Übersetzer

Vorlesung WS 1997/98
Uwe Kastens

Ziele:

- grundlegende **Verfahren der Sprachimplementierung** erlernen,
- den Einsatz von **generierenden Werkzeugen** und von **Standardverfahren** kennen und schätzen lernen,
- Übersetzerkonstruktion als Beispiel für das Zusammenwirken von grundlegenden **Algorithmen**, **Theorien** und von **Software-Engineering-Methoden** bei **präzise definierten Aufgaben** erkennen,
- Anwendbarkeit der Verfahren auch für **Sprachen jenseits der Programmiersprachen** erkennen.

Inhalt

1. Übersetzeraufgaben und -struktur

Aufgabenzerlegung mit Bezug zu Sprachdefinitionen,
Symbole, Syntax, statische und dynamische Semantik

2. Übersetzerstruktur, Schnittstellen

zentrale Datenstrukturen
Interpretierer, Debugger

3. Lexikalische Analyse

Spezifikation und Implementierung endlicher Automaten,
Generatoren dafür
Module zur Speicherung und Codierung von Symbolen

4. Syntaktische Analyse

Ziel- und quellorientierte Verfahren,
Rekursiver Abstieg, LR-Verfahren, Generatoren dafür,
Grammatikklassen, Fehlerbehandlung

5. Semantische Analyse

Bezeichneridentifikation, Typprüfung,
attributierte Grammatiken, Generatoren dafür

6. Optimierende Programmtransformation

Datenflußanalyse spezifiziert durch Gleichungssysteme,
Lösungsalgorithmen, Transformationen

7. Code-Erzeugung

Typische Eigenschaften von Zielprozessoren,
Verfahren zur Registerzuteilung
Optimierende Code-Auswahl, Generatoren dafür

8. Code-Parallelisierung

parallele Funktionseinheiten, Pipelines, Abhängigkeitsgraphen

Literatur

U. Kastens:
Übersetzerbau, Handbuch der Informatik 3.3,
Oldenbourg, 1990
(Hörerschein für Rabatt beim Kauf)

W. M. Waite, L. R. Carter:
An Introduction to Compiler Construction,
Harper Collins, New York, 1993

W. M. Waite, G. Goos:
Compiler Construction,
Springer-Verlag, 1983

R. Wilhelm, D. Maurer:
Übersetzerbau - Theorie, Konstruktion, Generierung,
Springer-Verlag, 1992

A. Aho, R. Sethi, J. D. Ullman:
Compilers - Principles, Techniques and Tools,
Addison-Wesley, 1986

A. W. Appel:
Modern Compiler Implementation in C,
Cambridge University Press, 1997
auch in Java und in ML

Was übersetzt ein Übersetzer?

Ein **Übersetzer** transformiert korrekte Sätze einer **Quellsprache** in **gleichbedeutende** Sätze einer **Zielsprache**.

Quellsprache:	**Zielsprache:**
Programmiersprache	**Maschinensprache**
C++	Sparc-Code
Programmiersprache	**Abstrakte Maschine**
Java	Java-Byte-Code
Programmiersprache	**Programmiersprache**
Source-to-Source	
C++	C
Anwendungssprache	**Anwendungssprache**
LaTeX	HTML
DB-Sprache (SQL)	DB-Systemaufrufe

Ein **Interpretierer** führt Sätze einer Quellsprache aus.

Lisp

Shell-Kommandosprache

Java-Byte-Code

Bedeutungserhaltende Übersetzung

Ein **Übersetzer** transformiert **korrekte Sätze** einer **Quellsprache** in **gleichbedeutende** Sätze einer **Zielsprache**.

$$\begin{tikzcd}
\text{Quellsprache} \arrow[r, "\text{Sprachdefinition}"] \arrow[d, "\text{Übersetzung}"'] & \text{Bedeutung beschrieben für abstrakte Maschine} \arrow[d, "\text{gleiche Ergebnisse}"] \\
\text{Zielsprache} \arrow[r, "\text{Maschinenbeschreibung}"'] & \text{Ausführung}
\end{tikzcd}$$

Die **Bedeutung** ist nur für **korrekte** Programme definiert.
Aufgabe: Fehlerbehandlung.

Der Übersetzer ermittelt **statische** Programmeigenschaften zur **Übersetzungszeit,** z. B. **übersetzbar**.

Dynamische Eigenschaften werden zur **Laufzeit** des Programms ermittelt, z. B. **ausführbar**.

Sprachdefinition - Übersetzeraufgaben

- **Notation der Grundsymbole**

(Wortsymbole, Bezeichner, Literale)

formal definiert: reguläre Ausdrücke

Aufgabe: lexikalische Analyse

- **syntaktische Struktur**

formal definiert: kontextfreie Grammatik

Aufgabe: syntaktische Analyse

- **statische Semantik**

(Definition von Namen für Objekte, Typregeln)

meist verbal definiert

Aufgaben: semantische Analyse, Transformation

- **dynamische Semantik**

(Bedeutung, Wirkung von Sprachkonstrukten)

meist verbal definiert, in Termen einer abstrakten Sprachmaschine

Aufgaben: Transformation, Code-Generierung

- **Definition der Zielsprache (-maschine)**

Aufgaben: Transformation, Code-Generierung, Assemblierung

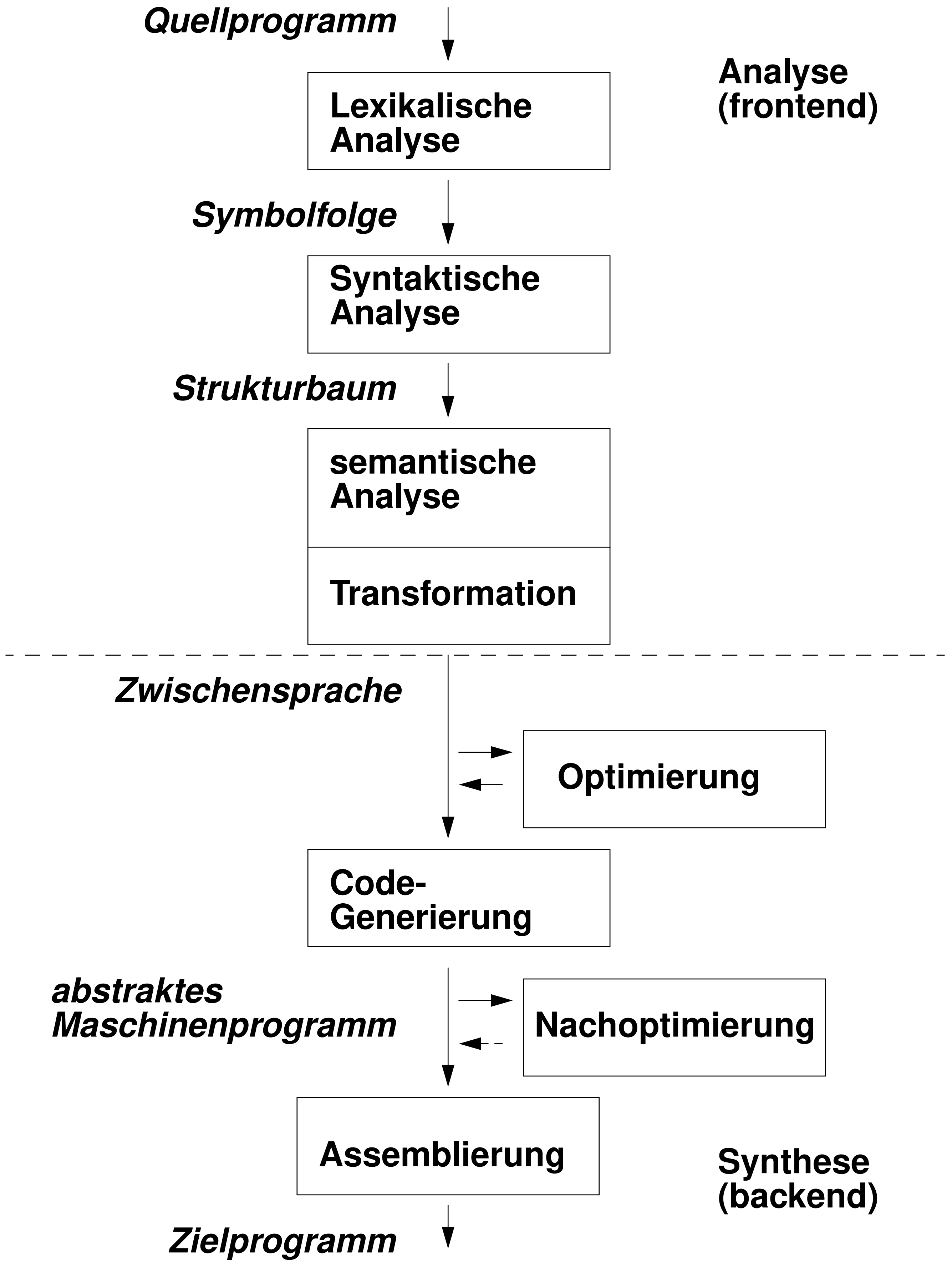
Übersetzeraufgaben

Strukturierung	Lexikalische Analyse	Symbol-erkennung Konversion
	Syntaktische Analyse	Zerteilung Baumaufbau
Translation	Semantische Analyse	Bezeichner-analyse Typ-analyse
	Transformation	Daten-abbildung Operations-abbildung
Kodierung	Code-Generierung	Ausführungs-reihenfolge Register-zuteilung Instruktions-Auswahl
	Assemblierung	Instruktions-kodierung Interne Adressierung Externe Adressierung

Zielmaschine - Übersetzeraufgaben

Eigenschaften von Zielprozessoren	**Aufgaben der Kodierung**
Instruktionssatz RISC - CISC	Instruktionsauswahl
Instruktionsformat 2-, 3-Adress	Instruktionskodierung Registerzuteilung
Register Klassen, Anzahl,	Registerzuteilung Ausführungsreihenfolge
Speicheradressierung Ausrichtung, Adressbildung	Datenabbildung interne Adressierung Instruktionsauswahl Instruktionskodierung
Instruktionsausführung Parallelverarbeitung	Ausführungsreihenfolge

Übersetzerstruktur und Schnittstellen



Übersetzerqualitäten

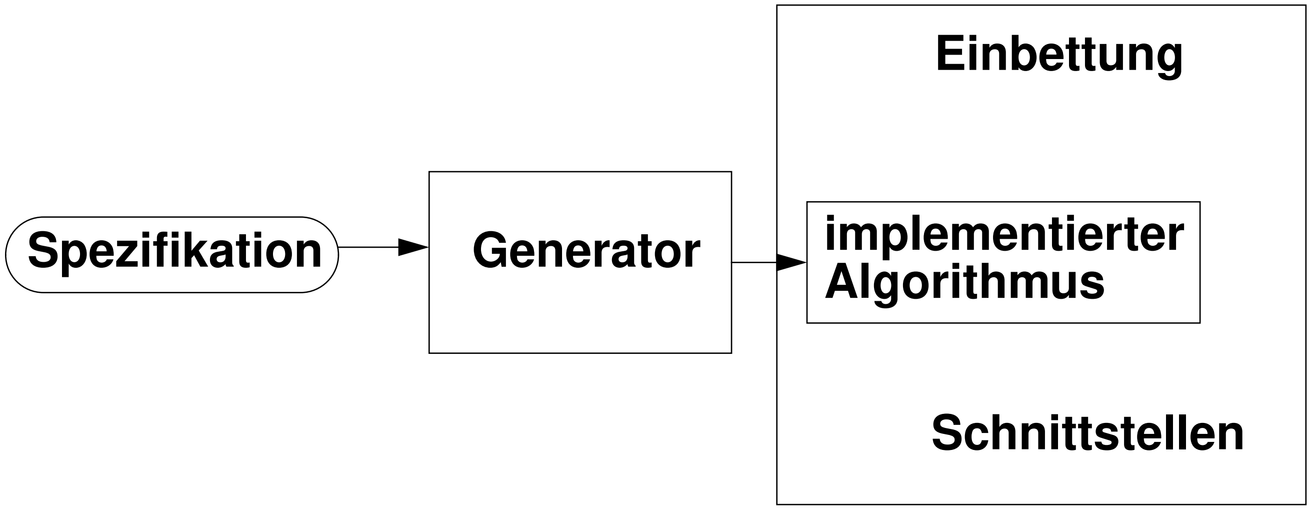
- **Korrektheit:**
  richtige Programme richtig übersetzen,
  falsche Programme zurückweisen
- **Effizienz:**
  Speicher- und Zeitbedarf des Übersetzers
- **Code-Effizienz:**
  Speicher- und Zeitbedarf des Ziel-Codes
  Aufgabe: Optimierung
- **Benutzerfreundlichkeit:**
  Aufgabe: Fehlerbehandlung
- **Robustheit:**
  jede Eingabe wird bewältigt

Strategien zur Übersetzerkonstruktion

- **Sprachdefinition exakt befolgen**
- **generierende Werkzeuge einsetzen**
- **Standard-Module verwenden**
- **Standard-Methoden anwenden**
- **Validieren (Test-Suite)**
- **Verifizieren**

Generatoren

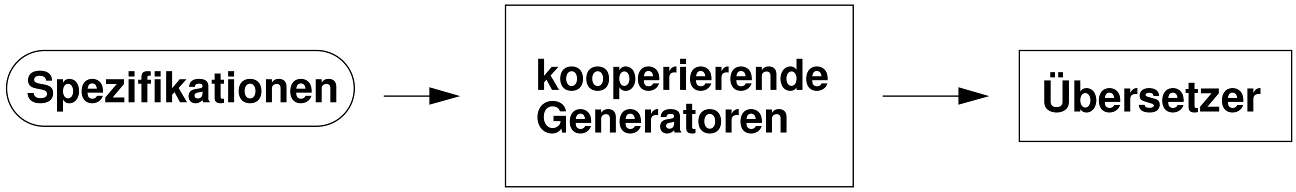
Schema:



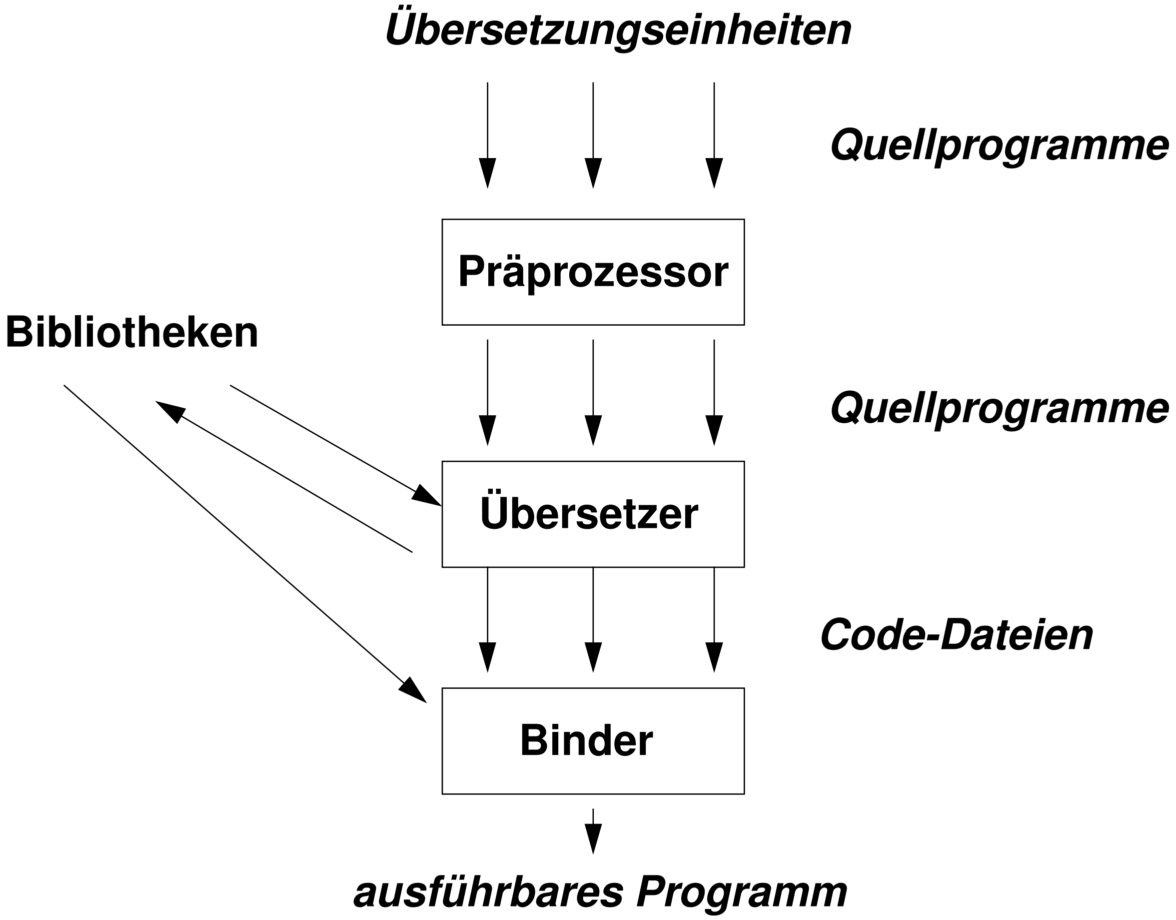
typische Generatoraufgaben:

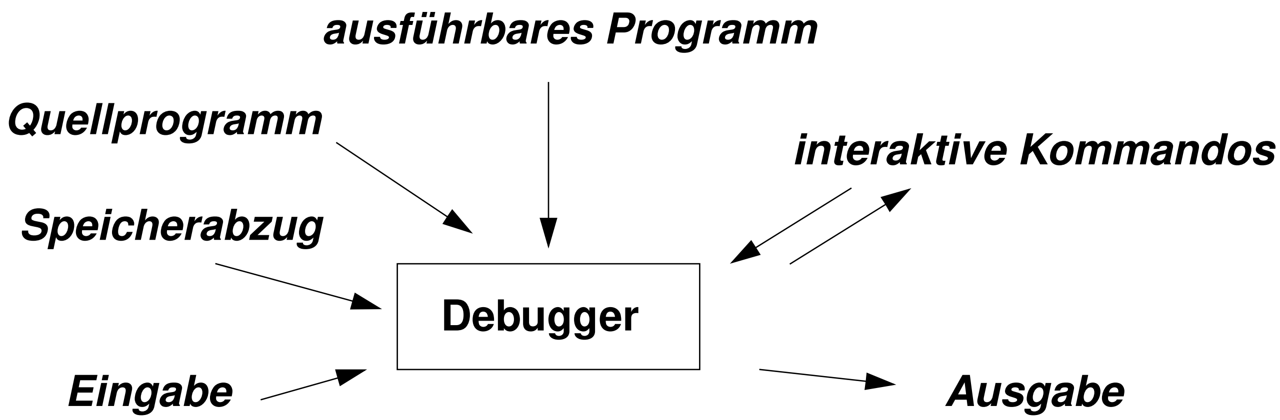
reg. Ausdrücke	Scanner-Generator	endl. Automat
kontextfreie Gram.	Parser-Generator	Kellerautomat
attributierte Gram.	Attributauswerter-Generator	Baumdurchlauf Berechnungen
Code-Muster	Code-Auswahl-Generator	Pattern-Matching

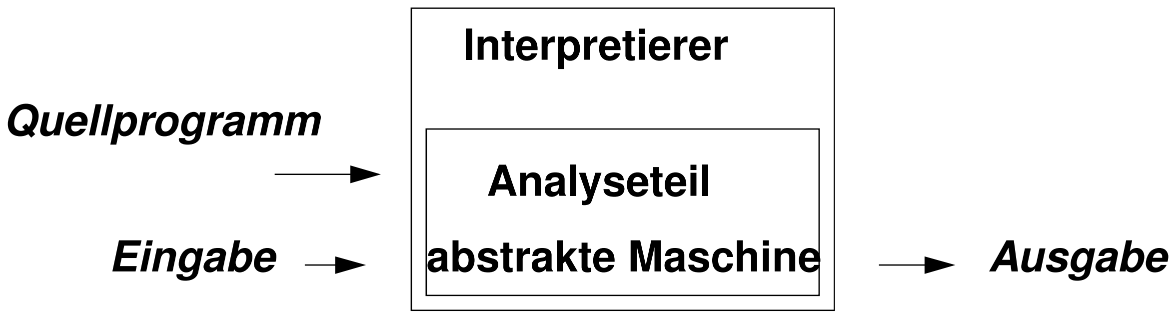
integriertes System Eli:



