Martin Glinz Harald Gall

Software Engineering

Wintersemester 2006/07

Kapitel 11

Statische Analyse

Universität Zürich
Institut für Informatik

11.1 Grundlagen

11.2 Analyse von Programmen

11.3 Darstellungen für die Programmanalyse

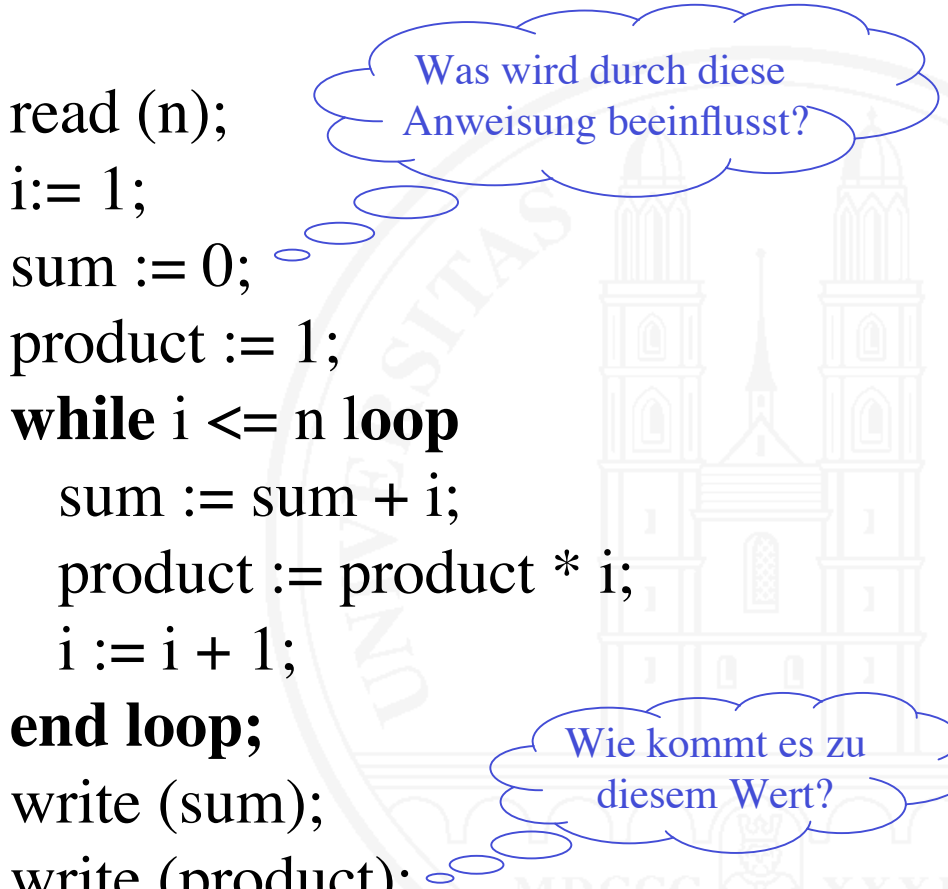
11.4 Program Dependency Graph (PDG)

Grundlagen

Statische Analyse ist die Untersuchung des statischen Aufbaus eines Prüflings auf die Erfüllung vorgegebener Kriterien.

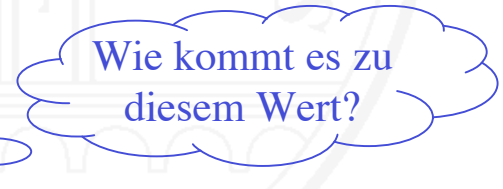
- Die Prüfung ist statisch. Sie grenzt sich damit vom (dynamischen) *Test* ab.
- Sie ist formal durchführbar und damit automatisierbar. Sie grenzt sich damit vom *Review* ab.
- Ziele
  - Bewertung von Qualitätsmerkmalen, z.B. Pflegbarkeit, Testbarkeit aufgrund struktureller Eigenschaften des Prüflings (zum Beispiel Komplexität, Anomalien)
  - Erkennung von Fehlern und Anomalien in neu erstellter Software
  - Erkennung und Analyse von Programmstrukturen bei der Pflege und beim Reengineering von Altsystemen (*legacy systems*)
  - Prüfen, ob ein Programm formale Vorgaben (zum Beispiel Codierrichtlinien,Namenskonventionen, etc.) einhält

Das tägliche Brot des Programmierers

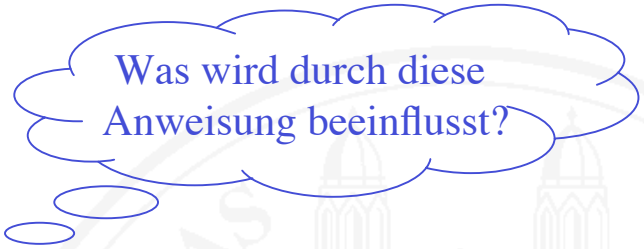


Programmanalyse /1

```
read (n);
i:= 1;
sum := 0;
product := 1;
while i <= n loop
   sum := sum + i;
   product := product * i;
   i := i + 1;
end loop;
write (sum);
write (product);
```



Programmanalyse /2




```
read (n);
i:= 1;
sum := 0;
product := 1;
while i <= n loop
   sum := sum + i;
   product := product * i;
   i := i + 1;
end loop;
write (sum);
write (product);
```

11.2 Analyse von Programmen

- Syntax
  - Analyse durch den Compiler

- Datentypen, Typverträglichkeit
  - typ-unverträgliche Operationen
  - typ-unverträgliche Zuweisungen
  - typ-unverträgliche Operationsaufrufe

Programm- und Datenstrukturen

- Formale Eigenschaften
  - Richtlinien eingehalten
  - Art und Menge der internen Dokumentation
- Strukturkomplexität des Progamms
  - Einhaltung von Strukturierungsregeln
  - Art und Tiefe von Verschachtelungen
- Strukturfehler bzw. -anomalien
  - nicht erreichbarer Code
  - vorhandene, aber nicht benutzte Operationen
  - benutzte, aber nicht vorhandene Operationen
- Fehler / Anomalien in den Daten
  - nicht deklarierte Variablen
  - nicht benutzte Variablen
  - Benutzung nicht initialisierter Variablen

Vorgehen

- Bei Programmen immer mit Werkzeugen
  - Compiler
  - syntaxgestützte Editoren
  - spezielle Analyse-Werkzeuge
- Programm wird wie bei der Übersetzung durch ein Werkzeug zerlegt (parsing) und in einer analysefreundlichen internen Struktur repräsentiert
- Dabei werden Syntaxfehler erkannt und erste Kenngrößen (z. B. Anzahl Codezeilen, Anzahl Kommentare) ermittelt
- Die resultierende interne Repräsentation des Programms wird verschiedenen Analysen unterworfen (z.B. statische Aufrufhierarchie, Strukturkomplexität, Datenflussanalyse)
- Die Befunde werden gesammelt, ggf. verdichtet und bewertet

