A++

Die kleinste Programmiersprache der Welt

Georg P. Loczewski

A++

Die kleinste Programmiersprache der Welt

Eine Programmiersprache zum Erlernen der Programmierung

Mit einer Einführung in das Lambda-Kalkül

tredition®
www.tredition.de

IMPRESSUM

A++ : Die kleinste Programmiersprache der Welt

1. Auflage 2018 – Hamburg
tredition GmbH

ISBN
978-3-7469-3098-5 (Paperback)
978-3-7469-3099-2 (Hardcover)
978-3-7469-3100-5 (e-Book)

Verlag & Druck: tredition GmbH

Meiner Frau Ursula
und meinen Söhnen Thomas und Johannes
in Liebe gewiedmet

Inhaltsverzeichnis

Vorwort xi

1 Einführung 1

1.1 Konstitutive Prinzipien in A++ . . . 1

Abstraktion . . . 1

Referenz . . . 2

Synthese . . . 2

Closure . . . 2

Lexical Scope . . . 3

2 Sprachdefinition 5

2.1 Syntax und Semantik von A++ . . . 5

Anmerkungen zur Syntax: . . . 5

2.2 Beispiele zur Syntax von A++ . . . 6

Beispiele zur Abstraktion 1. Alternative in 2.2 . . . 6

Beispiele zur Abstraktion 2. Alternative in 2.2 . . . 6

Beispiele zur Referenz 2.3 . . . 6

Beispiele zur Synthese 2.4 . . . 6

2.3 A++ Erweiterung . . . 8

Syntax von A++ mit vorgegebenen Primitiv-Abstraktionen . . . 9

Beispiele zu den Erweiterungen in A++ . . . 9

3 Erste Entfaltung von A++ 13

3.1 Programmierstil in A++ . . . 13

3.2 Grundlegende Logische Abstraktionen . . . 13

Abstraktionen ‘true’ und ‘false’ . . . 13

Abstraktionen ‘lif’ . . . 14

3.3 Erweiterte Logische Abstraktionen . . . 14

3.4 Numerische Abstraktionen . . . 15

Abstraktion für die Zahl ‘0’ . . . 15

Abstraktion für die Zahl ‘1’ 15
Abstraktion für die Zahl ‘2’ 15
Abstraktion für das Prädikat ‘zerop’ 16
Abstraktion für die Zahl ‘3’ 16
Utility-Abstraktion ‘compose’ 16
Abstraktion für die Addition 16
Abstraktion für die Inkrementierung 17
Abstraktion für die Multiplikation 17
3.5 Abstraktionen für Listen 17
Abstraktion für den Konstruktor 17
Abstraktion für den Selektor ‘lcar’ 18
Abstraktion für den Selektor ‘lcdr’ 18
Anwendung der grundlegenden Operationen für Listen 18
Abstraktion für das Ende einer Liste 18
Abstraktion für das Prädikat ‘nullp’ 19
Abstraktion für die Längenabfrage 19
Abstraktion zum Entfernen eines Objektes aus einer Liste 19
3.6 Erweiterte Arithmetische Abstraktionen 20
Abstraktion für ‘zeropair’ 20
Abstraktion für die Dekrementierung 20
Abstraktion für die Subtraktion 21
3.7 Relationale Abstraktionen 21
Abstraktion für das Prädikat ‘gleich’ 21
Abstraktion für das Prädikat ‘größer als’ 21
Abstraktion für das Prädikat ‘kleiner als’ 21
Abstraktion für das Prädikat ‘größer gleich’ 22
3.8 Funktionen Höherer Ordnung 22
Abstraktion ‘compose’ 22
Abstraktion für die ‘curry’-Funktion 23
Abstraktion für die Abbildung einer Liste 23
Abstraktion für die ‘curry map’-Funktion 23
Abstraktion für die Auswahl aus einer Liste 24
Abstraktion für die Suche nach einem Objekt in einer Liste 24
3.9 Mengen-Operationen 25
Abstraktion für das Prädikat ‘memberp’ 25
Abstraktion für das Hinzufügen eines Elementes 25
Abstraktion für die Vereinigung von Mengen 26

