

Sprachkonzepte

Teil 1: Motivation

Worum geht es im Software-Engineering?

Das Fachgebiet **Software-Engineering** fand beim großen Informatik-Pionier Edsger W. Dijkstra wenig Gnade. Er nannte es "*The Doomed Discipline*". Das Kernanliegen sei "*How to program if you cannot.*", ein Widerspruch in sich. (siehe https://en.wikipedia.org/wiki/Software_engineering#Criticism)

Aber das Software-Engineering wird nicht untergehen, solange es ungelöste **Probleme bei der Softwareentwicklung** gibt, mit denen es sich befasst, etwa diese:

- Dekompositionsproblem
- Abstraktionsproblem
- Redundanzproblem
- Abhängigkeitsproblem
- Formalisierungsproblem
- problematische Rahmenbedingungen

Software-Engineering: Dekompositionsproblem

Dekomposition (hierarchische Zerlegung von Systemen in Teilsysteme) ist Grundlage für arbeitsteilige Entwicklung und gute Wartbarkeit von Software

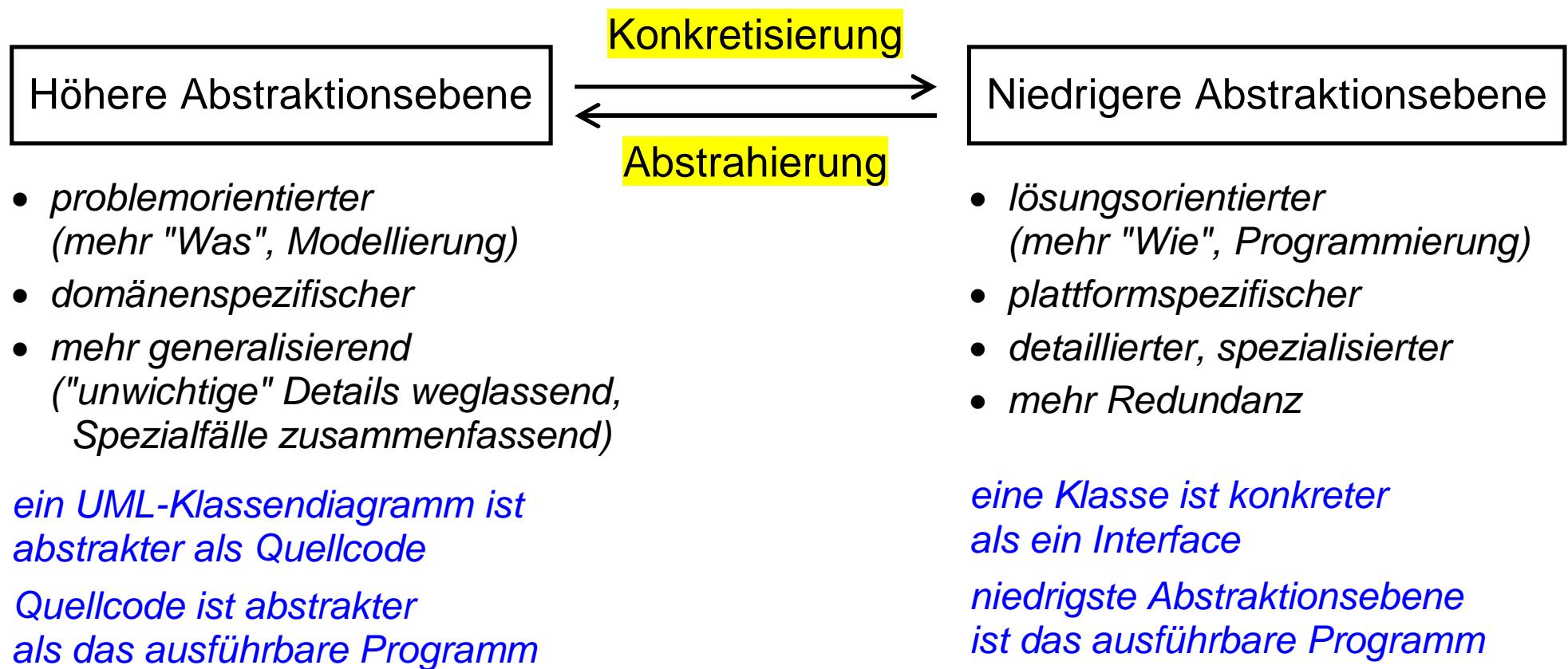
Aspekte für eine Dekomposition:

- Struktur der fachlichen Domäne
jedes Fachgebiet hat etablierte Zerlegung in Teilgebiete
- Struktur der technischen Plattform
z.B. Multitier-Architektur bei Web-Anwendungen
- Struktur der Abläufe und Daten, softwaretechnische Prinzipien
komplexe Funktionalitäten lassen sich in Arbeitsschritte zerlegen
komplexe Daten lassen sich in Typen bzw. Klassen einteilen
Information Hiding, Separation of Concerns, Wiederverwendung, ...
- Organisation der beteiligten Entwickler
Entwickler sind in Firmen, Abteilungen, Teams, Netzwerken, Communities, ... organisiert
jede Organisationseinheit hat Interessen, Fähigkeiten, Zuständigkeiten, ...

Problem: in jeder Dekomposition dominiert ein Aspekt auf Kosten der anderen

Software-Engineering: Abstraktionsproblem

Softwareentwicklung findet auf unterschiedlichen **Abstraktionsebenen** statt:



Problem: Beschreibungen auf höheren Abstraktionsebenen veralten, weil in späten Projektphasen nur noch auf niedrigen Abstraktionsebenen gearbeitet wird

Software-Engineering: Redundanzproblem

Redundanz bezeichnet allgemein das mehrfache Vorhandensein funktions-, inhalts- oder wesensgleicher Objekte

"Objekte" in diesem Sinne sind bei der Software-Entwicklung z.B. Bezeichner, Klassen, Entwurfsmuster, Codesequenzen, ...

- **Vertikale Redundanz**

Informationen aus höheren Abstraktionsebenen sind zwangsläufig explizit oder implizit auch in den niedrigeren Abstraktionsebenen enthalten.

Klassen aus UML-Diagrammen werden explizit im Quellcode wiederholt.

Anwendungsfälle aus Usecase-Diagrammen sind implizit im Quellcode enthalten.

- **Horizontale Redundanz**

Informationen wiederholen sich innerhalb einer Abstraktionsebene.

Klassen müssen programmiert und in Konfigurationsdateien eingetragen werden.

Informationssysteme: für jeden Datentyp "Anlegen", "Abfragen", "Ändern", "Löschen", ...

Problem: Erhöhter Entwicklungsaufwand und Inkonsistenzen durch Redundanz

Software-Engineering: Abhängigkeitsproblem

Jede Software hat interne und externe Abhängigkeiten:

- **interne Abhängigkeiten** entstehen, wenn Lösungen für Teilprobleme aufeinander aufbauen oder miteinander vermischt sind
Vermischung von fachlichen und technischen Aspekten, ...
- **externe Abhängigkeiten** entstehen durch Annahmen über den Einsatzkontext und die Entscheidung für eine Ablaufplattform
Datenaufkommen, Benutzerzahlen, Hardware, Betriebssystem, Standardsoftware, ...

Abhängigkeiten können explizit oder (schlimmer) implizit vorhanden sein:

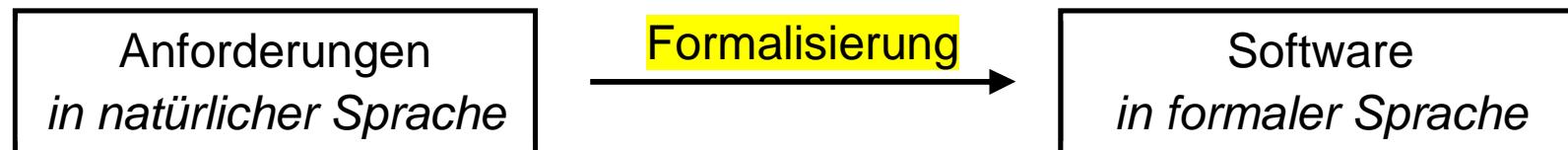
- **explizite Abhängigkeiten** sind programmiersprachlich formuliert und werden automatisch überwacht
Abhängigkeit zwischen Klassen durch Methodenaufrufe (Überwachung per Compiler), ...
- **implizite Abhängigkeiten** sind stillschweigende Annahmen, deren Einhaltung nicht automatisch überwacht wird
Magic Numbers, Reihenfolge von Elementen in einer Collection, ...

Problem: Verringerte Flexibilität und Wiederverwendbarkeit durch Abhängigkeiten

Software-Engineering: Formalisierungsproblem

Formalisierung ... bedeutet die Beschreibung eines Phänomens oder die Formulierung einer Theorie in einer formalen Sprache ... (*Wikipedia*)

In der Softwareentwicklung geht es darum, Anforderungen aus der realen Welt mit einer zu erstellenden Software zu erfüllen:



Einige Methoden für eine schrittweise Formalisierung:

- Strukturierung *z.B. standardisierte Gliederung von Texten*
- reduzierte Sprache *z.B. wohl definiertes Fachvokabular*
- Grafik *Skizzen, Diagramme, ...*

Problem: keine 1:1-Übersetzung von natürlicher in formale Sprache möglich, dadurch Verlust oder Verfälschung von Information

Software-Engineering: problematische Rahmenbedingungen

Rahmenbedingungen bei der Softwareentwicklung:

- zeitliche Rahmenbedingungen
Fertigstellungstermin, vorgesehene Lebensdauer der Software, ...
- finanzielle Rahmenbedingungen
Entwicklungsbudget, Betriebskosten, ...
- rechtliche Rahmenbedingungen
Lizenzen, Gewährleistungsansprüche, Datenschutz, ...
- technische Rahmenbedingungen
einzuhaltende Standards und Normen, vorgegebene Plattformen, ...
- personelle Rahmenbedingungen
Verfügbarkeit, Qualifikation, ...

Problem: Rahmenbedingungen schränken den Handlungsspielraum ein

Software-Engineering: nichts als Probleme ...

Einige der genannten Probleme bedingen sich gegenseitig:

- jede Dekomposition
führt zu horizontaler Redundanz bezüglich der nicht dominierenden Aspekte
- zusätzliche Abstraktionsebenen
führen zu mehr vertikaler Redundanz
- das Vermeiden horizontaler Redundanz
führt zu mehr Abhängigkeiten und umgekehrt
- horizontale Redundanz mit Konsistenzanforderungen
führt zu impliziten internen Abhängigkeiten

Es gibt weitere hier nicht genannte Probleme, z.B.:

- beim Projektmanagement
- bei der Anforderungsermittlung
- beim Lösen technischer Detailprobleme

Software-Engineering: Sprachkonzepte

Die genannten Probleme der Softwareentwicklung lassen sich nicht lösen, sondern nur mehr oder weniger gut mit Kompromissen bewältigen.

Bei der **Problembewältigung** spielen Sprachkonzepte eine wichtige Rolle:

- sprachliche Strukturierungsmittel helfen beim Dekompositions-, Abstraktions-, Redundanz- und Abhängigkeitsproblem
Module, Klassen, Funktionen, Datentypen, ...
- höhere Programmiersprachen und speziell domänenspezifische Sprachen helfen beim Formalisierungsproblem
Problemorientierung, Abstraktion von der Plattform, ...

