfahren beruht auf dem Prinzip der dynamischen Programmierung und ist in der Lage, auch die Kosten der einzelnen Maschinenbefehle zu berücksichtigen.

Gegeben ist wieder ein baumartiger DAG bzw. ein Syntaxbaum  $T$  und eine Anzahl  $r$  von frei verfügbaren Registern.

Bei der dynamischen Programmierung wird das Problem der Erzeugung optimalen Codes für einen arithmetischen Ausdruck  $E$  zerlegt in das Problem, optimale Übersetzungen für die Teilausdrücke zu finden. Man betrachte etwa einen Ausdruck  $E = E_1 \text{ op } E_2$ . Dann wird die optimale Übersetzung für  $E$  dadurch gebildet, dass man die optimalen Übersetzungen für  $E_1$  und  $E_2$  mit dem Code zur Auswertung von  $\text{op}$  kombiniert.

Sei  $T$  der Syntaxbaum für  $E$  und  $T_1$  bzw.  $T_2$  die Teilbäume für  $E_1$  bzw.  $E_2$



Ein Programm  $P$  wertet einen Baum  $T$  *kontinuierlich* aus, wenn zuerst die Teilbäume von  $T$  ausgewertet werden, deren Ergebnisse in den Speicher geschrieben werden. Anschließend wird der Rest von  $T$  ausgewertet, entweder in der Reihenfolge  $T_1, T_2$  oder in der Reihenfolge  $T_2, T_1$ , wobei die im Speicher abgelegten Werte mit benutzt werden. Abschließend wird dann der Wert für die Operation an der Wurzel von  $T$  berechnet.

Für die in diesem Abschnitt betrachteten Maschinenbefehle kann man zeigen, dass für jede Übersetzung  $P$  von  $T$  ein äquivalentes Programm  $P'$  existiert mit:

- 1)  $P'$  hat Kosten kleiner gleich den Kosten von  $P$
- 2)  $P'$  nutzt nicht mehr Register als  $P$
- 3)  $P'$  wertet den Baum kontinuierlich aus.

Daraus folgt, dass jeder Syntaxbaum eines arithmetischen Ausdrucks durch ein optimales Programm kontinuierlich ausgewertet werden kann.

**Bem.:** Dies gilt jedoch nur, wenn die erlaubten Maschinenbefehle vom beschriebenen Typ sind. Zum Beispiel für Maschinen, die Registerpaare für Multiplikation und Division benutzen gilt dies nicht. Man kann zeigen, dass es Beispiele optimaler Programme für derartige Maschinen gibt, in denen erst Teile von  $T_1$ , dann Teile von  $T_2$ , dann wieder Teile von  $T_1$  usw., ausgeführt werden müssen.

Die Eigenschaft der kontinuierlichen Auswertung eines arithmetischen Syntaxbaums garantiert, dass es ein optimales Programm gibt, das aus optimalen Programmen für die Teilbäume der Wurzel und anschließend einem Befehl zum Auswerten der Wurzel gibt. Diese Eigenschaft benutzen wir nun bei dem folgenden Algorithmus:

### 7.5.1 Der Algorithmus

Der Algorithmus verläuft in drei Phasen:

- 1) In der ersten Phase wird für jeden Knoten  $n$  des Syntaxbaumes ein  $r + 1$ -stelliger Kostenvektor  $C$  berechnet.  $C[0]$  enthält die minimalen Kosten eines Maschinenprogramms zur Berechnung des Teilbaums mit Wurzel  $n$ , wobei das Ergebnis im Speicher abgelegt wird.  $C[i]$  enthält die minimalen Kosten eines Maschinenprogramms zur Berechnung dieses Teilbaums unter Verwendung von  $i$  Registern, wobei das Ergebnis in einem Register gehalten wird.

Zu jedem Eintrag des Kostenvektors merkt man sich ausserdem den Maschinenbefehl, der zu den minimalen Kosten geführt hat, um später das optimale Programm zu konstruieren.

- 2) Dann bestimmt man mit Hilfe der Kostenvektoren, welche Teilbäume  $S$  von  $T$  so ausgewertet werden müssen, dass die Ergebnisse im Speicher abgelegt werden.
- 3) Man erzeugt für jeden unter 2) gefundenen Teilbaum  $S$  mit Hilfe der Kostenvektoren und der zugeordneten Maschinenbefehle die Übersetzung von  $S$ . Dann erzeugt man, wieder unter Benutzung der Kostenvektoren und der zugeordneten Befehle, die Übersetzung des restlichen Baums.

Jede dieser Phasen kann so implementiert werden, dass sie in einer Zeit proportional zur Größe des Syntaxbaumes arbeitet.

Die Kosten zur Berechnung eines Teilausdrucks mit Wurzel  $n$  enthalten auch die Kosten aller dazu notwendigen Lade- Speicheroperationen sowie die Kosten zur Auswertung des Operators am Knoten  $n$ .

Um die Kosten  $C[i]$  für den Knoten  $n$  zu bestimmen, betrachtet man alle Maschinenbefehle, die die Operation in  $n$  realisieren. Für jeden möglichen Maschinenbefehl  $B$  berechnet man mit Hilfe der Kostenvektoren der Nachfolgerknoten von  $n$  die Kosten von  $n$ . Für die Register-Operanden von  $B$  betrachtet man dabei alle möglichen Reihenfolgen der Auswertung. Der erste Teilbaum kann mit  $i$  Registern ausgewertet werden, der nächste mit  $i - 1$  und so weiter. Hinzu kommen noch die Kosten für den Befehl  $B$ . Der Wert von  $C[i]$  ergibt sich dann als das Minimum aller Möglichkeiten. Üblicherweise speichert man zum Wert  $C[i]$  auch noch den Befehl  $B$ , der zum Minimum führte.

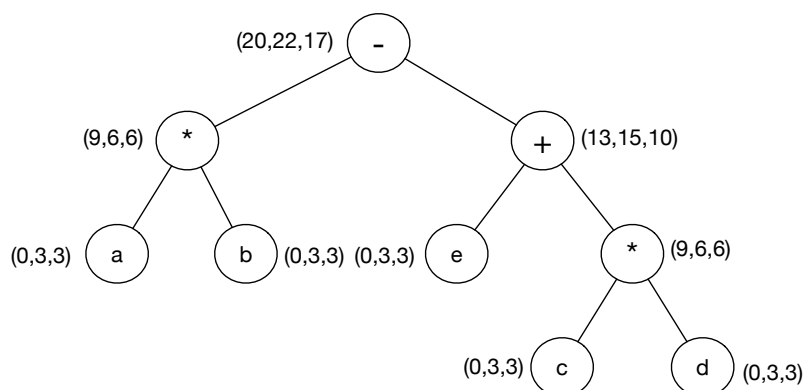
Letztlich gibt der minimale Wert im Kostenvektor der Wurzel von  $T$  die Kosten des optimalen Programms zur Auswertung von  $T$  an.

### Beispiel 7.10:

Wir betrachten eine Maschine, die zwei Register R0 und R1 frei hat und den folgenden Befehlssatz besitzt:

- LD *reg, mem*
- ST *mem, reg*
- OP *reg, reg, reg*
- OP *reg, reg, mem*

Übersetzt werden soll der Syntaxbaum aus Beispiel 7.7. Zunächst bestimmt man die Kostenvektoren für alle Knoten im Baum. Man erhält:





Betrachten wir den Kostenvektor  $(0, 3, 3)$  am Blatt  $a$ . Die Kosten um den Wert von  $a$  im Speicher zu erhalten sind offensichtlich 0, da  $a$  sich bereits im Speicher befindet. Also ist  $C[0] = 0$ . Um den Wert von  $a$  in einem Register zu erhalten, benötigt man einen Befehl `LD R0,a`, also ist  $C[1] = 3$ .  $C[2]$  sind die Kosten, den Wert von  $a$  in einem Register zu erhalten, wobei zwei Register verfügbar sind. Das sind natürlich die gleichen Kosten wie bei  $C[1]$ .

Um den Kostenvektor an der Wurzel des Baumes zu berechnen betrachten wir zunächst den Fall, dass nur ein Register zur Verfügung steht..