Umgebung des Übersetzers







Lexikalische Analyse

Eingabe: ***Programm als Zeichenfolge***

Aufgaben:	**Übersetzermodul:**
	Eingabeverarbeitung
Erkennen und klassifizieren der Symbole	Scanner (zentrale Phase, endlicher Automat)
Ausblenden bedeutungsloser Zeichen	
Kodieren der Symbole	Bezeichnermodul
Speichern von Symbolinformation	Literalmodule
	Zeichenspeicher
Konversion	

Ausgabe: ***Programm als kodierte Symbolfolge***

Symbolrepräsentation:

Syntax-Code	**Attribut**	**Quellposition**
Terminal der konkreten Syntax	Wert oder Referenz auf Datenmodul	für Fehlermeldungen in späteren Übersetzerphasen

Spezifikation von Grundsymbolen

reguläre Grammatik ⟷ **regulärer Ausdruck**

Syntax-diagramm

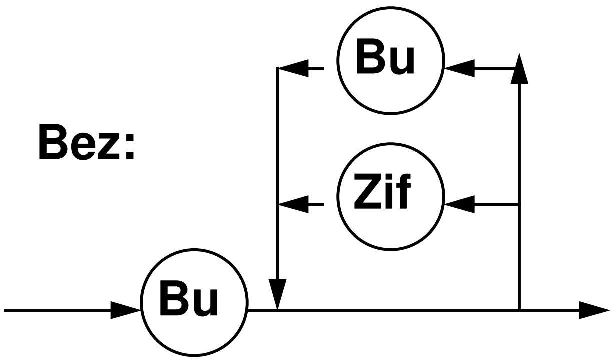
Akzeptormodell: **endlicher Automat**

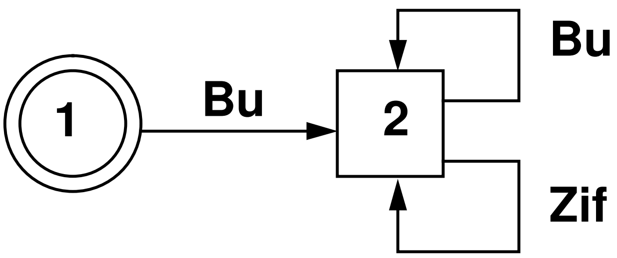
Beispiel Bezeichner:

Bez ::= Bu X
X ::= Bu X
X ::= Zif X
X ::=

Bez = Bu (Bu | Zif)*

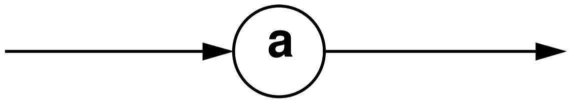
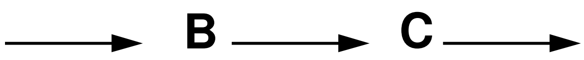
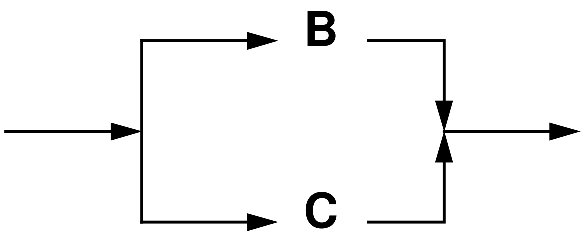
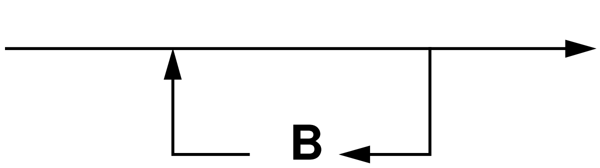
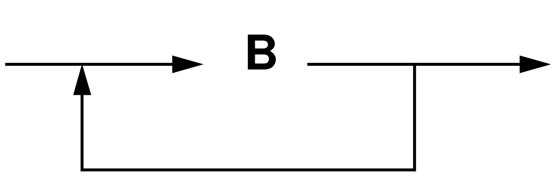
Bez:





Reguläre Ausdrücke in Syntaxdiagramme

Transformationsregeln:

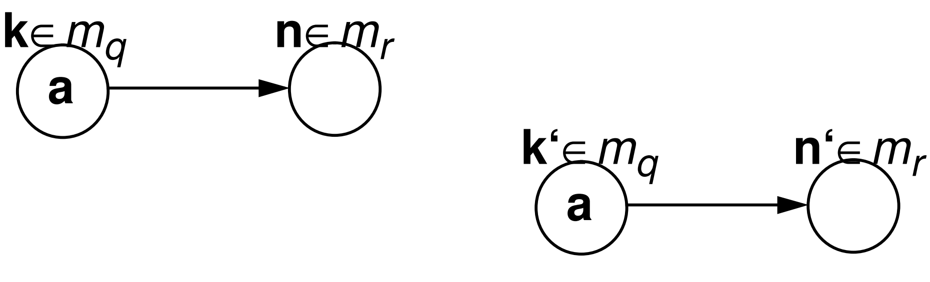
regulärer Ausdruck A	Syntaxdiagramm für A
leer	
a	
B C	
B | C	
B*	
B+	

Automat aus Syntaxdiagramm konstruieren

Syntaxdiagramm	**Automat**
Knoten, Kanten	Übergänge, Zustände
Knotenmenge m_q	**Zustand** q
unter gleichem Zeichen *a* verbundene Knotenmengen m_q und m_r	Übergänge q ---> r mit Zeichen *a*

Konstruktion:

1. **Knoten numerieren**
   Diagrammausgang = *0*
2. **Anfangsmenge** m_1 — **Anfangszustand** *1*
   enthält alle vom Anfang erreichbaren Knoten
3. **konstruiere neue Knotenmengen** und **Übergänge**
   für ein Zeichen *a* und eine Menge m_q mit Knoten *k*
   bilde Knotenmenge m_r mit allen Knoten *n*, wobei gilt



4. **Schritt 3 wiederholen** bis keine neuen Knotenmengen
5. Alle Zustände *q* sind **Endzustand** gdw. *0* in m_q ist.

Teilautomaten zusammensetzen

- **Anfangszustände der Teilautomaten identifizieren**

  ggf. auch Übergänge unter gleichem Zeichen identifizieren

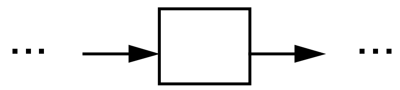
- **unterschiedliche Endzustände beibehalten**

  klassifizieren die Symbole

- **Teilautomaten für Kommentare und bedeutungslose Zeichen zufügen**

Übergänge aus Endzuständen:

wann hält der Automat?



Regel des längsten Musters:

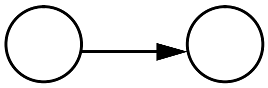
- **Automat hält in beliebigem Zustand, falls kein Übergang mit nächstem Zeichen möglich**
- **setzt zurück auf zuletzt durchlaufenen Endzustand**
- **Fehler, falls kein Endzustand durchlaufen**

Konsequenzen für die Trennung aufeinanderfolgender Symbole!

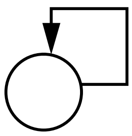
An der Sprachdefinition und der Syntax prüfen!

Scanner: Implementierungsaspekte

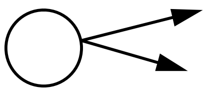
- **Laufzeit proportional zur Zahl der Zeichen des Programms**
- **Operationen pro Zeichen schnell - sonst dominiert der Scanner die Übersetzungszeit**
- **Tabellen-gesteuerte Automaten zu langsam**
  Schleife, 2-fache Array-Indizierung, Verzweigung
- **direkt programmierter Automat schneller**
  Übergänge in Programmstrukturen umsetzen:



Sequenz



Schleife



Verzweigung
o. Sprung

- **schnelle Schleife für bedeutungslose Zwischenräume**
- **Klassen gleichartiger Zeichen: Bitmuster** oder **Indizierung -** statt teurer Mengenoperationen
- **Zeichen nicht aus Eingabepuffer kopieren, sondern**
  Zeiger in Eingabepuffer

Bezeichner- und Literalmodule

- **einheitliche Schnittstelle:**

  Eingabeparameter: Zeiger auf Symboltext

  Ausgabeparameter: Attribut, Syntax-Code

- **Bezeichnermodul kodiert Bezeichner bijektiv, erkennt Wortsymbole**

  Implementierung: Hash-Vektor, erweiterbare Tabelle,

  Kollisionslisten

- **Literalmodule**

  (Gleitpunktzahlen, ganze Zahlen, Zeichenreihen)

  Repräsentation zur Speicherung:

  Quellsymbol,

  Wert konvertiert in Übersetzerdatum

  Wert konvertiert in Zieldarstellung

  Vorsicht:

  Überlauf bei Konversion abfangen!

  Übersetzerdatum evtl. verschieden von Zieldarstellung!

- **Zeichenspeicher**

  speichert Zeichenfolgen ohne Längenbegrenzung

  wird von den übrigen Modulen benutzt

Scanner-Generatoren

generieren die zentrale Funktion der lexikalischen Analyse

GLA	University of Colorado, Boulder; Komponente von Eli
Lex	Unix Standardwerkzeug
Flex	Nachfolgesystem zu Lex
Rex	GMD Karlsruhe

Symbolspezifikation: reguläre Ausdrücke

GLA	+ programmierte Teilautomaten
	+ Muster für vordefinierte Symbole
Lex, Flex, Rex	+ programmierbare Zustandsübergänge

Schnittstellen:

GLA	wie in diesem Kapitel beschrieben,
	kooperiert mit anderen Eli-Komponenten
Lex, Flex, Rex	Aktionen zu reg. Ausdrücken für Symbole
	(beliebige Anweisungen der Impl.sprache),
	Schnittstelle zu Parser-Generator Yacc

Implementierung:

GLA	direkt programmierter Automat in C
Lex, Flex, Rex	Tabellen-gesteuerter Automat in C
Rex	auch in Modula-2
Flex, Rex	schneller, kleiner als bei Lex

4. Syntaktische Analyse

Eingabe: Symbolfolge

Aufgaben:

Ableitung bilden mit konkreter Grammatik (parsing)

Strukturbaum aufbauen mit abstrakter Grammatik

Fehlerbehandlung

Ausgabe: Strukturbaum

Übersetzermodul Parser (dt. Zerteiler):

Kellerautomat; hier nur deterministische

kann bei Anwenden einer Produktion Aktion ausführen

(Baumknoten erzeugen)

spezifiziert durch konkrete KFG mit Aktionen

Zielbezogene Parser:

Linksableitung; Baum top-down oder bottom-up

Quellbezogene Parser:

Rechtsableitung rückwärts; Baum bottom-up

Strukturbaum (vergröberter Ableitungsbaum):

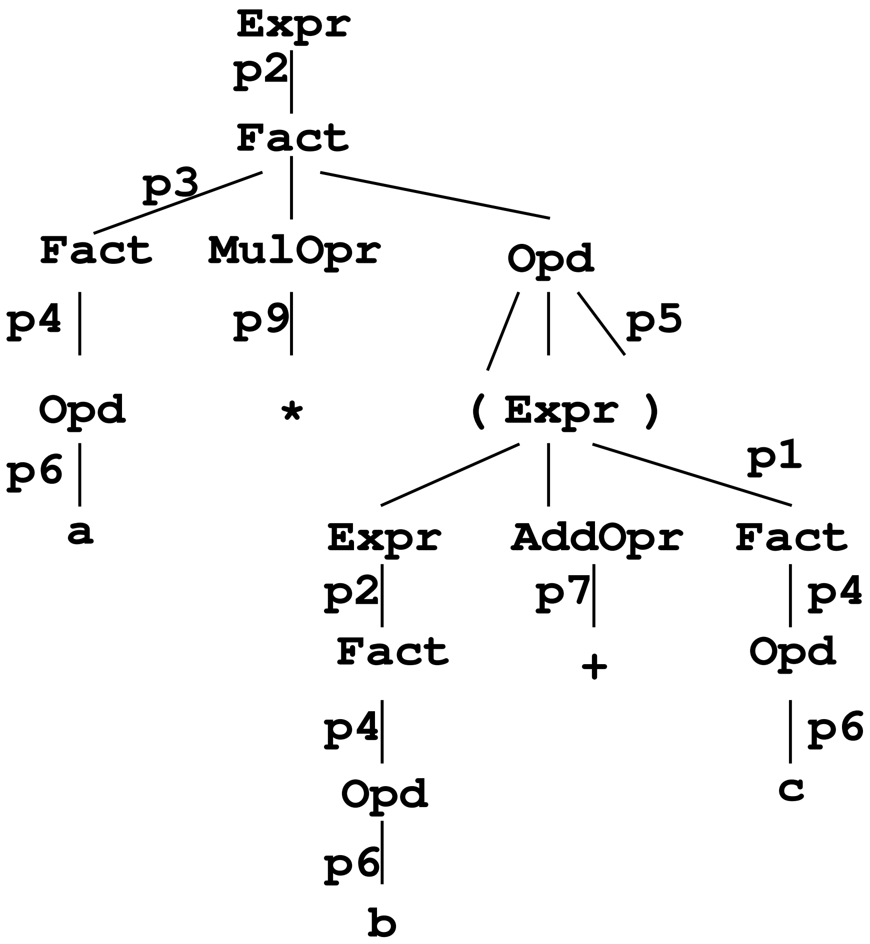
als Datenstruktur für Translationsphase, oder

implizit, Aufrufe von Operationen der Translationsphase

Beispiel: Ausdrucksgrammatik

Name	Produktion			Aktion
p1:	Expr	::=	Expr AddOpr Fact	BinEx
p2:	Expr	::=	Fact	
p3:	Fact	::=	Fact MulOpr Opd	BinEx
p4:	Fact	::=	Opd	
p5:	Opd	::=	'(' Expr ')'	
p6:	Opd	::=	Ident	IdEx
p7:	AddOpr	::=	'+'	PlusOpr
p8:	AddOpr	::=	'-'	MinusOpr
p9:	MulOpr	::=	'*'	TimesOpr
p10:	MulOpr	::=	'/'	DivOpr

Ableitungsbaum für `a * (b + c)`



Konkrete und abstrakte Syntax

konkrete Syntax	**abstrakte Syntax**
KFG	KFG
definiert Struktur der Quellprogramme	definiert Strukturbäume
eindeutig	i.a. mehrdeutig
spezifiziert Parser, Ableitung	Grundlage für Translationsphase
Aktionen spezifizieren	Baumaufbau
Kettenproduktionen `Expr ::= Fact`	unnötige weglassen
haben keine Aktion	
beteiligte Symbole	zu Symbolklasse vereinigt `Exp = {Expr, Fact}`
gleiche Aktion zu strukturgleichen Produktionen:	

```
Expr ::= Expr AddOpr Fact &BinEx
Fact ::= Fact MulOpr Opd &BinEx
```

Terminale	unnötige weglassen, keine Baumknoten

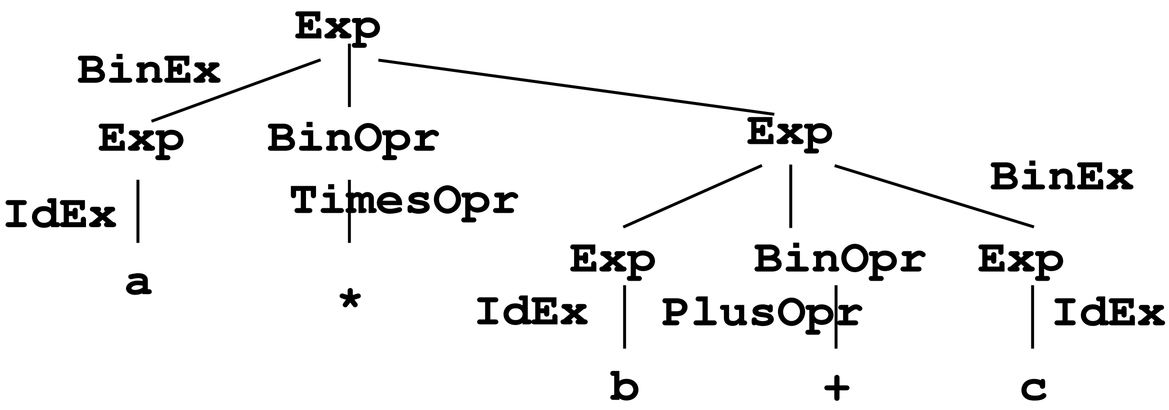
Aus konkreter Syntax und Symbolklassen sind die abstrakte Syntax und die Aktionen automatisch generierbar.

Beispiel: abstrakte Ausdrucksgrammatik

Name Produktion

```
BinEx:     Exp    ::= Exp BinOpr Exp
IdEx:      Exp    ::= Ident
PlusOpr:   BinOpr ::= '+'
MinusOpr:  BinOpr ::= '-'
TimesOpr:  BinOpr ::= '*'
DivOpr:    BinOpr ::= '/'
```

Strukturbaum für `a * (b + c)`



Symbolklassen:

```
Exp = { Expr, Fact, Opd }
BinOpr = { AddOpr, MulOpr }
```

Aktionen der konkreten Syntax
Produktionen der abstrakten Syntax

keine Aktion an konkreter Produktion
kein Knoten im Strukturbaum

Recursive Descent Parser

dt.: rekursiver Abstieg; zielorientierte, prädiktive Methode

KFG in Menge von Funktionen transformieren:

Nichtterminal	Funktion
alternative Produktionen	Zweige im Rumpf
Nichtterminal auf rechter Seite	Aufruf
Terminal auf rechter Seite	akzeptiere Symbol und lies
Entscheidungsmenge zu Produktion	Verzweigungsentscheidung

Beispiel:

```
p1:  Stmt ::= Variable ':=' Expr
p2:       |   'while' Expr 'do' Stmt

void Stmt ()
{  switch (CurrSymbol)
   { case Entscheidungsmenge für p1:
            Variable ();
            accept (assignSym);
            Expr ();
            break;
     case Entscheidungsmenge für p2:
            accept (whileSym);
            Expr ();
            accept (doSym);
            Stmt ();
            break;
     default: Fehlerbehandlung;
}  }
```

Grammatikbedingung für Recursive Descent

Eine Grammatik ist **stark LL(1)**, wenn für alle Produktionenpaare mit gleicher linker Seite die Entscheidungmengen disjunkt sind:

	Entscheidungmengen
A ::= u	Anf (u Folge (A))
A ::= v	Anf (v Folge (A))

Anfangs- und Folgemengen:

Anf (u) := $\{t \in T \mid$ es gibt $u \Rightarrow^* tv\} \quad \cup \{\varepsilon\}$ falls $u \Rightarrow^* \varepsilon$ existiert

Folge (A) := $\{t \in T \mid$ es gibt $S \Rightarrow^* uAv$ und $t \in$ Anf (v)$\}$

Konsequenz:

Starke LL(1)-Grammatiken haben keine links-rekursiven Produktionen, und keine Produktionen mit gleichen Anfängen.

EBNF-Konstrukte, z.B.:

A ::= u (v)* w

in Funktion für A:

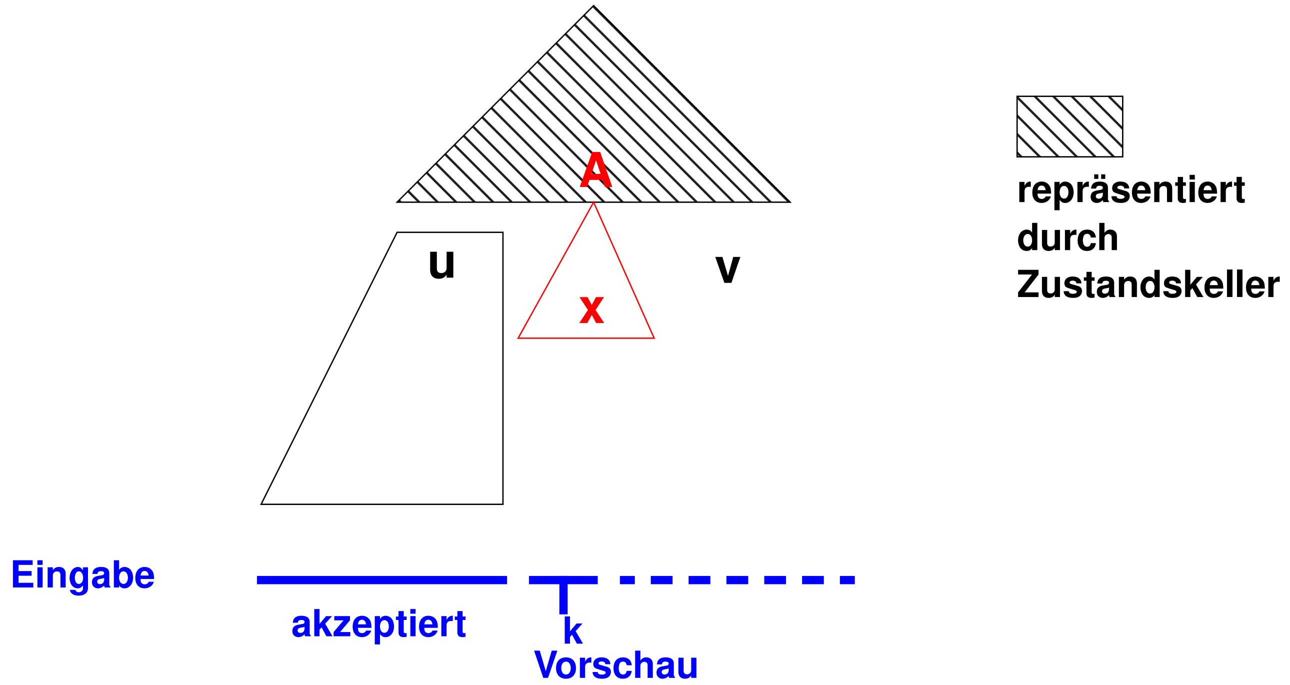
... u ... while (CurrSymbol in Anf (v)) { ... v ...} ...w ...

