Zwischendarstellungen und Programminformationen

Compiler–Technologie für das Reengineering

Compilerbau–Technologie

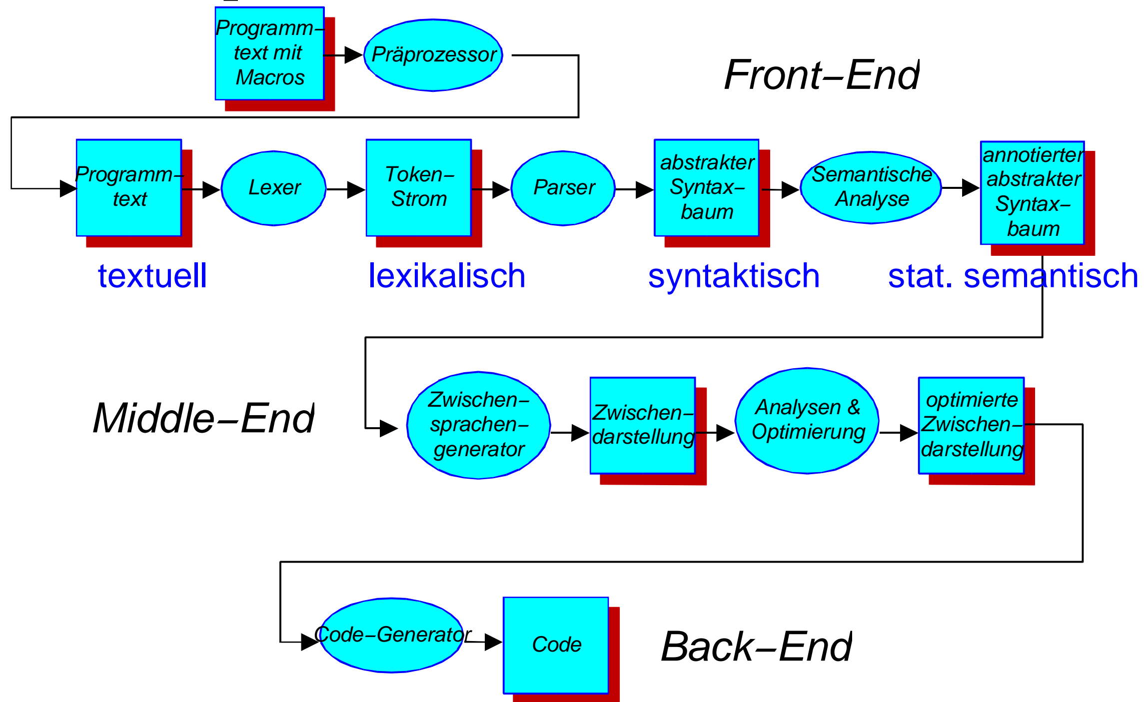
Lernziele

- Grundlagen aus Compilerbau für grundlegende Programmanalysen
- Unterschiedliche Anforderungen innerhalb des Reengineerings
- Verständnis der Abstraktionsebenen von Programmdarstellungen

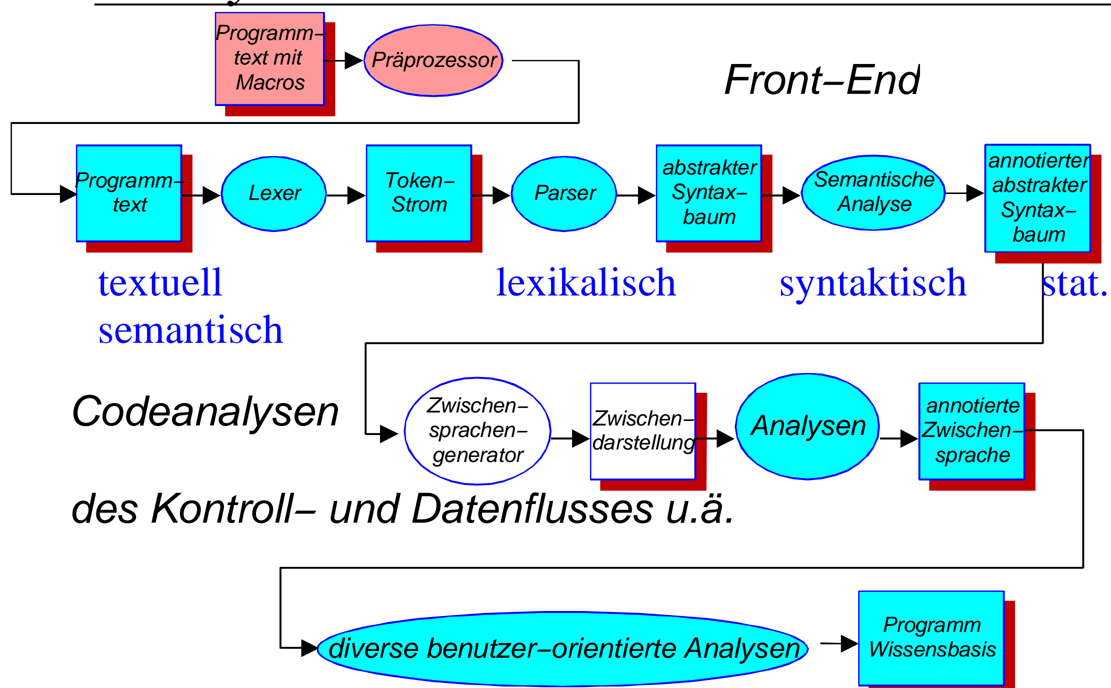
Kontext

- Grundlegende Programmanalysen sind Basis aller weiteren Analysen

Compiler–Struktur



Analysator–Struktur



Phasen eines Compilers

- Wissen über das Programm nimmt zu
  - syntaktische Dekomposition
  - semantische Attributierung
    - Namensbindung
    - Kontrollflussinformation
    - Datenflussinformation
- Abstraktion nimmt ab
  - abstrakter Syntaxbaum
  - maschinennahe einheitliche Zwischensprache, z.B. Register Transfer Language (RTL) von Gnu GCC
  - Maschinensprache

