Teil I

Theorie

Kapitel 1

Patternmatching

Brute Force

Einfachst Lösung, einfach einzeln von links nach rechts oder rechts nach links durch.

Knuth-Morris-Pratt

Hat Tabelle um mögliche Überschneidungen gut zu behandeln. Tabelle bauen $O(n)$ + Algorithmus selbst $O(n+m)$

Rabin-Karp

Hash basierend. Rechnet nur erstes Zeichen raus und neues dazu. $O(n)$ ungünstig (selte) $O(n*m)$

Boyer-Moore

Springt Pattern Distanz wenn möglich. Arbeitet am effizientesten wenn Zeichen nicht in Pattern vorkommt. Günstig $O(n+m)$, Ungünstig $O(n*m)$ Avg $O(n/m)$

Kapitel 2

Datenkapsel und Module

ADS

- Eine Abstrakte Datenstruktur definiert/realisiert eine Menge von „unsichtbaren" Datenobjekten zusammen mit einer Menge von exportierten Operationen (Zugriffsalgorithmen) die zur Manipulation der Datenobjekte herangezogen werden können und ggf eine Menge „unsichtbarer" Algorithmen, die von den Zugriffsalgorithmen verwendet werden können.
- Eine Datenkapsel repräsentiert eine abstrakte Datenstruktur (wir nennen Datenkapseln auch „aktive Datenobjekte")

Modul

- Ein Modul ist ein Konstrukt zur Implementierung von Datenkapseln

ADT

- Ein Abstrakter Datentyp ermöglicht es, beliebig viele Exemplare einer abstrakten Datenstruktur (d.h. aktive Datenobjekte, die alle die gleiche Struktur haben) zu erzeugen und zu verwenden

Kapitel 3

Systementwurf

Grundsätzliche Entwurfsprinzipien

Wichtigstes Prinzip zur Meisterung der Komplexität, ist die Abstraktion

- Zerlegung von Systemen in Subsysteme und Komponenten (Modellierung von System- und Komponentenschnittstellen)
- Zerlegung von Aufgaben in Teilaufgaben
- Sukzessive Konkretisierung von Subsystemen, Komponenten und erforderlichen Algorithmen und Datenstrukturen
- Vernachlässigung (Abstraktion) von (Implementierungs-/Realisierungs-)details

Betrachtung jeweils nur jener Aspekte eines Systems, die für den nächsten Lösungsschritt von Bedeutung sind

Topdown-Entwurf

- vom Abstrakten zum Konkreten
- Zerlegung eines großen Systems (einer Aufgabe) in mehrere kleinere Subsysteme (Teilaufgaben)

Bottomup-Entwurf

- vom Konkreten zum Abstrakten
- zusammenfassen mehrerer kleiner Bausteine zu einem größeren System (Subsystem)

Programmierparadigmen: Zerlegung

Aufgabenorientierte Zerlegung

Aufgaben- und modulorientierte Programmierung

- geht von Aufgabe aus und zerlegt sie in Teilaufgaben
- (hierarchische) Zerlegung eines Systems in seine funktionalen Bestandteile

im Mittelpunkt stehen die funktionalen Aspekte

TL;DR: Teilung nach Aufgaben, kann dann entweder in Schichten oder Teilbereichen behandelt werden.

Werkzeug: Stepwise Refinement

Lang:

Entwurf der Systemarchitektur durch Anwendung des Prinzips der schrittweisen aufgabengetriebenen Verfeinerung (stepwise refinement)

- Zerlege eine Aufgabe in Teilaufgaben
- Betrachte jede Teilaufgabe für sich und möglichst unabhängig von anderen Teilaufgaben
- Zerlege sie wieder in Teilaufgaben, bis diese so einfach geworden sind, dass ihre Lösung sich auf einfache Weise durch einen Algorithmus beschreiben lässt

Das bedeutet

- zunächst nur Algorithmen-Schnittstellen festzulegen
- unwichtige Einzelheiten zurückzustellen
- zuerst wichtigste Aspekte der Aufgabe zu identifizieren
- die Aufgaben-Komplexität stetig zu vermindern (Teile und herrsche !)

Verfeinerung ist ein **iterativer** Prozess!

Ziele

- Systematisierung des Entwurfsprozesses
- Finden einer günstigen Zerlegung eines Programmsystems in Prozeduren
- Meisterung der Komplexität durch Abstraktion

Vorteile

- mit etwas Übung fast mechanisch anwendbar
- universell anwendbar (auch für Teilaufgaben, die nicht durch schrittweise Verfeinerung entstanden sind)
- ermöglicht arbeitsteiligen Entwurf

Nachteile

- unterstützt die Modularisierung nicht explizit (Syntheseschritt erforderlich)
- unterstützt nicht die Zerlegung der Datenstrukturen
- konsequentes topdown-Vorgehen bei großen Aufgaben oft schwierig

Datenorientierte Zerlegung

Aufgaben- und modulorientierte Programmierung

- geht von Ein/Ausgabe-Daten aus, deren Struktur und Komplexität die Systemstruktur beeinflussen
- Zerlegung eines Systems in Teile, gemäß den zu verarbeitenden Datenstrukturen

im Mittelpunkt stehen die datenbezogenen Aspekte

TL;DR: Transformationsprozess (wie Compiler) nimmt strukturierte Daten und verarbeitet sie mit Lexer/Scanner und Parser.

Werkzeug: Attributierte Grammatik

Lang:

Das Denkmodell für den datenorientierten Entwurf in der Softwareentwicklung geht davon aus, dass aufbauend auf der Struktur (Syntax) eines Eingabedatenstroms seine Bedeutung (Semantik) und der angestrebte Transformationsprozess definiert und daraus die Systemarchitektur abgeleitet werden kann. Damit kommt der Syntaxanalyse des Eingabedatenstroms eine zentrale Bedeutung im Rahmen des Transformationsprozesses zu.

Die Konstruktion datenorientierter Systemarchitekturen erfordert die Entwicklung von Analysatoren, mit deren Hilfe die syntaktische Korrektheit der zu verarbeitenden Eingabedatenströme geprüft werden kann (den so genannten lexikalischen Analysator und den so genannten Syntaxanalysator).