Es muß gelten: Anf (v) und Anf (w Folge (A)) sind disjunkt.

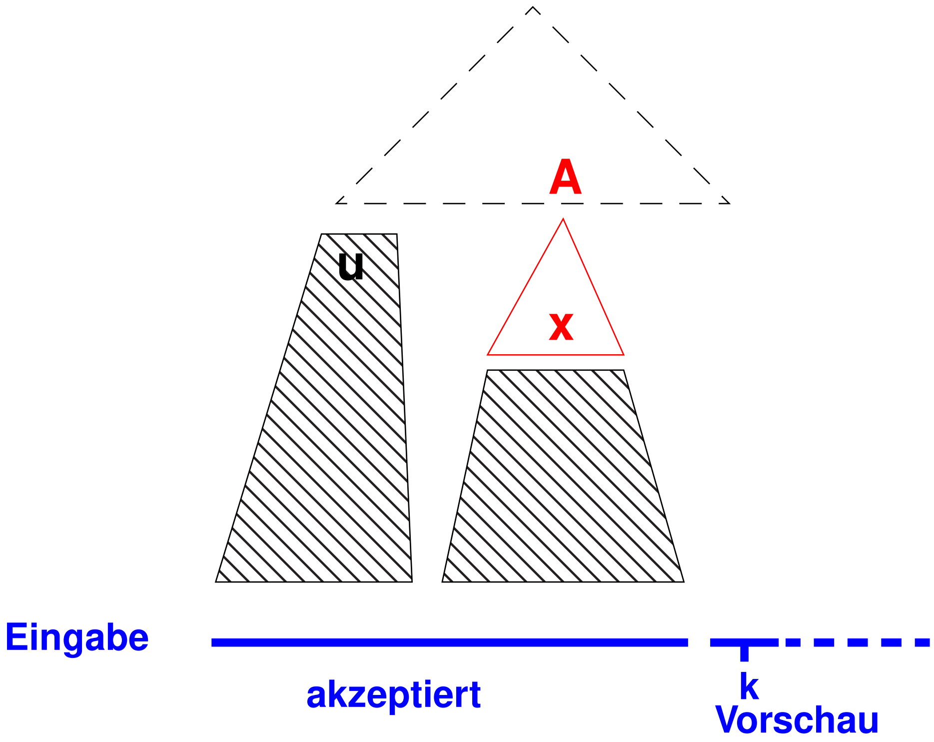
Vergleich: quellbezogen - zielbezogen

Information des Kellerautomaten bei der Entscheidung, die Produktion `A ::= x` anzuwenden.

zielbezogen, prädiktiv, Linksableitung:



quellbezogen, Rechtsableitung rückwärts:



LR(k)-Bedingung

Eine kontext-freie Grammatik ist eine LR(k)-Grammatik mit

$k \geq 0$, wenn für alle Paare von Rechtsableitungen

Z =>* u A w => u x w **u, u', x, x' $\in$ V***

Z =>* u' B w' => u' x' w' **w, w', v $\in$ T***

gilt:

falls es ein v gibt, so daß

u x v = u' x' w'

und die ersten k Zeichen von w und v gleich sind,

dann sind auch A = B, x = x', u = u'

(d. h. der letzte Ableitungsschritt ist gleich)

Diese Bedingung ist nicht konstruktiv und so i.a. nicht prüfbar!

Automat konstruieren, deterministisch g.d.w. LR(k).

Mächtigkeit der Grammatikklasse:

LL(k) => LR(k) => eindeutig

Praktische Bedeutung haben Unterklassen wie LALR(1).

LR(1)-Automat

Zustände werden gekellert.

Operationen:	**shift**	lesen und Zustand kellern
	reduce	mit Produktion reduzieren, Zustände entkellern

Ein **Zustand** repräsentiert eine **Menge von Situationen**.

Eine **Situation** repräsentiert den Analysestand bzgl. einer Produktion:

[A ::= u . v R]

Analyseposition **.**

R erwarteter **Rechtskontext**
Menge von Terminalen,
die folgen können, wenn mit
der Produktion reduziert ist.

Reduktionssituation:

[A ::= u v . R]

Automat reduziert,
falls nächstes Zeichen der Eingabe in R

Konstruktion von LR(1)-Automaten

Anfangszustand, Übergänge, und Zustände konstruieren bis keine neuen mehr entstehen.

Dabei auf jeden Zustand *Hüllenbildung* anwenden:

Wenn [A ::= u . B v R] im Zustand q ist, dann ist auch für jede Produktion B ::= w

[B ::= . w Anf (v R)] im Zustand q.

Anfangszustand:

Hülle von [S ::= . u {#}]

S ::= u ist eindeutige Startproduktion,

 # ist künstliches Schlußzeichen (eof)

Übergänge in Nachfolgezustände mit Symbolen x , die in Situationen auf die Analyseposition folgen, x Terminal oder Nichtterminal:

... A ::= u . x v R1 ... B ::= w . x z R2 ...	$\xrightarrow{x}$	A ::= u x . v R1 B ::= w x . z R2 *Hülle bilden*

Konflikte: Automat nicht determ., Grammatik nicht LR(1):

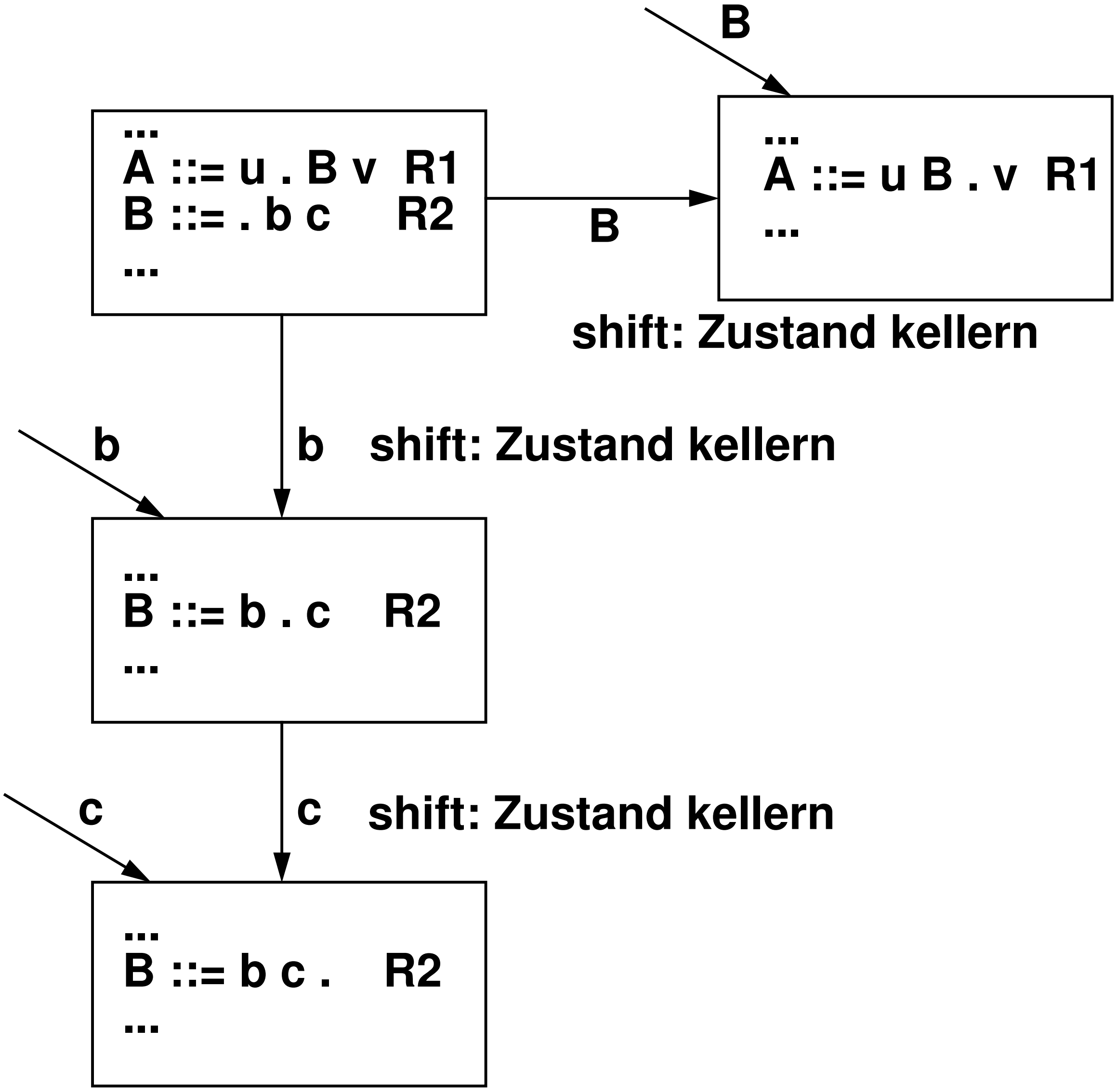
reduce / reduce

... A ::= u . R1 B ::= v . R2 ...	R1, R2 nicht disjunkt

shift / reduce

... A ::= u .t v R1 B ::= w . R2 ...	t $\in$ R2

Operationen des LR(1)-Automaten schematisch



reduziere B ::= b c , falls Eingabe in R2
entkellere 2 Zustände (Länge rechte Seite)
Übergang mit B (linke Seite)

Fehler: kein Übergang möglich

Halt: reduziere Startproduktion bei #

Vereinfachte LR-Klassen

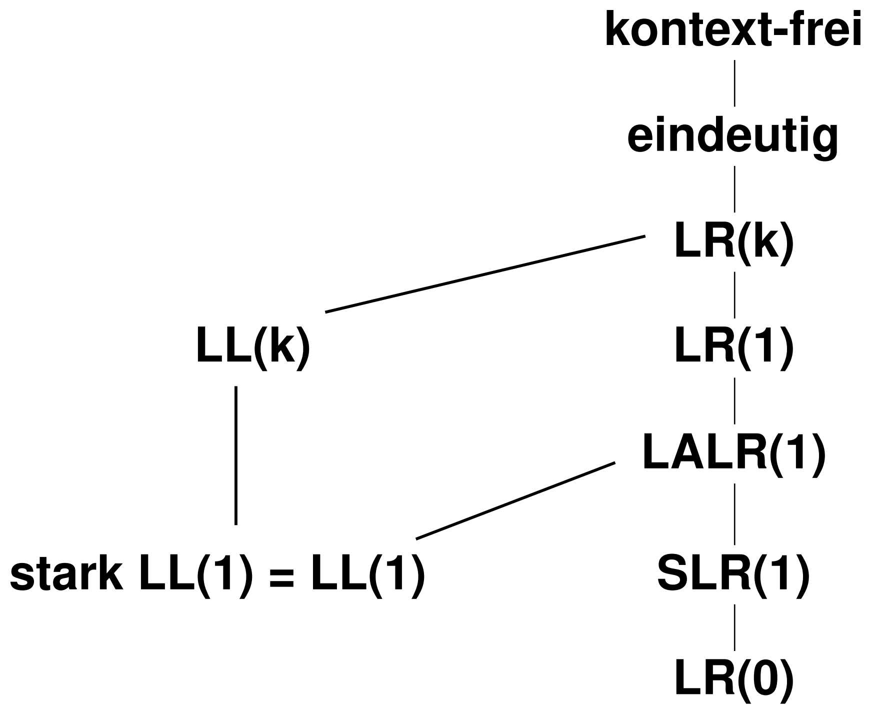
LR(1): zuviele Zustände für praktischen Einsatz
Ursache: Rechtskontexte differenzieren stark
Abhilfe: Rechtskontexte vereinfachen, weniger Zustände, Grammatik weniger mächtig

LR(0): Alle Situationen **ohne** Rechtskontext
Konsequenz: Reduktionssituationen einzeln im Zustand

SLR(1): **LR(0)-Zustände**; bei Reduktionssituationen vergröberten Rechtskontext zur Entscheidung:
[A ::= u . Folge (A)]

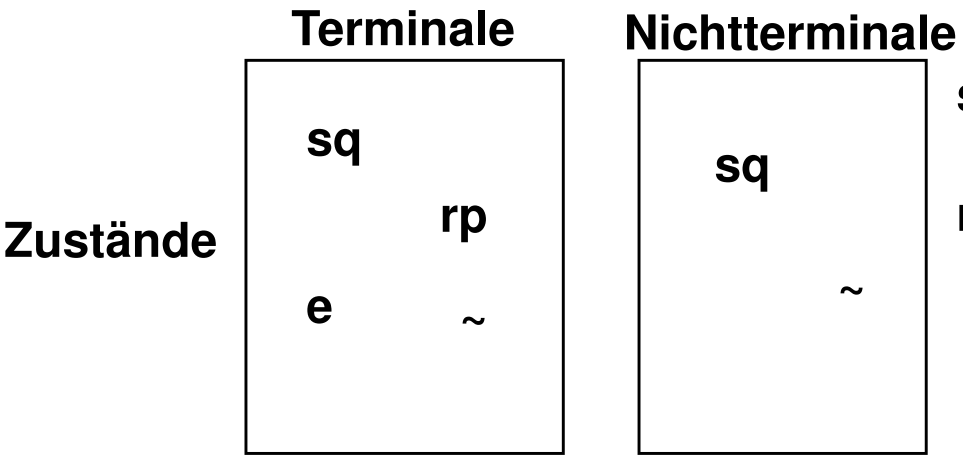
LALR(1): LR(1)-Zustände zusammenlegen, wenn sich ihre Situationen nur im Rechtskontext unterscheiden; liefert **LR(0)-Zustände** mit "exaktem" Rechtskontext; **wichtigste Parser-Generatoren**

Grammatik-Hierarchie:



Implementierung von LR-Automaten

Tabellen-gesteuert:
(direkt programmmiert möglich, aber zu groß)



sq: shift in Zustand q
rp: reduce Produktion p
e: Fehler
~: nie erreicht

Tabellen komprimieren:

- **verträgliche Zeilen / Spalten zusammenlegen**
  Graphfärbung
- **separate Fehlermatrix für Terminaltabelle (Bit-Matrix)**
  erhöht die Zahl verträglicher Zeilen / Spalten
- **kleinsten Wert von Zeilen / Spalten subtrahieren**
  erhöht die Zahl verträglicher Zeilen / Spalten
  verkleinert Wertebereich der Einträge
- **LR(0)-Reduktionszustände streichen, neue Operation im Vorgängerzustand: shift - reduce**
  eliminiert ca. 30% in der Praxis

Insgesamt Tabellenreduktion auf ca. 10-20% möglich.

Behandlung syntaktischer Fehler

Allgemeine Kriterien:

- **Fehler so früh wie möglich erkennen**
- **Symptom quellbezogen melden**
- **Parsing möglichst kurz nach Fehlerstelle fortsetzen**
- **Folgefehler vermeiden**
- **strukturell korrekter Baum trotz Fehler**
- **kein Zurücksetzen, Zurücknehmen von Aktionen**
- **möglichst kein zusätzlicher Aufwand für korrekte Programme**

LL- und LR-Verfahren erkennen Fehler frühest möglich.

Begriffe: **Fehlerstelle t, korrekter Präfix w:**

Eingabe	$w\ t\ x \in T^*$	
korrekter Präfix	w	d.h. es gibt ein $u \in T^*$ mit $w\,u \in L(G)$
	$w\ t$	ist kein korrekter Päfix

Parser akzeptiert t nicht.

Aufsetzpunkt d für die Fortsetzung des Parsing:

$$w\ t\ x \;=\; w\ y\ d\ z$$

$\downarrow$ ersetzen (konzeptionell)

$w\ v\ d\ z$ mit $w\ v\ d$ ist korrekter Präfix

Methode: simulierte Fortsetzung

anwendbar bei LL- und LR-Verfahren (mit kopierbarem Keller)

Idee: Aus Fehlerstelle und Keller Menge von Symbolen als mögliche Aufsetzpunkte bestimmen.

Verfahren:

1. **bei Fehler Keller speichern**
   Fehlersymbol überlesen
2. **Menge D möglicher Aufsetzpunkte bestimmen**
   Automat zuende laufen lassen (Modifikation siehe unten),
   Symbole nicht aus der Eingabe lesen, sondern
   in D einfügen
3. **einen Aufsetzpunkt finden**
   Eingabe überlesen bis Symbol in D
4. **Aufsetzpunkt erreichen**
   gespeicherter Keller wird wieder Keller des Automaten,
   Übergänge bis der Aufsetzpunkt akzeptabel ist,
   dabei Symbole einfügen statt aus Eingabe lesen.
   Damit ist ein korrekter Präfix konstruiert.
5. **Automat normal fortsetzen**

Automatenkonstruktion für Schritt 2 erweitern:

Automat muß Keller leeren und terminieren,
dazu in jedem Zustand ein zulässiges Symbol für
Übergang auswählen
(automatisch generierbar)

Parser-Generatoren

PGS	Univ. Karlsruhe; in Eli	**LALR(1), Tabellen-gest.**
Cola	Univ. Paderborn; in Eli	**LALR(1), Tab. o. dir. progr.**
Lalr	Univ. / GMD Karlsruhe	**LALR(1), Tabellen-gest.**
Yacc	Unix-Werkzeug	**LALR(1), Tabellen-gest.**
Bison	Gnu	**LALR(1), Tabellen-gest.**
Llgen	Amsterdam Comp. Kit	**LL(1), recursive descent**
Deer	Univ. Col. Bouder	**LL(1), recursive descent**

Spezifikationsform:

EBNF: Cola, PGS, Lalr
BNF: Yacc, Bison

Fehlerbehandlung:

simulierte Fortsetzung (automatisch): Cola, PGS, Lalr
Fehlerproduktionen (manuell): Yacc, Bison

Aktionen:

Anweisungen in Implementierungssprache
am Ende von Produktionen: Yacc, Bison
auch innerhalb von Produktionen: Cola, PGS, Lalr

Konfliktlösung:

Einfluß auf Zustände (reduziere bei ...)Cola, PGS, Lalr
Reihenfolge von Produktionen: Yacc, Bison
Präzedenz und Assoziativität: Yacc, Bison

Implementierungssprache:

C: Cola, Yacc, Bison
C, Pascal, Modula-2, Ada: PGS, Lalr

5. Semantische Analyse und Transformation

Eingabe: Strukturbaum

Aufgaben:	**Übersetzermodul:**
Namensanalyse	Umgebungsmodul
Eigenschaften von Programmobjekten	Definitionsmodul
Typanalyse, Operatoridentifikation	Signaturmodul
Transformation	Baumgenerator

Ausgabe: Zielbaum, Zwischen-Code
Zielprogramm bei Source-to-Source

Standardimplementierungen und Generatoren für Übersetzermodule

Operationen der Übersetzermodule werden an Knoten des Strukturbaums aufgerufen

Modell: abhängige Berechnungen im Baum

Spezifikation: **attributierte Grammatiken**

generiert: Baumdurchlauf mit Aufrufen der Operationen

Definitionsmodul

zentrale Datenstruktur

ordnet Programmobjekten z*. B. Typen, Variablen, ...*
Eigenschaften zu
z. B. T*yp einer Variablen, Elementtyp eines Array-Typs.*

Objekte werden durch *Schlüssel (key)* identifiziert.