Inhalte der Lehrveranstaltung

Die wichtigsten Sprachkonzepte kennen und beurteilen lernen.

Die Funktionsweise von Compilern und Interpretern verstehen.

Compilerbau-Werkzeuge anwenden können.

Teil 2: Sprachen

Syntax, Semantik, Pragmatik

Teil 3: Programmierparadigmen

imperative / deklarative Programmierung, Anwendungs- / Systemprogrammierung, Scripting, Textgenerierung

Teil 4: Namen

Bindungen, Scopes, Lebensdauern

Teil 5: Typsysteme

Typprüfung, Typinferenz, parametrische Polymorphie

Sprachkonzepte

Teil 2: Sprachen Syntax, Semantik, Pragmatik

Sprachen

Sprache = System von Zeichen, das der Gewinnung und Ausprägung von Gedanken, ihrem Austausch zwischen verschiedenen Menschen sowie der Fixierung von erworbenem Wissen dient (*Meyers Großes Standardlexikon, 1983*)

- in der Informatik geht es um die Fixierung von Gedanken zu Daten und Algorithmen in einer Form, die auf Rechnern nutzbar ist
- natürliche Sprachen gibt es etwa 6500
(https://de.wikipedia.org/wiki/Sprachfamilien_der_Welt)
- Programmiersprachen gibt es auch mindestens einige hundert, je nachdem wie eng man den Begriff der Programmiersprache fasst

Programmiersprachen

Programmiersprachen sind formale Sprachen, mit denen Aufgabenstellungen so formuliert werden können, dass ein Rechner sie bearbeiten kann.

Der Programmierbegriff ist hier bewusst sehr weit gefasst.

*Traditionell wird nur das Implementieren von Algorithmen als Programmieren aufgefasst.
Sprachen wie etwa SQL, XML, HTML, CSS sind dann keine Programmiersprachen.*

Aspekte einer Programmiersprache:

- **Syntax**

Welche Symbole gibt es?

Vokabular

Wie dürfen die Symbole kombiniert werden?

Grammatik

- **Semantik**

Welche Symbolfolgen haben Bedeutung?

statische Semantik

Und welche Bedeutung ist das?

dynamische Semantik

- **Pragmatik**

Für welche Zwecke ist die Sprache geeignet?

Domänen

Wie drückt man sich in der Sprache am besten aus?

Stil

1) Syntax

2) Semantik

3) Pragmatik

Syntax: Vokabular

Übliches Vokabular von Programmiersprachen:

- Kommentare *ohne Einfluss auf die Bedeutung des Programms*
- Zwischenraum (*whitespace*)
- Terminalsymbole (*tokens*)

Schlüsselwörter (*keywords*) *if, else, while, ...*

Trennzeichen (*seperators*) *Klammern, Punkt, Komma, Semikolon, ...*

Operatoren *+, -, *, /, ...*

Literale *Zahlen, Strings, ...*

Bezeichner (*identifiers*) *Variablennamen, ...*

Vokabulare werden mit regulären Ausdrücken formal spezifiziert.

die regulären Ausdrücke beschreiben die Zusammenfassung von Einzelzeichen zu Elementen des Vokabulars

Vokabular: Reguläre Ausdrücke (1)

Prinzipieller Aufbau von **regulären Ausdrücken**:

- **Zeichenklassen** sind elementare reguläre Ausdrücke:

a *das Zeichen 'a'*

[abc-e] *eines der Zeichen 'a', 'b', 'c', 'd' oder 'e'*

[^abc-e] *keines der Zeichen 'a', 'b', 'c', 'd' oder 'e'*

. *beliebiges Zeichen (außer Zeilenwechsel)*

- **Quantifizierung** eines beliebigen regulären Ausdrucks **r**:

r{2} *zweimal hintereinander*

r{1,3} *ein bis drei Wiederholungen*

r? *entspricht r{0,1}*

r* *entspricht r{0,}, d.h. beliebig oft inklusive keinmal*

r+ *entspricht r{1,}, d.h. mindestens einmal*

- **Verknüpfung** beliebiger regulärer Ausdrücke **r** und **s**:

rs *Konkatenation, d.h. erst r, dann s*

r|s *Alternative, d.h. r oder s*

(r) *Klammerung zum Überschreiben der Vorrangregeln
(Quantifizierung vor Konkatenation vor Alternative)*

Vokabular: Reguläre Ausdrücke (2)

Viele Implementierungen von regulären Ausdrücken enthalten Erweiterungen, z.B. `java.util.regex`:

- vordefinierte Zeichenklassen

`\d` steht für `[0-9]`

`\D` steht für `[^0-9]`

...

- unterschiedlich aggressive Einstellung der Quantifizierungen

greedy `.*foo` findet in `fooxxxxfoo` einzig den Token `fooxxxxfoo`

*. * verbraucht zunächst die gesamte Eingabe, in einem anschließenden Backtracking wird dann vom Ende her geprüft, ob unter den verbrauchten Zeichen `foo` war*

possessive `.*+foo` findet in `fooxxxxfoo` keinen Token

wie greedy, nur ohne Backtracking

reluctant `.*?foo` findet in `fooxxxxfoo` zwei Tokens `foo` und `xxxxfoo`

*. *? verbraucht zunächst kein Zeichen der Eingabe, der Verbrauch wird dann gegebenenfalls zeichenweise erhöht, um ein `foo` zu matchen*

Reguläre Ausdrücke: Beispiel (1)

Lexikalische Analyse arithmetischer Ausdrücke mit dem Compilerbau-Werkzeug ANTLR4 von Terence Parr:

```
// ExprLexer.g4
lexer grammar ExprLexer;
Number: Digits ('.' Digits)? ;           regulärer Ausdruck für den Token Number  
(Tokennamen beginnen mit Großbuchstaben)
fragment Digits: ([0-9])+ ;               Digits ist kein Token,  
sondern benennt nur einen Hilfsausdruck
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
LPAREN: '(';
RPAREN: ')';
WS: [ \t\r\n]+ -> channel(HIDDEN);      Zwischenraum (whitespace) soll nicht in die  
Ergebnis-Tokenfolge aufgenommen werden
```

Reguläre Ausdrücke: Beispiel (2)

Aus der Datei `ExprLexer.g4` generiert ANTLR4 eine Klasse `ExprLexer.java`. Die Klasse enthält einen Automaten, der die Tokens gemäß den regulären Ausdrücken erkennt.

Die erstellte Tokenfolge kann anschließend mit einer in EBNF geschriebenen Grammatik auf eine syntaktische korrekte Reihenfolge geprüft werden.

- Beispiel:

Aus einer eingelesenen Zeichenfolge

1.2 + 34.5

wird in der lexikalischen Analyse eine Tokenfolge

`Number("1.2") PLUS["+"] Number("34.5")`

bzw. mit dem verborgenen Whitespace

`Number("1.2") WS(" ") PLUS["+"] WS(" ") Number("34.5")`

Syntax: Grammatik

Grammatikregeln (Produktionen) legen die erlaubten Tokenfolgen fest.

- bei den üblicherweise verwendeten kontextfreien Grammatiken haben **Produktionen** die Form

Nichtterminalsymbol = Folge von Terminal- und Nichtterminalsymbolen

- eines der Nichtterminalsymbole wird als **Startsymbol** ausgezeichnet
aus dem Startsymbol lassen sich durch wiederholte Anwendung von Produktionen alle syntaktisch gültigen Tokenfolgen (= Folgen von Terminalen) ableiten

Grammatiken werden mit EBNF (*extended Backus-Naur form*) formal spezifiziert.

Grammatik: EBNF

Aufbau von **EBNF (Erweiterte Backus-Naur-Form)**:

- **Terminale** sind nicht weiter ersetzbar elementare Symbole (= Tokens)
"abc" *Zeichenfolge abc*
- **Nichtterminale** sind per Produktionsregel ersetzbare elementare Symbole
name *frei wählbarer Bezeichner*
- **Symbolfolgen** aus Symbolfolgen s und t mit Terminalen und Nichtterminalen
 - s t** *s gefolgt von t*
 - s | t** *entweder s oder t*
 - { s }** *beliebig viele s inklusive keinmal*
 - [s]** *kein oder ein s*
 - (s)** *Klammerung zum Überschreiben der Vorrangregeln*
- **Produktionsregeln:**
Nichtterminal = **Symbolfolge** ;

EBNF: Beispiel (1)

Syntaxanalyse arithmetischer Ausdrücke mit ANTLR4:

```
// ExprParser.g4
parser grammar ExprParser;
options { tokenVocab=ExprLexer; }

expr : multExpr
      | expr (PLUS | MINUS) multExpr
      ;
multExpr : primary
          | multExpr (MUL | DIV) primary
          ;
primary : LPAREN expr RPAREN
         | value
         ;
value : (PLUS | MINUS)? Number
       ;
```

EBNF: Beispiel (2)

Aus `ExprParser.g4` generiert ANTLR4 eine Klasse `ExprParser.java`.

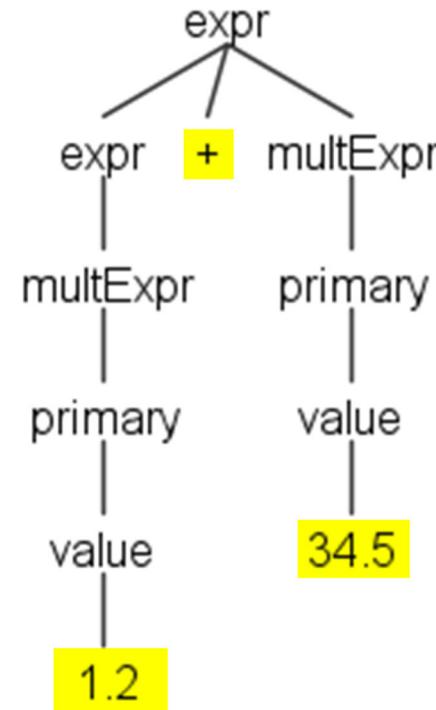
Die Klasse enthält Methoden, die eine eingelesene Tokenfolge auf syntaktisch korrekte Reihenfolge prüfen.

ANTLR4 erstellt aus der Tokenfolge einen Ableitungsbaum (parse tree).

- Beispiel:

Ableitungsbaum zum Ausdruck

1.2 + 34.5



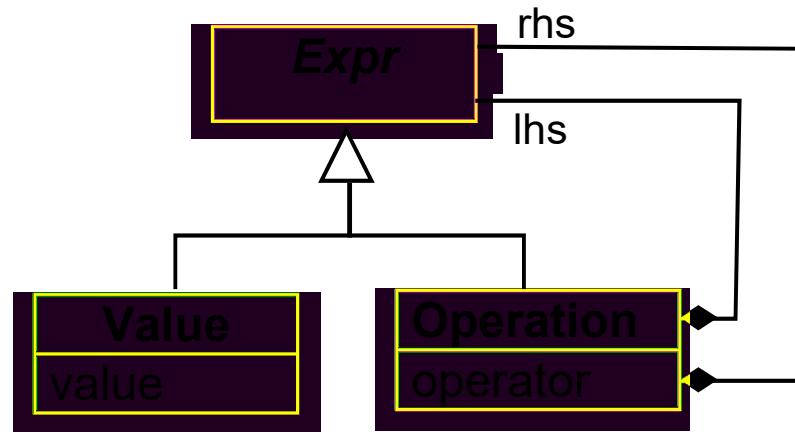
Abstrakte Syntax (1)

Die **abstrakte Syntax** einer Sprache beschreibt, *was* man mit der Sprache ausdrücken kann, und abstrahiert davon, *wie* man es ausdrückt.