Um einen Eingabedatenstrom verarbeiten zu können, d. h. die gewünschten Ergebnisse aus ihm ermitteln bzw. diesen in eine andere Gestalt transformieren zu können, ist es notwendig, auch die semantischen Aspekte des Eingabedatenstroms zu berücksichtigen (deshalb benötigen wir den so genannten Semantikauswerter

Zur Beschreibung der syntaktischen Struktur des Eingabedatenstromes können wir das aus dem Gebiet der formalen Sprachen bekannte Konzept der Grammatiken und für die Beschreibung des Transformationsprozesses das der attributierten Grammatiken heranziehen.

Geeignet für

- die Spezifikation von Softwaresystemen
- den Entwurf von Softwaresystemen
- die Dokumentation von Softwaresystemen

nutzbringend einsetzbar

- wenn im Wesentlichen ein Eingabedatenstrom vorliegt
- wenn der Eingabedatenstrom genügend strukturiert ist
- wenn im Wesentlichen der Eingabedatenstrom in eine andere Form transformiert werden soll

Vorteile

- Deskriptive Entwurfstechnik
- Modularisierungstechnik
- zwingt zur Trennung von Syntax und Semantik, Syntax ist Routine, der Entwickler kann sich auf Semantik konzentrieren
- knappe und (mit etwas Übung) gut lesbare Darstellung (hervorragendes Dokumentationsmittel)
- Systementwurf läßt sich automatisch in Programme transformieren

Nachteile

- nicht anwendbar bei mehreren parall zu verarbeitenden Eingabedatenströmen
- Anwendung setzt Grundkenntnisse des Übersetzerbaus voraus

Objektorientierte Zerlegung

Aufgaben- und modulorientierte Programmierung

- geht davon aus, dass Datenobjekte und Operationen untrennbare Einheiten bilden
- Zerlegung eines Systems in Objekte/Komponenten (= Datenobjekt und darauf anwendbare Operationen – sogenannte aktive Datenobjekte)

im Mittelpunkt steht die Auffassung, dass Datenobjekte und Operationen eine Einheit bilden

TL;DR:	Es wird versuche reale Probleme anhand von Objekten sinngemäßt zu beschreiben. Das Softwaresystem ensteht aus Objeketen die miteinander kommunizieren.
Werkzeug	UML; im weiteren Sinn: Vererbung, dynamische Bindung, Polymorphie
siehe	OOP

Architektur- und Komponenten-Design

Architektur-Design (Grobentwurf)

System-, Subsystementwurf Festlegung der System-, Subsystemstruktur (Wechselwirkungen zwischen den Komponenten), Komponentenschnittstellen

Komponenten-Design (Feinentwurf)

Komponentenentwurf. Festlegung der erforderlichen Algorithmen, Datenstrukturen, ...

Kapitel 4

Formale Sprachen und Grammatiken

Eine Grammatik G mit dem Satzsymbol S, geschrieben G(S), ist eine endliche, nicht leere Menge von Ersetzungsregeln, die zwei Arten von Symbolen enthalten: Terminalsymbole, die in Sätzen vorkommen, und Nonterminalsymbole, die der Strukturbeschreibung dienen. Das Satzsymbol S ist ein ausgezeichnetes Nonterminalsymbol, das auf mindestens einer linken Seite der Ersetzungsregeln vorkommt.

TL;DR Grammatik bildet sich aus Terminalsymbolen und Nonterminalsymbolen. Die Nonterminalsymbole müssen sich zu Terminalsymbolen auflösen.

kontextfrei

Die Bezeichnung kontextfrei für eine Grammatik drückt aus, dass in allen Symbolketten, die aus dem Satzsymbol durch Anwendung der Ersetzungsregeln abgeleitet werden können, jedes darin vorkommende Nonterminalsymbol unabhängig von den Symbolen links oder rechts davon, also unabhängig von seiner Umgebung, dem Kontext, durch eine seiner Alternativen ersetzt werden kann.

TL;DR kontextfrei = Symbole sind nicht voneinander abhängig.

LL(1)- Bedingung

Wenn bei der Analyse in jeder Situation das aktuelle Terminalsymbol (genannt Vorgriffssymbol) ausreicht, um in der aktuellen Ersetzungsregel die richtige Alternative auszuwählen, erfüllt die Grammatik die so genannte LL(1)- Bedingung . Das bedeutet, dass in jeder Ersetzungsregel der Grammatik alle Alternativen mit unterschiedlichen Terminalsymbolen beginnen. Auch bei Optionen und Wiederholungen muss mit einem Vorgriffssymbol die richtige Entscheidung getroffen werden können.

TL;DR LL(1) heißt das ein 1 Symbol ausreicht um die nächste Entscheidung zu treffen. LL steht für (**L**eft to Right, **L**eftmost derivation)

LL(1)- Bedingung prüfen

LL(1) Grammatik darf nicht **mehrdeutig**, **links rekursiv** und **links faktorisiert** sein.

...aber wir prüf ich das jetzt auf die schnelle bei der Klausur?

Schnelltest

Alle Produktionen (gscheidsprech für die Regeln) die nur eine Ableitung haben sind sowieso LL(1) und können ignoriert werden.
Also Sachen wie

```
S = B
(S Nonterminal, B ist Terminal oder Nonterminal)
```

können wir ignorieren.

Spannen werden jetzt folgendes...

```
S = A | B | C
```

A, B, C und sollten jetzt disjunkt sein (ned des selbe). A, B, C solange folgen bis sich zu Terminalsymbolen auflösen. Sollten hier die gleichen Rauskommen haben wir eine LL1 Verletzung.

Beispiel

```
S = A | a.
A = a. ➔ Eine Ableitung, einfach ignorieren
S =A | a. ➔ Wabwabwab, wenn wir A folgen kommt a heraus, damit haben wir a | a
und damit eine LL1 Verletzung!

ℹ Außerdem würde eine Mehrdeutigkeit für a enstehen
S = a
S = A = a
```

Merke Mulitple Optionen prüfen, solange alle Ergebnisse disjunkt dann LL1. Prinzipiell sollte man sich immer Fragen, könnte ich mich nur mit dem aktuellen Symbol entscheiden.

Attributierte Grammatiken

Eine attributierte Grammatik ist eine (kontextfreie) Grammatik, die mit semantischen Aktionen und Attributen so angereichert ist, dass damit der Transformations-/Verarbeitungsprozess vollständig beschrieben ist.