Weitere Analysen

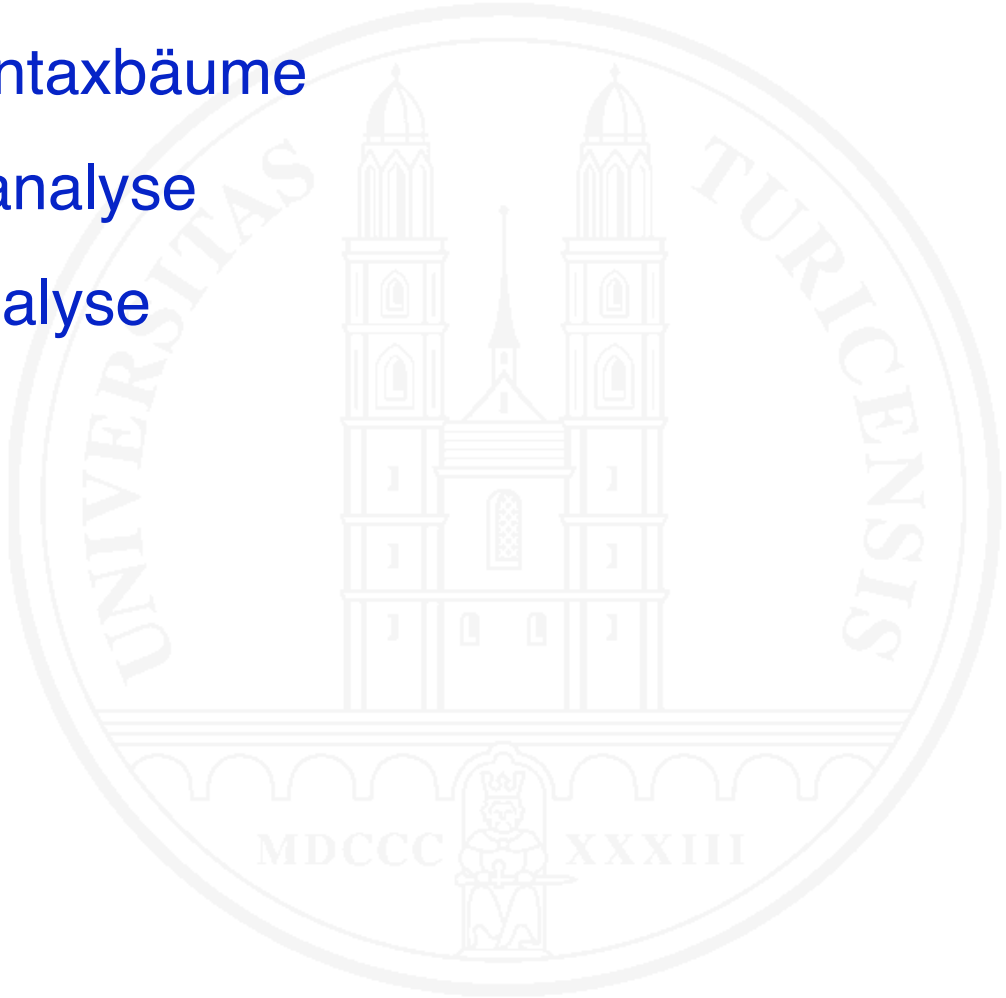
- Analyse von Algorithmen
  - Laufzeiteffizienz
  - Speichereffizienz
  - Gültigkeitsbereich
  - Erfolgt weitestgehend manuell
- Analyse von Spezifikationen, Entwürfen, Prüfvorschriften
  - Syntaxanalyse der formal beschriebenen Teile
  - Teilweise Struktur- und Flussanalysen (je nach verwendeter Modellierungsmethode)
  - Analyse der Anforderungsverfolgung, wenn dokumentiert ist, welche Anforderung wo umgesetzt bzw. geprüft wird (werkzeuggestützt möglich)