- Beschreibung mit einem **Modell**, das die Elemente der Sprache mit ihren Beziehungen zeigt

- Beispiel:

Modell für arithmetische Ausdrücke mit Elementen **Value** und **Operation**

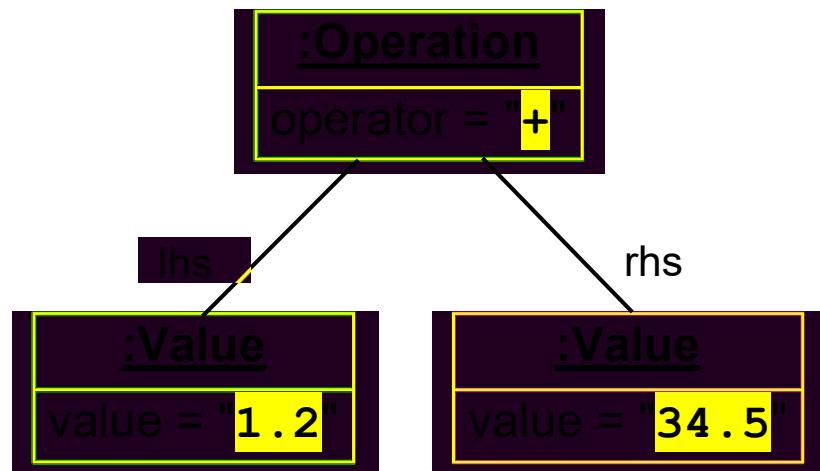


Abstrakte Syntax (2)

Ein **AST** (*Abstract Syntax Tree*) zeigt eine Formulierung als Instanz des Modells der Sprache.

- Beispiel:

Abstrakter Syntaxbaum (**AST**) des Ausdrucks **1 . 2 + 34 . 5**



der AST wird aus dem Ableitungsbaum (parse tree) der konkreten Syntax abgeleitet

1) Syntax

2) Semantik

3) Pragmatik

Semantik (1)

Nicht jede syntaktisch korrekte Symbolfolge ist auch semantisch korrekt

- Beispiel für semantischen Unsinn in syntaktisch korrektem Deutsch:

Dunkel war's, der Mond schien helle,
schneebedeckt die grüne Flur,
als ein Wagen blitzesschnelle,
langsam um die Ecke fuhr.

Drinnen saßen stehend Leute,
schweigend ins Gespräch vertieft,
als ein totgeschoss'ner Hase
auf der Sandbank Schlittschuh lief.

...

Quelle: https://de.wikipedia.org/wiki/Dunkel_war's,_der_Mond_schien_helle

Semantik (2)

- Beispiel für semantischen Unsinn in syntaktisch korrektem Java:

```
public abstract final class Beispiel {  
    public private static void main(String[] args) {  
        System.out.print(12345678901234567890);  
        System.out.print(args[x]);  
    }  
}
```

*Das obige Programm enthält vier semantische Fehler,
genau genommen sogar fünf.*

Statische Semantik (1)

Die **statische Semantik** einer Programmiersprache regelt die Wohlgeformtheit von Formulierungen in Form von Konsistenzregeln für den AST.

- einfache Konsistenzregeln lassen sich unter Umständen mit einer strikten konkreten Syntax erzwingen
allerdings kann eine strikte Grammatik sehr umfangreich werden und eventuell unflexibel hinsichtlich späterer Spracherweiterungen sein
- Konsistenzregeln für komplexe Beziehungen zwischen Sprachelementen können in der Regel erst nach der Syntaxanalyse auf dem AST geprüft werden
klassische Beispiele:
> Regeln zur Typsicherheit von Ausdrücken
> Wertebereiche von Zahlliteralen
> Regeln zur Eindeutigkeit von Namen

Statische Semantik (2)

- Beispiele für semantische Fehler in Java:

```
public abstract final class Beispiel {  
    public private static void main(String[] args) {  
        System.out.print(12345678901234567890);  
        System.out.print(args[x]);  
    }  
}
```

*eine Klasse kann nicht zugleich abstract und final sein **

*eine Methode kann nicht zugleich public und private sein **

*eine Variable muss vor ihrer Benutzung definiert werden***

ein ganzzahliges Literal ohne L muss im Zahlbereich von int liegen

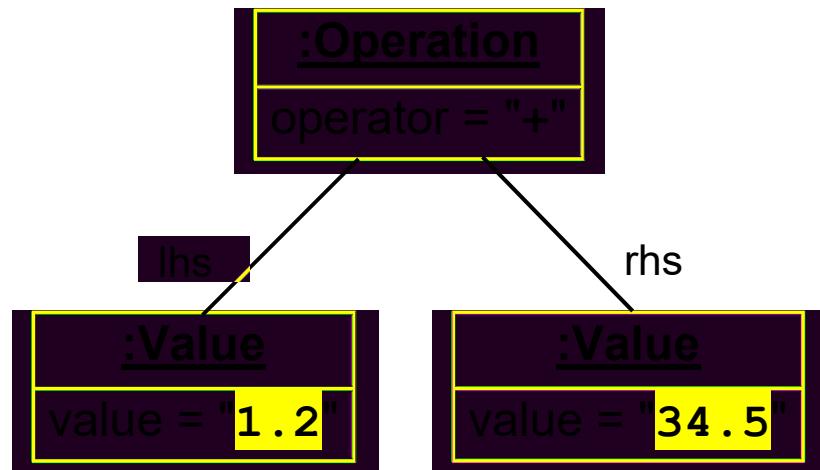
* die Kombinationen **abstract final** und **public private** könnte man alternativ auch mit einer strikten konkreten Syntax verhindern

** der Wert von **x** muss außerdem im gültigen Indexbereich von **args** liegen, was aber nicht mehr eine Frage der statischen Semantik ist, sondern der dynamischen Semantik

Statische Semantik (3)

- mögliche statische Semantik für den AST des arithmetischen Ausdrucks:

die Zahlenwerte der Operanden müssen im Zahlbereich des 32-Bit-Gleitkomma-formats von IEEE 754 liegen, d.h. zwischen ca. $-3,4 \cdot 10^{38}$ und $+3,4 \cdot 10^{38}$



Dynamische Semantik (1)

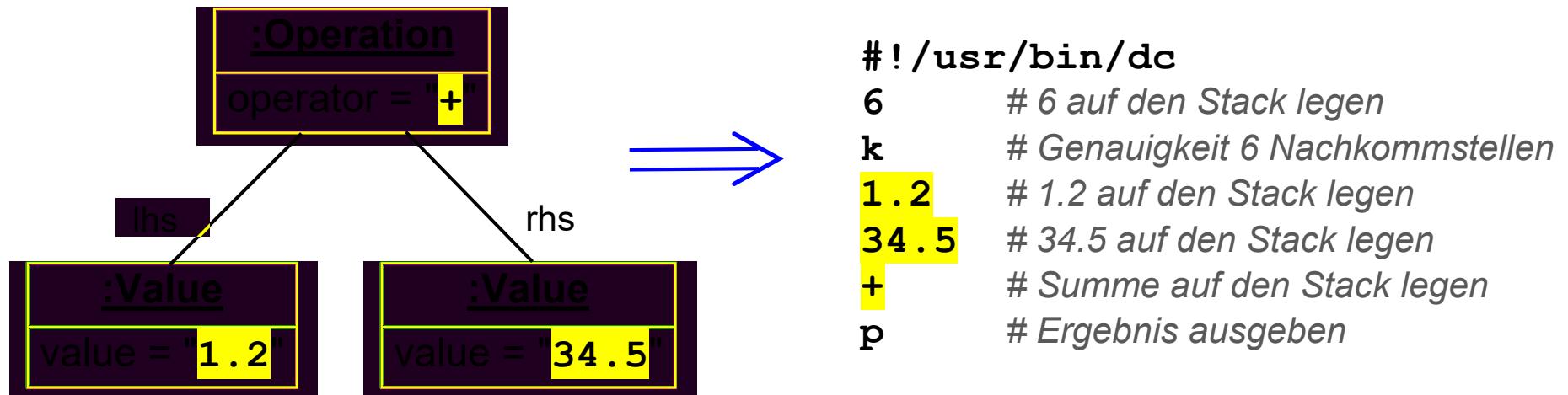
Die **dynamische Semantik** einer Programmiersprache ist das, was Formulierungen bei der Weiterverarbeitung des AST bewirken.

anders als die statische Semantik wird die dynamische Semantik also nicht auf dem AST geprüft, sondern sie wird dem AST durch die Weiterverarbeitung zugeordnet

- Beispiel:
 - in Java bewirkt eine Formulierung `a[i]` zur Laufzeit
 - entweder die Auswahl des Elements mit Index `i` im von `a` referenzierten Array
 - oder eine `NullPointerException`, wenn `a` kein Array referenziert
 - oder eine `ArrayIndexOutOfBoundsException` bei falschem Index
- dabei sind die beiden Exceptions keine Verletzung der dynamischen Semantik, sondern sie sind Aspekte der dynamischen Semantik des Array-Zugriffs*

Dynamische Semantik (2)

- dynamische Semantik in Form eines **Interpreters**:
der Interpreter führt je nach AST-Knoten bestimmte Aktionen aus
Beispiel arithmetischer Ausdruck: den AST in Tiefensuche ablaufen und dabei den Wert des Ausdrucks schrittweise berechnen
- dynamische Semantik in Form eines **Compilers**:
der Compiler bildet AST-Knoten auf bestimmte Konstrukte einer Zielsprache ab
Beispiel arithmetischer Ausdruck: den AST auf eine Befehlsfolge für einen Prozessor abbilden, z.B. für den Linux Desktop-Calculator dc



- 1) Syntax*
- 2) Semantik*
- 3) Pragmatik*

Pragmatik: Domänen (1)

Eine **Domäne** ist in der Softwaretechnik ein abgrenzbares Problemfeld oder ein bestimmter Einsatzbereich für eine Software (siehe Wikipedia "Problemdomäne")

Einteilung von Programmiersprachen nach ihrem Domänenbezug:

- **GPL** (*General-purpose Language*)

GPLs haben eine universelle Syntax und Semantik, die sie für viele verschiedene Domänen einsetzbar macht

der Domänenbezug entsteht durch domänenspezifische Bibliotheken und Frameworks, die in der Sprache implementiert sind

*alle höheren Programmiersprachen werden hier üblicherweise eingeordnet,
z.B. Java, C, C++, Python, Scala, ...*

- **DSL** (*Domain-specific Language*)

DSLs sind in Syntax und Semantik auf eine spezifische Domäne abgestimmt

z.B. SQL für die Domäne Datenbanken

z.B. HTML und CSS für die Domäne Webseiten

Pragmatik: Domänen (2)

Der Übergang zwischen GPLs und DSLs ist fließend:

- auch eine GPL kann für bestimmte Domänen besser und für andere schlechter oder gar nicht geeignet sein

C und C++ eignen sich z.B. besonders gut für die Domäne Systemprogrammierung, unter anderem, weil sie den Typ Adresse unterstützen

in manchen Domänen sind funktionale Spracheigenschaften besonders nützlich, in anderen objektorientierte, usw.

für die Auswahl einer bestimmten GPL sind oft die verfügbaren Bibliotheken und Frameworks wichtiger, als die bei vielen GPLs ähnlichen Spracheigenschaften

- manche APIs von GPL-Bibliotheken lassen sich fast wie eine DSL verwenden
die Namen der Bibliotheksfunktionen bilden das Vokabular und die möglichen Aufrufreihenfolgen und Aufrufverschachtelungen der Funktionen die Syntax
man nennt solche APIs Fluent Interfaces oder interne DSLs
- DSLs können in eine GPL eingebettet sein
in Form von String-Literalen und speziellen Kommentaren

Pragmatik: Stil (1)

Bei **Stil** geht es um diejenigen Merkmale eines Textes, die nicht die Bedeutung betreffen, sondern nur die Art und Weise, wie diese Bedeutung sprachlich formuliert ist. (*sinngemäß aus Wikipedia "Stil", Abschnitt "Sprache"*)

beim Programmieren lässt sich auch mit ein und derselben Sprache die gleiche Bedeutung meist auf unterschiedliche Art und Weise erzielen, also mit unterschiedlichem Stil

Stilaspekte bei Programmiersprachen:

- Programmorganisation
Aufteilung in Dateien und Ordner, Gliederung, ...
- Layout
Einrückung, Whitespace, Zeilenlänge, ...
- Namenskonventionen
Zeichenvorrat, Groß- / Kleinschreibung, Aufbau, Länge, ...
- Idiome (*Redewendungen*)
übliche Formulierungen für wiederkehrende Situationen, bevorzugte und geächtete Sprachelemente, ...

