4 Ausklang

Inhalt des Kapitels im Überblick:

In diesem Kapitel sollen zum Ausklang der Vorlesung folgende Schwerpunkte behandelt werden:

- Software-Entwicklungsprozess
- Verarbeitung von Programmiersprachen: Übersetzer, Interpreter
- Historisches zu Programmiersprachen

Gliederung des Kapitels:

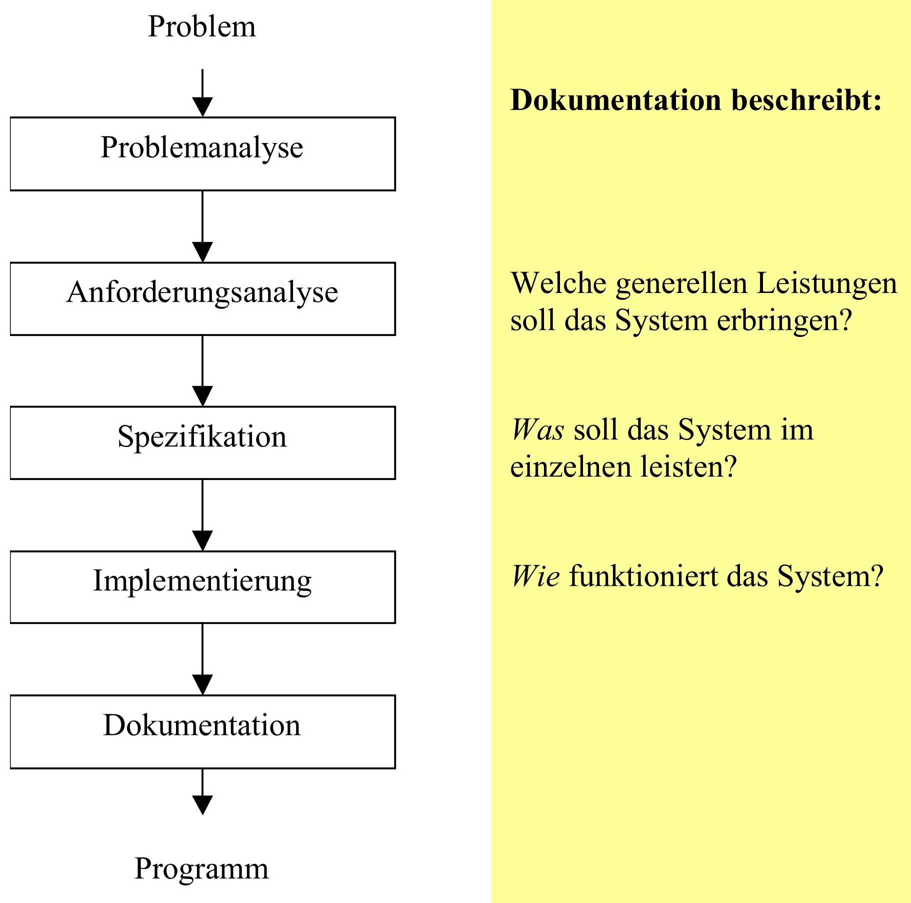
4 Ausklang 1
4.1 Software-Entwicklungsprozess 2
4.1.1 Phasen des SW-Entwicklungsprozesses 2
4.1.2 Korrektheit von Programmen 2
4.2 Verarbeitung von Programmiersprachen 3
4.2.1 Programmübersetzung 3
4.2.2 Interpreter 4
4.3 Historisches zu Programmiersprachen 4
4.4 Zusammenfassung und Ausblick auf ALP 3 6

4.1 Software-Entwicklungsprozess

Es gibt keinen Algorithmus zum Schreiben eines Programms bzw. Algorithmus.

4.1.1 Phasen des SW-Entwicklungsprozesses

Die Entwicklung einer Software sollte im Idealfall folgende Phasen durchlaufen:



4.1.2 Korrektheit von Programmen

Man unterscheidet:

- syntaktische Korrektheit
- semantische Korrektheit (sofern Programm syntaktisch korrekt)

Wie kommt man zu einem korrekten Programm?

