Grundlagen der Programmiersprachen

Prof. Dr. Uwe Kastens

Sommersemester 2016

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 001

Ziele:

Anfang

in der Vorlesung:

Begrüßung

Ziele

Die Vorlesung soll Studierende dazu befähigen,

- die **Grundkonzepte** von Programmier- oder Anwendungssprachen zu **verstehen**,
- **Sprachdefinitionen** zu verstehen,
- **neue Programmiersprachen** und deren Anwendung **selbständig erlernen** zu können
  (dies wird an der **Sprache C** in der Vorlesung erprobt)
- typische Eigenschaften **nicht-imperativer Programmiersprachen** zu verstehen.
- Freude am Umgang mit Sprachen haben.

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 002

Ziele:
Ziele der Vorlesung

in der Vorlesung:
Ziele und Schwerpunkte

Verständnisfragen:
Stimmen die Ziele mit Ihrer Vorstellung von der Vorlesung überein?

Inhalt

Vorlesung	Thema	Kapitel im Buch Sebesta	Mitchell
1, 2	Einführung	1	1, 4
3, 4	Definition Syntaktischer Strukturen	3	4
5	Gültigkeit von Definitionen,	4.4, 4.8	7.1
6	Lebensdauer von Variablen Laufzeitkeller	4.9	7.2, 7.3
7, 8	Datentypen	4.5, 5	6
9	Aufruf, Parameterübergabe	8	
10, 11, 12	Funktionale Programmierung: Grundbegriffe, Rekursionsparadigmen, Funktionen höherer Ordnung	14	3, 7.4
13, 14	Logische Programmierung: Grundlagen, Auswertung logischer Programme	15	15
	Zusammenfassung		

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 003

Ziele:

Übersicht über die Vorlesungsthemen

in der Vorlesung:

Ziele und Schwerpunkte

Bezüge zu anderen Vorlesungen

In GPS verwendete Kenntnisse aus

- **Grundlagen der Programmierung 1, 2:**
  Eigenschaften von Programmiersprachen im allgemeinen
- **Modellierung:**
  reguläre Ausdrücke, kontext-freie Grammatiken,
  abstrakte Definition von Wertemengen, Terme, Unifikation

Kenntnisse aus GPS werden benötigt z. B. für

- **weiterführende Veranstaltungen im Bereich Programmiersprachen und Übersetzer:**
  Verständnis für Sprachkonzepte und -konstrukte
  5. Sem: **PLaC;** Master: noch offen
- **Software-Technik:** Verständnis von Spezifikationssprachen
- **Wissensbasierte Systeme:** logische Programmierung, Prolog
- **alle Veranstaltungen, die Programmier-, Spezifikations- oder Spezialsprachen verwenden:**
  Grundverständnis für Sprachkonzepte und Sprachdefinitionen,
  z. B. VHDL in GTI/GRA; SQL in Datenbanken

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 004

Ziele:

Verzahnung mit anderen Veranstaltungen bewusst machen

in der Vorlesung:

Beispiele dazu

nachlesen:

Sammlung der Inhaltsbeschreibungen zu Vorlesungen des Informatikstudiums

GPS-Literatur

Zur Vorlesung insgesamt:

- **elektronisches Skript GPS: http://ag-kastens.upb.de/lehre/material/gps**
- R. W. Sebesta: Concepts of Programming Languages, 9th Ed., Pearson, 2010
- John C. Mitchell: Concepts in Programming Languages, Cambridge University Press, 2003

Zu Funktionaler Programmierung:

- L. C. Paulson: ML for the Working Programmer, 2nd ed., Cambridge University Press, 1996

Zu Logischer Programmierung:

- W.F. Clocksin and C.S. Mellish: Programming in Prolog , 5th ed. Springer, 2003

C, C++, Java:

- Carsten Vogt: C für Java-Programmierer, Hanser, 2007
- S.P. Harbison, G.L. Steele: C: A - Reference Manual (5th ed.), Prentice Hall, 2002
- Timothy Budd: C++ for Java Programmers, Pearson, 1999.
- K. Arnold, J. Gosling: The Java Programming Language, 4th Edition, Addison-Wesley, 2005
- J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley: The Java Language Specification, Java SE 8 Edition, Oracle, 2014

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 005

Ziele:

Einige wichtige Bücher zur Vorlesung kennenlernen.

in der Vorlesung:

Hinweise zur Verwendung der Bücher:

- [Sebesta]: passt gut zur Vorlesung
- [Mitchell]: Sehr gute Übersicht, tiefer und breiter als der Stoff in GPS; auch noch für PLaC geeignet
- [Paulson]: Praxis der funktionalen Programmierung, tiefer als GPS
- [Clocksin, Mellish]: Prolog: Sprache und einfache Beispiele
- [Vogt]: leichter Einstieg nach C
- [Harbison, Steele]: Referenz für C, inkl. Standard
- [Budd]: Umstieg nach C++
- [Arnold, Gosling]: Java bescrieben vom Java-Autor (Rationale)
- [Bracha, et.al.]: Java Referenz, tief gehend

Organisation: Das GPS-Skript im WWW

http://ag-kastens.upb.de/lehre/material/gps

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 006

Ziele:

in der Vorlesung:

Erläuterungen des Vorlesungsmaterials im Web

Übungsaufgaben:

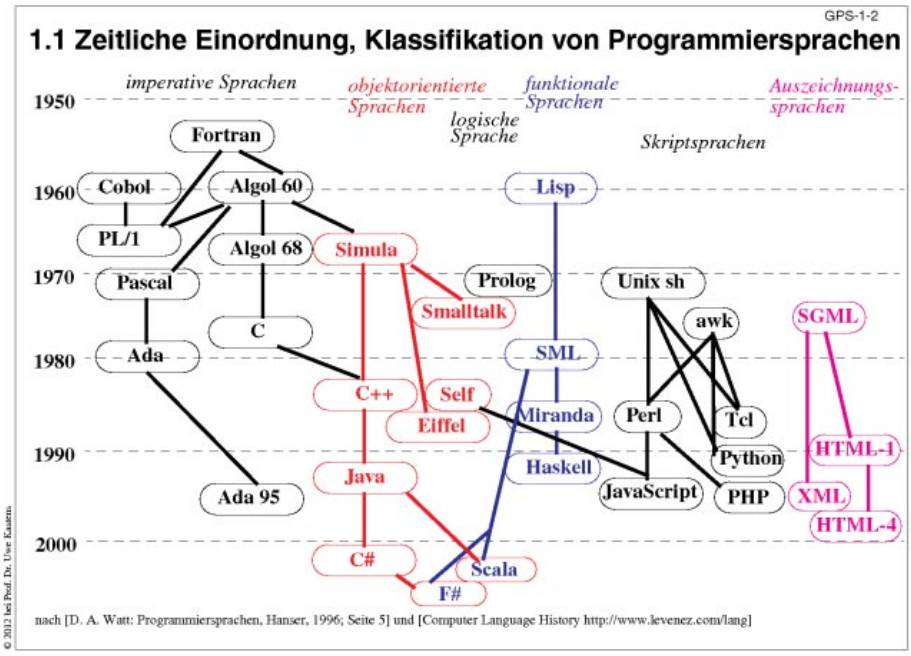
Alle wichtigen Infos gibts auf diesen Seiten. Öfter mal reinschauen!

Verständnisfragen:

- Schon gesehen?

Erläuterte Folien im Skript

Grundlagen der Programmiersprachen SS 2016 - Folie 102



Autor: Prof. Dr. Uwe Kastens

Ziele:
Sprachen zeitlich einordnen und klassifizieren

in der Vorlesung:
Kommentare zur zeitlichen Entwicklung.

Verwandschaft zwischen Sprachen:

1. Notation: C, C++, Java, C#, JavaScript, PHP;
2. gleiche zentrale Konzepte, wie Datentypen, Objektorientierung;
3. Teilsprache: Algol 60 ist Teilsprache von Simula, C von C++;
4. gleiches Anwendungsgebiet: z. B. Fortran und Algol 60 für numerische Berechnungen in wissenschaftlich-technischen Anwendungen

nachlesen:
Text dazu im Buch von D. A. Watt

Übungsaufgaben:

Verständnisfragen:
In welcher Weise können Programmiersprachen miteinander verwandt sein?

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 007

Ziele:

in der Vorlesung:

Erläuterungen des Vorlesungsmaterials im Web

Übungsaufgaben:

Alle wichtigen Infos gibts auf diesen Seiten. Öfter mal reinschauen!

Verständnisfragen:

- Schon gesehen?

Organisation im Sommersemester 2016

Termine	Vorlesung	Die 14:15 - 15:45	L1, Uwe Kastens
		Mi 14:15 - 15:45	L1, Uwe Kastens
		Beginn: Mi. 01.06.	
	Zentralübung	Mi 13:15 - 14:00	L1, Uwe Kastens
		Beginn: Mi 15. 6.	
	Übungen	**Beginn: Mo 06.06.**	
Übungsbetreuer	Dr. Peter Pfahler		
	Clemens Boos	Felix Barczewicz	Marius Meyer
	Aaron Nickl	Patrick Steffens	Jonas Klauke
Übungstermine	siehe Organisationsseite des Vorlesungsmaterials im Web gemäß Anmeldung in PAUL		
Hausaufgaben	erscheinen wöchentlich (bis Die.), Bearbeitung in Gruppenarbeit (2-4), Abgabe bis Die 14:15 Uhr; Lösungen werden korrigiert und bewertet.		
1 Test	wird während einer Zentralübung durchgeführt (Termine im Web), können bestandene Klausur um 1 - 2 Notenschritte verbessern.		
Klausur	**voraussichtliche Termine: 26.07. und 23.09** **Anmeldung in PAUL / ZPS**		

Vorlesung Grundlagen der Programmiersprachen SS 2016 / Folie 008

Ziele:

Organisation im aktuellen Semester

in der Vorlesung:

Termine, Betreuer, Übungen, Tests, Klausuren

Verständnisfragen:

- Gibt’s noch Fragen zur Organisation?
- Wo bekommt man die Antworten?

1. Einführung

Themen dieses Kapitels:

1.1. Zeitliche Einordnung, Klassifikation von Programmiersprachen

1.2. Implementierung von Programmiersprachen

1.3. Dokumente zu Programmiersprachen

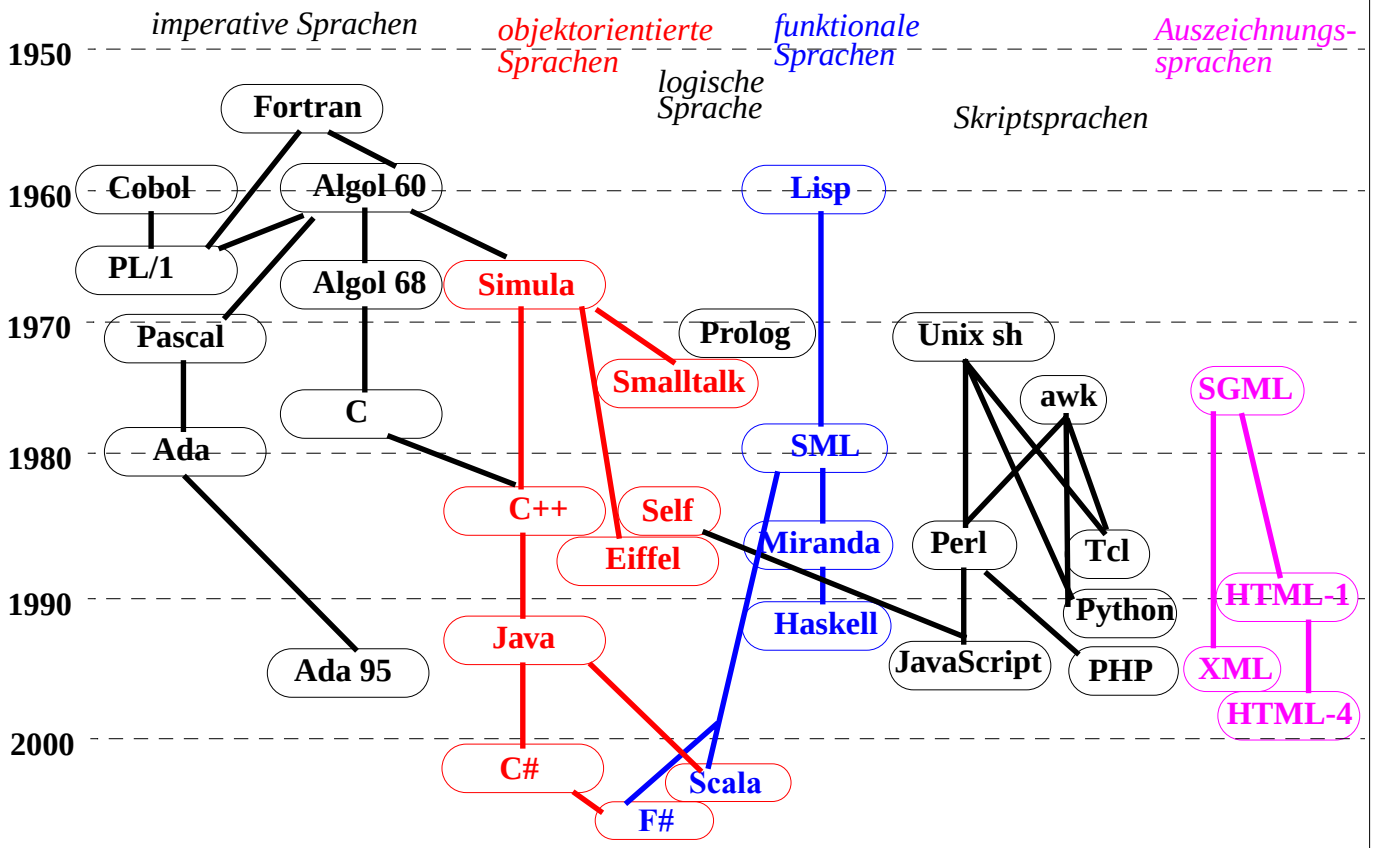
1.4. Vier Ebenen der Spracheigenschaften

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 101

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

1.1 Zeitliche Einordnung, Klassifikation von Programmiersprachen



nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5] und [Computer Language History http://www.levenez.com/lang]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 102

Ziele:

Sprachen zeitlich einordnen und klassifizieren

in der Vorlesung:

Kommentare zur zeitlichen Entwicklung.

Verwandschaft zwischen Sprachen:

- Notation: C, C++, Java, C#, JavaScript, PHP;
- gleiche zentrale Konzepte, wie Datentypen, Objektorientierung;
- Teilsprache: Algol 60 ist Teilsprache von Simula, C von C++;
- gleiches Anwendungsgebiet: z. B. Fortran und Algol 60 für numerische Berechnungen in wissenschaftlich-technischen Anwendungen

nachlesen:

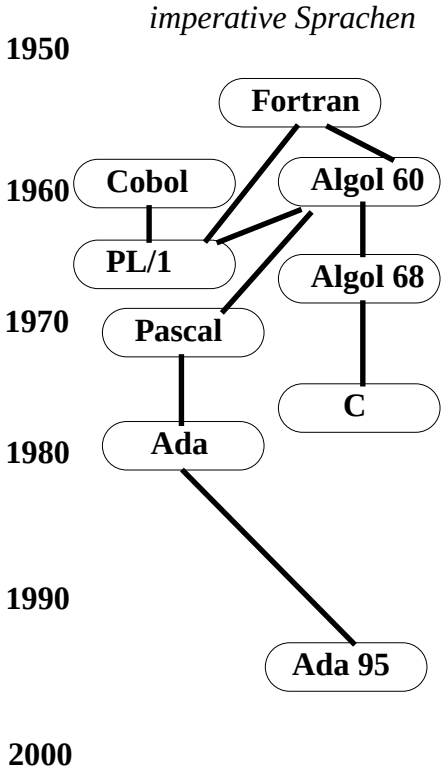
Text dazu im Buch von D. A. Watt

Übungsaufgaben:

Verständnisfragen:

In welcher Weise können Programmiersprachen miteinander verwandt sein?

Klassifikation: Imperative Programmiersprachen



charakteristische Eigenschaften:

Variable mit Zuweisungen,
veränderbarer Programmzustand,

Ablaufstrukturen (Schleifen, bedingte Anweisungen, Anweisungsfolgen)

Funktionen, Prozeduren

implementiert durch Übersetzer

nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103a

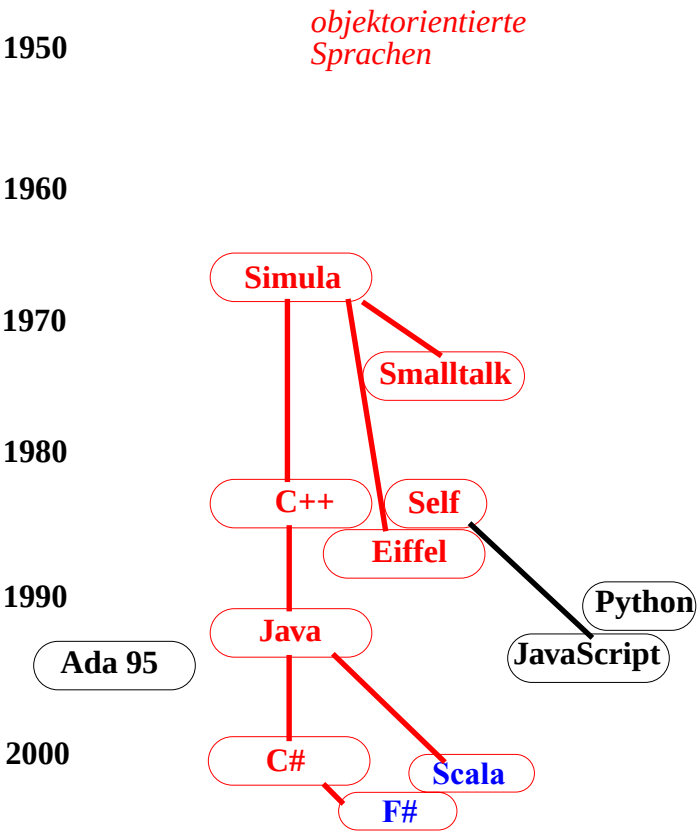
Ziele:

Charakteristika imperativer Sprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Klassifikation: objektorientierte Programmiersprachen



nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

charakteristische Eigenschaften:

Klassen mit Methoden und Attributen, Objekte zu Klassen

Vererbungsrelation zwischen Klassen

Typen:
objektorientierte Polymorphie:
Objekt einer Unterklasse kann verwendet werden, wo ein Objekt der Oberklasse benötigt wird

dynamische Methodenbindung

Self und JavaScript haben keine Klassen; Vererbung zwischen Objekten

Fast alle oo Sprachen haben auch Eigenschaften imperativer Sprachen

implementiert:
Übersetzer: Simula, C++, Eiffel, Ada
Übersetzer + VM: Smalltalk, Java, C#
Interpretierer: Self, Python, JavaScript

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103b

Ziele:

Charakteristika objektorientierter Sprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Klassifikation: logische Programmiersprachen

1950

logische Sprache

1960

1970 **Prolog**

1980

1990

2000

charakteristische Eigenschaften:

Prädikatenlogik als Grundlage

Deklarative Programme ohne Ablaufstrukturen, bestehen aus Regeln, Fakten und Anfragen

Variable ohne Zuweisungen, erhalten Werte durch Termersetzung und Unifikation

keine Zustandsänderungen
keine Seiteneffekte

keine Typen

implementiert durch Interpretierer

nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103c

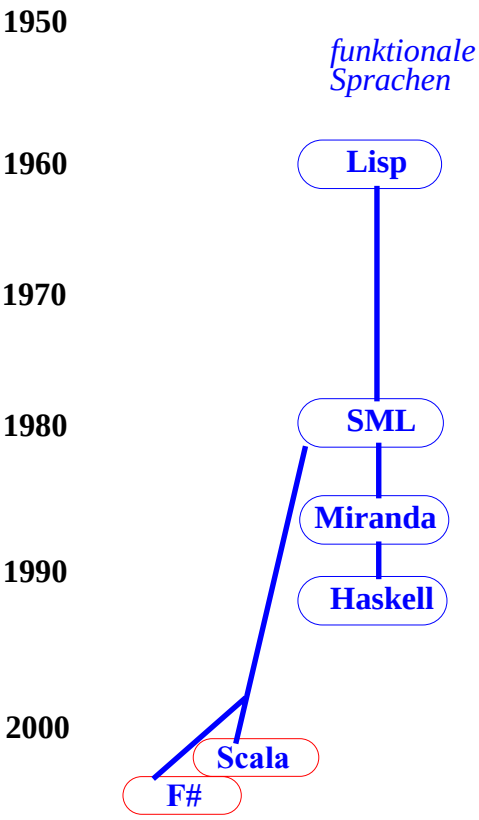
Ziele:

Charakteristika logischer Sprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Klassifikation: funktionale Programmiersprachen



nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

charakteristische Eigenschaften:

rekursive Funktionen,
Funktionen höherer Ordnung
d.h. Funktionen als Parameter oder als Ergebnis

Deklarative Programme ohne Ablaufstrukturen;
Funktionen und bedingte Ausdrücke

Variable ohne Zuweisungen,
erhalten Werte durch Deklaration oder Parameterübergabe

keine Zustandsänderung,
keine Seiten-Effekte

Typen:
Lisp: keine
SML, Haskell: parametrische Polymorphie

implementiert durch
Lisp: Interpretierer
sonst: Übersetzer und/oder Interpretierer

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103d

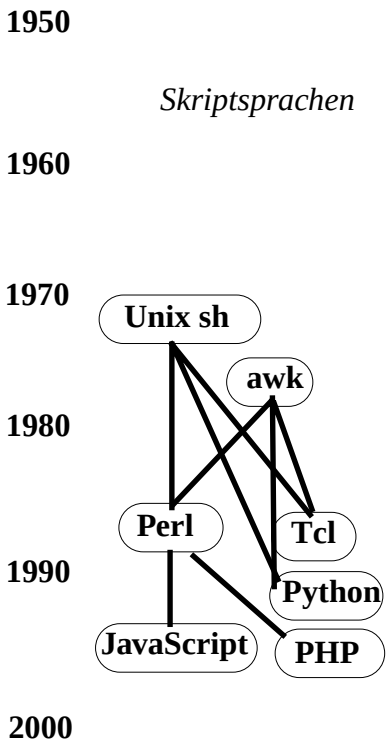
Ziele:

Charakteristika funktionaler Sprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Klassifikation: Skriptsprachen



nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

charakteristische Eigenschaften:

Ziel: einfache Entwicklung einfacher Anwendungen (im Gegensatz zu allgemeiner Software-Entwicklung),
insbes. Textverarbeitung und Web-Anwendungen

Ablaufstrukturen, Variable und Zuweisungen wie in imperativen Sprachen

Python, JavaScript und spätes PHP auch oo

Typen:
dynamisch typisiert, d.h. Typen werden bei Programmausführung bestimmt und geprüft

implementiert durch Interpretierer
ggf integriert in Browser und/oder Web-Server

ggf Programme eingebettet in HTML-Texte

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103e

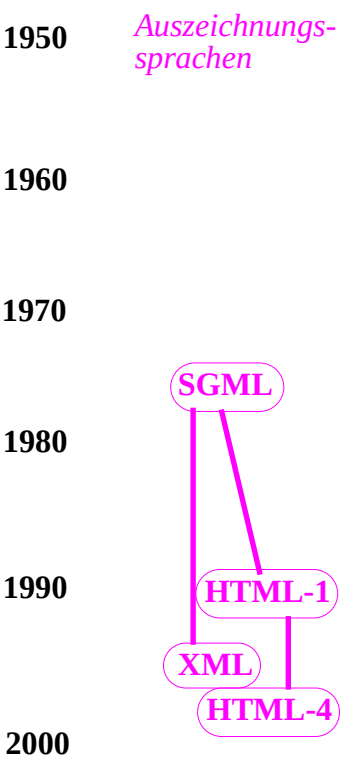
Ziele:

Charakteristika von Skriptsprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Klassifikation: Auszeichnungssprachen



charakteristische Eigenschaften:

Annotierung von Texten zur Kennzeichnung der Struktur, Formatierung, Verknüpfung

Ziele: Repräsentation strukturierter Daten (XML), Darstellung von Texten, Hyper-Texten, Web-Seiten (HTML)

Sprachkonstrukte:
Baum-strukturierte Texte, Klammerung durch *Tags,* Attribute zu Textelementen

keine Ablaufstrukturen, Variable, Datentypen zur Programmierung

Ggf. werden Programmstücke in Skriptsprachen als spezielle Textelemente eingebettet und beim Verarbeiten des annotierten Textes ausgeführt.

nach [D. A. Watt: Programmiersprachen, Hanser, 1996; Seite 5]
[Computer Language History http://www.levenez.com/lang]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 103f

Ziele:

Charakteristika von Auszeichnunugssprachen kennenlernen

Verständnisfragen:

Ordnen Sie die Beispiele von Folie 104 ein.

Eine Funktion in verschiedenen Sprachen

Sprache A:

```
function Length (list: IntList): integer;
   var len: integer;
begin
   len := 0;
   while list <> nil do
   begin len := len + 1; list := list^.next end;
   Length := len
end;
```

Sprache B:

```
int Length (Node list)
{  int len = 0;
   while (list != null)
   { len += 1; list = list.link; }
   return len;
}
```

Sprache C:

```
fun Length list =
    if null list then 0
                 else 1 + Length (tl list);
```

Sprache D:

```
length([], 0).
length([Head | Tail], Len):-
       length(Tail, L), Len IS L + 1.
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 104

Ziele:

Vergleich von Spracheigenschaften

in der Vorlesung:

- Identifikation der Sprachen (?);
- unterschiedliche Notation für gleiche Konstrukte, z. B. while-Schleife in A und B;
- unterschiedliche Konstrukte für gleiche Wirkung, z. B. Funktionsergebnis bestimmen: A Zuweisung, B return-Anweisung, C Ergebnis eines bedingten Ausdruckes;
- gleiche abstrakte Struktur, z. B. A und B;
- unterschiedliche Typisierung, z. B. A, B statisch explizit, C statisch implizit, D typlos;
- unterschiedliche Klassifikation (?).

(Auflösung der (?) in der Vorlesung)

nachlesen:

Übungsaufgaben:

Verständnisfragen:

- Diskutieren Sie die Beispiele nach obigen Kriterien.
- Aus welchen Sprachen stammen die Beispiele?

Hello World in vielen Sprachen

COBOL

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000300 DATE-WRITTEN.   02/05/96        21:04.
000400*       AUTHOR    BRIAN COLLINS
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500     DISPLAY "HELLO, WORLD." LINE 15 POSITION 10.
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

FORTRAN IV

```
   PROGRAM HELLO
   DO 10, I=1,10
   PRINT *,'Hello World'
10 CONTINUE
   STOP
   END
```

Pascal

```
Program Hello (Input, Output);
Begin
  repeat
        writeln('Hello World!')
  until 1=2;
End.
```

C

```
main()
{ for(;;)
      { printf ("Hello World!\n");
      }
}
```

Java

```
class HelloWorld {
  public static void main (String args[]) {
    for (;;) {
       System.out.print("HelloWorld");
       }
    }
}
```

Perl

```
print "Hello, World!\n" while (1);
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 105a

Ziele:

Eindruck von unterschiedlichen Sprachen

in der Vorlesung:

Identität der Sprachen und Hinweise auf einige Eigenschaften

Hello World in vielen Sprachen

Prolog

```
hello :-
printstring("HELLO WORLD!!!!").
printstring([]).
printstring([H|T]) :- put(H), printstring(T).
```

Lisp

```
(DEFUN HELLO-WORLD ()
   (PRINT (LIST ,HELLO ,WORLD)))
```

SQL

```
CREATE TABLE HELLO (HELLO CHAR(12))
UPDATE HELLO
SET HELLO = 'HELLO WORLD!'
SELECT * FROM HELLO
```

HTML

```
<HTML>
<HEAD>
<TITLE>Hello, World Page!</TITLE>
</HEAD>
<BODY>
Hello, World!
</BODY>
</HTML>
```

Make

```
default:
        echo "Hello, World\!"
        make
```

Bourne Shell (Unix)

```
while (/bin/true)
do
    echo "Hello, World!"
done
```

LaTex

```
\documentclass{article}
\begin{document}
\begin{center}
\Huge{HELLO WORLD}
\end{center}
\end{document}
```