11.3 Darstellungen für die Programmanalyse

Abstrakte Syntaxbäume

Kontrollflussanalyse

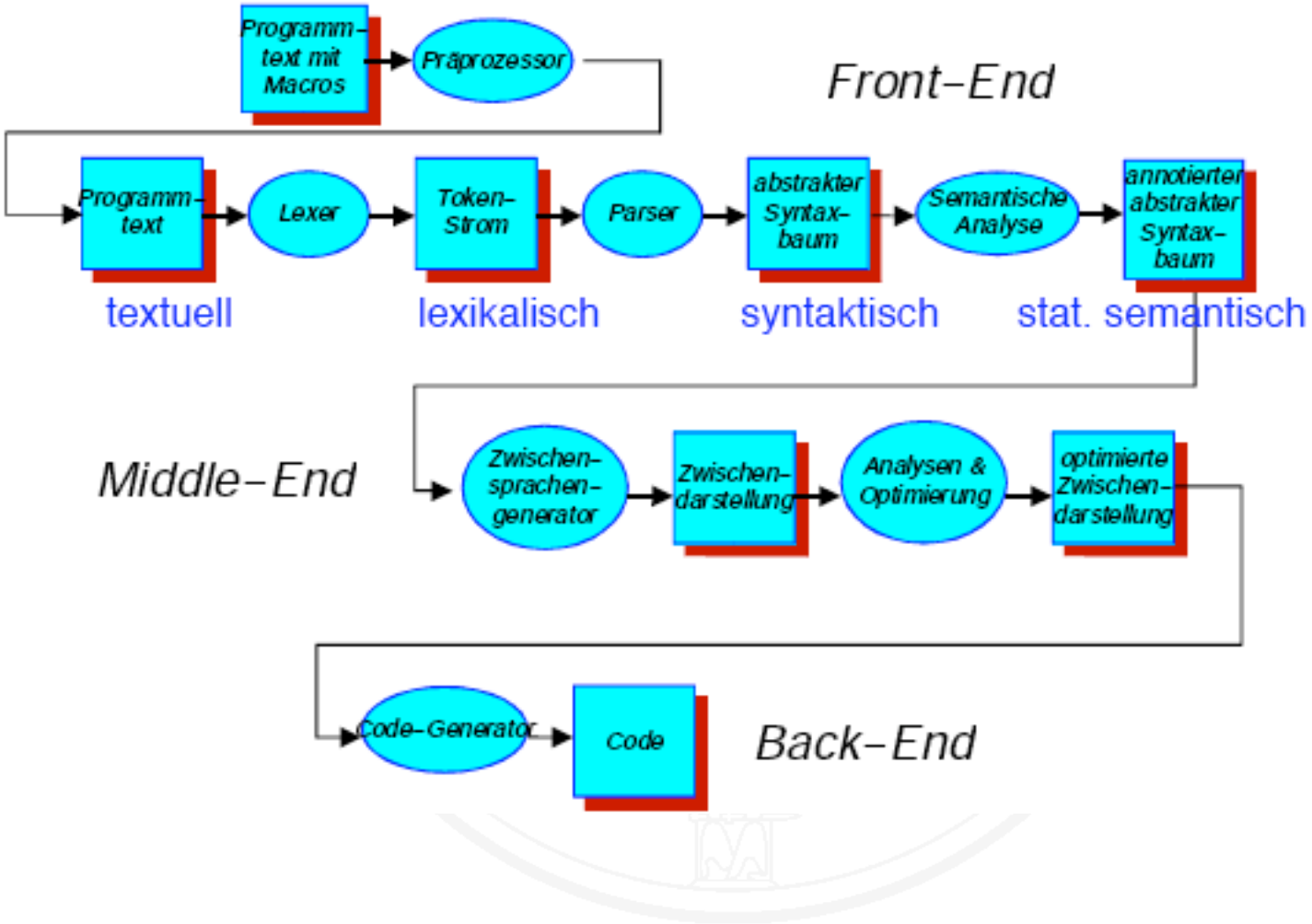
Datenflussanalyse



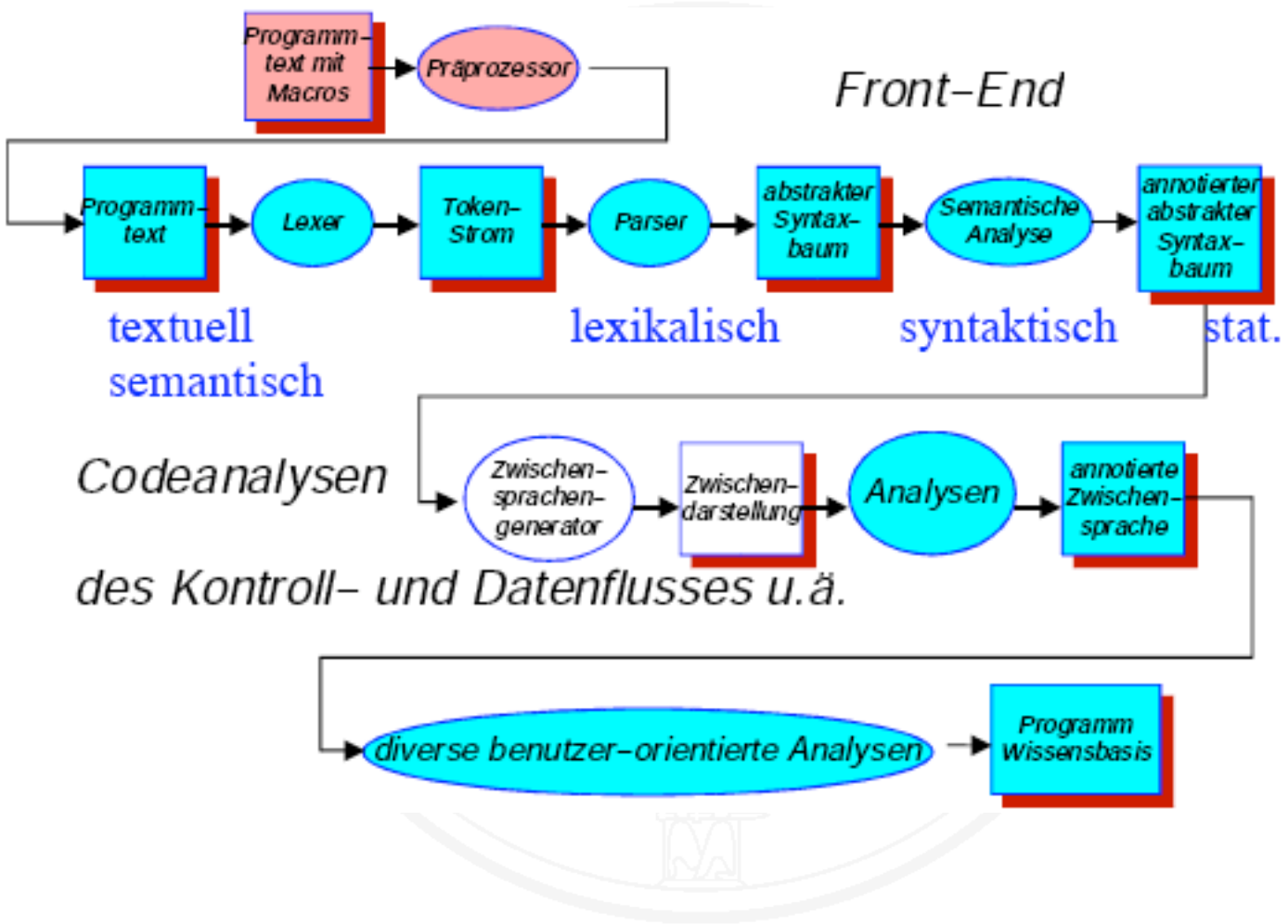
Kurzüberblick

- Lernziele
  - Grundlagen für Programmanalysen
  - Unterschiedliche Anforderungen innerhalb des (Re)Engineerings
  - 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

Der Tokenstrom

Für das Programmstück

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

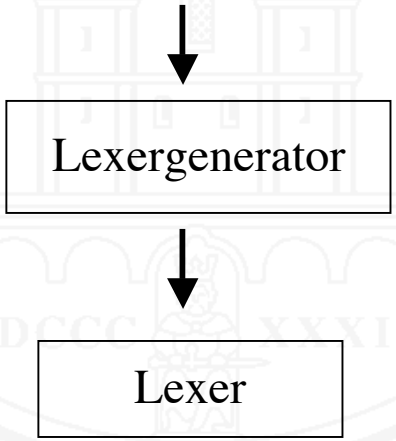
liefert der Lexer neben der Quellposition die Lexeme (Token) durch Integer-Kennung der erkannten Kategorie und ggf. die Zeichenkette:

```
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”
```

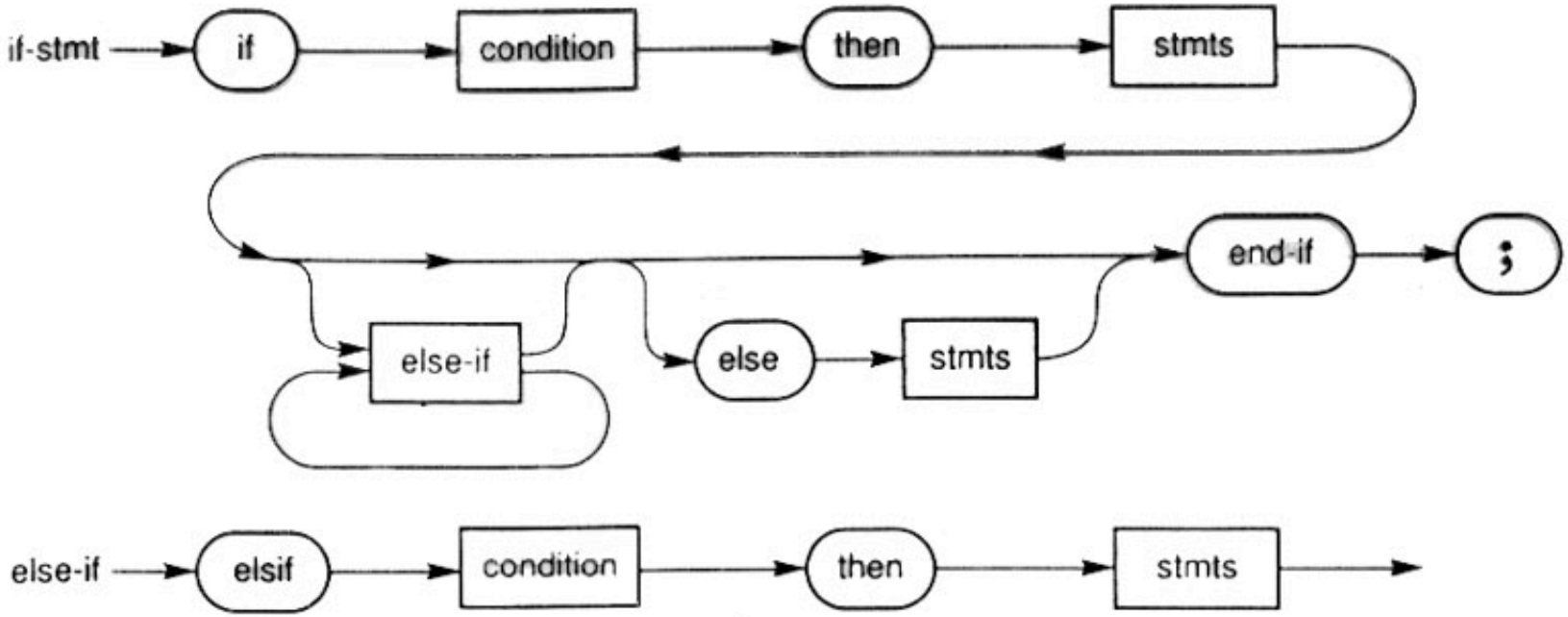
Der Lexer

Grammatik

Digit	[0-9]
Literal	[a-z,A-Z]
“else”	{ return(ELSE); }
…	
{L} ({L} | {D})*	{ yylval.id = register_name(yytext); return (ID); }