Operationen:

NewKey ( )	liefert neuen Schlüssel
ResetP (k, v)	setzt zum Schlüssel k die Eigenschaft P mit Wert v
SetP (k, v, d)	setzt (ersetzt) zum Schlüssel k die Eigenschaft P mit Wert v (bzw. d)
GetP (k, d)	liefert Wert der Eigenschaft P des Schlüssels k; liefert d, falls P zu k nicht gesetzt ist

Aufrufe: abhängige Operationen im Baum

Implementierung: z. B. Liste von Eigenschaften zu jedem Schlüssel

Generierung des Definitionsmoduls:
Aus Spezifikationen

Eigenschaftsname: Eigenschaftstyp;

werden Funktionen Reset, Set, Get generiert.

Namensanalyse (Bezeichneridentifikation)

Bezeichner benennen (statische) Programmobjekte.

Definition für Bezeichner b führt neues Programmobjekt ein und bindet es an den Bezeichner.

Die Bindung gilt in einem bestimmten Bereich des Programms: **Gültigkeitsbereich** der Definition.

Aufgabe: jedem Auftreten eines Bezeichners den Schlüssel des Programmobjektes zuordnen. *(konsistente Umbenennung)*

Grundlage: Gültigkeitsregeln der Sprache *(scope rules)*

Verdeckungsregeln für Sprachen mit geschachtelten Strukturen:

- **Algol-Regel:** Die Definition eines Bezeichners b gilt im **ganzen** kleinsten sie umfassenden Abschnitt, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b.

- **C-Regel:** Die Definition eines Bezeichners b gilt im kleinsten sie umfassenden Abschnitt **von der Definitionsstelle an**, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b von der Definitionsstelle an**.**

Abschnitt je nach Sprache: Block. Prozedur, Modul, ...

Implementierung: Operationen des **Umgebungsmoduls** in Baumkontexten aufrufen.

Umgebungsmodul

implementiert den abstrakten Datentyp **Environment**:

hierarchisch geschachtelte Mengen von Bindungen
(Bezeichner, Schlüssel)

Funktionen:

NewEnv () erzeugt ein neues Environment e als Wurzel.

NewScope (e1) erzeugt neues Environment e2.
Alle Bindungen aus e1 sind auch in e2, falls sie nicht durch Bindungen in e2 verdeckt werden.

DefineIdn (e, b) fügt Bindung (b, k) in e ein,
falls sie dort noch nicht exisitiert;
k ist dann ein neuer Objektschlüssel.
Liefert in jedem Fall den an b gebundenen Schlüssel k.

KeyInEnv (e, b) liefert den Schlüssel k eines Paares (b, k) aus e (einschließlich aller e_i, aus denen e erzeugt wurde).
Liefert NoKey, falls kein solches Paar existiert.

KeyInScope (e, b) liefert den Schlüssel eines Paares (b, k), falls unmittelbar in e enthalten, NoKey sonst.

Umgebungsoperationen im Baum

Operationen in Baumkontexten und Reihenfolge ihrer Ausführung modelliert Gültigkeitsregeln.

Wurzelkontext `Root`:

```
Root.Env = NewEnv ();
```

Abschnitte `Range`, die Definitionen enthalten können:

```
Range.Env =
  NewScope(INCLUDING (Range.Env, Root.Env));
           nächst umgebende Range oder Root
```

definierendes Auftreten eines Bezeichners `IdDefScope`:

```
IdDefScope.Key =
  DefineIdn(INCLUDING Range.Env,
            IdDefScope.Symb);
```

angewandtes Auftreten eines Bezeichners `IdUseEnv`:

```
IdUseEnv.Key =
  KeyInEnv (INCLUDING Range.Env,
            IdUseEnv.Symb);
```

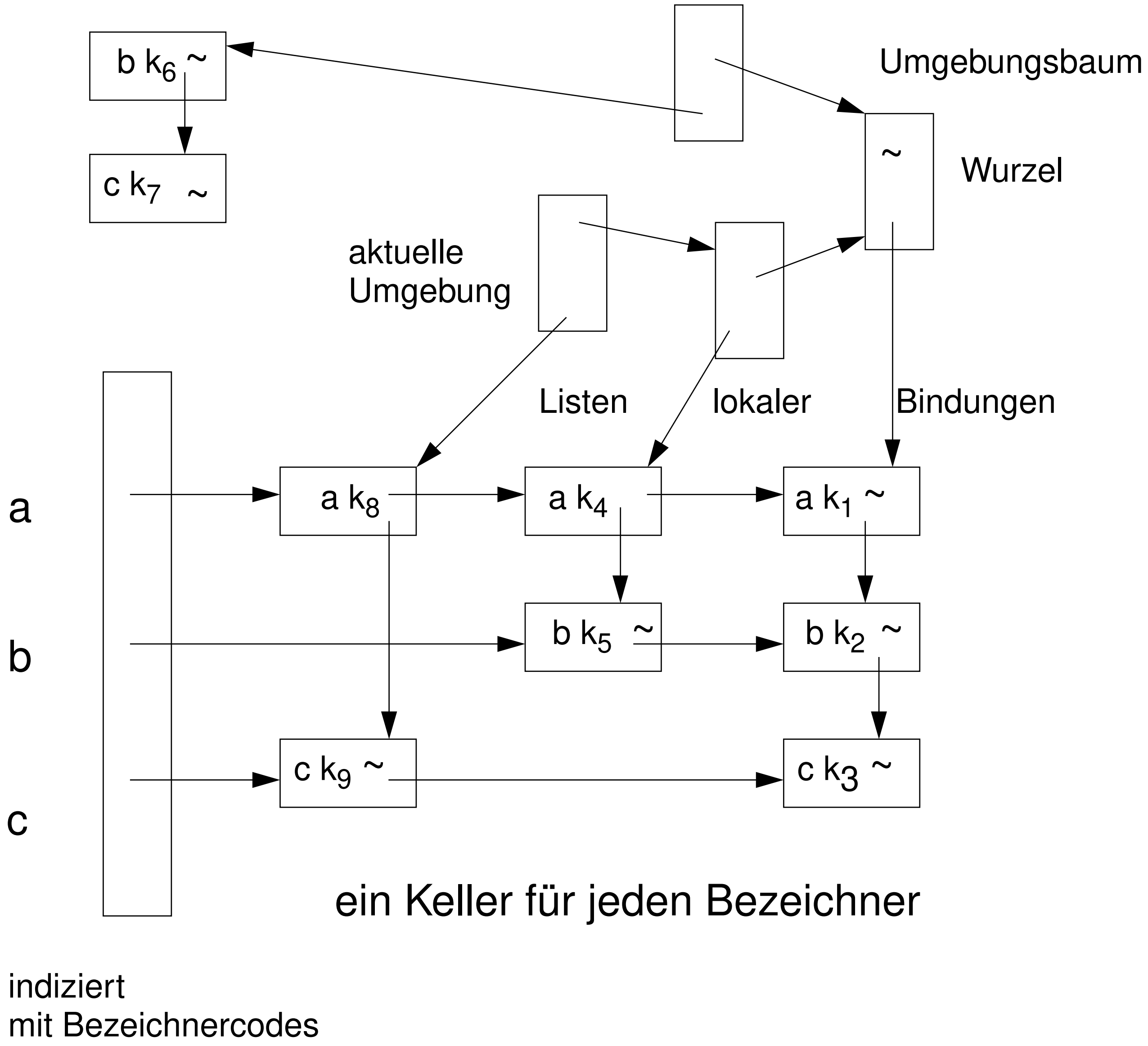
Vorbedingungen je nach Gültigkeitsregeln:
Algol-Regel: Vorbedingung für `KeyInEnv`:
alle `DefineIdn` aller umgebenden Ranges sind ausgeführt.

C-Regel: `KeyInEnv` und `DefineIdn` werden in der Reihenfolge des Programmtextes ausgeführt (links-abwärts).

weitere Prüfungen an dem Ergebnis der Bez.identifikation:

- keine Anwendung ohne Definition
- keine 2 Definitionen für einen Bezeichner in einem Range
- keine Anwendung vor der Definition (Pascal)
- Vorwärtsdefinitionen
- ...

Umgebungsmodul: Datenstruktur



k_i: Schlüssel des definierten Objektes

Operationen:

enter (e) lokale Definitionen von e kellern

leave (e) lokale Definitionen von e entkellern

KeyInEnv hat <u>kostanten</u> Aufwand!

Typanalyse: Aufgaben

Bestimmung und Prüfung von Typen zur Übersetzungszeit

- **Definierte Objekte** (z. B. Variable)
  haben Typ als Eigenschaft (im Definitionsmodul speichern)

- **Programmkonstrukte** (z. B. Ausdruck, Indizierung)
  haben Typ als Attribut des Baumknotens
  Spezielle Aufgabe: überladene Operatoren identifizieren

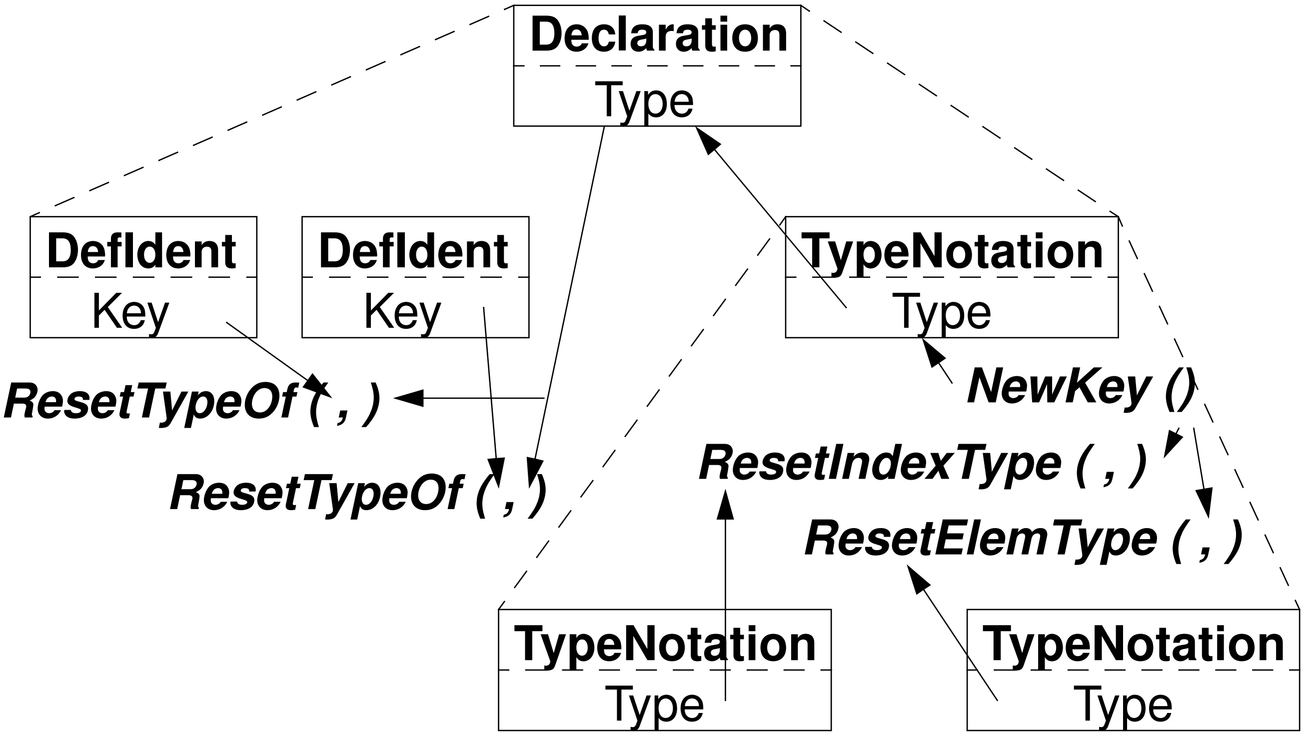
- **Typen selbst sind Programmobjekte**
  repräsentiert durch key
  benannt: in Typdefinitionen
  unbenannt: in komplexen Typangaben

- **Typen haben Eigenschaften**
  z. B. Elementtyp eines Array-Typs
  bestimmt in Typangaben

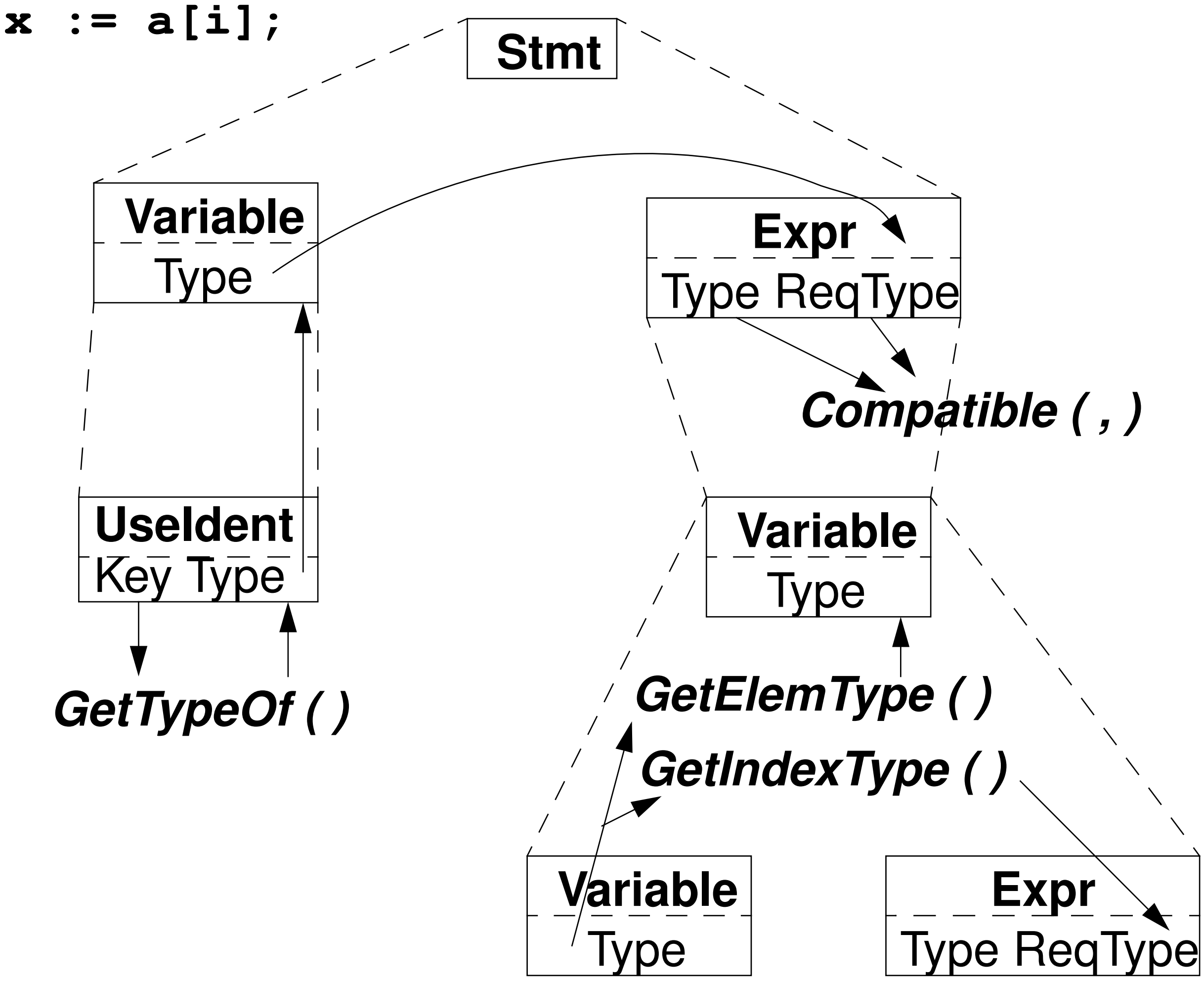
- **Typprüfung für Programmobjekte und Konstrukte:**
  Typ muß/darf nicht bestimmte Eigenschaften haben
  Vergleich erwarteter - tatsächlicher Typ
  (gleich, verträglich, ...)

Typanalyse: Operationen im Baum

```
a, b: array [1..10] of real;
```




```
x := a[i];
```



Identifikation überladener Operatoren

Gleiches Operatorsymbol (Quelloperator) für Zieloperatoren mit verschiedenen Signaturen und Bedeutungen

Tabelle aus Sprachdefinition:

Zeichen	Signatur	Bedeutung
+	int × int -> int	Ganzzahladdition
+	real × real -> real	Gleitpunktaddition
+	set × set -> set	Mengenvereinigung
/	real × real -> real	Gleitpunktdivision
=	t × t -> boolean	Vergleich

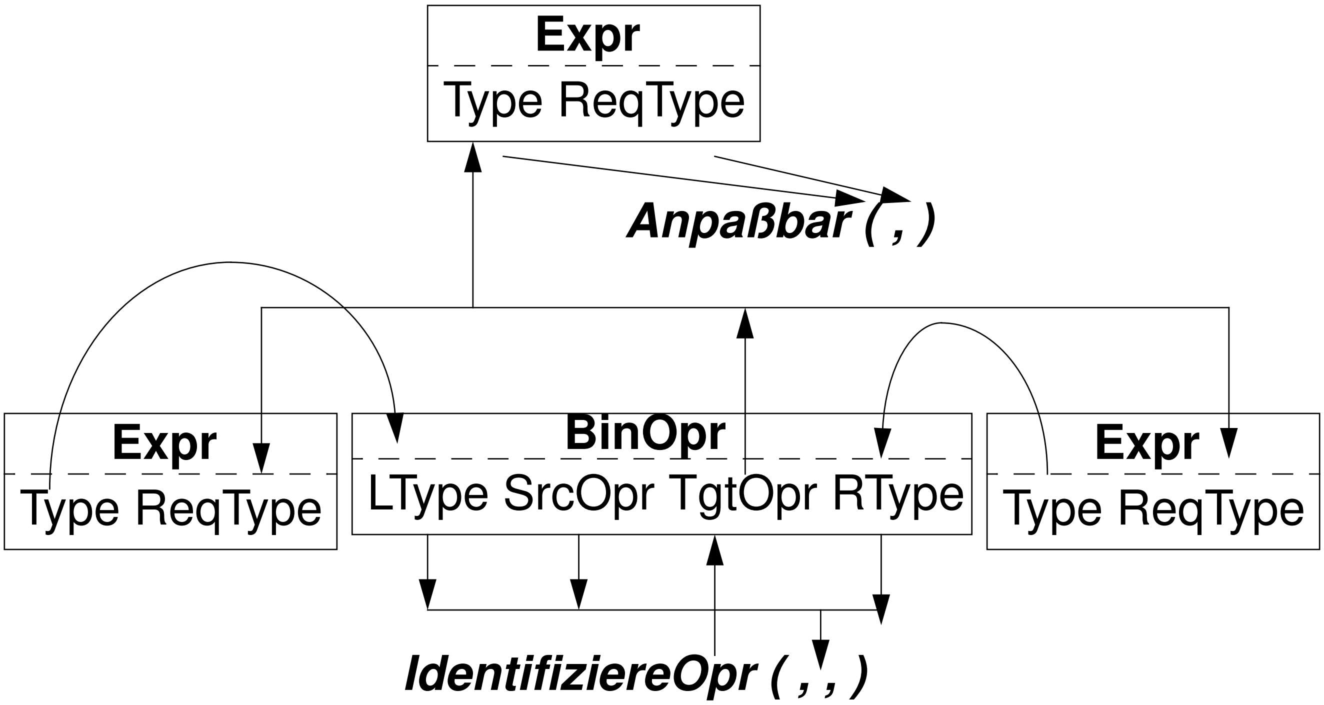
Coercion: implizit anwendbare Typanpassung

z. B. int -> real, char -> string, ...