4 Erste Anwendung von A++ **27**
4.1 Utility-Abstraktionen . . . 27
Abstraktion für das sortierte Einfügen in eine Liste . . . 27
Abstraktion für die Sortierung . . . 27
4.2 Rekursion . . . 28
Abstraktion für die Berechnung der Fakultät . . . 29
Abstraktion für die Summation . . . 29
Abstraktion für den Zugriff auf ein Element einer Liste . . . 29
Abstraktion für die Iteration über die Elemente einer Liste . . . 30
4.3 Imperative Programmierung in A++ . . . 30
Die Abstraktion 'while' in A++ . . . 31

5 Objekt-Orientierte Anwendung von A++ **33**
5.1 Einleitung . . . 33
Bildung von Klassen . . . 33
Instanzen von Klassen . . . 33
Beispiele für Objekt-orientierung in A++ . . . 34
5.2 Erstes Beispiel zur Objektorientierung in *A++* . . . 34
Konstruktor für Objekte der Klasse "account" . . . 34
Erzeugung des Objektes "acc1" durch Aufruf von "make-account" . . . 36
Senden der Botschaft "deposit" an das Objekt "acc1" . . . 36
Ausführen der Funktion "deposit" . . . 36
Detaillierter Code in A++ . . . 40
Anmerkungen zu dem ersten Beispiel zur Objektorientierung . . . 41
Anmerkungen zum Aufruf der Funktion 'konto' . . . 42
5.3 Zweites Beispiel zur Objektorientierung in *A++* . . . 42
Anmerkung zu den einzelnen Klassen . . . 43
Beziehungen zwischen den Klassen . . . 45
Zusammenfassung . . . 45
5.4 Drittes Beispiel zur Objektorientierung in *A++* . . . 49
Anmerkung zu den einzelnen Klassen . . . 49

6 Abstraktion, Referenz und Synthese im Detail **59**
6.1 Addition der zwei Zahlen 'two' und 'three' . . . 59
Synthese von 'add' und 'two three' (1) . . . 59
Abstraktion von 'add' (1) . . . 59
Auflösung der Referenz von 'add' in [1] . . . 59
Synthese von (lambda(m n) ...) und 'two three' in [3] . . . 59

Abstraktion von ‘compose’ . . . 59
Auflösung der Referenz von ‘compose’ in [4] . . . 60
Synthese von (lambda(f g) ...) und ‘(two f) (three f)’ in [6] . . . 60
Abstraktion von three: . . . 60
Auflösung der Referenz von ‘three’ in [7] . . . 60
Synthese von (lambda(f) ...) und ‘f’ in [9] . . . 60
Synthese von (lambda(x) ...) und ‘x’ in [10] . . . 60
Abstraktion von two: . . . 60
Auflösung der Referenz von ‘two’ in [11] . . . 60
Synthese von (lambda(f) ...) und ‘f’ in [13] . . . 60
Synthese vom inneren (lambda(x) ...) und ‘(f (f (f x)))’ in [14] . . . 60
6.2 Multiplikation der zwei Zahlen ‘two’ und ‘three’ . . . 61
Synthese von mult und ‘two three’ . . . 61
Abstraktion von mult: . . . 61
Auflösung der Referenz von ‘mult’ in [17] . . . 61
Synthese von (lambda(m n) ...) und ‘two three’ in [19] . . . 61
Abstraktion von compose: . . . 61
Auflösung der Referenz von ‘compose’ in [20] . . . 61
Synthese von (lambda(f g) ...) und ‘two three’ in [22] . . . 61
Abstraktion von two: . . . 61
Abstraktion von three: . . . 61
Auflösung der Referenz von ‘two’ und ‘three’ in [23] . . . 61
Synthese vom inneren (lambda(f) ...) und ‘x’ in [26] . . . 62
Synthese von (lambda(f) ...) und ‘(lambda(x0) ...)’ in [28] . . . 62
Synthese vom inneren (lambda(x0) ...) und ‘x1’ . . . 62
Synthese von (lambda(x0) ...) und ‘(x(x(x x1)))’ . . . 62
Umbenennung der Variablen: x –> f und x1 –> x . . . 63

7 Infrastruktur für A++ 65
7.1 Support-Funktionen . . . 65
Abstraktion für die Ausgabe einer Zahl . . . 65
Abstraktion für die Ausgabe eines boole’schen Wertes . . . 65
Abstraktion für die Ausgabe von Listen . . . 65
7.2 A++ Interpreter . . . 65
Linux . . . 66
MS-Windows . . . 67
Programmbeendigung . . . 68
7.3 Initialisierungsdatei für den ARS-Interperter . . . 68
7.4 WWW-Adressen . . . 71

