Inhaltsverzeichnis

Diese Arbeit ist den Anwenderinnen und Anwendern von SNNS gewidmet, in der Hoffnung, daß sie deren natürliche neuronalen Netze etwas entlaste.

Ohne die Absicht, damit dieser Arbeit eine ihr ungebührende Wesentlichkeit beizumessen, möchte ich ihr an dieser Stelle eine Danksagung voranstellen.

Zunächst sei Prof. Paul Levi und Dipl.-Inf. Günter Mamier für die Betreuung und Prüfung dieser Arbeit gedankt. Insbesondere möchte ich mich auch für das gute Arbeitsklima in der Abteilung Bildverstehen des IPVR bedanken. Namentlich erwähnen möchte ich die Mitarbeiter Niels Mache und Harald Bayer für ihre Denkanstöße.

Die Arbeit wäre ohne die Unterstützung der im folgenden aufgeführten Personen und Institutionen sicherlich auch durchführbar gewesen, hätte jedoch beträchtlich weniger Spaß gemacht. Mein Dank gilt:

Ioannis Iglezakis *ευχαριστω για ολλα*;
The GNU Software Foundation for their software and especially Richard Stallman for writing *excellent* manuals about it;
Linus Torvalds und den Entwicklerinnen und Entwicklern des Betriebssystems Linux für Linux;
Leslie Lamport für The LaTeX document preparation system;
Fa. Bayer für Aspirin;
CD–Verleih Fun Cooperation;
Sheryl Crow für „Tuesday Night Music Club";
Stuttgarter Hofbräu;
The American Forces Radio and Television Service (AFRTS) für AFN;
Tonis Kneipe Rashomon;
Andreas Pyter für 4 MB und HP LaserJet 4L;
Elke Eiternick und Holger Pressel für das Sportprogramm;
Clemens Wieland für den Sorgenbeutel;
Angela für Canon BJ–200 und ♡ ♡ ♡ .

Ohne die abschließend erwähnten Personen wäre die Arbeit sicherlich nicht durchführbar gewesen:

Helga und Klaus Wieland.

Kapitel 1

Aufgabenstellung

Thema dieser Arbeit ist der Entwurf und die Implementierung einer interpretierten Sprache zur Batch–Simulation künstlicher neuronaler Netze.
Am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) existiert seit einigen Jahren ein sehr leistungsfähiger Simulator für künstliche neuronale Netze, der Stuttgarter Neuronale Netze Simulator (SNNS). Der Simulator wurde am IPVR entwickelt und besitzt als wesentliches Merkmal eine umfangreiche, komfortable graphische Benutzeroberfläche. Diese ist zum interaktiven Erstellen von und Experimentieren mit Netzarchitekturen bestens geeignet.

Bei der Durchführung langer Trainingsläufe hingegen oder bei wiederholter Ausführung ähnlicher Experimente ist die graphische Benutzeroberfläche jedoch nicht nur nicht mehr erforderlich, sondern eher hinderlich. Diese Aufgaben werden üblicherweise im Hintergrund als sogenannter Batchjob gestartet. Zu diesem Zweck verfügt SNNS über eine einfache Kommandosprache, die es ihm gestattet, die Befehle der graphischen Benutzeroberfläche aus einer Steuerdatei zu lesen und im Batchbetrieb abzuarbeiten. Die Kommandosprache bietet zwar alle zum Training erforderlichen Befehle, ist aber in ihrer Mächtigkeit vor allem in Bezug auf die Steuerung des Trainingsablaufs sehr eingeschränkt.

Ziel dieser Arbeit ist die Entwicklung einer Sprache, die es erlaubt, den Simulatorkern ohne Einbeziehung der graphischen Benutzeroberfläche als Batchjob zu starten und das Training der neuronalen Netze elegant mittels eines Ablaufplans zu steuern. Die Leistungsfähigkeit der graphischen Benutzeroberfläche soll dabei möglichst weitgehend erhalten bleiben beziehungsweise sogar ausgebaut werden.