Der einzig anwendbare Befehl ist `SUB R0,R0,M`. Die minimalen Kosten sind gleich der Summe der Kosten, den linken Teilbaum mit einem Register auszuwerten plus den Kosten, den Wert des rechten Teilbaums im Speicher zu halten plus den Kosten des Befehls. Da keine andere Möglichkeit besteht, sind die Kosten also  $6 + 13 + 3 = 22$ .

Bestimmen wir nun den die Kosten für den Fall, dass zwei Register verfügbar sind.

Es sind drei Fälle zu unterscheiden:

- 1) Betrachten wir zunächst den Befehl `SUB R0,R0,M`. Wir können den linken Teilbaum in das Register `R0` auswerten (bei zwei verfügbaren Registern) und den Wert des rechten Teilbaums aus dem Speicher holen. Die Kosten wären  $6 + 13 + 3 = 22$ .
- 2) Wenn wir den Befehl `SUB R0,R0,R1` zur Auswertung der Wurzel benutzen und zunächst den linken mit zwei und dann den rechten Teilbaum mit einem verfügbaren Register auswerten, erhält man die Kosten  $6 + 15 + 1 = 22$ .
- 3) Wenn wir den Befehl `SUB R0,R0,R1` zur Auswertung der Wurzel benutzen und zunächst den rechten mit zwei und dann den linken Teilbaum mit einem verfügbaren Register auswerten, erhält man die Kosten  $10 + 6 + 1 = 17$ .

Also ergibt sich  $C[2] = 17$  als das Minimum der Werte.

Die minimalen Kosten zur Berechnung des Wurzelwerts im Speicher sind offensichtlich die minimalen Kosten zur Auswertung mit zwei Registern plus den Kosten eines Speicherbefehl, also  $C[0] = 17 + 3 = 20$ .

Nach der Berechnung der Kostenvektoren kann man nun den Baum ein weiteres mal durchlaufen und die Befehle, die zu den jeweils minimalen Kosten führten, zu einem optimalen Programm zusammensetzen. Man erhält:

```
LD  R0,c
MUL R0,R0,d
LD  R1,e
ADD R1,R1,R0
LD  R0,a
MUL R0,R0,b
SUB R0,R0,R1
```

Diese Methode ist auch für Rechner mit sehr viel komplexeren Maschinenoperationen verwendbar. Auch wenn man nicht in allen Fällen die Erzeugung eines optimalen Programms garantieren kann, liefert dieses Verfahren im Allgemeinen sehr gute Ergebnisse.

## 7.6 Globale Register Allokation

Da die Geschwindigkeit des Prozessors häufig um Größenordnungen schneller als die des Speichers ist, spielt die effiziente Nutzung der Register eine wichtige Rolle bei der Maschinencode-Erzeugung in einem optimierenden Compiler. Unser einfacher Maschinencode-Erzeuger aus dem vorigen Abschnitt hat aus diesem Grund die berechneten Werte so lange wie möglich in den Registern belassen. Allerdings werden diese Werte am Ende eines einfachen Blocks immer in den Speicher zurückgeschrieben. Es stellt sich somit die Frage, ob man nicht besser einen Teil der Register global, also über mehrere einfache Blöcke hinweg, einzelnen häufig benutzten Variablen zuordnet.

Welche Variablen einem Register über Blockgrenzen hinweg zugeordnet werden soll, kann bei einigen Programmiersprachen der Programmierer bestimmen, etwa durch die Deklaration einer Variablen mit der `register`-Anweisung in C, die ein Hinweis für den Compiler ist, falls möglich der deklarierten Variablen ein freies Register zuzuordnen. Heutzutage wird diese Aufgabe aber besser durch optimierende Compiler gelöst. Meist werden ein paar Register für die Variablen in inneren Schleifen reserviert und dann für diese Schleife fest zugeordnet. Die Frage bleibt natürlich, welche Variablen auf diese Weise einem Register zugeordnet werden sollen.

### 7.6.1 Usage Counts

Diese Methode beruht auf der Idee, für alle in einer inneren Schleife  $L$  auftretenden Variablen  $v$ , den Zeitgewinn abzuschätzen, den man durch eine feste Zuordnung dieser Variablen zu einem Register erzielen kann. Wir wollen annehmen, dass die Maschinencode-Erzeugung für einen einfachen Block wie im vorigen Abschnitt durchgeführt wird und wir wollen die Abarbeitungszeit der Befehle wie im Abschnitt 7.2 durch Kosten approximieren.

Wenn wir die Variable  $v$  für die Schleife  $L$  einem Register  $R_v$  fest zuordnen, erhalten wir (für unser Maschinenmodell) eine Kostenersparnis von 2 für jeden Gebrauch von  $v$  in einem Block  $B$  von  $L$ , in dem  $v$  nicht vorher definiert wurde (statt `ADD ...,v,...` hätte man den Befehl `ADD ...,Rv,...`). Ausserdem sparen wir auch noch Kosten von 3, wenn  $v$  am Ende des Blocks  $B$  lebendig ist, da wir auf das Abspeichern des Werts von  $v$  mit `ST v,Ri` verzichten können. Wird also  $v$  für die Schleife  $L$  fest einem Register zugeordnet, sparen wir etwa

$$\sum_{B \in L} 2 \cdot \text{use}(v, B) + 3 \cdot \text{live}(v, B)$$

Kosten, wobei

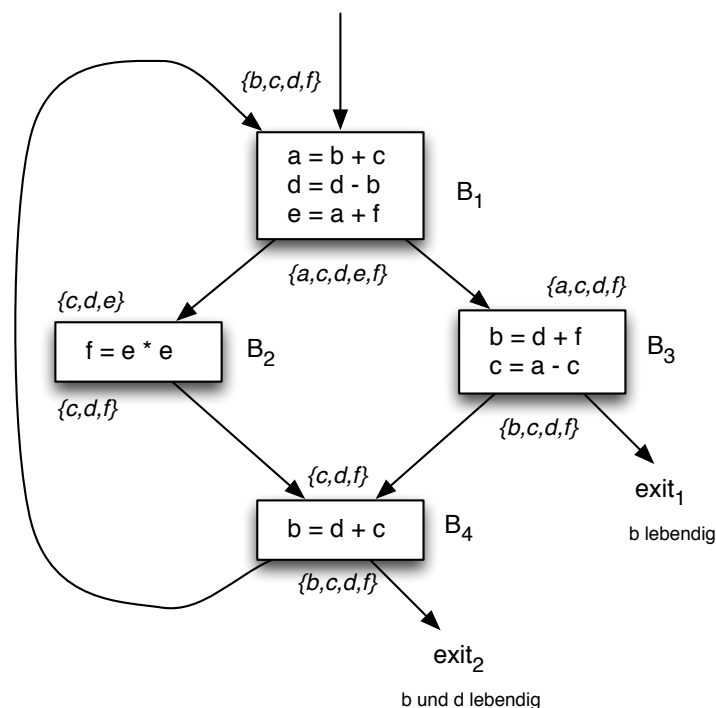
- $\text{use}(v, b)$  die Anzahl des Gebrauchs von  $v$  in  $B$  vor einer Definition von  $v$  in  $B$  und
- $\text{live}(v, B) = \begin{cases} 1 & \text{falls } v \text{ in } B \text{ definiert wurde und lebendig am Ende von } B \text{ ist} \\ 0 & \text{sonst.} \end{cases}$

Dieser Wert stellt natürlich nur eine grobe Näherung der eingesparten Kosten dar, weil verschiedenen Blöcke in der Schleife unterschiedlich häufig durchlaufen werden können. Auch ignorieren wir den Aufwand für das Laden und Speichern von  $v$  beim Eintritt und Austritt aus der Schleife, da dieser Aufwand nur einmal anfällt, wogegen wir annehmen können, dass die Blöcke in der Schleife häufiger durchlaufen werden.