Vorgehensmodell zur datenorientierten Programmierung

1. Beschreiben der lexikalischen Struktur des Eingabedatenstroms (vorzugsweise mit regulären Grammatiken oder regulären Ausdrücken).
2. Generierung eines lexikalischen Analysators und Bereitstellung eines rudimentären Koordinationsrahmens, damit der lexikalische Analysator getestet werden kann.
3. Beschreiben der syntaktischen Struktur des Eingabedatenstroms mit einer kontextfreien Grammatik, die die LL(1)-Bedingung erfüllt.
4. Generierung eines Syntaxanalysators und Erweiterung des Koordinationsrahmens, damit der Syntaxanalysator getestet werden kann.
5. Festlegen der lexikalischen und der semantischen Attribute und Erweiterung der kontextfreien Grammatik mit den für den beabsichtigten Transformationsprozess notwendigen semantischen Aktionen (unter Beachtung, dass die Bedingung für eine L-attributierte Grammatik erfüllt wird). Das Ergebnis ist eine attributierte Grammatik.
6. Erweiterung des lexikalischen Analysators, so dass auch die Werte der lexikalischen Attribute ermittelt werden können und Generierung der vollständigen Parserkomponente (d.h. inklusive Semantikauswerter). Erweiterung des Koordinationsrahmens, so dass damit das gesamte Programmsystem getestet werden kann.

Kapitel 5

OOP

Arten von OOP

klassenbasiert objekt-orientiert

Beim klassenbasierten OOP wird das Objekt anhand der Klassenschablone generiert.

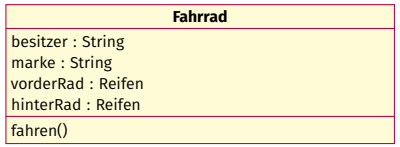
objektbasiert objekt-orientiert

Beim objektbasierten OOP wird ein Prototyp des Objektes definiert und dieser anschließend kopiert.

Klassen

Klassen sind „Schablonen“ oder eine „Rezeptur“ für Objekte. Sie beschreiben das/die aus Ihnen erzeugte(n) Objekt(e). Objekte die aus den Klassen erzeugt werden bestehen aus Methoden / Operationen und Datenkomponenten (Attributen).

Beispiel

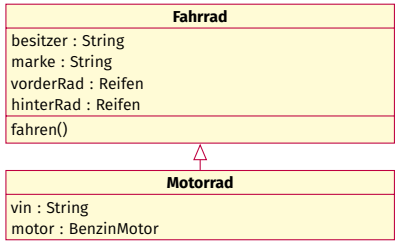


Ein Fahrradobjekt

Vererbung

Durch vereerbung oder auch „ableiten“ werden für die Kindklasse alle öffentlichen (public) und protected Methoden und Attribute/Properties/Datenkomponenten der Elternklasse frei zugänglich, die private Datenfelder existieren weiterhin sind nur für die Kindklasse unsichtbar gekapselt. Mit Vererbung entsteht ein Superset aus der Kind und Elternklasse. Die Kindklasse kann mindestens alles was die Elternklasse kann.

Beispiel



Ein Fahrrad ist ein Motorrad mit Motor, es würde sich also anbieten das Motorrad von dem Fahrrad abzuleiten.

Polymorphie

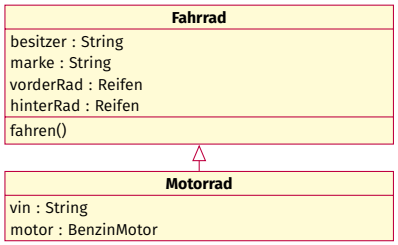
TL;DR Dadruch das eine Kindklasse ein Superset der Elternklasse ist, kann eine Variable des Types der Elternklasse aufnehmen. Dies nennt man Polymorphie.

Lang (Folien)

In der objektorientierten Programmierung bedeutet Polymorphismus, dass mit (Zeiger-)Variablen vom Typ einer (Basis-)Klasse nicht nur Objekte dieser, sondern auch Objekte aller direkt oder indirekt davon abgeleiteten Klassen referenziert werden können. Damit können solche (Zeiger-)Variablen auf Objekte unterschiedlicher Gestalt verweisen, weshalb wir in dieser Situation von Polymorphismus sprechen. Diese Objekte haben allerdings eine Gemeinsamkeit: ihre Datentypen (Klassen) stammen von derselben Basisklasse ab.

Der Datentyp einer Variablen kann sich somit zur Laufzeit ändern. Wir sprechen daher vom dynamischen Datentyp. Variable im OO Sinne besitzen also einen **statischen** und einen **dynamischen** Datentyp.

Beispiel



Fahrrad ist die Elternklasse von Motorrad. Das Motorrad hat mindestens alles was das Fahrrad hat plus einen Motor und eine VIN. Anhand dieser Informationen wissen wir, dass beides fahren() kann. nun ist es egal welches der beiden Zweiräder wir als Objekt gegeben haben, es können beide fahren. Deshalb kann man Fahrrad auch einen Motorrad Typen zuweisen und anschließend mit diesem fahren().

Dynamische Bindung

TL;DR Dynamische Bindung ermöglicht das korrekte Aufrufen einer Methode in einer polymorphen Umgebung. Sei **A** ein Objekt von der Elternklasse und **B** eines der Kindklasse, so lässt sich B auf eine Variable von Typ A zuweisen. Sollte B nun eine Methode von A überschrieben haben, so garantiert die dynamische Bindung, dass die richtige Methode aufgerufen wird.

Lang (Folien)

Unter Bindung versteht man im Kontext von Programmiersprachen die Verknüpfung des Aufrufs einer Operation (einer Prozedur, Funktion oder Methode) mit der auszuführenden Folge von Anweisungen. In Abhängigkeit vom Zeitpunkt, wann die Bindung stattfindet, kann man zwei Ausprägungen unterscheiden: Bei nicht objektorientierten Programmiersprachen kann die Verknüpfung für Prozedur- und Funktionsaufrufe entweder der Compiler zur Übersetzungszeit (wenn das gesamte Programm in einer Übersetzungseinheit vorliegt) oder der Binder (linker) zur Bindezeit herstellen. In beiden Fällen erfolgt die Bindung vor der Programmausführung, weshalb diese Art von Bindung als **statische Bindung** bezeichnet wird.