Syntaxdiagramm

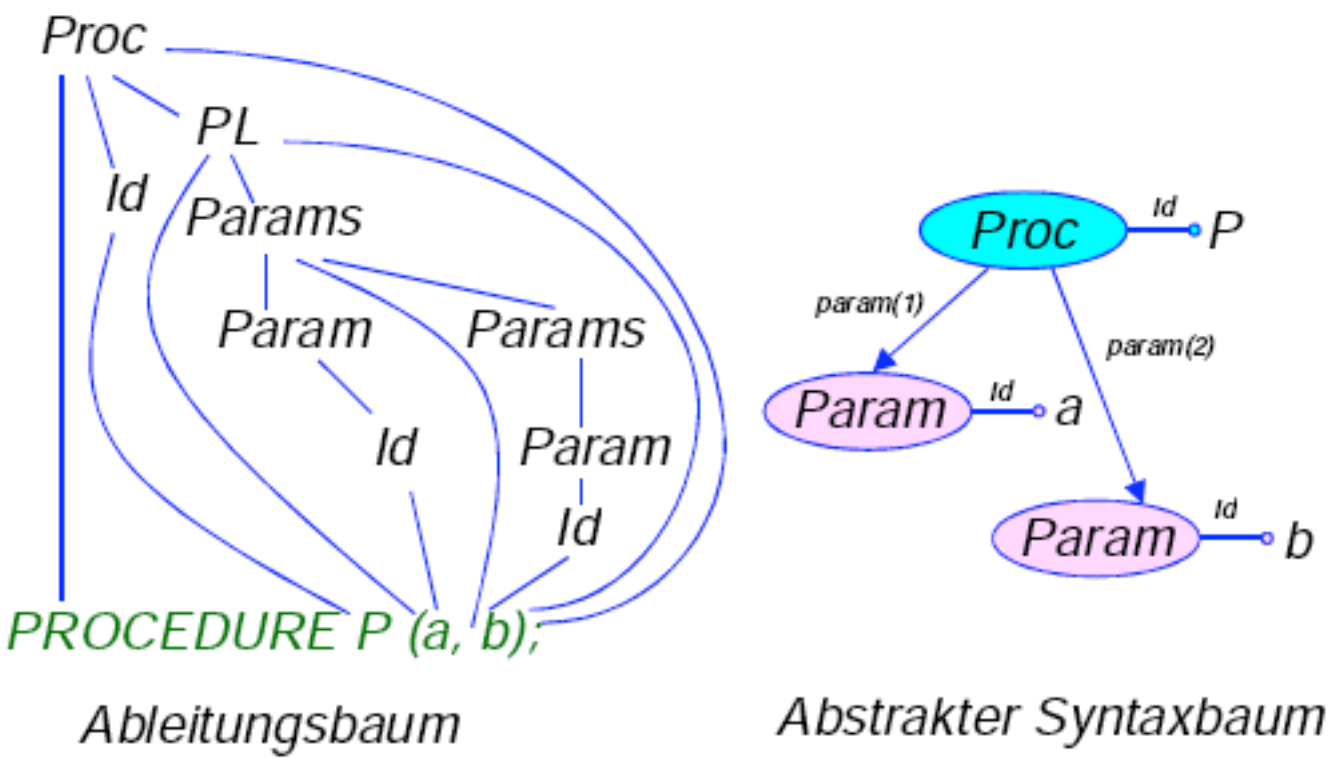


Abstrakte Syntaxbäume (AST)

- 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 .

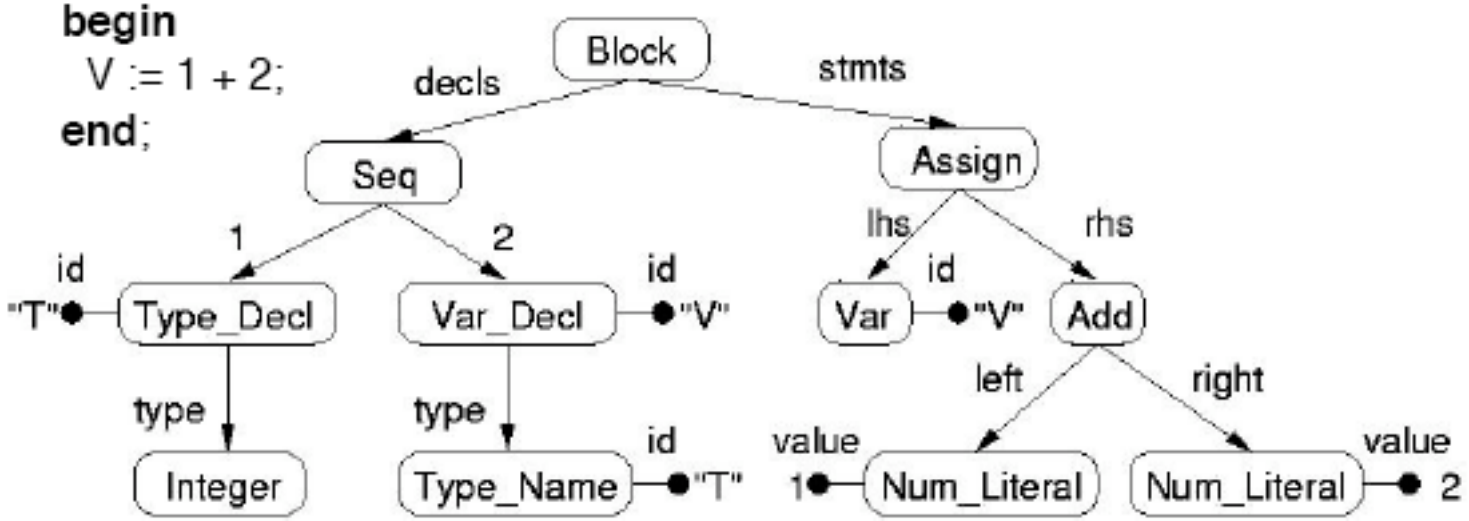
BNF .. Backus-Naur Form
EBNF .. Extended BNF

Ableitungsbaum / abstrakter Syntaxbaum



Beispiel eines ASTs

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

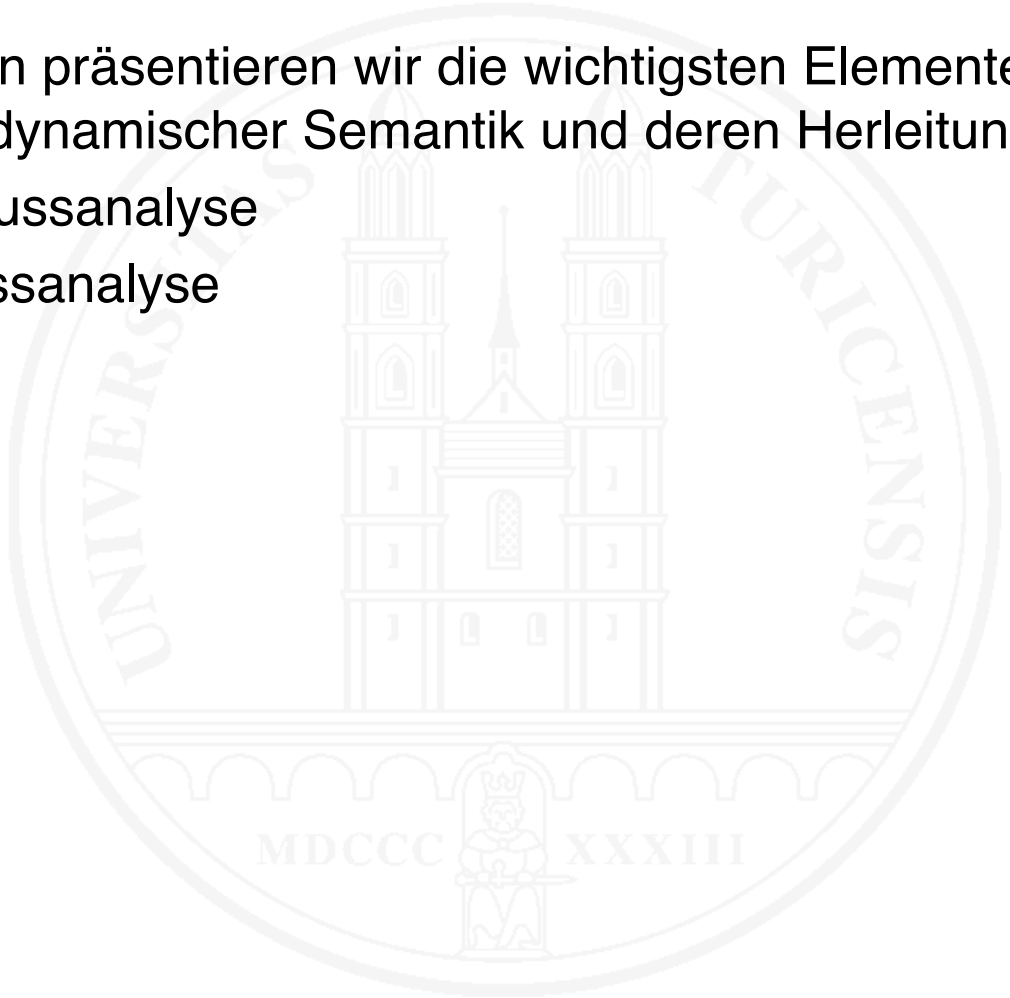


Anforderungen an Zwischendarstellungen

- 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 Programmier-sprachen zu vereinfachen
    - d.h. möglichst wenige, allgemeine Konstrukte