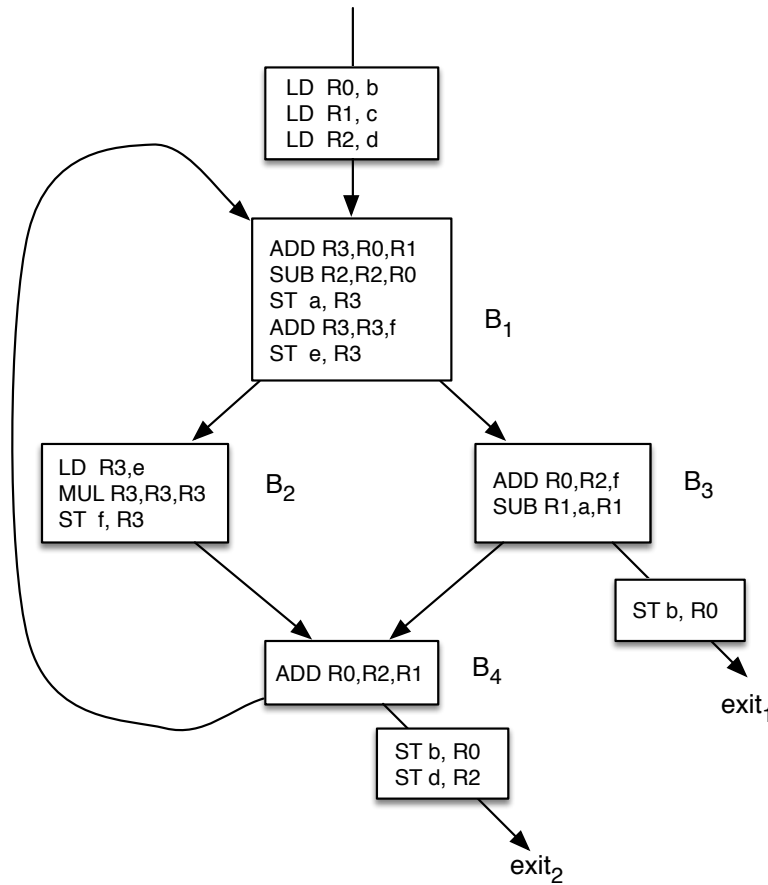
**Bemerkung:** Diese Strategie zur festen Zuordnung von Registern zu Variablen lässt sich natürlich auch für eine  $L$  umfassende Schleife  $L'$  verallgemeinern, wobei eventuell Registerwerte beim Verlassen von  $L$  gespeichert werden müssen.

**Beispiel 7.11:**

Gegeben sei der folgende Flußgraph einer Schleife, bei dem die Sprungbefehle und Label weggelassen wurden. Die lebendigen Variablen am Blockanfang und -ende sind jeweils angegeben.



Wenn man jetzt mit dieser Information die Kostenersparnis für die Variable  $a$  berechnen will, stellt man fest, dass  $a$  nur in  $B_3$  ohne vorige Definition benutzt wird und nur am Ende von  $B_1$  lebendig ist. Also wäre die Ersparnis 5. Die entsprechenden Werte für  $b$ ,  $c$ ,  $d$ ,  $e$  und  $f$  sind 10, 9, 9, 7 und 7. Hat man etwa die drei Register R0, R1 und R2 zur festen Zuordnung zur Verfügung und das Register R3 zur freien Verfügung, so könnte man die ersten drei Register den Variablen  $b$ ,  $c$  und  $d$  fest zuordnen. Dann würde sich mit der Maschinencode-Erzeugung aus dem vorigen Abschnitt folgendes Programmfragment ergeben (die Marken und Sprungbefehle sind wieder nicht angegeben):



### 7.6.2 Register Allokation durch Graph-Färbung

Wenn für eine Berechnung ein Register benötigt wird, aber im Moment alle Register in Gebrauch sind, also Werte von Variablen enthalten, die noch benötigt werden, muss ein Register ausgewählt werden und der Inhalt in den Speicher zurückgeschrieben werden (*spill*). Eine systematische Methode, für einen Flussgraphen Register zu allokalieren und das Zurückschreiben zu organisieren, beruht auf der Färbung von Graphen.

Diese Methode läuft in zwei Phasen ab. In der Phase 1 wird der Zwischencode so Maschinenbefehle übersetzt, als ob eine beliebige Zahl von (symbolischen) Registern zur Verfügung stehen würde. Die Namen von Variablen werden einfach als Registerbezeichnungen interpretiert und meist kann man die Drei-Adress-Befehle direkt als Maschinenbefehle interpretieren. Falls ein Zugriff auf Werte über spezielle Register, wie etwa über den Stackpointer oder Framepointer geschieht, nehmen wir an, dass es spezielle Register für diese Aufgaben gibt, die nicht weiter betrachtet werden.

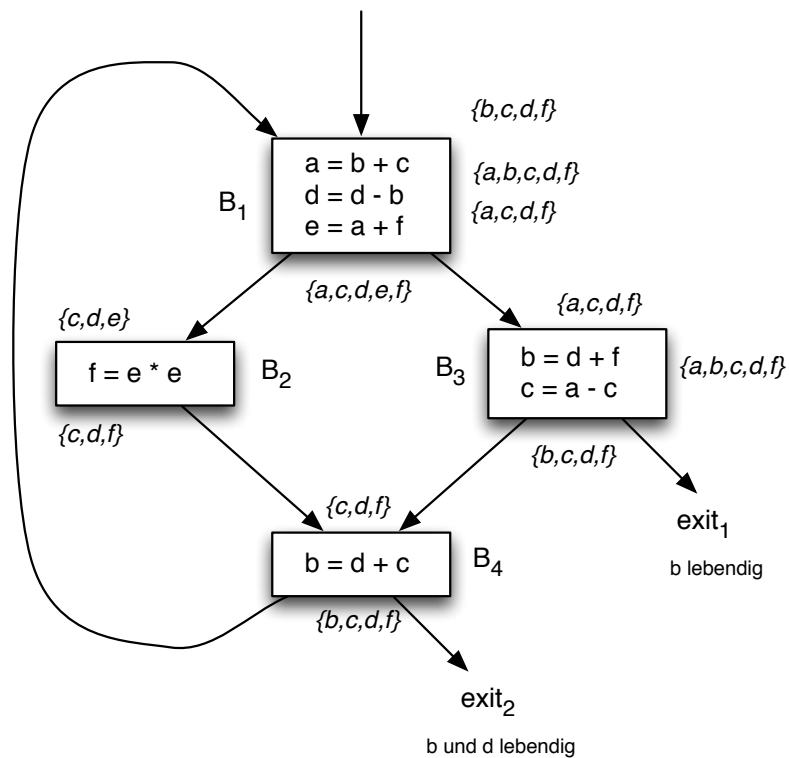
In einer zweiten Phase werden die vorhandenen freien Register den symbolischen Registern zugeordnet, wobei das Ziel ist, die Anzahl der Rückspeicherungen von Werten möglichst klein zu halten. Zunächst bestimmt man für jeden Programmpunkt die lebendigen Variablen. Als nächstes konstruiert man den Register Inference Graphen (RIG). Dies ist ein ungerichteter Graph, der als Knotenmenge die symbolischen Register hat. Zwei Knoten sind verbunden, wenn an einem Punkt im Programm beide symbolischen Register gleichzeitig lebendig sind. Das bedeutet, dass die Werte dieser beiden symbolischen Register nicht dem gleichen realen Register zugeordnet werden können.

Damit ist aber klar, dass die minimale Zahl von Registern, die notwendig ist um den Flussgraphen oder Zurückspeichern auszuwerten, gleich der minimalen Zahl von Farben ist, mit denen man den RIG zulässig (d.h. verbundene Knoten haben unterschiedliche Farben) färben kann. Hat man also  $k$  Register zur Verfügung, kann man eine Register-Allokation ohne Zurückspeicherung finden, falls der RIG  $G$   $k$ -färbbar ist. Leider ist die Frage, ob ein gegebener Graph mit  $k$  Farben zulässig zu färben ist, NP-Vollständig. Aber es gibt eine in der Praxis häufig erfolgreich eingesetzte heuristische Näherung für dieses Problem.

