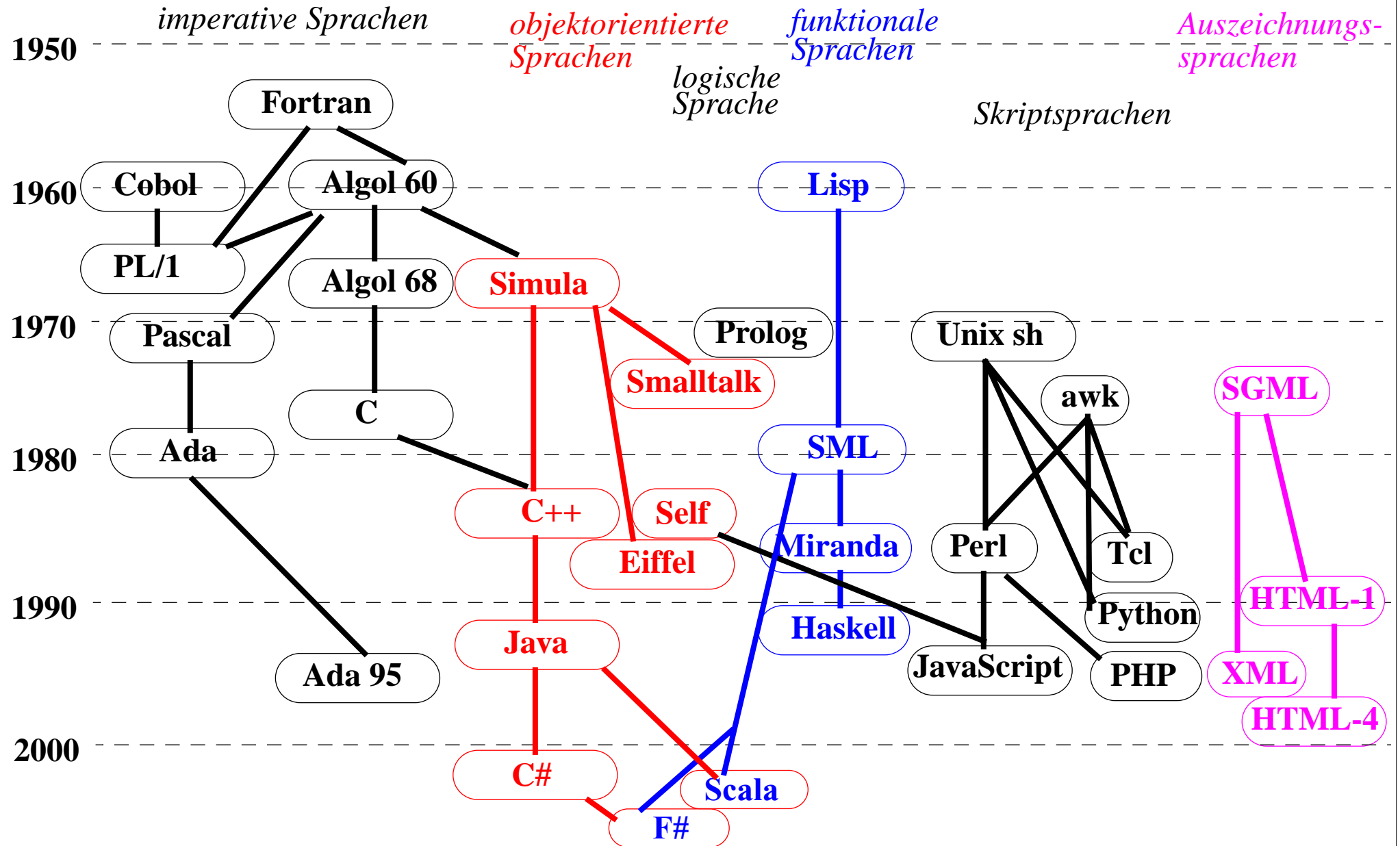


1. Einführung

Themen dieses Kapitels:

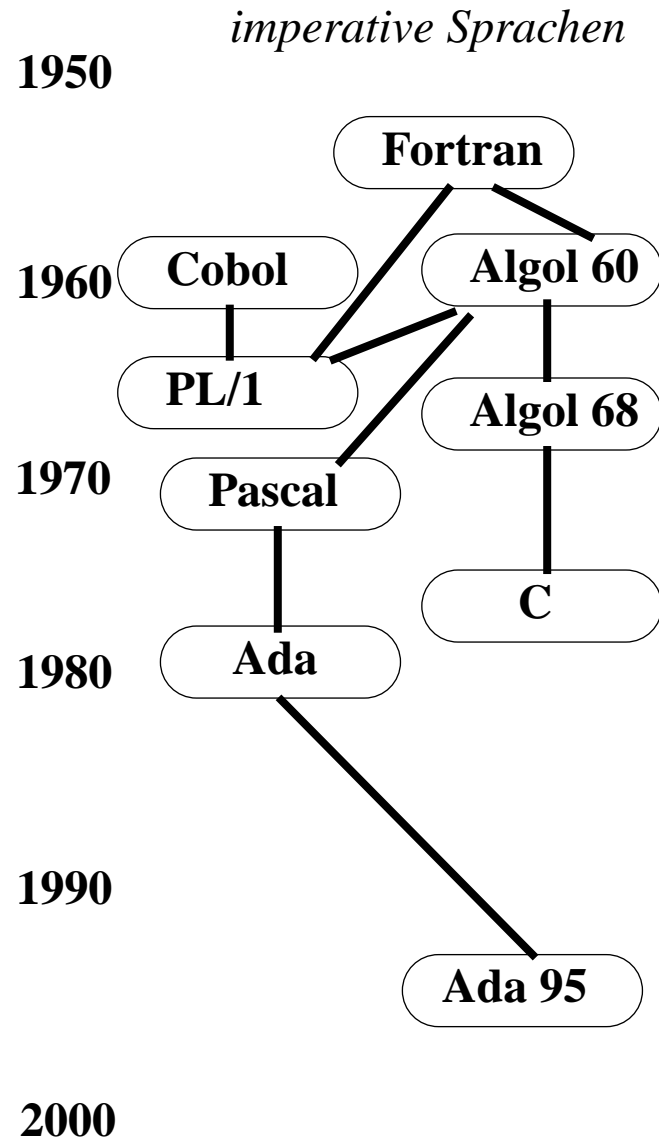
- 1.1. Zeitliche Einordnung, Klassifikation von Programmiersprachen
- 1.2. Implementierung von Programmiersprachen
- 1.3. Dokumente zu Programmiersprachen
- 1.4. Vier Ebenen der Spracheigenschaften

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>]