Die Sprache ist in ANSI–C zu implementieren und soll durch semantische Aktionen die bereits bestehenden Routinen des Simulatorkerns aufrufen. Ferner soll die Dokumentation so erfolgen, daß Teile davon als Benutzeranleitung in das SNNS–Handbuch übernommen werden können. Die neue Sprache zur Batch–Simulation wird als Bestandteil von SNNS in die SNNS–Distribution des IPVR aufgenommen werden.

Kapitel 2

Einleitung

„… . So scheint sich die Vermutung zu bestätigen, daß wir selber *nichts* anderes als statistische geometrische Lernsysteme sind.“[a]

Raúl Rojas in: Theorie der neuronalen Netze

[a]Hervorhebung nicht im Original

Dieses Kapitel gibt einen kurzen Abriß über das Wie und das Warum der Simulation künstlicher neuronaler Netze. Es wird der Begriff Batchbetrieb eingeführt und auf Unterschiede zwischen interaktivem und Batchbetrieb eingegangen.
Im letzen Abschnitt des Kapitels wird begründet, warum die Wahl auf eine interpretierte Sprache zur Batch–Simulation fiel.

2.1 Simulation künstlicher neuronaler Netze

Was versteht man unter dem Begriff künstliches neuronales Netz? Ein künstliches neuronales Netz ist die technische Nachbildung eines natürlichen Nervennetzes, wie es in Lebewesen vorkommt. Das natürliche Nervennetz ist ein mitunter hochkomplexes Gebilde aus Neuronen als den informationsverarbeitenden Grundbausteinen und Axonen als den Verbindungen zwischen den Neuronen. In höheren Lebewesen geht die Anzahl der Neuronen in die Milliarden, die Anzahl der Axone ist nochmal ein Vielfaches davon. Beim künstlichen neuronalen Netz bildet man das biologische Vorbild in vereinfachter Weise auf ein Netz von Neuronen und sie verbindender Gewichte ab (siehe Abbildung 2.1). Die Realisierung kann sowohl schaltungstechnisch, also in Hardware, als auch in Form eines sequentiellen oder parallelen Programms erfolgen. Im zweiten Fall spricht man von der Simulation eines künstlichen neuronalen Netzes.

Künstliche neuronale Netze sind wie ihre biologischen Vorbilder in der Lage, die Lösung von Problemen durch wiederholte Präsentation von Beispielen und eventuell deren Lösungen zu lernen. Sie zeichnen sich außerdem durch Eigenschaften wie Generalisierungsfähigkeit, Fehlertoleranz, parallele Verarbeitung und verteilte Wissensrepräsentation aus und sind aufgrund ihrer Lernfähigkeit in der Lage, Probleme verschiedenster Art zwar nur nähe-

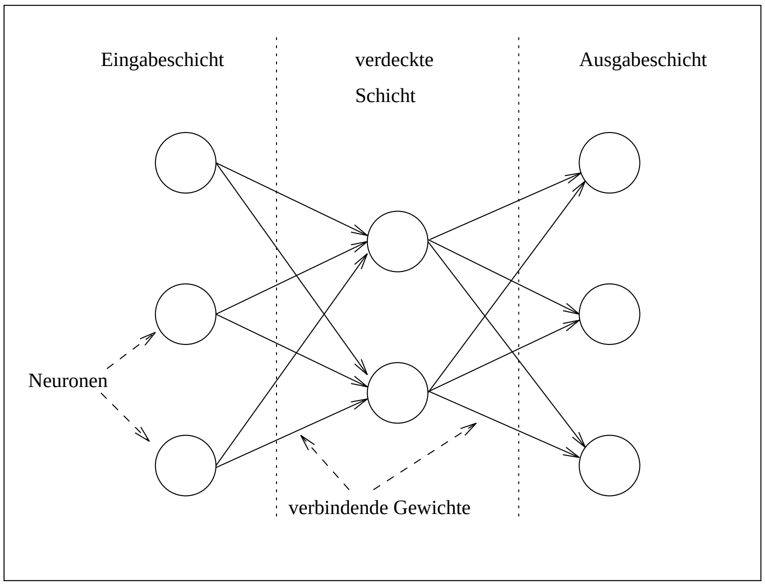

Abbildung 2.1: Einfaches künstliches neuronales Netz

rungsweise, oft aber hinreichend genau zu lösen.