PostScript

```
/Font /Helvetica-Bold findfont def
/FontSize 12 def
Font FontSize scalefont setfont
{newpath 0 0 moveto (Hello, World!) show showpage} loop
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 105b

Ziele:

Eindruck von unterschiedlichen Sprachen

in der Vorlesung:

Identität der Sprachen und Hinweise auf einige Eigenschaften

Sprachen für spezielle Anwendungen

• **technisch/wissenschaftlich:**		FORTRAN, Algol-60
• **kaufmännisch**		RPG, COBOL
• **Datenbanken:**		SQL
• **Vektor-, Matrixrechnungen:**		APL, Lotus-1-2-3
• **Textsatz:**		TeX, LaTeX, PostScript
• **Textverarbeitung, Pattern Matching:**		SNOBOL, ICON, awk, Perl
• **Skriptsprachen:**		DOS-, UNIX-Shell, TCL, Perl, PHP
• **Auszeichnung (Markup):**		HTML, XML
• **Spezifikationssprachen:**	SETL, Z	Allgemeine Spezifikationen von Systemen
	VHDL	Spezifikationen von Hardware
	UML	Spezifikationen von Software
	EBNF	Spezifikation von KFGn, Parsern

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 106

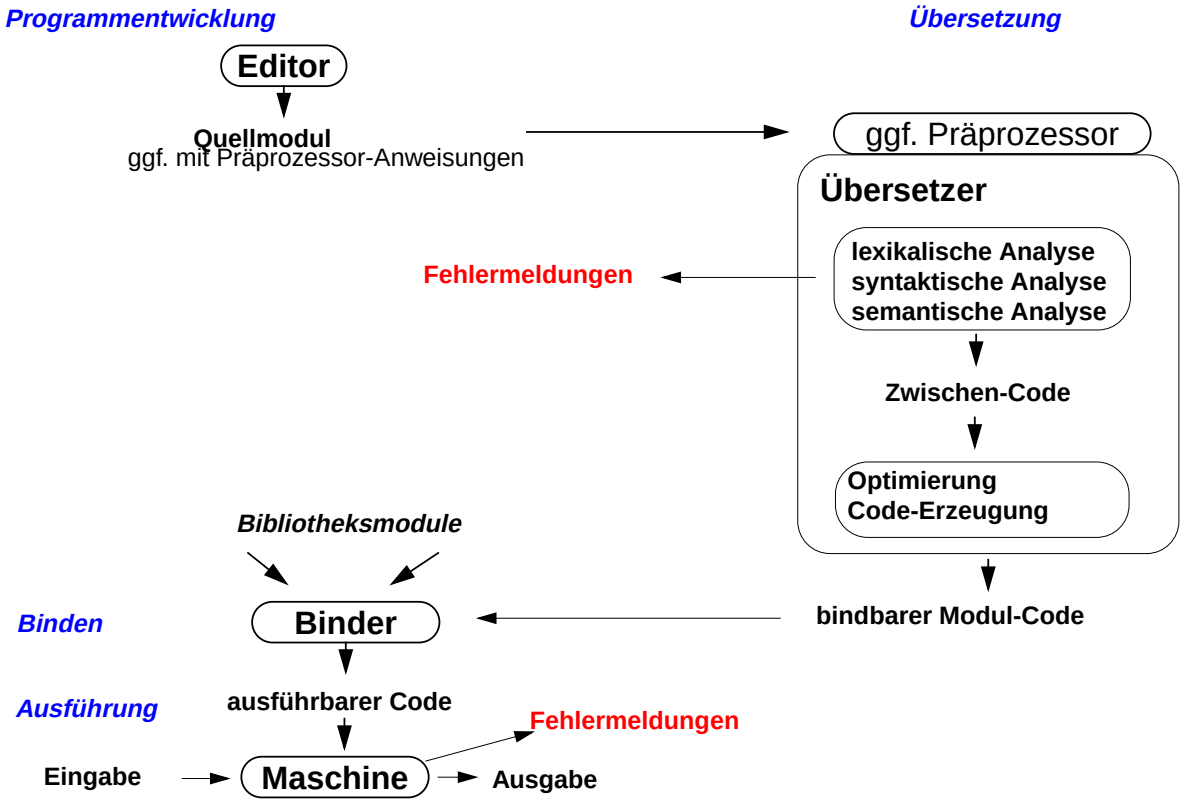
Ziele:
Es gibt nicht nur Programmiersprachen!

in der Vorlesung:
Erläuterungen dazu; Beispiele zeigen

nachlesen:
Interessante Web Site: The Language List

Verständnisfragen:
Geben Sie mindestens 3 weitere Spezialsprachen an.

1.2 Implementierung von Programmiersprachen Übersetzung



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 108

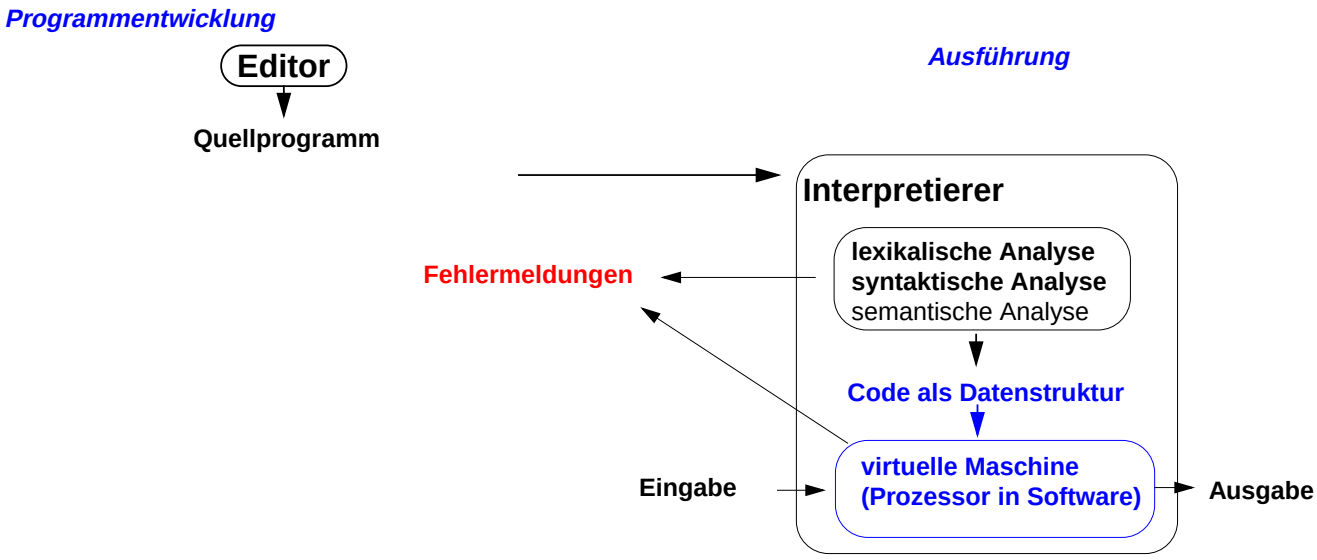
Ziele:

Übersetzer und seine Umgebung kennenlernen

in der Vorlesung:

- Erläuterung der Komponenten;
- getrennte Übersetzung von Moduln, Klassen, Modul-Code binden;
- integrierte Entwicklungsumgebungen (IDE);
- Präprozessoren auf Folie GPS 1.10

Interpretation



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 108a

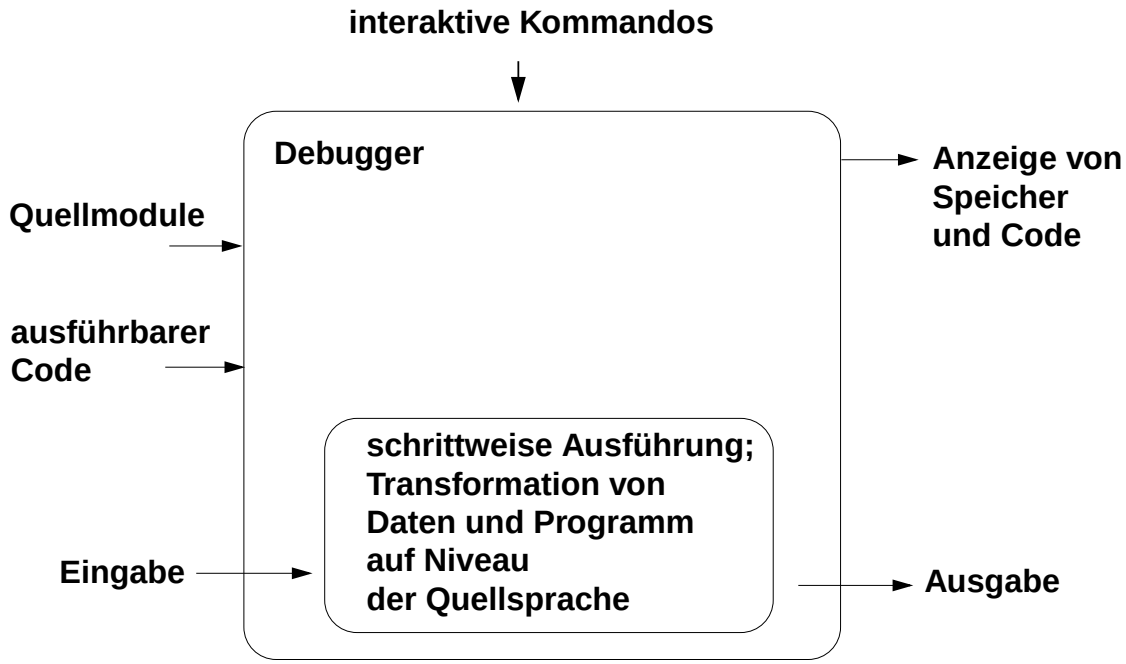
Ziele:

Interpretation statt Übersetzung

in der Vorlesung:

- Erläuterung des Interpretierers
- Konsequenzen für Sprachen und Benutzung

Testhilfe: Debugger



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 108b

Ziele:

Prinzip von Debuggern kennenlernen

in der Vorlesung:

- Erläuterung der Funktionsweise

Präprozessor CPP

Präprozessor:

- bearbeitet Programmtexte, bevor sie vom Übersetzer verarbeitet werden
- Kommandos zur Text-Substitution - ohne Rücksicht auf Programmstrukturen
- sprachunabhängig
- cpp gehört zu Implementierungen von C und C++, kann auch unabhängig benutzt werden

```
#include <stdio.h>                          Datei an dieser Stelle einfügen
#include "induce.h"

#define MAXATTRS  256                       benannte Konstante

#define ODD(x)    ((x)%2 == 1)              parametrisiertes Text-Makro
#define EVEN(x)   ((x)%2 == 0)

static void early (int sid)
   {  int attrs[MAXATTRS];                  Konstante wird substituiert
      ...
      if (ODD (currpartno)) currpartno--;   Makro wird substituiert
#ifndef GORTO                               bedingter Textblock
      printf ("early for %d currpartno: %d\n",
              sid, currpartno);
#endif
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 109

Ziele:

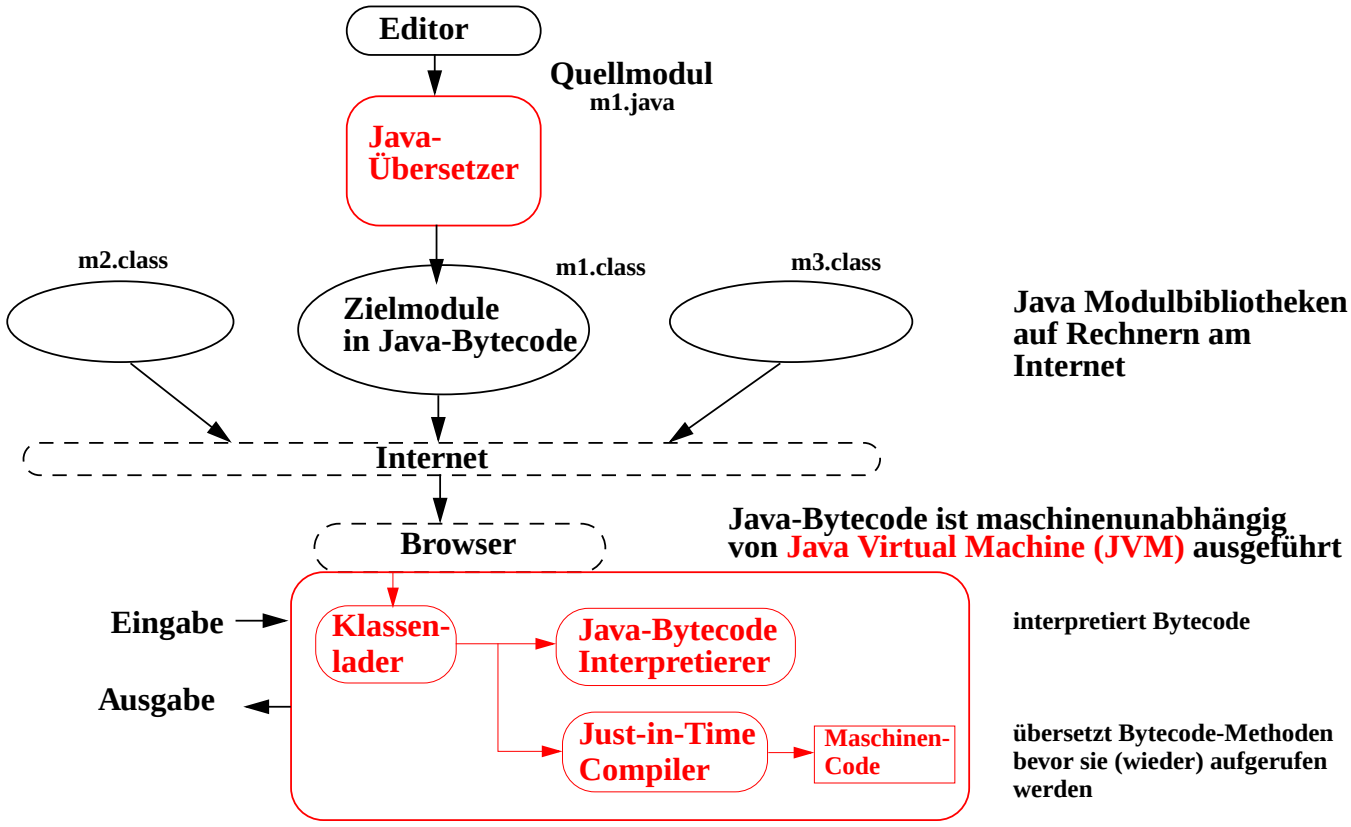
Präprozessoren kennenlernen

in der Vorlesung:

Erläuterungen

- der Eigenschaften,
- der Kommandos,
- von Einsatzmöglichkeiten.

Ausführung von Java-Programmen



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 110

Ziele:

Maschinenunabhängigkeit durch Interpretation

in der Vorlesung:

Java Bytecode, abstrakte Maschine, Rolle des Internet erläutern

Verständnisfragen:

Siehe SWE-16

1.3 Dokumente zu Programmiersprachen

Reference Manual:

verbindliche Sprachdefinition, beschreibt alle Konstrukte und Eigenschaften vollständig und präzise

Standard Dokument:

Reference Manual, erstellt von einer anerkannten Institution, z.B. ANSI, ISO, DIN, BSI

formale Definition:

für Implementierer und Sprachforscher,
verwendet formale Kalküle, z.B. KFG, AG, vWG, VDL, denotationale Semantik

Benutzerhandbuch (Rationale):

Erläuterung typischer Anwendungen der Sprachkonstrukte

Lehrbuch:

didaktische Einführung in den Gebrauch der Sprache

Implementierungsbeschreibung:

Besonderheiten der Implementierung, Abweichungen vom Standard, Grenzen, Sprachwerkzeuge

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 111

Ziele:

Unterschiedliche Zwecke von Sprachdokumenten erkennen

in der Vorlesung:

- Auszüge aus Dokumenten
- typische Formulierungen in Reference Manuals

nachlesen:

..., Abschnitt

nachlesen:

nur ansehen:

- Java Reference Manual,
- Java Benutzerhandbuch,
- ein Java Lehrbuch

Übungsaufgaben:

- Aussagen zu einer Spracheigenschaft in o.g. Dokumenten vergleichen

Verständnisfragen:

Beispiel für ein Standard-Dokument

6.1 Labeled statement [stmt.label]

A statement can be labeled.

```
labeled-statement:
            identifier : statement
            case constant-expression : statement
            default : statement
```

An identifier label declares the identifier. The only use of an identifier label is as the target of a goto. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a goto statement before its definition. Labels have their own name space and do not interfere with other identifiers.

[**Aus einem C++-Normentwurf, 1996]**

Begriffe zu Gültigkeitsregeln, statische Semantik (siehe Kapitel 3).

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 112

Ziele:

Eindruck eines Standard-Dokumentes vermitteln.

in der Vorlesung:

Erläuterungen dazu

Verständnisfragen:

Wie würden Sie diesen Absatz ins Deutsche übersetzen?

Beispiel für eine formale Sprachdefinition

```
Prologprogramm  ::= ( Klausel | Direktive )+ .

Klausel         ::= Fakt | Regel .

Fakt            ::= Atom | Struktur .

Regel           ::= Kopf ":-" Rumpf "." .

Direktive       ::= ":-" Rumpf
                |   "?-" Rumpf
                |   "-" CompilerAnweisung
                |   "?-" CompilerAnweisung .
```

[Spezifikation einer Syntax für Prolog]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 113

Ziele:

Eindruck einer formalen Sprachdefinition vermitteln.

in der Vorlesung:

Erläuterungen dazu

Verständnisfragen:

Für welche der Symbole in der Prolog-Grammatik fehlen die Regeln?

Beispiel für ein Benutzerhandbuch

R.5. Ausdrücke

Die Auswertungsreihenfolge von Unterausdrücken wird von den Präzedenz-Regeln und der Gruppierung bestimmt. Die üblichen mathematischen Regeln bezüglich der Assoziativität und Kommutativität können nur vorausgesetzt werden, wenn die Operatoren tatsächlich assoziativ und kommutativ sind. Wenn nicht anders angegeben, ist die Reihenfolge der Auswertung der Operanden undefiniert. Insbesondere ist das Ergebnis eines Ausdruckes undefiniert, wenn eine Variable in einem Ausdruck mehrfach verändert wird und für die beteiligten Operatoren keine Auswertungsreihenfolge garantiert wird.

Beispiel:

```
i = v[i++];          // der Wert von i ist undefiniert
i = 7, i++, i++;     // i hat nach der Anweisung den Wert 9
```

[Aus dem C++-Referenz-Handbuch, Stroustrup, 1992]

Eigenschaften der dynamischen Semantik

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 114

Ziele:

Eindruck eines Benutzerhandbuches

in der Vorlesung:

Kurze Erläuterung des Reihenfolgeproblems der Ausdrucksauswertung

Verständnisfragen:

Was bedeuten die Operatoren "++" und ","?

Beispiel für ein Lehrbuch

Chapter 1, The Message Box

This is a very simple script. It opens up an alert message box which displays whatever is typed in the form box above. Type something in the box. Then click „Show Me“

HOW IT‘S DONE

Here‘s the entire page, minus my comments. Take a few minutes to learn as much as you can from this, then I‘ll break it down into smaller pieces.

```
 <HTML>  <HEAD>
<SCRIPT LANGUAGE=“JavaScript“>
      function MsgBox (textstring) {alert (textstring)}
</SCRIPT>
</HEAD>  <BODY>
<FORM>  <INPUT NAME=“text1“ TYPE=Text>
        <INPUT NAME=“submit“ TYPE=Button VALUE=“Show Me“
               onClick=“MsgBox(form.text1.value)“>
</FORM>
</BODY> </HTML>
```

[Aus einem JavaScript-Tutorial]

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 115

Ziele:

Eindruck einer Programmiersprach-Lehrbuches oder -Kurses vermitteln.

in der Vorlesung:

Erläuterungen dazu

Übungsaufgaben:

Verständnisfragen:

Funktioniert der abgebildetete HTML / JavaScript-Code in Ihrem Lieblingsbrowser?

1.4 Vier Ebenen der Spracheigenschaften

Die Eigenschaften von Programmiersprachen werden in 4 Ebenen eingeteilt:

Von a über b nach c werden immer größere Zusammenhänge im Programm betrachtet. In d kommt die Ausführung des Programmes hinzu.

Ebene	definierte Eigenschaften
a. Grundsymbole	**Notation**
b. Syntax (konkret und abstrakt)	**Struktur**
c. Statische Semantik	**statische Zusammenhänge**
d. Dynamische Semantik	**Wirkung, Bedeutung**

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 116

Ziele:

Einordnung von Spracheigenschaften

in der Vorlesung:

- Ebenen gegeneinander abgrenzen;
- statische und dynamische Semantik definieren die Bedeutung von Konstrukten - nicht nur ihre Korrektheit;
- Beispiele anhand der Folien GPS-1-17 bis GPS-1-19 und aus Reference Manuals

Übungsaufgaben:

- Geben Sie je 2 Verletzungen von Regeln zu **a** bis **d** in Java an.
- Schreiben Sie ein kurzes, fehlerhaftes Java-Programm, das zu **a** bis **d** je mindestens eine Fehlermeldung provoziert.

Verständnisfragen:

Können Sie sich Sprachen vorstellen, die keine statische Semantik haben? Welche Aufgaben würde ein Übersetzer für solche Sprachen erledigen?

Beispiel für die Ebene der Grundsymbole

Ebene — **definierte Eigenschaften**

a. Grundsymbole — **Notation**

typische **Klassen von Grundsymbolen**:

Bezeichner,
Literale (Zahlen, Zeichenreihen),
Wortsymbole,
Spezialsymbole

formal definiert z. B. durch **reguläre Ausdrücke**

Folge von Grundsymbolen:

```
int dupl ( int a ) { return 2 * a ; }
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 117a

Ziele:
Ebene der Grundsymbole verstehen

in der Vorlesung:
Erläuterung des Beispiels

Verständnisfragen:
Warum sollte man beim Programmieren alle Wortsymbole der Sprache kennen?

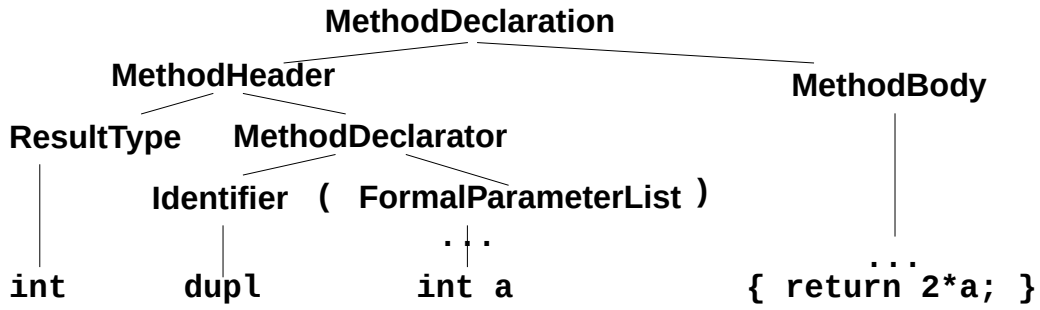
Beispiel für die Ebene der Syntax

Ebene	definierte Eigenschaften
b. Syntax (konkrete und abstrakte Syntax)	**syntaktische Struktur**

Struktur von Sprachkonstrukten

formal definiert durch **kontext-freie Grammatiken**

Ausschnitt aus einem Ableitungs- bzw. Strukturbaum:



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 117b

Ziele:

Syntaktische Ebene verstehen

in der Vorlesung:

Erläuterung des Beispiels

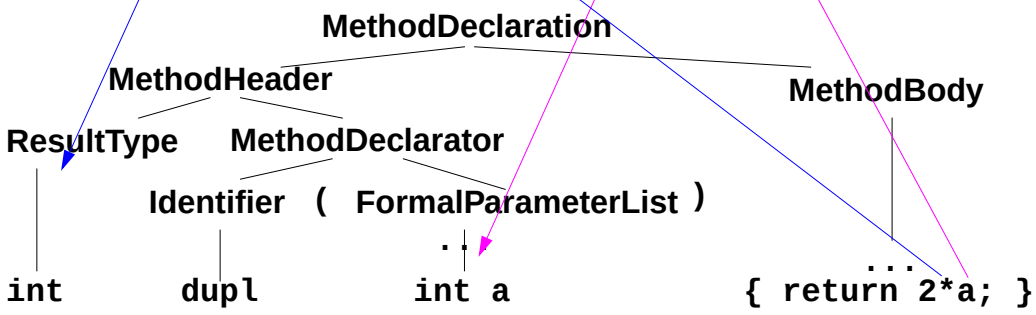
Beispiel für die Ebene der statischen Semantik

Ebene	definierte Eigenschaften
c. statische Semantik	**statische Zusammenhänge, z. B.**

meist verbal definiert;
formal definiert z. B. durch **attributierte Grammatiken**

a ist an die Definition des formalen Parameters gebunden. **Bindung von Namen**

Der **`return`**-Ausdruck hat den gleichen Typ wie der ResultType. **Typregeln**



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 117c

Ziele:

Ebene der statischen Semantik verstehen

in der Vorlesung:

Erläuterung des Beispiels

Verständnisfragen:

Kann es sein, dass das Einfügen der Beispielzeile

```
int dupl ( int a ) { return 2 * a; }
```

in ein korrektes Java-Programm zu einem nicht mehr übersetzbaren Java-Programm führt?.

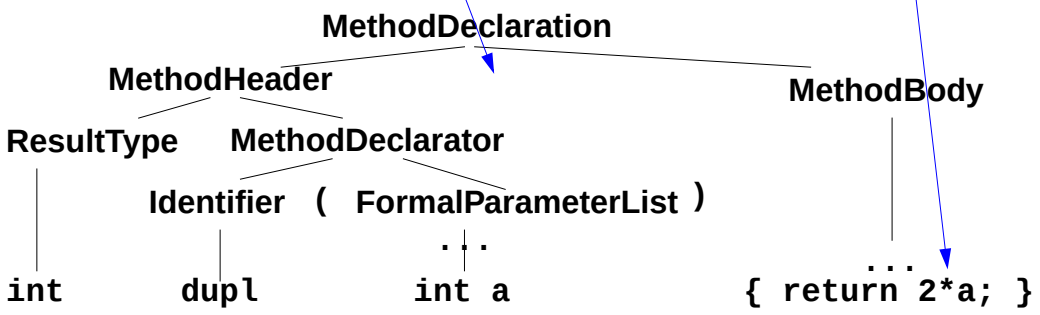
Beispiel für die Ebene der dynamischen Semantik

Ebene	definierte Eigenschaften
d. dynamische Semantik	**Bedeutung, Wirkung der Ausführung**

von Sprachkonstrukten, Ausführungsbedingungen

meist verbal definiert;
formal definiert z. B. durch **denotationale Semantik**

Ein Aufruf der Methode **`dupl`** liefert das Ergebnis
der Auswertung des **`return`**-Ausdruckes



Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 117d

Ziele:

Ebene der dynamischen Semantik verstehen

in der Vorlesung:

Erläuterung des Beispiels.

Verständnisfragen:

Beschreiben Sie einen Fehler, der erst bei der Ausführung eines Java-Programms erkannt werden kann.

Statische und dynamische Eigenschaften

Statische Eigenschaften: aus dem Programm bestimmbar, ohne es auszuführen

statische Spracheigenschaften:
Ebenen a, b, c: Notation, Syntax, statische Semantik

statische Eigenschaften eines Programmes:
Anwendung der Definitionen zu a, b, c auf das Programm

Ein Programm ist **übersetzbar**, falls es die Regeln zu (a, b, c) erfüllt.

Dynamische Eigenschaften: beziehen sich auf die Ausführung eines Programms

dynamische Spracheigenschaften:
Ebene d: dynamische Semantik

dynamische Eigenschaften eines Programmes:
Wirkung der Ausführung des Programmes mit bestimmter Eingabe

Ein Programm ist **ausführbar**, falls es die Regeln zu (a, b, c) und **(d)** erfüllt.

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 118

Ziele:

Begriffe statisch und dynamisch verstehen

in der Vorlesung:

Begriffe in Bezug zu den Ebenen erläutern

Beispiel: Dynamische Methodenbindung in Java

Für den Aufruf einer Methode kann im Allgemeinen erst **beim Ausführen** des Programms bestimmt werden, **welche Methode** aufgerufen wird.

```
class A {
   void draw (int i){...};
   void draw () {...}
}

class B extends A {
   void draw () {...}
}

class X {
   void m () {
      A a;
      if (...)
            a = new A ();
      else a = new B ();

      a.draw ();
   }
}
```

statisch wird am Programmtext bestimmt:

- der Methodenname: **`draw`**
- die Typen der aktuellen Parameter: keine
- der statische Typ von **`a`**: **`A`**
- ist eine Methode **`draw`** ohne Parameter in A oder einer Oberklasse definiert? ja
- **`draw()`** in **`B`** überschreibt **`draw()`** in **`A`**

dynamisch wird bei der Ausführung bestimmt:

- der Wert von **`a`**:
  z. B. Referenz auf ein **`B`**-Objekt
- der Typ des Wertes von a: **`B`**
- die aufzurufende Methode: **`draw`** aus **`B`**

Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 118a

Ziele:

Begriffe statisch und dynamisch verstehen

in der Vorlesung:

- Es wird erläutert, welche Aspekte der Methodenbindung in Java zur statischen und welche zur dynamischen Semantik gehören.
- Die Entscheidung zwischen überladenen Methoden wird vollständig statisch getroffen.

Fehler im Java-Programm

Fehler klassifizieren: lexikalisch, syntaktisch, statisch oder dynamisch semantisch:

```
1    class Error
2    {  private static final int x = 1..;
3       public static void main (String [] arg)
4       {  int[] a = new int[10];
5          int i
6          boolean b;
7          x = 1; y = 0; i = 10;
8          a[10] = 1;
9          b = false;
10         if (b) a[i] = 5;
11      }
12   }
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 119

Ziele:
Ebenen **a** bis **d** anhand von Fehlern erkennen

in der Vorlesung:
- Auflösung und Diskussion von nicht eindeutigen Klassifizierungen

Übungsaufgaben:
siehe Folie 116

Verständnisfragen:
- Warum kann man lexikalische und syntaktische Fehler nicht sicher unterscheiden?
- Regelt die Sprachdefinition immer eindeutig, in welchen Fällen ein Übersetzer einen Fehler melden muss?

Fehlermeldungen eines Java-Übersetzers

```
Error.java:2: <identifier> expected
          {  private static final int x = 1..;
                                          ^

Error.java:5: ';' expected
              int i
                   ^

Error.java:2: double cannot be dereferenced
          {  private static final int x = 1..;
                                         ^

Error.java:7: cannot assign a value to final variable x
              x = 1; y = 0; i = 10;
              ^

Error.java:7: cannot resolve symbol
symbol  : variable y
location: class Error
              x = 1; y = 0; i = 10;
                     ^

Error.java:9: cannot resolve symbol
symbol  : variable b
location: class Error
              b = false;
              ^

Error.java:10: cannot resolve symbol
symbol  : variable b
location: class Error
              if (b) a[i] = 5;
              ^

7 errors
```


Vorlesung Grundlagen der Programmiersprachen SS 2015 / Folie 120

Ziele:

Fehlermeldung klassifizieren

in der Vorlesung:

Erläuterungen dazu

Verständnisfragen:

- Warum gibt es zu Zeile 2 zwei Meldungen?
- Was bedeuten die Meldungen zu den Zeilen 9 und 10?

Zusammenfassung zu Kapitel 1

Mit den Vorlesungen und Übungen zu Kapitel 1 sollen Sie nun Folgendes können:

- Wichtige Programmiersprachen zeitlich einordnen
- Programmiersprachen klassifizieren
- Sprachdokumente zweckentsprechend anwenden
- Sprachbezogene Werkzeuge kennen
- Spracheigenschaften und Programmeigenschaften in die 4 Ebenen einordnen

Ziele:
Ziele des Kapitels erkennen

in der Vorlesung:
Erläuterungen dazu

2. Syntax

Themen dieses Kapitels:

- 2.1 Grundsymbole
- 2.2 Kontext-freie Grammatiken
- Schema für Ausdrucksgrammatiken
- Erweiterte Notationen für kontext-freie Grammatiken
- Entwurf einfacher Grammatiken
- abstrakte Syntax
- 2.3 XML

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 201

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

2.1 Grundsymbole

Grundsymbole:

Programme bestehen aus einer **Folge von Grundsymbolen**. (Ebene (a) auf GPS-1-16)
Jedes Grundsymbol ist eine **Folge von Zeichen**.
Ihre Schreibweise wird z.B. durch **reguläre Ausdrücke** festgelegt.
Grundsymbole sind die **Terminalsymbole der konkreten Syntax**. (Ebene (b) GPS-1-16)

Folgende 4 **Symbolklassen** sind typisch für Grundsymbole von Programmiersprachen:
Bezeichner, Wortsymbole, Literale, Spezialsymbole

1. **Bezeichner (engl. identifier):**

   zur Angabe von Namen, z. B. **`maximum findValue res_val _MIN2`**

   Definition einer Schreibweise durch reg. Ausdruck: *Buchstabe (Buchstabe | Ziffer)**

2. **Wortsymbole (engl. keywords):**

   kennzeichnen Sprachkonstrukte

   Schreibweise fest vorgegeben; meist wie Bezeichner, z. B.**`class static if for`**
   Dann müssen Bezeichner verschieden von Wortsymbolen sein.

   Nicht in PL/1; dort unterscheidet der Kontext zwischen Bezeichener und Wortsymbol:
   `IF THEN THEN THEN = ELSE ELSE ELSE = THEN;`

   Es gibt auch gekennzeichnete Wortsymbole, z.B. **`$begin`**

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 202

Ziele:

Klassen und Schreibweisen von Grundsymbolen kennenlernen

in der Vorlesung:

- Grundsymbolklassen erläutern
- Typische Schreibweisen erläutern
- Beispiele aus Java, C, Pascal, FORTRAN

Verständnisfragen:

- Wie sind Bezeichner in Java definiert?

Literale und Spezialsymbole

2. Literale (engl. literals):

Notation von Werten, z. B.

ganze Zahlen:	`7  077  0xFF`
Gleitpunktzahlen:	`3.7e-5  0.3`
Zeichen:	`'x'  '\n'`
Zeichenreihen:	`"Hallo"`

Unterscheide Literal und sein Wert: `"Sage \"Hallo\""` und `Sage "Hallo"`
verschiedene Literale - gleicher Wert: `63  077  0x3F`

Schreibweisen werden durch reguläre Ausdrücke festgelegt

4. Spezialsymbole (engl. separator, operator):

Operatoren, Trenner von Sprachkonstrukten, z. B. `; , = * <=`

Schreibweise festgelegt, meist Folge von Sonderzeichen

Bezeichner und Literale tragen außer der Klassenzugehörigkeit weitere Information:
Identität des Bezeichners und **Wert des Literals**.

Wortsymbole und Spezialsymbole stehen nur für sich selbst, tragen keine weitere Information.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 202a

Ziele:

Klassen und Schreibweisen von Grundsymbolen kennenlernen

in der Vorlesung:

- Grundsymbolklassen erläutern
- Typische Schreibweisen erläutern
- Beispiele aus Java, C, Pascal, FORTRAN

Verständnisfragen:

- Erläutern Sie den Unterschied zwischen dem Literal für eine ganze Zahl und ihrem Wert.
- Wie wird in Java das Berandungszeichen in Zeichen- oder Zeichenreihenliteralen dargestellt?
- Welche Regeln gelten dafür in Pascal?