Klassifikation: Imperative Programmiersprachen

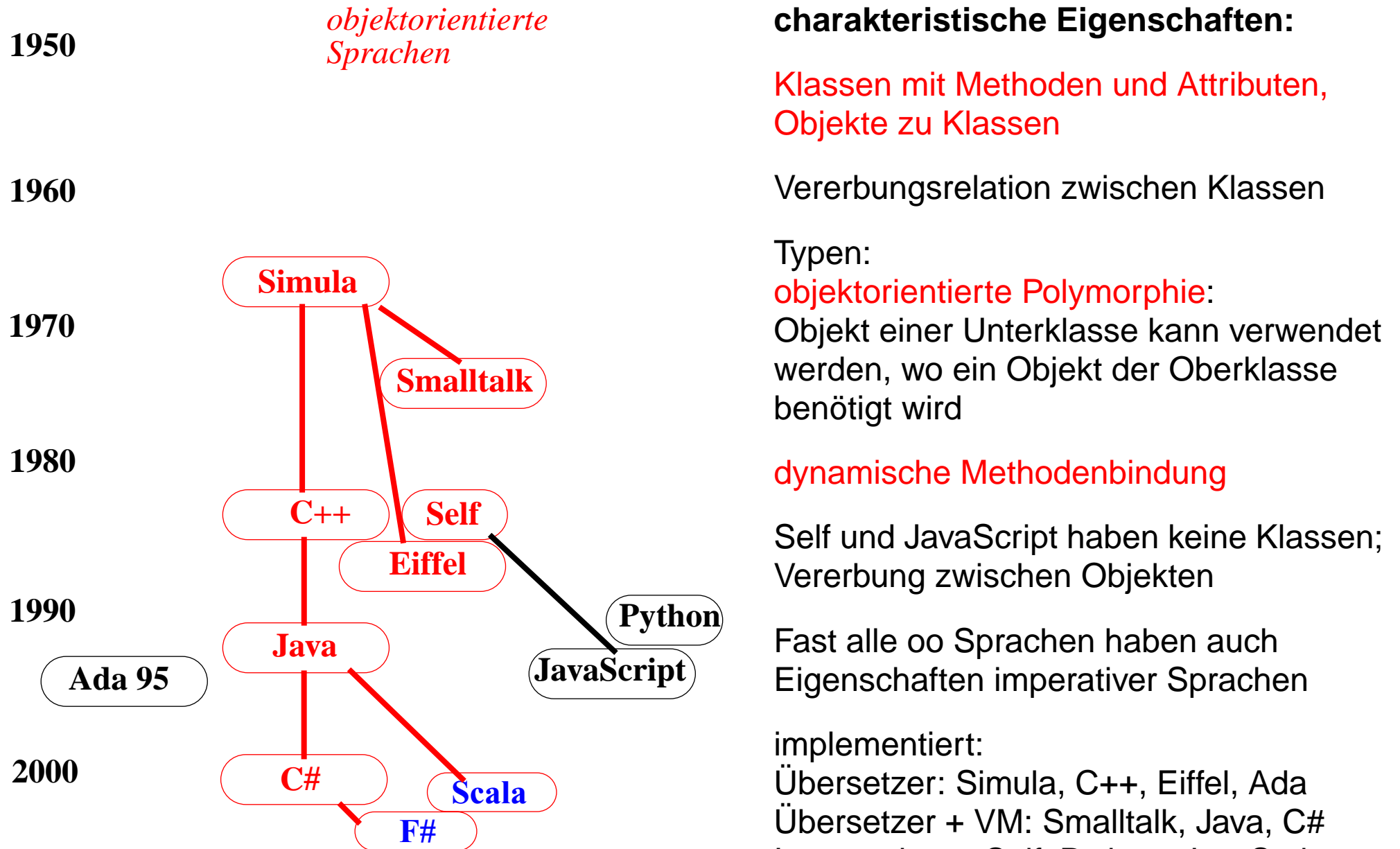


charakteristische Eigenschaften:

- 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>]

Klassifikation: objektorientierte Programmiersprachen



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

Klassifikation: logische Programmiersprachen

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

1950

*logische
Sprache*

1960

1970

Prolog

1980

1990

2000

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

Klassifikation: funktionale Programmiersprachen

1950

*funktionale
Sprachen*

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

1960

Lisp

1970

1980

SML

Miranda

1990

Haskell

2000

Scala

F#

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

Klassifikation: Skriptsprachen

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

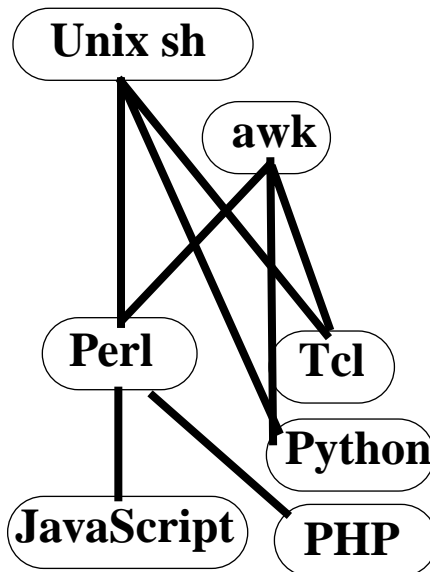
ggf Programme eingebettet in HTML-Texte

1950

Skriptsprachen

1960

1970



2000

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

Klassifikation: Auszeichnungssprachen

1950

*Auszeichnungs-
sprachen*

charakteristische Eigenschaften:

Annotierung von Texten zur Kennzeichnung der Struktur, Formatierung, Verknüpfung

1960

Ziele: **Repräsentation strukturierter Daten** (XML), Darstellung von Texten, Hyper-Texten, Webseiten (HTML)