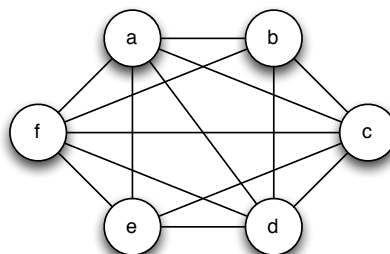
- 1) Hat ein Knoten  $n$  in  $G$  weniger als  $k$  Nachbarn, entfernt man diesen Knoten und die anhängenden Kanten aus  $G$ . Man erhält so einen Graphen  $G'$ . Ist  $G'$   $k$ -färbbar, so kann man diese Färbung leicht zu einer  $k$ -färbung für  $G$  erweitern, in dem man den Knoten  $n$  mit einer der Farben färbt, die kein Nachbar von  $n$  hat.
- 2) Erhält man durch Iteration dieses Schritts einen leeren Graphen, so kann man eine  $k$ -Färbung vom Originalgraphen  $G$  konstruieren, indem man die entfernten Knoten in umgekehrter Reihenfolge ihrer Entfernung wieder einfügt und färbt.
- 3) Erhält man durch Iteration von Schritt 1 einen Graphen  $\tilde{G}$ , in dem jeder Knoten  $\geq k$  Nachbarn, muss man ein Register durch Zurückspeichern frei machen. Man wählt eines der symbolischen Register nach heuristischen Regeln aus, etwa eine Variable, die nur wenig gebraucht wird. Man sollte aber vermeiden, in inneren Schleifen eine Zurückspeicherung einzufügen. An jeder Stelle im Programm, an der diese Variable gebraucht wird, fügt man eine Lade-Operation ein und an jeder Stelle, an der diese Variable definiert wird, fügt man eine Speicher-Operation ein. Dann wird die Information über die Lebendigkeit der Variablen und damit ein RIG aktualisiert und man iteriert das Verfahren weiter.

**Beispiel 7.12:**

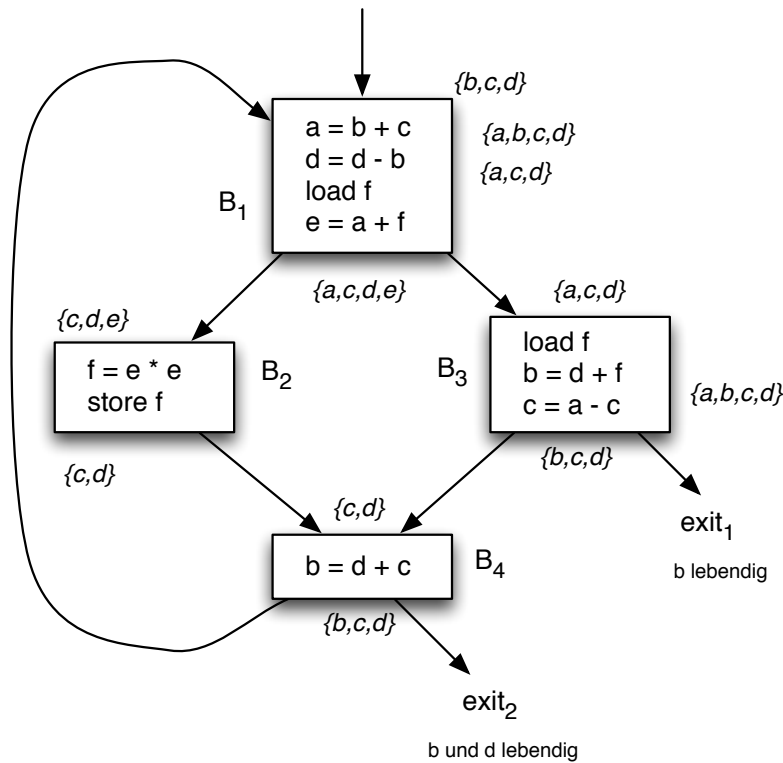
Betrachten wir den Flussgraph aus Beispiel 7.11 und nehmen wir an, dass auch wieder die vier Register R0, R1, R2 und R3 zur Verfügung stehen. In unserem Fall können die Variablennamen direkt als Namen der symbolischen Register benutzt werden. Für jeden Punkt im Flussgraph bestimmt man die lebendigen Variablen. Man erhält:



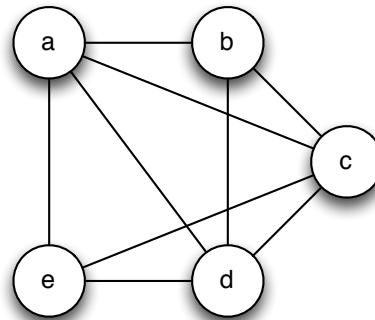
Mit diesen Informationen konstruiert man den Register Inference Graph (RIG):



Nun beginnt die zweite Phase, also die Zuweisung der vier Register zu den Variablen. Wie man sofort sieht, gibt es keinen Knoten mit weniger als vier Nachbarn, also muss eine der Variablen quasi „im Speicher leben“. Wählt man zum Beispiel die Variable  $f$  aus, da sie nur dreimal im Flussgraphen auftritt und viele Nachbarn hat, erhält man den folgenden geänderten Flussgraphen:



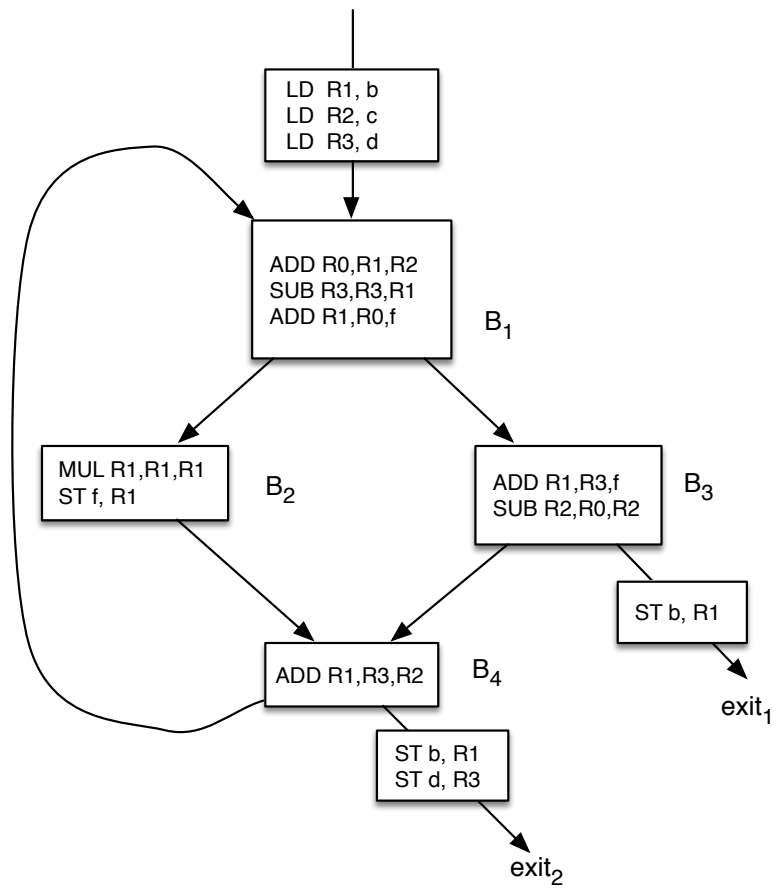
und den folgenden RIG:



Nun können wir den Knoten  $e$  auswählen, da er nur 3 Nachbarn besitzt. Dann könnte man die Knoten  $d$ ,  $a$ ,  $b$  und  $c$  (in dieser Reihenfolge) entfernen und landen beim leeren Graphen. Also ist dieser RIG 4-färbbar. Man ordnet den Knoten in umgekehrter Reihenfolge jetzt Farben zu. Für die vier Knoten  $a$ ,  $b$ ,  $c$  und  $d$  benötigt man jeweils eine andere Farbe; der Knoten  $e$  kann aber in der gleichen Farbe gefärbt werden, die Knoten  $b$  hat.

Da die Farben Registern entsprechen, bedeutet dies, dass ein Register die Wert der Variablen  $b$  und  $e$  nacheinander ohne störende Interaktionen aufnehmen kann.