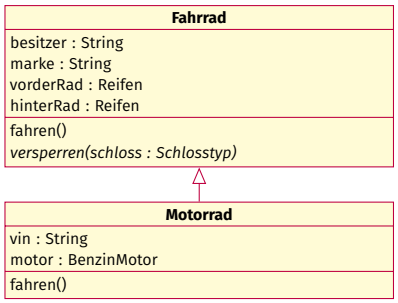
Bei objektorientierten Programmiersprachen
gibt es (wegen des objektorientierten Paradigmas) auch Situationen, in denen die Bindung von Methodenaufrufen mit einem bestimmten Codestück erst zur Laufzeit durchgeführt werden kann, weshalb diese Art von Bindung als **dynamische Bindung** bezeichnet wird.

Bindung = Verknüpfung zwischen Senden einer Nachricht und Auswahl einer Methode

Virtual Method Table

Die VMT bildet die Grundlage für die dynamische Bindung. In Ihr sind alle virtuellen Methoden gespeichert. Wird die Methode in einem Codeabschnitt aufgerufen, so wird zuerst in der VMT der original Klasse nachgeschaut (der Typ mit dem das Objekt erzeugt wurde), sollte die Methode hier nicht gefunden werden beginnt die „Super-Kaskade“ - es wird in der nächsten Elternklasse nach der Methode gesucht; sollte sie hier nicht gefunden werden so wird erneut in die nächst höhere VMT geschaut. So lang bis die Methode gefunden wurde oder keine Elternklasse mehr gefunden wird.

Beispiel



Gegeben sei die von Fahrrad abgeleitete Klasse Motorrad. wir erzeugen nun ein neues Motorradobjekt in der Variable vehicle vom Typen „Fahrrad“. (a.e. Fahrrad vehicle = new Motorrad())

Rufen wir nun bei vehicle die Methode **.fahren()** auf wird in der VMT von Motorad nach ihr gesucht, dort wird sie auch gleich gefunden und ausgeführt. Rufen wir nun die Methode **.verperren(schloss)** auf so wird in der VMT von Motorrad gesucht und nicht gefunden, als nächstes wird die VMT der Elternklasse (Fahrrad) gesucht und hier wird sie anschließend gefunden und ausgeführt.

Abstract

TL;DR Ein abstraktes Objekt gibt einen bestimmten Prozedur-teil vor, der nicht konkret implementiert ist (= abstrakt), den aber jede abgeleitete Klasse implementieren muss.

Lang (Folien)

- Zur Faktorisierung von Gemeinsamkeiten und Strukturierung ist es sinnvoll abstrakte Klassen, d.h. Datentypen zu denen es keine konkreten Objekte gibt, zu verwenden
- für Methoden solcher Klassen existiert uU keine sinnvolle Implementierung, Nachrichten sollen aber möglich sein
- es hat keinen Sinn, Objekte von abstrakten Klassen zu erzeugen
- die Methoden für die Messages abstrakter Klassen werden in den von ihnen abgeleiteten (konkreten) Klassen implementiert
- es können „Familien“ von gemeinsamen Schnittstellen realisiert werden

Faktorisierung

- Zusammenfassen von Gemeinsamkeiten verwandter Klassen
- Gemeinsamkeiten werden in (abstrakten) Basisklassen definiert

Sichtbarkeit

private	Nur innerhalb der Klasse
protected	innerhalb der Klasse und aller abgeleiteten Klassen
public	von außen Sichtbar und zugreifbar
Achtung	Datenkomponenten nur in Ausnahmefällen public setzen! Stets einen Getter / Setter implementieren. (Funktion zum setzen und Funktion zum lesen des Wertes)
Pitfall	Sichtbarkeit ist auf Klassenlevel! Das heißt, dass wenn ein anderes Objekt der Klasse als Argument übergeben wird die private Komponenten ebenfalls in der Klasse sichtbar sind!

Generics

TL;DR Generics ermöglichen es einen Typen für ein Objekt meist einen Container fix mitzugeben. Zum Beispiel für eine Liste mit Type Generic List<INTEGER> für Integerliste und List<STRING> haben die selbe Implementierung, lediglich der Typ der Datenkomponente in der Node ist anders. Wenn alle Datentypen / Objekte möglich sind ist es allgemeine Generizität, wenn man definiert das nur Objekte die von einer bestimmten Klasse abgeleitet sind oder ein bestimmten Interface implementieren, so spricht man von eingeschränkter Generizität. (Zum Beispiel: Die Liste darf nur Shapes beinhalten, also geht List<Circle> List<Rect> aber List<Int> nicht)

Lang (Folien)

Generische Klassen sollen die Möglichkeit bieten, einen Container so einzuschränken, dass er nur Objekte einer bestimmten Klasse (bzw. von davon abgeleiteten Klassen) aufnehmen kann.

Probleme bei allgemeiner Generizität

- die Parametrisierung kann mit jeder Klasse vorgenommen werden
- innerhalb der generischen Klasse können keine Annahmen darüber gemacht werden, welche Messages von Objekten der Parameterklasse akzeptiert werden
- wenn es eine Wurzelklasse Object gibt, können nur die in ihr definierten Messages verwendet werden

Eingeschränkte Generizität

Parameterklasse wird mit der Definition einer Mindestbasisklasse versehen

Kapitel 6

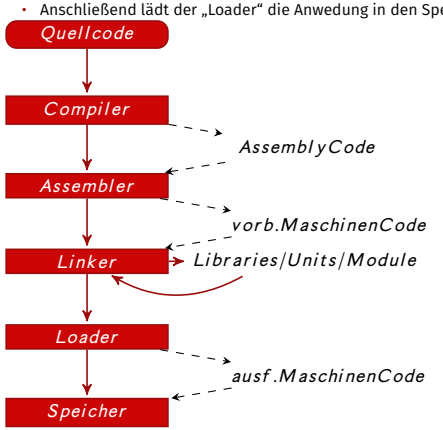
Compiler- bau / design

Der Compiler

Die Aufgabe des Compilers ist es eine Hochsprache in ausführbaren Maschinencode zu übersetzen.

Illustriert anhand von einem Pascal Programm

- Der Benutzer schreibt ein Pascal Programm (*.pas) (high-level)
- Der Pascal Compiler übersetzt es in „assembler-ähnlichen" Code (low-level)
- Ein Assembler übersetzt nun den low-level Code in Maschinencode.
- Der Linker führt nun alle vorbereiteten Teile zusammen in einen ausführbaren Maschinencode.
- Anschließend lädt der „Loader" die Anwedung in den Speicher und führt diese aus.