1970

Sprachkonstrukte:

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

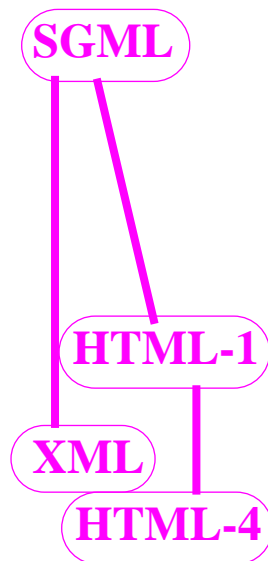
1980

keine Ablaufstrukturen, Variable, Datentypen zur Programmierung

1990

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

2000



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

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.
```

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");
  }
}

```

Perl

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

Java

```

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

```

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
```

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 Spezifikationen von Hardware Spezifikationen von Software Spezifikation von KFGn, Parsern
VHDL	
UML	
EBNF	

1.2 Implementierung von Programmiersprachen

Übersetzung

Programmentwicklung

Editor

Quellmodul
ggf. mit Präprozessor-Anweisungen

Übersetzung

ggf. Präprozessor

Übersetzer

lexikalische Analyse
syntaktische Analyse
semantische Analyse

Zwischen-Code

Optimierung
Code-Erzeugung

Fehlermeldungen

Bibliotheksmodule

Binder

bindbarer Modul-Code

Binden

ausführbarer Code

Ausführung

Eingabe

Maschine

Ausgabe

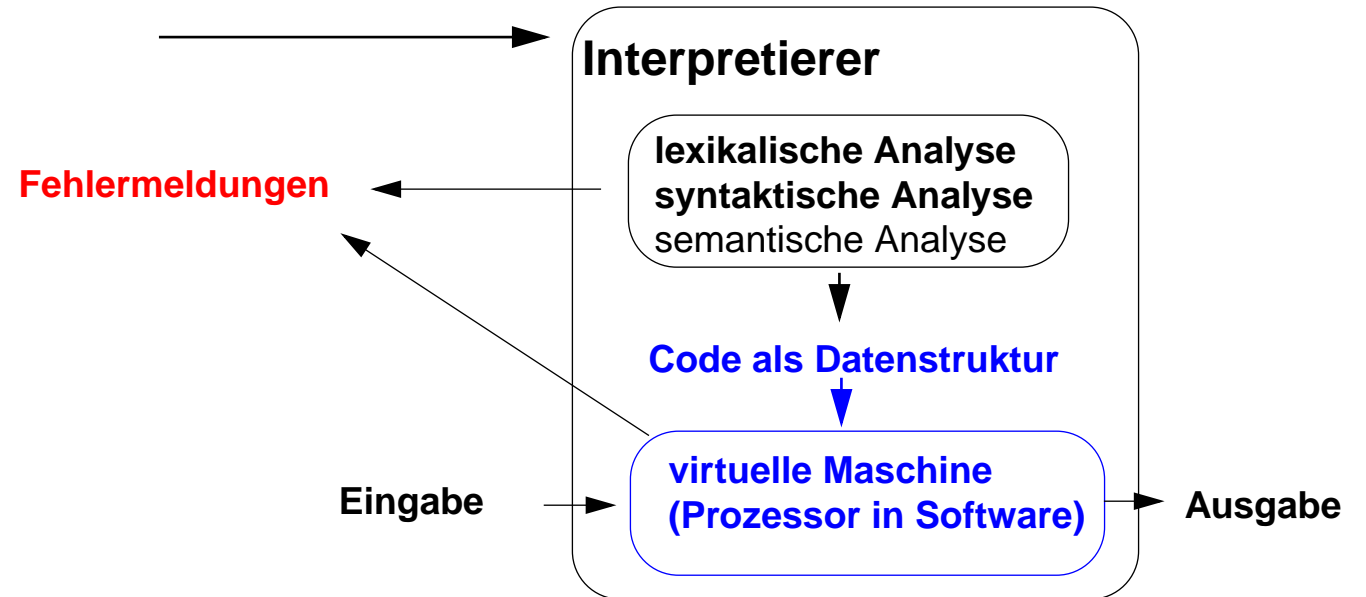
Fehlermeldungen

Interpretation

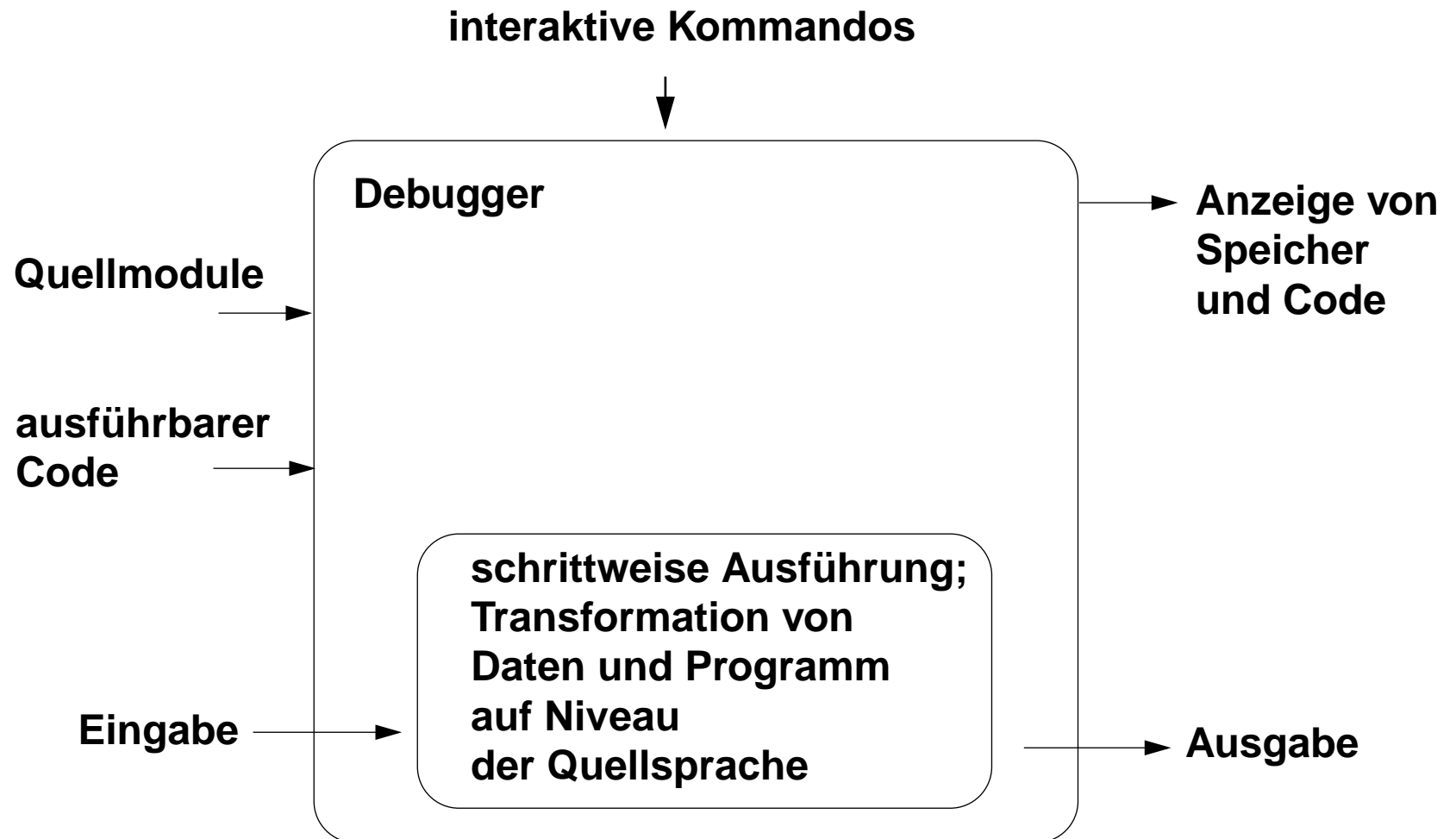
Programmentwicklung



Ausführung



Testhilfe: Debugger



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>
#include "induce.h"
```