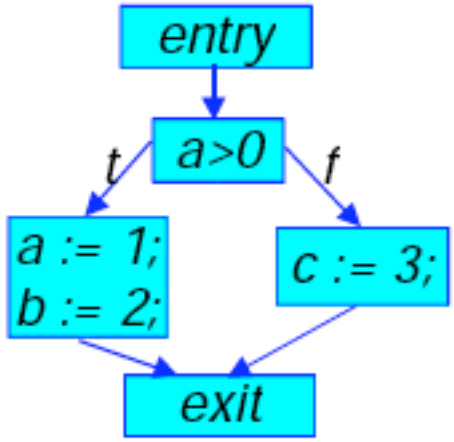
Zwischendarstellungen

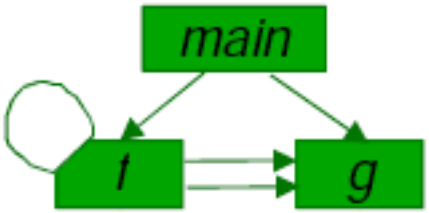
- Im Folgenden präsentieren wir die wichtigsten Elemente der Darstellung dynamischer Semantik und deren Herleitung ...
  - Kontrollflussanalyse
  - Datenflussanalyse



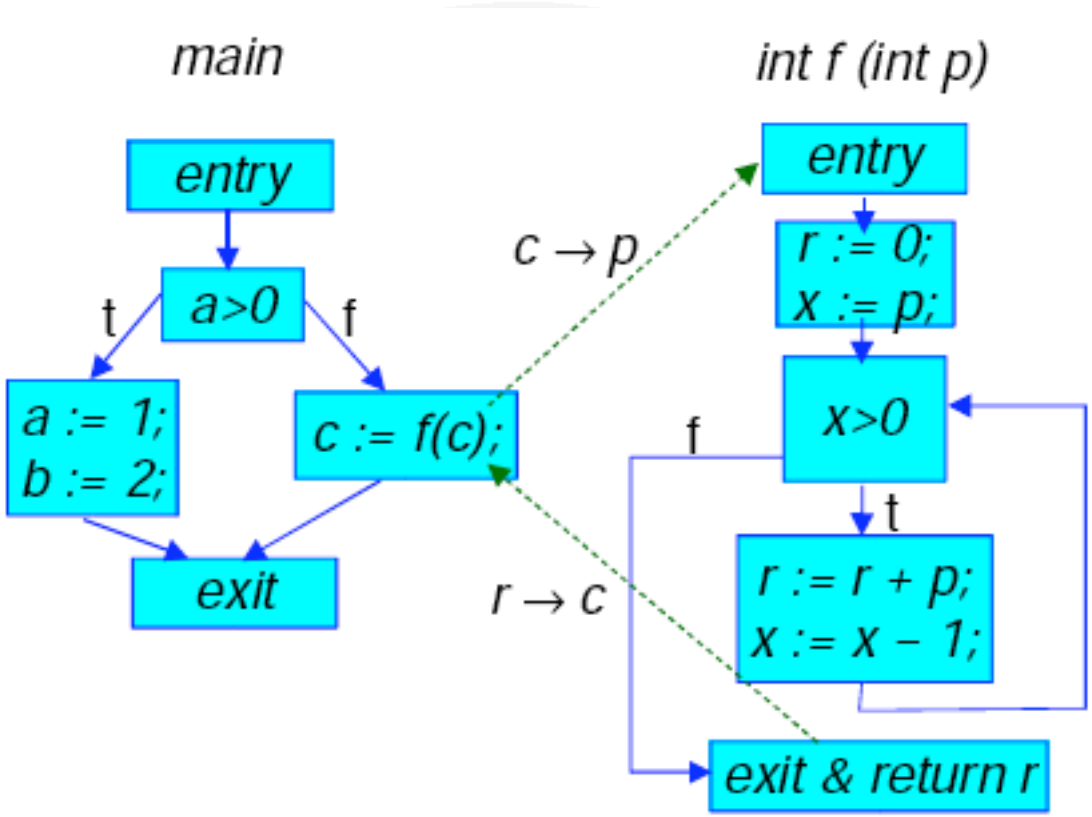
Kontrollflussinformation

- intra-prozedural: Flussgraph
  - Knoten: Grundblöcke
  - Kanten: (bedingter/unbedingter Kontrollfluss
  - ergibts sich aus syntaktischer Struktur und etwaigen Goto's, Exit's, Continue's, etc.
- inter-prozedural: Aufrufgraph
  - Multigraph
  - Knoten: Prozeduren
  - Kanten: Aufruf
  - ergibt sich aus expliziten Aufrufen im Programmcode sowie Aufrufe über Funktionszeiger