Ordnen wir der Variablen  $a$  das Register R0,  $c$  entsprechend R2,  $d$  R3 und den Variablen  $b$  und  $e$  das Register R1 zu und übersetzen die Lade- und Speicheroperationen durch entsprechende Adressierung, so erhält man folgendes Programmfragment:





## 7.7 Generatoren für die Maschinencode-Erzeugung

**Ziel:** Die automatische Generierung eines Maschinencode-Erzeugers aus einer Beschreibung der Zielmaschine.

Dazu existieren mehrere wichtige Ansätze:

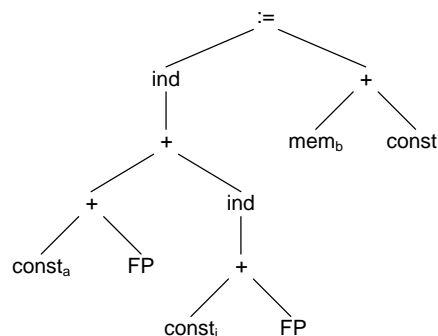
- über Baum-Grammatiken (allgemeiner Ansatz von Cattell [27])
- über SDTS und LR-Grammatiken für sequentiell dargestellte Syntaxbäume (Ansatz von Graham und Glanville [30])
- über Baum-Matching in Verbindung mit dynamischer Programmierung (Ansatz von Aho, Ganapathi, Tjiang, [22])

Bei allen Ansätzen müssen die Syntaxbäume relativ weit „aufgefächert“ sein, denn in allen Fällen wird die Arbeitsweise einzelner Maschinenbefehle durch Bäume bzw. Zeichenketten dargestellt.

### Beispiel 7.13:

Der Syntaxbaum für  $a[i] := b + 1$  wird durch eine explizite Speicherzuordnung relativ groß. Es wird angenommen, dass  $b$  eine globale Variable ist und dass  $a$  und  $i$  lokal auf dem Laufzeit-Stack gespeichert sind. Das Feld  $a$  habe als unteren Index die 0. Zudem seien alle Variablen vom Typ `character` bzw. `integer`.

$\text{const}_a$  und  $\text{const}_i$  sind die Abstände der Speicherorte von  $a$  und  $i$  relativ zum Framepointer FP,  $\text{mem}_b$  ist die Speicheradresse von  $b$ .



### 7.7.1 Verwendung von Baum-Grammatiken bzw. Baum-Übersetzungs-Schemata

Die prinzipielle Vorgehensweise ist wie bei üblichen kontextfreien Grammatiken, nur werden statt Zeichenketten Bäume erzeugt. Für die Übersetzung benötigt man Produktionen der Form:

$$A \longrightarrow T\{\text{Regel}\} \quad \text{oder} \\ A \longrightarrow T\{\text{Kosten}\}\{\text{Regel}\}.$$

Dabei ist  $A$  ein nichtterminales Symbol,  $T$  ist ein Baum, dessen Blätter mit terminalen oder nichtterminalen Symbolen markiert sind, und alle internen Knoten von  $T$  sind mit terminalen Symbolen markiert.

Die „Regel“ gibt die semantische Aktion an. In unserer Anwendung wird jede Produktion die Abarbeitung eines Befehls darstellen. Die semantische Aktion besteht dann aus der Ausgabe des Befehls. Häufig sind derartige Baumgrammatiken, die die Wirkungsweise der Maschinenbefehle eines Rechners beschreiben, stark mehrdeutig. Um die verschiedenen Möglichkeiten bewerten zu können, wird den Produktionen auch noch eine Kostenfunktion zugeordnet, die dann zur Auswahl der „richtigen“ Produktionen benutzt wird.

**Beispiel 7.14:**

**Bem.:** Die tiefgestellten Buchstaben haben die Bedeutung von Zusatzbedingungen bzw. semantischen Prädikaten (Attributen).

**Beispiel 7.15:**

Es soll jetzt ein Ausschnitt einer Baumgrammatik für einige Maschinenbefehle eines hypothetischen Rechners angegeben werden. In den Regeln wird hier zur besseren Lesbarkeit statt einer ausführbaren Anweisung der erzeugte Assemblercode angegeben.

Nr.	Produktion	Regel, (erzeugter Befehl)
1	$\text{reg}_i \rightarrow \text{const}_c$	$\{ \text{LD } R_i, \#c \}$
2	$\text{reg}_i \rightarrow \text{mem}_a$	$\{ \text{LD } R_i, a \}$
3	$\text{reg}_{FP} \rightarrow \text{FP}$	
4	$\text{mem} \rightarrow \begin{array}{c} := \\ \swarrow \quad \searrow \\ \text{mem}_a \quad \text{reg}_i \end{array}$	$\{ \text{ST } a, R_i \}$
5	$\text{mem} \rightarrow \begin{array}{c} := \\ \swarrow \quad \searrow \\ \text{ind} \quad \text{reg}_j \\   \\ \text{reg}_i \end{array}$	$\{ \text{ST } *R_i, R_j \}$
6	$\text{reg}_i \rightarrow \begin{array}{c} \text{ind} \\   \\ + \\ \swarrow \quad \searrow \\ \text{const}_c \quad \text{reg}_j \end{array}$	$\{ \text{LD } R_i, c(R_j) \}$
7	$\text{reg}_k \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_i \quad \text{ind} \\ \quad \quad   \\ \quad \quad + \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad \text{const}_c \quad \text{reg}_j \end{array}$	$\{ \text{ADD } R_k, R_i, c(R_j) \}$
8	$\text{reg}_k \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_i \quad \text{reg}_j \end{array}$	$\{ \text{ADD } R_k, R_i, R_j \}$
9	$\text{reg}_i \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_i \quad \text{const}_1 \end{array}$	$\{ \text{INC } R_i \}$

**Bem.:** FP bezeichnet den Framepointer, der hier immer in einem fest zugeordneten Register  $\text{reg}_{FP}$  gespeichert ist. In diesem Beispiel wird keine Kostenfunktion verwendet, sondern es wird beim Parsing die heuristische Regel verwendet, dass bei mehreren Möglichkeiten immer die Produktion mit der größeren rechten Seite verwendet wird.

Nun stellt sich die Frage: Wie übersetzt man mit Hilfe einer derartigen Grammatik? Es ist klar, dass man auch hier den gegebenen Baum parsen muss und dann, ähnlich einer attribuierten Grammatik (oder einem SDTS), die entsprechenden semantischen Aktionen oder Regeln ausführen muss.

Man kann sich vorstellen, dass Parsingverfahren für Baum-Grammatiken signifikant komplexer sind als Parsingverfahren für Zeichenketten. Trotzdem beruhen sie auf den gleichen Prinzipien. So könnte man etwa nach dem Bottom-Up Verfahren vorgehen. Also wird man versuchen, den vorgelegten Syntaxbaum von unten nach oben mit Hilfe der gegebenen Regeln zu reduzieren.

Dabei stellen sich folgende Fragen:

- 1) In welcher Reihenfolge sollen die Reduktionen durchgeführt werden?
  - 2) Was ist zu tun, wenn mehrere Regeln anwendbar sind?
  - 3) Was ist zu tun, wenn keine Regel anwendbar ist?
  - 4) Wie verhütet man unendliche Schleifen in der Reduktionsphase?
- zu 1): Meistens wird dieses Verfahren mit Varianten der dynamischen Programmierung verbunden. Dadurch wird ein Bottom-Up-Vorgehen festgelegt und das ganze wird dann zu einem Pattern-Matching-Problem über Bäumen.
- zu 2): Man kann z.B. eine heuristische Regel wie „wähle einen möglichst großen Teilbaum“ verwenden, um zu einem möglichst kurzen Maschinenprogramm zu gelangen. Bessere Ergebnisse erzielt man aber bei der Anwendung der dynamischen Programmierung. Hierbei betrachtet man die Kosten von Übersetzungen von Teilbäumen für unterschiedliche Registerzahlen und benutzt diese Teillösungen, um eine Gesamtlösung zusammenzusetzen.
- zu 3): Im schlimmsten Fall bleibt keine andere Möglichkeit als Backtracking.
- zu 4): Dieses Problem kann man etwa mit einer Kostenfunktion und dynamischer Programmierung lösen.