- Testen von Programmen:
  - generiere Testfälle, die alle logischen Verzweigungen eines Programms repräsentieren

  - aber Dijkstra: Mit dem Testen kann man die *Fehlerhaftigkeit* eines Programmes beweisen, nicht jedoch die *Fehlerfreiheit*.
- Formale Verifikation:
  - Voraussetzung: Problem wurde formal spezifiziert: Mittels Aussagen- bzw. Prädikatenlogischer Formeln
  - Verifikation: Mittels Ableitungsregeln wird die Erfüllbarkeit einer Klausel gezeigt.

4.2 Verarbeitung von Programmiersprachen

4.2.1 Programmübersetzung

Präprozessor → Compiler → Binder und Lader

Präprozessor:

- Vorbereitung des Quellcodes für Compiler
- textuelle Ersetzungen von Codesegmenten
- Makro-Ersetzung und Einkopieren von Deklarationsdateien bei C

Beispiel:

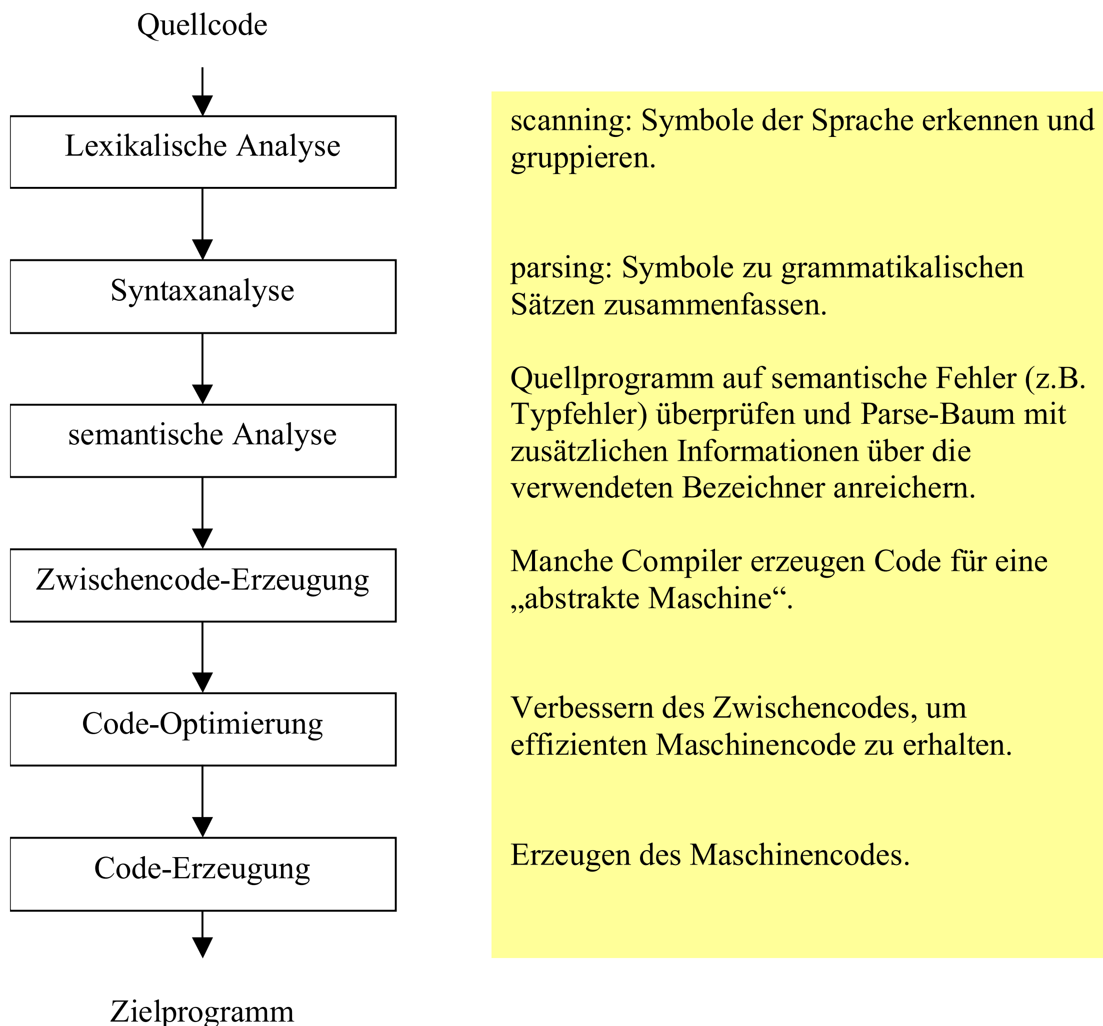
```
#include <stdio.h> // Einkopieren von stdio.h

#define ende(x) ((x=='\0')||(x=='\n')||(x=='\r')||(x=='\t')) // Makro
...
```