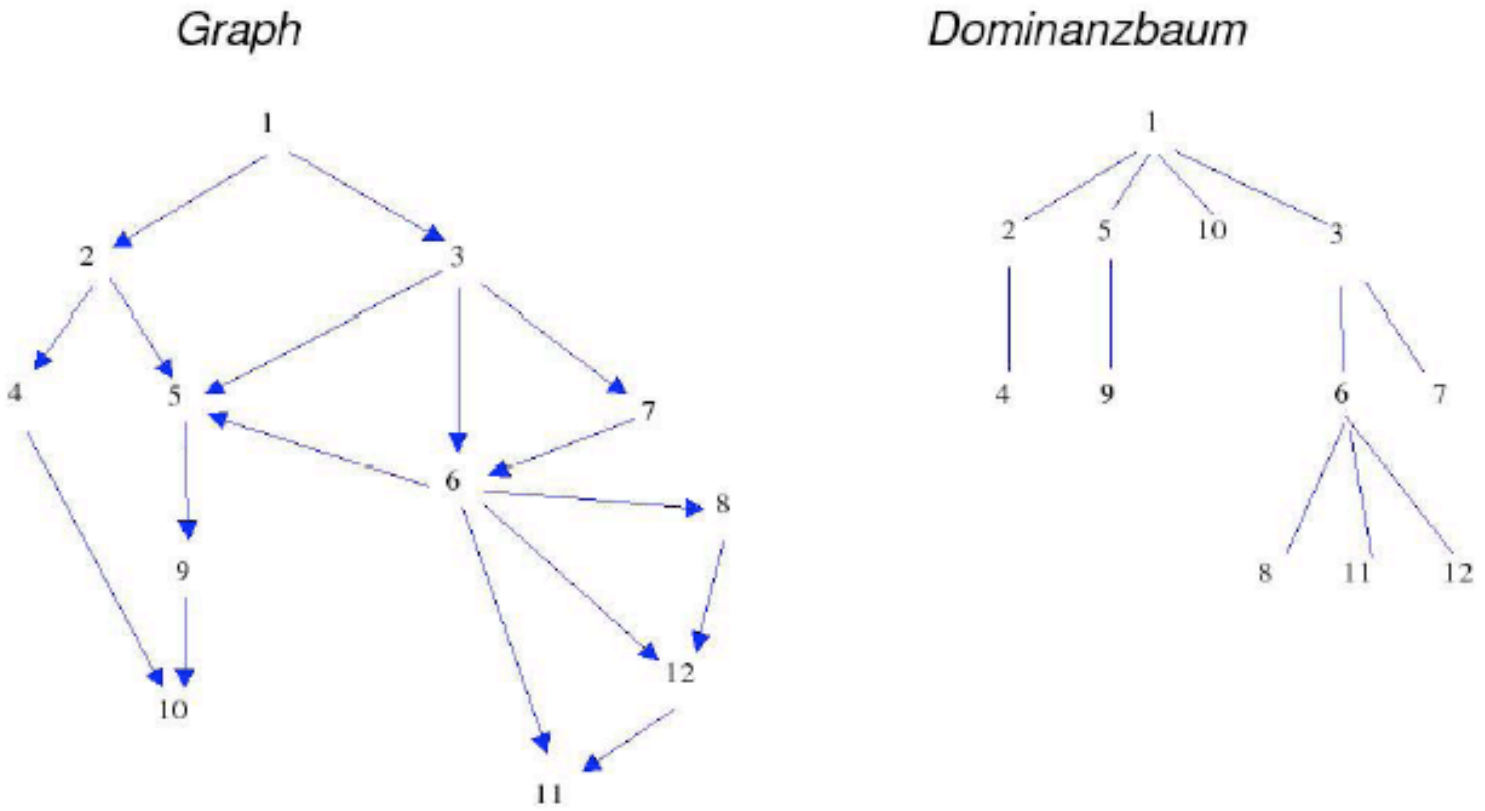
Intra- und interproz. Kontrollfluss



Dominanz

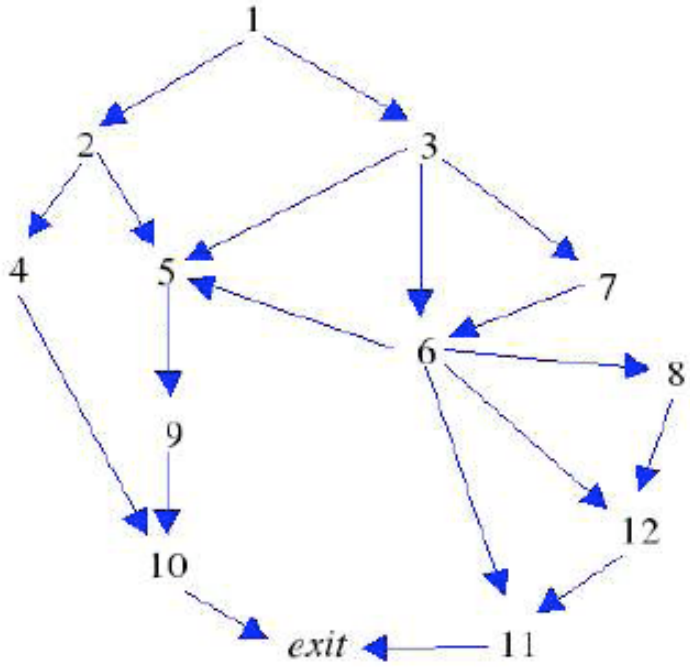
- Fragestellungen
  - Welche Prozeduren sind lokal zueinander?
    Genauer: Gibt es eine Prozedur D, über nur die ein Aufruf von N erfolgt?
  - Welcher Block D im Flussgraph muss in jedem Falle passiert werden, damit Block N ausgeführt werden kann?
- Antwort: D ist der Dominator von N
  - Ein Knoten 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

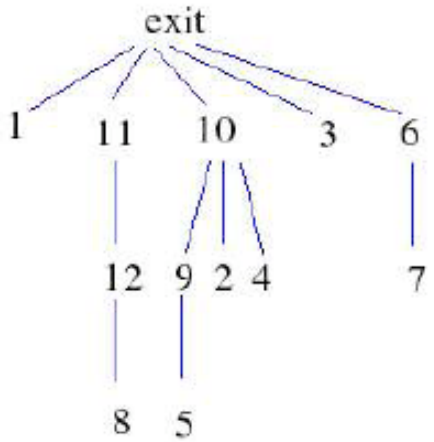
Dominanz II



Postdominanz

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





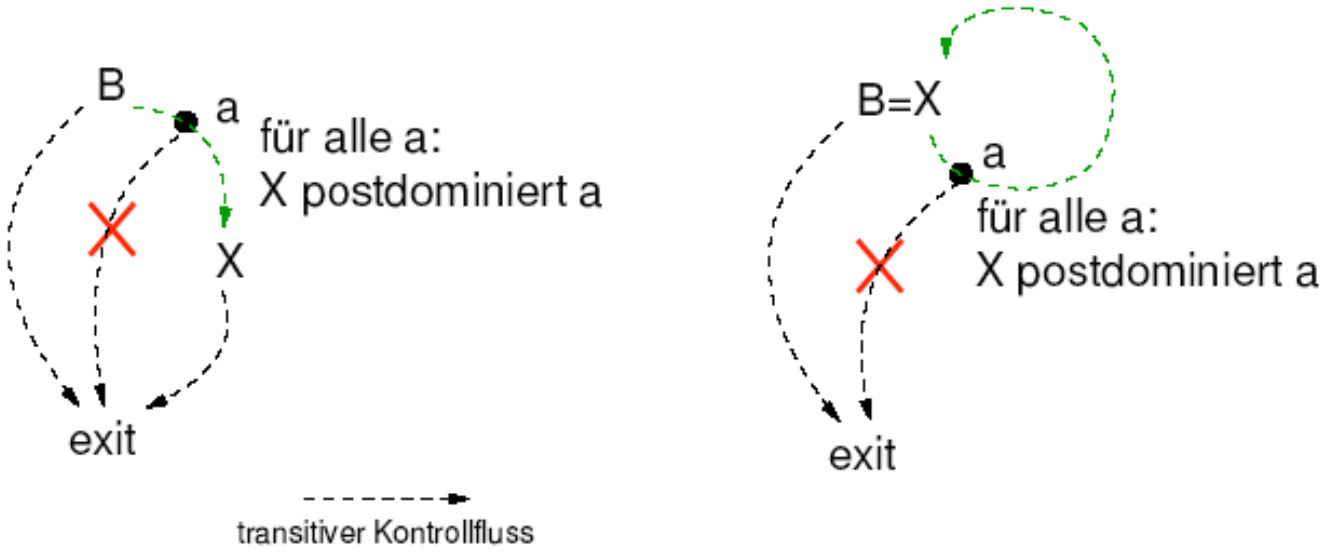
Kontrollabhängigkeit I

- Kontrollabhängigkeit = Bedingung B, von welcher die Ausführung eines anderen Knotens X abhängt.
  - B muss mehrere direkte Nachfolger haben und
  - B hat einen Pfad zum Endknoten, der X vermeidet (d.h. B kann nicht von X postdominiert werden) und
  - B hat einen Pfad zu X, d.h. insgesamt hat B mindestens 2 Pfade:
    - einer führt zu X
    - einer umgeht X