Unterschiede Compiler / Analysator

- lokal versus global
- optimistisch versus pessimistisch
- quellennah versus sprachenunabhängig

Der Tokenstrom

Für das Programmstück

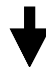
declare
 X: real;
begin
 X := A * 5;
end

liefert der Lexer neben der Quell–position die Lexeme (Token) durch Integer–Kennung der erkannten Kategorie und ggf. die Zeichenkette:

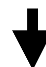
9 –– „declare“
4 –– id, „X“
7 –– „:“
4 –– id, „real“
5 –– „;“
6 –– „begin“
4 –– id, „X“
3 –– „:=“
4 –– id. „A“
8 –– „*“
10 –– int_lit, „5“
5 –– „;“
11–– „end“

Der Lexer

D	*[0–9]*
L	*[a–zA–Z_]*
"else"	*{ return(ELSE); }*
....	
*{L}({L}|{D})**	*{ yylval.id = register_name(yytext); return(ID); }*



Lexergenerator

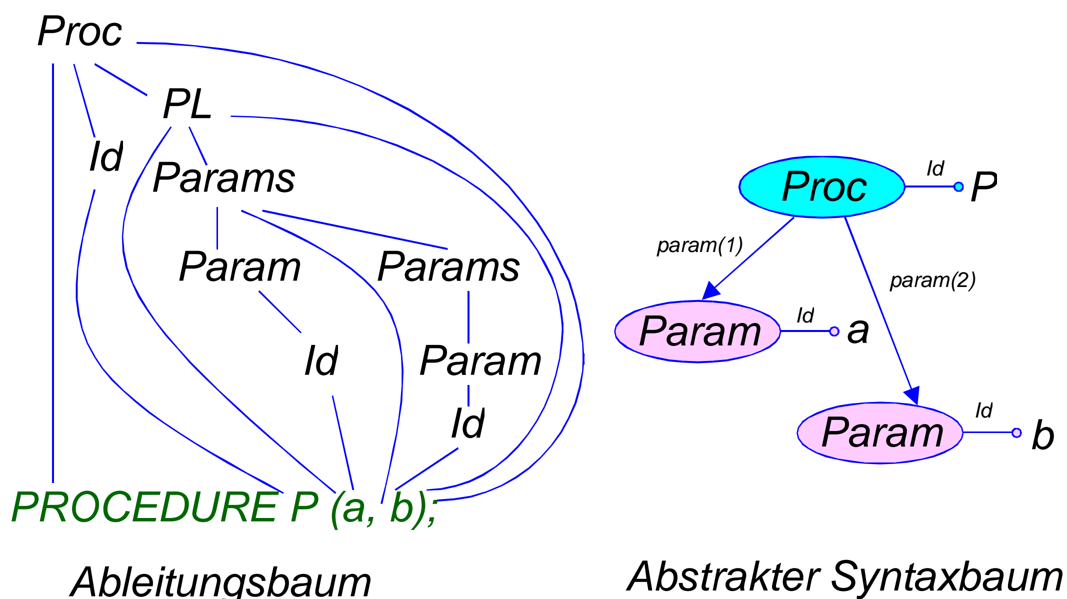


Lexer

Abstrakte Syntaxbäume

- Darstellung der syntaktischen Dekomposition des Eingabeprogramms
  - vereinfachter Ableitungsbaum: keine Kettenregeln, Sequenzen ersetzen Rekursionen etc.
  - syntaktische Kanten bilden einen Baum
- Eine formale Syntaxbeschreibung (BNF) legt die syntaktische Struktur fest:
  - Proc ::= PROCEDURE Id PL ";" .
  - PL ::= "(" Params ")" | e .
  - Params ::= Param "," Params | Param .
  - Param ::= Id .

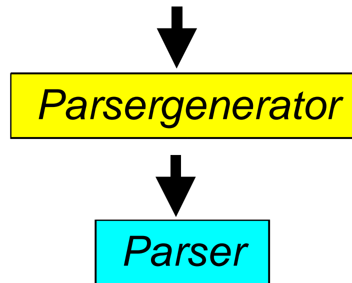
Ableitungsbaum / abstrakter Syntaxbaum



Der Parser

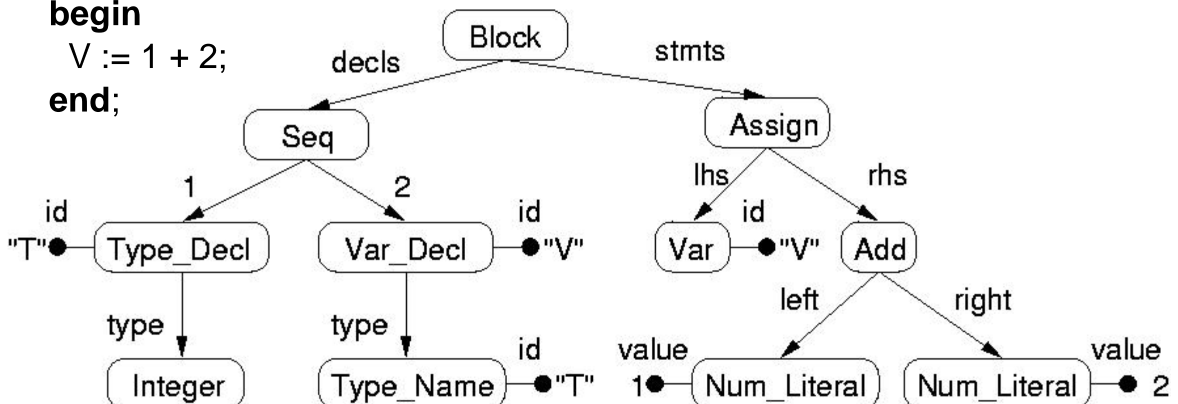
multiplicative_expression = –> [Tree:Standard.Tree.TTree] <

= LOp:multiplicative_expression Op:"" ROp:cast_expression*
{Tree := \Tree.Mbinary_op (Op:Position, LOp:Tree,
C_Operators.Op_Times, ROp:Tree);
} .