Operatoridentifikation:

geg.: Quelloperator und Operandentypen
ges.: Zieloperator mit passender Signatur bei minimalen Anpassungen

Operationen im Strukturbaum:



Behandlung semantischer Fehler

Grundsätze für den Entwurf:

Quellbezogene Meldungen:

- **explizite oder implizite Forderung der Sprachdefinition**
  ein oder mehrere Prüfungen als Operationen im Baum
- **Prüfungen im kleinsten betroffenen Kontext**
  liefert präzise Quellposition für Fehlermeldung
  ggf. nötige Information dorthin transportieren
- **aussagekräftiger Meldungstext**
- **unterschiedliche Meldungen zu verschiedenen Symptomen**

Trotz Fehler werden alle Operationen im Baum ausgeführt:

- **Fehlerwerte einführen,** z. B. `NoKey, NoType, NoOpr`
- **Operationen, die Werte liefern, tun das auch im Fehlerfall**
- **Jede Operation akzeptiert auch Fehlerwerte als Parameter**
- **Meldungen zu Folgefehlern vermeiden**
  z. B. jeder Typ ist mit `NoType` verträglich

Transformation

Programm der Zwischensprache als **Zielbaum** (Wald) erzeugen.

Zwischensprache vorgegeben oder gemäß Operationen der abstrakten Quellmaschine **entwerfen**.

Entwurfsgrundsätze:

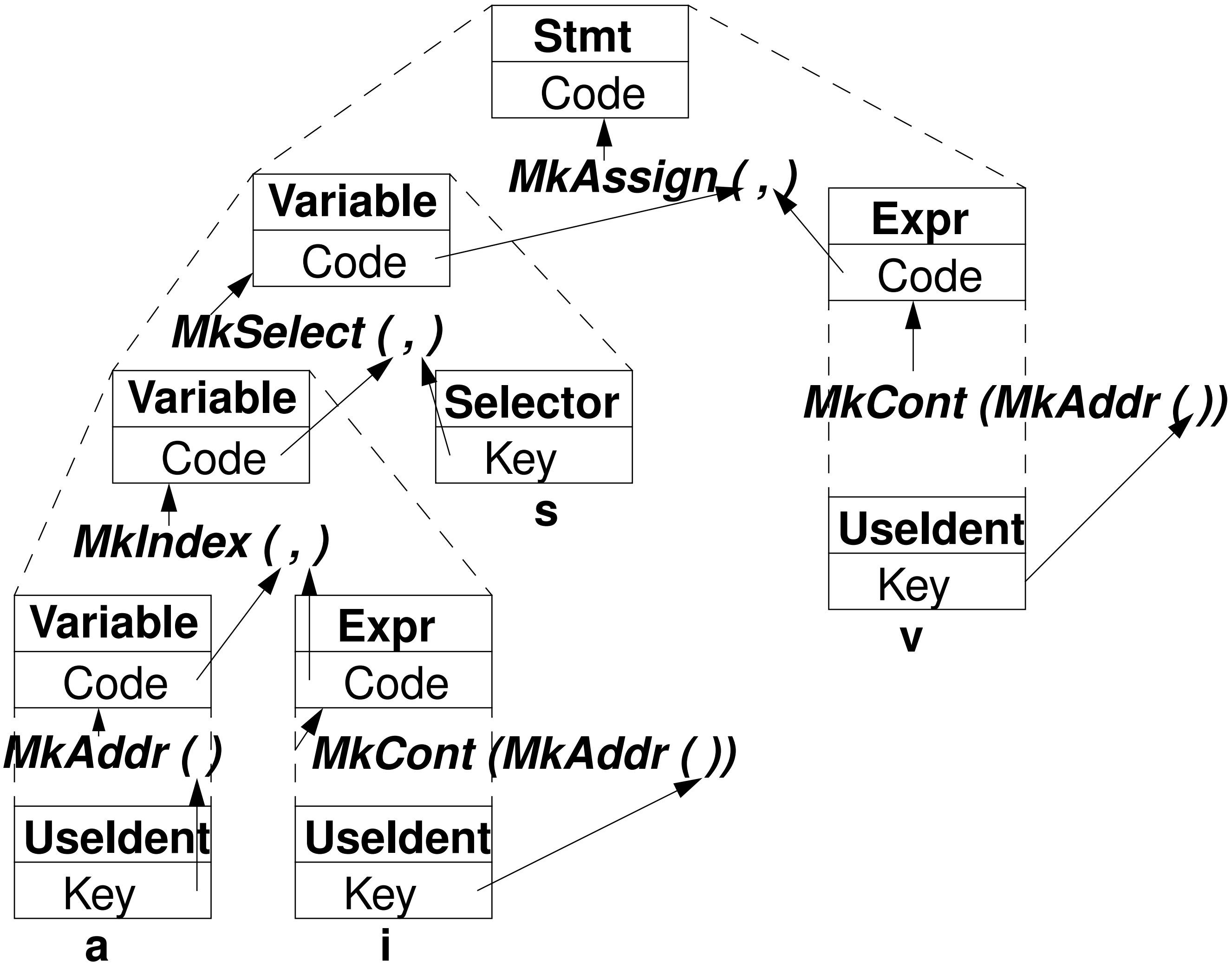
- **Struktur vereinfachen:**
  nur Strukturen und Eigenschaften, die für Synthese nötig,
  Deklarationen, Typangaben weglassen
  (sind im Definitionsmodul)
- **Strukturen vereinheitlichen**
  z. B. Schleifen
- **Zieloperatoren für überladene Operatoren**
- **implizite Operatoren einsetzen**
  Typanpassung, Laufzeitprüfungen

Zwischensprachbaum und Definitionsmodul an Synthese übergeben (Datenstruktur, Aufruffolge oder Datei)

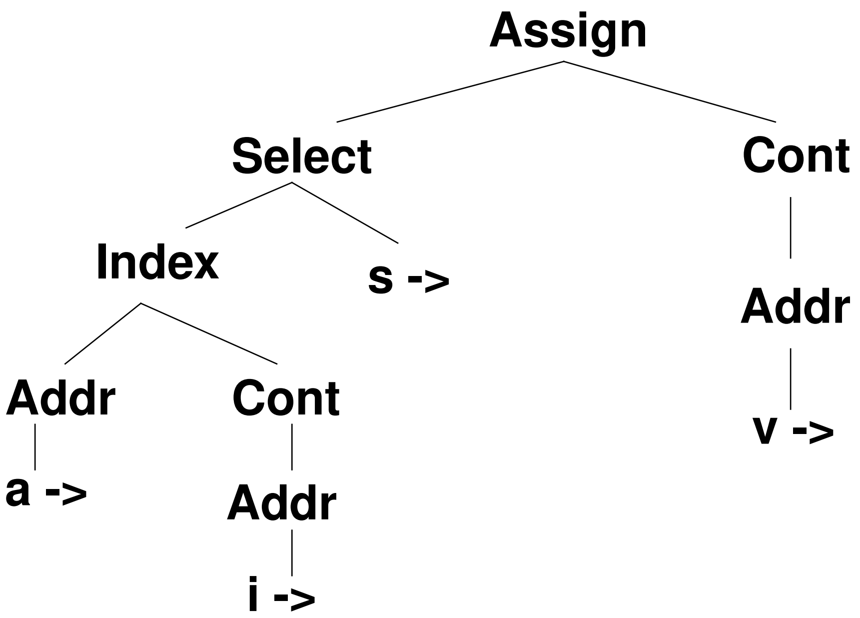
Bei **Source-to-Source-Übersetzung** repräsentiert der Zielbaum das **Zielprogramm**.
Text aus Baum erzeugen durch **rekursive Anwendung von Textmustern**.

Beispiel: Zielbaum erzeugen

Strukturbaum für `a[i].s := v;`



Zielbaum:



Definitionsmodul:

a -> ...
i -> ...
s -> ...
v -> ...

Attributierte Grammatiken

AG spezifiziert abhängige Berechnungen im Strukturbaum

deklarativ: nur explizite Abhängigkeiten
keine implizite Ausführungsreihenfolge

Generator erzeugt **Plan für Baumdurchlauf**
mit Aufrufen der Berechnungen
spezifizierte Abhängigkeiten werden eingehalten
berechnete Werte werden durch den Baum transportiert

Ergebnis: Attributauswerter
(anwendbar für jeden Baum zur AG)

Strategien für Baumdurchlauf:	**AG-Klasse**
k-mal links-abwärts	**LAG (k)**
k-mal alternierend links/rechts-abwärts	**AAG (k)**
1-mal aufwärts	**SAG**
Besuchssequenzen: individueller Plan für jede Produktion der abstrakten Syntax	**OAG**

Generator analysiert Abhängigkeiten, prüft Anwendbarkeit der Strategie, erzeugt Plan.

Paß-orientierte Strategien (LAG, AAG, SAG) auch manuell anwendbar; Besuchssequenzen mit Generator.

AG zu: Zielbaum erzeugen (U-45)

```
RULE:   Stmt ::= Variable ':=' Expr            COMPUTE
    Stmt.Code = MkAssign (Variable.Code, Expr.Code);
END;

RULE:   Variable ::= Variable '.' Selector     COMPUTE
    Variable[1].Code =
        MkSelect (Variable[2].Code, Selector.Key);
END;

RULE:   Variable ::= Variable '[' Expr ']'     COMPUTE
    Variable[1].Code =
        MkIndex (Variable[2].Code, Expr.Code);
END;

RULE:   Variable ::= UseIdent                  COMPUTE
    Variable.Code = MkAddr (UseIdent.Key);
END;

RULE:   Expr ::= UseIdent                      COMPUTE
    Expr.Code = MkCont (MkAddr (UseIdent.Key));
END;
```

Grundkonzepte von AGn

AG spezifiziert Berechnungen im Strukturbaum:

kontextfreie Baumgrammatik

Produktion + Berechnungen

p: Y ::= u   g (...)

wird in jedem Knotenkontext p ausgeführt

AG spezifiziert Abhängigkeiten zwischen Berechnungen

Symbole + Attribute

X.b = f (Y.a)   benutzt Ergebnis von g ( )

Y.a = g (...)   vor f () ausgeführt

Nachbed. Vorbedingung

abhängige Berechnungen auch in benachbarten Kontexten:

r: X ::= v Y w   X.b = f (Y.a)

p: Y ::= u   Y.a = g (...)

Attribute können Abhängigkeiten spezifizieren, ohne einen Wert zu transportieren.

X.GotType = ResetTypeOf (...);

Y.Type = GetTypeOf (...) <- X.GotType;

vereinfachende Notationen (siehe AG-Sprache Lido):

- **Berechnungen zu Symbolen**
- **Zugriff auf Attribute entfernter Symbole**
- **links-rechts-Abhängigkeit zwischen Berechnungen**

AG Binärzahlen

Attribute:

- **v** Wert
- **lg** Anzahl der Ziffern in der Teilfolge
- **s** Skalierung

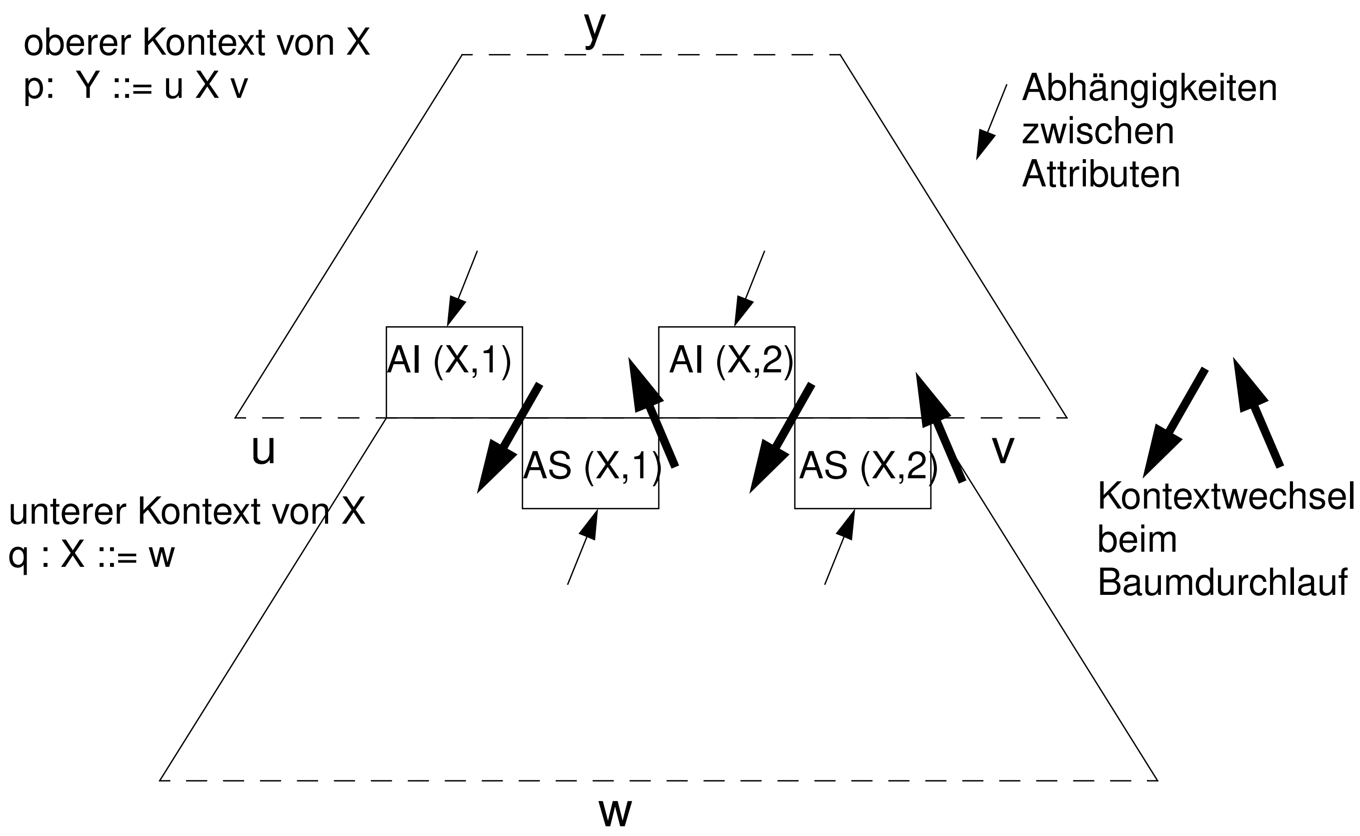
```
RULE p1:    D ::= L '.' L        COMPUTE
    D.v = ADD (L[1].v, L[2].v);
    L[1].s = 0;
    L[2].s = NEG (L[2].lg);
END;
RULE p2:    L ::= L B            COMPUTE
    L[1].v = ADD (L[2].v, B.v);
    B.s = L[1].s;
    L[2].s = ADD (L[1].s, 1);
    L[1].lg = ADD (L[2].lg, 1);
END;
RULE p3:    L ::= B              COMPUTE
    L.v = B.v;
    B.s = L.s;
    L.lg = 1;
END;
RULE p4:    B ::= '0'            COMPUTE
    B.v = 0;
END;
RULE p5:    B ::= '1'            COMPUTE
    B.v = Power2 (B.s);
END;
```

Abhängigkeitsanalyse für AGn

Attributmenge zum Symbol X 2-geteilt:

AI (X) : inherited (erworben), in oberen Kontexten berechnet

AS (X): synthesized (abgeleitet), in unteren Kontexten berechnet.



Ziel: Zerlegung aller Attributmengen, so daß

AI (X, i) vor i-ten Besuch von X berechnet wird

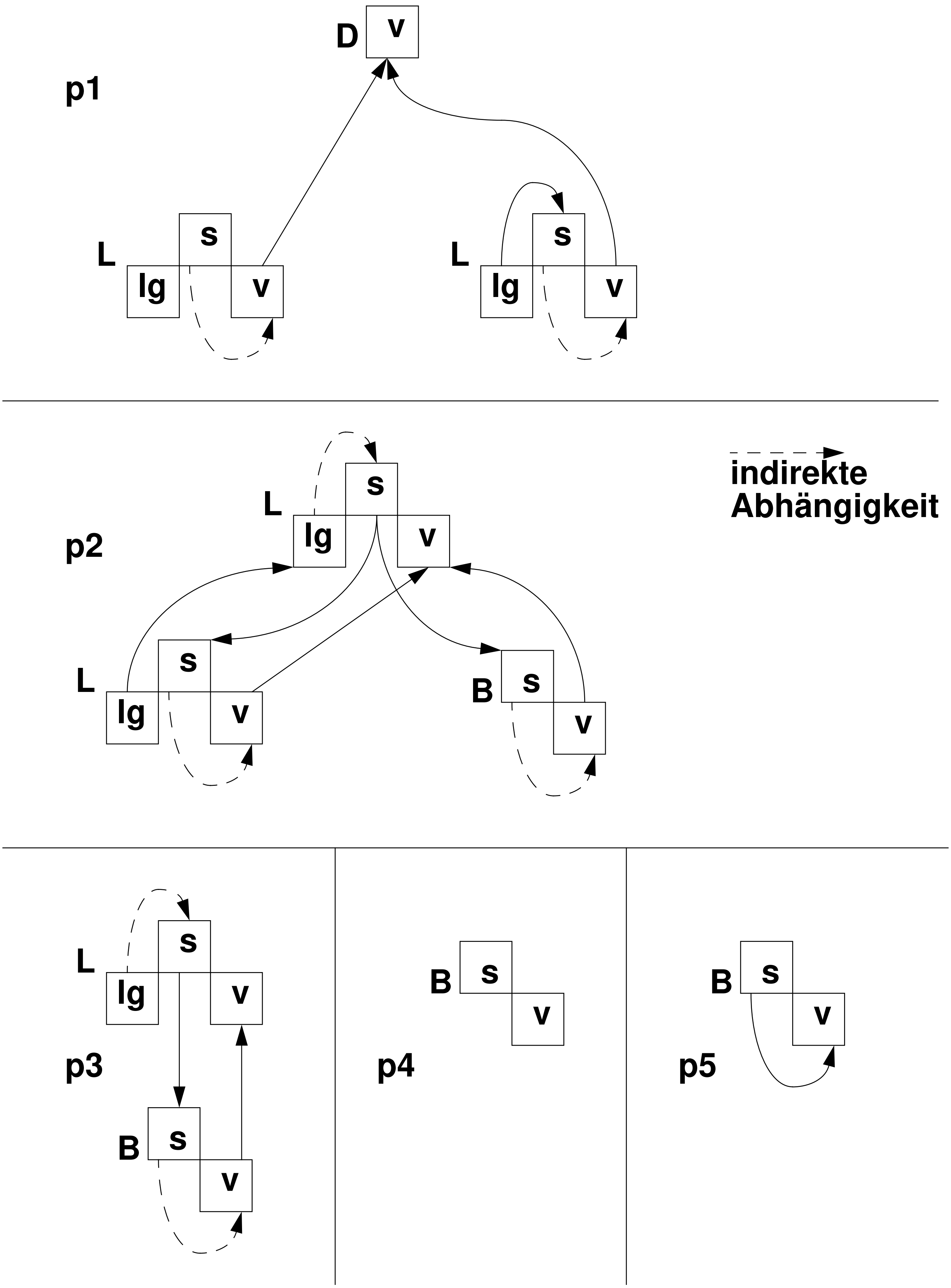
AS (X, i) bei dem i-ten Besuch von X berechnet wird

Vorraussetzung für Zerlegung:

An keinem X-Knoten in keinem Strukturbaum gibt es direkte oder indirekte Abhängigkeiten entgegen der Folge

AI (X, 1), AS (X, 1), ..., AI (X, k), AS (X, k)

Abhängigkeitsgraphen zu AG Binärzahlen



Besuchssequenzen

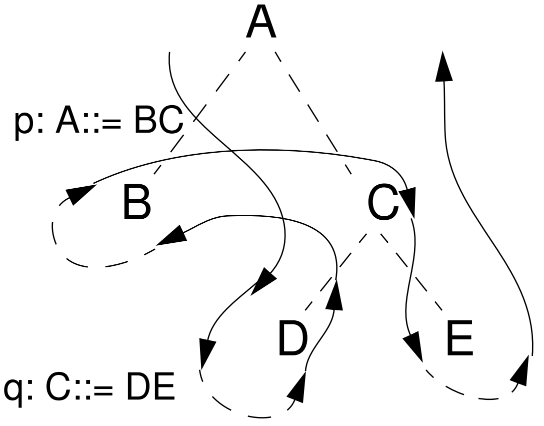
Eine Besuchssequenz (visit-sequence) vs_p zu jeder Produktion der Baumgrammatik:

$$p: X_o ::= X_1 \ldots X_i \ldots X_n$$

Folge von Operationen

$\downarrow$ i, j	j-ter Besuch des i-ten Unterbaums
$\uparrow$ j	j-te Rückkehr zum Vaterknoten
$eval_c$	Ausführung einer p zugeordneten Berechnung c

Beispiel im Baum:

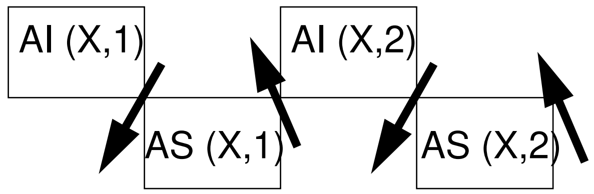


Besuchssequenzen

vs_p: ... $\downarrow$C,1 ...$\downarrow$B,1 ...$\downarrow$C,2 ...$\uparrow$1

vs_q: ... $\downarrow$D,1 ... $\uparrow$1 ... $\downarrow$E,1 ... $\uparrow$2

Attributzerlegung garantiert korrekte Verzahnung



Implementierung:

je eine Prozedur für Besuchssequenz bis $\uparrow$

Aufruf (mit Verzweigung über Produktion) für $\downarrow$

Besuchssequenzen für AG Binärzahlen

vs_{p1}: D ::= L '.' L

↓L[1],1; L[1].s=0; ↓L[1],2; ↓L[2],1; L[2].s=NEG(L[2].lg);

↓L[2],2; D.v=ADD(L[1].v, L[2].v); ↑1

vs_{p2}: L ::= L B

↓L[2],1; L[1].lg=ADD(L[2].lg,1); ↑1

L[2].s=ADD(L[1].s,1); ↓L[2],2; B.s=L[1].s; ↓B,1;

L[1].v=ADD(L[2].v, B.v); ↑2

vs_{p3}: L ::= B

L.lg=1; ↑1

B.s=L.s; ↓B,1; L.v=B.v; ↑2

vs_{p4}: B ::= '0'

B.v=0; ↑1

vs_{p5}: B ::= '1'

B.v=Power2(B.s); ↑1

Implementierung:

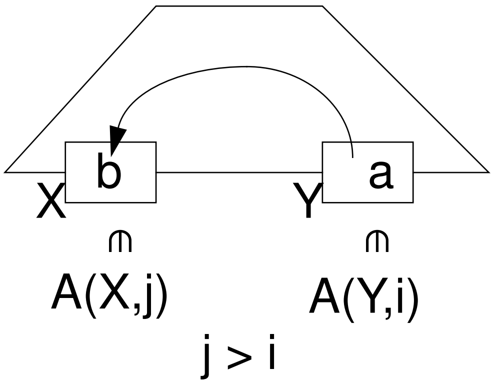
Prozedur vs<i><p> für jeden Abschnitt von vs_p bis ↑i

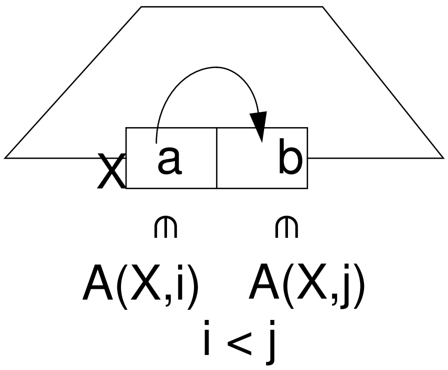
Aufruf mit Verzweigung für ↓X,i

LAG (k)-Bedingung

Für alle X gibt es eine Zerlegung A (X,1), ..., A (X, k), so daß A (X, i) im i-ten links-abwärts-Durchgang berechnet werden können.

Notwendige und hinreichende Bedingung über Abhängigkeitsgraphen zu Produktionen:





Algorithmus: berechnet A (1), ..., A (k), falls existiert:

für i = 1, 2, ...
A (i) := alle noch nicht zugeordneten Attribute
streiche Attribute solange folgende Regeln anwendbar:

- **streiche X.b, wenn es in einem Kontext von einem Attribut aus A (i) nach obigem Muster abhängt**
- **streiche Z.c, wenn es von einem gestrichenen Attribut abhängt**

falls alle Attribute zugeordent: **AG ist LAG (k)**, k = i
falls A (i) leer: **AG ist nicht LAG (k)**

Generatoren für AGn

LIGA	Universität Paderborn, in Eli	OAG
FNC-2	INRIA	ANCAG
Synthesizer Generator	Cornell University	OAG, inkrementell
CoCo	Universität Linz	LAG(1)

Eigenschaften von LIGA

- **höhere Spezifikationssprache Lido**
- **Modularisierung, Wiederverwendung von AG-Komponenten**
- **Objekt-orientierte Konstrukte zur Abstraktion von Berechnungsmustern**
- **Berechnungen: Funktionsaufrufe; implementiert außerhalb der AG**
- **auch Abhängigkeiten für Seiteneffekt-Berechnungen**
- **Notationen für Zugriff auf entfernte Attribute**
- **generiert Besuchssequenz-gesteuerte Attributauswerter in C**
- **Speicheroptimierung für Attribute**

6. Optimierung

Laufzeit und/oder Größe des Programms verringern bei unveränderter Wirkung.

Redundante Berechnungen eliminieren,

Berechnungen vereinfachen

Eingabe:	Programm in Zwischensprache
Aufgaben:	**Analyse** (Redundanz erkennen) **Transformationen** durchführen
Ausgabe:	verbessertes Programm in Zwischensprache

Analyse:

ermittelt statische Eigenschaften über Struktur und Ablauf des Programms (sicher, pessimistisch)

Analysekontext:

Ausdruck	lokal	
Grundblock	lokal	
Ablaufgraph (Prozedur)	global	intra-prozedural
Ablaufgraph u. Aufrufgraph	global	inter-prozedural

Transformation:

viele verschiedene Maßnahmen (U-53)

Vorbedingung für Anwendbarkeit (Analyse)

Wechselwirkung zwischen Transformationen

Verbessernde Transformationen

Liste einzelner Maßnahmen:

- **Algebraische Vereinfachung von Ausdrücken, Faltung**
  `2 * 3.14    x + 0   x * 2   x * * 2`
- **Konstantenweitergabe (constant propagation)**
  `x = 2; ... y = x * 5;`
- **Gemeinsame Teilausdrücke (common subexpressions)**
  `x = a * (b + c); ... y = (b + c)/2;`
- **Überflüssige Zuweisungen (dead variables)**
  `x = a + b; ... x = 5;`
- **Überflüssige Kopieranweisungen (copy propagation)**
  `x = y; ... ; z = x`
- **Nicht erreichbarer Code (dead code)**
  `b = true; ... if (b) x = 5; else y := 7;`
- **Code-Verschiebung (code motion)**
  `if (c) x = (a + b) * 2; else x = (a + b)/2;`
- **Einsetzen von Funktionsaufrufen (inlining)**
  `int Sqr (int i)  {return i * i;}`
- **Schleifen-invarianter Code**
  `while (b) {... x = 5; ...}`
- **Operationen mit Schleifeninduktionsvariablen**
  `i = 1; while (b){k = i * 3; f(k); i = i + 1;}`

Zu jeder Transformation
Vorbedingung für sichere Anwendung
größere Chance für Anwendung, wenn Vorbedingung in größerem Kontext analysiert.

Wechselwirkungen:
Anwendung einer Transformation kann andere ermöglichen oder verhindern.

Analysemethoden - Übersicht

Struktur des Programms darstellen:

Ausdrucksbaum

gerichteter, azyklischer Ausdrucksgraph (DAG)

nach Erkennung gemeinsamer Teilausdrücke

Grundblock (basic block):

Folge von Anweisungen maximaler Länge, die alle

ausgeführt werden, wenn die erste ausgeführt wird.

Ablaufgraph:

Knoten: Grundblöcke

Kanten: Verzweigungen zwischen Grundblöcken

Aufrufgraph:

Knoten: Funktionen

Kanten: `f->g` Funktion `f` enthält Aufruf von `g`

Analysen auf den Strukturen:

- **allgemeine Datenflußanalyse**
  z. B. Def-Use-Relation, Lebendigkeit für Variable, ...
- **Schleifen in beliebigem Ablaufgraph erkennen**
- **Induktionsvariable in Schleifen erkennen**
- **Analyse von Array-Indizes in Schleifen**
- **Aliasanalyse**

Datenflußgleichungen

Berechnung globaler Optimierungsinformation formuliert als Datenflußproblem über Ablaufgraph

z. B. „erreichende Definitionen“ (d: z := a):

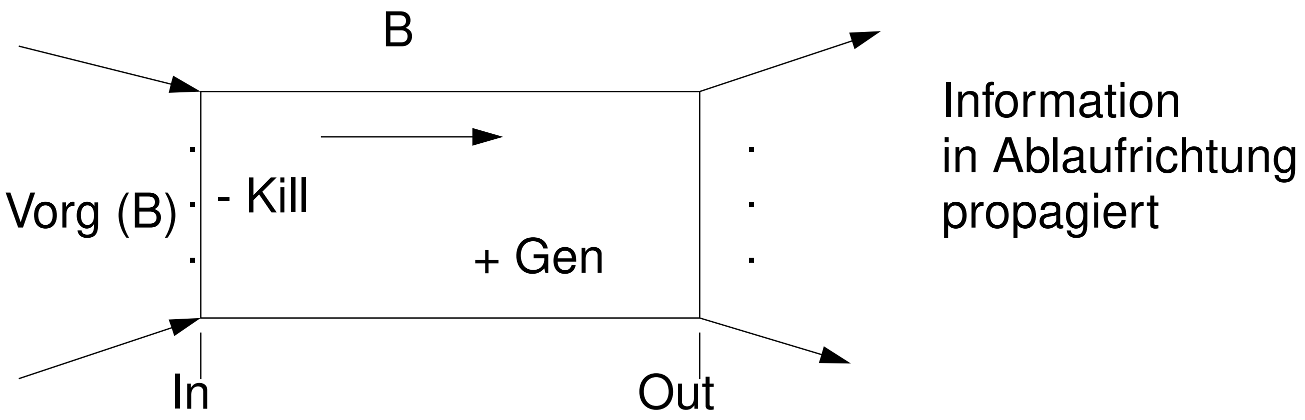
Welche Definitionen $\{d_i\}$ erreichen den Anfang von Block B? D. h. **es gibt einenPfad** von d_i nach B, auf dem nicht an die gleiche Variable zugewiesen wird.

Gleichungssystem für Vorwärtsprobleme

2 Gleichungen für jeden Block:

$$\text{Out }(B) = \text{Gen }(B) \quad \cup \quad (\text{In }(B) - \text{Kill }(B))$$

$$\text{In }(B) = \underset{h \in \text{Vorg }(B)}{\Theta} \text{Out }(h)$$



In, Out, Gen, Kill: Mengen von Objekten
z. B. Programmstellen, Variable, Ausdrücke

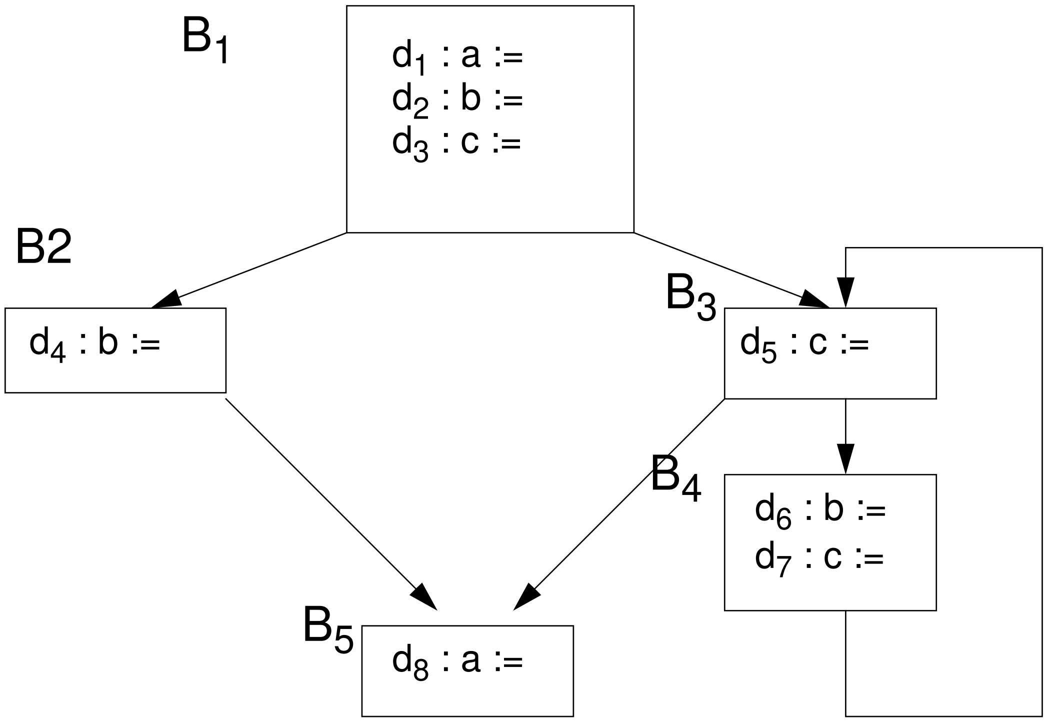
In, Out: Variablen des Gleichungssystems

Gen, Kill: Konstante für jeden Block

$\Theta = \cup$ **Existenzproblem** (es gibt einen Pfad); **minimale** Lösung

$\Theta = \cap$ **Allproblem** (für alle Pfade); **maximale** Lösung

Beispiel: Erreichende Definitionen



	Gen	Kill	In	Out
B_1	d_1, d_2, d_3	d_4, d_5, d_6, d_7, d_8	$\emptyset$	d_1, d_2, d_3
B_2	d_4	d_2, d_6	d_1, d_2, d_3	d_1, d_3, d_4
B_3	d_5	d_3, d_7	d_1, d_2, d_3, d_6, d_7	d_1, d_2, d_5, d_6
B_4	d_6, d_7	d_2, d_3, d_4, d_5	d_1, d_2, d_5, d_6	d_1, d_6, d_7
B_5	d_8	d_1	$d_1, d_2, d_3, d_4, d_5, d_6$	$d_2, d_3, d_4, d_5, d_6, d_8$

Gen (B): Definition d_i: v := A in Block B, auf die in B keine weitere Definition für v folgt.

Kill (B): alle Definitionen d_j: v := A nicht in B, an deren Variable auch in B zugewiesen wird.

Rückwärtsprobleme

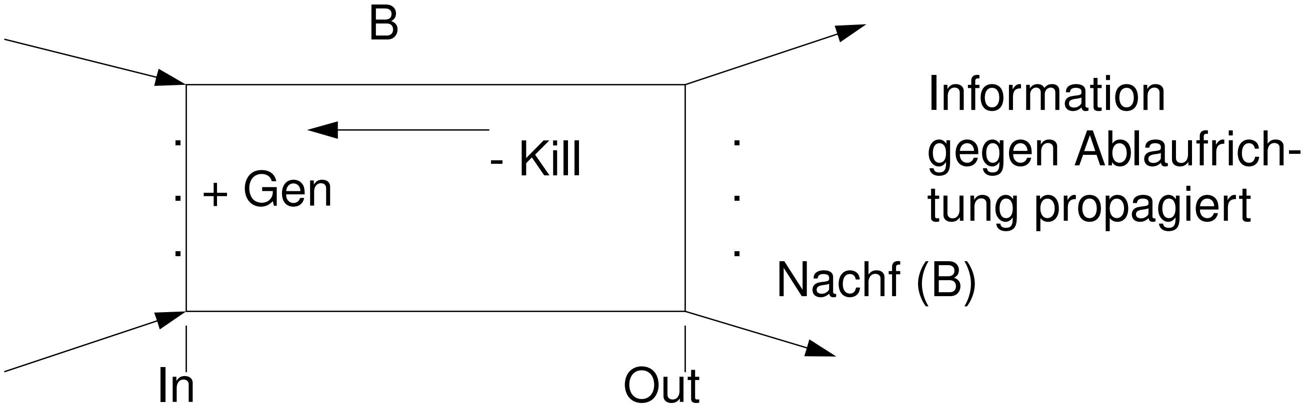
z. B. „lebendige Variable“:

Welche Variablen $\{v_i\}$ sind am Ende von Block B lebendig? D. h. **es gibt einen Pfad** von B zu einer Benutzung von v_i, auf dem nicht an v_i zugewiesen wird.

Gleichungsschema für Rückwärtsprobleme:

$$In\ (B) = Gen\ (B) \cup (Out\ (B) - Kill\ (B))$$

$$Out\ (B) = \underset{h \in Nachf\ (B)}{\Theta} In\ (h)$$



„lebendige Variable“: $\Theta = \cup$

Gen (B): Variable, die in B benutzt, und vor der Benutzung in B nicht definiert werden

Kill (B): Variable, die in B definiert, und vor der Definition in B nicht benutzt werden

Existenz- und **Allprobleme** wie bei Vorwärtsproblemen

Iterative Lösung von Datenflußgleichungen

Initialisierung Existenzproblem:

```
for all B do begin In(B):=∅; Out(B):=Gen(B) end;
```

Initialisierung Allproblem:

```
for all B do
   begin In(B) := U; {volle Menge}
         Out(B):= Gen(B) ∪ (U-Kill(B))
   end;
repeat
   stable := true;
   for all B ≠ Ba   {*}
   do begin
      for all V ∈ Vorg(B) do
         In(B):= In(B) Θ Out(V);
         oldout:= Out(B);
         Out(B):= Gen(B) ∪ (In(B)-Kill(B));
         stable:= stable and Out(B)=oldout
   end
until stable
```

Komplexität: $0(n^3)$ n = Anzahl Grundblöcke

bzw. $0(n^2)$ falls $|\text{Vorg (B)}| \leq k << n$

heuristische Verbesserung:

Änderungen möglichst schnell propagieren. Deshalb Blöcke ohne Rückwärtskanten topologisch sortieren und bei {*} in dieser Ordnung durchlaufen. Dann wird die Anzahl der repeat-Durchläufe nur durch die Zahl der Rückwärtskanten bestimmt.

Bei Rückwärtsproblem entsprechend mit umgekehrter Kantenrichtung sortieren.

hierarchisches Verfahren Intervallanalyse:
prägt dem Ablaufgraphen Struktur auf; berechnet Gen, Kill aufwärts, In, Out abwärts in der Hierarchie.

Schleifenerkennung im Ablaufgraphen

Schleifen gebildet durch Sprünge, aus Quellprogramm oder im Zwischen-Code erzeugt

Relation über Knoten des Ablaufgraphen:

a **dominiert** b: (reflexiv: a dominiert a):

Jeder Weg vom Anfangsknoten zu b führt über a.

Schleifeneigenschaft:

- **eindeutiger Eingangsknoten h** dominiert alle Knoten der Schleife, h ist Schleifenkopf
- **von jedem Knoten der Schleife führt ein Weg zu h**

Rückwärtskante im Ablaufgraph n -> d mit d dominiert n

Schleife zur Rückwärtskante n -> d:

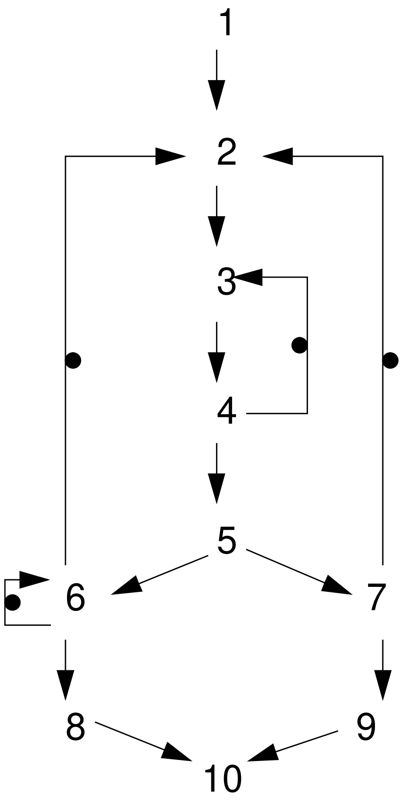
d und alle Knoten, die n erreichen, ohne d zu berühren

d ist der Eingangsknoten

$S = \{d, n\} \cup \{ v \mid v = Vorg(a), a \in S, a \neq d\}$

Rückwärtskanten und Schleifen im Ablaufgraph

Beispiel:



Rückwärtskante:	Schleife:
4 -> 3	$S_1 = \{3,4\}$
6 -> 2	$S_2 = \{2, 3, 4, 5, 6\}$
7 -> 2	$S_3 = \{2, 3, 4, 5, 7\}$
6 -> 6	$S_4 = \{6\}$

Schleifen sind

- **disjunkt** $S_1 \cap S_4 = \emptyset$
- **geschachtelt** $S_1 \subset S_2$
- **nicht geschachtelt, aber gleicher Kopf** S_2, S_3
  (zu einer Schleife vereinigen)

7. Code-Erzeugung

Eingabe: Programm in Zwischensprache

Aufgaben:

Speicherabbildung	Definitionsmodul erweitern
Code-Auswahl	Instruktionssequenzen erzeugen
Registerzuteilung	Verwendung von Registern für Zwischenergebnisse

Ausgabe: Maschinenprogramm als Datenstruktur

Entwurf der Code-Erzeugung:

- Maschineneigenschaften untersuchen
- Speicherabbildung entwerfen
- Code-Sequenzen zu allen Zwischensprachoperatoren aufstellen; ggf. Alternativen

Implementierung:

Speicherabbildung:

Programm und Definitionsmodul durchlaufen,

Adressen und Speicherumfang berechnen

Code-Auswahl:

Generatoren für Mustersetzung in Bäumen

Registerzuteilung:

Algorithmen für unterschiedlich großen Kontext:

Ausdrucksbäume, Grundblöcke, Ablaufgraphen

Speicherabbildung

Ziel: Zu Speicherobjekten des Programms bestimmen:

Speicherklasse, Relativadresse darin, Umfang

Implementierung:

als Eigenschaften im Definitionsmodul berechnen.