Bei der Realisierung künstlicher neuronaler Netze hat man die Wahl zwischen einer Vielzahl von Netztopologien, Lernverfahren und anderen Parametern. Im Gegensatz zu biologischen neuronalen Netzen, bei denen meist viele Topologien als Mischformen nebeneinander vorkommen und im übrigen genetisch festgelegt sind, ist es beim Einsatz künstlicher neuronaler Netze dem Geschick und der Erfahrung des Experimentators überlassen, die richtige Auswahl aus Topologie und Parametern zu treffen. Vor dem endgültigen Einsatz der Netze sind oft viele Versuche zur Optimierung der Parameter durchzuführen. Hier liegt der große Vorteil der Softwaresimulatoren, die es erlauben, jederzeit Parameter oder Topologie der Netze zu ändern und die Auswirkung der Änderungen in kürzester Zeit zu analysieren.

2.2 Trainieren von neuronalen Netzen

Der Vorgang, bei dem ein neuronales Netz die Lösung eines Problems erlernen soll, wird im allgemeinen als Training bezeichnet. Dabei werden dem neuronalen Netz Beispiele, sogenannte Muster, aus der Menge der Symbole, die das zu lernende Problem darstellt, präsentiert. Bei einem Symbol handelt es sich zumeist um einen Vektor von Fließkommazahlen. Man unterscheidet zwischen überwachten und unüberwachten Lernverfahren. Bei ersteren wird dem Netz mit jedem Eingabemuster auch die Sollausgabe präsentiert, ebenfalls in Form eines Zahlenvektors. Bei den letzteren, den unüberwachten Lernverfahren, wird dem Netz keine Sollausgabe präsentiert. Es nimmt selber eine Klassifikation der Eingabemuster vor.

Nun gibt es viele Faktoren, die das Lernverhalten des Netzes beeinflussen. Hier seien nur Netztyp, Netztopologie, Lernverfahren und Auswahl der Muster genannt; es existieren viele weitere Parameter. Die Möglichkeiten von SNNS werden im nächsten Kapitel vorge-