Beispiel eines ASTs

declare
type T **is new** Integer;
V : T;
begin
V := 1 + 2;
end;



Attributierter Syntaxbaum

Syntaktische Kanten des ASTs repräsentieren syntaktische Dekomposition.

- bilden einen Baum

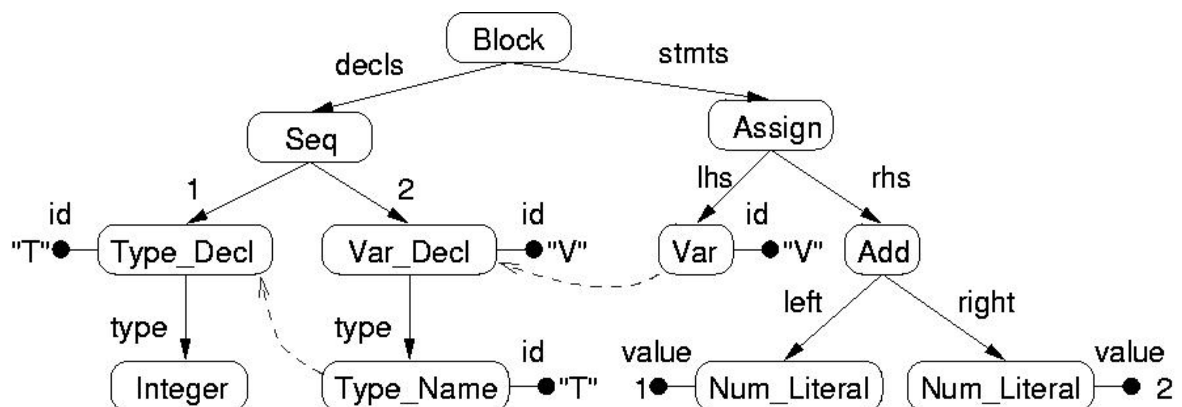
Semantische Analyse attributiert AST mit semantischen Informationen

- z.B. die Namens– und Typbindung.

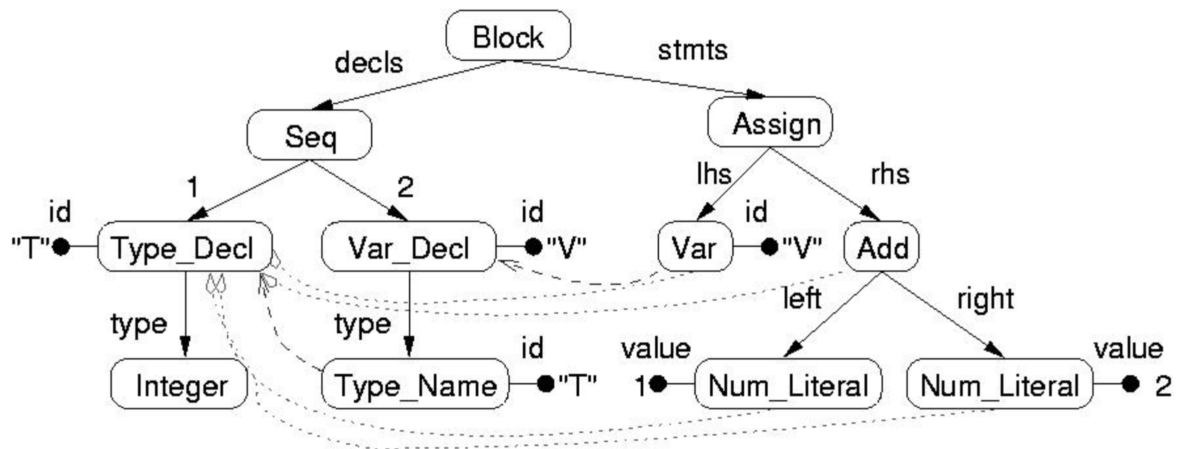
Semantische Kanten repräsentieren semantische Verweise auf andere Knoten.

- bilden einen allgemeinen Graphen
- Abstract Syntax Graph (ASG)

Namensbindung



Typinformation



Phasenspezifische Analysen (Abstraktionsebenen)

- Programmtext mit Makros
  - Suche nach Zeichenketten (einschließlich Makros)
- Programmtext nach Makro−Expansion
  - Suche nach Zeichenketten (ohne Makros)
- Token−Strom
  - Zeichenketten sind klassifiziert
  - Suche nach Tokens
- abstrakter Syntaxbaum
  - gibt syntaktische Dekomposition wieder
  - Suche nach syntaktischen Mustern; evtl. auch spezialisierte Attributberechnungen

Phasenspezifische Analysen (Abstraktionsebenen)

- attributierter abstrakter Syntaxbaum
  - Namensbindung und Typinformation verfügbar
  - Cross−Reference−Information
  - Suche nach spezifischen Variablen möglich
- Kontroll− und Datenflussinformation
  - Kontrollflussgraph
  - Aufrufgraph
    - explizit
    - Aufrufe über Funktionszeiger
  - Explizite Darstellung von Datenabhängigkeiten
  - Alias−Information
  - Points−To−Information