Datei an dieser Stelle einfügen

```
#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

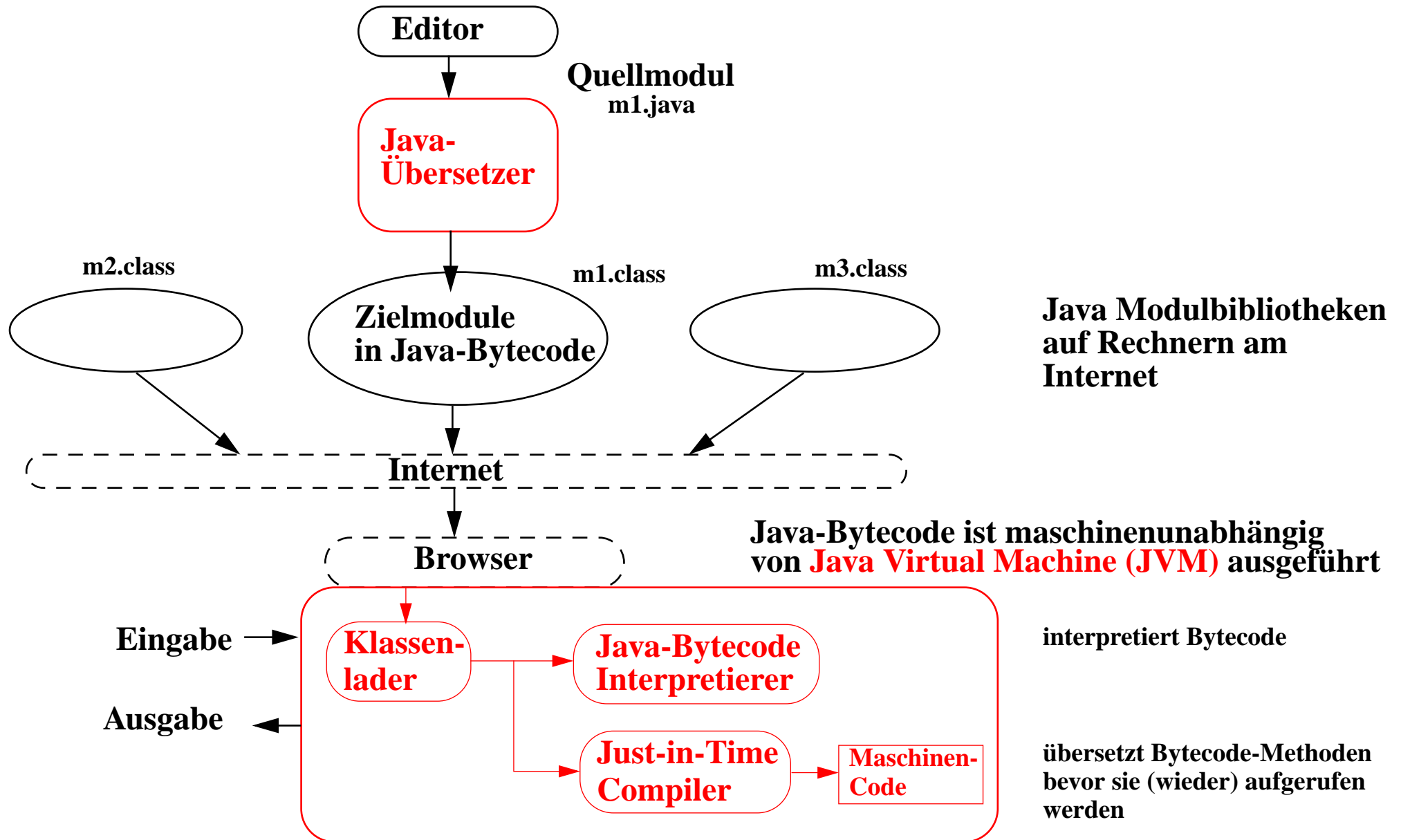
```
#ifndef GORTO
```

bedingter Textblock

```
printf ("early for %d currpartno: %d\n",
        sid, currpartno);
```

```
#endif
```


Ausführung von Java-Programmen



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

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).

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]

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

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]

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

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 ; }
```