8 Erweiterung von A++ **73**
8.1 ARS++ . . . 73
8.2 ARSAPI . . . 74

Anhänge 77

A Das Lambda-Kalkül **77**
A.1 Syntax eines Lambda-Ausdrucks . . . 77
A.2 Begriffe und Regeln des *Lambda-Kalküls* . . . 77
Assoziativitätsregeln . . . 77
Gebundene und freie Variable . . . 78
Alpha-Konvertierung . . . 78
Beta-Reduktion . . . 78
Eta-Reduktion . . . 79
Y-Kombinator . . . 79
A.3 Beispiele für Beta-Reduktion . . . 80
Lambda-Kalkül-Programmierung in Scheme-Codierung . . . 80
Auszuwertende Lambda-Ausdrücke in Scheme-Codierung . . . 81
Basis-Abstraktionen in Scheme-Codierung . . . 81
Anwendung mit Beta-Reduktion . . . 83

B Gültigkeitsbereich von Namen **87**
B.1 Interpretation von Namen . . . 87
Dynamic Scope . . . 87
Static Scope . . . 87
Global Scope . . . 88
Local Scope . . . 88
B.2 Auswirkung der Art der Symbolinterpretation auf die Programmierung . . . 88
Auswirkung von „Dynamic Scope“ auf die Programmierung . . . 88
Auswirkung von „Static Scope“ auf die Programmierung . . . 89
Verdeutlichung der Unterschiede von „dynamic scope“ und „lexical scope“ anhand von Beispielen . . . 89

Schlusswort **93**

Biographische Daten zur Person des Autors **95**

Verzeichnis der Fundamentalbegriffe **97**

Abbildungsverzeichnis **99**

Listings **101**

Literaturverzeichnis **105**

Index **107**

Vorwort

Zweck des Buches

In diesem kleinen Büchlein, geht es primär darum an der Programmierung interessierten Leserinnen und Lesern ein Instrument vorzustellen, mit dem sie sehr schnell und sehr effizient Programmieren lernen können, ohne sich schon für eine der populären, voll-ausgebauten Programmiersprachen entscheiden zu müssen und ohne einen großen Kostenaufwand zu haben.

Dass das in diesem Taschenbuch Gelernte und Eingeübte eine solide Grundlage für das Programmieren in den meisten Sprachen ist, wird in dem 768-seitigen Buch 'Programmierung pur'(Siehe [Loc03].) ausführlich gezeigt. Dort werden die in A++ enthaltenen Denkmuster ausgeweitet auf das Programmieren in populären Sprachen, wie Java, C++, C, Python und Scheme. [1]

Thematik des Buches

Das Wesentliche der Programmierung

A++ ist die kleinste Programmiersprache der Welt, deren Sinn es ist einzig das *Wesentliche der Programmierung* darzustellen, und zwar in einer Form dass damit gearbeitet werden kann, dass man es einüben kann. So soll A++ hilfreich sein beim Erlernen des Programmierens ganz allgemein, aber auch beim Erlernen von konkreten Programmiersprachen.

Elementarteilchen der Programmierung

In **A++** werden die *Elementarteilchen der Programmierung* in reinster Form sichtbar gemacht. Man kann diese gründlich studieren, den richtigen Umgang mit ihnen einüben und sich so die wichtigsten Rüstzeuge der Programmierung aneignen.

Vereinfachung der Programmierung

In dem Bemühen, Programmierung auf das Wesentliche zu reduzieren, geht es darum, Lernende zu bewahren, sich von einer Unzahl von Vorschriften und Regeln einer bestimmten Programmiersprache die Programmierung an sich vergraulen zu lassen.

[1] Zu verschiedenen umfangreichen Fallstudien, die in 'Programmierung pur' eingesetzt werden, gehört auch der A++ - Interpreter, der selbst als Anwendung der A++ - Denkmuster in all den aufgeführten Sprachen implementiert ist.

Energien, die in den meisten Sprachen für die Beherrschung und das Einhalten der Syntax aufgebracht werden müssen, kommen in A++ der wichtigeren Aufgabe der logischen Bewältigung des zu lösenden Problems zugute.