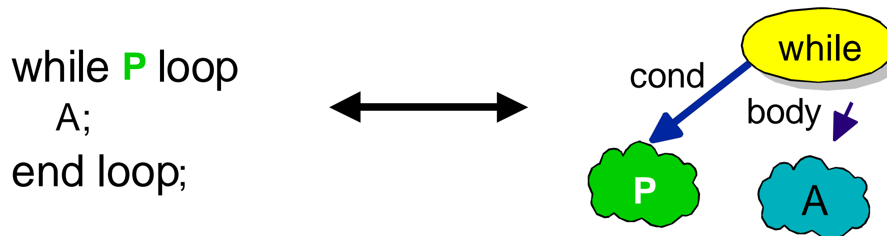
Anforderungen an Zwischendarstellungen

widersprüchliche Anforderungen

- möglichst quellennah, da Informationen an Wartungsprogrammierer ausgegeben werden sollen bzw. tatsächlicher Code wieder generiert werden soll
  - d.h. alle spezifischen Konstrukte müssen dargestellt werden
- möglichst vereinheitlicht, um das Schreiben von Programmanalysen für verschiedene Programmiersprachen zu vereinfachen
  - d.h. möglichst wenige, allgemeine Konstrukte

Quellennahe Betrachtung

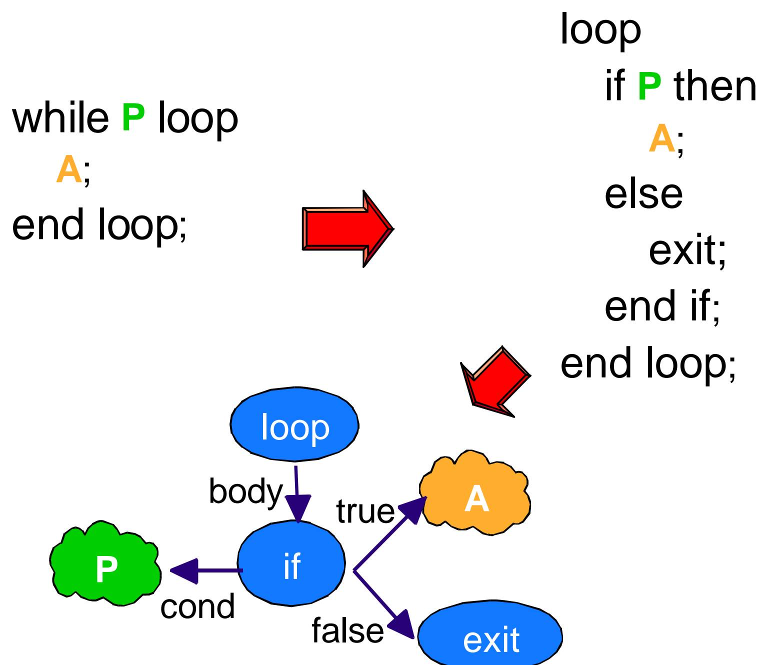
```
while P loop
   A;
end loop;
```



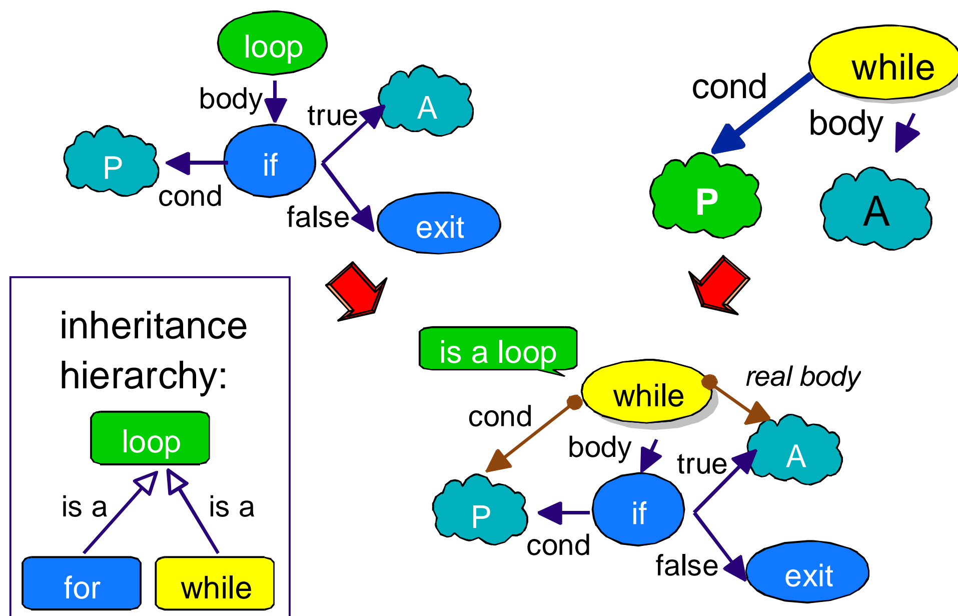
Einfache allgemeine Konstrukte

```
while P loop
   A;
end loop;
```

```
loop
   if P then
      A;
   else
      exit;
   end if;
end loop;
```



Vereinigung der Sichten



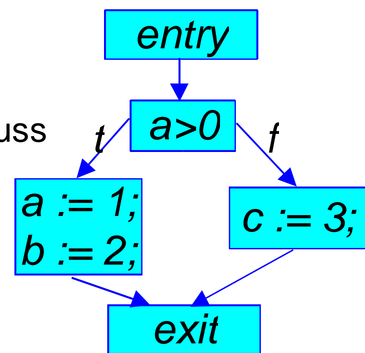
Zwischendarstellungen

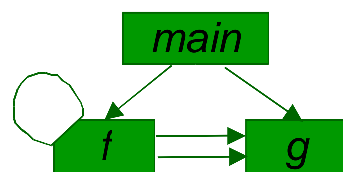
Im Folgenden präsentieren wir die wichtigsten Elemente der Darstellung dynamischer Semantik und deren Herleitung ....