    und
  - B ist der letzte Knoten mit einer solchen Eigenschaft

Kontrollabhängigkeit II

- 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

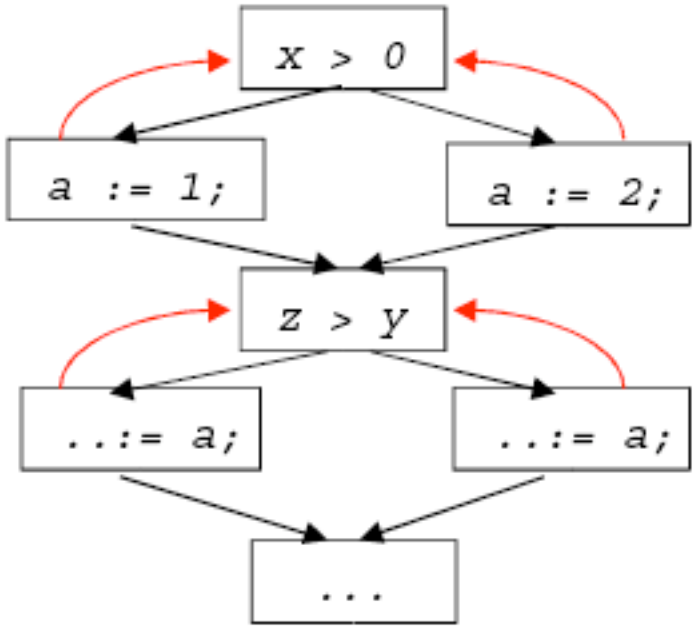
- Für strukturierte Programme
  - eine Anweisung ist kontrollabhängig von der Bedingung der nächstumgebenden Schleife oder bedingten Anweisung

```
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

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;
```

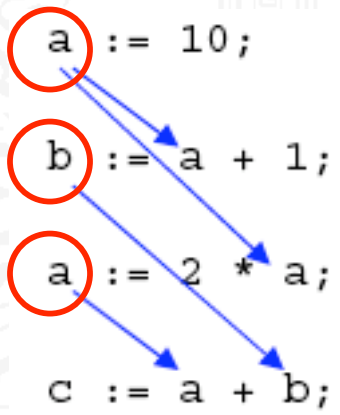


Datenabhängigkeitsanalyse

- 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)

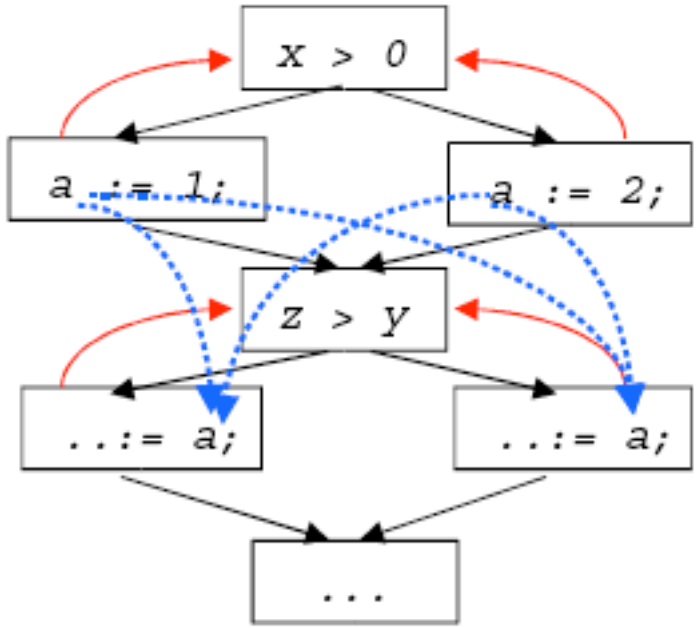
Für die Zwecke der Analyse ist die Set-Use Beziehung die wichtigste Beziehung und wird intern oft explizit repräsentiert (meist in der umgekehrten Richtung).