Einfache, umfassende und mächtige Denkmuster

Es wird hier dank des Lambda-Kalküls eine Sicht der Programmierung gewonnen, die eine befreiende Wirkung hat. Das Denken wird aus den Niederungen des komplexen Regelwerks einer bestimmten Programmiersprache herausgeholt und heraufgehoben auf die Höhen eines *einfacheren*, *umfassenderen* und deshalb *mächtigeren* Denkens. Das Lambda-Kalkül bietet die theoretische Grundlage für eine solche Sicht.

Verallgemeinerung des Lambda-Kalküls

Der Name **A++** ist eine Abkürzung von *Abstraktion plus Referenz plus Synthese*. Hiermit werden die drei **Prinzipien von A++** benannt, die gleichzeitig ihr einziger Inhalt sind. Diese Prinzipien stellen eine Verallgemeinerung der Grundoperationen des Lambda-Kalküls von Alonzo Church dar. Das Lambda-Kalkül ist ein mathematisch-logisches System, das 1941 von dem Logiger **Alonzo Church** in seinem Buch: *'The Calculi of Lambda Conversion'* der Welt vorgestellt wurde. Siehe hierzu unsere kurze Zusammenfassung im Anhang A auf Seite 77.

Während das Lambda-Kalkül der Funktionalen Programmierung ihre theoretische Grundlage liefert, ist **A++** in der Lage eine *theoretische Grundlage für die drei bekanntesten Paradigmen der Programmierung* zu liefern, d.h. für die funktionale, die objekt-orientierte und die imperative Programmierung.

Verallgemeinerung der Grundoperationen des Lambda-Kalküls:

- **Abstraktion:** Etwas einen Namen geben
- **Referenz:** Auf etwas mit seinem Namen Bezug nehmen
- **Synthese:** Aus zwei oder mehr Dingen etwas Neues erzeugen

Die Primitiv-Operationen von **A++** gehen inhaltsmäßig über die Grundoperationen des Lambda-Kalküls hinaus, indem ihnen in der Anwendung in einem Programm jedwede Einschränkung genommen wird.

Die Verallgemeinerung besteht darin, dass der Begriff der Abstraktion allgemeiner als im Lambda-Kalkül definiert wird, nämlich als 'etwas einen Namen geben'. Hinter dem Namen verbergen sich alle Details des Definierten. *Eine solche Namensvergabe setzt eine explizite Namensdefinition voraus*.

Im **Lambda-Kalkül** dagegen ist eine explizite Vergabe eines Namens für eine Lambda-Abstraktion bei deren Bildung nicht vorgesehen. Dort erfolgt sie lediglich implizit bei einer Synthese von Lambda-Ausdrücken.

Die Auswirkungen dieses zunächst als klein erscheinenden Unterschiedes sind gewaltig: Während ein Ausbau des *Lambda-Kalküls* immer in die Funktionalen Programmiersprachen mündet, können in **A++** allgmeine Muster der Programmierung definiert werden, die sowohl auf die *Funktionale* Programmierung als auch auf die *Objekt-orientierte* und die *Imperative* Programmierung angewandt werden können.

Wer sich in seinem Programmieren von den Prinzipien in A++ leiten lässt, wird Programme erstellen, die nicht nur funktionieren, sondern die auch schön sind, Programme, bei denen Leser und Leserinnen mit Bewunderung die Abstraktionen und Synthesen nachempfinden und genießen können.

Erlernen von neuen Programmiersprachen

In **'Programmierung pur'** wird *erstmalig der Versuch unternommen*, den Weg zum Erlernen der Programmiersprachen Scheme, Java, Python, C und C++ anhand der mittles **A++** erarbeiteten Denkmuster aufzuzeigen. 'Programmierung pur' behandelt diese Thematik nicht nur theoretisch, sondern präsentiert umfangreiche Fallstudien, um den Bezug zur Programmierpraxis zu gewährleisten.

Adressatenkreis

Dieses Buch wendet sich ebenso wie 'Programmierung pur' an **Programmierer** und solche, die es werden wollen. Es möchte ihnen mit der speziellen Denkweise die Programmierung erleichtern, besonders auch das Erlernen neuer Sprachen. Mit der nahegelegten Sicht der Programmierung wird eine Sprachenunabhängigkeit gewonnen, ja man ist sogar offen für verschiedene Paradigmen der Programmierung. Mit der Erfahrung der gewonnenen Flexibilität ausgerüstet wird ein Programmierer oder eine Programmiererin *mit mehr Freude und größerer Effizienz* die Probleme der Programmierung meistern.