Trennung von Grundsymbolen

In den meisten Sprachen haben
die Zeichen **Zwischenraum, Zeilenwechsel, Tabulator** und **Kommentare**
keine Bedeutung außer zur Trennung von Grundsymbolen; auch **white space** genannt.
z. B. `int pegel;` statt `intpegel;`

Ausnahme **Fortran**:
Zwischenräume haben auch innerhalb von Grundsymbolen keine Bedeutung
z. B. Zuweisung `DO 5 I = 1.5` gleichbedeutend wie `DO5I=1.5` aber
Schleifenkopf `DO 5 I = 1,5`

In **Fortran, Python, Occam** können Anweisungen
durch Zeilenwechsel getrennt werden.
In **Occam** und **Python** werden Anweisungen durch
gleiche Einrücktiefe zusammengefasst

```
if (x < y)
   a = x
   b = y
print (x)
```

Häufigste Schreibweisen von **Kommentaren**:
geklammert , z. B.

```
int pegel; /* geklammerter Kommentar */
```

oder **Zeilenkommentar** bis zum Zeilenende, z. B.

```
int pegel; // Zeilenkommentar
```

Geschachtelte Kommentare z.B. in **Modula-2**:

```
/* aeusserer /* innerer */ Kommentar */
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 203

Ziele:

Zeichen zwischen Grundsymbolen

in der Vorlesung:

Verständnisfragen:

- Warum ist die Regel für Zwischenräume in Fortran problematisch?
- Welche Schreibweisen für Kommentare gibt es in Modula-2?

2.2 Kontext-freie Grammatiken; Definition

Kontext-freie Grammatik (KFG, engl. CFG):
formaler Kalkül zur **Definition von Sprachen** und **von Bäumen**

Die **konkrete Syntax** einer Programmiersprache oder anderen formalen Sprache wird durch eine KFG definiert. (Ebene b, GPS 1-16)

Die **Strukturbäume** zur Repräsentation von Programmen in Übersetzern werden als **abstrakte Syntax** durch eine KFG definiert.

Eine **kontext-freie Grammatik G = (T, N, P, S)** besteht aus:

T	Menge der **Terminalsymbole**	Daraus bestehen Sätze der Sprache; Grundsymbole
N	Menge der **Nichtterminalsymbole**	Daraus werden Sprachkonstrukte abgeleitet.
$S \in N$	**Startsymbol** (auch **Zielsymbol**)	Daraus werden Sätze abgeleitet.
$P \subseteq N \times V^*$	Menge der **Produktionen**	Regeln der Grammatik.

außerdem wird $V = T \cup N$ als **Vokabular** definiert; T und N sind disjunkt

Produktionen haben also die Form **A ::= x**, mit $A \in N$ und $x \in V^*$
d.h. x ist eine evtl. leere Folge von Symbolen des Vokabulars.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 204

Ziele:

Kalkül kontext-freie Grammatik wiederholen

in der Vorlesung:

- Erläuterungen dazu am Beispiel der Ausdrucksgrammatik;
- Grundsymbole sind die Terminalsymbole der konkreten Syntax;
- KFG wird benutzt, um Programme zu schreiben und um existierende Programme zu strukturieren, prüfen, verstehen.

nachlesen:

Skript zu Modellierung, Kap. 5.1

Verständnisfragen:

- Wie kann man aus der Menge der Produktionen die Mengen der Terminale, Nichtterminale und das Startsymbol bestimmen?

KFG Beispiel: Grammatik für arithmetische Ausdrücke

G_{aA} = (T, N, P, S) besteht aus:

T	**Terminalsymbole**	{ '(', ')', '+', '-', '*', '/', Ident }
N	**Nichtterminalsymbole**	{ Expr, Fact, Opd, AddOpr, MulOpr}
S ∈ N	**Startsymbol**	**Expr**
P ⊆ N × V*	**Produktionen**	

P Menge der Produktionen:

Häufig gibt man Produktionen Namen:

```
p1:  Expr   ::= Expr AddOpr Fact
p2:  Expr   ::= Fact
p3:  Fact   ::= Fact MulOpr Opd
p4:  Fact   ::= Opd
p5:  Opd    ::= '(' Expr ')'
p6:  Opd    ::= Ident
p7:  AddOpr ::= '+'
p8:  AddOpr ::= '-'
p9:  MulOpr ::= '*'
p10: MulOpr ::= '/'
```

Unbenannte Terminalsymbole kennzeichnen wir in Produktionen, z.B. '+'

Es werden meist nur die Produktionen (und das Startsymbol) einer kontext-freien Grammatik angegeben, wenn sich die übrigen Eigenschaften daraus ergeben.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 204a

Ziele:

Kalkül kontext-freie Grammatik wiederholen

in der Vorlesung:

- Erläuterungen dazu am Beispiel der Ausdrucksgrammatik;
- Grundsymbole sind die Terminalsymbole der konkreten Syntax;
- KFG wird benutzt, um Programme zu schreiben und um existierende Programme zu strukturieren, prüfen, verstehen.

nachlesen:

Skript zu Modellierung, Kap. 5.1

Verständnisfragen:

- Wie kann man aus der Menge der Produktionen die Mengen der Terminale, Nichtterminale und das Startsymbol bestimmen?

Ableitungen

Produktionen sind **Ersetzungsregeln**:
Ein Nichtterminal A in einer Symbolfolge u A v kann durch die rechte Seite x einer Produktion A ::= x ersetzt werden.

Das ist ein **Ableitungsschritt** u **A** v ⇒ u **x** v
z. B. `Expr AddOpr Fact` ⇒ `Expr AddOpr Fact MulOpr Opd` mit Produktion p3

Beliebig viele Ableitungsschritte nacheinander angewandt heißen **Ableitung**: u ⇒* v

Eine kontext-freie Grammatik **definiert eine Sprache**, d. h. die Menge von Terminalsymbolfolgen, die aus dem Startsymbol S ableitbar sind:

$$L(G) = \{ w \mid w \in T^* \text{ und } S \Rightarrow^* w \}$$

Die Grammatik aus GPS-2-4a definiert z. B. Ausdrücke als Sprachmenge:

$$L(G) = \{ w \mid w \in T^* \text{ und } Expr \Rightarrow^* w \}$$

$$\{ \texttt{Ident, Ident + Ident, Ident + Ident * Ident} \} \subset L(G)$$

oder mit verschiedenen Bezeichnern für die Vorkommen des Grundsymbols `Ident:`

$$\{ \texttt{a, b + c, a + b * c} \} \subset L(G)$$

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 205

Ziele:
Das Prinzip der Ableitung verstehen

in der Vorlesung:
Erläuterungen am Beispiel der Ausdrucksgrammatik

Übungsaufgaben:

Verständnisfragen:
- Geben Sie eine Ableitung zu `a*b/c` an.
- Gibt es weitere zum selben Satz? Wie unterscheiden sie sich?
- Geben Sie Folgen von Terminalsymbolen an, die nicht zur Sprache der Grammatik gehören.

Beispiel für eine Ableitung

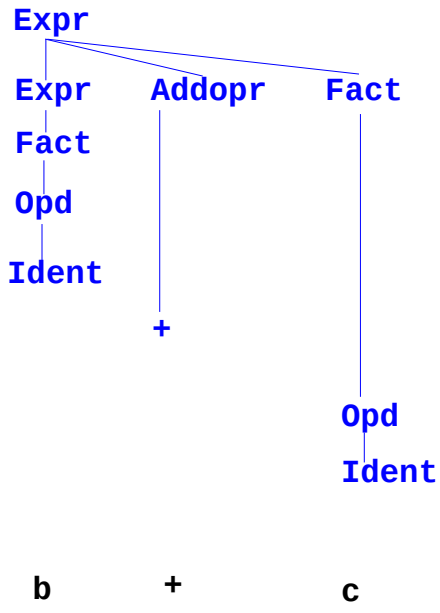
Satz der Ausdrucksgrammatik **b + c**

Ableitung:

```
          Expr
p1   => Expr    Addopr Fact
p2   => Fact    Addopr Fact
p4   => Opd     Addopr Fact
p6   => Ident   Addopr Fact
p7   => Ident   +      Fact
p4   => Ident   +      Opd
p6   => Ident   +      Ident

          b     +      c
```

Ableitungsbaum:



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 206

Ziele:

Ableitung wiederholen

in der Vorlesung:

Zusammenhang zum Ableitungsbaum zeigen

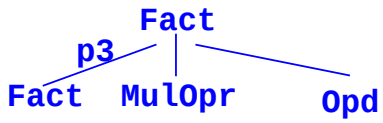
nachlesen:

Text

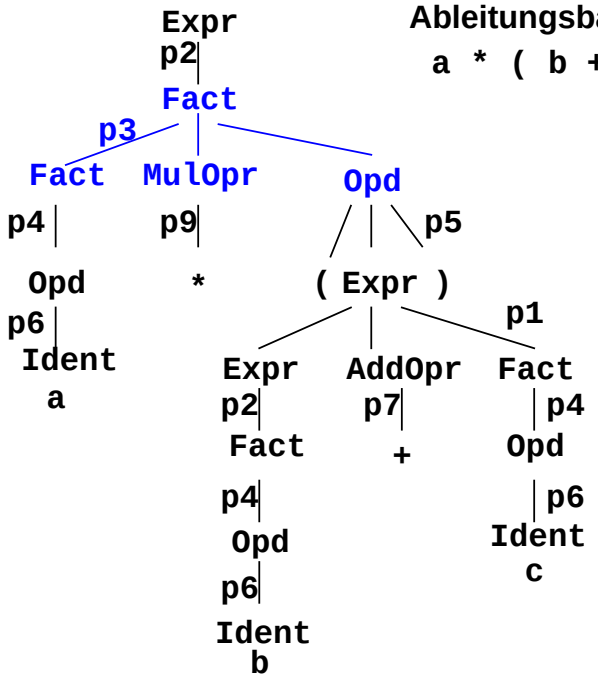
Ableitungsbäume

Jede Ableitung kann man als **Baum** darstellen. Er **definiert die Struktur des Satzes**.
Die **Knoten** repräsentieren **Vorkommen von Terminalen und Nichtterminalen**.
Ein **Ableitungsschritt** mit einer Produktion wird dargestellt durch Kanten zwischen dem Knoten für das Symbol der linken und denen für die Symbole der rechten Seite der Produktion:

Anwendung der Produktion p3:



Ableitungsbaum für
a * (b + c)



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 207

Ziele:

Ableitungsbäume verstehen

in der Vorlesung:

- Erläuterungen dazu
- Beispiele für mehrdeutige Grammatiken

Übungsaufgaben:

Verständnisfragen:

- Zeigen Sie an dem Satz `a*b+c`, dass der Ableitungsbaum wichtige Aussagen zur Struktur des Satzes enthält.

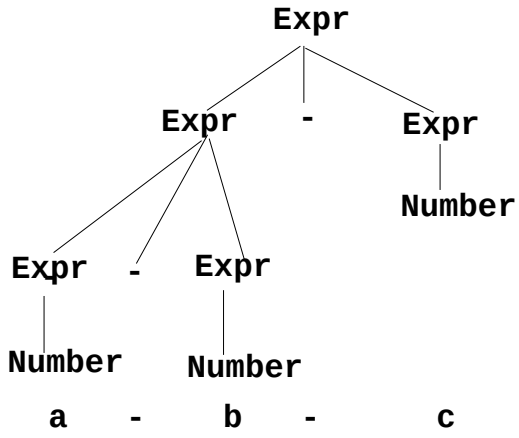
Mehrdeutige kontext-freie Grammatik

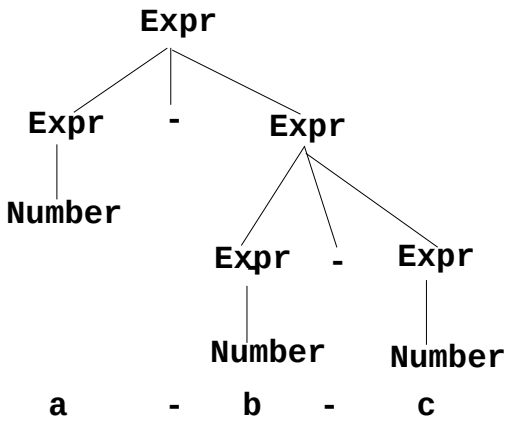
Eine kontext-freie Grammatik ist genau dann **mehrdeutig**, wenn
es einen **Satz aus ihrer Sprache gibt**,
zu dem es **zwei verschiedene Ableitungsbäume gibt.**

Beispiel für eine mehrdeutige KFG:

```
Expr ::= Expr '-' Expr
Expr ::= Number
```

ein Satz, der 2 verschiedene Ableitungsbäume hat:





Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 208

Ziele:

Mehrdeutigkeit verstehen

in der Vorlesung:

- Definition wiederholen;
- Beispiel erläutern;
- ein Satz mit verschiedenen Ableitungsbäumen ist mehrdeutig;
- zeigen, dass verschiedene Strukturen unterschiedliche Bedeutung haben können.

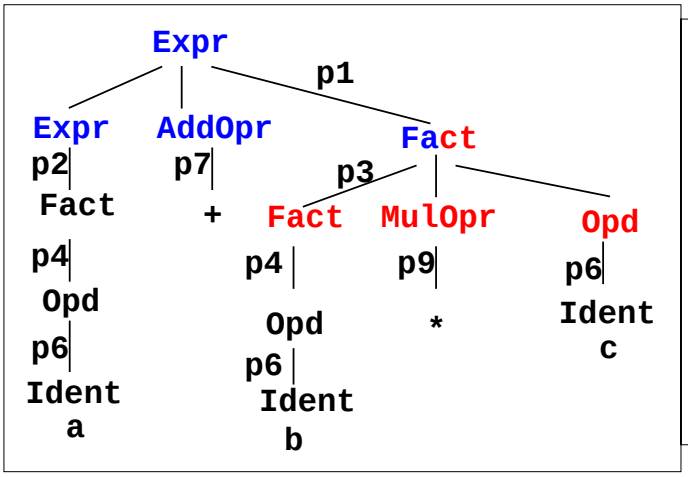
Verständnisfragen:

- Geben Sie andere mehrdeutige Sätze zu der Grammatik an.
- Geben Sie Sätze zu der Grammatik an, die nicht mehrdeutig sind.
- Geben Sie andere mehrdeutige Grammatiken an.

Ausdrucksgrammatik

Die Struktur eines Satzes wird durch seinen Ableitungsbaum bestimmt.
Ausdrucksgrammatiken legen dadurch die **Präzedenz** und **Assoziativität** von Operatoren fest.

Im Beispiel hat **`AddOpr` geringere Präzedenz als `MulOpr`**, weil er
höher in der Hierarchie der Kettenproduktionen `Expr ::= Fact`, `Fact ::= Opd` steht.



Name	Produktion
p1:	`Expr ::= Expr AddOpr Fact`
p2:	`Expr ::= Fact`
p3:	`Fact ::= Fact MulOpr Opd`
p4:	`Fact ::= Opd`
p5:	`Opd ::= '(' Expr ')'`
p6:	`Opd ::= Ident`
p7:	`AddOpr ::= '+'`
p8:	`AddOpr ::= '-'`
p9:	`MulOpr ::= '*'`
p10:	`MulOpr ::= '/'`

Im Beispiel sind `AddOpr` und `MulOpr` **links-assoziativ**, weil ihre **Produktionen links-rekursiv** sind, d. h. **`a + b - c`** entspricht **`(a + b) - c`**.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 209

Ziele:

Systematische Struktur von Ausdrucksgrammatiken verstehen

in der Vorlesung:

- Erläuterungen dazu am Beispiel
- Variation des Beispiels

Übungsaufgaben:

Geben Sie eine Ausdrucksgrammatik für die Java-Operatoren auf SWE-30 an.

Verständnisfragen:

- Wie sind die Operatoren in der Java-Grammatik definiert?
- Wie ändert sich die Sprache, wenn Produktion `p1` durch `Expr ::= Fact '+' Fact` ersetzt wird? Für welche Art von Operatoren wäre das sinnvoll?

Schemata für Ausdrucksgrammatiken

Ausdrucksgrammatiken konstruiert man **schematisch**,
sodass **strukturelle Eigenschaften** der Ausdrücke definiert werden:

eine Präzedenzstufe, binärer Operator, linksassoziativ:

```
A ::= A Opr B
A ::= B
```

eine Präzedenzstufe, binärer Operator, **rechtsassoziativ**:

```
A ::= B Opr A
A ::= B
```

eine Präzedenzstufe, **unärer Operator**, präfix:

```
A ::= Opr A
A ::= B
```

eine Präzedenzstufe, unärer Operator, **postfix**:

```
A ::= A Opr
A ::= B
```

Elementare Operanden: nur aus dem Nichtterminal der höchsten Präzedenzstufe (sei hier H) abgeleitet:

```
H ::= Ident
```

Geklammerte Ausdrücke: nur aus dem Nichtterminal der höchsten Präzedenzstufe (sei hier H) abgeleitet; enthalten das Nichtterminal der niedrigsten Präzedenzstufe (sei hier A)

```
H ::= '(' A ')'
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 210

Ziele:

Schemata anwenden können

in der Vorlesung:

Erläuterungen dazu

Übungsaufgaben:

Anwenden der Schemata zur Konstruktion und zum Verstehen von Ausdrucksgrammatiken

Notationen für kontext-freie Grammatiken

Eine kontext-freie Grammatik wurde 1959 erstmals zur Definition einer Programmiersprache (Algol 60) verwendet. Name für die Notation - noch heute: **Backus Naur Form (BNF)**.

Entweder werden **Symbolnamen gekennzeichnet**,
z. B. durch Klammerung **<Expr>** oder durch den Schrifttyp *Expr.*
oder unbenannte **Terminale**, die für sich stehen, werden **gekennzeichnet**, z. B. **'('**

Zusammenfassung von Produktionen mit gleicher linker Seite:

```
Opd   ::=   '(' Expr ')'
      |     Ident
```

oder im Java -Manual:

```
Opd:
   ( Expr )
   Ident
```


Ziele:
Gebräuchliche Notationen kennenlernen

in der Vorlesung:
Erläuterungen dazu

Erweiterte Notation EBNF

Backus Naur Form (BNF) erweitert um Konstrukte regulärer Ausdrücke zu **Extended BNF**

EBNF		gleichbedeutende BNF-Produktionen		
X ::= u **(v)** w	Klammerung	X ::= u Y w	Y ::= v	
X ::= u **[v]** w	optional	X ::= u Y w	Y ::= v	Y ::= ε
X ::= u **s*** w	optionale	X ::= u Y w	Y ::= s Y	Y ::= ε
X ::= u **{ s }** w	Wiederholung			
X ::= u **s ...** w	Wiederholung	X ::= u Y w	Y ::= s Y	Y ::= s
X ::= u **s+** w				
X ::= u (**v || s**) w	Wdh. mit Trenner	X ::= u Y w	Y ::= v s Y	Y ::= v
X ::= u (**v1 | v2**) w	Alternativen	X ::= u Y w	Y ::= v1	Y ::= v2

Dabei sind u, v, v1, v2, w ∈ V* s ∈ V X, Y ∈ N
Y ist ein Nichtterminal, das sonst nicht in der Grammatik vorkommt.

Beispiele:

Block ::= '{' **Statement*** '}'	Block ::= '{' **Y** '}'	Y ::= Statement Y	Y ::= ε
Decl ::= Type **(Ident || ',')** ';'	Decl ::= Type **Y** ';'	Y ::= Ident ',' Y	Y ::= Ident

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 212

Ziele:

EBNF kennenlernen

in der Vorlesung:

- Erläuterungen der EBNF Konstrukte
- Transformation von EBNF in BNF

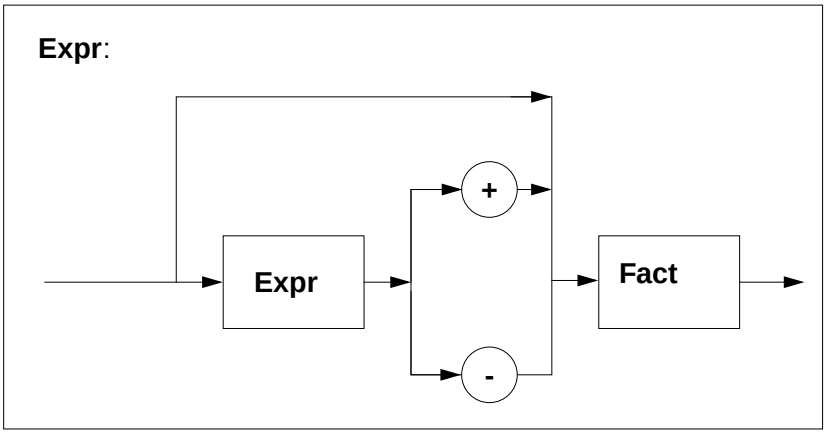
nachlesen:

..., Abschnitt

Verständnisfragen:

- Welche EBNF-Notationen werden in der Java-Sprachspezifikation verwendet?

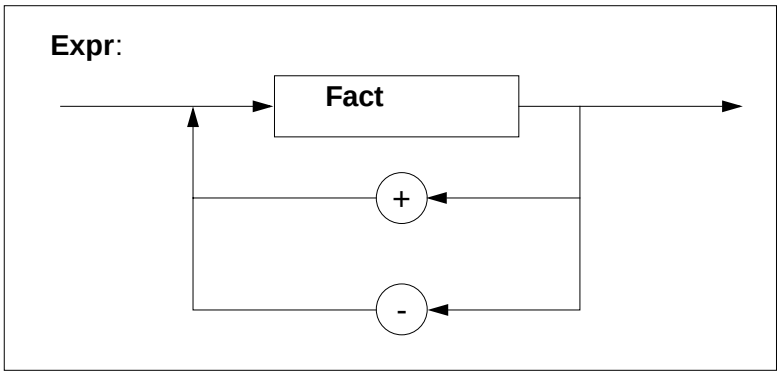
Syntaxdiagramme



Ein **Syntaxdiagramm repräsentiert eine EBNF-Produktion**:

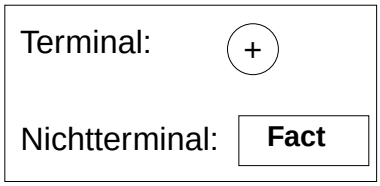
Expr ::= [Expr ('+' | '-')] Fact

Option und Alternative



Expr ::= (Fact || ('+' | '-'))

Wiederholung mit alternativem Trenner



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 213

Ziele:

Grafische Notation verstehen

in der Vorlesung:

- Erläuterungen dazu
- Vergleich mit textuellen Produktionen

nachlesen:

siehe z. B. Pascal-Report

Verständnisfragen:

- Zeigen Sie die Zuordnung zwischen EBNF-Formen und Syntaxdiagrammen.

Produktionen-Schemata für Folgen

Beschreibung	Produktionen	Sprachmenge
nicht-leere Folge von b	A ::= A b | b	{b, bb, bbb, ...}
nicht-leere Folge von b	A ::= b A | b	{b, bb, bbb, ...}
evtl. leere Folge von b	A ::= A b |	{ε, b, bb, bbb, ...}
evtl. leere Folge von b	A ::= b A |	{ε, b, bb, bbb, ...}
nicht-leere Folge von b getrennt durch t	A ::= A t b | b	{b, btb, btbtb, ...}
nicht-leere Folge von b getrennt durch t	A ::= b t A | b	{b, btb, btbtb, ...}

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 214

Ziele:

Folgen konstruieren können

in der Vorlesung:

- Erläuterungen und Beispiele dazu

nachlesen:

..., Abschnitt

nachlesen:

Übungsaufgaben:

Grammatik-Entwurf: Folgen

Produktionen für **Folgen von Sprachkonstrukten** systematisch konstruieren.
Schemata hier am Beispiel von Anweisungsfolgen (`Stmts`)

Folgen mit Trenner:

```
a. Stmts    ::=  Stmts ';' Stmt  | Stmt                 linksrekursiv
b. Stmts    ::=  Stmt  ';' Stmts | Stmt                 rechtsrekursiv
c. Stmts    ::=  ( Stmt || ';')                         EBNF
d. StmtsOpt::=  Stmts |                                 mit leerer Folge
```

Folgen mit Terminator:

```
a. Stmts    ::=  Stmt ';' Stmts | Stmt ';'              rechtsrekursiv
b. Stmts    ::=  Stmt     Stmts | Stmt                  Terminator an den Elementen
   Stmt     ::=  Assign ';' | ...
c. Stmts    ::=  Stmts Stmt| Stmt                       linksrekursiv
   Stmt     ::=  Assign ';' | ...
d. Stmts    ::=  ( Stmt ';')+                           EBNF
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 214a

Ziele:

Folgen konstruieren können

in der Vorlesung:

- Erläuterungen und Beispiele dazu

nachlesen:

..., Abschnitt

nachlesen:

Übungsaufgaben:

Geben Sie eine kontext-freie Grammatik für bedingte Anweisungen und für while-Schleifen an.

Grammatik-Entwurf: Klammern

Klammern: Paar von Terminalen, das eine Unterstruktur einschließt:

```
Operand ::=  '(' Expression ')'
Stmt    ::=  'while' Expr 'do' Stmts 'end'
Stmt    ::=  'while' Expr 'do' Stmts 'end'

MethodenDef ::=
   ErgebnisTyp MethodenName '(' FormaleParameter ')' Rumpf
```

Stilregel: Öffnende und schließende Klammer immer in derselben Produktion

gut: `Stmt ::= 'while' Expr 'do' Stmts 'end'`

schlecht:
```
Stmt       ::= WhileKopf Stmts 'end'
WhileKopf  ::= 'while' Expr 'do'
```

Nicht-geklammerte (offene) Konstrukte können **Mehrdeutigkeiten** verursachen:

```
Stmt    ::= 'if' Expr 'then' Stmt
          |   'if' Expr 'then' Stmts 'else' Stmt
```

Offener, optionaler else-Teil verursacht **Mehrdeutigkeit** in C, C++, Pascal, sog. "dangling else"-Problem:

```
if c then if d then S1 else S2
```

In diesen Sprachen gehört `else S2` zur **inneren** if-Anweisung.

Java enthält das gleiche if-Konstrukt. Die Grammatik vermeidet die Mehrdeutigkeit durch Produktionen, die die Bindung des `else` explizit machen.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 215

Ziele:

Mit Klammern umgehen können

in der Vorlesung:

- Erläuterungen und Beispiele dazu
- Erläuterung des "dangling else"

Verständnisfragen:

- Zeigen Sie die Mehrdeutigkeit des "dangling else" an Ableitungsbäumen.

Abstrakte Syntax

konkrete Syntax

KFG definiert
Symbolfolgen (Programmtexte) und deren **Ableitungsbäume**

konkrete Syntax bestimmt die Struktur von Programmkonstrukten, z. B. Präzedenz und Assozitivität von Operatoren in Ausdrücken

Präzedenzschemata benötigen **Kettenproduktionen**, d.h. Produktionen mit genau einem Nichtterminal auf der rechtenSeite:

```
Expr ::= Fact
Fact ::= Opd
Opd  ::= '(' Expr ')'
```

Mehrdeutigkeit ist problematisch

Alle Terminale sind nötig.

abstrakte Syntax

KFG definiert
abstrakte Programmstruktur durch **Strukturbäume**

statische und dynamische Semantik werden auf der abstrakten Syntax definiert

solche Kettenproduktionen sind hier **überflüssig**

Mehrdeutigkeit ist akzeptabel

Terminale, die nur für sich selbst stehen und **keine Information** tragen, sind hier **überflüssig (Wortsymbole, Spezial-symbole)**, z.B. `class ( ) + - * /`

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 216

Ziele:

Prinzip der abstrakten Syntax verstehen

in der Vorlesung:

- KFG ausschließlich zur Definition von Bäumen.
- Zusammenhang zur konkreten Syntax zeigen.
- Beispiel auf der nächsten Folie erläutern.

Verständnisfragen:

- Geben Sie eine abstrakte Syntax zu den Anweisungsformen auf SWE-31 an.

Abstrakte Ausdrucksgrammatik

konkrete Ausdrucksgrammatik

```
p1:    Expr   ::= Expr AddOpr Fact
p2:    Expr   ::= Fact
p3:    Fact   ::= Fact MulOpr Opd
p4:    Fact   ::= Opd
p5:    Opd    ::= '(' Expr ')'
p6:    Opd    ::= Ident
p7:    AddOpr ::= '+'
p8:    AddOpr ::= '-'
p9:    MulOpr ::= '*'
p10:   MulOpr ::= '/'
```

abstrakte Ausdrucksgrammatik

Name	Produktion
BinEx:	Exp ::= Exp BinOpr Exp
IdEx:	Exp ::= Ident
PlusOpr:	BinOpr ::=
MinusOpr:	BinOpr ::=
TimesOpr:	BinOpr ::=
DivOpr:	BinOpr ::=

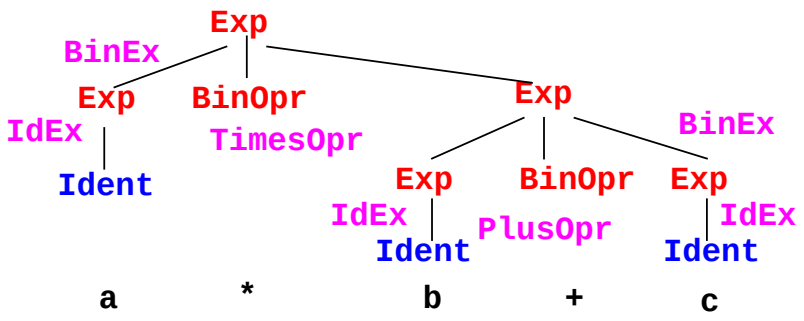
Abbildung konkret -> abstrakt

```
Expr,Fact,Opd -> Exp
AddOpr,MulOpr -> BinOpr

p1,p3 -> BinEx
p2,p4,p5 ->
p6 -> IdEx

p7 -> PlusOpr
...
```

Strukturbaum für `a * (b + c)`



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 217

Ziele:

Beispiel zur vorigen Folie

in der Vorlesung:

- Bezüge zwischen konkreten und abstrakten Produktionen und Nichtterminalen zeigen;
- Strukturbaum und Ableitungsbaum vergleichen.

2.3 XML
Übersicht

XML (Extensible Markup Language, dt.: Erweiterbare Auszeichnungssprache)

- seit 1996 vom W3C definiert, in Anlehnung an SGML
- Zweck: Beschreibungen **allgemeiner Strukturen** (nicht nur Web-Dokumente)
- **Meta-Sprache** (“erweiterbar”):
  Die Notation ist festgelegt (Tags und Attribute, wie in HTML),
  Für beliebige Zwecke kann **jeweils eine spezielle syntaktische Struktur** definiert werden (DTD)
  Außerdem gibt es Regeln (XML-Namensräume), um XML-Sprachen in andere **XML-Sprachen zu importieren**
- X**HTML** ist so als XML-Sprache definiert
- Weitere aus XML **abgeleitete Sprachen**: SVG, MathML, SMIL, RDF, WML
- **individuelle XML-Sprachen** werden benutzt, um strukturierte Daten zu speichern, die von **Software-Werkzeugen geschrieben und gelesen** werden
- XML-Darstellung von strukturierten Daten kann mit verschiedenen Techniken **in HTML transformiert** werden, um sie **formatiert anzuzeigen:**
  XML+CSS, XML+XSL, SAX-Parser, DOM-Parser

Dieser Abschnitt orientiert sich eng an **SELFHTML** (Stefan Münz), http://de.selfhtml.org

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 219

Ziele:

Rolle von XML verstehen

in der Vorlesung:

Die Aspekte werden einführend erklärt.

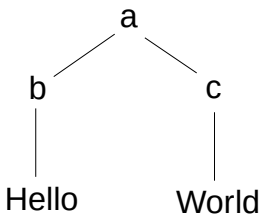
3 elementare Prinzipien

Die XML-Notation basiert auf 3 elementaren Prinzipien:

A: Vollständige Klammerung durch Tags

```
<a>
    <b>Hello</b>
    <c>World</c>
</a>
```

B: Klammerstruktur ist äquivalent zu gewurzeltem Baum



C: Kontextfreie Grammatik definiert Bäume; eine DTD ist eine KFG

```
a ::= b c
b ::= PCDATA
c ::= PCDATA
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 219a

Ziele:

Prinzipien der XML-Notation

in der Vorlesung:

Kurze Erklärung der Prinzipien.

Notation und erste Beispiele

Ein Satz in einer XML-Sprache ist ein Text, der durch Tags strukturiert wird.

Tags werden **immer** in **Paaren von Anfangs- und End-Tag** verwendet:

```
<ort>Paderborn</ort>
```

Anfangs-Tags können Attribut-Wert-Paare enthalten:

```
<telefon typ="dienst">05251606686</telefon>
```

Die **Namen von Tags und Attributen** können für die XML-Sprache **frei gewählt** werden.