Entwurf:

1. Speicherklassen:

Code-Speicher

globale Daten

Laufzeitkeller

Prozedurschachteln (activation records); siehe U-61

Halde für dynamisch allokierte Daten, Speicherbereinigung

Register für

Adressierung der Speicherbereiche

Funktionsergebnisse, ggf. Parameter

lokale Variable, Zwischenergebnisse (Registerzuteilung)

2. Abbildung der Grundtypen

Maschineneigenschaften:

Wertdarstellung

Verarbeitungsumfang, Speicherumfang, Ausrichtung

verfügbare Operationen

3. Abbildung zusammengesetzter Typen

Arrays, Verbunde, Mengen, Klassen, Funktionen, ...

Prozedurschachteln

Prozedurschachtel (activation record) enthält lokale Daten für einen Prozeduraufruf und Verkettung auf dem Laufzeitkeller.

Entwurf der Schachtelstruktur zusammen mit Aufruf-Code

Schachtelstruktur	**Aufruf-Code**	
	Aufrufstelle	Prozedur-Code
1. Parameter	Parameter kellern	
n-ter Parameter		
stat. Vorgänger (*)	stat. Vorg. kellern	
Rückkehradresse	Unterprogrammsprung	
dynamischer Vorgänger		dyn. Vorg. kellern
(**)		Schachtelreg :=
		Kellerpegel
lokale Variable		Kellerpegel für
		lok. Var. erhöhen
gesicherte Register		Register sichern
		...
		Prozedurrumpf
		...
		Register zurückladen
		lok. Variable
		freigeben
		Schachtelreg. :=
		entkellern
		Rücksprung
	stat. Vorg. entkellern (*)	
	Parameter entkellern	

(*) Nur bei Sprachen mit geschachtelten Prozeduren.

(**) Relativadresse 0 der Schachtel

Code-Abbildung Ablaufstrukturen

Ablaufstrukturen in Code-Sequenzen mit Marken und Sprüngen umsetzen. Je nach Zwischensprache evtl. schon in der Transformationsphase.

Für jede Ablaufstruktur eine oder mehrere alternative Code-Sequenzen entwerfen, z. B.

```
while (B) S :   M1:  Code (B, false, M2)
                     Code (S)
                     goto M1
                M2:
```

oder

```
                     goto M2
                M1:  Code (S)
                M2:  Code (B, true, M1)
```

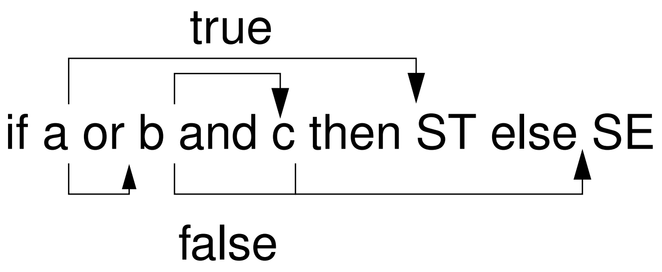
es bedeuten

Code (S): Code für `S` einsetzen

Code (B, true, M1): Code für Bedingung `B` so erzeugen, daß bei Ergebnis `true` auf die Marke `M1` gesprungen, sonst danach fortgesetzt wird

Kurzauswertung logischer Operatoren

Logische Ausdrücke in Sprungstrukturen übersetzen; falls Sprache erlaubt, nicht alle Operanden auszuwerten.



2 Code-Sequenzen für jeden Operator;Auswahl top-down:

Code (A and B, true, M):	Code (A, false, N) Code (B, true, M) N:
Code (A and B, false M):	Code (A, false, M) Code (B, false, M)
Code (A or B, true, M):	Code (A, true, M) Code (B, true M)
Code (A or B, false, M):	Code (A, true, N) Code (B, false, M) N:
Code (not A, X, M):	Code (A, not X, M)
Code (A < B, true, M):	Code (A, Ri); Code (B, Rj) cmp Ri, Rj bralt M
Code (A < B, false, M):	Code (A, Ri); Code (B, Rj) cmp Rj, Ri bralt M
Code für Blätter:	bedingter Sprung

Code-Auswahl für Ausdrücke

- Benachbarte Operationen können evtl. zu einer Instruktion zusammengefaßt werden
- verschiedene Alternativen möglich
- Wertdeskriptoren zu Baumknoten beschreiben Unterbringung des Zwischenergebnisses, z. B.

$\mathbf{R_i}$	Wert im Register R_i
$\mathbf{R_i,c}$	Adresse gebildet aus Inhalt R_i + Konstante c
c	konstanter Wert c
(...)	Inhalt der in Klammern angegebenen Adresse

Übersetzungsmuster für Baumknoten, z. B.

addr = Variablenadresse, **cont** = Inhaltsoperation,
const = Konstanter Wert, **addradd** = Adresse + Wert

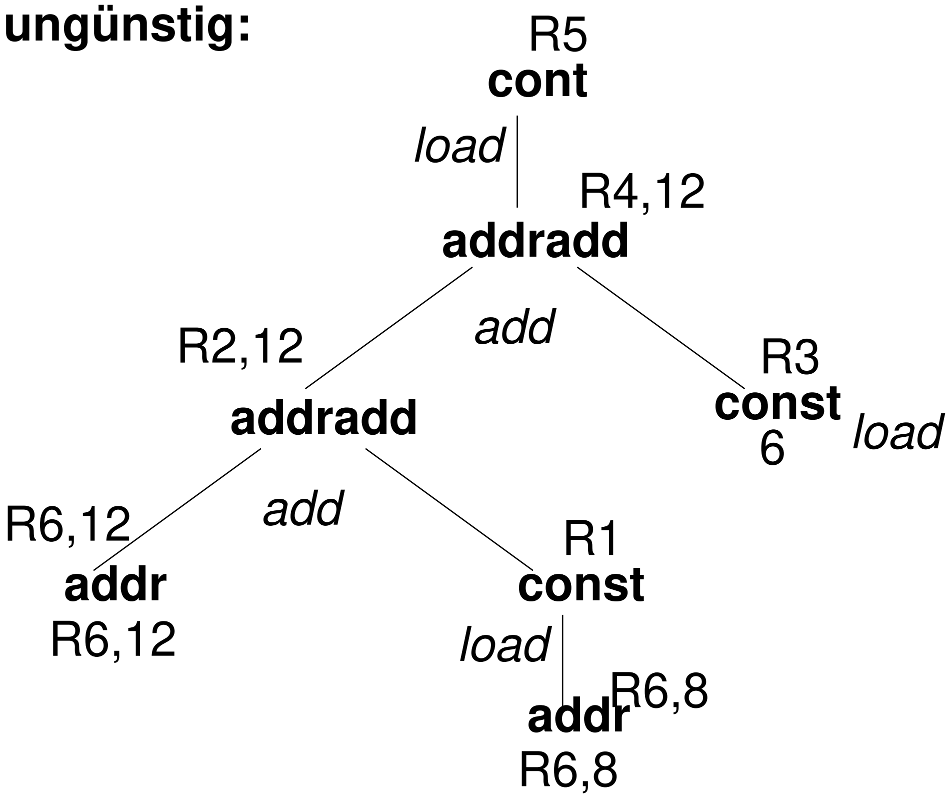
Operator	Wertdeskriptoren Operand(en)		Ergebnis	Code
addr	R_i, c		-> R_i,c	./.
const	c		-> c	./.
const	c		-> R_i	load c, R_i
cont	R_i, c		-> (R_i, c)	./.
cont	R_i		-> (R_i)	./.
cont	R_i, c		-> R_j	load (R_i, c), R_j
cont	R_i		-> R_j	load (R_i), R_j
addradd	R_i	c	-> R_i, c	./.
addradd	R_i, c1	c2	-> R_i, c1 + c2	./.
addradd	R_i	R_j	-> R_k	add Ri, R_j, R_k
addradd	R_i, c	R_j	-> R_k, c	add R_i, R_j, R_k

Aufgabe: Baum mit Mustern passend überdecken, so daß möglichst wenige Instruktionen erzeugt werden.

Beispiel Code-Auswahl

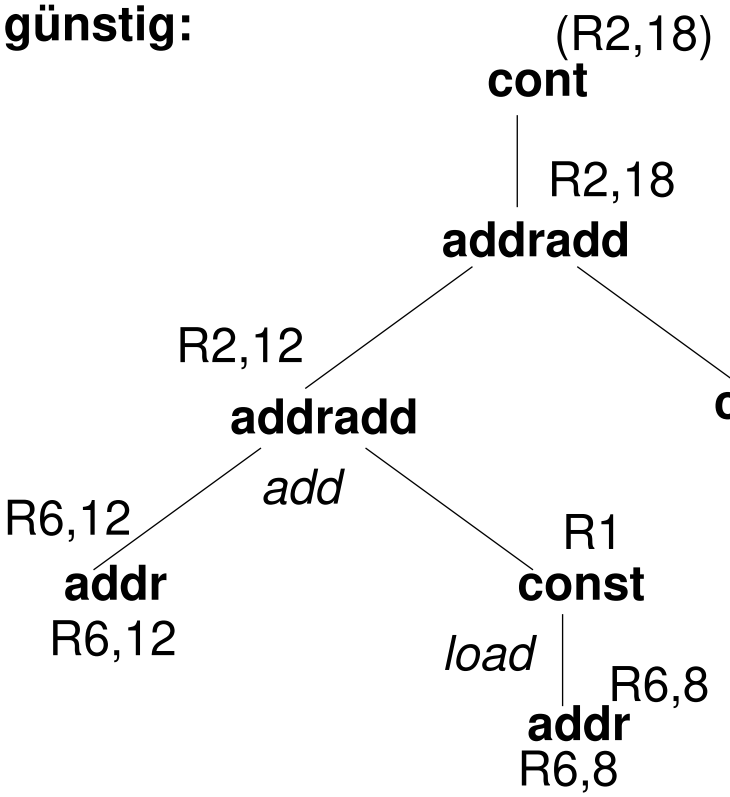
Baum zu `... := a[i].s`

ungünstig:



load (R6,8), R1
add R6,R1,R2
load 6,R3
add R2,R3,R4
load (R4,12),R5
store R5, ...

günstig:



load (R6,8), R1
add R6,R1,R2
store (R2,18),...

Musterersetzung in Bäumen

Methode: Baum bottom-up durchlaufen, anwendbare Muster auswählen, Entscheidung nach oben verzögern (vergl. dynamische Programmierung)

Spezifikation der Baummuster durch KFG-Produktionen plus Code; Wertdeskriptoren als Nichtterminale, z. B.

RegConst ::= addradd Reg   Const

RegConst ::= addradd RegConst Reg   add R_i, R_j, R_k

auch tiefere Baummuster, z. B.

Void ::= assign RegConst addradd Reg Const

store (R_i, c1),(R_j, c2)

Kosten für Auswahl: Anzahl Instruktionen

Methode „Graham, Glenville“, Generatoren : GG, GGSS:

Baum in Präfix-Form,

Parsing mit mehrdeutiger KFG

Entscheidung nach Kosten

Methode „Bottom-up Rewrite“, Generator BURG:

Muster zur Generierungszeit analysieren

Baumautomat generieren

Auswahlentscheidungen werden als Zustände von Knoten codiert

Registerzuteilung

Registerverwendungen:

Zwischenergebnis in Ausdrücken

mehrfach benutzte Ausdrucksergebnisse

Variable

Prozedurparameter

Funktionsergebnis

Kellerpegel, Schachtelzeiger, Haldenzeiger, ...

Anzahl der Register (in jeder Klasse) ist beschränkt

gute Registernutzung reduziert

- Zahl der Speicherzugriffe
- Code zum Zwischenspeichern (spill code)

verschiedene Zuteilungsverfahren für unterschiedlich großen Kontext:

- Ausdrucksbäume (Sethi, Ullman)
- Grundblöcke (Belady)
- Ablaufstrukturen (Graphfärbung)

nützliche Technik: Verschieben der Registerzuteilung in spätere Phase durch Zuteilung **symbolischer Register** aus unbegrenztem Vorrat

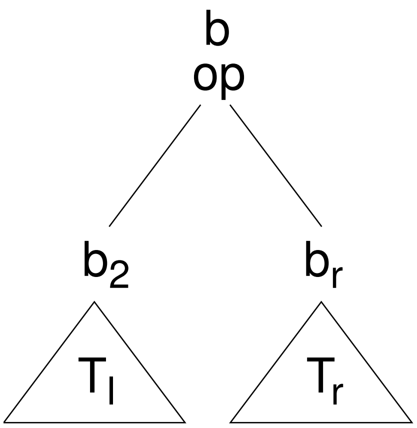
Registerzuteilung für Ausdrucksbäume

Voraussetzung für Optimalität: [Sethi, Ullman '70]

optimaler Code kann „baumartig“ zusammengesetzt werden

(nicht bei mehreren Registerklassen oder Registerpaaren)

Prinzip: bei 2-stelligen Operatoren erst den Teilbaum mit höherem Registerbedarf auswerten



Ann.: Erg. v. T_l und T_r in Registern

Reihenfolge	Reg.bedarf b =
T_l T_r op	max (b_l, b_r + 1)
T_r T_l op	max (b_r, b_l + 1)

b minimieren

falls b > regmax (Anzahl verfügbarer Register):

dann gilt $b_l = b_r$ = regmax;

Spill-Code: T_r; **store R_r, h;** T_l; **load h, R_r**; op R_r, R_l

für T_l sind alle Register frei

load h, R_r entfällt, wenn

Speicheroperand h in op h, R_l möglich ist.

Implementierung z. B. durch Attributierung:

<u>1. Phase:</u> Bedarf, Ausw.reihenf. = synth. Attribute (aufwärts)

<u>2. Phase:</u> Registerzuteilung = inh. Attribute (abwärts)

<u>3. Phase:</u> Code-Erz. in Auswertungsreihenfolge (aufwärts)

Grundblöcke in 2 Durchgängen (Belady)

1. Durchgang:

Bestimmung der **Lebensdauer** der Werte, d.h. von Berechnung bis letzter Benutzung (mehrfache Benutzung ist möglich!).

Lebensdauer dargestellt als Intervallgraph

Schnitt im Graph = Anzahl belegter Register

maximaler Schnitt im Graph = Registerbedarf des Blockes

Register im Graph zuordnen

Bei Registermangel Werte zum Zwischenspeichern auswählen, Kriterien:

- Wert, der schon im Speicher steht
- Wert, der am spätesten wiederverwendet wird

2. Durchgang:

Register im Code zuteilen

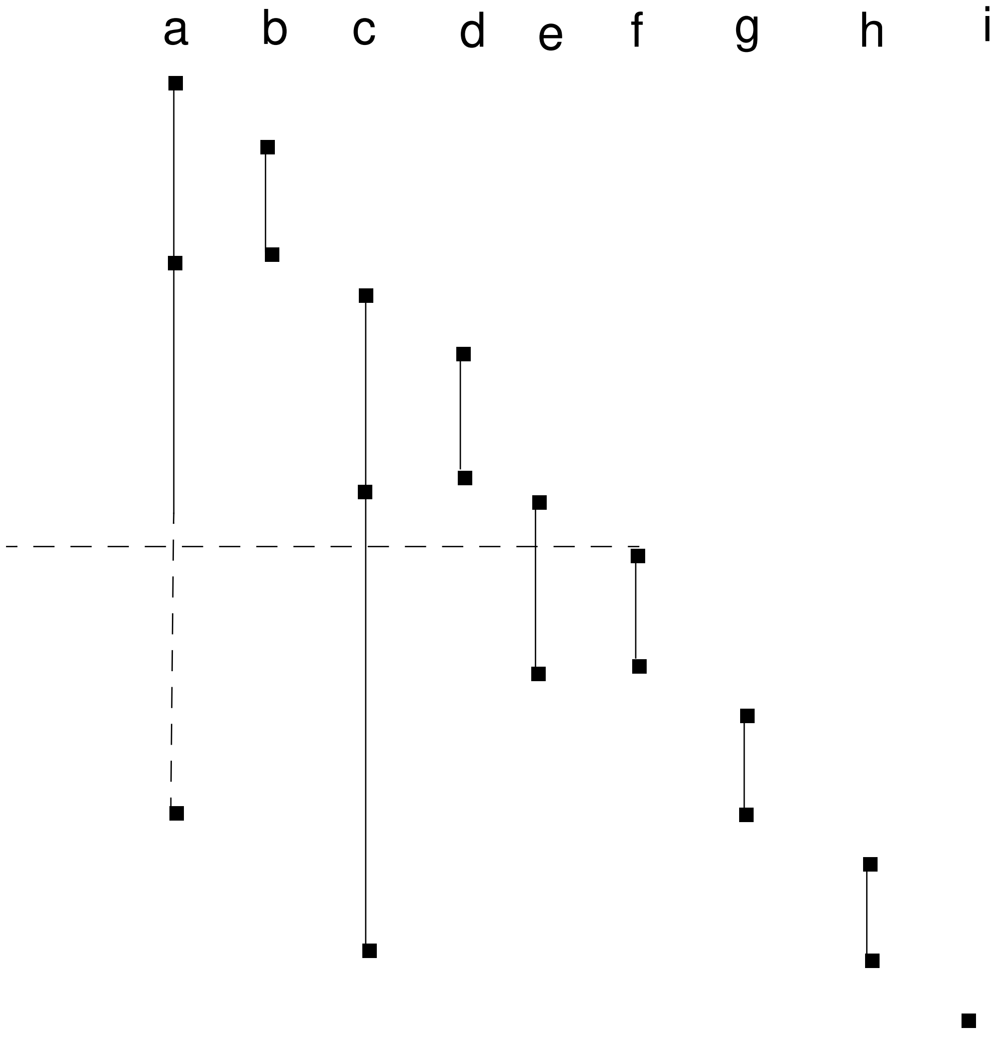
(Ausführungsreihenfolge wird nicht verändert.)

Verfahren ursprünglich als **Seitentauschverfahren** für Speicherverwaltungen von **Belady** 1966 vorgestellt.

Beispiel für Belady-Verfahren

Lebensdauer der Werte eines Grundblockes

a := x
b := y
c := b + a
d := z
e := d * c
f := s
g := e / f
h := g + a
i := h * c



Registerzuteilungen:

		a	b	c	d	e	f	g	h	i
4 Reg:	(a)	d1	d2	d2	d3	d3	d4	d3	d3	d3
3 Reg:	(b)			*			d2			
3 Reg:	(c)	*					d1			

* zwischenspeichern

Registerzuteilung durch Graphfärbung

Anwendung auf **Ablaufstrukturen**

Berechnung und Benutzung der Werte in verschiedenen Grundblöcken.

Zunächst symbolische Register zuteilen; dann

Unverträglichkeitsgraph aufstellen:

Knoten: symbolische Register

Kante {a, b}: Lebensdauer von a und b überlappt

Lebensdauern werden mit Datenflußanalyse ermittelt.