Das Buch wendet sich auch an **Anfänger der Programmierung**. Jedoch sollte ein *großes Interesse für die Programmierung* aufgrund einer persönlichen Eignung und Neigung vorhanden sein.

Zusammenfassend kann Zielgruppe des Buches wie folgt beschrieben werden:

> *Das Buch ist gedacht für Menschen, die einen Ausbildungsbedarf in den Grundlagen der Programmierung besitzen*.

- Dies sind *Studenten aller Fachrichtungen der Informatik* sowie Studenten der Mathematik und Physik.
- Dies sind *Lehrer und Schüler an Gymnasien*, die in der Oberstufe Informatikunterrich gestalten oder an ihm teilnehmen.

- Dies sind ferner alle *Angestellten in der Industrie*, die sich, aus welchen Gründen auch immer, mit der Programmierung auseinandersetzen müssen.
- *Programmierer, die bereits programmieren können*, sich aber nicht scheuen, etwas Neues kennen zu lernen, kommen als potentielle Nutznießer dieses Büchleins gewiss ebenfalls in Betracht.

Danksagung

Danken möchte ich vor allem den Vielen, die in uneigennütziger Weise durch die Bereitstellung ihrer Software zur kostenlosen Nutzung zum Zustandekommen dieses Buches beigetragen haben. Zu dieser frei verfügbaren Software gehört natürlich TEX, LATEX, teTeX, Linux, XFree86, vim, psutils, ghostview und die vielen Scheme-Implementierungen, ganz besonders GambitC und libscheme. Dann gilt mein Dank natürlich ganz besonders Alonzo Church, dem wir das Lambda-Kalkül verdanken und Guy L. Steele mit Gerald J. Sussman, die dieses Lambda-Kalkül als Ausgangsbasis für die Entwicklung ihrer Programmiersprache Scheme genommen haben. Der Dank erstreckt sich auch auf die im Literaturverzeichnis aufgeführten Autoren, die uns das Lambda-Kalkül näher gebracht haben.

Herrn Jens Toeche-Mittler möchte ich herzlich danken, von Anfang an das 'Besondere' an ARS bzw. A++ erkannt und gewürdigt zu haben.

An Fr. Anke Meenenga ergeht ebenso herzlicher Dank für den Entwurf eines zu dem Titel passenden Umschlags und an Frau Cora Toeche-Mittler, für ihre Mitwirkung an dessen Gestaltung.

Dank sagen möchte ich auch meinem Sohn Johannes für verschiedene Tips aus der Sicht eines Studierenden und meiner Frau für ihre unsägliche Geduld und ihr Verständnis.

Georg P. Loczewski Groß-Zimmern, im Dezember 2002

Kapitel 1

Einführung

1.1 Konstitutive Prinzipien in A++

A++ steht für **Abstraktion plus Referenz plus Synthese**. Diese drei Begriffe entsprechen den *sprachlichen Strukturelementen* und den *Grundoperationen in A++* und werden deshalb im nächsten Kapitel im Zusammenhang mit der Syntax der Sprache noch ausführlich behandelt.

Zu den konstitutiven Prinzipien, d.h. den Prinzipien, die A++ wesentlich zu dem machen, was es ist, gehören außerdem noch die Begriffe **'Closure'** und **'Lexical Scope'**. Wir werden sie der Reihe nach definieren und beschreiben.

Abstraktion

FUNDAMENTALBEGRIFF 1 (ABSTRAKTION)
Abstrahieren bedeutet: Etwas einen Namen geben. *Es besteht darin, etwas Komplexes zu behandeln, als wäre es etwas Einfacheres, indem Details ignoriert werden.*

Eine solche Abstraktion wird auch als **Lambda-Abstraktion** bezeichnet, wenn mit ihr die *Definition einer Funktion* verbunden ist, die zwangsläufig zur *Erzeugung einer 'Clos-*

ure' mündet. Bezüglich des letzteren Punktes siehe weiter unten die Definition des vierten konstitutiven Prinzips.