`if (ende(zeichen))` wird ersetzt zu

```
if ((zeichen =='\0')||( zeichen =='\n')||( zeichen =='\r')||( zeichen =='\t'))
```

Compiler:

- Überführen des in einer höheren Programmiersprache formulierten Programms (Algorithmus) in eine andere Sprache, z.B. eine Maschinensprache
- Erzeugen des Codes der Zielsprache. Phasen der Codeerzeugung:



Binder und Lader:

- Zusammenfassen verschiedener Maschinencode-Fragmente (mit relativen Adressen) zu einem ausführbaren Programm (z.B. Code aus Bibliotheken und eigener Code)
- Umwandeln relativer in absolute Adressen und Laden des Programms an eine geeignete Stelle im Hauptspeicher

4.2.2 Interpreter

Ein Interpreter analysiert wie ein Compiler den Quelltext, führt aber keine vollständige Übersetzung in Maschinensprache durch.

- Programmtext wird entweder unmittelbar ausgeführt oder
- in einen Zwischencode übersetzt, der den Zwischencode interpretiert.

Beispiele: Perl, Basic

4.3 Historisches zu Programmiersprachen

Anfänge:

- Programmierung in Maschinen- bzw. Assemblersprache

- Geringes Abstraktionsniveau, maschinenabhängig

Entwicklung problemorientierter Sprachen für wissenschaftlich-technische Aufgaben und algorithmisches Problemlösen:

- Abstraktion von der Maschine
- Sprachelemente für „höhere“ Operationen und Datentypen (Integer, Schleifen)

Problemorientierte Sprachen:

Jahr	Sprache	Entwickler	Anwendung/Merkmale	Vorläufer
ab 1954	Fortran (Formula Translator)	John Backus (IBM)	für numerisch-wissenschaftliche Anwendungen	
ab 1958	Algol (60) (Algorithmic Language)	firmenunabhängige Entwicklung durch Hochschulen	prozedurale Sprache, Blockstrukturen, Anweisungen, Ausdrücke, Typkonzepte, großer Einfluss auf Programmiersprachenentwicklung, geringe praktische Bedeutung	
ab 1960	Cobol (Common Business Oriented Language)	firmenunabhängige Standardisierung	betriebswirtschaftliche Anwendungen, besondere Berücksichtigung von Ein-/Ausgabeoperationen, hohe sprachliche Redundanz (230 reservierte Wörter)	
ab 1960	Lisp (List Processing Language)	MIT	funktionale Programmiersprache, große Bedeutung in der künstlichen Intelligenz, noch heute viel Benutzt (z.B. im Emacs-Editor), Rückbesinnung auf λ-Kalkül, Speicherverwaltung und Garbage Collector	
ab 1966	PL/I (Programming Language I)	IBM	Versuch der Verbindung von numerisch und kommerziell orientierten Programmiersprachen und ALGOL, erste Sprache mit formaler Semantik	Algol 60, Cobol
ab 1967	Simula 67	Nygaard, Dahl	erste objektorientierte Programmiersprache, `class` als neues Sprachelement, Konstruktion von Systemen als Motivation	Algol 60
ab 1970	C	Kernighan, Richie, ATT Labs	ursprünglich reine Systemimplementierungssprache (für UNIX) entwickelt, große Bedeutung durch Verbreitung von UNIX-Beriebssystemen	Algol 60