Vereinfachte Darstellung In dieser Darstellung ist kein Pre-Processor miteingabaut, welcher den Code vor dem eigentlichen Compiler verarbeitet. Dieser bereitet im Normalfall Codeteile die nicht von der ausführenden Maschine abhängig sind so auf, dass sie beim weiteren kompilieren für eine andere Maschine nicht neu kompiliert werden müssen.

Architektur

Ein Compiler kann in zwei Phasen geteilt werden.

„Analysis Phase" - Analytische Phase

Auch bekannt als „front-end" des Compilers. In der Analyse-Phase liest der Compiler den Quellcode ein, teilt diesen in seine Kernstücke und überprüft auf lexikalische, syntaktische oder Grammatik-Fehler. Aus der Analyse Phase heraus entsteht eine erste Repräsentation des Quellcodes und eine Symboltabelle, beides dient der Synthese-Phase als „Input".

„Synthesis Phase" - Synthese-Phase

Folglich das „back-end" des Compilers. In der Synthese-Phase wird das Programm aus den in der Analyse-Phase gesammelten Informationen generiert.

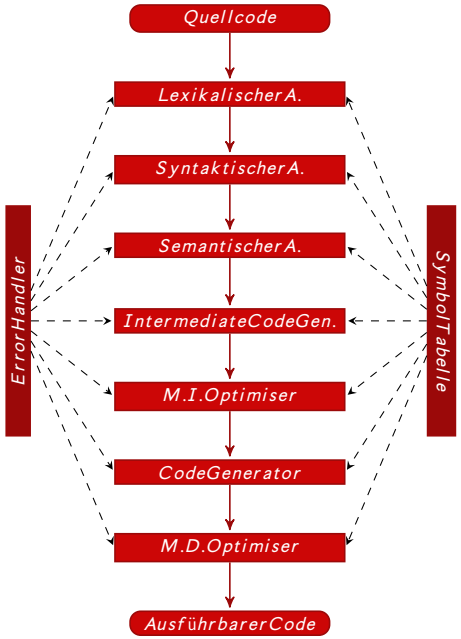
Ein Compiler kann aus mehreren „Phases" und „Passes" bestehen

Pass: Ein „Pass" ist ein kompletter Durchlauf durch den Programmcode. Der Standard-Pascalcompiler ist ein so genannter „Single-Pass-Compiler", dies ist unter anderem der Grund für „forward"-Deklarationen, dieser läuft den Quellcode genau einmal durch. „Multi-Pass-Compiler" benötigen diese im Regenfall nicht.

Phase: Eine „Phase" ist immer ein Schritt den der Compiler während des Übersetzens geht. Dieser benötigt stets Daten von der vorherigen „Phase". Ein Pass kann aus mehreren „Phases" bestehen.

„Phases"

Während des kompilierens durchläuft der Compiler verschiedene Phasen, „Phases", die jeweils auf den Daten der vorherigen Phase weiterarbeiten.



Anmerkungen M.I. Opt. (Maschine Independent) macht Code Optimierungen die nicht von der jeweiligen Maschine abhängen für die kompiliert wird. M.D. Opt. (Maschine Dependent) macht Code Optimierung speziell für die Maschine auf der der Code laufen wird.

Lexikalische Analyse

Der Scanner der ersten Phase ist nichts anderes als ein Plain-Text Scanner. Dieser nimmt das rohe Textfile und konvertiert Zeichen für Zeiche in so genannte „lexemes" bzw. Tokens. So werden aus dem Plain Text-File für den Rechner sinnvolle Symbole mit denen er weiter arbeiten kann. Eine gängige Form für diese Tokens ist (Name,Wert).

Syntaktische Analyse

Die nächste Phase ist die syntaktische Analyse oder auch „Parsing" genannt. Hier werden die Token aus der lexikalischen Analyse genommen und ein eine „Parse-Tree" bzw. „Syntax Tree" generiert. Dieser Prozess prüft die Anordnung der Tokens anhand der Grammatik, sprich er prüft die syntaktische Korrektheit der Eingaben.

Semantische Analyse

In der semantischen Analyse liest der Compiler den „Parse-Tree" / „Syntax Tree" der aus der syntaktischen Analyse ensteht und prüft ob dieser den Sprachstandards folgt. Es wird unter anderem geprüft ob die Datentypen miteinander kompatiblen sind (z.B. String wird auf Integer addiert ⟶ geht, je nach Sprache, nicht). Des Weiteren werden alle „Identifier" mit deren Typ erfasst und alle „Expressions" in denen die „Identifier" vorkommen und ob diese überhaupt bereits richtig deklariert wurden. Das Ergebnis ist eine so genannte „annotated syntax tree".

Intermediate Code Generation

Nach der semantischen Analyse generiert der Compiler einen „intermediate code" für die Maschine für die der Code kompiliert wird (x64,ARM etc). Er stellt ein Programm für eine abstrakte Maschine da, er ist das Bindeglied zwischen „high-level" und Maschinsprache. Er wird üblicherweise so generiert, dass dieser möglichst einfach in Maschinencode für die Maschine auf der er laufen soll, übersetzt werden kann.

Code Optimierung

Der nächste Schritt ist die Optimierung des „intermediate code" optimiert wird. Hier werden überflüssige „Lines of Code" entfernt oder „Statements" neu angeordnet um Performance zu gewinnen.

Code Generierung

In dieser Phase generiert der Compiler aus dem „intermediate code" den tatsächlichen Maschinencode für die Zielmaschine.

Symbol Table

Die „Symbol Table" begleitet den gesamten Prozess. In ihr stehen alle Identifiernamen mit ihren Typen. Sie ermöglicht es dem Compiler Identifier schnell zu suchen und zu finden. Ihr Aufgabe ist auch die Sichtbar/den Geltungsbereich von Variablen sicherzustellen (Scope).

Error Handler

Der Errorhandler begleitet ebenfalls den gesamten Prozess und dient dazu Fehler zu sammeln um diese anschließend den Benutzer anzeigen zu können.

Kapitel 7

Attributierte Grammatiken

EBNF

[]	Ein optionales Element kommt maximal 1x vor: Eckige Klammern Ja/Nein
{}	Mehrfache Wiederholungen: Geschweifte Klammern
()	Klammern verwendet man für das Leseverständnis
|	Alternativen werden mit Pipe-Symbol gekennzeichnet.

Nonterminalsymbole (Grammatikregeln)beginnen mit einem großen Buchstaben.

Terminalsymbolebeginnen mit einem kleinen Buchstaben.