Pragmatik: Stil (2)

Für GPLs gibt es oft viele, mitunter auch konkurrierende **Stilempfehlungen**.

Anspruch der Empfehlungen ist immer, dass sie die Codequalität verbessern, etwa hinsichtlich Lesbarkeit, Wartbarkeit und Fehlerrisiken. Ob sie diesen Anspruch einlösen, ist teilweise umstritten

Beispiele für Stilempfehlungen zur Sprache C:

- MISRA C (*Motor Industry Software Reliability Association*)
- SEI CERT C Coding Standard
- Linux kernel coding style
- GNU coding standards

Sprachkonzepte

Teil 3: Programmierparadigmen
imperative / deklarative Programmierung,
Anwendungs- / Systemprogrammierung,
Scripting, Textgenerierung

Programmierparadigmen

Ein **Paradigma** ist eine grundsätzliche Denkweise (*Wikipedia, 2021*).

Bei **Programmierparadigmen** bezieht sich die Denkweise darauf, was man als die fundamentalen Bausteine, Prinzipien und Zwecke von Programmen ansieht. Beispiele sind etwa die imperative und die deklarative Denkweise:

- **imperative Programmierung**

Programme bestehen aus einer Folge von *Anweisungen*, die festlegen, wie ein Anfangszustand schrittweise in einen gewünschten Endzustand überführt wird
die prozedurale und die objektorientierte Programmierung sind imperative Paradigmen

- **deklarative Programmierung**

Programme bestehen aus *Ausdrücken*, die eine gesuchte Lösung beschreiben, ohne dabei den Lösungsweg detailliert festzulegen
die funktionale und die logische Programmierung sind deklarative Paradigmen

Viele gängige Programmiersprachen unterstützen mehrere Paradigmen zugleich.

Imperative Programmierung: Motivation

Die Rechnerhardware legt die imperative Programmierung nahe:

- der Rechner hat einen **Zustand**, bestehend aus den Inhalten von Prozessorregistern, Caches, Hauptspeicher, Dateien usw.
- der Prozessor ändert beim Ausführen von Befehlen diesen Zustand

Imperative höhere Programmiersprachen abstrahieren von den technischen Details der Rechnerhardware, behalten aber das Verarbeitungsprinzip der schrittweisen Zustandsänderung bei:

- der Zustand ist über **Variablen** und Ein-/Ausgabekanäle zugänglich
eine Variable ist hier ein Name für einen Speicherbereich
- die Befehle erzeugt der Compiler aus **Anweisungen**
grundlegende Anweisung ist die Variablenzuweisung
- die Menge der Variablen und Anweisungen kann je nach Sprache auf unterschiedliche Weise **strukturiert** werden

Imperative Programmierung: prozedural

Die **prozedurale Programmierung** stellt die Strukturierung der Anweisungen mittels Prozeduren in den Mittelpunkt:

- Leitfrage: Was muss das Programm tun?
- Programmieraufgaben werden mittels **Prozeduren** schrittweise in überschaubare Teile zerlegt
bzw. bei Bottom-up-Vorgehen: wiederholt auftretende Anweisungsfolgen werden zu Prozeduren zusammengefasst
- aus den Prozeduren ergibt sich die Strukturierung der **Variablen**
Variablen treten als Aufrufparameter und lokale Variablen von Prozeduren auf oder als globale Variablen außerhalb der Prozeduren
- Prozeduren liefern Ergebnisse als Rückgabewert oder als **Seiteneffekt**
Zuweisungen an Ausgabeparameter und globale Variablen sind Seiteneffekte die Ergebnisse können vom gesamten erreichbaren Programmzustand abhängen, nicht nur von den Parameterwerten

Imperative Programmierung: objektorientiert

Die **objektorientierte Programmierung** stellt die Strukturierung der Variablen mittels Objekte in den Mittelpunkt:

- Leitfrage: Womit muss das Programm umgehen?
- der Programmzustand wird mit **Objekten** modelliert

neue Objekte entstehen bei Klassen-basierten Sprachen durch Klasseninstanziierung und bei Prototyp-basierten Sprachen durch Ableitung von bestehenden Objekten

Variablen treten zuvorderst gekapselt als Instanzvariablen in Objekten auf

- die **Methoden** der Objekte bzw. Klassen strukturieren die Anweisungen
die Anweisungen in den Methoden legen das Verhalten von Objekten fest und sorgen für jederzeit konsistente Objektzustände
- kontrollierte **Seiteneffekte**
in Methoden Zuweisungen nur an die Variablen des Aufrufobjekts

Deklarative Programmierung: Motivation

Die Mathematik legt die deklarative Programmierung nahe:

- **Formeln** beschreiben, welche Eigenschaften eine gesuchte Lösung haben muss, aber nicht unmittelbar, wie die Lösung berechnet werden kann oder soll
 - z.B. beschreiben Formeln mit verschachtelten Funktionen die Abbildung von Eingangsgrößen auf Ausgangsgrößen*
 - z.B. beschreiben Formeln mit prädikatenlogischen Ausdrücken Einschränkungen einer Lösungsmenge*

Deklarative Programmiersprachen übernehmen das Prinzip der mathematischen Formel und ergänzen automatische Lösungsverfahren:

- Variablen sind Platzhalter für **Ausdrücke**, nicht Namen von Speicherbereichen
Variablen sind nicht änderbar (immutable)
- der Compiler / Interpreter legt die Auswertungsreihenfolge der Ausdrücke fest
es sind viele automatische Optimierungen möglich, weil es keine Seiteneffekte in Form von Zustandsänderungen gibt

Deklarative Programmierung: funktional

Bei der **funktionalen Programmierung** besteht ein Programm aus Ausdrücken mit verschachtelten Funktionen:

- Leitfrage: Wie hängen die Programmausgaben von den Eingaben ab?
- **Funktionen** sind Prozeduren, die keine Seiteneffekte haben und deren Rückgabewert nur von den Aufrufparametern abhängt
Funktionen höherer Ordnung haben Funktionen als Parameter und / oder Rückgabewert
- **Lambdas** (= *anonyme Funktionen*) können dynamisch erstellt werden
zusammen mit dem Erstellungskontext bilden sie Closures
häufig als Argumente oder Rückgabewerte von Funktionen höherer Ordnung
- **Rekursion** statt Schleifen
bevorzugt als Endrekursion (tail recursion), bei der die Rekursion als letzte Operation vor dem return auftritt

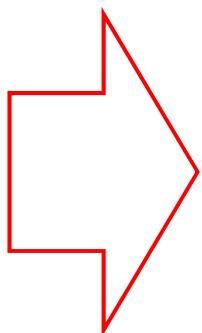
Funktionale Programmierung: Currying

Currying erlaubt es, eine Funktion mit mehreren Parametern durch mehrere Funktionen mit je einem Parameter zu ersetzen

funktionale Programmiersprachen brauchen deshalb im Prinzip keine Funktionen mit mehreren Parametern zu unterstützen (ist z.B. bei Haskell der Fall)

- Beispiel in Python (entnommen aus de.wikipedia.org/wiki/Currying):

```
#!/usr/bin/python3
def uncurried_add(x, y):
    return x + y
n = uncurried_add(3, 5)
```



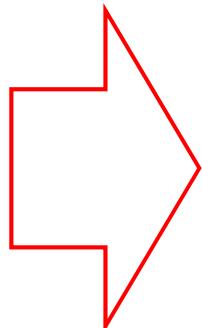
```
#!/usr/bin/python3
def curried_add(x):
    return lambda y: x + y
n = curried_add(3)(5)
add_three = curried_add(3)
n = add_three(5)
```

curried_add ist eine Funktion höherer Ordnung mit einem Parameter, die eine Funktion mit wiederum einem Parameter zurückliefert

Funktionale Programmierung: Endrekursion

Endrekursion kann ein optimierender Compiler leicht durch eine Schleife ersetzen und so die Laufzeit und den Stack-Speicher der Funktionsaufrufe einsparen:

```
R f(P p) {  
    if (condition) {  
        return value;  
    }  
    ...  
    return f(expression);  
}
```



```
R f(P p) {  
    while (true) {  
        if (condition) {  
            return value;  
        }  
        ...  
        p = expression;  
        continue;  
    }  
}
```

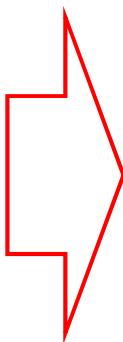
Funktionale Programmierung: Continuations

Continuations sind Funktionen, die verbleibende Berechnungen repräsentieren. An die Stelle der Funktionsrückkehr tritt der Aufruf der übergebenen Continuation. (siehe de.wikipedia.org/wiki/Continuation-Passing_Style)

- eine rekursive Funktion kann per Continuation endrekursiv gemacht werden:

```
#!/usr/bin/python3
def fakultaet(n) :
    if (n == 0) :
        return 1
    return n * fakultaet(n - 1)

print(fakultaet(5))
```



```
#!/usr/bin/python3
def fakultaet(c, n) :
    if (n == 0) :
        return c(1)
    fakultaet(lambda f : c(n * f), n - 1)

fakultaet(lambda f: print(f), 5)
```

Funktionale Sprachen mit Tail-Call Optimization können in der endrekursiven Version das dynamische Wachsen des Stacks unterbinden.

Bei Python sind die Continuations hier kontraproduktiv, weil deren Aufrufe die Zahl der Stack-Frames verdoppelt. Python hat keine Tail-Call Optimization.