„Färbung“ mit realen Registernummern:

Kante {a, b} im Graph:

a und b in verschiedenen Registern

Graphfärbung NP-vollständig

heuristisches Verfahren zur Färbung mit k Farben (Registern):

sukzessive Knoten mit Grad < k streichen

falls am Ende Graph leer:

Graph ist k-färbbar

Knoten in umgekehrter Reihenfolge färben

sonst

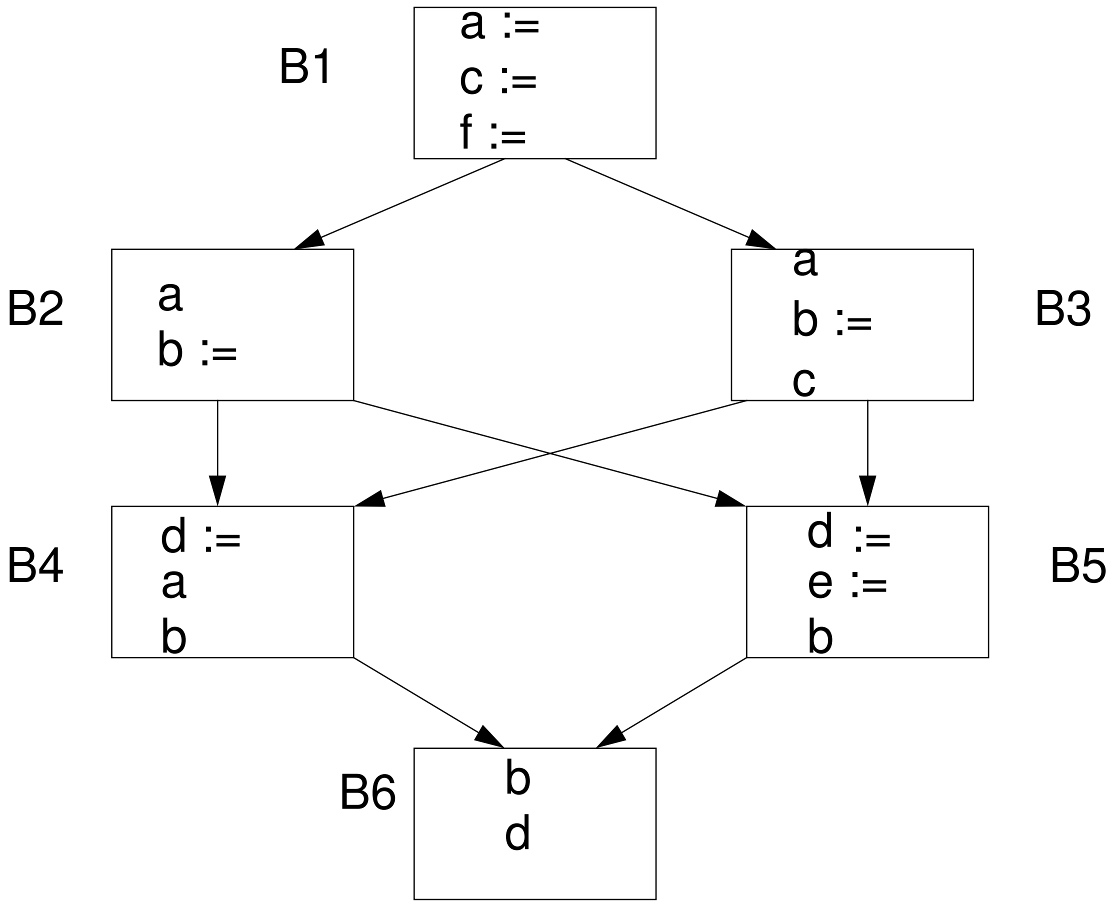
Graph ist so nicht k-färbbar

Knoten zum Zwischenspeichern auswählen (spill code)

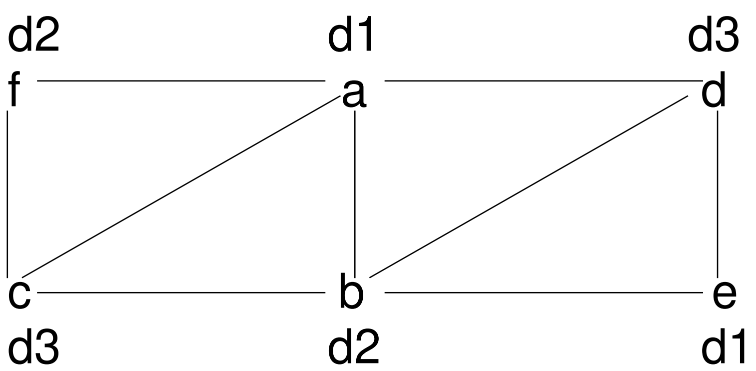
Verfahren erneut anwenden

Beispiel: Graphfärbung

Ablaufgraph mit Benutzung von Werten



Unverträglichkeitsgraph

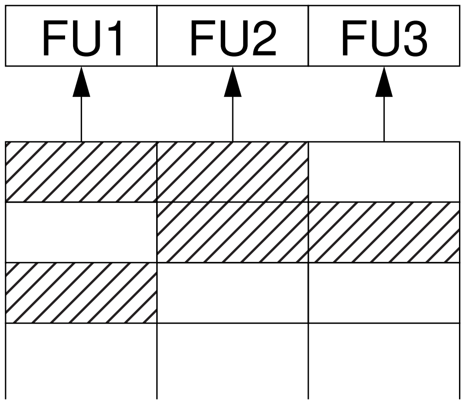


8. Code-Erzeugung für Parallelverarbeitung

Prozessor kann mehrere Instruktionen zugleich ausführen.
Übersetzer ordnet Instruktionen für möglichst kurze Ausführungszeit an (instruction scheduling).

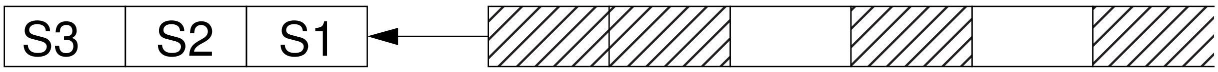
Modelle der Prozessor-Parallelität:

1. Parallele Funktionseinheiten (super scalar, VLIW):



parallel angeordneter Code

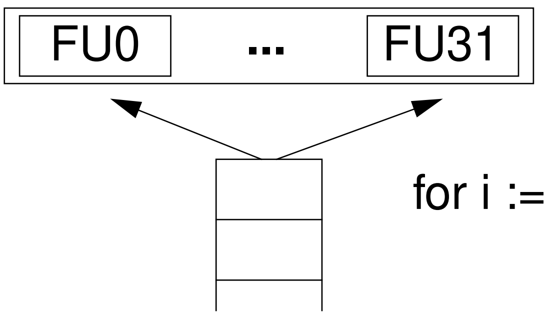
2. Pipeline-Prozessor



sequentiell angeordneter Code

3. Datenparalleler (Vektor-) Prozessor

alle Funktionseinheiten führen zugleich die gleiche Instruktion auf verschiedenen Daten aus



for i := 0 to 31 do c[i] := a[i] + b [i];
eine Instruktion!

Schleifen für parallele Ausführung erkennen und transformieren.

Datenabhängigkeitsgraph

Feingranulare, nutzbare Parallelität von Operationen erkennen.
Sequentielle Anordnung ist überspezifiziert.

Datenabhängigkeitsgraph (für Grundblock):

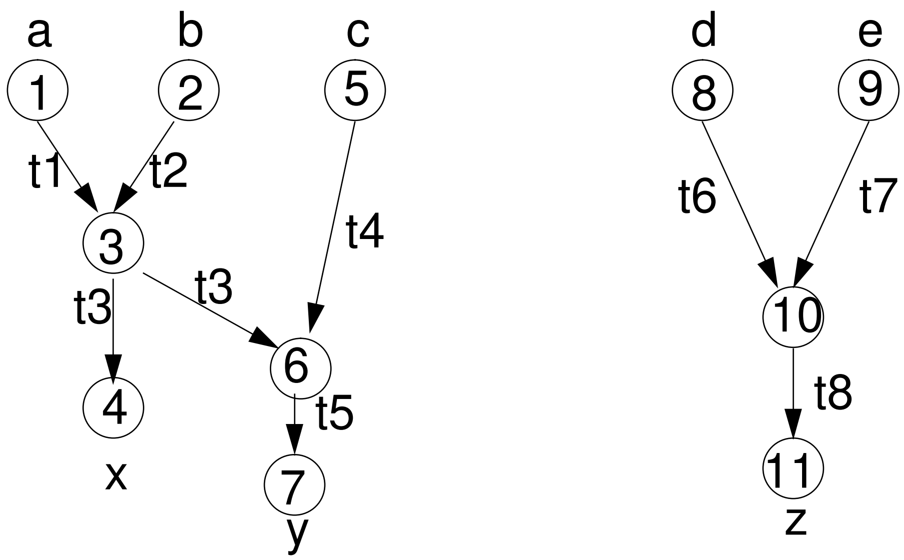
Knoten: Operationen

Kanten: Benutzung von Operationsergebnissen

Beispiel Grundblock:

1:	t1	:= a
2:	t2	:= b
3:	t3	:= t1 + t2
4:	x	:= t3
5:	t4	:= c
6:	t5	:= t3 + t4
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
10:	t8	:= t6 + t7
11:	z	:= t8

Datenabhängigkeitsgraph:



List-Scheduling

gegeben: Datenabhängigkeitsgraph

gesucht: Zuordnung von maximal k Operationen zu jeweils einem Takt, so daß Abhängigkeiten vorwärts weisen (Schedule).

Algorithmus:

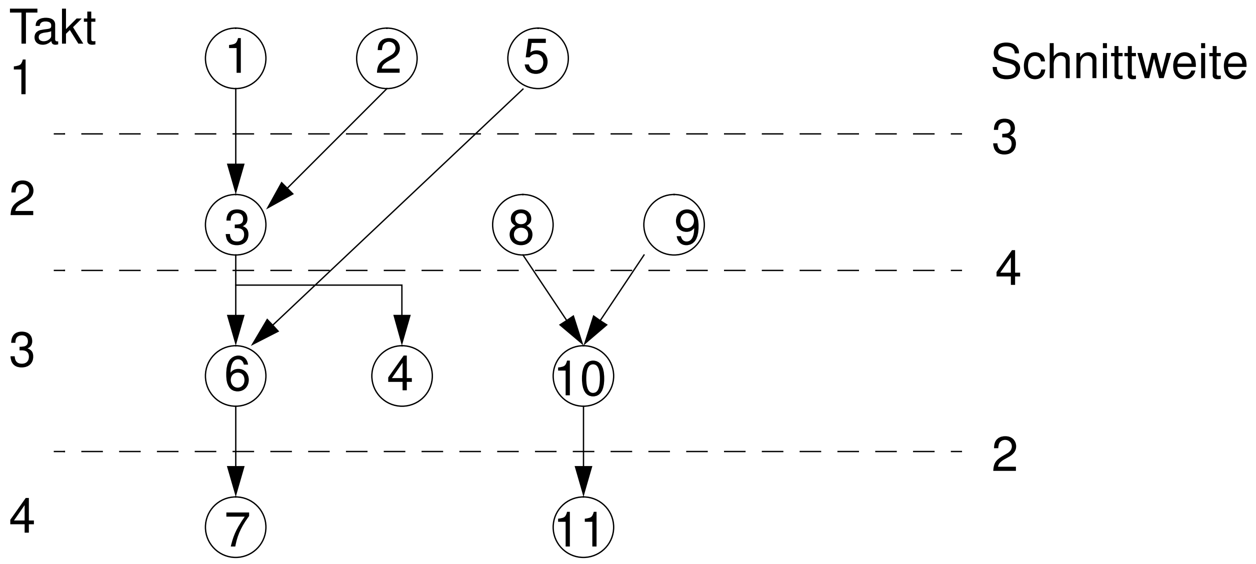
Ready-Liste enthält alle noch nicht angeordneten Operationen, deren Vorgänger alle angeordnet sind.

Iteriere bis alle Operationen angeordnet sind:

Wähle maximal k Operationen für den nächsten Takt aus (Heuristik).

aktualisiere Ready-Liste

Beispiel:



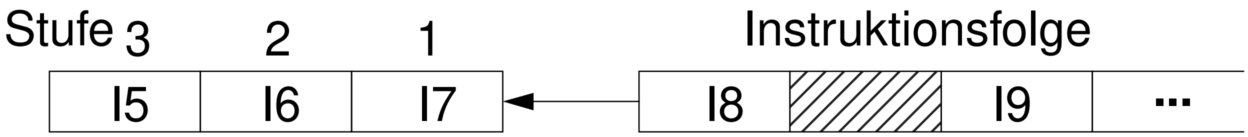
Schnitte im Graphen zeigen **Registerbedarf**.

Bei **unterschiedlichen Ausführungszeiten** von Operationen so auswählen, daß kein Konflikt mit angeordneten Vorgängern.

Algorithmus ist nur **für Bäume optimal**.

Instruktionsanordnung für Pipelining

Instruktions-Pipeline:



Abhängige Instruktionen müssen Mindestabstand haben (hier 2).

Algorithmus: z. B. modifiziertes List-Scheduling

Auswahl der nächsten Instruktion aus der Ready-Liste, so daß

- Mindestabstand zu vorangehenden Instruktionen eingehalten
- Instruktion hat viele Nachfolger (Heuristik)
- Instruktion hat langen Pfad zu Endknoten (Heuristik)

leer-Instruktion einfügen, wenn keine andere wählbar.

Annahme: symbolische Registerzuteilung; reale erfolgt später.

Beispiel:

ohne Umordnung

1:	t1	:= a
2:	t2	:= b
	leer	
3:	t3	:= t1 + t2
	leer	
4:	x	:= t3
5:	t4	:= c
	leer	
6:	t5	:= t3 + t4
	leer	
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
	leer	
10:	t8	:= t6 + t7
	leer	
11:	z	:= t8

mit Umordnung (gem. Alg.)

1:	t1	:= a
2:	t2	:= b
5:	t4	:= c
3:	t3	:= t1 + t2
8:	t6	:= d
9:	t7	:= e
6:	t5	:= t3 + t4
10:	t8	:= t6 + t7
4:	x	:= t3
7:	y	:= t5
11:	z	:= t8

Software-Pipelining

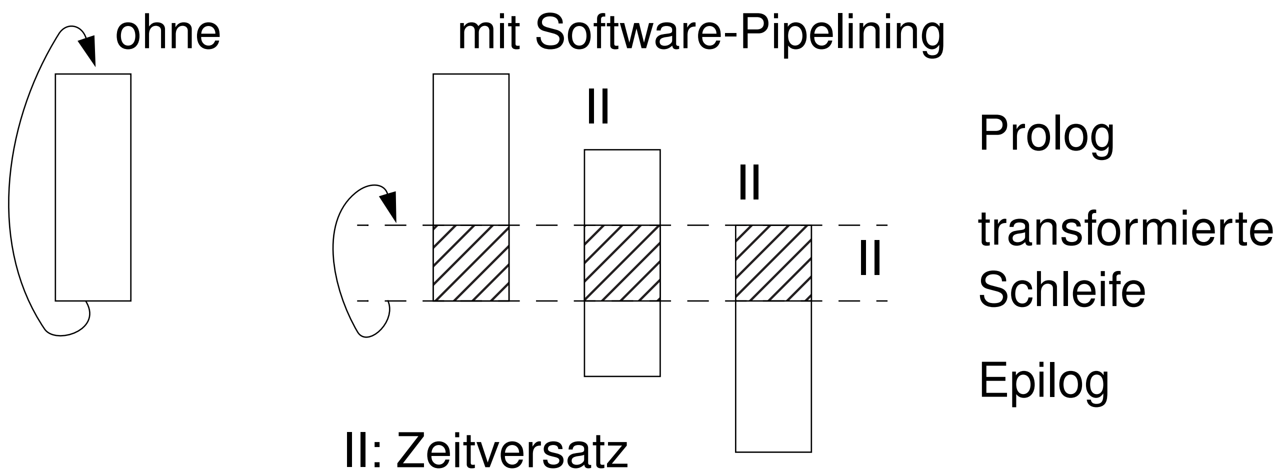
Technik zur **Parallelisierung von Schleifen**

Schleifenrumpf isoliert betrachtet hat wenig Parallelität, daher dünner Schedule.

Prinzip Software-Pipelining:

transformierter Schleifenrumpf bearbeitet mehrere Iterationen parallel, zeitversetzt, daher mehr Parallelität, dichter Schedule

Prolog, Epilog: Anlauf-, Auslauf-Code



II: Zeitversatz
(Initiation Interval)

Verfahren:

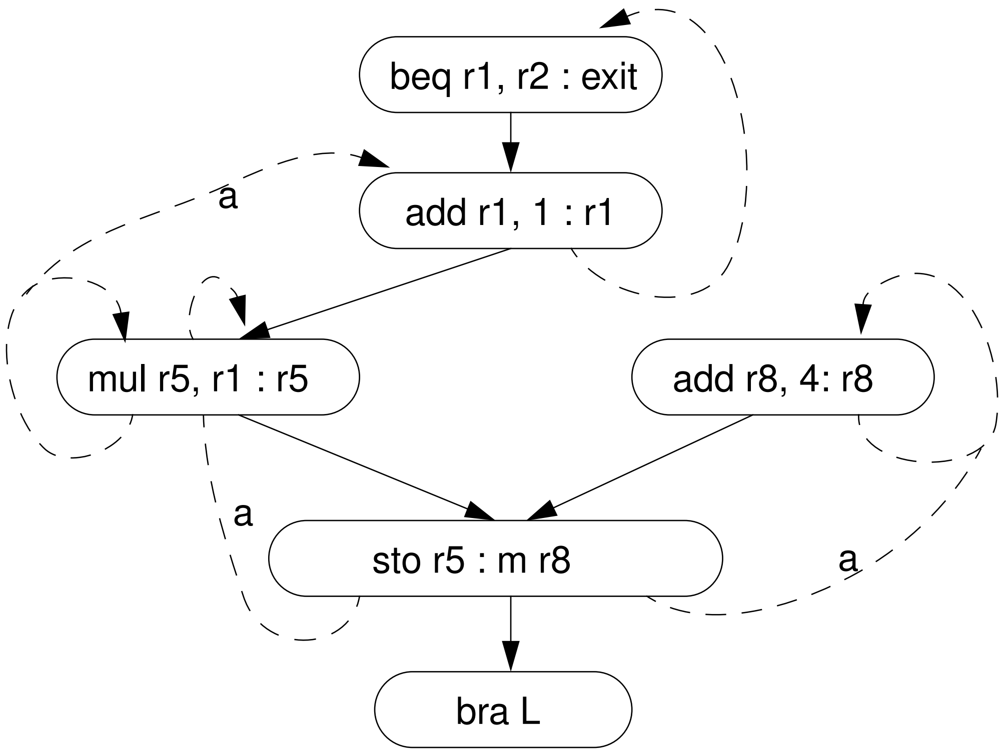
1. Abhängigkeitsgraph für Schleifenrumpf aufstellen;
   neu: Abhängigkeiten in nachfolgende Iterationen.
2. Schedule herstellen, so daß Iterationen mit möglichst kleinem Zeitversatz (II) überlagert werden können.
3. Neue Schleife, Prolog, Epilog herstellen.

Beispiel Software-Pipelining

Beispiel Fakultätsberechnung:

Programm	seq. Maschinencode
`i = 0; f = 1;`	
`while ( i != n)`	`L: beq r1, r2 : exit`
`{ i = i + 1;`	`add r1, 1 : r1`
`f = f * i;`	`mul r5, r1 : r5`
	`add r8, 4 : r8`
`m[i] = f;`	`sto r5 : m[r8]`
`}`	`bra L`

Datenabhängigkeitsgraph:



Abhängigkeit in nachfolgende Iteration

u a v Antiabhängigkeit:
u benutzt Wert bevor v ihn überschreibt

Software-Pipelining am Beispiel

Schedule der Schleife:

t	t_m		ADD	MUL	MEM	CTR
0	0	L:				beq r1, r2:exit
1	1		add r1, 1: r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		
3	1			... mul		
4	0				sto r5 : m r8	
5	1				... sto	
6	0					
7	1					bra L

Software-Pipeline mit II = 2, Prolog, Schleife, Epilog:

t	t_m		ADD	MUL	MEM	CTR
0	0					beq r1;r2:exit
1	1		add r1, 1 : r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		beq r1; r2 : ex
3	1		add r1, 1 : r1	... mul		
4	0	L:	add r8, 4 : r8	mul r5, r1 : r5	sto r5 : m r8	beq r1; r2 : ex
5	1		add r1, 1 : r1	... mul	... sto	bra L
6	1	ex:		... mul	... sto	
7	0				sto r5 : m r8	
8	1				... sto	
9	0					bra exit