Mit Tags gekennzeichnete Texte können geschachtelt werden.

```
<adressBuch>
<adresse>
   <name>
      <nachname>Mustermann</nachname>
      <vorname>Max</vorname>
   </name>
   <anschrift>
      <strasse>Hauptstr 42</strasse>
      <ort>Paderborn</ort>
      <plz>33098</plz>
   </anschrift>
</adresse>
</adressBuch>
```

$(a+b)^2$ in MathML:

```
<msup>
   <mfenced>
      <mrow>
         <mi>a</mi>
         <mo>+</mo>
         <mi>b</mi>
      </mrow>
   </mfenced>
   <mn>2</mn>
</msup>
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 220

Ziele:

Notation von XML verstehen

in der Vorlesung:

An den Beispielen wird erklärt:

- Tags und Attribute werden für den speziellen Zweck frei erfunden,
- ein Tag-Paar begrenzt ein Element und benennt seine Rolle,
- geschachtelte Strukturen.
- Wir entwerfen eigene Sprachen!!

Ein vollständiges Beispiel

Kennzeichnung des Dokumentes als XML-Datei

Datei mit der Definition der Syntaktischen Struktur dieser XML-Sprache (DTD)

Datei mit Angaben zur Transformation in HTML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE produktnews SYSTEM "produktnews.dtd">
<?xml-stylesheet type="text/xsl" href="produktnews.xsl" ?>
<produktnews>
   Die neuesten Produktnachrichten:
   <beschreibung>
      Die Firma <hersteller>Fridolin Soft</hersteller> hat eine neue
      Version des beliebten Ballerspiels
      <produkt>HitYourStick</produkt> herausgebracht. Nach Angaben des
      Herstellers soll die neue Version, die nun auch auf dem
      Betriebssystem <produkt>Ganzfix</produkt> läuft, um die
      <preis>80 Dollar</preis> kosten.
   </beschreibung>
   <beschreibung>
      Von <hersteller>Ripfiles Inc.</hersteller> gibt es ein Patch zu der Sammel-CD
      <produkt>Best of other people's ideas</produkt>. Einige der tollen
      Webseiten-Templates der CD enthielten bekanntlich noch versehentlich nicht
      gelöschte Angaben der Original-Autoren. Das Patch ist für schlappe
      <preis>200 Euro</preis> zu haben.
   </beschreibung>
</produktnews>
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 221

Ziele:

Technische Angaben sehen

in der Vorlesung:

Am Beispiel wird erklärt:

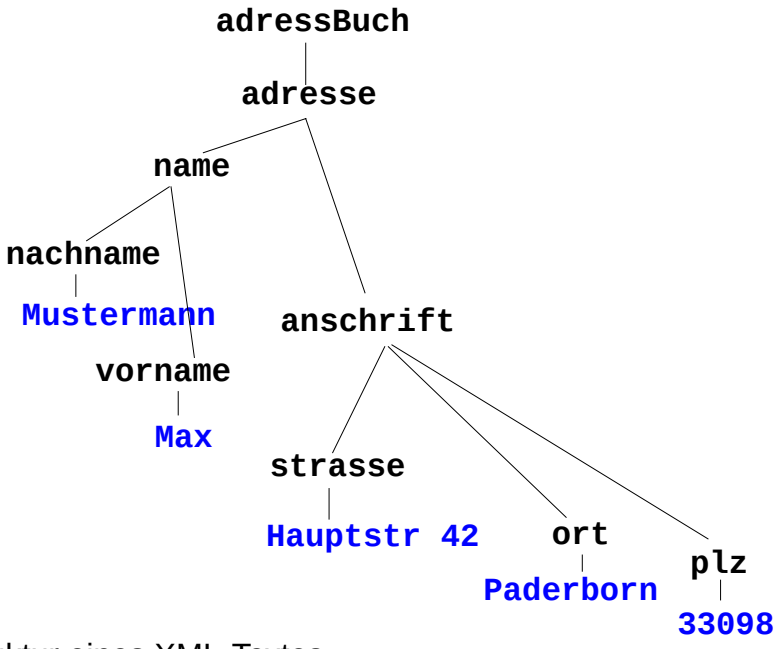
- die 3 technischen Angaben,
- Text-Dokument als Beispiel.
- Beispiel wird noch weiterverwendet.

Baumdarstellung von XML-Texten

Jeder XML-Text ist durch Tag-Paare **vollständig geklammert** (wenn er *well-formed* ist).

Deshalb kann er eindeutig **als Baum dargestellt** werden. (Attribute betrachten wir noch nicht)
Wir markieren die inneren Knoten mit den Tag-Namen; die Blätter sind die elementaren Texte:

```
<adressBuch>
<adresse>
   <name>
      <nachname>Mustermann
      </nachname>
      <vorname>Max
      </vorname>
   </name>
   <anschrift>
      <strasse>Hauptstr 42
      </strasse>
      <ort>Paderborn</ort>
      <plz>33098</plz>
   </anschrift>
</adresse>
</adressBuch>
```



XML-Werkzeuge können die Baumstruktur eines XML-Textes ohne weiteres ermitteln und ggf. anzeigen.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 222

Ziele:

XML-Text als Baum verstehen

in der Vorlesung:

Am Beispiel wird erklärt:

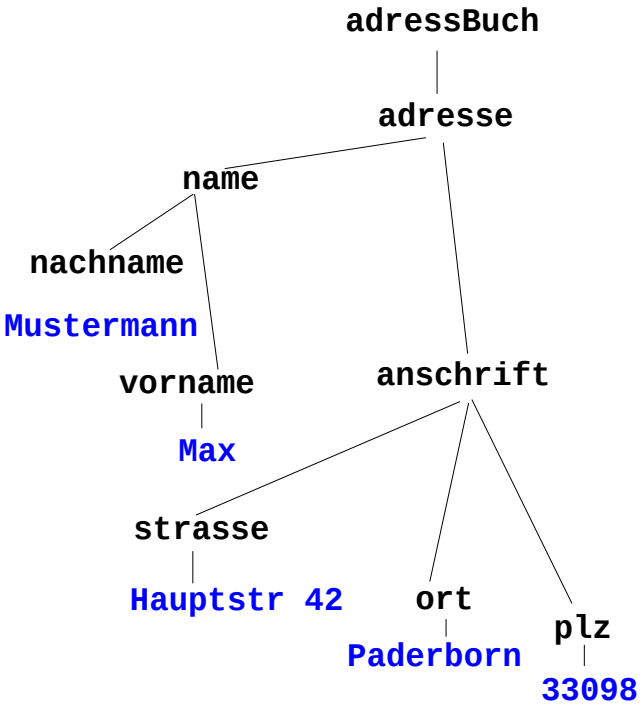
- vollständige Klammerung durch Tags,
- definiert einen Baum,
- aus dem Baum kann man den Text wiederherstellen

Grammatik definiert die Struktur der XML-Bäume

Mit **kontextfreien Grammatiken (KFG)** kann man **Bäume definieren**.

Folgende KFG definiert korrekt strukturierte Bäume für das Beispiel Adressbuch:

```
adressBuch ::= adresse*
adresse    ::= name anschrift
name       ::= nachname vorname
Anschrift  ::= strasse ort plz
nachname   ::= PCDATA
vorname    ::= PCDATA
strasse    ::= PCDATA
ort        ::= PCDATA
plz        ::= PCDATA
```



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 223

Ziele:

Definition durch KFG verstehen

in der Vorlesung:

Am Beispiel wird erklärt:

- Tag-Namen werden Nichtterminale,
- PCDATA ist das Terminal für die elementaren Texte,
- weiteren Baum skizzieren.

Document Type Definition (DTD) statt KFG

Die Struktur von XML-Bäumen und -Texten wird in der **DTD-Notation** definiert. Ihre Konzepte entsprechen denen von KFGn:

KFG

```
adressBuch ::= adresse*
adresse    ::= name anschrift
name       ::= nachname vorname
Anschrift  ::= strasse ort plz
nachname   ::= PCDATA
vorname    ::= PCDATA
strasse    ::= PCDATA
ort        ::= PCDATA
plz        ::= PCDATA
```

DTD

```
<!ELEMENT adressBuch(adresse)*          >
<!ELEMENT adresse   (name, anschrift)   >
<!ELEMENT name      (nachname, vorname)>
<!ELEMENT anschrift (strasse, ort, plz)>
<!ELEMENT nachname  (#PCDATA)           >
<!ELEMENT vorname   (#PCDATA)           >
<!ELEMENT strasse   (#PCDATA)           >
<!ELEMENT ort       (#PCDATA)           >
<!ELEMENT plz       (#PCDATA)           >
```

weitere Formen von DTD-Produktionen:

`X (Y)+`	nicht-leere Folge
`X (A | B)`	Alternative
`X (A)?`	Option
`X EMPTY`	leeres Element

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 224

Ziele:

DTD-Notation als KFG verstehen

in der Vorlesung:

Am Beispiel wird erklärt:

- Zuordnung der KFG- zu DTD-Konstrukten,
- Erklärung der weiteren Formen an Beispielen.
- Hinweis: Die DTD-Notation zur Definition von Attributlisten in Anfangs-Tags wird hier nicht beschrieben.

Zusammenfassung zu Kapitel 2

Mit den Vorlesungen und Übungen zu Kapitel 2 sollen Sie nun Folgendes können:

- Notation und Rolle der Grundsymbole kennen.
- Kontext-freie Grammatiken für praktische Sprachen lesen und verstehen.
- Kontext-freie Grammatiken für einfache Strukturen selbst entwerfen.
- Schemata für Ausdrucksgrammatiken, Folgen und Anweisungsformen anwenden können.
- EBNF sinnvoll einsetzen können.
- Abstrakte Syntax als Definition von Strukturbäumen verstehen.
- XML als Meta-Sprache zur Beschreibung von Bäumen verstehen
- DTD von XML als kontext-freie Grammatik verstehen
- XML lesen können

Ziele:
Ziele des Kapitels erkennen

in der Vorlesung:
Erläuterungen dazu

3. Gültigkeit von Definitionen

Themen dieses Kapitels:

- Definition und Bindung von Bezeichnern
- Verdeckungsregeln für die Gültigkeit von Definitionen
- Gültigkeitsregeln in Programmiersprachen

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 301

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

Definition und Bindung

Eine **Definition** ist ein Programmkonstrukt, das die **Beschreibung eines Programmgegenstandes an einen Bezeichner bindet.**

Programmkonstrukt: zusammengehöriger Teil (Teilbaum) eines Programms
z. B. eine Deklaration `int i;`, eine Anweisung `i = 42;` Ausdruck `i+1`

Programmgegenstand: wird im Programm beschrieben und benutzt
z. B. die Methode `main`, der Typ `String`, eine Variable `i`, ein Parameter `args`

Meist legt die Definition Eigenschaften des Programmgegenstandes fest, z. B. den Typ:

```
public static void main (String[] args)
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 302

Ziele:

Grundbegriffe der Bindung von Namen verstehen

in der Vorlesung:

- Erläuterung der Begriffe an Beispielen
- "Definitionen" werden auch "Deklarationen" genannt (Java); für C und C++ werden beide Bezeichnungen mit unterschiedlicher Bedeutung verwendet.

nachlesen:

..., Abschnitt Kapitel 4 Bindung

Übungsaufgaben:

Verständnisfragen:

- Welche Arten von benannten Programmgegenständen gibt es in Java?
- Geben Sie zu jeder eine Definition und eine Benutzung des Bezeichners an.
- Erläutern Sie: In Java gibt es auch unbenannte Klassen und Packages.

Statische und dynamische Bindung

Ein Bezeichner, der in einer Definition gebunden wird, tritt dort **definierend** auf; an anderen Stellen tritt er **angewandt** auf.
Definierendes und angewandtes Auftreten von Bezeichnern kann man meist **syntaktisch unterscheiden**, z. B.

```
static int ggt (int a, int b)
{ ...
  return ggt(a % b, b);
...
}
```

Regeln der Sprache entscheiden, in welcher Definition ein angewandtes Auftreten eines Bezeichners gebunden ist.

Statische Bindung:
Gültigkeitsregeln entscheiden die Bindung am **Programmtext**, z. B.

```
{ float a = 1.0;
  { int a = 2;
    printf ("%d",a);
  }
}
```

statische Bindung im Rest dieses Kapitels und in den meisten Sprachen, außer ...

Dynamische Bindung:
Wird bei der **Ausführung des Programms** entschieden:
Für einen angewandten Bezeichner **a** gilt die zuletzt für **a** **ausgeführte** Definition.

dynamische Bindung
in Lisp und einigen Skriptsprachen

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 302a

Ziele:

Grundbegriffe der statische Bindung verstehen

in der Vorlesung:

- Begriffe erläutern,
- Unterscheide: Im Programmtext steht die "Beschreibung einer Objektvariablen" (statisch). Bei der Programmausführung werden Objekte erzeugt, in deren Speicher jeweils eine Objektvariable zu der Beschreibung enthalten ist (dynamisch).

nachlesen:

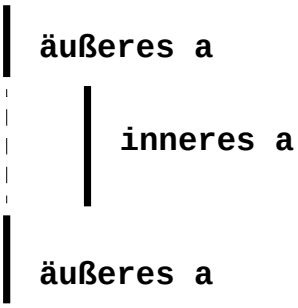
..., Abschnitt Kapitel 4 Bindung

Übungsaufgaben:

Gültigkeitsbereich

Der **Gültigkeitsbereich (scope)** einer Definition **D** für einen Bezeichner **b** ist der Programmabschnitt, in dem angewandte Auftreten von **b** an den in **D** definierten Programmgegenstand gebunden sind.

```
{ def a;
  def b;
  { def a;
    def c;
    use a;
  }
  use a;
}
```



In **qualifizierten Namen**, können Bezeichner auch außerhalb des Gültigkeitsbereiches ihrer Definition angewandt werden:

```
Thread.sleep(1000); max = super.MAX_THINGS;
```

sleep ist in der Klasse **Thread** definiert, **MAX_THINGS** in einer Oberklasse.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 303

Ziele:

Begriff Gültigkeitsbereich verstehen

in der Vorlesung:

- Erläuterung des Begriffs am Beispiel

Verdeckung von Definitionen

In Sprachen mit geschachtelten Programmstrukturen kann eine Definition eine andere für den gleichen Bezeichner **verdecken** (**hiding**).

Es gibt **2 unterschiedliche Grundregeln** dafür:

Algol-Verdeckungsregel (in Algol-60, Algol-68, Pascal, Modula-2, Ada, Java s. u.**):**

Eine Definition gilt im kleinsten sie umfassenden Abschnitt **überall**, ausgenommen darin enthaltene Abschnitte mit einer Definition für denselben Bezeichner.

oder operational formuliert:
Suche vom angewandten Auftreten eines Bezeichners **b** ausgehend nach außen den kleinsten umfassenden Abschnitt mit einer Definition für **b**.

C-Verdeckungsregel (in C, C++, Java**):**

Die Definition eines Bezeichners **b** gilt **von der Definitionsstelle** bis zum Ende des kleinsten sie umfassenden Abschnitts, **ausgenommen die Gültigkeitsbereiche von Definitionen für b** in darin enthaltenen Abschnitten.

Die **C-Regel** erzwingt definierendes vor angewandtem Auftreten.

Die **Algol-Regel** ist einfacher, toleranter und vermeidet Sonderregeln für notwendige Vorwärtsreferenzen.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 304

Ziele:

Verdeckungsprinzip verstehen

in der Vorlesung:

- Erläuterung der Verdeckungsregeln
- Verdeutlichung der Unterschiede
- Auswirkungen auf die Programmierung

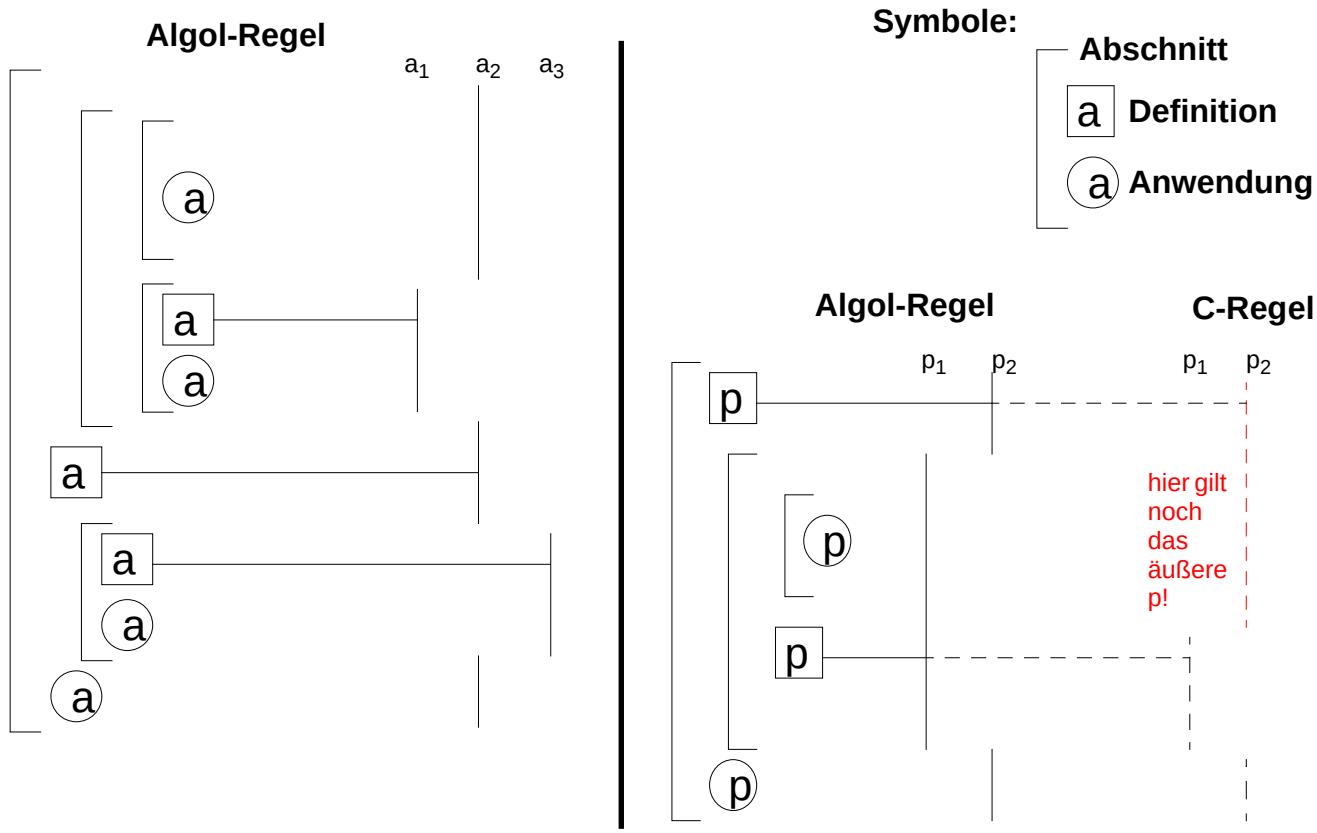
nachlesen:

..., Abschnitt Kapitel 4 Bindung

Verständnisfragen:

- Warum ist die Algol-Regel einfacher zu handhaben?
- Warum erfordern rekursive Definitionen von Funktionen oder Typen Ausnahmen von dem Zwang zur Definition vor der Anwendung?

Beispiele für Gültigkeitsbereiche



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 305

Ziele:

Gültigkeitsregeln an Beispielen zu Folie 304

in der Vorlesung:

Erläuterungen zusammen mit Folie 304

nachlesen:

..., Abschnitt Kapitel 4 Bindung

Übungsaufgaben:

- Geben Sie zu dem rechten Beispiel 2 Pascal-Programme an: `a.p` ist als Prozedur definiert, `b.p` ist als Pointer-Typ definiert.

Verständnisfragen:

- Kann das rechte Beispiel als korrektes Pascal-Programm verstanden werden?

Getrennte Namensräume

In manchen Sprachen werden die Bezeichner für Programmgegenstände bestimmter Art jeweils einem **Namensraum** zugeordnet

z. B. in **Java** jeweils ein Namensraum für

- Packages, Typen (Klassen und Interfaces), Variable (lokale Variable, Parameter, Objekt- und Klassenvariable), Methoden, Anweisungsmarken

Gültigkeits- und Verdeckungsregeln werden **nur innerhab eines Namensraumes** angewandt - nicht zwischen verschiedenen Namensräumen.

Zu welchem Namensraum ein Bezeichner gehört, kann am **syntaktischen Kontext** erkannt werden. (In Java mit einigen zusätzlichen Regeln)

Eine Klassendeklaration nur für Zwecke der Demonstration:

```
class Multi {
   Multi () { Multi = 5;}
   private int Multi;
   Multi Multi (Multi Multi) {
      if (Multi == null)
         return new Multi();
      else return Multi (new Multi ());
   }
}
```

Typ

Variable

Methode

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 306

Ziele:

Begriff Namensraum verstehen

in der Vorlesung:

- Erläuterung der Namensräume.

Gültigkeitsbereiche in Java

Package-Namen:
sichtbare Übersetzungseinheiten

Typnamen:
in der ganzen Übersetzungseinheit, Algol-60-Verdeckungsregel

Methodennamen:
umgebende Klasse, Algol-60-Verdeckungsregel, aber
Objektmethoden der Oberklassen werden überschrieben oder überladen - nicht verdeckt

Namen von **Objekt- und Klassenvariablen**:
umgebende Klasse, Algol-60-Verdeckungsregel,
Objekt- und Klassenvariable können Variable der Oberklassen verdecken

Parameter:
Methodenrumpf, (dürfen nur durch innere Klassen verdeckt werden)

Lokale Variable:
Rest des Blockes (bzw. bei Laufvariable in for-Schleife: Rest der for-Schleife),
C-Verdeckungsregel (dürfen nur durch innere Klassen verdeckt werden)

Terminologie in Java:
shadowing für *verdecken* bei Schachtelung, *hiding* für *verdecken* beim Erben

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 306a

Ziele:

Gültigkeitsregeln von Java kennenlernen

in der Vorlesung:

- Erläuterung der Besonderheiten in Java

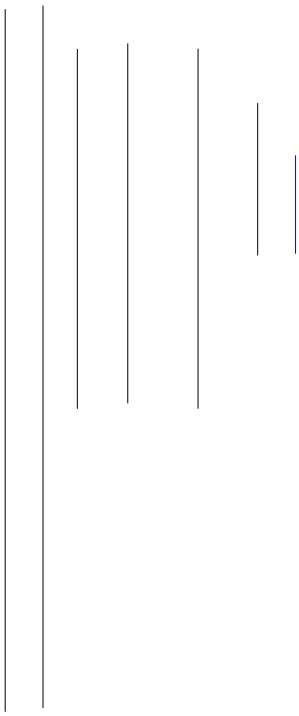
Beispiele für Gültigkeitsbereiche in Java

```
                                A B m mm cnt p f
class A
{
   void m (int p)
   {  cnt += 1;
      float f;
      ...
   }
   B mm ()
   { return new B();
   }
   int cnt = 42;
}

class B
{

   ...

}
```



```
class Ober
{  int k;
   ...
}

class Unter extends Ober
{  int k;
   void m ()
   {  k = 5;
   }
   void g (int p)
   {  int k = 7;
      k = 42;
      for (int i = 0;
            i<10; i++)
      {
         int k; // verboten
         ...
      }

   }
}
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 306aa

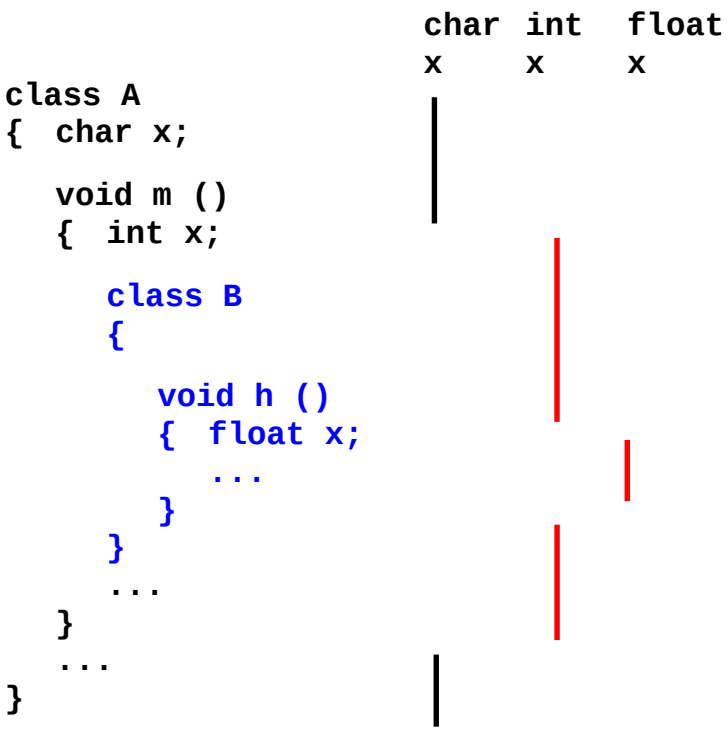
Ziele:

Beispiele zu Gültigkeitsregeln in Java

in der Vorlesung:

- Erläuterung der Gültigkeitsregeln

Innere Klassen in Java: Verdeckung von lokalen Variablen



Innere Klasse B:
Lokale Variable **`float x`** in **`h`** verdeckt
lokale Variable **`int x`** in **`m`** der äußeren Klasse

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 306ab

Ziele:

Beispiele zu Gültigkeitsregeln in Java

in der Vorlesung:

- Erläuterung der Gültigkeitsregeln

Gültigkeitsregeln in anderen Programmiersprachen

C, C++:
grundsätzlich gilt die **C-Regel**;
für Sprungmarken gilt die Algol-Regel.

```
void f () {
   ...
   goto finish;
   ...
finish: printf (...);
}
```

Pascal, Ada, Modula-2:
grundsätzlich gilt die **Algol-Regel.**
Aber eine **Zusatzregel** fordert:

Ein **angewandtes Auftreten** eines Bezeichners darf **nicht vor seiner Definition** stehen.

Davon gibt es dann in den Sprachen unterschiedliche **Ausnahmen**, um wechselweise rekursive Definitionen von Funktionen und Typen zu ermöglichen.

Pascal:
```
type ListPtr = ^ List;
     List =  record
                i: integer;
                n: ListPtr
             end;
```

C:
```
typedef struct _el *ListPtr;
typedef struct _el
   { int i; ListPtr n;} Elem;
```

Pascal:
```
procedure f (a:real) forward;

procedure g (b:real)
   begin ... f(3.5); ... end;

procedure f (a:real)
   begin ... g(7.5); ... end;
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 306b

Ziele:

Ausprägung von Gültigkeitsregeln kennenlernen

in der Vorlesung:

- Erläuterung der Regeln in Pascal.

Verständnisfragen:

- Logelei: Begründen Sie ausschließlich mit dem Text der Zusatzregel zum angewandten und definierenden Auftreten, dass in Pascal die Algol-Regel und nicht etwa die C-Regel gilt.

Zusammenfassung zum Kapitel 3

Mit den Vorlesungen und Übungen zu Kapitel 3 sollen Sie nun Folgendes können:

- Bindung von Bezeichnern verstehen
- Verdeckungsregeln für die Gültigkeit von Definitionen anwenden
- Grundbegriffe in den Gültigkeitsregeln von Programmiersprachen erkennen

Ziele:
Ziele des Kapitels erkennen

in der Vorlesung:
Erläuterungen dazu

4. Variable, Lebensdauer

Themen dieses Kapitels:

- Variablenbegriff und Zuweisung
- unterschiedliche Lebensdauer von Variablen
- Laufzeitkeller als Speicherstruktur für Variablen in Aufrufen

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 401

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

Variable in imperativen Sprachen

Variable: wird **im Programm beschrieben**, z. B. durch Deklaration **(statisch),**
wird **bei der Ausführung** im Speicher **erzeugt** und verwendet **(dynamisch),**
wird charakterisiert durch das Tripel **(Name, Speicherstelle, Wert).**

Einem **Namen im Programm** werden (bei der Ausführung) eine oder mehrere **Stellen im Speicher** zugeordnet.

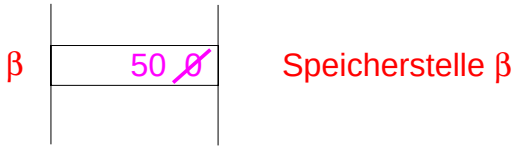
Das Ausführen von **Zuweisungen** ändert den **Wert der Variablen** (**Inhalt der Speicherstelle**). Bei der Ausführung eines imperativen Programms wird so der **Programmzustand** verändert.

Der Deklaration einer **globalen (static) Variable** ist genau eine Stelle zugeordnet.
Der Deklaration einer **lokalen Variablen einer Funktion** wird bei jeder Ausführung eines Aufrufes eine neue Stelle zugeordnet.

im Programm:

```
int betrag = 0;
...
betrag = 50;
```

im Speicher bei der Ausführung:



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 402

Ziele:

Variablenbegriff verstehen

in der Vorlesung:

- Erläuterungen zu Namen, Stelle, Wert

nachlesen:

..., Abschnitt 3, 3.1, 3.2

Veränderliche und unveränderliche Variable

In **imperativen Sprachen** kann der Wert einer Variablen grundsätzlich **durch Ausführen von Zuweisungen verändert** werden.

```
int betrag = 0;
...
betrag = 50;
```

In manchen **imperativen Sprachen**, wie Java, kann für bestimmte Variable **verboten** werden, nach ihrer Initialisierung an sie **zuzuweisen**.

```
final int hekto = 100;
```

In **funktionalen Sprachen** wird bei der Erzeugung einer **Variablen** ihr **Wert unveränderlich** festgelegt.

```
val sechzehn = (sqr 4);
```

In **mathematischen Formeln** wird ein **Wert unveränderlich an** den Namen einer **Variablen gebunden**. (Die Formel kann mit verschiedenen solchen Name-Wert-Bindungen ausgewertet werden.)

$\forall$ **x, y** $\in \Re$**: y = 2 * x - 1**

definiert eine Gerade im $\Re^2$

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 402a

Ziele:

Variablenbegriff verstehen

in der Vorlesung:

- Erläuterung der unterschiedlichen Variablenbegriffe.

nachlesen:

..., Abschnitt 3, 3.1, 3.2

Verständnisfragen:

Vergleichen Sie:

- In Java ist der Wert einer "Variable" mit der Eigenschaft *final* nicht veränderbar.
- In Ada kann man einen Zugriffsweg auf eine Variable auf lesenden Zugriff beschränken.
- In Pascal definiert "const hekto = 100;" einen Namen für einen Wert - nicht eine Variable!

Zuweisung

Zuweisung: **LinkeSeite = RechteSeite;**

Ausführen einer Zuweisung:

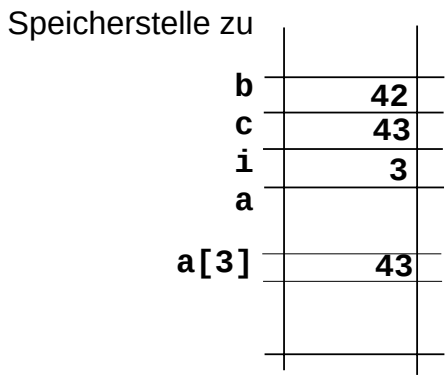
1. **Auswerten der linken Seite**;
   muss die **Stelle einer Variablen** liefern.
2. **Auswerten der rechten Seite**
   liefert einen **Wert**.
   I**n Ausdrücken** stehen **Namen von Variablen für ihren Wert**, d. h. es wird implizit eine **Inhaltsoperation** ausgeführt.
3. Der **Wert der Variablen** aus (1) **wird** durch den Wert aus (2) **ersetzt**.

Beispiel

im Programm:

```
b = 42;
c = b + 1;
i = 3;
a[i] = c;
```

im Speicher:



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 403

Ziele:

Zuweisungen verstehen

in der Vorlesung:

- Erläuterung der Begriffe und der Ausführung.

nachlesen:

..., Abschnitt 3, 3.1, 3.2

Stellen als Werte von Variablen

In objektorientierten Sprachen, wie Java oder C++, liefert die Ausführung von **`new C(...)`** die Stelle (Referenz) eines im Speicher erzeugten Objektes. Sie kann in Variablen gespeichert werden.