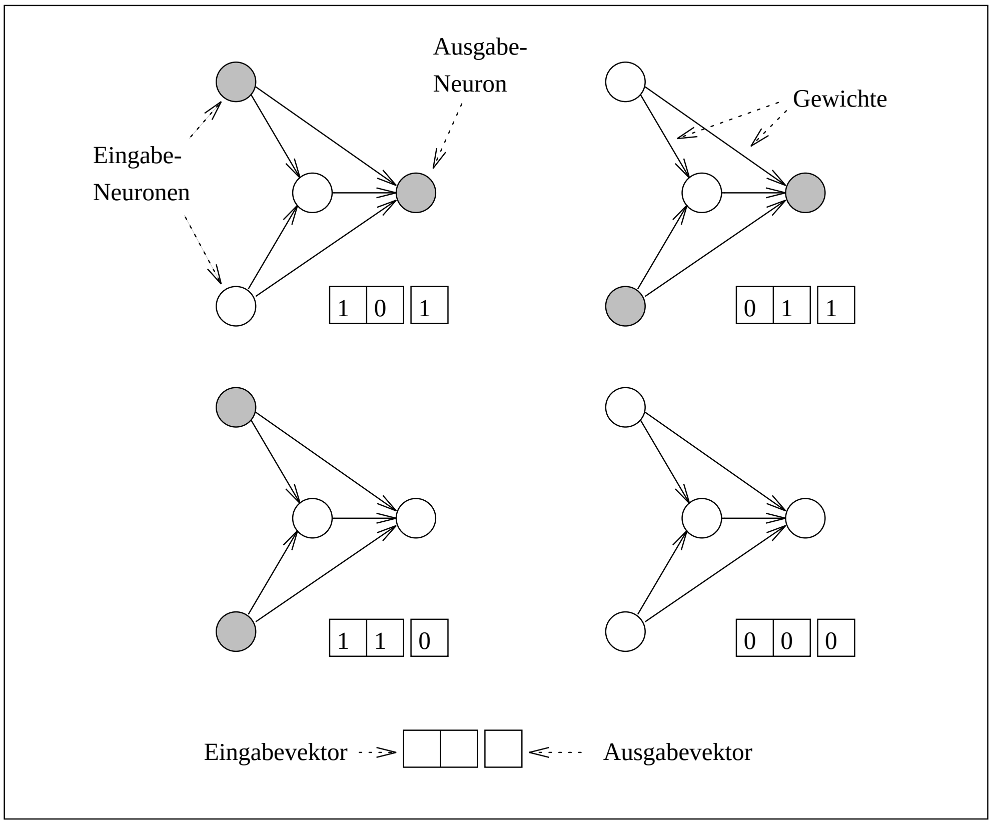

Abbildung 2.2: Neuronales Netz für das XOR–Problem

stellt, eine umfangreiche Darstellung der Simulation neuronaler Netze findet sich in [Zell 94]. Zur Einführung in die Theorie neuronaler Netze empfehlen sich [Rumelhart et al. 86] und [Rojas 93].

Als Maß für das erreichte Lernniveau dient oft der Netzwerkfehler, der sich als die Abweichung zwischen Sollausgabe und tatsächlicher Ausgabe der Ausgabeneuronen des Netzes ergibt. Üblicherweise reicht es nicht aus, dem Netz beim Training alle Muster je einmal zu präsentieren. Zumeist sind einige wenige bis zu vielen tausend Trainingszyklen erforderlich, um ein befriedigendes Ergebnis zu erreichen. Ein Zyklus oder eine Epoche bedeutet hier, daß dem Netz jedes Muster der Trainingsmenge einmal präsentiert worden ist. Sind die Faktoren geeignet gewählt und ist die Architektur des Netzes für das Problem geeignet, so sinkt im Verlauf des Trainings der Netzwerkfehler. Es werden soviele Zyklen trainiert, bis der Fehler eine vorgegebene Grenze unterschritten hat.

Abbildung 2.2 zeigt ein einfaches neuronales Netz, das die Lösung des XOR–Problems lernen kann. Dabei soll das Ausgabeneuron genau dann aktiviert werden, wenn entweder das eine oder das andere Eingabeneuron aktiviert ist. Da das XOR–Problem lediglich vier Trainingsmuster besitzt, kann in der Abbildung die gesamte Trainingsmustermenge sowie die Sollausgabe dargestellt werden. Wird das Netz nun je einmal in einen der vier abgebildeten Zustände versetzt, so ist ein Lernzyklus vollendet.

2.3 Batchbetrieb

Zur Simulation neuronaler Netze wird der Simulator sowohl im interaktiven wie auch im Batchbetrieb verwendet. Beide Betriebsarten haben ihre Berechtigung. Batchbetrieb bedeutet, daß alle zur Steuerung des Trainings notwendigen Anweisungen für den Simulator in einer Steuerdatei abgelegt werden. Der Simulator führt die Anweisungen aus, ohne sie auf der graphischen Benutzeroberfläche darzustellen. Der ganze Trainingslauf kann dann unbeaufsichtigt durchgeführt werden. Die Ergebnisse werden in einer Datei für die spätere Auswertung protokolliert.

Der interaktive Betrieb des Simulators ist für kurze Versuche geeignet und durch die Möglichkeit der Visualisierung von Netztopologie, Lernverlauf und Ergebnissen eine unverzichtbare Hilfe beim Entwurf von Netzen, der Auswahl von geeigneten Lernverfahren und vielem mehr.

Oft ist es jedoch nötig, längere Trainingsläufe durchzuführen. Auch die Optimierung von Netzen, Musterdateien oder verschiedenster Parameter ist sehr rechenzeitaufwendig. Eine interaktive Steuerung ist hier nicht nötig und auch oft nicht durchführbar, weil zu Zeiten geringer Rechnerlast (nachts und am Wochenende) simuliert werden muß. Hier hat der Batchbetrieb seine Stärken. Versuche können vollständig vorher formuliert werden und zu geeigneter Zeit auf einem unbelasteten Rechner im Netz durchgeführt werden.