Deklarative Programmierung: logisch

Bei der **logischen Programmierung** ist ein Programm eine Liste von speziellen prädikatenlogischen Ausdrücken (Horn-Klauseln):

- Leitfrage: Was ist über die Lösungsmenge bekannt?
- **Fakten** nennen Eigenschaften konkreter Objekte und Relationen zwischen konkreten Objekten
- **Regeln** beschreiben, unter welchen Bedingungen beliebige Objekte bestimmte Eigenschaften haben oder Relationen zwischen beliebigen Objekten bestehen
- an die **Wissensbasis** aus Fakten und Regeln werden **Anfragen** gestellt

Fragen nach Eigenschaften von konkreten Objekten oder nach Relationen zwischen konkreten Objekten werden mit ja (true) oder nein (false) beantwortet

Fragen nach der Existenz von Objekten, die bestimmte Eigenschaften haben oder in bestimmten Relationen stehen, werden im positiven Fall mit konkreten Objekten beantwortet (auf Anforderung auch schrittweise mit allen passenden Objekten)

Weitere Programmierparadigmen (1)

Denkweisen, die sich mehr auf den Charakter der zu erstellenden Software als auf den Charakter der Programmiersprachen beziehen:

- **Anwendungsprogrammierung**

Umsetzen der Anforderungen von Endanwendern

*als Programmiersprachen kommen eine Vielzahl von GPLs und DSLs in Frage
weit verbreitet sind etwa C#, Java, Python, HTML / CSS / JavaScript*

- **Systemprogrammierung**

Verwaltung und Steuerung von Ressourcen in Rechensystemen

Bereitstellen von Plattformen für Anwendungsprogramme

dominierende höhere Programmiersprachen sind die GPLs C und C++

Die beiden Denkweisen lassen sich nicht scharf voneinander abgrenzen, sie treten mitunter vermischt auf oder sind eine Frage des Blickwinkels

Ist ein Webbrowser ein Anwendungs- oder ein Systemprogramm?

Weitere Programmierparadigmen (2)

Unter dem Begriff Programmierparadigmen werden zahllose weitere Denkweisen beschrieben (siehe die deutsche und englische Wikipedia)

Hier nur ein paar weitere Beispiele:

- **Scripting**
*für Kommandoprozeduren, die die Bedienung von Rechnern automatisieren
für Gluecode, der z.B. GUI-Bedienelemente mit Programmlogik verknüpft
Sprachen: z.B. sh, JavaScript, Python, VBScript, ...*
- **Textgenerierung**
*von formatierten Programmausgaben bis zur Generierung von Quellcode und Webseiten
Sprachen: z.B. printf, Stringtemplate, ...*
- Textauszeichnung und -präsentation
Sprachen: z.B. TeX / LaTeX, Html / CSS, XML
- Datenabfrage
Sprachen: z.B. SQL, XQuery / XPath
- ...

Scripting: Eigenschaften

Bei der **Script-Programmierung** werden neue Anwendungen durch einfaches Zusammenfügen bereits vorhandener Anwendungen und Komponenten erstellt.

Mit mächtigen GPLs kann man komplexe Anwendungen und Komponenten sicher und effizient von Grund auf implementieren. Will man solche Anwendungen und Komponenten lediglich verwenden, reichen flexiblere und leichter zu lernende Scriptsprachen.

Einige Typische Eigenschaften von Scriptsprachen:

- deklarationsfreie Syntax

Namen werden implizit deklariert und ihre Typen dynamisch bestimmt

- ausgereifter Umgang mit Strings

Pattern-Matching mit regulären Ausdrücken

Strings als Array-Index

Strings als Code ausführen

- zeilenweise Interpretation des Quelltextes statt Übersetzung

dadurch auch interaktiv nutzbar

Scripting: Beispiele (1)

Mit einer Unix-Shell wie **sh** lassen sich Anwendungen bequem aus vorhandenen ausführbaren Programmen und Systemressourcen zusammensetzen.

- Umlenkung der Standard-Ein-/Ausgabe

Programm1 | Programm2

Pipe als Standard-Ausgabe für Programm1 und Standard-Eingabe für Programm2

Programm < Datei1 > Datei2 2>&1

*Datei1 als Standard-Eingabe, Datei2 als Standard-Ausgabe
und Fehlerausgabe (2) auf Standardausgabe (1) umleiten*

- sequentielle Ausführung

Programm1 ; Programm2 unbedingte Sequenz

Programm1 && Programm2 Programm2 nur, wenn Programm1 mit Exitcode 0

Programm1 || Programm2 Programm2 nur, wenn Programm1 mit Exitcode nicht 0

- Dateien mit regulären Ausdrücken auswählen

** . [ch] alle Dateien mit Endung .c und .h im aktuellen Arbeitsverzeichnis*

Scripting: Beispiele (2)

Mit einer Scriptsprache wie **Python** lassen sich zusammengesetzte Datentypen deklarationsfrei und flexibel nutzen.

- Listen können Elemente verschiedener Typen enthalten und sind zugleich wie Arrays nutzbar:

```
liste = [123, 456, 890]      # neue Liste
liste[1] = 4.56                # zweites Element bekommt neuen Typ und Wert
liste.insert(1, 'Hallo')       # zusätzliches Element hinter dem ersten
liste.append(liste.pop(0))     # erstes Element wird ans Ende verschoben
print(liste)                  # Ausgabe: ['Hallo', 4.56, 890, 123]
```

- weitere zusammengesetzte Typen:

```
tupel = (10, 20, 10)          # im Gegensatz zur Liste nicht änderbar
menge = {'Hallo', 'Hi'}        # kein Index und keine doppelten Werte
dictionary = {'x':1, 'y':2}     # Schlüssel-Wert-Paare, Schlüssel als Index
```

Textgenerierung: Eigenschaften

Bei DSLs für die **Textgenerierung** werden Lückentexte mit Daten gefüllt.

*spielt vor allem bei der Generierung von HTML-Seiten eine große Rolle,
aber auch bei der Generierung von Programmcode aus Modellen*

- **Formatierungssprachen** sind in eine Wirtssprache eingebettete DSLs:
der Lückentext ist aus Sicht der Wirtssprache ein gewöhnlicher String
Bibliotheksfunktionen bzw. -klassen interpretieren zur Laufzeit bestimmte
Zeichenfolgen im String als Lücken und setzen dort formatierte Daten ein
Beispiele: C printf, Java String.format, Python str.format
- **Templatesprachen** sind eigenständige DSLs:
die Lückentexte werden abgesetzt in eigenen Dateien erstellt
es gibt syntaktische Mittel, um zum Füllen der Lücken durch ein Datenmodell zu
iterieren, die Syntax wird von einer Templateengine interpretiert
Beispiele: VTL (Velocity Template Language), StringTemplate

Textgenerierung: Beispiel (1)

StringTemplate ist eine Templatesprache und Java Template-Engine von Terence Parr (siehe auch ANTLR in Teil 2).

Lückensyntax (*statt der spitzen Klammern sind auch andere Begrenzer möglich*):

- Referenzen erlauben den Zugriff auf Java-Objekte eines Modells:

`<attribute.property>`

Attribute sind sozusagen Parameter von Templates und werden javaseitig mit Objekten eines Modells initialisiert.

Properties werden auf Instanzvariablen oder getter-Methoden der Objekte abgebildet.

- funktionaler Stil für die Iteration über Modellelemente:

`<attribute.property:Template()>`

`<attribute.property:Template() ; separator="...">`

auf alle Elemente der Property das angegebene Template anwenden

- Bedingte Template-Abschnitte:

`<if(Bedingung)> ... <elseif(Bedingung)> ... <else> ... <end>`

Textgenerierung: Beispiel (2)

- Template-Gruppe mit zwei Templates für die Generierung einer HTML-Seite:

```
delimiters "$", "$"          spitze Klammern als Begrenzer bei HTML ungeeignet
notenspiegel(n) ::= <<
<!DOCTYPE html>
<html lang="de">
<body>
<b>NOTENSPIEGEL</b>
<table>
$n:fachnote(); separator="\n"$  wendet das Template fachnote auf
</table> alle Elemente aus n an und fügt nach
</body> jedem Element einen Zeilenwechsel ein
</html>
>>

fachnote(f) ::= <<           Template fachnote mit Attribut f
<tr>
<td>$f.fach$</td>
<td>$if(f.benotet)$L$else$$endif$</td>
<td>$f.note$</td>
</tr>
>>
```

greift auf *fach*, *benotet* und *note* im Datenmodell zu

Textgenerierung: Beispiel (3)

- Datenmodell `Fachnote.java`:

```
public final class Fachnote {  
    public final String fach;  
    public final boolean benotet;  
    public final String note;  
  
    public Fachnote(String fach, double note) {  
        this.fach = fach;  
        this.benotet = true;  
        this.note = String.format("%.1f", note);  
    }  
  
    public Fachnote(String fach, boolean bestanden) {  
        this.fach = fach;  
        this.benotet = false;  
        this.note = bestanden ? "be" : "nb";  
    }  
}
```

Textgenerierung: Beispiel (4)

- Modellinstanziierung und Aufruf der Template-Engine:

```
import org.stringtemplate.v4.STGroupFile;
import org.stringtemplate.v4.ST;

...
Fachnote[] fachnoten = new Fachnote[] {
    new Fachnote("Sprachkonzepte", 1.3),
    new Fachnote("Sprachkonzepte Uebungen", true)
};

ST template
    = new STGroupFile("notenspiegel.stg").getInstanceOf("notenspiegel");
template.add("n", fachnoten);
String htmlSeite
    = template.render();
...

```

Initialisierung des Template-Attributs

Name des Templates in der Gruppen-Datei

Sprachkonzepte

Teil 4: Namen

Bindungen, Scopes, Lebensdauern

Namen

Namen sind ein sprachliches Abstraktionsmittel.

Gibt man einem Gegenstand einen Namen, kann man über den Namen den Gegenstand jederzeit in einfacher Weise referenzieren, unabhängig davon, wie komplex der Gegenstand ist.

in einer Programmiersprache wird von Registern, Speicheradressen usw. abstrahiert relevante Gegenstände sind dort Variablen, Typen, Operationen usw.

Wichtige Fragen im Zusammenhang mit Namen:

- **Bindungszeitpunkte:**
Zu welchen Zeitpunkten können Namen an Gegenstände gebunden werden?
- **Scopes:**
In welchen Programmberichen gelten Bindungen (*bindings*) zwischen Namen und Gegenständen?
Wie hängen die Geltungsdauern von Bindungen und die Lebensdauern von Gegenständen zusammen?

Bindungszeitpunkte (1)

Zeitpunkte bei der Programmierung, zu denen Namen an Gegenstände gebunden werden können (*aufsteigend von den frühesten zu den spätesten Zeitpunkten*):

- beim Entwerfen und Implementieren der Programmiersprache
das Vokabular der Sprache an Programmierkonzepte binden
- beim Schreiben eines Programms
Namen an benutzerdefinierte Typen, Anweisungsfolgen usw. binden
- beim Übersetzen (*compile time*), Binden (*link time*), Laden (*load time*) und spätestens beim Ausführen (*run time*) eines Programms
z.B. Variablen und Funktionen an Speicheradressen binden

Die Wahl der Bindungszeitpunkte hat einen großen Einfluss auf einerseits die Effizienz und andererseits die Flexibilität von Programmen.

frühere Bindung fördert die Effizienz, spätere Bindung die Flexibilität

Bindungszeitpunkte (2)

Oft wird nur grob zwischen statischer und dynamischer Bindung unterschieden:

- **statische Bindung** (*frühe Bindung*):
alle Bindungen, die vor der Laufzeit (run time) passieren
- **dynamische Bindung** (*späte Bindung*):
Bindung zur Laufzeit

Geltungsbereiche von Bindungen (Scopes)

Der Textbereich eines Programms, in dem die Bindung zwischen einem Namen und einem Objekt aktiv ist, wird als der **Scope** der Bindung bezeichnet.