- Kontrollflussanalysen
- Datenflussanalysen

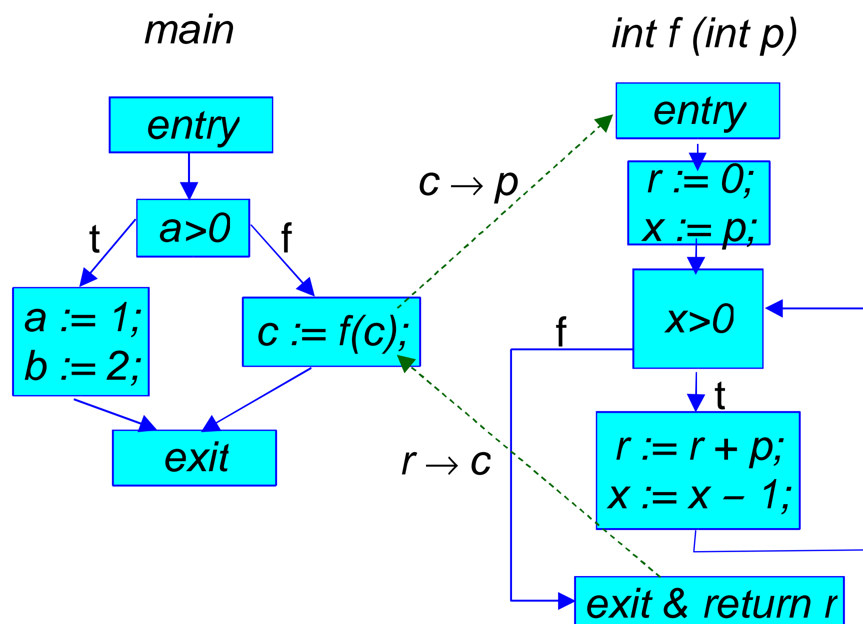
Kontrollflussinformation

- intraprozedural: Flussgraph
  - Knoten: Grundblöcke
  - Kanten: (bedingter/unbedingter) Kontrollfluss
  - ergibt sich aus syntaktischer Struktur und etwaigen Gotos, Exits, Continues, etc.
- interprozedural: Aufrufgraph
  - Multigraph
  - Knoten: Prozeduren
  - Kanten: Aufruf
  - ergibt sich aus expliziten Aufrufen im Programmcode sowie Aufrufe über Funktionszeiger





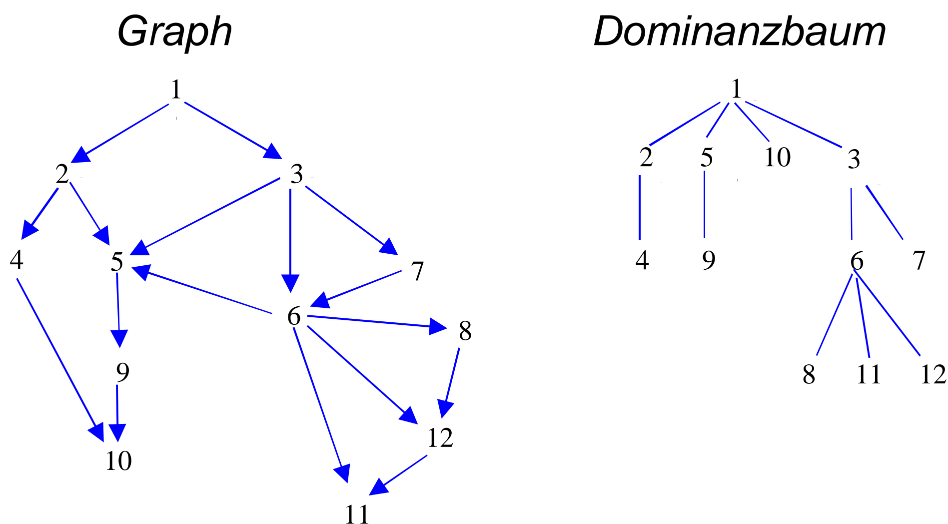
Intra– und interprozed. Kontrollfluss



Dominanz

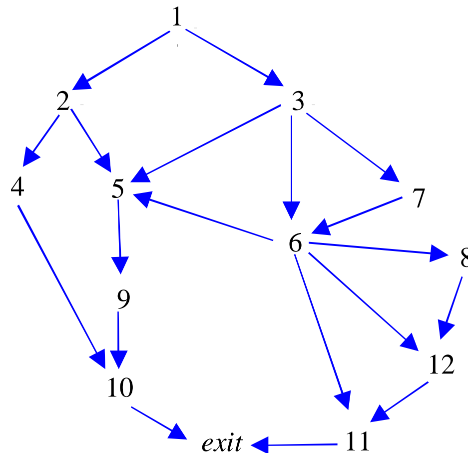
- Fragestellungen:
  - Welche Prozeduren sind lokal zueinander?
    - Genauer: Gibt es eine Prozedur D, über nur die ein Aufruf von N erfolgt?
  - Welcher Block D im Flussgraph muss in jedem Falle passiert werden, damit Block N ausgeführt werden kann?
- Antwort: D ist der Dominator von N
  - Ein Knoten D **dominiert** einen Knoten N, wenn D auf allen Pfaden vom Startknoten zu N liegt.
  - Ein Knoten D ist der **direkte Dominator** von N, wenn
    - D dominiert N und
    - alle weiteren Dominatoren von N dominieren D

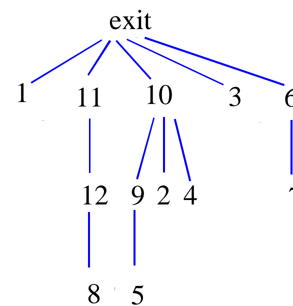
Dominanz



Postdominanz

- Ein Knoten D postdominiert einen Knoten N, wenn jeder Pfad von N zum Endknoten den Knoten D enthält (entspricht Dominanz des umgekehrten Graphen).