Attribute haben kurze Namen. $\uparrow_A$usgangsattribut,$\downarrow_E$ingangsattribut

Beispiel: eine Notenliste

```
1 [ADF2]
2 Max Mustermann: NGD, BEF, SGT
3 Susi Super: BEF, GUT, SGT
4 Paul Faul: NGD, NGD
```

Aufgabe 1

Jede Notenliste hat eine Vorlesung und mehrere Schüler + Noten, jeder Schüler darf eine bis maximal drei Noten haben.

Ermittlung der Terminalsymbole
Space/Leerzeichen und \r\n / Zeilen sind keine Terminalsymbole. Ein Leerzeichen trennt stets Terminalsymbole. (Beispiel **137** ist ein Terminalsymbol aber **13 7** sind zwei Terminalsymbole)

Terminalsymbole

[	leftBr
]	rightBr
:	colon
,	comma
SGT	sgt
GUT	gut
BEF	bef
GEN	gen
NGD	ngd

Terminalklassen

name	Buchstaben + Zahlen

Achtung
Kleinbuchstaben!

Ermittlung der Non-Terminalsymbole / Grammatik
⟨*ResultList*⟩ = LVA Results.

⟨*LVA*⟩ = `leftBr name rightBr`.

⟨*Results*⟩ = Result {Result}.

⟨*Result*⟩ = Student `colon` Grades.

⟨*Grades*⟩ = Grade [comma Grade [comma Grade]].

⟨*Grade*⟩ = `sgt` | `gut` | `bef` | `gen` | `ngd`.

Aufgabe 2: Semantik : Average

Notenschnitt von letzter Note aller Schüler in Liste

Anmerkung
Laut Horn sollen wir das, wenns wirs schaffen, immer gleich so hinschreiben. Ist wurscht wenns dann eine A4 Seite braucht, wir können dann gleich hier die Semantik machen.

```
ResultList↑avg:REAL
        =
        LVA
        Results↑avg
        .

LVA
        =
        leftBr name
        rightBr
        .

Results↑avg:REAL                    <sem> VAR grade,sum,count: INTEGER; <endsem>
        =                                   <sem> count := 1; sum := 0; <endsem>
        Result↑grade                        <sem> sum := grade; <endsem>
        {Result↑grade               <sem> sum := sum + grade; count := count + 1; <endsem>
        }↑avg                               <sem>avg := sum / count; <endsem>
        .

Result↑grade:INTEGER
        =
        Student
        colon Grades↑grade
        .

Student
        =
        name
        name
        .

Grades↑last:INTEGER
        =
        Grade↑last
        [comma Grade↑last
        [comma Grade↑last
        ]
        ]
        .

Grade↑v:INTEGER
        =
        sgt                         <sem> v := 1 <endsem>
        | gut                       <sem> v := 2 <endsem>
        | bef                       <sem> v := 3 <endsem>
        | gnd                       <sem> v := 4 <endsem>
        | ngd                       <sem> v := 5 <endsem>
```

Tipps

- Terminalsymbole sind Sinneinheiten
- Ein Buchstabe ist kein Terminalsymbol! Da es keine sinntragende Einheit ist (Ausnahme z.B. "a" im Englisch)
- In unseren Angaben stehen IMMER nur Terminalsymbole!
- Kann man in 2 Einheiten gliedern:
  - Name, Lehrveranstaltungen, Noten sind Terminalklassen, da es unendliche viele Ausprägungen gibt! -> Eigennamen
  - Terminalklassen brauchen eine Bildungsregel!
  - Terminalsymbole ([],:,...)
- Frage, ob eine Liste 0, 1 oder mehrere Einträge haben kann: Immer 1 oder mehrere nehmen, weil 0 viel schwerer zu handeln ist.
- Bei der Semantik: für jedes Symbol eine eigene Zeile! außer bei einem Terminalsymbol kann man das nächste Symbol gleich dranschreiben, da ein Terminalsymbol für sich eh keine Bedeutung hat.

"Terminaklasse Name besteht aus lauter Buchstaben",
Man kann "[", "Vorlesungsname", "]" nehmen, oder "[Vorlesungsname]" als ganzes festlegen! Da hat man Spielraum. Aber auch nur, wenn KEIN Leerzeichen dazwischen steht. Sonst sinds sowieso eigene Terminalsymbole

Teil II

Code

Kapitel 8

Code Snippets

Dateizugriff

IO Result abfragen

```
PROCEDURE HandleIOResult(res : INTEGER);
BEGIN
   WriteLn('result = ', res);
   CASE res OF
      0: WriteLn('Ok');
      2: WriteLn('File not found');
      3: WriteLn('Path not found');
      4: WriteLn('Too many files open');
      5: WriteLn('Access denied');
      6: WriteLn('Invalid file handle');
      ELSE WriteLn('Generic error: ', res, '!');
   END;
END;

BEGIN
   (*$I-*)
   Assign(f,'/.../../.');
   Reset(f);
   result := IOResult;
   IF (result <> 0) THEN BEGIN
      WriteLn('IOResult =', result);
      HandleIOResult(result);
   END;
END.
```

Patternmatching

Alle Pattermatching Algorithmen unten setzen die Funktion Eq(a1,a2) : BOOLEAN vorraus, welche einen Vergleich der beiden Argumente durchführt.

Knuth-Morris-Pratt

```
PROCEDURE KnuthMorisPratt(s,p : STRING; VAR pos : INTEGER);
VAR
    next : ARRAY[1..maxStrLen] OF INTEGER;
    sLen,pLen,i,j : INTEGER;
PROCEDURE InitNextImproved;
VAR
    i,j : INTEGER;
BEGIN
  i := 1;
  j := 0;
  next[1] := 0;
  WHILE i < pLen DO
  BEGIN
    IF (j=0) OR Eq(p[i],p[j]) THEN
    BEGIN
      Inc(i);
      Inc(j);
      IF NOT Eq(p[j], p[i]) THEN
      next[i] := j
      ELSE
      next[i] := next[j];
    END ELSE
    j := next[j];
  END;
END;
BEGIN
  sLen := Length(s);
  pLen := Length(p);
  InitNextImproved;
  i := 1;
  j := 1;
  REPEAT
    IF (j = 0) OR Eq(s[i],p[j]) THEN
    BEGIN
      Inc(i);
      Inc(j);
    END ELSE
    j := next[j];
  UNTIL (i > sLen) OR (j > pLen);
  IF (j > pLen) THEN
  pos := i - pLen
  ELSE
  pos := 0;
END;
```