Java:

```
Circles  cir =
     new Circles(0, 1.0);
x = cir.getRadius();
```

C++:

```
Circles *cir =
     new Circles(0, 1.0);
x = cir->getRadius();
```

In C können Pointer-Variable Stellen als Werte haben (wie in C++). Die Ausführung von **`malloc (sizeof(Circles))`** liefert die Stelle (Referenz) eines im Speicher erzeugten Objektes.

C:

```
Circles *cir =
   malloc(sizeof(Circle));
cir->radius = 1.0;
```

Der Ausdruck **`&i`** liefert die Stelle der deklarierten Variable **`i`**, d. h. der **`&`**-Operator **unterdrückt die implizite Inhaltsoperation**. Der Ausdruck **`*i` bewirkt eine Inhalts-operation** - zusätzlich zu der impliziten.

```
int i = 5, j = 0;
int *p = &i;
j = *p + 1;
p = &i;
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 403a

Ziele:

Stellen als Werte von Variablen verstehen

in der Vorlesung:

- Erläuterung der Operationen

nachlesen:

..., Abschnitt 3, 3.1, 3.2

Lebensdauer von Variablen im Speicher

Lebensdauer: Zeit von der Bildung (Allokation) bis zur Vernichtung (Deallokation) des Speichers einer Variablen. Begriff der **dynamischen Semantik**!

Art der Variablen	Lebensdauer ist die Ausführung ...	Unterbringung im Speicher
globale Variable Klassenvariable	... des gesamten Programms	globaler Speicher
Parametervariable, lokale Variable	... eines Aufrufes	Laufzeitkeller
Objektvariable	... des Programms von der Erzeugung bis zur Vernichtung des Objekts	Halde, ggf. mit Speicher-bereinigung

Variable mit gleicher Lebensdauer werden zu **Speicherblöcken** zusammengefasst. (Bei Sprachen mit geschachtelten Funktionen kommen auch Funktionsrepräsentanten dazu.)

Speicherblock für

- Klassenvariable einer Klasse
- einen Aufruf mit den Parametervariablen und lokalen Variablen
- ein Objekt einer Klasse mit seinen Objektvariablen

Ziele:

Unterschiedliche Lebensdauern

in der Vorlesung:

Erläuterungen dazu, siehe SWE-40

nachlesen:

..., Abschnitt 3.4.1

Laufzeitkeller

Der **Laufzeitkeller** enthält für jeden noch nicht beendeten Aufruf einen Speicherblock (**Schachtel**, activation record) mit Speicher für Parametervariable und lokale Variable.
Bei **Aufruf wird eine Schachtel gekellert**, bei **Beenden des Aufrufes entkellert**.

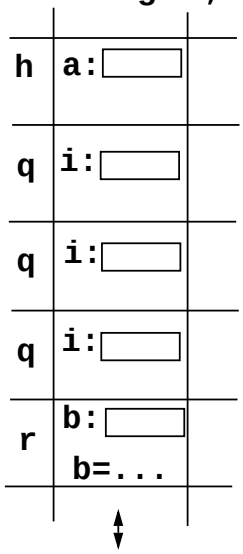
Programm mit Funktionen (Methoden)

```
h ┌──────────┐
  │ float a;  │
  │           │
  │ q();      │
  └──────────┘

q ┌──────────┐
  │ int i;    │
  │ if (..) q();
  │ r();      │
  └──────────┘

r ┌──────────┐
  │ int b;    │
  │           │
  │ b=...;    │
  └──────────┘
```

Laufzeitkeller bei der Aufruffolge h, q, q, q, r



kellern, entkellern

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 405

Ziele:
Das Speichermodell "Laufzeitkeller" verstehen

in der Vorlesung:
Erläuterung

- des Prinzips,
- des Beispiels.
- Bei rekursiven Aufrufen liegen mehrere Schachteln zur selben Funktion zugleich auf dem Laufzeitkeller.
- Die folgende PDF-Datei zeigt die Entwicklung des Laufzeitkellers

nachlesen:
..., Abschnitt 3.4.1

Übungsaufgaben:
- Geben Sie Programme an, deren Ausführung vorgegebene Laufzeitkeller erzeugt.

Laufzeitkeller bei geschachtelten Funktionen

Bei der Auswertung von Ausdrücken kann auf Variablen aus der **Umgebung** zugegriffen werden. Das sind die Speicherblöcke zu den Programmstrukturen, die den Ausdruck umfassen.

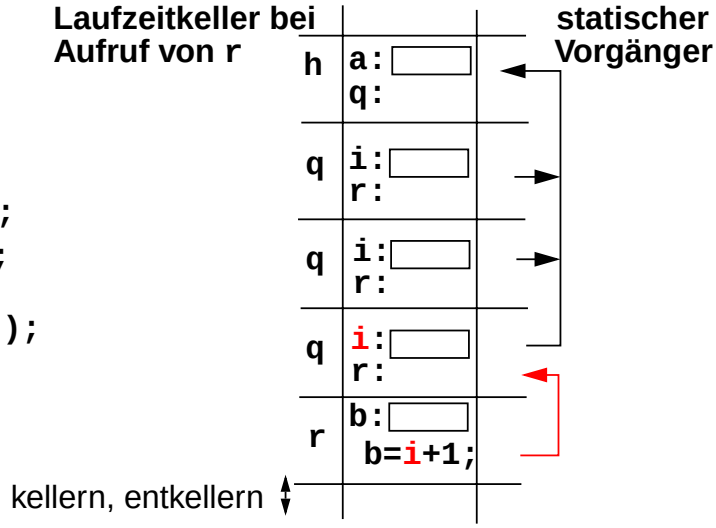
in Pascal, Modula-2, in funktionalen Sprachen: geschachtelte Funktionen
in Java: Methoden in Klassen, geschachtelte Klassen

Im **Laufzeitkeller** wird die **aktuelle Umgebung** repräsentiert durch die aktuelle Schachtel und die Schachteln entlang der Kette der **statischen Vorgänger**. Der statische Vorgänger zeigt auf die Schachtel, die die Definition der aufgerufenen Funktion enthält.

Programm mit geschachtelten Funktionen

```
h  float a;
     q  int i;
          r  int b;
             b=i+1;
          if(..) q();
          r();
   q();
```

Laufzeitkeller bei Aufruf von r



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 406

Ziele:

Laufzeitkeller für geschachtelte Funktionen verstehen

in der Vorlesung:

Erläuterung

- des Umgebungsbegriffs
- der Bedeutung der statischen Vorgänger
- des Beispiels.
- Jeder Schachtel zur Funktion q ist eine Definition von r zugeordnet. Sie sind zur Verdeutlichung in den Schachteln des Laufzeitkellerbildes eingezeichnet (r:), obwohl sie dort nicht wie Variable gespeichert sind. Ebenso ist die Zuweisung in der Schachtel zu r nur angegeben, um zu verdeutlichen, in welcher Umgebung sie ausgeführt wird.
- Die folgende PDF-Datei zeigt die Entwicklung des Laufzeitkellers

nachlesen:

..., Abschnitt 3.4.1

Übungsaufgaben:

- Geben Sie Programme mit geschachtelten Funktionen an, deren Ausführung vorgegebene Laufzeitkeller erzeugt.

Verständnisfragen:

Tüftelei: Ändern Sie in dem abgebildeten Laufzeitkeller, den statischen Vorgänger der Schachtel zum Aufruf von `r` auf die erste Schachtel von `q`. Wie müssen Sie das Programm modifizieren, damit es solch einen Keller erzeugt? Sie müssen die Funktion `r` als Parameter übergeben.

Zusammenfassung zum Kapitel 4

Mit den Vorlesungen und Übungen zu Kapitel 4 sollen Sie nun Folgendes verstanden haben:

- Variablenbegriff und Zuweisung
- Zusammenhang zwischen Lebensdauer von Variablen und ihrer Speicherung
- Prinzip des Laufzeitkellers
- Besonderheiten des Laufzeitkellers bei geschachtelten Funktionen

Ziele:
Ziele des Kapitels erkennen

in der Vorlesung:
Erläuterungen dazu

5. Datentypen

Themen dieses Kapitels:

5.1 Allgemeine Begriffe zu Datentypen

- Typbindung, Typumwandlung
- abstrakte Definition von Typen
- parametrisierte und generische Typen

5.2 Datentypen in Programmiersprachen

- einfache Typen, Verbunde, Vereinigungstypen, Reihungen
- Funktionen, Mengen, Stellen

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

5.1 Allgemeine Begriffe zu Typen

Typ: Wertemenge mit darauf definierten Operationen
z. B. **`int`** in Java: Werte von **`Integer.MIN_VALUE`** bis **`Integer.MAX_VALUE`**
mit arithmetischen Operationen für ganze Zahlen

Typ als Eigenschaft von

Literal:	Notation für einen Wert des Typs,
Variable:	speichert einen Wert des Typs,
Parameter:	übergibt einen Wert des Typs an den Aufruf,
Ausdruck:	Auswertung liefert einen Wert des Typs,
Aufruf:	Auswertung liefert einen Wert des Typs,
Funktion, Operator:	Signatur (Parameter- und Ergebnistypen)

Typen werden **in der Sprache definiert**:
z. B. in C: **`int`**, **`float`**, **`char`**, ...

Typen können **in Programmen definiert** werden:
Typdefinition bindet die Beschreibung eines Typs an einen Namen,
z. B. in Pascal:

```
type Datum = record tag, monat, jahr: integer; end;
```

Typprüfung (type checking):
stellt sicher, dass jede Operation mit Werten des dafür festgelegten Typs ausgeführt wird,
Typsicherheit

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 502

Ziele:

Typbegriff in Programmiersprachen

in der Vorlesung:

Erläuterung der Begriffe und weitere Beispiele

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Verständnisfragen:

Geben Sie Typdefinitionen in Java und in C an.

Statische oder dynamische Typbindung

Statische Typbindung:

Die **Typen** von Programmgegenständen (z.B. Variable, Funktionen) und Programmkonstrukten (z. B. Ausdrücke, Aufrufe) werden **durch den Programmtext festgelegt**.

z. B. in Java, Pascal, C, C++, Ada, Modula-2 **explizit durch Deklarationen**
z. B. in SML, Haskell **implizit durch Typinferenz** (siehe GPS-7.4 ff)

Typprüfung im Wesentlichen zur Übersetzungszeit.
Entwickler muss erkannte Typfehler beheben.

Dynamische Typbindung:

Die **Typen** der Programmgegenstände und Programmkonstrukte werden erst **bei der Ausführung bestimmt**. Sie können bei der Ausführung nacheinander Werte unterschiedlichen Typs haben.

z. B. Smalltalk, PHP, JavaScript und andere Skriptsprachen

Typprüfung erst zur Laufzeit.
Evtl. werden Typfehler erst beim Anwender erkannt.

Keine Typisierung:

In den Regeln der Sprache wird der **Typbegriff nicht verwendet**.

z. B. Prolog, Lisp

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 502a

Ziele:

Zeitpunkt der Typbindung unterscheiden

in der Vorlesung:

Erläuterung der Begriffe

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Verständnisfragen:

Die Verwendung von Programmiersprachen mit dynamischer Typbindung wird zuweilen als "teuer" bezeichnet. Finden Sie Argumente für diese Ansicht.

Beispiele für statische Typregeln

1. Eine **Variable mit Typ T** kann nur einen Wert aus der Wertemenge von T speichern.
   ```
   float x; ... x = r * 3.14;
   ```
2. Der **Ausdruck einer return-Anweisung** muss einen Wert liefern, der aus der Wertemenge des **Ergebnistyps** der umgebenden Funktion ist (oder in einen solchen Wert konvertiert werden kann (siehe GPS-5.4)).
   ```
   float sqr (int i) {return i * i;}
   ```
3. Im **Aufruf einer Funktion** muss die Zahl der Parameterausdrücke mit der Zahl der formalen Parameter der Funktionsdefinition übereinstimmen und jeder **Parameterausdruck** muss einen Wert liefern, der aus der Wertemenge des T**yps des zugehörigen formalen Parameters** ist (oder ... s.o.)).
4. Zwei Methoden, die in einer Klasse deklariert sind und **denselben Namen** haben, **überladen** einander, wenn sie in einigen **Parameterpositionen unterschiedliche Typen** haben. Z. B.
   ```
   int add (int a, int b) { return a + b; }
   Vector<Integer> add (Vector<Integer> a, Vector<Integer> b) {...}
   ```

   In einem Aufruf einer überladenen Methode wird anhand der Typen der Parameterausdrücke entschieden, welche Methode aufgerufen wird:
   ```
   int k; ... k = add (k, 42);
   ```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 502b

Ziele:

Typregeln an Beispielen kennenlernen

in der Vorlesung:

Erläuterung der Regeln

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Verständnisfragen:

- Geben Sie Programmbeispiele an, die jeweils eine Regel verletzen.
- Formulieren Sie weitere solche Regeln.

Streng typisiert

Streng typisierte Sprachen (strongly typed languages):

Die Einhaltung der **Typregeln** der Sprache stellt sicher, dass **jede Operation** nur mit **Werten des dafür vorgesehenen Typs** ausgeführt wird.

Jede Verletzung einer Typregel wird erkannt und als Typfehler gemeldet - zur Übersetzungszeit oder zur Laufzeit.

FORTRAN	nicht streng typisiert	Parameter werden nicht geprüft
Pascal	nicht ganz streng typisiert	Typ-Uminterpretation in Variant-Records
C, C++	nicht ganz streng typisiert	es gibt undiscriminated Union-Types
Ada	nicht ganz streng typisiert	es gibt Direktiven, welche die Typprüfung an bestimmten Stellen ausschalten
Java	streng typisiert	alle Typfehler werden entdeckt, zum Teil erst zur Laufzeit

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 503

Ziele:

Strenge Typregeln

in der Vorlesung:

Erläuterung der Begriffe

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Verständnisfragen:

- Warum ist in Pascal der Zugriff auf Verbundvarianten unsicher?
- Konstruieren Sie eine unsichere Verwendung von Vereinigungstypen in C.

Typumwandlung (Konversion)

Typumwandlung, Konversion (conversion):
Der Wert eines Typs wird in einen entsprechenden Wert eines anderen Typs umgewandelt.

ausweitende Konversion:
jeder Wert ist im Zieltyp ohne Informationsverlust darstellbar, z. B.

```
float --> double
```

einengende Konversion:
nicht jeder Wert ist im Zieltyp darstellbar, ggf. Laufzeitfehler, z. B.

```
float --> int
```
(Runden, Abschneiden oder Überlauf)

Uminterpretation ist unsicher, ist nicht Konversion!:
Das Bitmuster eines Wertes wird als Wert eines anderen Typs interpretiert.
z. B. Varianten-Records in Pascal (GPS-5.14)

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 504

Ziele:

Konversionen verstehen

in der Vorlesung:

Erläuterung der Begriffe

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Verständnisfragen:

- Klassifizieren Sie Beispiele von casts in Java und C als sicher oder unsicher.

Explizite und implizite Typumwandlung

Eine Konversion kann **explizit im Programm als Operation** angegeben werden (**type cast**), z. B.

```
float x = 3.1; int i = (int) x;
```

Eine Konversion kann **implizit vom Übersetzer eingefügt** werden (**coercion**), weil der Kontext es erfordert, z. B.

```
double d = 3.1;        implizit float --> double
d = d + 1;             implizit int --> double
```

Java: **ausweitende** Konversionen für Grund- und Referenztypen **implizit**,
einengende müssen **explizit** angegeben werden.

Konversion für Referenzen ändert weder die Referenz noch das Objekt:

```
Object val = new Integer (42); implizit Integer --> Object
Integer ival = (Integer) val;  explizit Object --> Integer
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 504a

Ziele:

implizite und explizite Konversion verstehen

in der Vorlesung:

Erläuterung der Begriffe

nachlesen:

..., Abschnitt 2.1, 2.5.1, 7

Abstrakte Definition von Typen

Datenstrukturen werden in Programmen mit Typen modelliert => Modellierungskonzepte

Abstrakte Grundkonzepte zur Bildung einfacher und zusammengesetzter Wertemengen **D**:
(Hier: nur Wertemengen der Typen; Operationen darauf werden davon nicht erfasst.)

1. einfache Mengen: $D = \{ e_1, e_2, \ldots, e_n \}$ extensionale Aufzählung der Elemente

$D = \{a \mid \text{Eigenschaft von } a\}$ intensionale Definition

z. B. Grundtypen, Aufzählungstypen, Auschnittstypen

2. kartesisches Produkt: $D = D_1 \times D_2 \times \ldots \times D_n$

Tupel z. B. Verbunde (records); Reihungen (arrays) (mit gleichen `Di`)

3. Vereinigung: $D = D_1 \mid D_2 \mid \ldots \mid D_n$

Alternativen zusammenfassen
z. B. union in C und Algol 68, Verbund-Varianten in Pascal, Ober-, Unterklassen

4. Funktion: `D = Dp -> De`

Funktionen als Werte des Wertebreiches **D**
z. B. Funktionen, Prozeduren, Methoden, Operatoren; auch Reihungen (Arrays)

5. Potenzmenge: $D = P(De)$

z. B. Mengentypen in Pascal

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 505

Ziele:

Abstraktionen verstehen

in der Vorlesung:

- Wiederholung aus der Vorlesung "Modellierung"
- Erläuterung der Konzepte

nachlesen:

..., Abschnitt 2.2, 2.3, 2.4

Übungsaufgaben:

- Definieren Sie Wertemengen zu gegebenen Beschreibungen

Abstrakte Definition von Typen: Beispiele

1. einfache Mengen: `Farbe = {blau,gelb,rot}`

Pascal:
```
type Farbe = (blau,gelb,rot);
```

C:
```
typedef enum {blau,gelb,rot} Farbe;
```

2. kartesisches Produkt: `Graph = Knoten × Kanten`

Pascal:
```
type Graph = record n: Knoten; v: Kanten end;
```

C:
```
typedef struct { Knoten n; Kanten v; } Graph;
```

3. Vereinigung: `Geo = Kreis | Rechteck`

Pascal:
```
type Geo = record case boolean of
                    false: (k: Kreis);
                    true: (r: Rechteck)
           end;
```

C:
```
typedef union {Kreis k; Rechteck r} Geo;
```

4. Funktion: `IntOprSig = int × int -> int`

Pascal: Funktionen nicht allgemein als Daten, nur als Parameter ohne Angabe der Signatur

C:
```
typedef int IntOprSig(int,int);
```

5. Potenzmenge: `Mischfarbe = P ( Farbe )`

Pascal:
```
type Mischfarbe = set of farbe;
```

C:
```
typedef unsigned Mischfarbe;
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 507

Ziele:

Erste Beispiele zu den Grundkonzepten

in der Vorlesung:

- Erläuterung der Beispiele und der Notation
- Vertiefung auf späteren Folien

nachlesen:

..., Abschnitt 2.2, 2.3, 2.4

Kombination von Typen

Die Grundkonzepte zur Typkonstruktion sind prinzipiell **beliebig kombinierbar**,
z. B. Kreise oder Rechtecke zusammengefasst zu 2-dimensionalen geometrischen Figuren:

```
Koord2D = float × float
Form = {istKreis, istRechteck}
Figur = Koord2D ×    Form ×    (float | float ×float)
```

Position Kennzeichen Radius Kantenlängen

z. B. Signatur einer Funktion zur Berechnung von Nullstellen einer als Parameter gegebenen Funktion:

```
(float –> float) ×float × float -> P (float)
```

Funktion Bereich Menge der Nullstellen

Ziele:

Abstrakte Typkonzepte anwenden

in der Vorlesung:

- Erläuterungen zu den Definitionen

Rekursive Definition von Typen

Wertemengen können auch **rekursiv definiert** werden:
z. B. ein Typ für **lineare Listen** rekursiv definiert durch Paare:

```
IntList = int × IntList | {nil}
```

{nil} ist eine einelementige Wertemenge. **nil** repräsentiert hier die leere Liste.

Werte des Typs sind z. B.

```
nil, (1,nil), (2,nil),...,(1,(1,nil)), (8,(9,(4,nil))), ...
```

Entsprechend für Bäume:

```
IntTree = IntTree ×int ×IntTree | {TreeNil}
```

Eine rekursive Typdefinition ohne nicht-rekursive Alternative ist so nicht sinnvoll, da keine Werte gebildet werden können:

```
X = int × X
```

In funktionalen Sprachen können Typen direkt so rekursiv definiert werden, z. B. in SML:

```
datatype IntList = cons of (int × IntList) | IntNil;
```

In imperativen Sprachen werden rekursive Typen mit Verbunden (struct) implementiert, die Verbundkomponenten mit Stellen als Werte (Pointer) enthalten, z. B. in C:

```
typedef struct _IntElem  *IntList;
typedef struct _IntElem { int head; IntList tail;} IntElem;
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 508a

Ziele:

Abstrakte Typkonzepte anwenden

in der Vorlesung:

- Erläuterungen zu rekursiven Definitionen
- Lineare Liste, abstrakt: Paar von Wert und Liste; implementiert: Paar von Wert und Referenz auf Liste.
- Zusammenhang zur Implementierung in imperativen Sprachen
- weitere Beispiele

Verständnisfragen:

- Warum wäre die abstrakte Definition des Typs `IntList` ohne die `nil`-Alternative sinnlos?

Parametrisierte Typen

Parametrisierte Typen (Polytypen):
Typangaben mit **formalen Parametern, die für Typen** stehen.

Man erhält aus einem Polytyp einen konkreten Typ durch **konsistentes Einsetzen eines beliebigen Typs** für jeden Typparameter.

Ein Polytyp beschreibt die **Typabstraktion**, die allen daraus erzeugbaren konkreten Typen gemeinsam ist.

Beispiele in SML-Notation mit **`'a, 'b`**, ... für Typparameter:

Polytyp	gemeinsame Eigenschaften	konkrete Typen dazu
`'a` × `'b`	Paar mit Komponenten **beliebigen** Typs	**`int` × `float`** **`int` × `int`**
`'a` × `'a`	Paar mit Komponenten **gleichen** Typs	**`int` × `int`** **`(int->float)` × `(int->float)`**
**`'a list = 'a` × `'a list	{nil}`**	homogene, lineare Listen	**`int list`** **`float list`** **`(int` × `int) list`**

Verwendung z. B. in **Typabstraktionen** und in **polymorphen Funktionen** (GPS-5-9a)
In SML werden konkrete Typen zu parametrisierten Typen statisch bestimmt und geprüft.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 509

Ziele:

Parametrisierte Typen verstehen

in der Vorlesung:

- Wichtiges Prinzip erläutern
- Vorkommen in funktionalen Sprachen zeigen

nachlesen:

..., Abschnitt 7.1, 7.3, 7.4

Verständnisfragen:

- Geben Sie weitere Beispiele zu Polytypen an

Polymorphe Funktionen

(Parametrisch) **polymorphe Funktion**:
eine Funktion, deren **Signatur ein Polytyp** ist, d. h. Typparameter enthält.

Die Funktion ist auf Werte eines jeden konkreten Typs zu der Signatur anwendbar.
D. h. sie muss unabhängig von den einzusetzenden Typen sein;

Beispiele:
eine Funktion, die die Länge einer beliebigen homogenen Liste bestimmt:

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

polymorphe Signatur: `'a list -> int`

Aufrufe: `length ([1, 2, 3]); length ([(1, true), (2, true)]);`

eine Funktion, die aus einer Liste durch elementweise Abbildung eine neue Liste erzeugt:

```
fun map (f, l) = ...
```

polymorphe Signatur: `(('a -> 'b) × 'a list) -> 'b list`

Aufruf: `map (even, [1, 2, 3])` liefert `[false, true, false]`

`int->bool,int list` `bool list`

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 509a

Ziele:

Polymorphe Signaturen verstehen

in der Vorlesung:

- Vorkommen in funktionalen Sprachen zeigen
- Weitere Beispiele zu polymorphen Funktionen

nachlesen:

..., Abschnitt 7.1, 7.3, 7.4

Verständnisfragen:

- Geben Sie weitere Beispiele zu polymorphen Funktionen an.

Generische Definitionen

Eine **Generische Definition** hat **formale generische Parameter.**
Sie ist eine **abstrakte Definition einer Klasse** oder eines Interfaces.
Für jeden generischen Parameter kann ein **Typ eingesetzt** werden.
(Er kann auf Untertypen eines angegebenen Typs eingeschränkt werden.)

Beispiel in Java:
Generische Definition einer Klasse **`Stack`** mit generischem Parameter für den Elementtyp

```
class Stack<Elem>
{  private Elem [] store ;
   void push (Elem el) {... store[top]= el;...}
   ...
};
```

Eine **generische Definition wird instanziiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht zur Übersetzungszeit eine Klassendefinition. Z. B.

```
Stack<Float> taschenRechner = new Stack<Float>();
Stack<Frame> windowMgr = new Stack<Frame>();
```

Generische Instanziierung kann im Prinzip durch **Textersetzung** erklärt werden: Kopieren der generischen Definition mit Einsetzen der generischen Parameter im Programmtext.

Der Java-Übersetzer erzeugt für jede generische Definition eine Klasse im ByteCode, in der **`Object`** für die generischen Typparameter verwendet wird. Er setzt Laufzeitprüfungen ein, um zu prüfen, dass die ursprünglich generischen Typen korrekt verwendet wurden.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 510

Ziele:

Generische Parametrisierung verstehen

in der Vorlesung:

Erläuterung

- des Prinzips
- der Notation
- der Sicherheit von ByteCode-Programmen.

nachlesen:

..., Abschnitt 6.4

Generische Definitionen in C++

Generische Definitionen wurden in Ada und C++ schon früher als in Java eingeführt. Außer Klassen können auch Module (Ada) und Funktionen generisch definiert werden. **Formale generische Parameter** stehen für beliebige Typen, Funktionen oder Konstante. (Einschränkungen können nicht formuliert werden.)

Beispiel in C++:
Generische Definition einer Klasse **`Stack`** mit generischem Parameter für den Elementtyp

```
template <class Elem>
   class Stack
   {  private Elem store [size];
      void push (Elem el) {... store[top]=el;...}
      ...
   };
```

Eine **generische Definition wird instanziiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht Übersetzungszeit eine Klassen-, Modul- oder Funktionsdefinition.

```
Stack<float>* taschenRechner = new Stack<float>();
Stack<Frame>* windowMgr = new Stack<Frame>();
```

Auch **Grundtypen** wie **`int`** und **`float`** können als aktuelle generische Parameter eingesetzt werden.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 510j

Ziele:

Generische Parametrisierung in Java

in der Vorlesung:

Erläuterung der Notation

nachlesen:

..., Abschnitt 6.4

Verständnisfragen:

Warum ist es fragwürdig, ein Sortierschema mit zwei generischen Parametern für den Elementtyp und die Vergleichsfunktion zu definieren?

Nutzen generischer Definitionen

Typische Anwendungen:

homogene Behälter-Typen, d. h. alle Elemente haben denselben Typ:
Liste, Keller, Schlange, ...
generischer Parameter ist der Elementtyp (und ggf. die Kapazität des Behälters)

Algorithmen-Schemata: Sortieren, Suchen, etc.
generischer Parameter ist der Elementtyp mit Vergleichsfunktion

Generik sichert statische Typisierung trotz verschiedener Typen der Instanzen!
Übersetzer kann Typkonsistenz garantieren, z. B. Homogenität der Behälter

Java hat **generische Definitionen** erst seit Version 1.5
Behälter-Typen programmierte man vorher mit **`Object`** als Elementtyp,
dabei ist **Homogenität nicht garantiert**

Generische Definitionen gibt es z. B. in C++, Ada, Eiffel, Java ab 1.5

Generische Definitionen sind **überflüssig in dynamisch typisierten Sprachen** wie Smalltalk

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 510n

Ziele:

Generische Parametrisierung verstehen

in der Vorlesung:

- Vergleich mit Java-Lösungen ohne Generik

nachlesen:

..., Abschnitt 6.4

5.2 Datentypen in Programmiersprachen
Typen mit einfachen Wertemengen (1)

a. Ausschnitte aus den **ganzen Zahlen** mit arithmetischen Operationen
unterschiedlich große Ausschnitte: Java: **`byte`**, **`short`**, **`int`**, **`long`**;
C, C++: **`short, int, long int, unsigned`**; Modula-2: **`INTEGER`** und **`CARDINAL`**

b. **Wahrheitswerte** mit logischen Operationen
Pascal, Java: **`boolean`** = (**`false, true`**);
in C: durch **`int`** repräsentiert; **0** repräsentiert **false**, alle anderen Werte **true**

Kurzauswertung logischer Operatoren in C, C++, Java, Ada:
Operanden von links nach rechts auswerten bis das Ergebnis feststeht:

```
a && b || c     i >= 0 && a[i] != x
```

c. **Zeichen eines Zeichensatzes** mit Vergleichen, z. B. **`char`**

d. **Aufzählungstypen** (enumeration)

Pascal:	**`Farbe = (rot, blau, gelb)`**
C:	**`typedef enum {rot, blau, gelb} Farbe;`**
Java:	**`enum farbe {rot, blau, gelb}`**

Die Typen (a) bis (d) werden auf ganze Zahlen abgebildet (ordinal types) und können deshalb auch exakt verglichen, zur Indizierung und in Fallunterscheidungen verwendet werden.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 511

Ziele:

Arten von Grundtypen kennenlernen

in der Vorlesung:

Erläuterungen und Beispiele dazu

nachlesen:

..., Abschnitt 2.2

Verständnisfragen:

- Was bedeutet "Kurzauswertung logischer Operatoren" in Java, C, C++ und Ada ? Welchen Nutzen haben Programmierer davon?

Typen mit einfachen Wertemengen (2)

e. Teilmenge der **rationalen Zahlen** in Gleitpunkt-Darstellung (floating point), z. B. `float`, mit arithmetischen Operationen,
Gleitpunkt-Darstellung:
Tripel (s, m, e) mit Vorzeichen s, Mantisse m, Exponent e zur Basis b = 2;
Wert der Gleitpunktzahl: $x = s * m * b^e$

f. Teilmenge der **komplexen Zahlen** mit arithmetischen Operationen z. B. in FORTRAN

g. **Ausschnittstypen** (subrange)
in Pascal aus (a) bis (d): `Range = 1..100`;
in Ada auch aus (e) mit Größen- und Genauigkeitsangaben

Zur Notation von Werten der Grundtypen sind **Literale** definiert:
z. B. `127, true, '?', 3.71E-5`

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 511a

Ziele:

Arten von Grundtypen kennenlernen

in der Vorlesung:

Erläuterungen und Beispiele dazu

nachlesen:

..., Abschnitt 2.2

Verständnisfragen:

- Ist die Typausweitung von `int` nach `float` immer werterhaltend?

Verbunde

Kartesisches Produkt: $D = D_1 \times D_2 \times ... \times D_n$ mit beliebigen Typen D_i; **n-Tupel**

Verbundtypen in verschiedenen Sprachen:

SML: `type Datum = int * int * int;`

Pascal, Modula-2, Ada:
```
type Datum = record tag, monat, jahr: integer; end;
```

C, C++: `typedef struct {int tag, monat, jahr;} Datum;`

Selektoren zur Benennung von **Verbundkomponenten:**
```
Datum heute = {27, 6, 2006};
heute.monat oder  monat of heute
```

Operationen:
meist nur Zuweisung; komponentenweise Vergleiche (SML) sehr aufwändig

Notation für Verbundwerte:
in **Algol-68**, **SML**, **Ada** als Tupel**:** `heute := (27, 6, 2006);`
in **C** nur für Initialisierungen: `Datum heute = {27, 6, 2006};`
in **Pascal**, **Modula-2 keine** Notation für Verbundwerte
sehr lästig, da Hilfsvariable und komponentenweise Zuweisungen benötigt werden
```
Datum d; d.tag:=27; d.monat:=6; d.Jahr:=2006; pruefeDatum (d);
```
statt `pruefeDatum ((27, 6, 2006));`

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 512

Ziele:

Verbundtypen in verschiedenen Sprachen

in der Vorlesung:

Erläuterungen zu

- Tupelbildung
- Selektion
- Notation

nachlesen:

..., Abschnitt 2.3.1

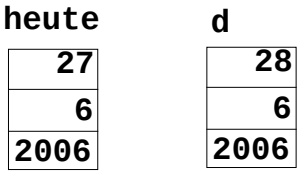
Verständnisfragen:

- Erläutern Sie anhand eines Funktionsaufrufes mit Verbundparameter: Mit Notationen für Verbundwerte kann man den Aufruf geschlossen angeben. Ohne solche Notationen muss man Variable und Zuweisungen zusätzlich verwenden.