Kontrollabhängigkeit

Kontrollabhängigkeit = Bedingung B, von welcher die Ausführung eines anderen Knotens X abhängt.

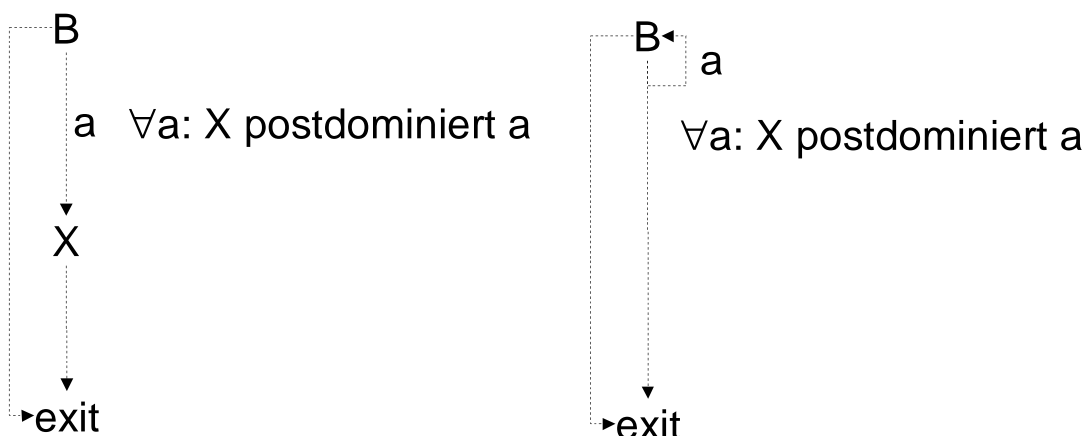
- B muss mehrere direkte Nachfolger haben
- B hat einen Pfad zum Endknoten, der X vermeidet (d.h. B kann nicht von X postdominiert werden)
- B hat einen Pfad zu X, d.h. insgesamt hat B mindestens 2 Pfade:
  - einer führt zu X
  - einer umgeht X
- B ist der letzte Knoten mit einer solchen Eigenschaft

Kontrollabhängigkeit

Ein Knoten X ist kontrollabhängig von einem Knoten B genau dann, wenn

- es einen nicht–leeren Pfad von B nach X gibt, so dass X jeden Knoten auf dem Pfad (ohne B) postdominiert
- entweder X = B oder X postdominiert B nicht

Kontrollabhängigkeit

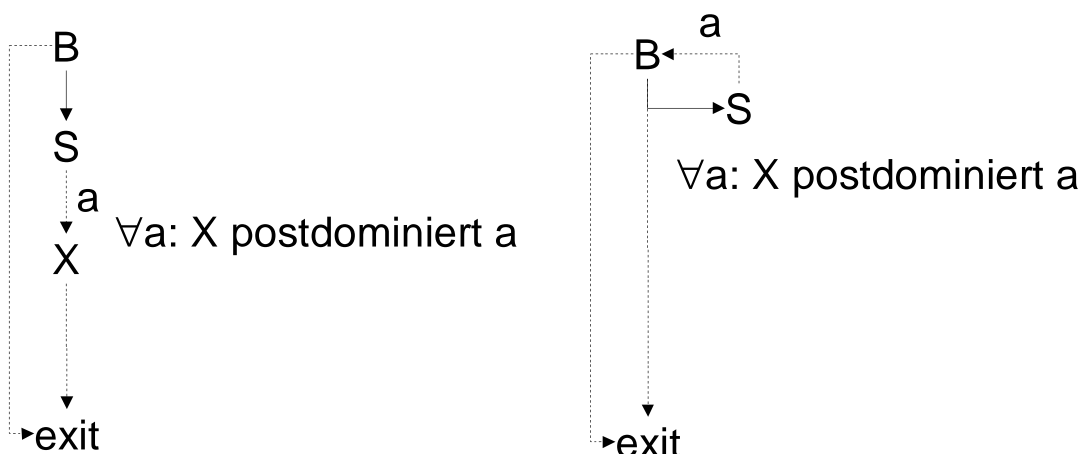


Kontrollabhängigkeit

Ein Knoten X ist kontrollabhängig von einer Kante (B, S) genau dann, wenn

- es einen nicht–leeren Pfad beginnend mit (B, S) nach X gibt, so dass X jeden Knoten auf dem Pfad (ohne B) postdominiert
- entweder X = B oder X postdominiert B nicht

Kontrollabhängigkeit

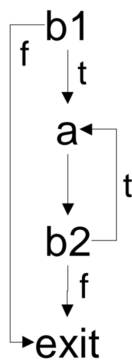


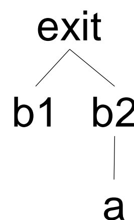
Kontrollabhängigkeit

```
function CD ((B, S) : edge) return list of nodes
begin -- control dependency
   depends := ∅;
   X := S;
   while X ≠ pdom (B) loop
        add X to depends;
        X := pdom (X);
   end loop;
   return depends;
end CD;
```


Beispiel für Kontrollabhängigkeit

```
if b1 then
  repeat
    a;
  until b2;
end if;
```