Zwischen der 2. und 3. Anweisung besteht eine “Use-Set”, zwischen der 1. und 3. Anweisung 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;
```



Kontroll–
abhängigkeit

Daten–
abhängigkeit

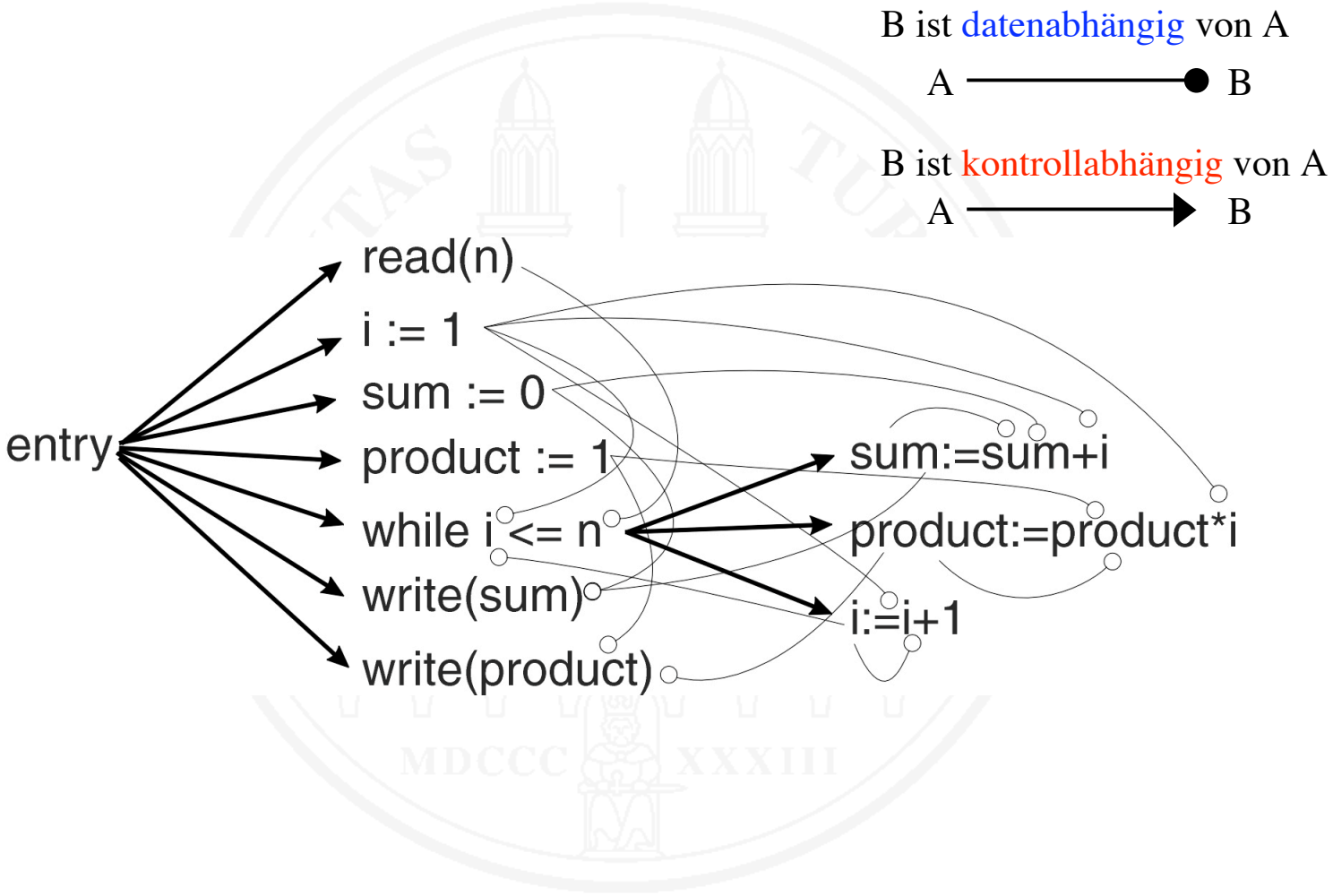
11.4 Program Dependency Graph

Multigraph für Daten- und Kontrollflussabhängigkeiten

Program Dependency Graph (PDG)

- PDG = gerichteter Multigraph für Daten- und Kontrollflussabhängigkeiten
  - Knoten repräsentieren Zuweisungen und Prädikate
    - zusätzlich einen speziellen Entry-Knoten
    - zusätzlich Ø-Knoten zur Reduktion der Datenabhängigkeitskanten
    - (sowie einen Initial-Definition-Knoten für jede Variable, die benutzt werden kann, bevor sie gesetzt wird)
  - Kontroll-Kanten repräsentieren Kontrollabhängigkeiten
    - Startknoten jeder Kontrollabhängigkeitskante ist entweder der Entry-Knoten oder ein Prädikatsknoten
  - Fluss-Kanten repräsentieren Set-Use-Abhängigkeiten

Beispiel PDG



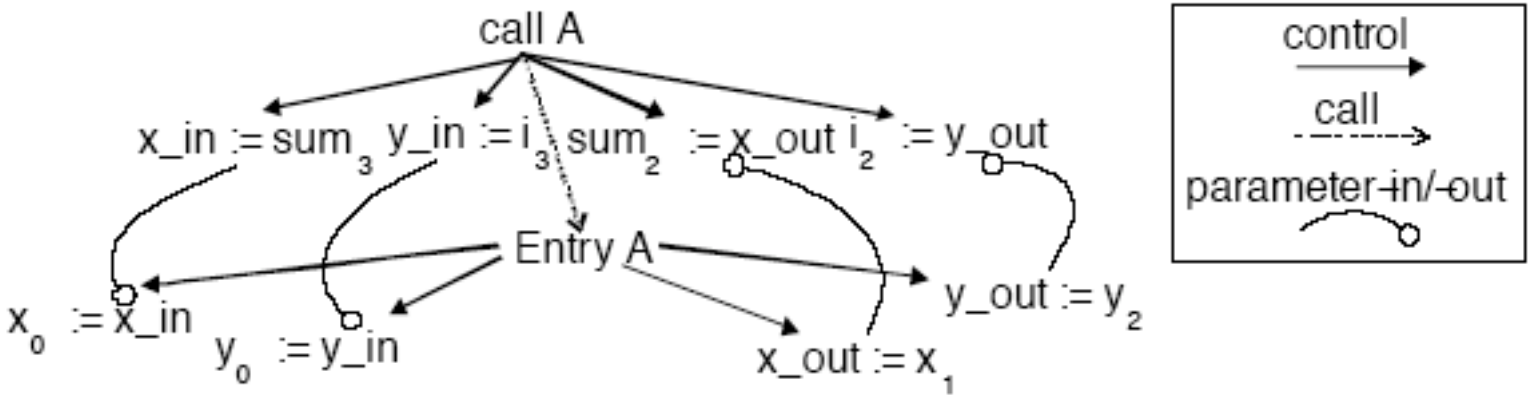
SDG und PDG

- PDG stellt Abhängigkeiten innerhalb einer Funktion dar
- System Dependency Graph (SDG) stellt globale Abhängigkeiten dar: PDGs für verschiedene Unterprogramme werden vernetzt über interprozedurale Kontroll- und Datenflusskanten
  - Aufruf: Call-Knoten
  - aktuelle Parameter: actual-in / actual-out-Knoten (copy-in/copy-out-Parameterübergabe vorausgesetzt)
  - formale Parameter: formal-in / formal-out-Knoten
  - transitive Abhängigkeiten via PDG: Summary-Edges

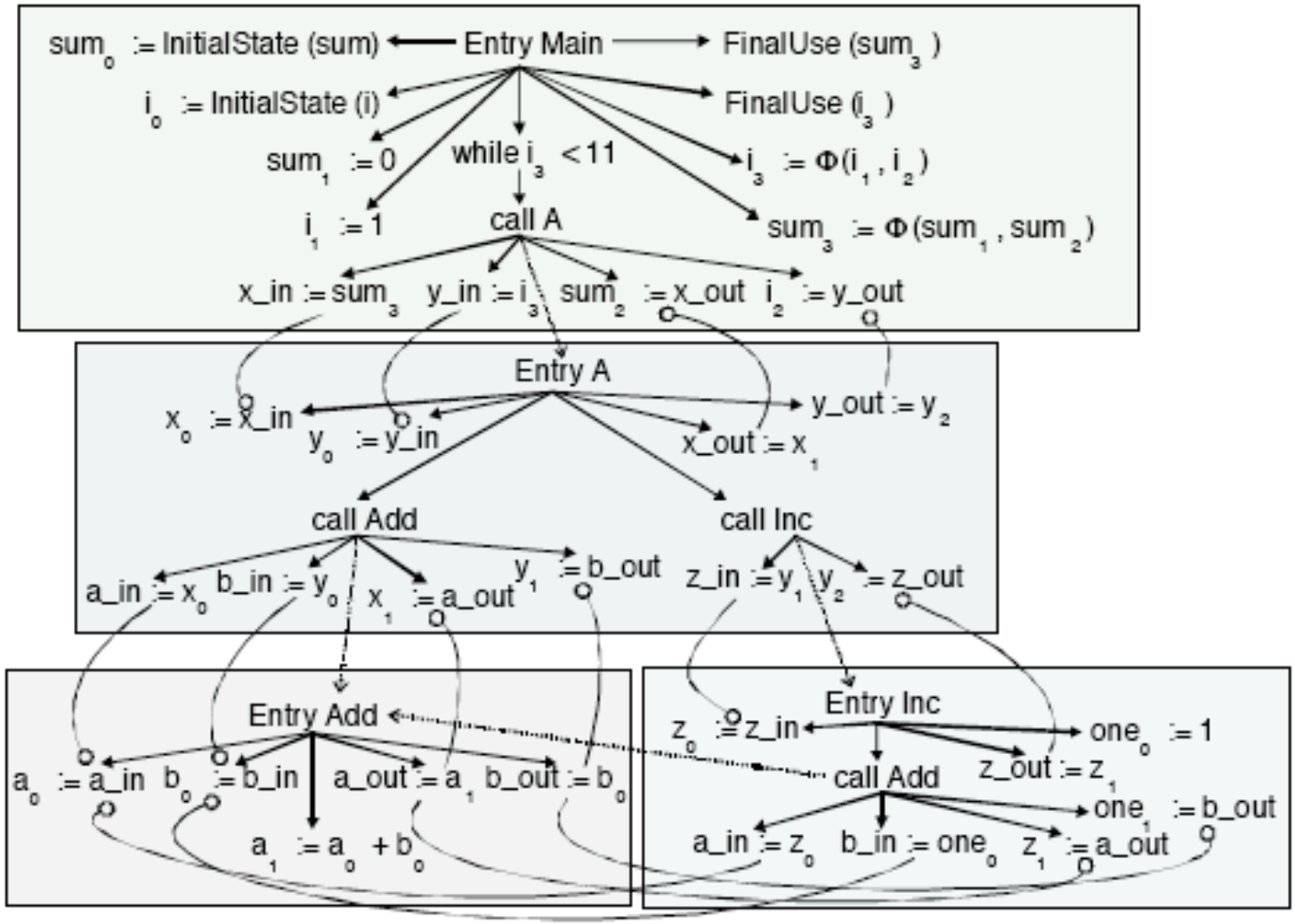
Modellierung des Prozeduraufrufs

```
x_in := sum3;
y_in := i3;
A;
sum2 := x_out;
i2 := y_out;
```

```
procedure A is
begin
  x0 := x_in; y0 := y_in;
  ...
  x_out := x1; y_out := y1;
end A;
```



System Dependence Graph



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”, Proceedings of the Working Conference on Reverse Engineering, IEEE CS, 1998
  - beschreibt Zwischendarstellungen für das Reverse Engineering und dessen spezielle Anforderungen
- K.B. Gallagher, J.R. Lyle. Using Program Slicing in Software Maintenance. IEEE Transactions on Software Engineering, 17(8):751-761, August 1991.
- S. Horwitz, T. Reps, D. Binkeley. Interprocedural Slicing using Dependence Graphs. ACM Transactions on Programming Languages and Systems, 12(1):35-46, January 1990.
- M. Weiser. Program Slicing. IEEE Transactions on Software Engineering, 10(4):352-257, 1984