**Beispiel 7.16:**

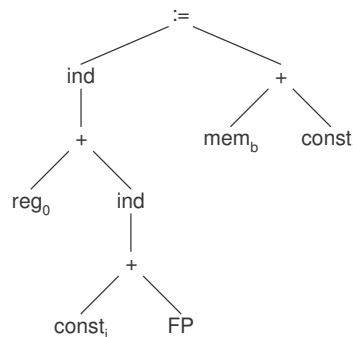
Betrachten wir den Syntaxbaum aus Beispiel 7.13 und die Beispielgrammatik aus Beispiel 7.15. Man versucht nun, den Syntaxbaum Bottom-Up und von links nach rechts zu parsen. Dabei muss man die folgenden Regeln nacheinander anwenden:

1) Regel (1):  $\text{reg}_0 \rightarrow \text{const}_a$  erzeugt  $\{\text{LD R0}, \#a\}$

2) Regel (3):  $\text{reg}_{FP} \rightarrow \text{FP}$

3) Regel (8):  $\text{reg}_0 \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_0 \quad \text{reg}_{FP} \end{array}$  erzeugt  $\{\text{ADD R0}, \text{R0}, \text{FP}\}$

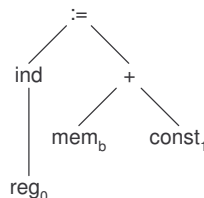
Damit hat man als Zwischenergebnis den Baum:



Nach der erneuten Anwendung von Regel (3) sind jetzt zwei Regeln anwendbar: die Regel (6) und die Regel (7). Da die rechte Seite der Regel (7) „größer“ ist (einen größeren Teilbaum überdeckt), wird diese Regel verwendet:

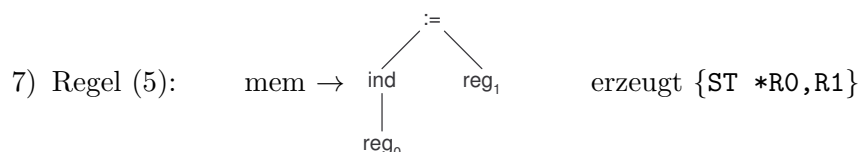
4) Regel (7):  $\text{reg}_0 \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_0 \quad \text{ind} \\ \quad \quad \quad \swarrow \quad \searrow \\ \quad \quad \quad \text{const}_i \quad \text{reg}_{FP} \end{array}$  erzeugt  $\{\text{ADD R0}, \text{R0}, i(\text{FP})\}$ .

Damit hat man folgendes Zwischenresultat:



5) Regel (2):  $\text{reg}_1 \rightarrow \text{mem}_b$  erzeugt  $\{\text{LD R1}, b\}$

6) Regel (9):  $\text{reg}_1 \rightarrow \begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{reg}_1 \quad \text{const}_i \end{array}$  erzeugt  $\{\text{INC R1}\}$



Man erhält also folgendes Maschinencode-Programm:

```
LD    R0,#a
ADD   R0,R0,FP
ADD   R0,R0,i(FP)
LD    R1,b
INC   R1
ST    *R0,R1
```

### 7.7.2 Verwendung eines üblichen LR-Parsers

Ein anderer Ansatz [30] transformiert den vorliegenden Eingabebaum in eine Zeichenkette und verwendet die üblichen Parsermethoden, z.B. den Parsergenerator *yacc*. Bei den Baum-Produktionen müssen die rechten Seiten natürlich ebenfalls transformiert werden, so dass „übliche“ kontextfreie Produktionen entstehen.

#### Beispiel 7.17:

Wir betrachten wieder den Syntaxbaum aus Beispiel 7.13. Als Transformation verwenden wir die sequentielle Darstellung des Baumes in Präfix-Notation. Da der Verzweigungsgrad der internen Knoten implizit gegeben ist, müssen keine weiteren Informationen abgespeichert werden. Die sequentielle Darstellung des Syntaxbaums wäre dann:

:= ind + + const<sub>a</sub> FP ind + const<sub>i</sub> FP + mem<sub>b</sub> const<sub>1</sub>

Die transformierte Baum-Grammatik wird als SDTS umschreiben:

Nr.	Produktion	Regel (erzeugter Befehl)
(1)	$\text{reg}_i \rightarrow \text{const}_c$	LD Ri,#c
(2)	$\text{reg}_i \rightarrow \text{mem}_a$	LD Ri,a
(3)	$\text{reg}_{FP} \rightarrow \text{FP}$	
(4)	$\text{mem} \rightarrow \text{:= mem}_a \text{ reg}_i$	ST a,Ri
(5)	$\text{mem} \rightarrow \text{:= ind reg}_i \text{ reg}_j$	ST *Ri,Rj
(6)	$\text{reg}_i \rightarrow \text{ind} + \text{const}_c \text{ reg}_j$	LD Ri,c(Rj)
(7)	$\text{reg}_k \rightarrow + \text{reg}_i \text{ ind} + \text{const}_c \text{ reg}_j$	ADD Rk,Ri,c(Rj)
(8)	$\text{reg}_k \rightarrow + \text{reg}_i \text{ reg}_j$	ADD Rk,Ri,Rj
(9)	$\text{reg}_i \rightarrow + \text{reg}_i \text{ const}_1$	INC Ri

und man erzeugt dann, etwa mit Hilfe von *yacc* oder *bison*, einen Bottom-Up Parser für diese Grammatik und übersetzt die Eingabe in der üblichen Art und Weise.

#### Probleme:

- 1) Die aus den Maschinenbefehlen konstruierte Grammatik ist meist hochgradig mehrdeutig. Hier hilft eventuell die heuristische Regel, möglichst viel in einem Schritt zu reduzieren. Also wird bei shift-reduce-Konflikten auf shift entschieden und bei reduce-reduce-Konflikten die längere Produktion bevorzugt.

- 2) Die Reihenfolge der Auswertung von Teilbäumen ist durch das Parsingverfahren festgelegt (links  $\rightarrow$  rechts).
- 3) Es kann immer noch passieren, daß der Parsingprozess blockiert oder in eine Schleife gerät.
- 4) Die Registerzuweisung ist offen und muss von außen gesteuert werden.

#### Vorteile::

- Es ist prinzipiell relativ einfach, einen nach dieser Methode arbeitenden Maschinencode-Erzeuger auf eine neue Zielmaschine umzuprogrammieren.
- Unter gewissen Voraussetzungen ist es möglich zu beweisen, dass stets korrekter Code erzeugt wird, dass der Übersetzer nicht blockiert und nicht in eine Schleife gelangen kann.

Erweitert man die Grammatik, genauer gesagt, die Attributierung, so ist es möglich, gewisse Schwierigkeiten des ursprünglichen Ansatzes zu umgehen. So ist es zum Beispiel möglich, nur gewisse Zahlbereiche für Operanden zuzulassen. Hier wird als Beispiel angenommen, dass kleine Summanden in den Maschinenbefehl „inkodiert“ werden können, größere dagegen ein Erweiterungswort benötigen.

$$\text{reg}_i \rightarrow + \text{reg}_i \text{ const}_c \quad \left\{ \begin{array}{l} \text{if } 1 \leq c \leq 8 \text{ then ADDQ \#}c, \text{ Ri} \\ \text{else ADD \#}c, \text{ Ri} \end{array} \right\}$$

Man sollte die Werte der semantischen Prädikate natürlich mit in die Entscheidung zur Auswahl der passenden Produktion zur Reduktion einfließen lassen. Durch diese Erweiterung ist es aber meist nicht mehr möglich, die korrekte und störungsfreie Übersetzung zu beweisen.

### 7.7.3 Der Ansatz von Aho, Ganapathi und Tjiang

Der Ansatz von Aho, Ganapathi und Tjiang beruht wiederum auf Baumgrammatiken. Die Autoren definieren ein sogenanntes **Baum-Übersetzungs-Schema** (tree-translation-scheme) und geben eine Methode an, wie mit Hilfe von Pattern-Matching und dynamischer Programmierung ein solches System implementiert werden kann. Dieses System namens *twig* ist zur Generierung von Maschinencode-Erzeugern für mehrere Compiler benutzt worden.