(b1, a)	a, b2
(b1, exit)	–
(b2, a)	a, b2
(b2, exit)	–

Kontrollabhängigkeit

Für strukturierte Programme:

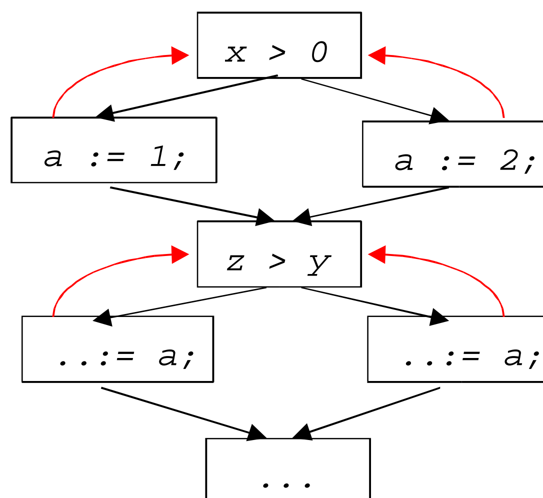
- eine Anweisung ist kontrollabhängig von der Bedingung der nächstumgebenden Schleife oder bedingten Anweisung.

```
while a loop
   if b then
     x := y;  –– kontrollabhängig von b
   end if;
   z := 1;   –– kontrollabhängig von a
end loop;
```

N.B.: Bedingungen in repeat–Schleifen sind von sich und dem umgebenden Konstrukt abhängig (siehe voriges Beispiel).

Repräsentation der Kontrollabhängigkeit

```
if x > 0 then
        a := 1;
else
        a := 2;
end if;
if z > y then
        z := a;
else
        y := a;
end if;
```



Kontroll–
abhängigkeit

Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

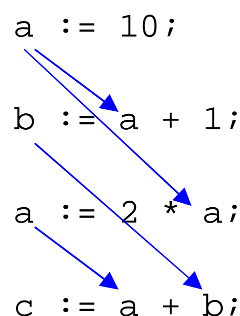
Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:

- Set–Use Beziehung (Datenabhängigkeit)
  - A setzt den Wert, der von B verwendet wird
- Use–Set Beziehung (Anti–Dependency)
  - A liest den Wert und B überschreibt ihn danach
- Set–Set Beziehung (Output–Dependency)
  - der von A gesetzte Wert wird von B überschrieben

Codetransformationen müssen diese Beziehungen erhalten.

Datenabhängigkeit (Set–Use)

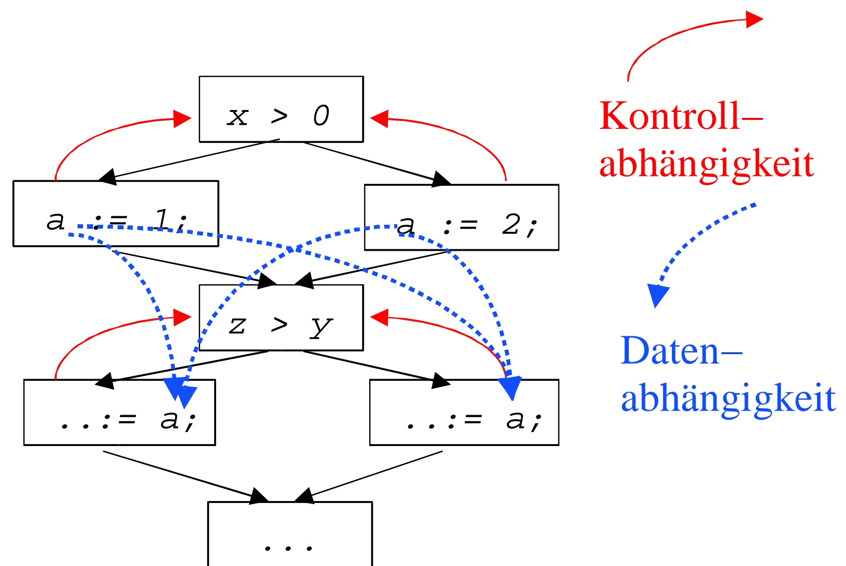
Für die Zwecke der Analyse ist die Set–Use Beziehung die wichtigste Beziehung und wird intern oft explizit repräsentiert (meist in der umgekehrten Richtung).



Zwischen der 2. und 3. Anweisung besteht eine „Use-Set“, zwischen der 1. und 3. eine „Set–Set“–Beziehung.

Repräsentation der Datenabhängigkeit

```
if x > 0 then
        a := 1;
else
        a := 2;
end if;
if z > y then
        z := a;
else
        y := a;
end if;
```

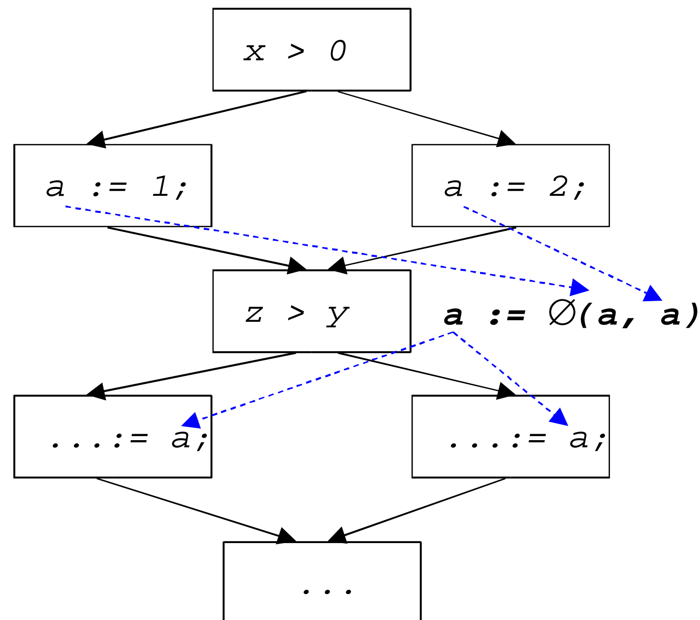


Repräsentation der Datenabhängigkeit

Wunsch nach einer kompakteren Darstellung:
Nur eine Definition für jede Verwendung.