ab 1970	PASCAL	Nikolaus Wirth, ETH Zürich	als Ausbildungssprache entwickelt, weiterentwickelt unter SW-Technik-Aspekten zu MODULA-2	Algol 60
ca. 1973	PROLOG (Programming in Logic)	A. Colmerauer	Deklarative Formulierung von Algorithmen, Problem-beschreibung statt Ablauf-steuerung, Programm als Menge von Aussagen (Fakten) und Regeln	
1978	FP (Functional Programming)	John Backus (IBM)	Sprachkonzept, Grundlage für moderne funktionale Sprachen, Plädoyer gegen zustands-behaftetes Programmieren	
1980	Smalltalk	Adele Goldberg	objektorientierte Sprache, besonderes für grafische Oberflächen	Simula 67
1980	Ada	vom amerikan. Verteidigungsministerium initiiert		Pascal
ab 1992	Java	Sun Microsystems	„Internet-Programmiersprache“	C++, Smalltalk
2000	C# (sprich: C Sharp)	Microsoft	objektorientierte Sprache	Java

Einflüsse auf die Wahl von Programmiersprachen

- Betriebssysteme: UNIX → C
- Softwarekonstruktion: Objektorientierung → C++, Java
- Plattformunabhängigkeit → Java
- Zuverlässigkeit/Korrektheit → Ada
- Produktivität der Softwareerstellung → Visual C, Visual Basic, Java
- Formale Spezifikation und Verifikation → funktionale Sprachen
- Hohe Performance → Assembler, C

4.4 Zusammenfassung und Ausblick auf ALP 3

Behandelt wurden:

- Imperative Programmierung: Einführung, imperativer Algorithmusbegriff; Grundelemente von Algorithmen bzw. Programmen; Einfache und zusammengesetzte Datentypen; Variablen, Zuweisungen und Konstanten; Ausdrücke und Operatoren; Lebensdauer und Sichtbarkeit (Scopes); Kontrollstrukturen; Prozeduren, Funktionen, Parameterübergabemechanismen; Ausnahmen- und Fehlerbehandlung

- Wichtige Algorithmen und Strategien: Suchalgorithmen; Sortieralgorithmen; Stack; Syntaxanalyse von arithmetischen Ausdrücken; Wegsuche aus einem Labyrinth; Rekursion vs. Iteration; Laufzeitanalyse; Teile-und-Herrsche (divide-and-conquer), Versuch-Irrtum (trial-and-error), Backtracking; Entrekursivizierung rekursiver Algorithmen.
- Objektorientierte Programmierung: Klassen und Objekte; Referenz- und Wertsemantik; Garbage Collector; Klassenvariablen und –methoden; Vererbung und Mehrfachvererbung; Polymorphie; Dynamisches Binden; Überladene Methoden; Abstrakte Klassen; Interfaces; Umgang mit Klassenbibliotheken (in der Übung: Grafische Benutzeroberflächen, Datei-Ein/Ausgabe)
- Software-Entwicklungsprozess; Verarbeitung von Programmiersprachen: Übersetzer, Interpretierer, Skriptsprachen; Historisches zu Programmiersprachen

Nicht behandelt wurden:

- Graphische Notation und Programmlayout: UML-Notation
- Spezifikation und Verifikation: Spezifikation imperativer Programme, Verifikation von imperativen Programmen (Hoare-Kalkül), Schrittweise korrekte Programmentwicklung

Ausblick auf ALP 3:

- Datenabstraktion: modellierende Spezifikation mit Haskell, Abstraktionsfunktion und Repräsentationsinvariante
- Testen: Glassbox, Whitebox, Java-Testframework
- Iteratoren
- Interfaces
- Effiziente Implementierung von Mengen, Folgen, Graphen
- Bäume
- Hashfunktionen mit Laufzeitabschätzung
- Speicherverwaltung
- Objektorientierter Entwurf: UML-Notation, einfache Programmiermuster
- größere Fallstudie