Referenz

FUNDAMENTALBEGRIFF 2 (REFERENZ)
Auf etwas, das einen Namen erhalten hat, kann jederzeit mit diesem Namen Bezug genommen werden. Diese Bezugnahme nennen wir Referenz.

Im Zusammenhang mit der Referenz ist von großer Bedeutung auf welche Namen Bezug genommen werden kann. Dies führt zum letzten konstitutiven Prinzip von A++, dem Begriff des *'Lexical Scope'*.

Synthese

FUNDAMENTALBEGRIFF 3 (SYNTHESE)
Eine Synthese zu bilden bedeutet: Zwei oder mehrere Dinge (die selbst Ergebnis einer Abstraktion sind!) miteinander zu verknüpfen um etwas Neues (Komplexes) zu schaffen.

Der Begriff der Synthese entspricht weitgehend dem des Aufrufs einer Funktion oder der Abbildung, bzw. der Applikation.

Closure

Die Bildung einer Abstraktion ist in A++ nicht ein absolutes, von Allem losgelöstes Ereignis. *Eine Abstraktion erfolgt immer in einem bestimmten Kontext*, der somit wesentlich zu der gebildeten Abstraktion gehört. Die Lambda-Abstraktion wird zum Zeitpunkt ihrer Erzeugung mit ihrem Kontext oder ihrer Umgebung verbunden. Das Resultat dieser Verkapselung wird 'Closure' genannt. Eine Closure ist eine Art der Verkapselung wie wir sie in der Objekt-Orientierung finden. In der **Objekt-Orientierung** sind in einem Objekt Daten und Funktionen verkapselt, die *Attribute des Objektes mit dessen Methoden* gemäß ausdrücklicher Festlegung in der Klassendefinition oder im Aufbau des Konstruktors.

Bei einer **Closure** dagegen erfolgt diese Verkapselung nicht durch willkürliche, ausdrückliche Definitionen, sondern *alles, was zum textlichen Umfeld einer Funktion gehört* wird automatisch in diese Verkapselung einbezogen. Wir haben deshalb in 'Programmierung pur' das *Bild einer Muschel*[1]. als Symbol für eine Closure gewählt.

Somit können wir eine Closure wie folgt definieren:

FUNDAMENTALBEGRIFF 4 (CLOSURE)
Eine Funktion wird eine "Closure" genannt, wenn sie mit der sie umgebenden Menge der Daten und Funktionen fest verkoppelt ist. Die Variablen der geerbten Umgebung werden "freie Variable" und die Argumente der Funktion werden als "gebundene Variable" bezeichnet. Die in der Funktion definierten Variablen heissen "lokale Variable". Alle Variable einer echten Closure haben unbegrenzte Lebensdauer.

Eine solche Funktion kann nur in der ihr eigenen Umgebung ausgeführt werden. Wir haben hier ein Beispiel der Verkapselung von ausführbarem Code mit den dazugehörigen Daten. So etwas Ähnliches finden wir wieder in der weiter unten beschriebenen Objekt-Orientierung.

Eine "closure" kann man sich nach ihrer Definition folgendermaßen vorstellen: Siehe hierzu Abbildung 1.1 auf der nächsten Seite. Die zwei Kreise nebeneinander sind das Symbol für

[1]Das englische Wort für Muschel ist bekanntlich 'clam'. Bei uns hat 'clam' noch sinnvoller Weise eine andere Bedeutung, nämlich: 'c-lambda-abstraction'. So bezeichnen wir in ARSAPI eine Lambda-Abstraktion in C. Siehe hierzu Abschnitt 8.2 auf Seite 74

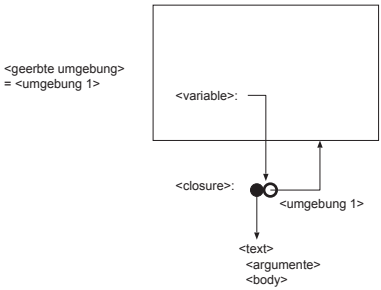

Abbildung 1.1: Definition einer “closure”