*umgekehrt werden auch Textbereiche selbst als Scopes bezeichnet,
wenn sie einen Geltungsbereich von Bindungen definieren*

Objekt wieder im allgemeinen Sinn: etwas, das Speicher belegt

- **statische Scope-Ermittlung** (*static scoping, lexical scoping*)
zur Compile-Zeit wird aus der Schachtelung der Blöcke im Programmtext ein Baum von Scopes bestimmt und darin die geltende Bindung von innen nach außen gesucht, d.h. ausgehend vom aktuellen Scope in Richtung Wurzel
die übliche Lösung in modernen Programmiersprachen
- **dynamische Scope-Ermittlung** (*dynamic scoping*)
die geltende Bindung wird entlang der Stack-Frames der Aufrufsequenz gesucht
*Programme sind dadurch schwieriger zu verstehen
und Zugriffe auf nicht-lokale Variablen brauchen mehr Laufzeit*

Scopes: Beispiel (1)

- Programm in einer fiktiven Programmiersprache:

```
n : integer          ← globale Variable
procedure f()
    n := 1

procedure g()
    n : integer          ← lokale Variable
    f()

n := 2
f()
print n
n := 3
g()
print n
```

- Ausgaben des Programms:

11 *bei statischer Scope-Ermittlung*

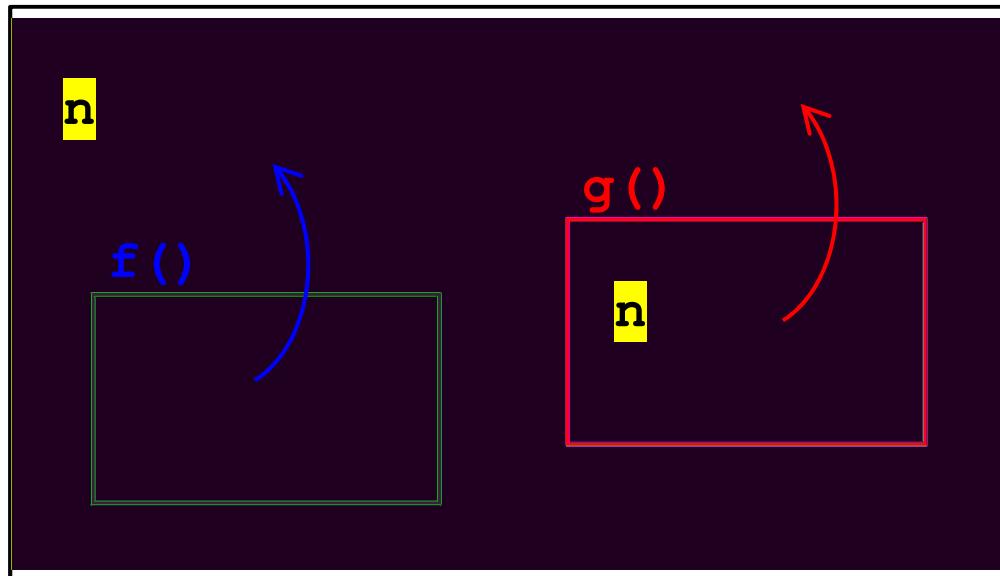
13 *bei dynamischer Scope-Ermittlung*

Scopes: Beispiel (2)

- statische Scope-Ermittlung:

Programme sind in geschachtelte Textblöcke gegliedert.

Bindungen, die im aktuellen Block fehlen, werden aus dem umgebenden Block bezogen, usw.



*in **f** wird **n** an die globale Variable gebunden*

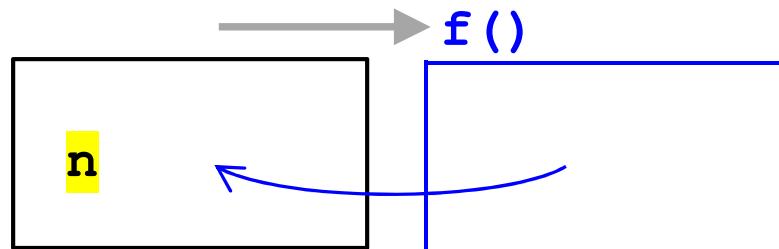
*in **g** wird **n** an die lokale Variable gebunden, die die globale Variable verdeckt (die Bindung von **n** an die globale Variable ist innerhalb von **g** inaktiv)*

Scopes: Beispiel (2)

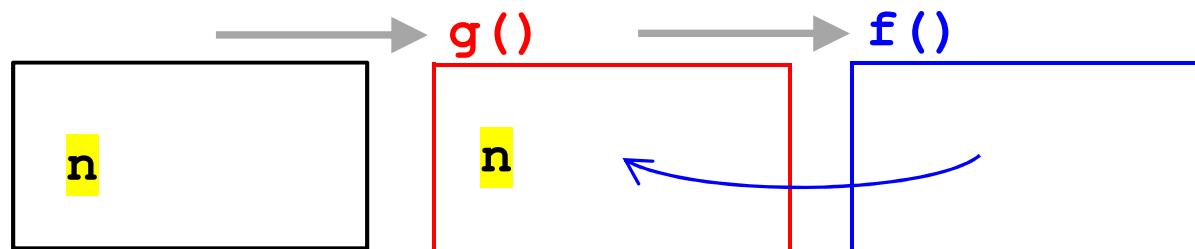
- dynamisch Scope-Ermittlung:

Bindungen, die im aktuellen Frame fehlen, werden aus dem vorhergehenden Frame (= dem Frame des Aufrufers) bezogen, usw.

bei Aufruf von `f` aus dem Hauptprogramm Bindung von `n` in `f` an die globale Variable



bei Aufruf von `f` aus `g` Bindung von `n` in `f` an die lokale Variable in `g`



Lebensdauer von Objekten (object lifetime)

Mit Objekten sind hier Gegenstände gemeint, die zur Laufzeit Speicher belegen.
das ist ein allgemeinerer Objektbegriff als der der objektorientierten Programmierung

Die Objektlebensdauer hängt von der Art der Speicherverwaltung ab:

- **statische Allokierung**

während der gesamten Programmlaufzeit feste absolute Adressen
verwendet für globale Variablen, Programmcode, unterstützende Daten für Debugging und dynamische Typprüfung usw.

- **Stack-basierte Allokierung**

Allokierung und Deallocierung in Last-in- / First-out-Reihenfolge
im Zusammenhang mit Funktionsaufrufen
verwendet für Parameter und lokale Variablen von Funktionen

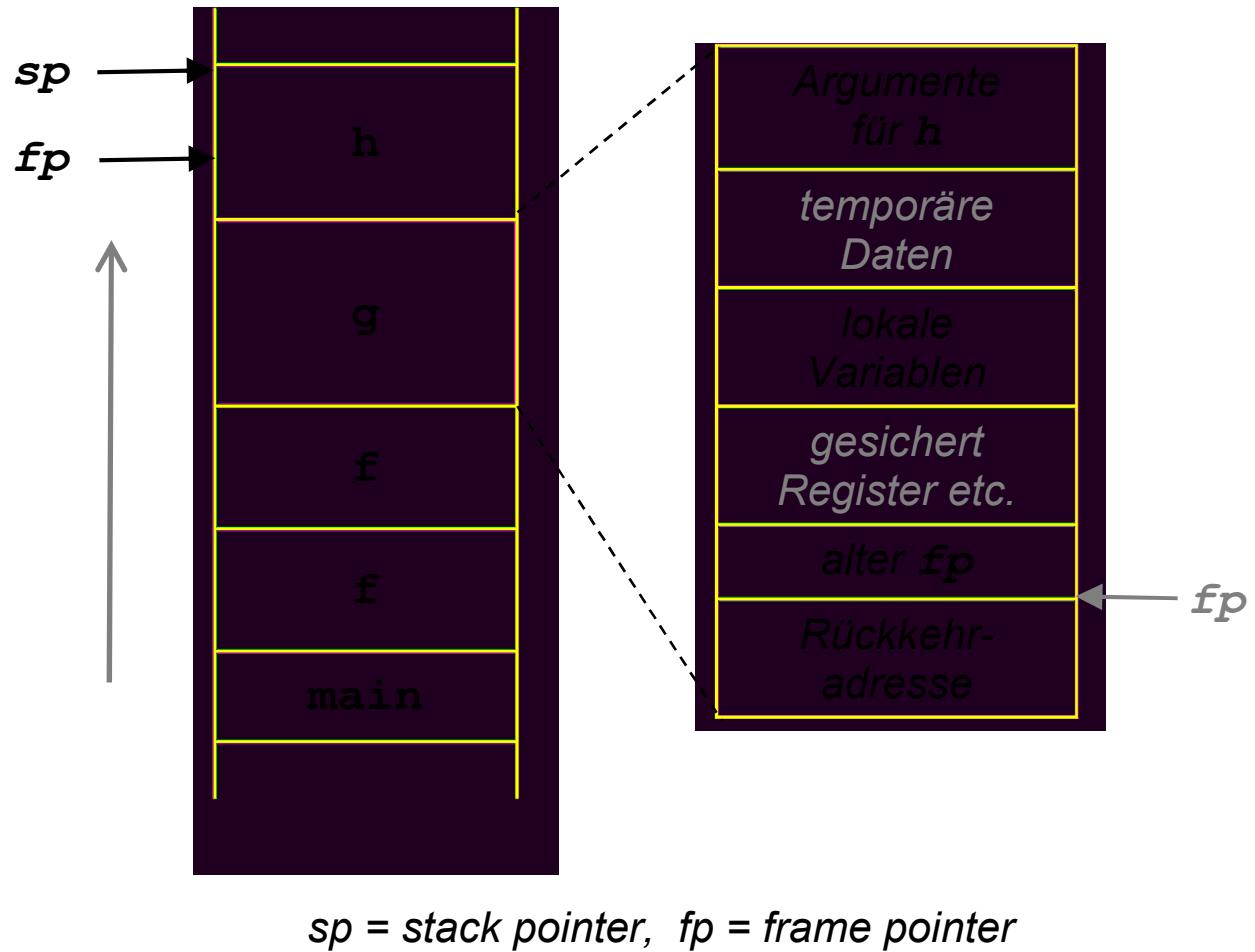
- **Heap-basierte Allokierung**

Allokierungen und Deallocierungen zu beliebigen Zeitpunkten
verwendet z.B. für Strings, Arrays, dynamische Datenstrukturen wie Listen usw.