Beispiel für die Ebene der Syntax

Ebene

definierte Eigenschaften

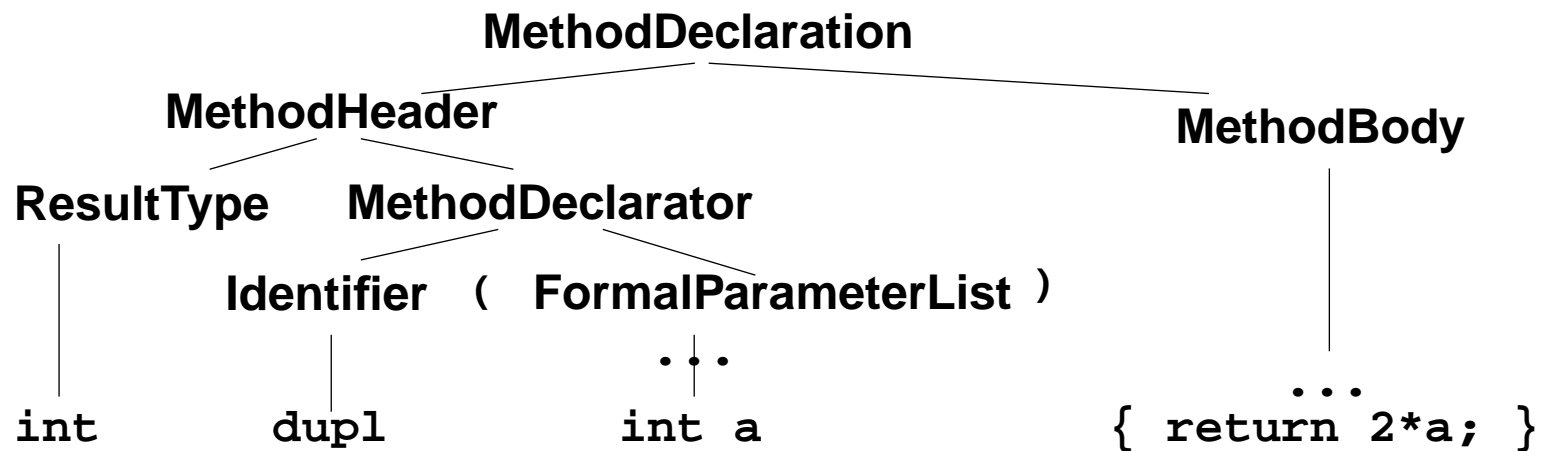
b. Syntax (konkrete und abstrakte Syntax)

syntaktische Struktur

Struktur von Sprachkonstrukten

formal definiert durch **kontext-freie Grammatiken**

Ausschnitt aus einem Ableitungs- bzw. Strukturbaum:



Beispiel für die Ebene der statischen Semantik

Ebene

definierte Eigenschaften

c. statische Semantik

statische Zusammenhänge, z. B.