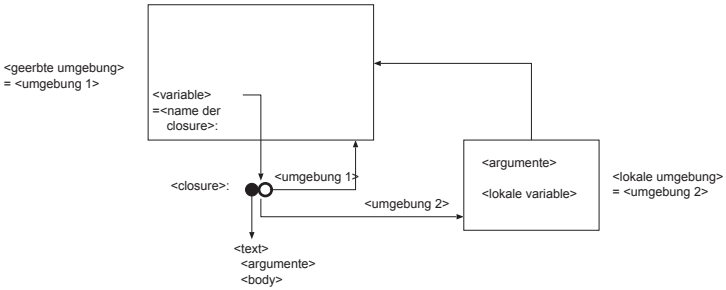

Abbildung 1.2: Aufruf einer “closure”

eine “closure”. Diese Diagrammtechnik wurde eingeführt von Harold Abelson und Gerald Jay Sussman in ihrem legendären SICP-Buch, d.h. dem offiziellen Lehrbuch der Informatik am Massachusetts Institute of Technology mit dem Titel: *Structure and Interpretation of Computer Programs*. Siehe hierzu im Literaturverzeichnis [AwJS96].

An den Programmtext fest gekoppelt ist die Umgebung des Programms, in dem die “closure” definiert wurde (“lexical scope”). Diese Heimatumgebung der “closure” hat in allen folgenden Diagrammen das Kennzeichen ’<umgebung 1>’.

Zum Zeitpunkt der Ausführung der Funktion sieht die Sache etwas anders aus. Der Funktion wird ein neuer Umgebungsbereich zugewiesen (“environment frame”), der allerdings wiederum mit der ursprünglichen Umgebung der “closure” verknüpft ist. Dieser neue Umgebungsbereich enthält die Argumente der Funktion und die in ihr definierten lokalen Variablen. In den folgenden Diagrammen trägt sie das Kennzeichen ‘<umgebung 2>’. Siehe hierzu Abbildung 1.2!

Lexical Scope

Es kann auf bereits definierte Abstraktionen über Namen Bezug genommen werden. Auf welche Namen an welcher Stelle im Programm Bezug genommen werden kann, definiert der sogeannte *Scope* in einer konkreten Programmiersprache. Es gibt drei Muster nach denen

die Gültigkeit von Namen in Programmteilen geregelt ist:

***Lexical* Scope oder *Static* Scope:** In diesem Schema gelten die Namen nur in dem Bereich, in dem sie definiert sind. Der Gültigkeitsbereich ist direkt aus dem Programmtext ersichtlich, woher der Name "lexical scope" rührt.

Wir definieren deshalb:

FUNDAMENTALBEGRIFF 5 (LEXICAL SCOPE)
Namen gelten nur in den Funktionen, die die Namensdefinition enthalten, bzw. in solchen, die innerhalb dieser Funktion als verschachtelte Funktionen definiert wurden.

Mit dem 'lexical scope' kann wie in der Programmiersprache Algol ein *'dynamic extent'* verbunden sein, oder wie in Common-Lisp und in Scheme der *'indefinite extent'*.

Letzerer bedeutet, dass alle Variablen unbegrenzte Lebensdauer haben. Die unbegrenzte Lebensdauer wird allerdings dadurch eingeschränkt, dass Variable, die von nirgendwoher mehr im Programm erreicht werden können, als Müll vom Garbage-Collector beseitigt werden.

Closures können auch gesehen werden als Funktionen mit 'lexical scope' und 'indefinite extent'.

Dynamic Scope: Eine Variable kann von überallaus im Programm direkt mit ihrem Namen angesprochen werden. Dynamic Scope ist von McCarthy ursprünglich in Lisp eingeführt worden, wird aber inzwischen wegen gewaltiger Nachteile in modernen Lisp-Dialekten nicht mehr oder nur bedingt verwendet. McCarthy selbst hat eingesehen, dass es ein Design-Fehler war, dynamic scope in Lisp zu verwenden.[2]

***Global* und *Local* Scope:** In diesem Schema hat eine Variable entweder 'global' oder 'local scope'. Im ersten Falle bedeutet das, dass eine Variable überall Gültigkeit besitzt. Im zweiten Fall gilt eine Variable nur in der Funktion, in der sie definiert wurde, ohne dass sich diese Gültigkeit auf tiefere Ebenen fortpflanzen würde. Dieses System wird momentan noch in der Programmiersprache Python verwendet.

Im nächsten Kapitel wird das sprachliche Gewand eingeführt, in das wir diese drei Operationen kleiden werden, um mit ihnen programmieren zu können.

[2] siehe Seite 180: R.Wexelblat(ed.), *History of Programming Languages*, Academic Press, New York, 1981.