Stack-basierte Allokierung (1)

Bei jedem Aufruf einer Funktion wird ein **Frame** auf dem Stack allokiert:

```
void h(...) {  
    ...  
}  
  
void g() {  
    h(...);  
}  
  
void f() {  
    if (...)  
        f();  
    else  
        g();  
}  
  
void main() {  
    f();  
}
```



Stack-basierte Allokierung (2)

Aufrufsequenz beim Aufrufer:

1. Register etc. sichern
2. Aufrufargumente auf den Stack legen
3. Unterprogrammsprung (*legt Rückkehradresse auf den Stack*)
4. Rückgabewert entgegennehmen
5. Register etc. wiederherstellen

Aufgerufene Funktion:

1. Frame allokieren
($sp -= frame\ size$)
2. Frame-Pointer sichern und dann aktualisieren
3. Register etc. sichern
4. Rumpf ausführen
5. Rückgabewert bereitstellen
6. Register etc. wiederherstellen
7. Frame-Pointer wiederherstellen
8. Frame deallozieren
($sp += frame\ size$)
9. Rücksprung

Prolog

Epilog

Stack-basierte Allokierung (3)

Nutzen und Vorteile der Stack-basierten Allokierung:

- ermöglicht rekursive Funktionen
weil pro Frame jeweils eine Instanz der lokalen Variablen und Parametern
- einfach und effizient
durch Register (fp, sp, ...) und Maschinenbefehle unterstützt
keine Fragmentierung des Speichers

Heap-basierte Allokierung (1)

Die Flexibilität von Allokierungen und Deallocierungen zu beliebigen Zeitpunkten macht die **Heap-Verwaltung** komplex.

schwieriger Kompromiss zwischen Laufzeit, Platzbedarf und Fehlerrisiko

- beim **Allokieren** muss ein passendes freies Stück gesucht und gegebenenfalls in ein belegtes Stück und ein freies Reststück zerlegt werden

*First-fit- oder Best-fit-Suche in einer Liste freier Speicherstücke
(Aufwand $O(n)$ bei n freien Speicherstücken)*

oder Verwaltung getrennter Pools für vorgegebene Stückelungen mit entsprechender Aufrundung der Speicheranforderungen (z.B. beim schnellen Buddy-Verfahren nur Zweierpotenzen als Stückelung)

- beim **Deallocieren** sollte versucht werden, benachbarte freie Stücke wieder zusammenzufassen

dafür gegebenenfalls Nachbarschaftslisten erforderlich (beim Buddy-Verfahren sind Nachbarschaften per Adressrechnung bestimmbar)

Heap-basierte Allokierung (2)

- sehr unterschiedliche Größen und Lebensdauern von Speicherstücken führen zu einer Fragmentierung des Heaps:

externe Fragmentierung

durch die verstreute Lage allokiertes Speicherstücke

interne Fragmentierung

durch Aufrundung von Speicheranforderungen auf vorgegebene Stückelungen

dadurch können große Speicheranforderungen eventuell nicht befriedigt werden, obwohl in Summe eigentlich genug Speicher frei wäre

- vergessenes Deallocieren führt zu **Speicherlecks**, verfrühtes Deallocieren zu **dangling Pointers**

ein richtiger Zeitpunkt kann je nach Programmiersprache eventuell nur automatisch mittels Garbage-Collection bestimmt werden

Garbage-Collection: Idee

Ein **Garbage-Collector** sorgt auf dem Heap für die Wiederverwendung allokierten Speicherstücke, sobald diese nicht mehr verwendet werden.

Das automatisierte Deallocieren verhindert Speicherlecks und dangling Pointers.

Das Zusammenschieben verbleibender belegter Speicherstücke wirkt zusätzlich der Fragmentierung des Heaps entgegen.

Technische Voraussetzungen:

- der Garbage-Collector muss zwischen belegten und freien Speicherstücken im Heap unterscheiden können
- der Garbage-Collector muss erkennen können, welche belegten Speicherstücke im Programm aktuell noch verwendet werden
*dazu muss er Referenz- / Zeiger-Variablen im globalen Datensegment, in den aktiven Stack-Frames und in belegten Heap-Stücken als solche erkennen können
(bei C und C++ ist das z.B. nicht exakt machbar)*

Garbage-Collection: Mark-and-Sweep

Das **Mark-and-Sweep-Verfahren** arbeitet in zwei Phasen:

- Markierungsphase (*mark*):
Alle vom Root-Set (= globale plus lokale Variablen) aus direkt oder indirekt erreichbaren Speicherstücke markieren.
- Bereinigungsphase (*sweep*):
Alle nicht markierten Speicherstücke deallozieren (dabei gegebenenfalls Zusammenfassung mit benachbarten bereits deallozierten Speicherstücken).
Bei allen markierten Speicherstücken die Markierung entfernen.

Bewertung:

- nur begrenzt wirksam gegen externe Fragmentierung,
weil erreichbare Speicherstücke niemals verschoben werden
- Sweep-Phase braucht viel Laufzeit, weil sie alle allokierten Speicherstücke bearbeitet, nicht nur die erreichbaren

Garbage-Collection: Stop-and-Copy

Das **Stop-and-Copy-Verfahren** teilt den Heap in zwei gleich große Bereiche:

- zwischen zwei Bereinigungen immer nur aus einem der Bereiche allokieren
- bei einer Bereinigung alle vom Root-Set aus erreichbaren Speicherstücke in den zweiten Bereich umkopieren
 - dabei jeweils die neue Adresse in der referenzierenden Variablen speichern und zusätzlich im ursprünglichen Speicherstück hinterlegen
- in allen umkopierten Speicherstücken die enthaltenen Zeiger auf die hinterlegten neuen Adressen umsetzen
- zuletzt die Rollen der beiden Bereiche tauschen

Bewertung:

- sehr wirksam gegen externe Fragmentierung, weil Speicherstücke beim Kopieren ohne Lücken abgelegt werden
- relativ laufzeiteffizient, weil nur erreichbare Speicherstücke betrachtet werden (typischerweise die deutliche Minderheit der allokierten Speicherstücke)

Garbage-Collection: Mehrgenerationen-Heap

Ein **Mehrgenerationen-Heap** verwaltet Speicherstücke, die seit der letzten Bereinigung neu allokiert wurden (*junge Generation*) und solche, die schon eine oder mehrere Bereinigungen überlebt haben (*alte Generation*), in getrennten Speicherbereichen:

- junge Generation mit Stop-and-Copy bereinigen
Speicherstücke, die eine bestimmte Anzahl Bereinigungen überlebt haben, in die alte Generation verlagern
- gelegentlich alte Generation mit Mark-and-Sweep bereinigen

Bewertung:

- vermeidet gegenüber reinem Stop-and-Copy das permanente Hin- und Her-Kopieren langlebiger Speicherstücke
- effizienter Umgang mit generationsübergreifenden Referenzen etwas aufwendig
die Speicherstücke der alten Generation erweitern im Prinzip das Root-Set für die Erreichbarkeitsprüfung in der jungen Generation

Bindung von lokalen Variablen (1)

Bei lokalen Variablen und Parametern einer Funktion stack-basierte Allokierung und Bindung der Namen zur Compile-Zeit:

- die Namen werden an Adressen relativ zum Framepointer gebunden

*der Compiler bestimmt für jede Funktion das Layout ihres Frames,
daraus ergibt sich für jeden Parameter und jede Variable ein bestimmter Offset,
also bei einem Stack, der in Richtung der kleinen Adressen wächst:*

Parameter p: $fp + offset_p$

lokale Variable v: $fp - offset_v$

- bei Programmiersprachen mit geschachtelten Funktionsdefinitionen wird zusätzlich eine Verkettung der Frames auf dem Stack genutzt (*static links*)

*über den static link im Frame werden die Parameter und lokalen Variablen
der umgebenden Funktion adressiert (usw. bei tieferer Verschachtelung):*

Parameter q: $ (fp + offset_{static\ link}) + offset_q$*

lokale Variable w: $ (fp + offset_{static\ link}) - offset_w$*

Bindung von lokalen Variablen (2)

Der Scope der Bindung einer lokalen Variablen ist maximal der Funktionsrumpf, weil der die Lebensdauer des Stack-Frames beschränkt.

Je nach Programmiersprachen kann der Scope aber auch kleiner sein:

- der Scope beginnt eventuell erst ab der Zeile der Variablendefinition
z.B. bei C (ab C99) und Java, aber nicht bei Python
- der Scope kann auf einen umschließenden Anweisungsblock beschränkt sein
z.B. bei C und Java definieren geschweifte Klammern einen Scope
- der Scope kann Unterbrechungen haben
bei geschachtelten Scopes kann eine Bindung in einem eingebetteten Scope eine Bindung des umgebenden Scopes mit gleichem Namen verdecken

Lokale Variablen mit nicht überlappenden Scopes kann der Compiler an dieselbe Adresse im Frame binden, um Speicherplatz zu sparen.

Bindung von Funktionen

Bei Funktionen (und globalen Variablen) statische Allokierung und Bindung der Namen zur Link-Zeit.

je nach Implementierung Bindung an eine absolute oder relative Adresse

Der Scope der Bindung ist die gesamte Programmlaufzeit, kann aber je nach Programmiersprache Unterbrechungen haben:

- globale Namen können in einigen Programmteilen verdeckt sein
- die Bindung modul-privater Funktionen ist außerhalb des Moduls inaktiv
z.B. static markierte Funktionen in C, Funktionen in anonymem namespace in C++
- Bindungen aus anderen Übersetzungseinheiten müssen explizit aktiviert werden
z.B. Prototypen in C

Bindung an Heap-Objekte

Adressen von Objekten auf dem Heap werden immer erst zur Laufzeit bekannt. Deshalb sind Zeiger bzw. Referenzen (je nach Programmiersprache) erforderlich, die die Heapadresse des Objekts aufnehmen.

Die Namen werden zur Compile-Zeit an die Zeiger bzw. Referenzen gebunden. Diese Indirektion kann die bereits genannten Probleme verursachen:

- **dangling pointer / reference**

Zeiger / Referenz existiert noch, das referenzierte Objekt aber nicht mehr

- **memory leak**

Heap-Objekt existiert noch, aber es gibt dazu keine Zeiger / Referenzen mehr

beide Probleme lassen sich durch die automatisierte Freigabe von Heapspeicher per Garbage-Collector vermeiden, oder wie in C++ per intelligente Zeiger deutlich reduzieren

Aliase und Überladung

Die Bindung zwischen Namen und Objekten ist nicht immer eine 1:1-Beziehung.

- **Aliase:**
es können mehrere Namen an dasselbe Objekt gebunden werden (n:1)
immer der Fall bei indirekter Bindung über Zeiger / Referenzen
- **Überladung (overloading):**
ein Name wird kontextabhängig an verschiedene Objekte gebunden (1:m)
der Aufrufname einer Funktion wird abhängig von Anzahl und Typ der Aufrufargumente an verschiedene Implementierungen gebunden

Bindungsumgebungen (*referencing environments*)

Die **Bindungsumgebung** (auch: *der Kontext*) einer Anweisung ist die Menge aller bei der Ausführung der Anweisung aktiven Bindungen.