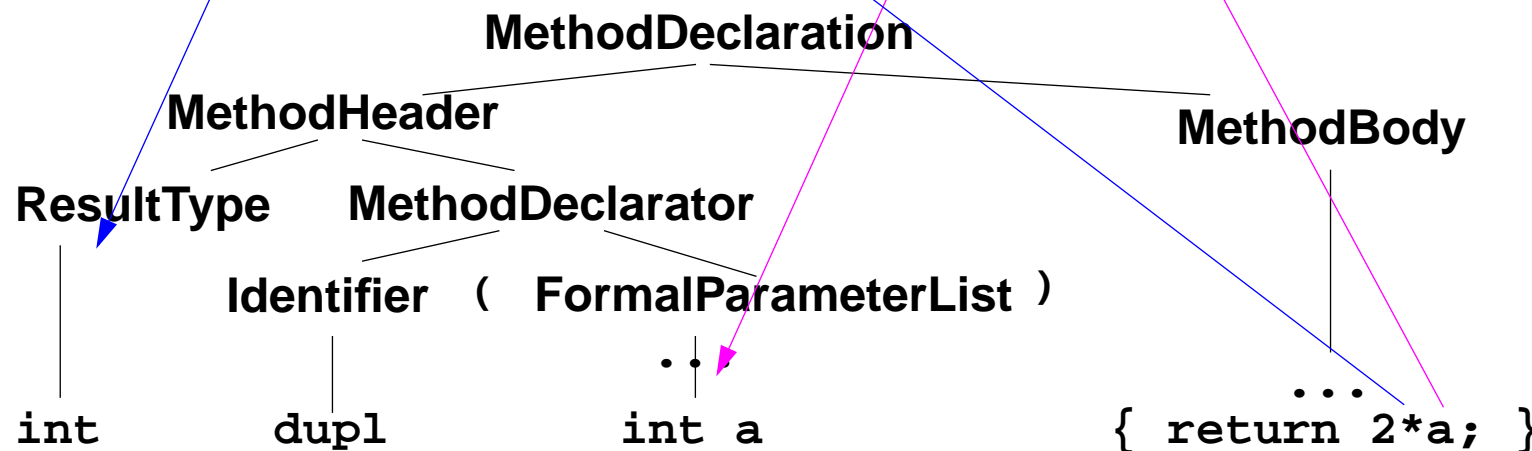
meist verbal definiert;
formal definiert z. B. durch **attributierte Grammatiken**

a ist an die Definition des formalen Parameters gebunden.

Bindung von Namen

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

Typregeln



Beispiel für die Ebene der dynamischen Semantik

Ebene

definierte Eigenschaften

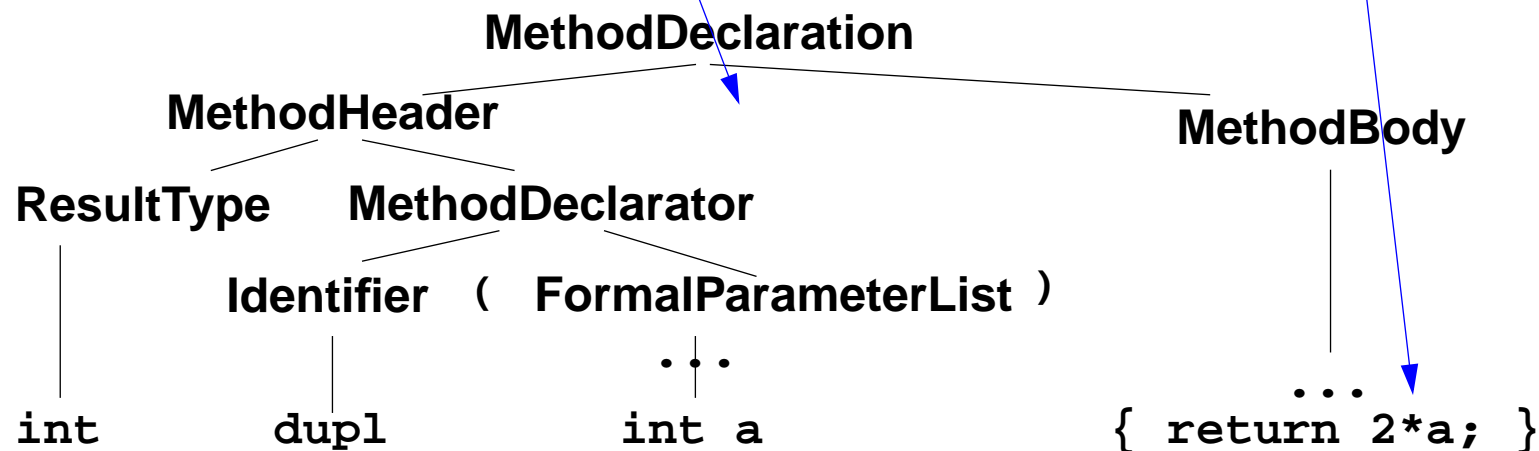
d. dynamische Semantik

Bedeutung, Wirkung der Ausführung

von Sprachkonstrukten, Ausführungsbedingungen

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

Ein Aufruf der Methode `dup1` liefert das Ergebnis
der Auswertung des `return`-Ausdruckes



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.

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 Methodename: **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**

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 }
```

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

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