Vergleich: Verbundwerte - Objekte

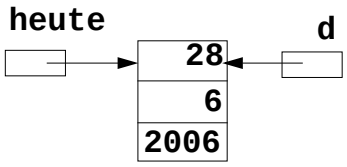
Verbundtypen in C, C++:

```
typedef struct {int tag, monat, jahr;} Datum;
Datum heute = {27, 6, 2006};
Datum d = heute; d.tag += 1;
```



Klassen in objekt-orientierten Sprachen wie Java, C++:

```
class Datum {int tag, monat, jahr;}
Datum heute = new Datum (27, 6, 2006);
Datum d = heute; d.tag += 1;
```



Vergleich

Werte von Typen	**Objekte von Klassen**
habe **keine Identität**	haben **Identität** (**Referenz**, Speicherstelle)
werden z. B. **in Variablen gespeichert** Werte haben **keinen veränderlichen Zustand**	haben **eigenen Speicher** können **veränderlichen Zustand** haben
beliebig **kopierbar**	werden **nicht kopiert, sondern geklont**
2 Werte sind gleich, wenn ihre Komponenten gleich sind, auch wenn die Werte an verschiedenen Stellen gespeichert sind	**2 Objekte sind immer verschieden**, auch wenn ihre Instanzvariablen paarweise gleiche Werte haben.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 513

Ziele:

Unterschied zwischen Werten und Objekten verstehen

in der Vorlesung:

- Wiederholung aus GP 1,
- Erläuterungen dazu.

nachlesen:

..., Abschnitt 2.3.1

Verständnisfragen:

- Was ist der wesentliche semantische Unterschied zwischen Verbundentypen in ML, Pascal oder C einerseits und Klassen in Java, C++ andererseits?

Vereinigung (undiscriminated union)

Allgemeines Konzept: **Vereinigung von Wertebereichen: $D = D_1 \mid D_2 \mid ... \mid D_n$**
Ein Wert vom Typ D_i ist auch ein Wert vom allgemeineren Typ D.
Variable vom Typ D können einen Wert jedes der vereinigten Typen D_i aufnehmen.

Problem: Welche Operationen sind auf den Inhalt solch einer Variable sicher anwendbar?

1. undiscriminated union: $D = D_1 \mid D_2 \mid ... \mid D_n$

z. B. zwei Varianten der Darstellung von Kalenderdaten, als Tripel vom Typ `Datum` oder als Nummer des Tages bezogen auf einen Referenztag, z. B.

union-Typ in C:

```
typedef union {Datum KalTag; int TagNr;} uDaten;
uDaten h;
```

Varianten-Record in Pascal:

```
type uDaten = record case boolean of
                true:  (KalTag: Datum);
                false: (TagNr: integer);
              end;
var h: uDAten;
```

Durch den **Zugriff** wird ein Wert vom Typ D als Wert vom Typ D_i interpretiert; unsicher!
z. B. **`h.TagNr = 4342;`** oder **`t = h.KalTag.tag;`**

Speicher wird für die größte Alternative angelegt und für kleinere Alternativen ggf. nicht genutzt.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 514

Ziele:

Unsichere Vereinigung von Wertebereichen

in der Vorlesung:

- alternative Nutzung des Speichers
- unsichere Zugriffe

nachlesen:

..., Abschnitt 2.3.2

Verständnisfragen:

- Warum ist die *undiscriminated union* in Pascal und C unsicher?

Vereinigung (discriminated union)

Allgemeines Konzept: **Vereinigung von Wertebereichen: $D = D_1 \mid D_2 \mid ... \mid D_n$** (wie auf 5.14)
Problem: Welche Operationen sind auf den Inhalt solch einer Variable sicher anwendbar?

2. discriminated union: $D = T \times (D_1 \mid D_2 \mid ... \mid D_n)$ mit $T = \{t_1, t_2, ... , t_n\}$
Unterscheidungskomponente vom Typ T (**tag field**) ist Teil des Wertes und kennzeichnet **Zugehörigkeit zu einem D_i**; z. B.

SML (implizite Unterscheidungskomponente)**:**

```
datatype Daten = KalTag of Datum | TagNr of int;
```

Pascal, Modula-2, Ada (explizite Unterscheidungskomponente)**:**

```
type uDaten = record case IstKalTag: boolean of
  true:  (KalTag: Datum);
  false: (TagNr: integer);
end;
```

Sichere Zugriffe durch Prüfung des Wertes der Unterscheidungskomponente oder Fallunterscheidung darüber.

Gleiches Prinzip in objekt-orientierten Sprachen (implizite Unterscheidungskomponente):
allgemeine Oberklasse mit speziellen Unterklassen

```
class Daten { ... }
class Datum extends Daten {int tag, monat, jahr;}
class TagNum extends Daten {int TagNr;}
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 514a

Ziele:

Sichere Vereinigung von Wertebereichen

in der Vorlesung:

- Dynamische Typkennung von Werten
- Bezug zu Vererbung in OO-Sprachen

nachlesen:

..., Abschnitt 2.3.2

Übungsaufgaben:

- Entwerfen Sie ein Sprachkonstrukt zur sicheren Fallunterscheidung über vereinigte Typen. Beispiel angeben und Semantik skizzieren.

Verständnisfragen:

- Warum ist auch die *discriminated union* in Pascal unsicher?

Reihungen (Arrays)

Abbildung des Indextyps auf den Elementtyp: D = I -> E
oder kartesisches Produkt mit fester Anzahl Komponenten $D = E \times E \times ... \times E$

in **Pascal**-Notation: `type D = array [ I ] of E`

Indexgrenzen, alternative Konzepte:

statische Eigenschaft des Typs (**Pascal**):	`array [0..9] of integer;`
statische Eigenschaft der Reihungsvariablen (**C**):	`int a[10];`
dynamische Eigenschaft des Typs (**Ada**):	`array (0..m*n) of float;`
dynamisch, bei Bildung von Werten, Objekten (**Java**)	`int[] a = new int[m*n];`

Mehrstufige Reihungen: Elementtyp ist Reihungstyp:
`array [ I1 ] of array [ I2 ] of E` kurz: `array [ I1, I2 ] of E`
zeilenweise Zusammenfassung in fast allen Sprachen; nur in FORTRAN spaltenweise

Operationen:
Zuweisung, Indizierung als Zugriffsfunktion: `x[i]` `y[i][j]` `y[i,j]`
in C, C++, FORTRAN ohne Prüfung des Index gegen die Grenzen

Notation für Reihungswerte in Ausdrücken: (fehlen in vielen Sprachen; vgl. Verbunde)

Algol-68:
```
a := (2, 0, 0, 3, 0, 0);
```
Ada:
```
a := ( 2 | 4 => 3, others => 0);
```
C:
```
int a[6] = {2, 0, 0, 3, 0, 0};
```
nur in Initialisierungen

Java:
```
int[] a = {2, 0, 0, 3, 0, 0};
a = new int [] {2, 0, 0, 3, 0, 0};
```
Pascal: keine

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 515

Ziele:

Reihungen in verschiedenen Sprachen

in der Vorlesung:

Erläuterung

- der Bestimmung der Indexgrenzen,
- der Notation von Reihungswerten.

nachlesen:

..., Abschnitt 2.3.3

Verständnisfragen:

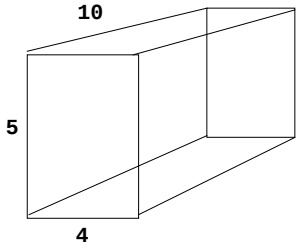
- Auch wenn Indexgrenzen erst dynamisch bestimmt werden, sind sichere Zugriffe möglich. Wie?

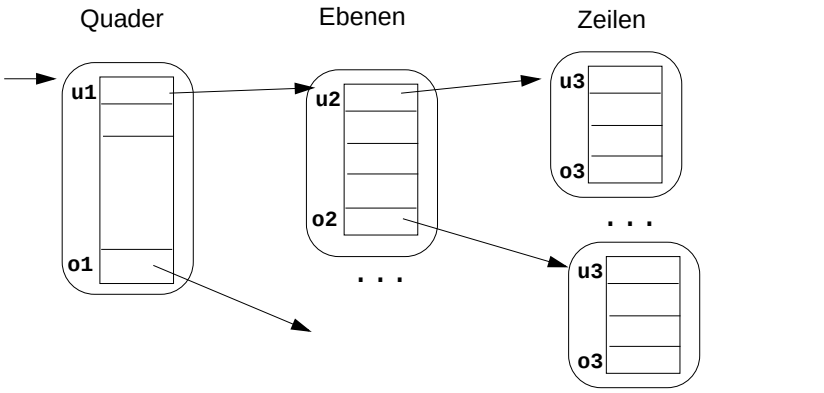
Speicherung von Arrays durch Pointer-Bäume

Ein n-dimensionales Array mit explizit gegebenen Unter- und Obergrenzen (Pascal-Notation):

```
a: array[u1..o1, u2..o2, ..., un..on] of real;
```

wird z. B. in **Java** als **Baum von linearen Arrays** gespeichert
n-1 Ebenen von Pointer-Arrays und Daten Arrays auf der n-ten Ebene





Jedes einzelne Array kann separat, dynamisch, gestreut im Speicher angelegt werden; nicht alle Teil-Arrays müssen sofort angelegt werden

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 516

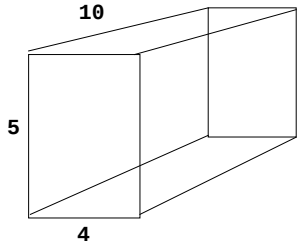
Ziele:

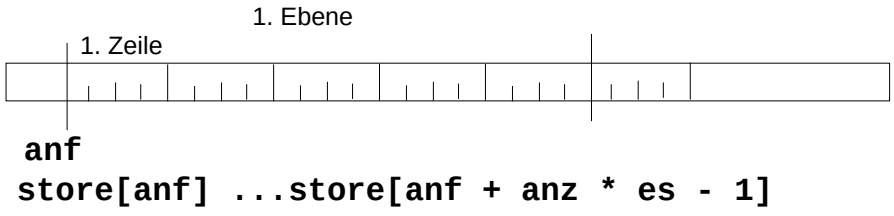
Speicherung von Arrays in Java wiederholen

in der Vorlesung:

Erläuterungen dazu

Linearisierte Speicherung von Arrays





zeilenweise Linearisierung eines n-stufigen Arrays (z. B. in Pascal):

```
a: array[u1..o1, u2..o2, ..., un..on] of real;
```

abgebildet auf linearen Speicher, z. B. Byte-Array **`store`** ab Index **`anf`**:

```
store[anf] ... store[anf + anz * es - 1]
```

mit Anzahl der Elemente `anz = sp1 * sp2 * ... * spn`

i-te Indexspanne `spi = oi - ui + 1`

Elementgröße in Bytes `es`

Indexabbildung: **`a[i1, i2, ..., in]`** entspricht **`store[k]`** mit

```
k = anf +  (i1-u1)*sp2*sp3*...*spn*es +
           (i2-u2)*    sp3*...*spn*es +...+
           (in-un)*                   es
= (...(i1*sp2 + i2)*sp3 + i3)* ...+ in)*es +
```

konstanter Term

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 517

Ziele:

Speicherabbildung für Arrays verstehen

in der Vorlesung:

Erläuterungen zur Linearisierung am Beispiel

Verständnisfragen:

- Wie vereinfacht sich die Formel zur Indexabbildung, wenn wie in C, C++, Java die Untergrenzen immer 0 sind?
- Welchen Wert hat der Index `k`, für `a[0,0,...,0]`?

Funktionen

Typ einer Funktion ist ihre Signatur: D = P -> R mit Parametertyp P, Ergebnistyp R
mehrere Parameter entspricht Parametertupel $P = P_1 \times ... \times P_n$,
kein Parameter oder Ergebnis: P bzw. R ist leerer Typ (`void` in Java, C, C++; `unit` in SML)

Funktion **höherer Ordnung (Higher Order Function):**
Funktion mit einer Funktion als Parameter oder Ergebnis, z. B. `(int×(int->int))->int`

Operationen: Aufruf

Funktionen in imperativen Sprachen: nicht als Ausdruck, nur als Deklaration

Funktionen als Parameter in den meisten Sprachen.
Geschachtelte Funktionen in Pascal, Modula-2, Ada - nicht in C.
Globale **Funktionen als Funktionsergebnis** und **als Daten** in C und Modula-2.
Diese Einschränkungen garantieren die **Laufzeitkeller-Disziplin**:
Beim Aufruf müssen alle statischen Vorgänger noch auf dem Laufzeitkeller sein.

Funktionen in funktionalen Sprachen:
uneingeschränkte Verwendung auch als Datenobjekte;
Aufrufschachteln bleiben solange erhalten, wie sie gebraucht werden

Notation für eine Funktion als Wert: Lambda-Ausdruck, meist nur in funktionalen Sprachen:

SML: `fn a => 2 * a`

Algol-68: `(int a) int: 2 * a`

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 518

Ziele:

Funktionen als Daten verstehen

in der Vorlesung:

- Beispiele dazu
- siehe auch Kapitel 7. Funktionale Programmierung

nachlesen:

..., Abschnitt 2.3.3

Verständnisfragen:

- Welche Einschränkungen garantieren in Pascal, C und Modula-2 Laufzeitkeller-Disziplin?

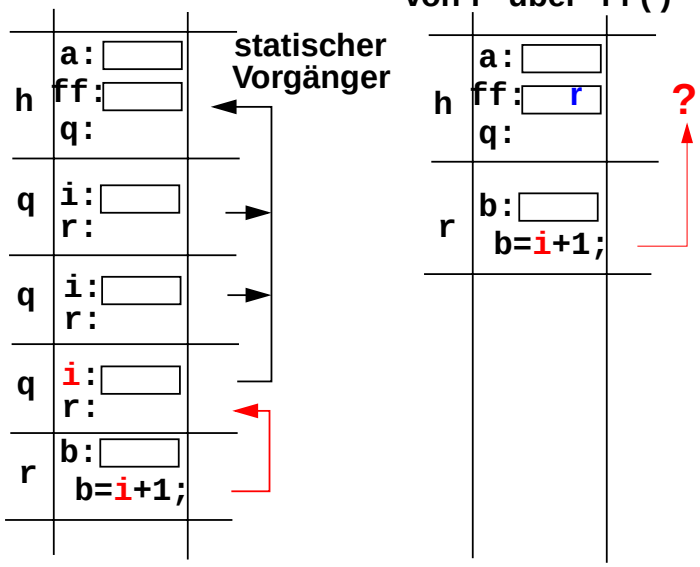
Beispiel für Verletzung der Laufzeitkeller-Disziplin

In imperativen Sprachen ist die Verwendung von Funktionen so eingeschränkt, dass bei Aufruf einer Funktion die Umgebung des Aufrufes (d. h. alle statischen Vorgänger-Schachteln) noch auf dem Laufzeitkeller liegen.
Es darf z. B. nicht eine eingeschachtelte Funktion an eine globale Variable zugewiesen und dann aufgerufen werden (vgl. GPS-4.6):

Programm mit geschachtelten Funktionen

```
h  float a;
   fct ff;
     q
        int i;
          r
            int b;
            b=i+1;
        if(..) q();
        r();
        ff = r;
   q();
   ff();
```

Laufzeitkeller



Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 518a

Ziele:

Verletzung der Laufzeitkeller-Disziplin verstehen

in der Vorlesung:

Beispiel wird erklärt

nachlesen:

..., Abschnitt 2.3.3

Verständnisfragen:

- Welche Einschränkungen garantieren in Pascal, C und Modula-2 Laufzeitkeller-Disziplin?

Mengen

Wertebereich ist die **Potenzmenge:** **D = P (D_e)** oder
Menge der charakteristischen Funktionen **D = D_e -> bool** mit Elementtyp D_e
D_e muss meist einfach, geordnet und von beschränkter Kardinalität sein.
(Allgemeine Mengentypen z. B. in der Spezifikationssprache **SETL**.)

Operationen: Mengenoperationen und Vergleiche

z. B. **in Pascal**:

```
var m, m1, m2: set of 0..15;
e in m      m1 + m2        m1 * m2     m1 - m2
```

Notation für Mengenwerte: in Pascal: `[1, 3, 5]`

Effiziente Implementierung durch **Bit-Vektor** (charakteristische Funktion):
`array [De] of boolean`
mit logischen Operationen auf Speicherworten als Mengenoperationen.

in Modula-2: vordefinierter Typ
`BITSET = SET OF [0..l-1]` mit `l` Bits im Speicherwort.

in C:
kein Mengentyp, aber logische Operationen `|, &, ~, ^`
auf Bitmustern vom Typ **`unsigned`**.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 519

Ziele:

Mengen und ihre Repräsentation

in der Vorlesung:

Erläuterungen dazu

nachlesen:

..., Abschnitt 2.3.4

Verständnisfragen:

- Begründen Sie die Äquivalenz der beiden abstrakten Definitionen der Wertemenge und den Zusammenhang zur Bitvektorimplementierung.

Stellen (Referenzen, Pointer)

Wertebereich **`D = S`$_W$ `| {nil}`**

`S`$_W$: Speicherstellen, die Werte eines Typs `W` aufnehmen können.

`nil` eindeutige Referenz, verschieden von allen Speicherstellen

Operationen: Zuweisung, Identitätsvergleich, Inhalt

Wertnotation und Konstruktor:

a. Stelle einer deklarierten **Variable**, z. B. in C: `int i; int *p = &i;`

b. Stelle eines dynamisch generierten Objektes als Ergebnis eines **Konstruktoraufrufs**, z. B. in Java `Circles cir = new Circles (0, 0, 1.0);`

Stellen als Datenobjekte werden nur in **imperativen Sprachen** benötigt!

Sprachen **ohne Zuweisungen** brauchen nicht zwischen einer Stelle und ihrem Inhalt zu unterscheiden ("**referentielle Transparenz"**)

Objekte in objektorientierten Sprachen haben eine **Stelle**.
Sie bestimmt die Identität des Objektes.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 520

Ziele:

Typisierte Speicherstellen in imperativen Sprachen

in der Vorlesung:

- Zusammenhang zu Variablen in Abschnitt 4

Zusammenfassung zum Kapitel 5

Mit den Vorlesungen und Übungen zu Kapitel 5 sollen Sie nun Folgendes können:

5.1 Allgemeine Begriffe zu Datentypen

- Typeigenschaften von Programmiersprachen verstehen und mit treffenden Begriffen korrekt beschreiben
- Mit den abstrakten Konzepten beliebig strukturierte Typen entwerfen
- Parametrisierung und generische Definition von Typen unterscheiden und anwenden

5.2 Datentypen in Programmiersprachen

- Ausprägungen der abstrakten Typkonzepte in den Typen von Programmiersprachen erkennen
- Die Begriffe Klassen, Typen, Objekte, Werte sicher und korrekt verwenden
- Die Vorkommen von Typkonzepten in wichtigen Programmiersprachen kennen
- Speicherung von Reihungen verstehen

Ziele:

Ziele des Kapitels erkennen

in der Vorlesung:

Erläuterungen dazu

6. Funktionen, Parameterübergabe

Themen dieses Kapitels:

- Begriffe zu Funktionen und Aufrufen
- Parameterübergabearten
  call-by-value, call-by-reference, call-by-value-and-result
  in verschiedenen Sprachen

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 601

Ziele:
Übersicht zu diesem Kapitel

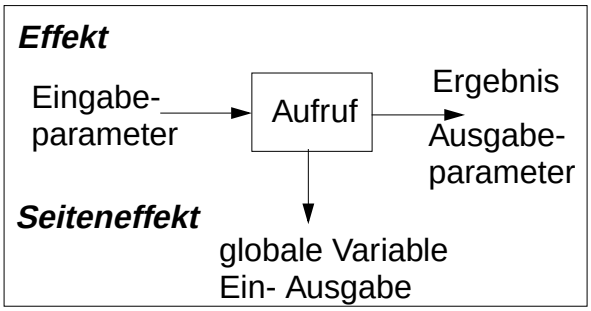
in der Vorlesung:
Erläuterungen dazu

Begriffe zu Funktionen und Aufrufen

Funktionen sind Abstraktionen von Rechenvorschriften.
Funktionen, die kein Ergebnis liefern, nennt man auch **Prozeduren**.
In objektorientierten Sprachen nennt man Funktionen auch **Methoden**.

Effekte eines Funktionsaufrufes:
Berechnung des **Funktionsergebnis** und ggf. der **Ausgabeparameter** aus den **Eingabeparametern**.

Seiteneffekte:
globale Variable schreiben,
Ein- und Ausgabe



Formale Parameter (FP): Namen für Parameter in der Funktionsdefinition.
Aktuelle Parameter (AP): Ausdrücke im Aufruf, deren Werte oder Stellen übergeben werden.

```
int Sqr (int i) { return i*i; }     Sqr (x+y)
```

Verschiedene Arten der Parameterübergabe:

call-by-value, call-by-reference, call-by-result, call-by-value-and-result, (call-by-name)

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 602

Ziele:

Grundbegriffe der Funktionsaufrufe

in der Vorlesung:

- Erläuterung der Begriffe

nachlesen:

..., Abschnitt 5, 5.1, 5.2

Verständnisfragen:

- Geben Sie Beispiele für Klassenmethoden in Java mit und ohne Seiteneffekt.
- Warum haben Objektmethoden i.a. Seiteneffekte?

Ausführung eines Funktionsaufrufes

Das Prinzip der Funktionsaufrufe ist in fast allen Sprachen gleich:

Ein Aufruf der Form `Funktionsausdruck (aktuelle Parmeter)`

wird in **3 Schritten** ausgeführt

1. **Funktionsausdruck auswerten**, liefert eine Funktion
2. **Aktuelle Parameter** auswerten und **an formale Parameter der Funktion binden** nach den speziellen Regeln der Parameterübergabe; Schachtel auf dem Laufzeitkeller bilden.
3. Mit diesen Bindungen den **Rumpf der Funktion ausführen** und ggf. das Ergebnis des Aufrufes berechnen; Schachtel vom Laufzeitkeller entfernen.

Beispiel:

```
z =  a[i].next.m   (x*y, b[j])
```

1. liefert Funktion
2. liefert zwei AP-Werte, werden an FP gebunden
3. Ausführung des Funktionsrumpfes liefert ein Ergebnis

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 602a

Ziele:

Funktionsaufrufe verstehen

in der Vorlesung:

- Erläuterung der Schritte am Beispiel

nachlesen:

..., Abschnitt 5, 5.1, 5.2

Beispiel zur Parameterübergabe

```
program
   i: integer;
   a: array [1..6] of integer;

   procedure p (x: integer, y: integer)
      t: integer;
      begin
         output x, y;          /* 2 formale Param. wie übergeben */
         t := x; x := y; y := t;
         output x, y;          /* 3 formale Param. nach Zuweisungen */
         output i, a[i];       /* 4 globale Variable der akt. Param.*/
      end;

begin
   i:= 3; a[3] := 6; a[6] := 10;
   output i, a[3];           /* 1 aktuelle Param. vor Aufruf */
   p (i, a[i]);
   output i, a[3];           /* 5 aktuelle Param. nach Aufruf */
end
```


Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 603

Ziele:

Parameterübergabe variieren, Wirkung unterscheiden

in der Vorlesung:

- Beispiel für folgende Folien
- Programmpositionen in Bezug auf die Parameter erläutern

Call-by-value

Der **formale Parameter ist eine lokale Variable**, die **mit dem Wert des aktuellen Parameters initialisiert** wird.

Zuweisungen im Funktionsrumpf haben keine Wirkung auf die aktuellen Parameter eines Aufrufes.

Die **Werte der aktuellen Parameter** werden in die Parametervariablen **kopiert**.

Sprachen: fast alle Sprachen, z. B. Java, C, C++, Pascal, Modula-2, Ada, FORTRAN

Variante **call-by-strict-value:**
Der formale Parameter ist ein Name für den Wert des aktuellen Parameters.

Zuweisungen im Funktionsrumpf an formale Parameter sind nicht möglich.

Implementierung:

a. wie call-by-value und Zuweisungen durch Übersetzer verbieten

b. wie call-by-reference und Zuweisungen durch Übersetzer verbieten; erspart Kopieren

Sprachen: Algol-68, funktionale Sprachen

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 604

Ziele:

Übergabeart präzise erklären können

in der Vorlesung:

- Allgemeinste, gebräuchlichste Übergabeart
- Ausgabe des Beispiels zeigen
- call-by-strict-value verdeutlichen

nachlesen:

..., Abschnitt 5.2.1, 5.2.2

Call-by-reference

Der **formale Parameter** ist ein **Name für die Stelle des aktuellen Parameters**. Sie wird zum Zeitpunkt des Aufrufs bestimmt.

geeignet für Eingabe- und Ausgabeparameter (**transient**)
Der **aktuelle Parameter muss eine Stelle haben**: unzulässig: `h (5)` oder `h (i+1)`
Stelle des Elementes `a[i]` wird bei Beginn des Aufrufes bestimmt: `h (a[i])`
Jede **Operation mit dem formalen Parameter wirkt sofort auf den aktuellen Parameter**.

Aliasing: Mehrere Namen für dieselbe Variable (aktueller und formaler Parameter)

Vorsicht bei mehreren gleichen aktuellen Parametern! `g (x, x)`

Implementierung:
Der formale Parameter wird eine Referenzvariable. Sie wird bei einem Aufruf initialisiert mit der Stelle des aktuellen Parameters. Bei jedem Zugriff wird einmal zusätzlich dereferenziert.

Sprachen: Pascal, Modula-2, FORTRAN, C++

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 605

Ziele:

Zusammenhang zu Variablen und Stellen verstehen

in der Vorlesung:

- Ausgabe des Beispiels zeigen

nachlesen:

..., Abschnitt 5.2.1, 5.2.2

Verständnisfragen:

- Wie simulieren Sie call-by-reference in C?

Call-by-result

Der formale Parameter ist eine **lokale, nicht initialisierte Variable**. Ihr Wert wird **nach erfolgreichem Abarbeiten des Aufrufes an die Stelle des aktuellen Parameters zugewiesen**. Die Stelle des aktuellen Parameters wird beim Aufruf bestimmt.

Geeignet als **Ausgabeparameter**.
Die Wirkung auf den aktuellen Parameter tritt erst beim Abschluss des Aufrufs ein.
Aktueller Parameter muss eine Stelle haben.
Kopieren erforderlich.
Sprachen: Ada (`out`-Parameter)

Call-by-value-and-result

Der formale Parameter ist eine **lokale Variable, die mit dem Wert des aktuellen Parameters initialisiert wird**. Ihr Wert wird nach erfolgreichem Abarbeiten des Aufrufes an die Stelle des aktuellen Parameters zugewiesen. Die Stelle des aktuellen Parameters wird beim Aufruf bestimmt.

Geeignet als Ein- und Ausgabeparameter (**transient**);
Die Wirkung auf den aktuellen Parameter tritt erst beim Abschluss des Aufrufs ein.
Aktueller Parameter muss eine Stelle haben.
Zweimal Kopieren erforderlich.

Sprachen: Ada (`in out`-Parameter)

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 606

Ziele:

Unterschied zu call-by-reference verstehen

in der Vorlesung:

- Ausgabe des Beispiels zeigen

nachlesen:

..., Abschnitt 5.2.1, 5.2.2

Verständnisfragen:

- Skizzieren Sie ein möglichst kurzes Programm, das mit call-by-value-and-result oder call-by-reference unterschiedliche Ausgabe erzeugt.

Parameterübergabe in verschiedenen Sprachen

Java: nur call-by-value (auch Objektreferenzen werden call-by-value übergeben)

Pascal, Modula-2, C++ wahlweise call-by-value, call-by-reference

C#: wahlweise call-by-value, call-by-reference, call-by-result

C: nur call-by-value;
call-by-reference kann simuliert werden durch die Übergabe von Stellen:

```
void p (int i, int *a) { ... *a = 42; ... }  int x;   p (5, &x);
```

Ada: wahlweise call-by-value (**`in`**), call-by-result (**`out`**), call-by-value-and-result (**`in out`**).
Bei zusammengesetzten Objekten ist für **`in out`** auch call-by-reference möglich.
Aktuelle Parameter können auch mit den Namen der formalen benannt und dann in beliebiger Reihenfolge angegeben werden: `p (a => y[k], i => 5)`.
Für formale Parameter können default-Werte angegeben werden; dann kann der aktuelle Parameter weggelassen werden.

FORTRAN:
call-by-value, falls an den formalen Parameter nicht zugewiesen wird,
sonst call-by-reference oder call-by-value-and-result (je nach Übersetzer)

Algol-60: call-by-value, call-by-name (ist default!)
Algol-68: call-by-strict-value

funktionale Sprachen: call-by-strict-value oder lazy-evaluation (entspricht call-by-name)

Vorlesung Grundlagen der Programmiersprachen SS 2010 / Folie 607

Ziele:

Parameterübergabe wichtiger Sprachen kennen

in der Vorlesung:

- Erläuterungen dazu
- Beispiele zu Besonderheiten in Ada

Verständnisfragen:

- Welche Übergabearten sind in einer Sprache sinnvoll definierbar, wenn sie keine Variablen mit Zuweisungen hat?

Zusammenfassung zum Kapitel 6

Mit den Vorlesungen und Übungen zu Kapitel 6 sollen Sie nun Folgendes können:

- Funktionen, Aufrufen und Parameterübergabe präzise mit treffenden Begriffen erklären können
- Die Arten der Parameterübergabe unterscheiden und sinnvoll anwenden können
- Die Parameterübergabe wichtiger Sprachen kennen

Ziele:

Ziele des Kapitels erkennen

in der Vorlesung:

Erläuterungen dazu

7. Funktionale Programmierung

Themen dieses Kapitels:

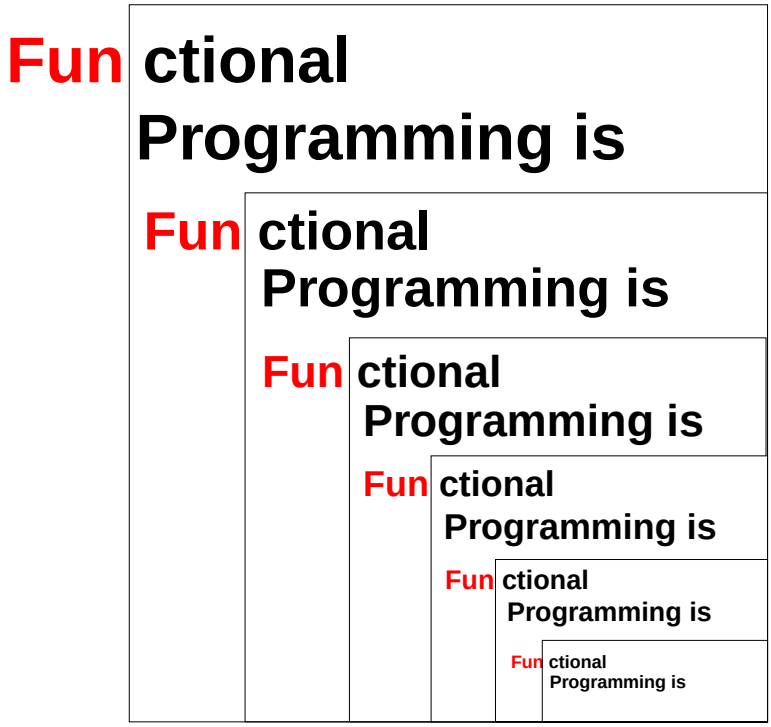
- Grundbegriffe und Notation von SML
- Rekursionsparadigmen: Induktion, Rekursion über Listen
- End-Rekursion und Programmiertechnik „akkumulierender Parameter“
- Berechnungsschemata mit Funktionen als Parameter
- Funktionen als Ergebnis und Programmiertechnik „Currying“

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 701

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

Functional Programming is Fun



Ziele:

Just fun

in der Vorlesung:

no comment

Übersicht zur funktionalen Programmierung

Grundkonzepte: Funktionen und Aufrufe, Ausdrücke
keine Variablen, Zuweisungen, Ablaufstrukturen, Seiteneffekte

Elementare Sprachen (pure LISP) brauchen nur wenige Konzepte:
Funktionskonstruktor, bedingter Ausdruck, Literale, Listenkonstruktor und -selektoren, Definition von Bezeichnern für Werte

Mächtige Programmierkonzepte durch Verwendung von:
rekursiven Funktionen und Datenstrukturen,
Funktionen höherer Ordnung als Berechnungsschemata

Höhere funktionale Sprachen (SML, Haskell):
statische Bindung von Bezeichnern und Typen,
völlig orthogonale, höhere Datentypen, polymorphe Funktionen (Kapitel 6),
modulare Kapselung, effiziente Implementierung

Funktionaler Entwurf:
strukturell denken - nicht in Abläufen und veränderlichen Zuständen,
fokussiert auf **funktionale Eigenschaften** der Problemlösung,
Nähe zur Spezifikation, Verifikation, Transformation

Funktionale Sprachen:
LISP, Scheme, Hope, SML, Haskell, Miranda, ...
früher: Domäne der KI; heute: Grundwissen der Informatik, praktischer Einsatz

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 703

Ziele:

Eigenschaften funktionaler Sprachen kennenlernen

in der Vorlesung:

- Erläuterungen dazu
- Verweis auf schon besprochene Konzepte: Rolle von Variablen, Datentypen, Funktionen

nachlesen:

..., Abschnitt 13

Verständnisfragen:

- Warum kann man Pascal nicht auf eine funktionale Teilsprache reduzieren?
- Gilt das genauso für C?

Wichtige Sprachkonstrukte von SML: Funktionen

Funktionen können direkt notiert werden, ohne Deklaration und ohne Namen:
Funktionskonstruktor (**lambda-Ausdruck**: Ausdruck, der eine Funktion liefert):

```
fn FormalerParameter => Ausdruck
```

```
fn i => 2 * i          Funktion, deren Aufruf das Doppelte ihres Parameters liefert
fn (a, b) => 2 * a + b
```

Beispiel, unbenannte Funktion als Parameter eines Aufrufes:

```
map (fn i => 2 * i, [1, 2, 3])
```

Funktionen haben **immer einen Parameter**:

statt mehrerer Parameter ein Parameter-Tupel wie (a, b)

`(a, b)` ist ein **Muster** für ein Paar als Parameter

statt keinem Parameter ein leerer Parameter vom Typ **unit**, entspricht **void**

Typangaben sind optional. Trotzdem prüft der Übersetzer streng auf korrekte Typisierung. Er berechnet die Typen aus den benutzten Operationen (**Typinferenz**)

Typangaben sind nötig zur **Unterscheidung von `int` und `real`**

```
fn i : int => i * i
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 704

Ziele:

Funktionen in SML verstehen

in der Vorlesung:

- Erläuterung von Funktionen als Werte
- Bestimmung der Funktionssignaturen aus den verwendeten Operationen ohne explizite Typangaben (Typinferenz)

nachlesen:

..., Abschnitt 13.1

nachlesen:

L. C. Paulson: ML for the Working Programmer, siehe Folie 004

Wichtige Sprachkonstrukte von SML: Funktionsaufrufe

allgemeine Form eines Aufrufes: ***Funktionsausdruck Parameterausdruck***

```
Dupl 3
(fn i => 2 * i) 3
```

Klammern können den Funktionsausdruck m it dem aktuellen Parameter zusammenfassen:

```
(fn i => 2 * i) (Dupl 3)
```

Parametertupel werden geklammert:

```
(fn (a, b) => 2 * a + b) (4, 2)
```

Auswertung von Funktionsaufrufen wie in GPS-6-2a beschrieben.