Rabin-Karp

```
PROCEDURE RabinKarp(s,p : STRING; VAR pos : INTEGER);
CONST
      q = 8355967;
      d = 256;
var
    sLen, pLen, i,j,dm,hp,hs : INTEGER;
    iMax : INTEGER;
    sPos : INTEGER;
BEGIN
  sLen := Length(s);
  pLen := Length(p);
  dm := 1;
  pos := 0;
  FOR i := 1 TO pLen - 1 DO
  dm := (d * dm) MOD q;
  hp := 0;
  hs := 0;
  FOR i := 1 TO pLen DO
  BEGIN
    hp := (hp * d + Ord(p[i])) MOD q;
    hs := (hs * d + Ord(s[i])) MOD q;
  END;

  i := 1;
  j := 1;
  iMax := sLen - pLen + 1;
  WHILE (i <= iMax) AND (j <= pLen) DO
  BEGIN
    IF hp = hs THEN
    BEGIN
      j := 1;
      sPos := i;
      WHILE (j <= pLen) AND Eq(p[j],s[sPos]) DO
      BEGIN
        Inc(j);
        Inc(sPos);
      END;
    END;

    hs := (hs + d * q - Ord(s[i]) * dm) MOD q;
    hs := (hs * d + ORD(s[i+pLen])) MOD q;
    Inc(i);
  END;
  IF (i > iMax) AND (j <= pLen) THEN
  pos := 0
  ELSE
  pos := i -1; (*match found *)
END;
```

Boyer Moore

```
PROCEDURE BoyerMoore(s ,p : STRING; VAR pos : INTEGER);
VAR
    skip : ARRAY[CHAR] OF INTEGER;
    sLen, pLen, i, j : INTEGER;

PROCEDURE InitSkip;
VAR
    ch : CHAR;
    j : INTEGER;
BEGIN
  FOR ch := CHR(0) TO CHR(255) DO
  skip[ch] := pLen;
  FOR j := 1 TO pLen DO
  skip[p[j]] := pLen - j;
END; (* init skip *)
BEGIN
  sLen := Length(s);
  pLen := Length(p);
  InitSkip;
  i := pLen;
  j := pLen;

  REPEAT
    IF Eq(s[i],p[j]) THEN
    BEGIN
      Dec(i);
      Dec(j);
    END
    ELSE
    BEGIN
      IF pLen - j + 1 > skip[s[i]] THEN (* missmatch s[i] not in pattern*)
      i := i + pLen - j + 1 (* skip whole pattern *)
      ELSE
      i := i + skip[s[i]];
      j := pLen;
    END;
  UNTIL (i > sLen) OR (j < 1);
  IF j < 1 THEN
  pos := i + 1 + shift
  ELSE
  pos := 0;
END; (*boyer moore *)
```

String Hashing

DJB2-custom

```
FUNCTION Shift(hash : LONGWORD; desiredLength : INTEGER) : LONGWORD;
BEGIN
  IF hash = 0 THEN
    hash := 1;
  Shift := hash;
END;

(* longword = unsinged *)
FUNCTION GetHashcodeFor(data : STRING; desiredLength : INTEGER) : LONGWORD;
VAR
  hash : LONGWORD;
  i : INTEGER;
BEGIN
  (* original start value from dan bersteins djb2 used in comp.lang.c *)
  hash := 5381;
  FOR i := 1 TO LENGTH(data) DO
  BEGIN
    (* update rev 2 back to 33 from 37*)
    hash := ((hash * 33) MOD desiredLength) + (Ord(data[i]) + (i * PosModifier));
  END;
  GetHashcodeFor := Shift((hash MOD desiredLength),desiredLength);
END;
```

DJB2-custom

```
FUNCTION Shift(hash : LONGWORD; desiredLength : INTEGER) : LONGWORD;
BEGIN
  IF hash = 0 THEN
    hash := 1;
  Shift := hash;
END;

(* longword = unsinged *)
FUNCTION GetHashcodeFor(data : STRING; desiredLength : INTEGER) : LONGWORD;
VAR
  hash : LONGWORD;
  i : INTEGER;
BEGIN
  (* original start value from dan bersteins djb2 used in comp.lang.c *)
  hash := 5381;
  FOR i := 1 TO LENGTH(data) DO
  BEGIN
    (* update rev 2 back to 33 from 37*)
    hash := ((hash * 33) MOD desiredLength) + (Ord(data[i]) + (i * PosModifier));
  END;
  GetHashcodeFor := Shift((hash MOD desiredLength),desiredLength);
END;
```

FNV1a („kleine“ Primzahl)

```
FUNCTION GetHashcodeFor(data : STRING; desiredLength : INTEGER) : LONGWORD;
VAR
  hash,prime : LONGWORD;
  i: INTEGER;
BEGIN
  hash := 2166136261;
  prime := 499;
  FOR i := 1 TO LENGTH(data) DO
  BEGIN
    hash := (hash XOR Ord(data[i]));
    hash := (hash MOD desiredLength) * prime;
  END;
  GetHashcodeFor := (hash + 1) MOD desiredLength;
END;
```

Rekursiver Abstieg

Der rekursive Abstieg ist eine Methode, die im Compilerbau angewendet wird. Ein Compiler, der diese Technik benützt, gehört zu den schnellsten Compilern überhaupt. Die Voraussetzung für den rekursiven Abstieg im Compilerbau ist, daß es sich bei der Grammatik der Programmiersprache um eine LL1-Sprache handelt.

Verwendete Variablen und Prozeduren

success	globale Variable, die mit TRUE initialisiert wird und beim ersten Fehler auf FALSE gesetzt wird – gibt an, ob kein Syntaxfehler aufgetreten ist
sy	globale Variable, die zu jedem Zeitpunkt das nächste noch nicht erkannte Terminalsymbol enthält
NewSy	Prozedur, die der (globalen) Variablen sy das nächste Terminalsymbol zuweist (sie entspricht dem "lexikalischen Analysator")

Erkennung von Terminal- und Nonterminalsymbolen

Terminalsymbol α

```
IF sy <> aSy THEN BEGIN success := FALSE; Exit; END; NewSy;
```

Nonterminalsymbol A

```
A; IF NOT success THEN Exit;
```

Achtung	Hierbei ist besonders zu beachten, dass NewSy immer nur nach der Erkennung eines Terminalsymbols aufgerufen werden darf.