Eine allgemeine Produktion hat bei diesem Ansatz folgendes Aussehen:

$$A \rightarrow T \quad \{\text{cost}\} = \{\text{action}\}$$

Dabei ist  $A$  ein nichtterminaler Knoten,  $T$  ein Baum,  $\text{cost}$  ist ein Code-Fragment, das zur Berechnung der Kosten dieses Musters aufgerufen wird. Fehlt dieses Feld, so werden Einheitskosten angenommen.  $\text{action}$  ist ein Code-Fragment, das bei Akzeptieren dieser Produktion ausgeführt wird und üblicherweise den Maschinencode generiert.

#### Beispiel 7.18:

Die Produktion (6) aus Beispiel 7.15 würde hier wie folgt dargestellt:

$$\text{reg}_i \rightarrow \begin{array}{c} \text{ind} \\ | \\ + \\ / \quad \backslash \\ \text{const}_c \quad \text{reg}_j \end{array} \quad \{ 3 + \text{cost.reg}_j \} = \{ \text{LD Ri, c(Rj)} \}$$

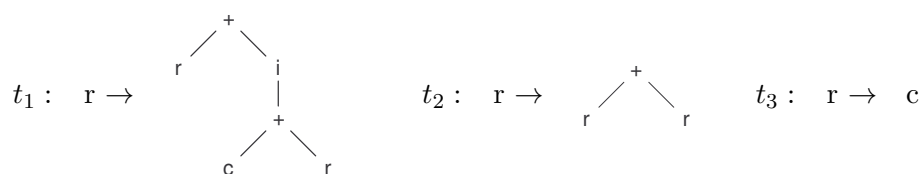
In die Berechnung der Kostenfunktion gehen in diesem Fall die notwendigen Kosten zur Berechnung des Wertes im Register Rj mit ein. Die Kosten werden zur Bestimmung der anzuwendenden Produktionen benutzt.

Interessant ist, dass hier nicht eines der üblichen Parsingverfahren modifiziert wird, sondern es wird ein Tree-Pattern-Matching angewendet. Dieses Verfahren basiert auf einer Idee von Hoffman und O' Donnell [32] und stellt eine Erweiterung des Aho-Corasick-Verfahrens [19] dar.

**Idee:** Die Bäume werden durch die Menge aller Wege von der Wurzel zum Blatt festgelegt, wobei die Verzweigungsnummern (die Nummer des Nachfolgers) auch mit notiert sind. Für die so erzeugten Pattern wird ein Aho-Corasick-Automat konstruiert, der eine parallele Suche nach all diesen Pattern im Baum erlaubt.

**Beispiel 7.19:**

Man betrachte die folgenden drei Produktionen einer Baumgrammatik:



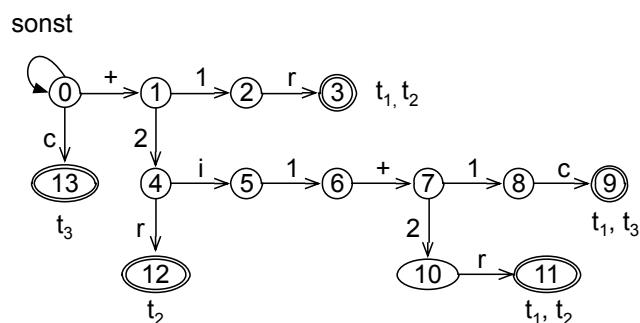
Die Menge der Wege ist:

```

+ 1 r
+ 2 i 1 + 1 c
+ 2 i 1 + 2 r
+ 2 r
c

```

wobei zu bemerken ist, dass der Weg + 1 r sowohl zu  $t_1$  als auch zu  $t_2$  gehört. Der Aho-Corasick-Automat ist dann also:



**Bem.:** Die Werte der Fehler-Funktion sind aus Gründen der besseren Übersichtlichkeit in der obigen Abbildung nicht mit eingetragen!

Anwendung findet nun eine Technik von Hoffman und O' Donnell, um jetzt die Pattern im Baum zu erkennen. Man durchläuft den Eingabe-Baum von oben nach unten und berechnet für jeden Knoten im Baum den Zustand des Automaten für den Weg von der Wurzel bis zu

diesem Knoten. Aus diesen Informationen lässt sich dann (bei geeigneter Datenstruktur) einfach erkennen, an welchen Stellen rechte Seiten von Regeln im Baum auftreten. Die berechneten Kosten werden dann zur Auswahl der „günstigsten“ Regel benutzt und die Regel dann angewendet, d.h. die rechte Seite der Regel wird im Eingabe-Baum durch die linke ersetzt. Es muss jetzt noch berücksichtigt werden, dass durch den Ersetzungsprozess ein Teilbaum durch einen Knoten ersetzt wird und damit wieder neue Matching-Möglichkeiten entstehen.



## Literatur

### allgemeine Literatur zur Vorlesung:

- [1] A.V. Aho, M.S. Lam, R. Sethi und J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, Pearson/Addison Wesley 2007.
- [2] H. Alblas und A. Nymeyer, *Practice and Principles of Compiler Building with C*, Prentice Hall 1996.
- [3] A.W. Appel *Modern Compiler Implementation in Java (2nd Edition)*, Cambridge University Press, 2002.
- [4] B. Bauer und R. Höllerer, *Übersetzung objektorientierter Sprachen* Springer 1998.
- [5] J.P. Bennett, *Introduction to Compiling Techniques (second edition), A first course using ANSI C, LEX, and YACC*, MacGraw-Hill 1996.
- [6] C. Fraser und D. Hanson, *A retargetable C Compiler: Design and Implementation*, The Benjamin/Cummings Publishing Company, 1995.
- [7] D. Grune, H.E. Bal, C.J.H. Jacobs und K.G. Langendoen *Modern Compiler Design*, Wiley 2000.
- [8] R.H. Güting und M. Erwig, *Übersetzerbau*, Springer 1999.
- [9] J. Levine, T. Mason und D. Brown, *Lex & Yacc (2nd Edition)*, O'Reilly & Associates, Inc. 1992.
- [10] S.S. Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufmann 1997.
- [11] T.W. Parsons, *Introduction to Compiler Construction*, Computer Science Press, 1992.
- [12] B.C. Pierce, *Types and Programming Languages*, MIT-Press, 2002.
- [13] T. Pittman und J. Peters, *The Art of Compiler Design*, Prentice Hall 1992.
- [14] T.W. Pratt und M.V. Zelkowitz, *Programming Languages - Design and Implementation*, Prentice-Hall 2001.
- [15] F.J. Schmitt, *Praxis des Compilerbaus*, Hanser 1992.
- [16] L. Schmitz, *Syntaxbasierte Programmierwerkzeuge*, Teubner 1995.
- [17] R. Wilhelm und D. Maurer, *Übersetzerbau - Theorie, Konstruktion, Generierung*, Springer 1992.
- [18] N. Wirth, *Compiler Construction*, Addison-Wesley 1996.

### spezielle Literatur

- [19] A.V. Aho und M.J. Corasick, *Efficient string matching: An aid to bibliographic search*, Commun. ACM 18 (1975), 333-340.
- [20] A.V. Aho, S.C. Johnson, *Optimal code generation for expression trees*, J. ACM 23 (1976), 488-501.