Parameterübergabe: **call-by-strict-value**

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 704a

Ziele:

Notation von Aufrufen

in der Vorlesung:

- Erläuterung der Funktionsaufrufe
- Bedeutung der Konstrukte

nachlesen:

..., Abschnitt 13.1

nachlesen:

L. C. Paulson: ML for the Working Programmer, siehe Folie 004

Wichtige Sprachkonstrukte von SML: Definitionen

Eine **Definition** bindet den Wert eines Ausdrucks an einen Namen:

```
val four = 4;
val Dupl = fn i => 2 * i;
val Foo = fn i => (i, 2*i);
val x = Dupl four;
```

Eine Definition kann ein **Tupel von Werten** an ein **Tupel von Namen**, sog. ***Muster***, binden: allgemeine Form:

```
val Muster = Ausdruck;
```

```
val (a, b) = Foo 3;
```

Der Aufruf **`Foo 3`** liefert ein Paar von Werten, sie werden gebunden an die Namen **`a`** und **`b`** im Muster für Paare **`(a, b)`**.

Kurzform für Funktionsdefinitionen:

```
fun Name FormalerParameter = Ausdruck;
```

```
fun Dupl i = 2 * i;
fun Fac n = if n <= 1 then 1 else n * Fac (n-1);
```

bedingter Ausdruck: Ergebnis ist der Wert des **`then`**- oder **`else`**-Ausdruckes

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 704b

Ziele:

Notation von Definitionen verstehen

in der Vorlesung:

- Bedeutung der Konstrukte
- Erläuterung der Bindung durch Muster
- Funktionsdefinitionen in allgemeiner und in Kurzform

nachlesen:

..., Abschnitt 13.1

nachlesen:

L. C. Paulson: ML for the Working Programmer, siehe Folie 004

Rekursionsparadigma Induktion

Funktionen für induktive Berechnungen sollen schematisch entworfen werden:

Beispiele:

induktive Definitionen:

$$n! = \begin{cases} 1 & \text{für } n <= 1 \\ n*(n-1)! & \text{für } n > 1 \end{cases}$$

$$b^n = \begin{cases} 1.0 & \text{für } n <= 0 \\ b*b^{n-1} & \text{für } n > 0 \end{cases}$$

rekursive Funktionsdefinitionen:

```
fun Fac n =
   if n <= 1
      then 1
      else n * Fac (n-1);

fun Power (n, b) =
   if n <= 0
      then 1.0
      else b * Power (n-1, b);
```

Schema:

fun F a = **if** Bedingung über **a**
then nicht-rekursiver Ausdruck über **a**
else rekursiver Ausdruck über **F**("verkleinertes **a**")

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 705

Ziele:

Induktionsschema an Beispielen

in der Vorlesung:

- Erinnerung an Terminierung rekursiver Funktionen
- Erläuterung der Beispiele

nachlesen:

..., Abschnitt 13.1

Verständnisfragen:

- Vergleichen Sie die Funktionen mit entsprechenden imperativen Programmen.

Induktion - effizientere Rekursion

Induktive Definition und rekursive Funktionen zur Berechnung von Fibonacci-Zahlen:

induktive Definition:

$$\text{Fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{Fib}(n-1)+\text{Fib}(n-2) & \text{für } n > 1 \end{cases}$$

rekursive Funktionsdefinition:

```
fun Fib n =
   if n = 0
      then 0
      else if n = 1
         then 1
         else Fib(n-1)+Fib (n-2);
```

`Fib` effizienter:
Zwischenergebnisse als Parameter, Induktion aufsteigend
(allgemeine Technik siehe „Akkumulierende Parameter“):

```
fun AFib (n, alt, neu) =
     if n = 1 then neu
     else AFib (n-1, neu, alt+neu);

fun Fib n = if n = 0 then 0 else AFib (n, 0, 1);
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 705a

Ziele:

Rekursionstyp erkennen

in der Vorlesung:

- Ineffizienz der Mehrfachrekursion zeigen

nachlesen:

..., Abschnitt 13.1

Verständnisfragen:

- Vergleichen Sie die Funktionen mit entsprechenden imperativen Programmen.

Funktionsdefinition mit Fallunterscheidung

Funktionen können übersichtlicher definiert werden durch

- **Fallunterscheidung** über den Parameter - statt bedingter Ausdruck als Rumpf,
- formuliert durch **Muster**
- **Bezeichner** darin werden **an Teil-Werte des aktuellen Parameters gebunden**

bedingter Ausdruck als Rumpf:

```
fun Fac n =
   if n=1  then 1
           else n * Fac (n-1);

fun Power (n, b) =
   if n = 0
      then 1.0
      else b * Power (n-1, b);
```

Fallunterscheidung mit Mustern:

```
fun  Fac (1) = 1
|    Fac (n) = n * Fac (n-1);

fun  Power (0, b) = 1.0
|    Power (n, b) =
          b * Power (n-1, b);

fun  Fib (0) = 0
|    Fib (1) = 1
|    Fib (n) =
          Fib(n-1) + Fib(n-2);
```

Die Muster werden in der **angegebenen Reihenfolge** gegen den aktuellen Parameter geprüft. Es wird der erste Fall gewählt, dessen Muster trifft. Deshalb muss ein allgemeiner **„catch-all“-Fall am Ende** stehen.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 705b

Ziele:

Kurz-Notation kennenlernen

in der Vorlesung:

- Erläuterung der Kurzschreibweise

nachlesen:

..., Abschnitt 13.1.2

Listen als rekursive Datentypen

Parametrisierter Typ für lineare Listen vordefiniert: (Typparameter `'a`; polymorpher Typ)

```
datatype 'a list = nil | :: of ('a * 'a list)
```

definert den 0-stelligen Konstruktor **`nil`** und den 2-stelligen Konstruktor **`::`**

Schreibweisen für Listen:

`x :: xs`	eine Liste mit erstem Element `x` und der Restliste `xs`
`[1, 2, 3]`	für `1 :: 2 :: 3 :: nil`

Nützliche **vordefinierte Funktionen auf Listen**:

`hd l`	erstes Element von `l`
`tl l`	Liste `l` ohne erstes Element
`length l`	Länge von `l`
`null l`	Prädikat: ist `l` gleich `nil`?
`l1 @ l2`	Liste aus Verkettung von `l1` und `l2`

Funktion, die die Elemente einer Liste addiert:

```
fun Sum l = if null l then 0
                      else (hd l) + Sum (tl l);

Signatur:               Sum: int list -> int
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 706

Ziele:

Notation und Operationen für Listen kennenlernen

in der Vorlesung:

Erläuterungen

- zur Notation,
- zur Typdefinition,
- zu den elementaren Funktionen über Listen,
- zur Konkatenation von Listen mit Kopie der linken Teilliste, um die referentielle Konsistenz zu bewahren
- Werte können nicht geändert werden!

nachlesen:

..., Abschnitt 13.1.1, 13.1.2

Verständnisfragen:

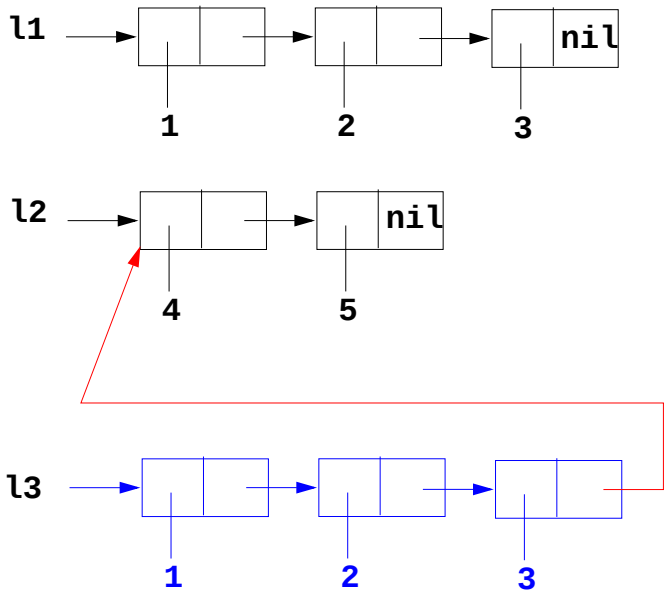
- Die Operation zur Listenverkettung erfordert Kopieren. Warum?
- Beschreiben Sie die Operation durch Angabe eines Speicherbildes dazu.

Konkatenation von Listen

In funktionalen Sprachen werden Werte nie geändert.

Bei der **Konkatenation** zweier Listen wird die **Liste des linken Operands kopiert**.

```
val l1 = [1, 2, 3];
val l2 = [4, 5];
val l3 = l1 @ l2;
```



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 706a

Ziele:

Werte sind unveränderlich!

in der Vorlesung:

- Konkatenation erläutern;
- es gibt keine veränderlichen Variablen;
- durch Kopieren können neue Werte erzeugt werden, ohne existierende zu verändern.

nachlesen:

..., Abschnitt 13.1

nachlesen:

L. C. Paulson: ML for the Working Programmer, siehe Folie 004

Einige Funktionen über Listen

Liste[n,...,1] erzeugen:

```
fun  MkList 0   = nil
|    Mklist n   = n :: MkList (n-1);
```

Signatur: `MkList: int -> int list`

Fallunterscheidung mit Listenkonstruktoren `nil` und `::` in Mustern:
Summe der Listenelemente:

```
fun Sum (nil)   = 0
|   Sum (h::t)  = h + Sum t;
```

Prädikat: **Ist das Element in der Liste enthalten?:**

```
fun Member (nil, m)= false
|   Member (h::t,m)= if h = m then true else Member (t,m);
```

Polymorphe Signatur: `Member: ('a list * 'a) -> bool`

Liste als Konkatenation zweier Listen berechnen (@-Operator):

```
fun Append (nil, r)= r
|   Append (l, nil)= l
|   Append (h::t, r)= h :: Append (t, r);
```

Die linke Liste wird neu aufgebaut!

Polymorphe Signatur: `Append: ('a list * 'a list) -> 'a list`

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 707

Ziele:

Techniken für Funktionen über Listen

in der Vorlesung:

- Erläuterungen zu den Funktionen und ihren Signaturen,
- Erläuterung der Kurzschreibweise
- Append-Funktion: Listen werden nicht verändert, sondern kopiert wenn nötig!

nachlesen:

..., Abschnitt 13.1.2

Übungsaufgaben:

- Weitere Listenfunktionen entwerfen

Verständnisfragen:

Weshalb kann man keine zyklische Liste erzeugen?

Rekursionsschema Listen-Rekursion

lineare Listen sind als **rekursiver Datentyp** definiert:

```
datatype 'a list = nil | :: of ('a * 'a list)
```

Paradigma: Funktionen haben die gleiche Rekursionsstruktur wie der Datentyp:

```
fun F l =   if l=nil   then nicht-rekursiver Ausdruck
                       else Ausdruck über hd l und F(tl l);

fun Sum l = if l=nil   then 0
                       else (hd l) + Sum (tl l);
```

Dasselbe in Kurzschreibweise mit Fallunterscheidung:

```
fun F (nil)     = nicht-rekursiver Ausdruck
|   F (h::t)    = Ausdruck über h und F t

fun Sum (nil)   = 0
|   Sum (h::t)  = h + Sum t;
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 708

Ziele:

Rekusionsschema anwenden lernen

in der Vorlesung:

Erläuterungen

- zum Rekursionsschema,
- Anwendung auf andere Funktionen über Listen

nachlesen:

..., Abschnitt 13.1.1, 13.1.2

Einige Funktionen über Bäumen

Parametrisierter Typ für Bäume:

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil
```

Paradigma: Funktionen haben die gleiche Rekursionsstruktur wie der Datentyp.

Beispiel: einen Baum spiegeln

```
fun Flip (treeNil)          = treeNil
|   Flip (node (l, v, r))   = node (Flip r, v, Flip l);
```

polymorphe Signatur: `Flip: 'a tree -> 'a tree`

Beispiel: einen Baum auf eine Liste der Knotenwerte abbilden (hier in Infix-Form)

```
fun Flatten (treeNil)        = nil
|   Flatten (node (l, v, r)) = (Flatten l) @ (v :: (Flatten r));
```

polymorphe Signatur: `Flatten: 'a tree -> 'a list`

Präfix-Form: ...

Postfix-Form: ...

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 709

Ziele:

Paradigma Datenrekursion für Bäume

in der Vorlesung:

- Datentyp und Schema erläutern
- Flatten erläutern
- Flatten auf Präfix- und Postfix-Form umstellen

nachlesen:

..., Abschnitt 13.1.2

Übungsaufgaben:

Stellen Sie Flatten auf Präfix- und Postfix-Form um.

End-Rekursion

In einer Funktion `f` heißt ein **Aufruf** von `f` **end-rekursiv**, wenn er (als letzte Operation) das Funktionsergebnis bestimmt, sonst heißt er **zentral-rekursiv**.

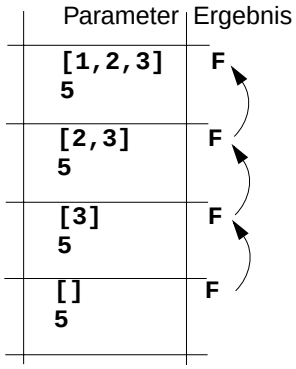
Eine **Funktion** heißt **end-rekursiv**, wenn **alle rekursiven Aufrufe end-rekursiv** sind.

`Member` ist end-rekursiv:

```
fun Member (l, a) =
   if null l then false
   else if (hd l) = a
        then true
        else Member (tl l, a);
```

`Sum` ist zentral-rekursiv:

```
fun Sum (nil)= 0
|   Sum (h::t)= h + (Sum t);
```



Laufzeitkeller für **`Member ([1,2,3], 5)`**

Ergebnis wird durchgereicht -
ohne Operation darauf

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 710

Ziele:

Prinzip: End-Rekursion verstehen

in der Vorlesung:

- End-Rekursion erkennen
- Aufrufergebnis verstehen

Übungsaufgaben:

- An Beispielen End-Rekursion erkennen

End-Rekursion entspricht Schleife

Jede **imperative Schleife** kann in eine **end-rekursive Funktion** transformiert werden.
Allgemeines Schema:

```
while ( p(x) ) {x = r(x);}  return q(x);
fun While x = if p x then While (r x) else q x;
```

Jede **end-rekursive** Funktion kann in eine imperative Form transformiert werden:
Jeder **end-rekursive Aufruf** wird durch einen **Sprung** an den Anfang der Funktion (oder durch eine **Schleife**) ersetzt:

```
fun Member (l, a) =
   if null l then false
   else if (hd l) = a then true else Member (tl l, a);
```

Imperativ in C:

```
int Member (ElemList l, Elem a)
{ Begin:   if (null (l)) return 0 /*false*/;
           else if (hd (l) == a) return 1 /*true*/;
                else { l = tl (l); goto Begin;}
}
```

Gute Übersetzer leisten diese Optimierung automatisch - auch in imperativen Sprachen.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 710a

Ziele:

Prinzip: End-Rekursion - imperative Schleife

in der Vorlesung:

- Schleife in Funktion transformieren
- Funktion in Schleife transformieren

Übungsaufgaben:

- An Beispielen End-Rekursion transformieren

Verständnisfragen:

- In welchen Fällen benötigt man Hilfsvariable um einen end-rekursiven Aufruf zu transformieren?

Technik: Akkumulierender Parameter

Unter bestimmten Voraussetzungen können **zentral-rekursive** Funktionen in **end-rekursive** transformiert werden:

Ein **akkumulierender Parameter** führt das bisher berechnete Zwischenergebnis mit durch die Rekursion. Die Berechnungsrichtung wird umgekehrt,

z. B.:Summe der Elemente einer Liste **zentral-rekursiv:**

```
fun Sum (nil)= 0
|   Sum (h::t)= h + (Sum t);
```

Sum [1, 2, 3, 4] berechnet

1 + (2 + (3 + (4 + (0))))

transformiert in end-rekursiv:

```
fun  ASum (nil, a:int) = a
|    ASum (h::t,a)    = ASum (t, a + h);

fun Sum l = ASum (l, 0);
```

ASum ([1, 2, 3, 4], 0) berechnet

((((0 + 1) + 2) + 3) + 4)

Die Verknüpfung (hier +) muß **assoziativ** sein.
Initial wird mit dem **neutralen Element der Verknüpfung** (hier `0`) aufgerufen.

Gleiche Technik bei AFib (GPS-7.5a); dort 2 akkumulierende Parameter.

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 711

Ziele:

Systematische Transformationstechnik verstehen

in der Vorlesung:

- Ausgeführte Verknüpfungen zeigen
- Aufrufkeller zeigen

Übungsaufgaben:

- Technik an Beispielen anwenden

Liste umkehren mit akkumulierendem Parameter

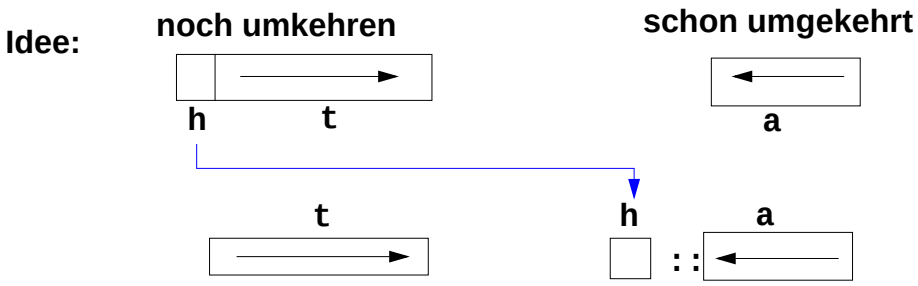
Liste umkehren:

```
fun Reverse (nil)=  nil
|   Reverse (h::t)= Append (Reverse t, h::nil);
```

`Append` dupliziert die linke Liste bei jeder Rekursion von **`Reverse`**,
benötigt also k mal **`::`**, wenn k die Länge der linken Liste ist. Insgesamt benötigt **`Reverse`** wegen der Rekursion (n-1) + (n-2) + ... + 1 mal **`::`**, also Aufwand $O(n^2)$.

Transformation von **`Reverse`** führt zu linearem Aufwand:

```
fun AReverse (nil, a)= a
|   AReverse (h::t,a)= AReverse (t, h::a);
fun Reverse l = AReverse (l, nil);
```



Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 711a

Ziele:

Effizienz durch akkumulierenden Parameter

in der Vorlesung:

- Aufrufkeller zeigen

Verständnisfragen:

- Zeigen Sie den Aufwand dieser `Reverse`-Implementierung.

Funktionen höherer Ordnung (Parameter): `map`

Berechnungsschemata mit Funktionen als Parameter

Beispiel: eine Liste elementweise transformieren

```
fun  map(f, nil)  = nil
|    map(f, h::t) = (f h) :: map (f, t);
Signatur: map: (('a ->'b) * 'a list) -> 'b list
```

Anwendungen von **`Map`**, z. B.

```
map (fn i => i*2.5, [1.0,2.0,3.0]);
```
Ergebnis: `[2.5, 5.0, 7.5]`

```
map (fn x => (x,x), [1,2,3]);
```
Ergebnis: `[(1,1), (2,2), (3,3)]`

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 712

Ziele:

Wichtiges Berechnungsmuster für Listenfunktionen

in der Vorlesung:

- Schema und Anwendungen von `Map` zeigen
- Zeigen wie die Typinferenz die allgemeinste polymorphe Signatur von Funktionen bestimmt.

nachlesen:

..., Abschnitt 13.1.1

Funktionen höherer Ordnung (Parameter): `foldl`

`foldl` verknüpft Listenelemente von links nach rechts

`foldl` ist mit akkumulierendem Parameter definiert:

```
fun foldl (f, a, nil) = a
|   foldl (f, a, h::t)= foldl (f, f (a, h), t);
Signatur: foldl: (('b * 'a)->'b * 'b * 'a list) -> 'b
```

Für	`foldl (f, 0, [1, 2, 3, 4])`
wird berechnet	`f(f(f(f(0, 1), 2), 3), 4)`

Anwendungen von `foldl`
assoziative Verknüpfungsfunktion und neutrales Element einsetzen:

```
fun Sum l = foldl (fn (a, h:int) => a+h, 0, l);
```

Verknüpfung: Addition; **`Sum`** addiert Listenelemente

```
fun Reverse l = foldl (fn (a, h) => h::a, nil, l);
```

Verknüpfung: Liste vorne verlängern; **`Reverse`** kehrt Liste um

Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 712a

Ziele:

Wichtiges Berechnungsmuster für Listenfunktionen

in der Vorlesung:

- Schema und Anwendungen von `LInsert` zeigen
- Zeigen wie die Typinferenz die allgemeinste polymorphe Signatur von Funktionen bestimmt.

nachlesen:

..., Abschnitt 13.1.1

Übungsaufgaben:

- Geben Sie ein Schema zur Verknüpfung der Listenelemente von rechts nach links an.

Polynomberechnung mit `foldl`

Ein **Polynom** $a_n x^n + ... + a_1 x + a_0$ sei durch seine **Koeffizientenliste** $[a_n,...,a_1, a_0]$ dargestellt

Berechnung eines Polynomwertes an der Stelle x nach dem Horner-Schema:

$$(...((0 * x + a_n) * x + a_{n-1}) * x + ... + a_1) * x + a_0$$

Funktion **`Horner`** berechnet den Polynomwert für x nach dem Horner-Schema:

```
fun Horner (koeff, x:real) = foldl (fn(a, h)=>a*x+h, 0.0, koeff);
```

Verknüpfungsfunktion `fn(a, h)=>a*x+h` hat freie Variable `x`,
sie ist gebunden als Parameter von **`Horner`**

Aufrufe z. B.

```
Horner ([1.0, 2.0, 3.0], 10.0);
Horner ([1.0, 2.0, 3.0], 2.0);
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 713

Ziele:

Wirksame Anwendung von LInsert

in der Vorlesung:

- Horner-Schema,
- Verknüpfungsfunktion,
- Bindung von x

erläutern

nachlesen:

..., Abschnitt 13.1.1

Funktionen höherer Ordnung (Ergebnis)

Einfaches Beispiel für **Funktion als Ergebnis**:

```
fun Choice true   = (fn x => x + 1)
|   Choice false  = (fn x => x * 2);
```

Signatur `Choice: bool -> (int -> int)`

Meist sind **freie Variable der Ergebnisfunktion** an Parameterwerte der **konstruierenden Funktion** gebunden:

```
fun Comp (f, g) = fn x => f (g x);
```

Hintereinanderausführung von `g` und `f`

Signatur `Comp:('b->'c * 'a->'b) -> ('a->'c)`

Anwendung: z. B. Bildung einer benannten Funktion `Hoch4`

```
val Hoch4 = Comp (Sqr, Sqr);
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 714

Ziele:

Funktionen als Daten errechnen

in der Vorlesung:

Erläuterung der Beispiele

nachlesen:

..., Abschnitt Abschnitt 13.1.3

Currying

Currying: Eine Funktion mit **Parametertupel** wird umgeformt in eine Funktion mit einfachem Parameter und einer **Ergebnisfunktion**; z. B. schrittweise Bindung der Parameter:

	Parametertupel	**Curry-Form**
	`fun Add (x, y:int) = x + y;`	`fun CAdd x = fn y:int => x + y;`
Signatur	`Add: (int * int) -> int`	`CAdd: int -> (int -> int)`

In Aufrufen müssen alle Parameter(komponenten) sofort angegeben werden

```
Add (3, 5)
```

können die Parameter schrittweise gebunden werden:

```
(CAdd 3) 5
```

Auch **rekursiv**:

```
fun CPower n = fn b =>
   if n = 0 then 1.0 else b * CPower (n-1) b;
```

Signatur `CPower: int -> (real -> real)`

Anwendung:

```
val Hoch3 = CPower 3;
```

eine Funktion, die „hoch 3“ berechnet

```
(Hoch3 4)          liefert 64
((CPower 3) 4)     liefert 64
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 715

Ziele:

Prinzip Currying verstehen

in der Vorlesung:

Erläuterung des Curry-Prinzips

- an Signaturen
- an Funktionen
- an Anwendungen

nachlesen:

..., Abschnitt Abschnitt 13.1.3

Übungsaufgaben:

- Wenden Sie Currying auf weitere Funktionen an.

Verständnisfragen:

- Erläutern Sie am Beispiel `CHorner`: Parameterdaten werden in der Ergebnisfunktion gebunden.

Kurzschreibweise für Funktionen in Curry-Form

Langform:

```
fun CPower n = fn b =>
   if n = 0 then 1.0 else b * CPower (n-1) b;
```

Signatur `CPower: int -> (real -> real)`

Kurzschreibweise für Funktion in Curry-Form:

```
fun CPower n b =
   if n = 0 then 1.0 else b * CPower (n-1) b;
```

Funktion **`Horner`** berechnet den Polynomwert für x nach dem Horner-Schema (GPS-7.13), in Tupelform:

```
fun Horner (koeff, x:real) = foldl (fn(a, h)=>a*x+h, 0.0, koeff);
```

Horner-Funktion in Curry-Form:
`CHorner` liefert eine Funktion; die Koeffizientenliste ist darin gebunden:

```
fun CHorner koeff x:real = foldl (fn(a, h)=>a*x+h, 0.0, koeff);
```

Signatur `CHorner: (real list) -> (real -> real)`

```
Aufruf:  val MyPoly = CHorner [1.0, 2.0,3.0];
         ...
         MyPoly 10.0
```


Vorlesung Grundlagen der Programmiersprachen SS 2014 / Folie 715a

Ziele:

Kurzschreibweise für Currying

in der Vorlesung:

Erläuterung

- der Kurzschreibweise,
- der Horner-Funktion,
- schrittweise Bindung der Parameter

nachlesen:

..., Abschnitt Abschnitt 13.1.3

Verständnisfragen:

- Erläutern Sie am Beispiel `CHorner`: Parameterdaten werden in der Ergebnisfunktion gebunden.

Zusammenfassung zum Kapitel 7

Mit den Vorlesungen und Übungen zu Kapitel 7 sollen Sie nun Folgendes können:

- Funktionale Programme unter Verwendung treffender Begriffe präzise erklären
- Funktionen in einfacher Notation von SML lesen und schreiben
- Rekursionsparadigmen Induktion, Rekursion über Listen anwenden
- End-Rekursion erkennen und Programmiertechnik „akkumulierender Parameter“ anwenden
- Berechnungsschemata mit Funktionen als Parameter anwenden
- Programmiertechnik „Currying“ verstehen und anwenden

Ziele:

Ziele des Kapitels erkennen

in der Vorlesung:

Erläuterungen dazu

8. Logische Programmierung

Themen dieses Kapitels:

- Prolog-Notation und kleine Beispiele
- prädikatenlogische Grundlagen
- Interpretationsschema
- Anwendbarkeit von Klauseln, Unifikation
- kleine Anwendungen

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 801

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

Übersicht zur logischen Programmierung

Deklaratives Programmieren:
Problem beschreiben statt Algorithmus implementieren (idealisiert).
Das System findet die Lösung selbst, z. B. Sortieren einer Liste:

```
sort(old, new) <= permute(old, new) ∧ sorted(new)
sorted(list) <= ∀j such that 1 <= j < n: list(j) <= list(j+1)
```

Relationen bzw. Prädikate (statt Funktionen):

```
(a, b)∈ R ⊆ (S x T)
magEssen(hans, salat)
```

Programmkonstrukte entsprechen eingeschränkten **prädikatenlogischen Formeln**

```
∀X,Y,Z: grossMutterVon(X, Z) <= mutterVon(X, Y)∧elternteilVon(Y, Z)
```

Resolution implementiert durch Interpretierer:
Programm ist Menge von PL-Formeln,
Interpretierer sucht Antworten (erfüllende Variablenbelegungen) durch **Backtracking**

```
?-sort([9,4,6,2], X).        Antwort:      X = [2,4,6,9]
```

Datenmodell: strukturierte **Terme mit Variablen** (mathematisch, nicht imperativ);
Bindung von Termen an Variable durch **Unifikation**

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 802

Ziele:

Einstimmung auf das Programmiermodell

in der Vorlesung:

- Hinweis auf Besprechung der Grundbegriffe in der Vorlesung "Modellierung"
- Rolle der Grundbegriffe in der logischen Programmierung

nachlesen:

..., Abschnitt Modellierung Kap.4.2

Verständnisfragen:

- Begründen Sie: Ein Prädikat definiert eine Relation.

Prolog Übersicht

Wichtigste logische Programmiersprache: Prolog (Colmerauer, Roussel, 1971)

Typische Anwendungen: Sprachverarbeitung, Expertensysteme, Datenbank-Management

Ein Programm ist eine **Folge von Klauseln** (Fakten, Regeln, eine Anfrage) formuliert über Terme.