- Static Single Assignment Form (SSA)
- künstliche Zuweisungen, sogenannte ∅–Knoten, werden eingefügt an den Knoten, an denen Kontrollflüsse zusammenlaufen, auf denen verschiedene Definitionen der gleichen Variablen liegen.
- die entsprechenden Stellen lassen sich effizient identifizieren (Cytron et al., 1991)

Static Single Assignment Form



Beispiel der SSA

```
x, y, z : T;
if ...   then y := 4; x := 5;
         else y := 3; end if;

while ... loop
   if y > 0 then z := x + z; end if;
   if y <= 0 then z := y; end if;
end loop;
... = z;
```

Beispiel der SSA

x, y, z : T; x_0 := init; y_0 := init; z_0 := init;

if ... **then** y_1 := 4; x_1 := 5;
else y_2 := 3; **end if**;

$x_2 := \varnothing(x_0, x_1)$; $y_3 := \varnothing(y_1, y_2)$;

while $z_1 := \varnothing(z_0, z_5)$; ... **loop**

if $y_3 > 0$ **then** $z_2 := x_2 + z_1$; **end if**;
$z_3 := \varnothing(z_1, z_2)$;
if $y_3 <= 0$ **then** $z_4 := y_3$; **end if**;
$z_5 := \varnothing(z_3, z_4)$;

end loop;

... = z_1;

Beispielanwendung der SSA

Finde alle potenziell lokal uninitialisierten Variablen

- (füge am Entry–Knoten eine Pseudodefinition für jede lokale Variable ein)
- für jede Pseudo–Definition D:
  - propagiere D längs der Use–Kanten und markiere alle dabei erreichten Verwendungen von Variablen
  - jede markierte Verwendung einer Variablen ist potenziell undefiniert

Beispielanwendung der SSA

```
x, y, z : T; x0 := init; y0 := init; z0 := init;
if ... then y1 := 4; x1 := 5;
   else y2 := 3; end if;
x2 := ∅ (x0, x1); y3 := ∅ (y1, y2);
while z1 := ∅ (z0, z5); ... loop
   if y3 > 0 then z2 := x2 + z1; end if;
   z3 := ∅ (z1, z2);
   if y3 <= 0 then z4 := y3; end if;
   z5 := ∅ (z3, z4);
end loop;
... = z1;
```


Wichtige Mengen beim Aufbau von SSA

MustMod (A):

Menge von Variablen, die sicher von A verändert werden

MayMod (A):

Menge von Variablen, die möglicherweise von A verändert werden

MayUse (A):

Menge von Variablen, die möglicherweise von A verwendet werden

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:

  x := ...; –– falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

```
y := 5;
if ... then x := 6; end if;

x, y := Φ (y, x);

... = y
```


Aliasing–Varianten

- parameter–induziertes Aliasing durch Referenzparameter
  - p (x, x) *(bzw. p(&x, &x) in C und C++)*
  - p (g), wobei g eine globalere Variable ist und in p verwendet wird
  - Lösung: Propagierung der Alias–Info über den Call–Graphen
  - (Ansonsten konservative Annahme: zwei Referenzparameter bzw. globale Variable/Referenzparameter sind Aliase; eventuell kann Typinformation helfen)

Aliasing−Varianten

- Arraykomponenten
  - a [2 * i] und a [j]
  - Lösungsansatz: Löse Gleichung 2*i = j
  - Ansonsten: Modelliere Zuweisungen an Teilkomponente als Zuweisung an gesamtes Array: Partial Update/Read m.a.W. konservative Annahme, dass jeder Index sich auf alle Array−Elemente beziehen kann.

Aliasing−Varianten

- stack−gerichtete Zeiger
  - p := &x;
  - Lösungsansatz: Points−To−Analyse
  - Konservative Annahme: *p könnte jede statische Variable betreffen, deren Adresse genommen wird.
- heap−gerichtete Zeiger
  - p := new T; q := new T; q.next = p;
  - Lösungsansatz: „Shape−Analyse“
  - Konservative Annahme: *p kann jedes Element des Heaps betreffen.

Probleme beim Aufbau von SSA

Aufrufe von Unterprogrammen

- welchen äußeren Effekt haben sie?
  x := 1;
  f (a, b);
  –– hat x noch den Wert 1?
- Lösung: Globale Analysen auf Seiteneffekte
  (aber: Problem von Bibliotheksroutinen, deren Quell–Code nicht vorhanden ist)
- Ansonsten konservative Annahme: Aufruf eines Unterprogramms U verändert alle globalen Variablen (nicht nur jene im Sichtbarkeitsbereich von U) sowie jeden Referenzparameter
- optimistische Annahme: ...verändert nur seine Referenzparameter und über sie erreichbare Variable

Weiterführende Literatur

- Robert Morgan, '*Building an Optimizing Compiler*', Addison–Wesley
  - beschreibt Zwischendarstellungen sowie Kontroll– und Datenflussanalysen
- R. Koschke, J.–F. Girard, M. Würthner, 'An Intermediate Representation for Reverse Engineering Analyses', *Proc. of the Working Conference on Reverse Engineering*, IEEE Computer Society Press, 1998.
  - beschreibt Zwischendarstellung für das Reverse Engineering und dessen spezielle Anforderungen