Erkennung von EBNF-Regeln

Für jede Regel (also jedes Nonterminalsymbol) der Grammatik wird eine eigene Prozedur mit dem Namen dieses Nonterminalsymbols geschrieben.

S = ...

```
PROCEDURE S;
BEGIN
  ...
END; (*S*)
```

... = a A b .

```
IF sy <> aSy THEN
BEGIN
  success := FALSE; Exit
END;
NewSy;
A; IF NOT success THEN Exit;
IF sy <> bSy THEN
BEGIN
  success := FALSE; Exit
END;
NewSy;
```

... = [a] A [b] .

```
IF sy <> aSy THEN BEGIN success := FALSE; Exit END;
NewSy;
IF sy = bSy THEN BEGIN
  NewSy;
  A; IF NOT success THEN Exit;
END; (*IF*)
```

... = a [b A] .

```
IF sy = aSy THEN NewSy;
A; IF NOT success THEN Exit;
IF sy = bSy THEN NewSy;
```

... = (a | b | C) . ⟹ C = d ...

```
CASE sy OF
  aSy: NewSy;
  bSy: NewSy;
  dSy:
  BEGIN
    C; IF NOT success THEN Exit;
  END
  ELSE
  BEGIN
    success := FALSE; Exit;
  END
END; (*CASE*)
```

... = [a | b | C] . ⟹ C = d ...

```
CASE sy OF
  aSy: NewSy;
  bSy: NewSy;
  dSy:
  BEGIN
    C; IF NOT success THEN Exit;
  END
END; (*CASE*)
```

... = { a | b | C } c . ⟹ C = d ...

```
WHILE (sy = aSy) OR (sy = bSy) OR (sy = dSy) DO
BEGIN
  CASE sy OF
    aSy: NewSy;
    bSy: NewSy;
    dSy:
    BEGIN
      C; IF NOT success THEN Exit;
    END;
  END; (*CASE*)
END; (*WHILE*)
IF sy <> cSy THEN
BEGIN
  success := FALSE; Exit
END;
NewSy;
```

oder

```
WHILE sy <> cSy DO
BEGIN
  CASE sy OF
    aSy: NewSy;
    bSy: NewSy;
    dSy:
    BEGIN
      C; IF NOT success THEN Exit;
    END
    ELSE
    BEGIN
      success := FALSE; Exit;
    END
  END; (*CASE*)
END; (*WHILE*)
NewSy;
```

Inhaltsverzeichnis

I Theorie 1

1 Patternmatching **1**
- Brute Force . . . 1
- Knuth-Morris-Pratt . . . 1
- Rabin-Karp . . . 1
- Boyer-Moore . . . 1

2 Datenkapsel und Module **1**
- ADS . . . 1
- Modul . . . 1
- ADT . . . 1

3 Systementwurf **1**
- Grundsätzliche Entwurfsprinzipien . . . 1
- Topdown-Entwurf . . . 1
- Bottomup-Entwurf . . . 1
- Programmierparadigmen: Zerlegung . . . 1
  - Aufgabenorientierte Zerlegung . . . 1
  - Datenorientierte Zerlegung . . . 1
    - nutzbringend einsetzbar . . . 1
  - Objektorientierte Zerlegung . . . 2
- Architektur- und Komponenten-Design . . . 2
  - Architektur-Design (Grobentwurf) . . . 2
  - Komponenten-Design (Feinentwurf) . . . 2

4 Formale Sprachen und Grammatiken **2**
- kontextfrei . . . 2
- LL(1)- Bedingung . . . 2
  - LL(1)- Bedingung prüfen . . . 2
    - Schnelltest . . . 2
    - Beispiel . . . 2
- Attributierte Grammatiken . . . 2
- Vorgehensmodell zur datenorientierten Programmierung . . . 2

5 OOP **3**
- Arten von OOP . . . 3
  - klassenbasiert objekt-orientiert . . . 3
  - objektbasiert objekt-orientiert . . . 3
- Klassen . . . 3
- Vererbung . . . 3
- Polymorphie . . . 3
- Dynamische Bindung . . . 3
  - Virtual Method Table . . . 3
- Abstract . . . 3
- Faktorisierung . . . 3
- Sichtbarkeit . . . 3
- Generics . . . 4

6 Compiler- bau / design **5**
- Der Compiler . . . 5
- Architektur . . . 5
  - „Analysis Phase“ - Analytische Phase . . . 5
  - „Synthesis Phase“ - Synthese-Phase . . . 5
- „Phases“ . . . 5
  - Lexikalische Analyse . . . 5
  - Syntaktische Analyse . . . 5
  - Semantische Analyse . . . 5
  - Intermediate Code Generation . . . 5
  - Code Optimierung . . . 5
  - Code Generierung . . . 5
  - Symbol Table . . . 5
  - Error Handler . . . 5

7 Attributierte Grammatiken **6**
- EBNF . . . 6
- Beispiel: eine Notenliste . . . 6
  - Aufgabe 1 . . . 6
    - Ermittlung der Terminalsymbole . . . 6
    - Ermittlung der Non-Terminalsymbole / Grammatik . . . 6
  - Aufgabe 2: Semantik : Average . . . 6
- Tipps . . . 6

II Code 7

8 Code Snippets **7**
- Dateizugriff . . . 7
  - IO Result abfragen . . . 7
- Patternmatching . . . 7
  - Knuth-Morris-Pratt . . . 7
  - Rabin-Karp . . . 7
  - Boyer Moore . . . 7
- String Hashing . . . 8
  - DJB2-custom . . . 8
  - DJB2-custom . . . 8
  - FNV1a („kleine“ Primzahl) . . . 8
- Rekursiver Abstieg . . . 8
  - Verwendete Variablen und Prozeduren . . . 8
  - Erkennung von Terminal- und Nonterminalsymbolen . . . 8
    - Terminalsymbol α . . . 8
    - Nonterminalsymbol A . . . 8
  - Erkennung von EBNF-Regeln . . . 8
    - S = … . . . 8
    - … = a A b . . . 8
    - … = [a] A [b] . . . 8
    - … = a [b A] . . . 8
    - … = (a | b | C) . $\Longrightarrow$ C = d … . . . 8
    - … = [a | b | C] . $\Longrightarrow$ C = d … . . . 8
    - … = { a | b | C } c . $\Longrightarrow$ C = d … . . . 8