- bei der funktionalen Programmierung wird die Kombination aus einer Funktion und einer Bindungsumgebung als **Closure** bezeichnet

Closures treten auf, wenn Funktionen als Aufrufargumente an Funktionen übergeben oder als Rückgabewerte von Funktionsaufrufen geliefert werden

die Funktionen sind dabei oft Lambdas, d.h. sie sind anonyme Funktionen

- die von der Bindungsumgebung einer Closure zugelieferten Variablen werden als **freie Variablen** der zugehörigen Funktion bezeichnet

im Gegensatz zu den Aufrufparametern und den lokalen Variablen

Closures: Beispiele

Beispiele für Closures in Python:

```
def f(c) :           Closure wird mit Wert 2 für x und Wert 1 für y ausgeführt
    print(c(1))

x = 2
f(lambda y: x + y) # Closure als Aufrufargument, x ist freie Variable
```



```
def g(x) :
    def h(y) :
        return x + y # x ist freie Variable von h
    return h # Closure als Rückgabewert

c = g(3)           Closure wird mit Wert 3 für x und Wert 4 für y ausgeführt
print(c(4))         (Achtung: der Wert 3 des Parameters x darf nicht im
                      Stackframe von g gespeichert sein, weil der Frame
                      hier bereits nicht mehr existiert!)
```

Sprachkonzepte

Teil 5: Typsysteme

Typprüfung, Typinferenz,
parametrische Polymorphie

Typsysteme

Das **Typsystem** einer Programmiersprache besteht aus

- einer Menge von **Typen**, die mit Sprachkonstrukten assoziiert werden können
Sprachen geben üblicherweise einige Typen vor (= primitive Datentypen) und erlauben darauf aufbauend die Konstruktion benutzerdefinierte Typen
typisierte Sprachkonstrukte sind solche, die einen Wert haben oder sich auf Werte beziehen, z.B. Literale, Variablen, Ausdrücke, Funktionen
- einer Menge von **Regeln** für die Äquivalenz, Kompatibilität und Inferenz von Typen

Äquivalenz Wann haben zwei Werte den gleichen Typ?

Kompatibilität Werte welcher Typen dürfen wo verwendet werden?

Inferenz Welchen Typ hat ein Ausdruck abhängig von seinen Bestandteilen und dem Kontext?

Typen

Es gibt verschiedene Sichten auf **Typen**, die man nach Bedarf einnehmen kann:

- **denotationale Sicht:**
ein Typ bezeichnet eine Menge von Werten
ein Sprachkonstrukt hat den gegebenen Typ, wenn sein Wert garantiert in der bezeichneten Menge liegt
- **strukturelle Sicht**
neue Typen werden aus vordefinierten Typen konstruiert
übliche Typkonstruktionen sind etwa Array und Record / Struct
- **abstraktionsbasierte Sicht**
ein Typ legt die zulässigen Operationen auf seinen Werten fest
besonders bei der objektorientierten Programmierung üblich (Interface-Methoden)

Klassifikation von Datentypen

skalare Datentypen:

- boolescher Typ
- Zeichtyp(en):
Zeichencodierung?
- Zahltypen: ganze Zahlen, Gleitkommazahlen, ...
Vorzeichen? Zahlbereiche? Genauigkeit? Dezimal?

zusammengesetzte Datentypen:

- Array
- Record / Struktur (*mathematische Sprechweise: Tupel*)
- Liste
- ...

Typprüfung

Die **Typprüfung** stellt fest, ob die Regeln zur Typkompatibilität eingehalten sind.
die Verletzung einer Regel wird als Typenkonflikt (type clash) bezeichnet

Eine Programmiersprache ist

- **stark typisiert** (*strongly typed*), wenn ihre Typprüfung garantiert, dass auf Werte nur die laut deren Typen zugelassenen Operationen anwendbar sind
*es gibt auch andere Definitionen starker Typisierung als diese von Michael L. Scott
viele Sprachen enthalten unterschiedlich stark typisierte Bereiche*
- **statisch typisiert** (*statically typed*), wenn sie stark typisiert ist und die Typprüfung (fast) vollständig zur Compilezeit stattfindet
erfordert explizite Variablen-deklarationen, erleichtert automatische Codeoptimierungen
- **dynamisch typisiert** (*dynamically typed*), wenn sie stark typisiert ist und die Typprüfung (fast) vollständig zur Laufzeit stattfindet
typisch beim Scripting, ermöglicht Verzicht auf Variablen-deklarationen

Typprüfung: Typäquivalenz

Typäquivalenz ist die einfachste Form der Typkompatibilität.

Man unterscheidet zwei Arten:

- **Namensäquivalenz**

jeder Typname steht für einen anderen Typ

*in C sind etwa `struct s { int a, b; };` und `struct t { int a, b; };`
zwei strikt verschiedene Typen*

- **strukturelle Äquivalenz**

gleich aufgebaute Typen sind unabhängig vom Namen äquivalent

*in C kann man mit `typedef` definierte Typen als strukturell äquivalent auffassen
(alternativ kann man sie aber auch als reine Aliasnamen betrachten)*

Typprüfung: Typumwandlung (1)

Typumwandlung ermöglicht Typkompatibilität für nicht äquivalente Typen:

- **implizite Typumwandlung (type coercion)**

viele Sprachen erlauben in Ausdrücken Werte mit anderen als den vom Kontext verlangten Typen

typische Beispiele sind gemischte Ausdrücke mit ganzen Zahlen und Gleitkommazahlen oder die Verwendung von Unterklassereferenzen anstelle von Oberklassereferenzen

in C++ kann man die implizite Typumwandlung für Objekte durch Konstruktoren und überladene typecast-Operatoren klassenspezifisch regeln

- **explizite Typumwandlung (type cast)**

in vielen Sprachen kann man den Typ eines Ausdrucks per Operator auf einen vom Kontext verlangten Typ wandeln

typische Beispiele sind Downcast und Crosscast von Objektreferenzen bei der objektorientierten Programmierung

Typprüfung: Typumwandlung (2)

Implementierung von Typumwandlungen:

- **Typaufprägung** (*type punning*)

der Zieltyp wird dem unveränderten Speicherinhalt einfach aufgeprägt,
d.h. der Speicherinhalt wird im Sinne des Zieltyps uminterpretiert

der Compiler muss dafür keinen Code erzeugen

z.B. Zweierkomplementdarstellung einer negativen ganzen Zahl als positive Zahl

*z.B. Teile eines Byte-Arrays als Verwaltungsstruktur mit ganzen Zahlen und Zeigern
(typisch in der Systemprogrammierung, etwa in einer Speicherverwaltung)*

- **Typabbildung**

jedem Wert des Quelltyps wird ein Wert des Zieltyps zugeordnet

der Compiler erzeugt Code, der zur Laufzeit den zugeordneten Wert bestimmt
(je nach Sprache und Typ kann es dabei auch zu Laufzeitfehlern kommen)

z.B. ganze Zahl mit 2 Byte auf 4 Byte erweitern

z.B. ganze Zahl in entsprechende Gleitkommazahl umrechnen

Typinferenz

Typinferenz ist die Herleitung des Typs eines Sprachkonstrukts aus seinen Bestandteilen und dem Kontext.

- Grundlage für die Typprüfung

Ist ein Ausdruck, der als Initialisierer / Operand / Argument auftritt, kompatibel mit dem erwarteten Typ?

- Grundlage für eine deklarationsfreie Typisierung

Welcher Typ soll einer Variablen zugeordnet werden, damit sie mit dem Wert eines angegebenen Ausdrucks initialisiert werden kann?

erspart bei komplizierten Typen Schreibarbeit und macht Code allgemeingültiger, z.B. in Java:

```
var v = new HashMap<String, Integer>(); // don't repeat yourself :-)  
for (var key : v.keySet()) { ... } // unabhängig vom Schlüsseltyp!
```

Parametrische Polymorphie (1)

Funktionen oder Datentypen sind **parametrisch polymorph**, wenn sie generisch programmiert sind, d.h. die Typen der behandelten Werte Parameter der Deklaration sind.

man muss für alle vorkommenden Typen nur eine einzige einheitliche Implementierung der Funktion bzw. des Datentyps bereitstellen

im Gegensatz dazu stellt man bei Overloading und Overriding typabhängig unterschiedliche Implementierungen von Funktionen bzw. Datentypen bereit

Typische Anwendungsgebiete:

- abstrakte Datentypen wie Listen, Maps, Mengen, ...

z.B. in Java: `class LinkedList<E> ... // Elementtyp als Typparameter E`

- Operationen und Algorithmen wie Minimum / Maximum, Suchen, Sortieren, ...

z.B. in Java: `static <T> void sort(T[] a, Comparator<? super T> c)`

Parametrische Polymorphie (2)

Für die Verwendung müssen parametrisch polymorphe Funktionen und Datentypen mit konkreten Typen instanziert werden:

- **explizite Instanziierung**

bei der Verwendung generischer Datentypen

z.B. in C++: `std::vector<int> v;` // Instanziierung mit Typparameter $T = \text{int}$

- **implizite Instanziierung**

bei der Verwendung generischer Funktionen,
realisiert mittels Typinferenz für die Aufrufargumente der Funktion

z.B. in C++: `std::min(1.2, 3.4);` // Instanziierung mit Typparameter $T = \text{double}$

Parametrische Polymorphie (3)

Implementierungsalternativen für die Instanziierung generischer Datentypen und Funktionen:

- Implementierung per **Typlösung** (*type erasure*)
der Compiler ersetzt Typparameter durch einen allgemeinen Typ und ergänzt notwendige Typanpassungen
z.B. bei Java Generics
- Implementierung per **Copy & Paste**
für jeden als Argument vorkommenden Typ erstellt der Compiler eine spezifische Kopie der Implementierung
z.B. bei C++ Templates

Parametrische Polymorphie: Constraints

Für Typargument können Einschränkungen (**Constraints**) gelten:

- bei Implementierung mit Typlöschung muss der als Argument übergebene Typ von einem allgemeinen Typ abgeleitet sein
z.B. bei Java nur Unterklassen von java.lang.Object, keine primitiven Grundtypen
- übergebene Typen müssen Operationen unterstützen, die in der generischen Implementierung genutzt werden
z.B. muss eine Ordnungsrelation implementiert sein, wenn Werte des übergebenen Typs sortiert werden

Die Einschränkungen für Typargument können explizit oder implizit gegeben sein.

Beispiel: Java Generics

Bei **Java Generics** sind Constraints an die Ableitungshierarchie von Klassen gekoppelt:

- Constraint bei einem unbeschränkter (*unbound*) Typparameter **<T>**
das Argument für T muss eine Unterklasse von `java.lang.Object` sein
- Constraint bei einem Typparameter mit oberer Schranke **<T extends S>**
das Argument für T muss eine Unter- bzw. Implementierungsklasse von S sein
die generische Implementierung darf in diesem Fall Methoden von S verwenden in der folgenden generischen Methode aus `java.util.Arrays` muss das Argument für T Comparable<C> implementieren, mit C gleich T oder Oberklasse von T, damit die Methode die natürliche Ordnung von T verwenden kann:

`static <T extends Comparable<? super T> int compare(T[] a, T[] b)`

Argument für C ist hier kein konkreter Typ, sondern ein Wildcard mit unterer Schranke T

Beispiel: C++ Templates

Bei C++ Templates sind Constraints an die verwendeten Operationen gekoppelt:

- implizite Constraints bei einem Typparameter `<typename T>`
das Argument für T muss alle Operationen unterstützen, die von der generischen Implementierung in der konkreten Anwendung genutzt werden
das Konzept wird auch "Duck Typing" genannt: alles was wie eine Ente läuft und wie eine Ente quakt, gilt als Ente
- ab C++20 auch explizite Constraints bei einem Typparameter `<Constraint T>`
das angegebene Constraint muss dazu als sogenanntes **concept** definiert sein
Sortierfunktion aus <algorithm> bis C++17 mit impliziten Constraints:
`template<typename T> void sort(T first, T last);`
das gleiche bei C++20 mit expliziten Constraints (etwas vereinfacht):
`template<std::random_access_iterator T> void sort(T first, T last);`