```
mother(mary, jake).
mother(mary, shelley).
father(bill, jake).
```

Fakten

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Regeln

```
?- parent(X, jake)
```

Anfrage

Antworten:
```
X = mary
X = bill
```

Ein **Interpretierer** prüft, ob Werte an die Variablen so gebunden werden können, dass die Anfrage mit den gegebenen Prädikaten und Regeln erfüllbar ist (Resolution).

Es wird ein **universelles Suchverfahren (Backtracking)** angewendet (Folie GPS-8-7).

Ziele:

Übersicht

in der Vorlesung:

- Erste Prolog-Beispiele

Prolog Sprachkonstrukte: Fakten

Fakten geben Elemente von **n-stelligen Relationen** bzw. **Prädikaten** an, z. B.

```
stern(sonne).
stern(sirius).
```

bedeutet, **sonne** und **sirius** sind Konstante,
sie erfüllen das Prädikat (die 1-stellige Relation) **stern**.

Einige Fakten, die Elemente der 2-stelligen Relation **umkreist** angeben:

```
umkreist(jupiter, sonne).
umkreist(erde, sonne).
umkreist(mars, sonne).
umkreist(mond, erde).
umkreist(phobos, mars).
```

Fakten können auch mit Variablen formuliert werden:

```
istGleich(X,X).
```

bedeutet in PL: $\forall$ **X: istGleich(X,X)**

Prolog hat **keine Deklarationen**. **Namen** für Prädikate, Konstante und Variablen werden **durch ihre Benutzung eingeführt**.
Namen für Konstante beginnen mit kleinem, für Variable mit großem Buchstaben.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 804

Ziele:

Fakten und Relationen kennenlernen

in der Vorlesung:

An den Beispielen

- die Bedeutung der Konstrukte,
- Definition von Relationen

erläutern.

Prolog Sprachkonstrukte: Regeln

Regeln definieren **n-stellige Relationen** bzw. **Prädikate**
durch **Implikationen** (intensional), z. B.

```
planet(B) :- umkreist(B, sonne).
satellit(B) :- umkreist(B, P), planet(P).
```

bedeutet in PL:

$\forall$ `B: planet(B)<= umkreist(B, sonne)`
$\forall$ `B,P: satellit(B) <= umkreist(B, P)` $\wedge$ `planet(P)`

In einer Klausel müssen an alle Vorkommen eines Variablennamen dieselben Werte gebunden sein, z. B. **`B/mond`** und **`P/erde`**

Allgemein definiert man eine Relation durch **mehrere Fakten und Regeln**.
sie gelten dann alternativ (oder-Verknüpfung)

```
sonnensystem(sonne).
sonnensystem(B) :- planet(B).
sonnensystem(B) :- satellit(B).
```

Man kann Relationen auch **rekursiv definieren**:

```
sonnensystem(sonne).
sonnensystem(X) :- umkreist(X, Y), sonnensystem(Y).
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 804a

Ziele:

Regeln zur Definition von Relationen verstehen

in der Vorlesung:

An den Beispielen erläutern:

- die Struktur von Regeln,
- die Bedeutung als Implikation,
- Definition von Relationen durch Regeln und Fakten,
- rekursive Definitionen.

Prolog Sprachkonstrukte: Anfragen

Das Prolog-System überprüft, ob eine **Anfrage mit den Fakten und Regeln** des gegebenen Programms (durch prädikatenlogische Resolution) **als wahr nachgewiesen** werden kann.

Beispiele zu den Fakten und Regeln der vorigen Folien:

	Antwort:
`?- umkreist(erde, sonne).`	**yes**
`?- umkreist(mond, sonne).`	**no**

Eine Anfrage `?- umkreist(mond, B).`
bedeutet in PL $\exists$`B: umkreist(mond, B)`

Wenn die **Anfrage Variablen** enthält, werden **Belegungen** gesucht, mit denen die Anfrage als wahr nachgewiesen werden kann:

	Antworten:
`?- umkreist(mond, B).`	**B=erde**
`?- umkreist(B, sonne).`	**B=jupiter; B=erde; B=mars**
`?- umkreist(B, jupiter).`	**no** (keine Belegung ableitbar)
`?- satellit(mond).`	**yes**
`?- satellit(S).`	**S=mond; S=phobos**

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 804b

Ziele:

Bedeutung von Anfragen verstehen

in der Vorlesung:

An den Beispielen erläutern:

- Notation von Anfragen,
- Anfragen mit Fakten und Regeln prüfen,
- gibt es Werte, sodass die Anfrage als wahr nachgewiesen werden kann?

Verständnisfragen:

- Begründen Sie die Antworten auf die Anfragen.
- Warum liefert `umkreist(B,jupiter)` die Antwort `no`, obwohl es Jupitermonde gibt?

Notation von Prolog-Programmen

Beliebige Folge von **Klauseln**: **Fakten**, **Regeln** und **Anfragen** (am Ende).
Klauseln mit **Prädikaten** `p(`t_1`, ..., `t_n`)`, Terme t_i

Terme sind beliebig zusammengesetzt aus Literalen, Variablen, Listen, Strukturen.

- **Literale** für Zahlen, Zeichen(reihen) `127 "text" 'a'`
- **Symbole** (erste Buchstabe klein) `hans`
- **Variablen** (erste Buchstabe groß) `X Person`
  unbenannte Variable `_`
- **Listen**-Notation: `[a, b, c]  []`
  erstes Element H, Restliste T `[H | T]` wie `H::T` in SML
- **Strukturen**: `kante(a, b)  a - b  datum(T, M, J)`
  Operatoren `kante, -` werden
  ohne Definition verwendet, nicht „ausgerechnet“

Grundterm: Term ohne Variablen, z. B. `datum(11, 7, 1995)`

Prolog ist **nicht typisiert**:

- An eine Variable können beliebige Terme gebunden werden,
- an Parameterpositionen von Prädikaten können beliebige Terme stehen.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 804c

Ziele:

Einfache Prolog-Programme schreiben können

in der Vorlesung:

Erläuterung der Notation an Beispielen

Prädikatenlogische Grundlagen

Prädikatenlogische Formeln (siehe Modellierung, Abschn. 4.2):

atomare Formeln p (t_1, ..., t_n) bestehen aus einem Prädikat p und Termen t_i mit Variablen, z. B. `last([X], X)`
darauf werden logische Junktoren (¬ ∧ ∨) und Quantoren (∀ ∃) angewandt,

z. B. ∀X ∀Y: **`sonnensystem(X)`** ∨ ¬ **`umkreist(X, Y)`** ∨ ¬ **`sonnensystem(Y)`**
äquivalent zu
∀X ∀Y: **`sonnensystem(X)`** ⇐ **`umkreist(X, Y)`** ∧ **`sonnensystem(Y)`**

Allgemeine PL-Formeln werden auf die 3 Formen von Prolog-Klauseln (Horn-Klauseln) eingeschränkt, z. B.

Prolog-Fakt: **`last ([X], X).`**
PL: ∀ **X:** **`last ([X], X).`**

Prolog-Regel: **`sonnensystem(X) :- umkreist(X,Y), sonnensystem(Y).`**
PL: ∀X ∀Y: **`sonnensystem(X)<=umkreist(X,Y)`**∧**`sonnensystem(Y).`**

Prolog-Anfrage: **`umkreist(X, erde), umkreist(X,jupiter).`**
PL: ∃X: **`umkreist(X, erde)`** ∧ **`umkreist(X,jupiter).`**

äquivalent zu: ¬∀X ¬**`umkreist(X, erde)`** ∨ ¬**`umkreist(X,jupiter).`**

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 805

Ziele:

Prolog-Klauseln als prädikatenlogische Formeln verstehen

in der Vorlesung:

Erläuterung

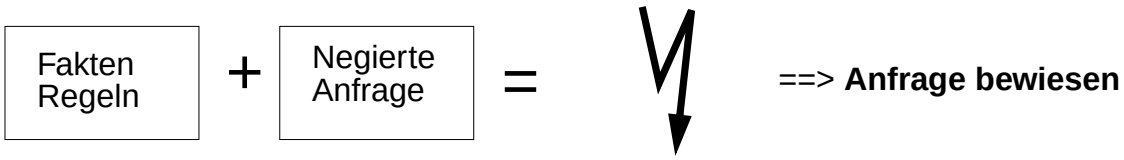
- Regeln, Fakten, Anfragen in prädikatenlogische Formeln transformieren
- Hornklauseln: eingeschränkte PL-Formeln

nachlesen:

..., Abschnitt Modellierung Kap. 4.2

Resolution

Resolution führt einen **Widerspruchsbeweis** für eine Anfrage:



Prolog-Anfrage: `umkreist(X, erde), umkreist(X,jupiter).`
PL: $\exists$`X: umkreist(X, erde)` $\wedge$ `umkreist(X,jupiter).`
äquivalent zu: $\neg\forall$`X` $\neg$`umkreist(X, erde)` $\vee$ $\neg$`umkreist(X,jupiter).`

negiert: $\forall$`X` $\neg$`umkreist(X, erde)` $\vee$ $\neg$`umkreist(X,jupiter).`

Die Antwort ist gültig für **alle** zu einem Programm durch induktive Anwendung von Operatoren **konstruierbaren Terme** (Herbrand-Universum, „Hypothese der abgeschlossenen Welt“).

Antwort Ja: Aussage ist mit den vorhandenen Fakten und Regeln beweisbar.
Antwort Nein: Aussage ist mit den gegebenen Fakten und Regeln nicht beweisbar. Das heißt nicht, dass sie falsch ist.

Daher kann eine Negation, wie in
Formel F gilt, wenn Formel H **nicht** gilt
in Prolog-Systemen nicht ausgedrückt werden.
Der vordefinierte Operator **`not`** ist „nicht-logisch“ und mit Vorsicht zu verwenden.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 806

Ziele:

Formulierung des Widerspruchsbeweises verstehen

in der Vorlesung:

Erläuterung

- Negierung der Anfrage;
- Resolution konstruiert Belegungen als Gegenbeispiel;
- sie sind Antworten auf die Anfrage.

nachlesen:

..., Abschnitt Modellierung Kap 4.2

Interpretationsschema Backtracking

Aus Programm mit Fakten, Regeln und Anfrage spannt der Interpretierer einen **abstrakten Lösungsbaum** auf (Beispiel auf nächster Folie):

Wurzel: Anfrage
Knoten: Folge noch zu verifizierender Teilziele
Kanten: anwendbare Regeln oder Fakten des Programms

Der Interpretierer iteriert folgende Schritte am aktuellen Knoten:

- **Wähle ein noch zu verifizierendes Teilziel** (Standard: von links nach rechts)
  Falls die Folge der Teilziele leer ist, wurde eine Lösung gefunden (success);
  ggf. wird nach weiteren gesucht: backtracking zum vorigen Knoten.
- **Wähle eine auf das Teilziel anwendbare Klausel** (Standard: Reihenfolge im Programm);
  bilde einen neuen Knoten, bei dem das Teilziel durch die rechte Seite der Regel bzw. bei einem Fakt durch nichts ersetzt wird; weiter mit diesem neuen Knoten.
  Ist keine Klausel anwendbar, gibt es in diesem Teilbaum keine Lösung: backtracking zum vorigen Knoten.

Bei rekursiven Regeln, z.b: **`nachbar(A, B) :- nachbar(B, A)`**
ist der **Baum nicht endlich.** Abhängig von der **Suchstrategie terminiert** die Suche dann eventuell **nicht**.

Die Reihenfolge, in der die Wahl (s.o.) getroffen wird, ist entscheidend für die **Terminierung** der Suche und die Reihenfolge, in der Lösungen gefunden werden!

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 807

Ziele:

Das Interpretationsmodell am Lösungsbaum verstehen

in der Vorlesung:

- Struktur des Lösungsbaums (Folie 808),
- schrittweise Erstellung des Lösungsbaums,
- Bedeutung der Suchreihenfolge für die Terminierung

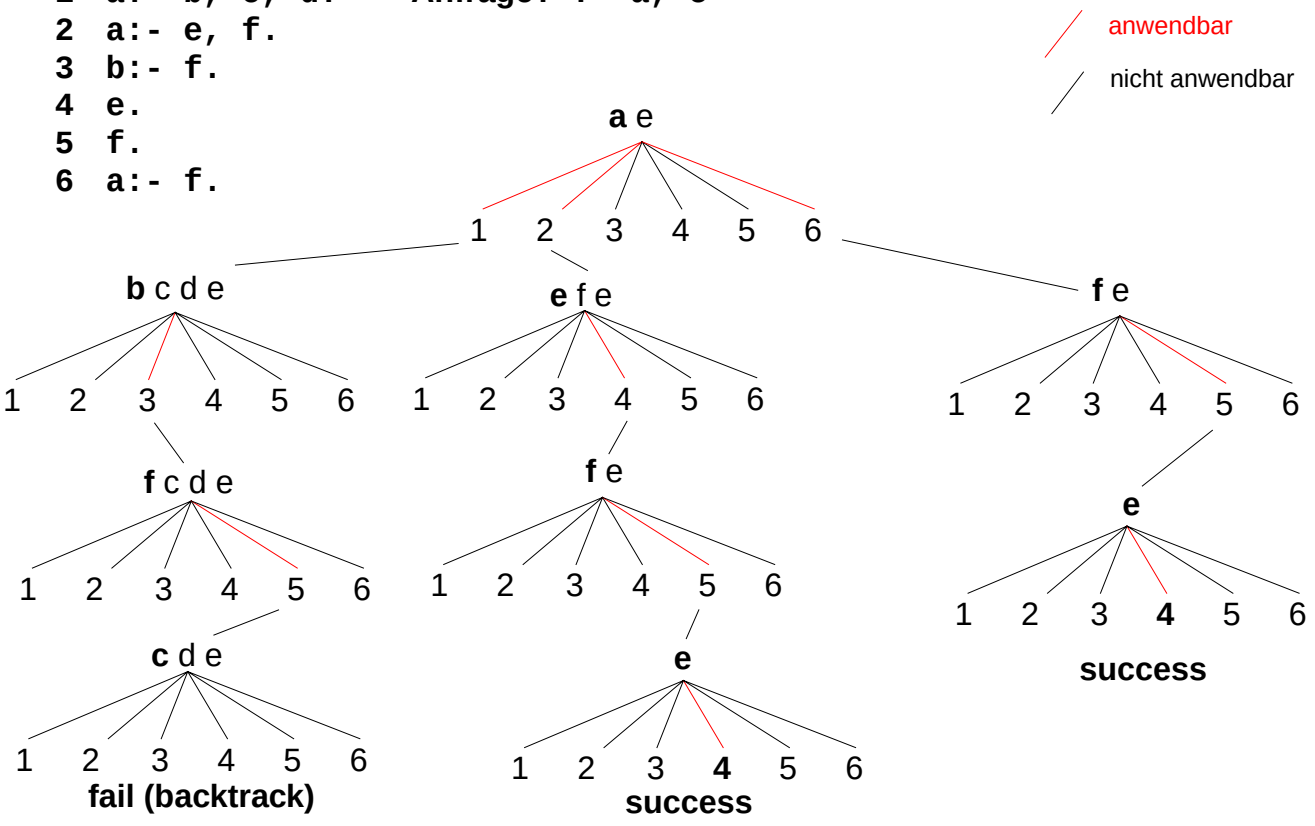
Verständnisfragen:

- Welche zwei Freiheitsgrade gibt es bei der Suche im Lösungsbaum? Wie legt sie der Standardinterpretierer fest?
- Welche Eigenschaften hat ein Lösungsbaum, wenn die Suche in Standardreihenfolge nicht terminiert, obwohl eine Lösung existiert?
- Geben Sie dazu die Struktur eines Programms in der Notation wie auf der Folie an.

Lösungsbaum Beispiel

Beispiel (**a, b,** ... stehen für Prädikate; Parameterterme sind hier weggelassen):

```
1  a:- b, c, d.      Anfrage: ?- a, e
2  a:- e, f.
3  b:- f.
4  e.
5  f.
6  a:- f.
```



Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 808

Ziele:

Veranschaulichung zum Interpretationsschema

in der Vorlesung:

Erläuterungen zusammen mit Folie 807

Anwendung von Klauseln

In Klauseln werden **Terme als Muster** verwendet.
Darin vorkommende Variablennamen müssen konsistent an Terme gebunden werden:

`last([X], X).` `[X]` Muster für eine einelementige Liste
`heuer(T, M, datum(T, M, 2013)).` Muster für ein `datum` mit bestimmten Teiltermen

Eine Klausel (Fakt oder linke Seite einer Regel) ist auf ein Teilziel anwendbar,
wenn es einen **Unifikator** gibt, der die **Parameterterme der Klausel und des Teilziels paarweise gleich macht**:

Fakt: `heuer(T, M, datum(T, M, 2013)).`
Anfrage: `?-heuer(12, 7, Z).` Unifikator: `[T/12,M/7, Z/datum(12,7,2013)]`

Fakt: `heuer(T, M, datum(T, M, 2013)).`
Anfrage: `?-heuer(X, Y, datum(14, 7, 2013).`
Unifikator: `[X/14,T/14,Y/7, M/7]`

Regel: `last([_|T], Y):- last(T, Y).`
Teilziel: `last([2,3], Z)` Unifikator: `[T/[3], Y/Z]`

Fakt: `last([X], X).`
Teilziel: `last([2,3], Z)` nicht unifizierbar, also nicht anwendbar

Wird die Klausel angewandt, werden die **Variablen gemäß Unifikator gebunden**.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 812

Ziele:

Anwendung von Klauseln verstehen

in der Vorlesung:

Erläuterung der Beispiele:

- Unifikation entscheidet über Anwendbarkeit,
- Unifikation bindet Terme an Variablen

Unifikation

siehe Modellierung, Kap. 3.1
Term: Formel bestehend aus Literalen, Variablen, Operatoren, Funktoren; z. B. `X + f(2*Y)`

Substitution `s = [X1/e1, ..., Xn/en]` angewandt auf `T`, geschrieben `T s` bedeutet:
alle Vorkommen der Variablen X_i in **T** werden gleichzeitig durch den Term e_i ersetzt.
z. B. `Y+Y [Y/3*Z]` ergibt `3*Z+3*Z`

Unifikation: Allgemeines Prinzip: Terme durch Substitution gleich machen.
gegeben: zwei Terme T_1, T_2
gesucht: eine Substitution **U**, sodass gilt T_1 `U` = T_2 `U`. Dann ist **U** ein **Unifikator** für **T1** und **T2**.

Beispiele:

```
datum(T, M, 2011)
datum (14, 7, 2011)

U =  [T/14, M/7]
```

```
X+f(2*g(1))
3+f(2*Y)

U = [X/3, Y/g(1)]
```

```
f(h(a, b), g(Y),    V)
f(X,       g(h(a,c)), Z)
```

allgemeinste Unifikatoren:

```
Ua = [X/h(a,b), Y/h(a,c), V/Z]

Ua = [X/h(a,b), Y/h(a,c), Z/V]
```

nicht-allgemeinster Unifikator,
unnötige Bindungen an V und Z:

```
U = [X/h(a,b), Y/h(a,c), V/a, Z/a]
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 812a

Ziele:

Prinzip der Unifikation wiederholen

in der Vorlesung:

Wiederholung der Begriffe

- Terme
- Bindung von Variablen durch Substitution
- Unifikation

nachlesen:

Skript zu Modellierung, Kap. 3.1

Verständnisfragen:

- Vergleichen Sie: Bindung durch Unifikation und Bindung durch Muster in SML.

Rekursive Anwendung von Klauseln

Variable sind lokal für jede Anwendung einer Klausel.

Bei **rekursiven Anwendungen** entstehen **neue lokale Variable.**

Mehrfache Auftreten einer Variable stehen für denselben Wert.

Beispiel: mit folgenden Klauseln

```
(1)last([X], X).
(2)last([_|T], Y):- last(T, Y).
```

wird die Anfrage berechnet:

```
     ?-last([1,2,3], Z).
(2)    last([_|T1],  Y1):- last([2,3], Z).
(2)                        last([_|T2],Y2):- last([3], Z).
(1)          T1 = [2,3]                      last([X], X). bindet Z=3
             Y1 = Z            T2 = [3]
                               Y2 = Z              X = 3
                                                   X = 3 = Z
```


Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 813

Ziele:

Rekursive Anwendung von Klauseln verstehen

in der Vorlesung:

- Erläuterung des Beispiels,
- Unifikation mit neuen lokalen Variablen T1, T2, Y1, Y2.

Beispiel: Wege im gerichteten Graph

Das folgende kleine Prolog-Programm beschreibt die Berechnung von Wegen in einem gerichteten Graph.

Die Menge der gerichteten Kanten wird durch eine Folge von Fakten definiert:

```
kante(a,b).
kante(a,c).
...
```

Die Knoten werden dabei implizit durch Namen von Symbolen eingeführt.
Die Relation **weg(X,Y)** gibt an, ob es einen Weg von **X** nach **Y** gibt:

```
weg(X, X).                          Weg der Länge 0
weg(X, Y):-kante(X, Y).             Weg der Länge 1
weg(X, Y):-kante(X, Z), weg(Z, Y).    weitere Weg e
```

Anfragen:

?-weg(a,c).	prüft, ob es einen Weg von **a** nach **c** gibt.
?-weg(a,X).	sucht alle von a erreichbaren Knoten.
?-weg(X,c).	sucht alle Knoten, von denen **c** erreichbar ist .

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 813a

Ziele:

Einige einfache Techniken kennenlernen

in der Vorlesung:

- Definition eines Graphen.
- Rekursive Wegesuche.
- Verschiedene Möglichkeiten von Anfragen.

Beispiel: Symbolische Differentiation

Das folgende Prolog-Programm beschreibt einige einfache Regeln zur Differentiation.
Sie werden auf Terme angewandt, die Ausdrücke beschreiben, und liefern die Ableitung in Form eines solchen Terms, z. B. **`?-diff(2*x,x,D).`** liefert z. B. **`D = 2*1+x*0.`**
Mit weiteren Regeln zur Umformung von Ausdrücken kann das Ergebnis noch vereinfacht werden.

In Prolog werden Ausdrücke wie **`2*x` nicht ausgewertet** (sofern nicht durch **`IS`** explizit gefordert), sondern als Struktur dargestellt, also etwa **`*(2, x)`**.

Prolog-Regeln zur Symbolischen Differentiation:

```
diff(X, X, 1):- !.
diff(T, X, 0):- atom(T).
diff(T, X, 0):- number(T).

diff(U+V, X, DU+DV):- diff(U, X, DU), diff(V, X, DV).

diff(U-V, X, DU-DV):- diff(U, X, DU), diff(V, X, DV).

diff(U*V, X, (U*DV)+(V*DU)):- diff(U, X, DU), diff(V, X, DV).

diff(U/V, X, ((V*DU)-(U*DV))/V*V):- diff(U, X, DU), diff(V, X, DV).
```

Falls die erste Regel anwendbar ist, bewirkt der **Cut** (**!**), dass bei beim Backtracking keine Alternative dazu versucht wird, obwohl die nächsten beiden Klauseln auch anwendbar wären.

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 814

Ziele:

Eine typische Prolog-Anwendung kennenlernen

in der Vorlesung:

- Erläuterungen zu den Regeln und zum Cut-Operator
- Ein Beispiel zeigen

Übungsaufgaben:

- Ergänzen Sie das Programm durch Regeln zur algebraischen Vereinfachung von Ausdrücken.

Verständnisfragen:

- Erläutern Sie diese Definition von `diff` als Relation (nicht als Funktion). Geben Sie Anfragen als Beispiele dafür an.

Erläuterungen zur Symbolischen Differentiation

1. Hier werden Terme konstruiert, z. B. zu **`2*x`** der Term **`2*1+x*0`**
   Ausrechnen formuliert man in Prolog durch spezielle IS-Klauseln:
   `dupl(X,Y):- Y IS X*2.` **`X`** muss hier eine gebundene Variable sein.
2. Problemnahe Beschreibung der Differentiationsregeln, z. B. Produktregel:

$$\frac{d(u*v)}{d\,x} = u * \frac{d\,v}{d\,x} + v * \frac{d\,u}{d\,x}$$

3. **`diff`** ist definert als Relation über 3 Terme:
   `diff` (abzuleitende Funktion, Name der Veränderlichen, Ableitung)
4. Muster in Klauselkopf legen die Anwendbarkeit fest, z. B. Produktregel:
   `diff(U*V, X, (U*DV)+(V*DU)):-` ...
5. Regeln 1 - 3 definieren: $\frac{d\,x}{d\,x} = 1$ $\frac{d\,a}{d\,x} = 0$ $\frac{d\,1}{d\,x} = 0$

   !-Operator (Cut) vermeidet falsche Alternativen.
6. **`diff`** ist eine Relation - nicht eine Funktion!!
   ?-**`diff(a+a,a,D).`** liefert **`D = 1 + 1`**
   ?-**`diff(F,a,1+1).`** liefert **`F = a + a`**

Vorlesung Grundlagen der Programmiersprachen SS 2013 / Folie 815

Ziele:

Differentiations-Programm verstehen

in der Vorlesung:

Erläuterungen dazu

Beispielrechnung zur Symbolischen Differentiation

```
?- diff(2*y, y,  D)
   diff(U*V, X1,(2*DV)+(y*DU)):- diff(2, y, DU),   diff(y, y, DV)
                                 diff(T1, X2, 0)   diff(X3, X3, 1)
                                     :-number(2)       :- !
                                       success           success
```

liefert Bindungen **DU=0 DV=1 D=(2*1)+(y*0)**

Das Programm kann systematisch erweitert werden, damit Terme nach algebraischen Rechenregeln vereinfacht werden, z. B.

```
simp(X*1, X).   simp(X+0, X).
simp(X*0, 0).   simp(X-0, X).
...
```

So einsetzen, dass es auf alle Teilterme angewandt wird.

Ziele:

Anwendung des Programms verstehen

in der Vorlesung:

Erläuterungen dazu

Zusammenfassung zum Kapitel 8

Mit den Vorlesungen und Übungen zu Kapitel 8 sollen Sie nun Folgendes können:

- Kleine typische Beispiele in Prolog-Notation lesen, verstehen und schreiben
- Interpretationsschema und prädikatenlogische Grundlagen verstehen
- Unifikation zum Anwenden von Klauseln einsetzen

Ziele:
Ziele des Kapitels erkennen

in der Vorlesung:
Erläuterungen dazu

9. Zusammenfassung

Themen dieses Kapitels:

- Zusammenfassung der Themen der Vorlesung
- Zusammenfassung der angestrebten Kompetenzen

Vorlesung Grundlagen der Programmiersprachen SS 2009 / Folie 901

Ziele:
Übersicht zu diesem Kapitel

in der Vorlesung:
Erläuterungen dazu

Zusammenfassung der behandelten Themen

allgemeine Spracheigenschaften:

- Grundsymbole, Syntax, statische Semantik, dynamische Semantik
- kontext-freie Grammatik, Ableitungsbäume, EBNF-Notation, Ausdruckgrammatiken
- Gültigkeit von Definitionen, Verdeckungsregeln
- Variablenbegriff, Lebensdauer, Laufzeitkeller, statischer Vorgänger, Umgebungen
- Datentypen, abstrakte Typkonstruktoren, rekursive und parametrisierte Typen
- konkrete Ausprägungen der abstrakten Typkonstruktoren in Programmiersprachen
- Parameterübergabe: call-by-value, call-by-reference, call-by-result, call-by-value-and-result

Funktionale Programmierung:

- Rekursionsparadigmen: Induktion, Funktionen über rekursiven Datentypen
- Rekursionsformen: End-Rekursion, Zentral-Rekursion,
- Technik „akkumulierender Parameter“, Funktionen über Listen
- Berechnungsschemata mit Funktionen als Parameter; Currying

Logische Programmierung:

- Klauselformen: Fakt, Regel, Anfrage; prädikatenlogische Bedeutung
- Interpretationsschema: Backtracking, Suchreihenfolge
- Unifikation von Termen: Anwendbarkeit von Klauseln, Bindung von Werten an Variable
- Prolog-Notation

Vorlesung Grundlagen der Programmiersprachen SS 2009 / Folie 903

Ziele:

Erinnerung an die behandelten Themen

in der Vorlesung:

Rat geben zur Nachbereitung der Vorlesung und Vorbereitung auf die Klausur.

Zusammenfassung der angestrebten Kompetenzen (1)

1. Einführung

- Wichtige Programmiersprachen zeitlich einordnen
- Programmiersprachen klassifizieren
- Sprachdokumente zweckentsprechend anwenden
- Sprachbezogene Werkzeuge kennen
- Spracheigenschaften und Programmeigenschaften in die 4 Ebenen einordnen

2. Syntax

- Notation und Rolle der Grundsymbole kennen.
- Kontext-freie Grammatiken für praktische Sprachen lesen und verstehen.
- Kontext-freie Grammatiken für einfache Strukturen selbst entwerfen.
- Schemata für Ausdrucksgrammatiken, Folgen und Anweisungsformen anwenden können.
- EBNF sinnvoll einsetzen können.
- Abstrakte Syntax als Definition von Strukturbäumen verstehen.

Ziele:

Erinnerung an die Ziele der Kapitel

in der Vorlesung:

Erläuterungen dazu

Zusammenfassung der angestrebten Kompetenzen (2)

3. Gültigkeit von Definitionen

- Bindung von Bezeichnern verstehen
- Verdeckungsregeln für die Gültigkeit von Definitionen anwenden
- Grundbegriffe in den Gültigkeitsregeln von Programmiersprachen erkennen

4. Variable, Lebensdauer

- Variablenbegriff und Zuweisung
- Zusammenhang zwischen Lebensdauer von Variablen und ihrer Speicherung
- Prinzip des Laufzeitkellers
- Besonderheiten des Laufzeitkellers bei geschachtelten Funktionen

Ziele:
Erinnerung an die Ziele der Kapitel

in der Vorlesung:
Erläuterungen dazu

Zusammenfassung der angestrebten Kompetenzen (3)

5. Datentypen

5.1 Allgemeine Begriffe zu Datentypen

- Typeigenschaften von Programmiersprachen verstehen und mit treffenden Begriffen korrekt beschreiben
- Mit den abstrakten Konzepten beliebig strukturierte Typen entwerfen
- Parametrisierung und generische Definition von Typen unterscheiden und anwenden

5.2 Datentypen in Programmiersprachen

- Ausprägungen der abstrakten Typkonzepte in den Typen von Programmiersprachen erkennen
- Die Begriffe Klassen, Typen, Objekte, Werte sicher und korrekt verwenden
- Die Vorkommen von Typkonzepten in wichtigen Programmiersprachen kennen
- Speicherung von Reihungen verstehen

Vorlesung Grundlagen der Programmiersprachen SS 2009 / Folie 906

Ziele:

Erinnerung an die Ziele der Kapitel

in der Vorlesung:

Erläuterungen dazu

Zusammenfassung der angestrebten Kompetenzen (4)

6. Funktionen, Parameterübergabe

- Funktionen, Aufrufen und Parameterübergabe präzise mit treffenden Begriffen erklären können
- Die Arten der Parameterübergabe unterscheiden und sinnvoll anwenden können
- Die Parameterübergabe wichtiger Sprachen kennen

7. Funktionale Programmierung

- Funktionale Programme unter Verwendung treffender Begriffe präzise erklären
- Funktionen in einfacher Notation von SML lesen und schreiben
- Rekursionsparadigmen Induktion, Rekursion über Listen anwenden
- End-Rekursion erkennen und Programmiertechnik „akkumulierender Parameter“ anwenden
- Berechnungsschemata mit Funktionen als Parameter anwenden
- Programmiertechnik „Currying“ verstehen und anwenden

Ziele:

Erinnerung an die Ziele der Kapitel

in der Vorlesung:

Erläuterungen dazu

Zusammenfassung der angestrebten Kompetenzen (5)

8. Logische Programmierung

- Kleine typische Beispiele in Prolog-Notation lesen, verstehen und schreiben
- Interpretationsschema und prädikatenlogische Grundlagen verstehen
- Unifikation zum Anwenden von Klauseln einsetzen
- Anwendungen wie die Symbolische Differentiation verstehen

Ziele:
Erinnerung an die Ziele der Kapitel

in der Vorlesung:
Erläuterungen dazu