- [21] A.V. Aho, S.C. Johnson und J.D. Ullman, *Code Generation for expressions with common subexpressions*, J. ACM 24 (1977), 146-160.
- [22] A.V. Aho, M. Ganapathi, S.W.K. Tijang, *Code Generation Using Tree Pattern Matching and Dynamic Programming*, ACM TOPLAS 11 (1989), 491-516.
- [23] D.F. Bacon, S.L. Graham und O.J. Sharp, *Compiler Transformations for High Performance Computing*, ACM Comp. Surveys 26 (1994) 345-420.
- [24] J. Bruno, R. Sethi, *Code generation for a one-register machine*, J. ACM 23 (1976), 502-510.
- [25] C. Becker und P. Hagemann, *GRAM - Ein Drei-Adresscode Interpreter*, Studienarbeit (2002)
- [26] G.V. Bochmann, *Semantic Evaluation from Left to Right*, Comm. ACM 19 (1976), 55-62.
- [27] G.G. Cattell, *Automatic derivation of code generators from machine descriptions*, TOPLAS 2 (1980), 173-190
- [28] A.P. Ershov, *On programming of arithmetic operations*, Comm. ACM 1 (1958), 3-6, auch in Comm. ACM 1(1958), 16.
- [29] M. Ganapathi, C.N. Fischer, *Description-Driven Code Generation Using Attribute Grammars*, 9. ACM Symposium on Principles of Programming Languages (1982), 107-119.
- [30] R.S. Glanville, S.L. Graham, *A New Method for Compiler Code Generation* 5. ACM Symposium on Principles of Programming Languages (1978), 231-240.
- [31] A. Goldberg, D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley 1983.
- [32] C.W. Hoffman und M.J. O'Donnel, *Pattern matching in trees*, J. ACM 29 (1982), 68-95.
- [33] E.T. Irons, *A syntax directed compiler for Algol 60*, Comm. ACM 4 (1961), 51-55. Springer, 1987.
- [34] M. Jazayeri, W.F. Ogden und W.C. Rounds, *The intrinsic exponential complexity of the circularity problem for attribute grammars*, Comm. ACM 18 (1975), 697-706. Springer, 1983.
- [35] D.E. Knuth, *Semantics of context-free languages*, Math. System Theory 2 (1968), p 127-145, Errata Math. System Theory 5 (1971), 95-96.
- [36] P.M. Lewis, D.J. Rosenkrantz und E.E. Stearns, *Attributed Translations*, JCSS 9 (1969), 524-549.
- [37] R. Sethi, J.D. Ullman, *The generation of optimal code for arithmetic expressions*, J. ACM 17 (1970), 715-728.
- [38] L. Sterling und E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge 1986.
- [39] T. Wichers, *Optimale Code-Generierung mit Hilfe von Tree-Pattern Matching*, Masterarbeit, Universität Hannover, (2005).

## Index

- überladenen Operator, 30
- twig*, 138
- 3-Adress-Befehle, 3
  
- Abhängigkeitsgraph, 11
- Aktivierungs-Record, 40
  - Aufbau, 45
- allgemeinste Substitution, 35
- Allokation, 52
- annotierter
  - Ableitungsbaum, 6
- array
  - Speicherung
    - column-major, 63
    - row-major, 63
- Attribut, 5
  - inherit, 5
  - synthetisch, 5
- attributierte Grammatik, 6
- attributierter Ableitungsbaum, 6
  - ausgewerteter, 6
  - Auswertung des, 6
  - on-the-fly Auswertung, 12
  - tree-walk Auswertung, 12
- Ausdrücke
  - verfügbare, 100
- Auswertereihenfolge, 11
- available expressions, 100
  
- Back-Edges, 104
- Back-End, 56
- Backpatch, 73
- Baum-Übersetzungs-Schema, 138
- Boolesche Ausdrücke, 68
- Byte-Code, 3
  
- Closure, 42
- Code-Erzeugung, 3
- Code-Segment, 43
- Codeoptimierer
  - Aufbau, 79
- Codeoptimierung, 3
  - lokale, 87
- Copy Propagation, 85
  
- DAG, 87
- Daten-Segment, 43
- Datenfluss-Analyse, 91
- Datenfluss-Domain, 91
- Datenfluss-Gleichungen, 79
- Deallokation, 52
- Definition von Variablen, *siehe* Variable
- Definitionen
  - verfügbare, 93
- direkter Dominator-Block, 103
- Display, 41
- dominiert, 103
- Drei-Adress-Befehle, 58
  - Quadrupel-Darstellung, 61
  - Tripel-Darstellung, 61
- du-Kette, 91
- dynamische Programmierung, 122
- dynamischer Link, 40
  
- einfache Blöcke, 80
- Ershov-Zahlen, 117
- expression, 69
- Extend, 38
  
- Flussgraph, 81
  - erweiterter, 81
  - Startknoten, 81
- forward references, 70
- Frame-Pointer, 43
  
- Gültigkeitsbereich, 38
- Garbage Collector, 44
- gemeinsame Teilausdrücke, 83
- Grammatik
  - L-attribuiert, 13
  - S-attribuiert, 7
  - Umformung, 17
  
- Handles, 44
- Heap, *siehe* Daten-Segment
  
- inheritance, 32
- Intermediate Language, 56
  
- Kette, 97
  
- l-Wert, 45
- Laufzeitstack, 40
- Lebensdauer, 38
- Lexem, 1

- Lexical Scanner, 2
- lexikale Analyse, 1
- lokale Entfernung gemeinsamer Teilausdrücke, 83
- Maschinencode-Erzeuger, 110
- mem, 117, 124
- Mutator, 53
- Parameterübergabe, 45
  - call-by-copy-restore, 45
  - call-by-name, 46
  - call-by-reference, 45
  - call-by-value, 45
- Parser, 2
- Pattern-Matching, 135
- polymorphe Funktionen, 33
- polymorphe Operatoren, 33
- Programm-Punkt, 91
- Prozedurparameter, 42
- r-Wert, 45
- Rückwärtsanalyse, 92
- Rückwärtskanten, 104
- reaching definitions, 93
- Reallokation, 53
- recursive descent parser, 15
- reference count, 52
- Regel
  - einfache, 59
- Regeln, 5
- Register Inference Graph, 128
- run-time stack, 40
- Scanner, 2
- Schachtelungsdiagramms, 40
- Schleife
  - Eintrittspunkt, 104
  - Header, 104
  - Kopf, 104
  - natürliche, 104
- Scope, 38
- SDD, 5
- SDTS, *siehe* syntax-gesteuertes Übersetzungsschema
- semantische Analyse, 2
- semantische Analyse, 21
- Semantische Funktionen, 5
- semantische Regeln, 5
- semantischen Prädikaten, 134
- stack-frame, 40
- Stack-Segment, 43
- statement, 69
- statischer Link, 41
- strongly typed language, 21
- subtype, 32
- syntaktische Analyse, 2
- Syntax-Bäume, 3
- Syntax-Directed Definition, 5
- syntax-gesteuertes Übersetzungsschema, 14
  - Aktionen, 14
- Syntaxbaum, 2
- tag-System, 21
- thunk, 46
- Token, 1
- Tokenklasse, 2
- Tokennamen, 2
- Tokenwert, 2
- Topologisches Sortieren, 11
- Transferfunktion
  - für Befehle, 92
  - für Blöcke, 92
- Typ
  - Äquivalenz, 26
  - Ausdruck, 22
  - Inferenz, 21
  - Namensäquivalenz, 28
  - Struktuträquivalenz, 28
  - Synthese, 22, 24
  - Variable, 34
- Typ-Ausdruck
  - Instanz, 35
- Typ-Definiton, *siehe* Variable
- Typ-Konstruktors, 22
- Typ-Prüfung
  - dynamische, 21
  - statische, 21
- Type-Checking, 21
- ud-Kette, 91
- Unifikation, 34
- Untertypen, 32
- use-definition-chaining, 93
- Variable
  - benötigt, 80
  - definiert, 80
  - gebraucht, 80
  - lebendig, 81

- lebendig, 97
- nächster Gebrauch, 110
- nicht lebendig, 81
- starke Typ-Prüfung, 21
- tot, 81, 97
- Typ, 21
- Typ-Äquivalenz, 26
  - Namensäquivalenz, 28
  - Strukturäquivalenz, 28
- Vererbung, 32
  - , einfache, 32
  - , mehrfache, 32
- Verkettung
  - dynamische, 40
  - statische, 41
- Vorwärtsanalyse, 92
- Weiterreichen von Kopien, 85
- Zielmaschine
  - Befehle, 107
- Zwischencode, 56
- Zwischencode-Erzeugung, 3