2.4 Interpreter oder Compiler für die Batch–Simulation neuronaler Netze?

Der Unterschied zwischen einem Interpreter und einem Compiler kann folgendermaßen formuliert werden: Ein Compiler übersetzt ein als Quelltext vorliegendes Programm in ein von der Maschine ausführbares Format und speichert dieses in einer Datei. Diese ausführbare Datei ist hernach als eigenständiges Programm lauffähig. Ein Interpreter hingegen führt bereits während seiner eigenen Laufzeit die Anweisungen des Quelltexts aus. Es wird keine ausführbare Datei erzeugt. Anstatt den Quelltext als Datei einzulesen, ist es im Prinzip auch möglich, einzelne Befehle auf die Standardeingabe des Terminals einzutippen und sofort ausführen zu lassen.

Im Zusammenhang mit der Simulation neuronaler Netze wird von beiden Varianten Gebrauch gemacht. Für SNNS existiert ein Netzwerkcompiler, der ein trainiertes neuronales Netz in ein ausführbares Programm übersetzt (das Programm SNNS2C). Dieses sozusagen „eingefrorene“ Netz kann nicht mehr weiter trainiert werden, sondern wird nur noch in der Abrufphase betrieben, das heißt, zur Auswertung der Eingabemuster. Zweck dieses so übersetzten Netzes ist die praktische Anwendung, bei der ein fertig trainiertes und optimiertes Netz seine Aufgabe, zum Beispiel die Steuerung von Geräten oder die Auswertung von Daten, erfüllt. Das ausführbare Programm ist von unnötigem Ballast des Simulators befreit.

Für die Batch–Simulation ist der Einsatz eines Interpreters sinnvoller. Die Aufgabe ist hier nicht, das gleiche Netz unverändert einzusetzen, sondern den Einfluß aller möglichen Faktoren zu verändern und zu untersuchen. Der Interpreter enthält alle Funktionen des Simulatorkerns und führt die Befehle des Quelltexts während seiner Laufzeit aus. Es wird kein ausführbares Programm erzeugt. Änderungen im Quelltext können sofort getestet werden.

Kapitel 3

SNNS

3.1 Das Projekt SNNS

SNNS, der Stuttgarter Neuronale Netze Simulator [Zell et al. 95], existiert seit 1989 am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart. Seither wird SNNS ständig verbessert und weiterentwickelt. Ziel des Projektes war und ist es, eine effiziente und flexible Simulationsumgebung für Forschung und Anwendung neuronaler Netze zu schaffen.

Der Simulator besteht aus drei Hauptkomponenten: dem Simulatorkern und der graphischen Benutzeroberfläche XGUI (X Graphical User Interface) sowie dem Netzwerkcompiler. Der Simulatorkern beinhaltet die internen Datenstrukturen zur Repräsentation der neuronalen Netze und führt alle Operationen, welche die Simulation betreffen, aus. Die graphische Benutzeroberfläche stellt das neuronale Netz graphisch dar und ermöglicht die Steuerung des Simulatorkerns während der Simulation. Zusätzlich können mittels der Benutzeroberfläche neuronale Netze in unterschiedlichen Architekturen erstellt und manipuliert werden. Der Netzwerkcompiler generiert Netze für den Simulatorkern aus einer Beschreibung in der Sprache NESSUS. Bei NESSUS handelt es sich um eine prozedurale Hochsprache zur Beschreibung der Topologie neuronaler Netze. Abbildung 3.1 zeigt den groben Aufbau von SNNS.

SNNS ist vollständig in ANSI–C implementiert. Der Simulatorkern läuft auf Workstations verschiedener Hersteller unter verschiedenen UNIX– Betriebssystemen. XGUI basiert auf der Version X11 Release 5 des MIT X Window System und dem Athena Toolkit und wurde auf dem twm window manager erprobt (nach [Zell 92] und [Zell et al. 95]).

Der in dieser Arbeit beschriebene Batchinterpreter setzt auf das sogenannte „SNNS Kernel Function Interface“ auf, eine Schnittstelle, die anderen Programmen über Funktionsaufrufe den Zugang zu den im Simulatorkern realisierten Funktionen ermöglicht. Auch die graphische Benutzeroberfläche XGUI kommuniziert über diese Schnittstelle mit dem Simulatorkern. Der Batchinterpreter arbeitet so als eigenständiges Programm und beinhaltet die graphische Benutzeroberfläche nicht. Details der Implementierung sind in Kapitel 6 ab Seite 22 beschrieben.

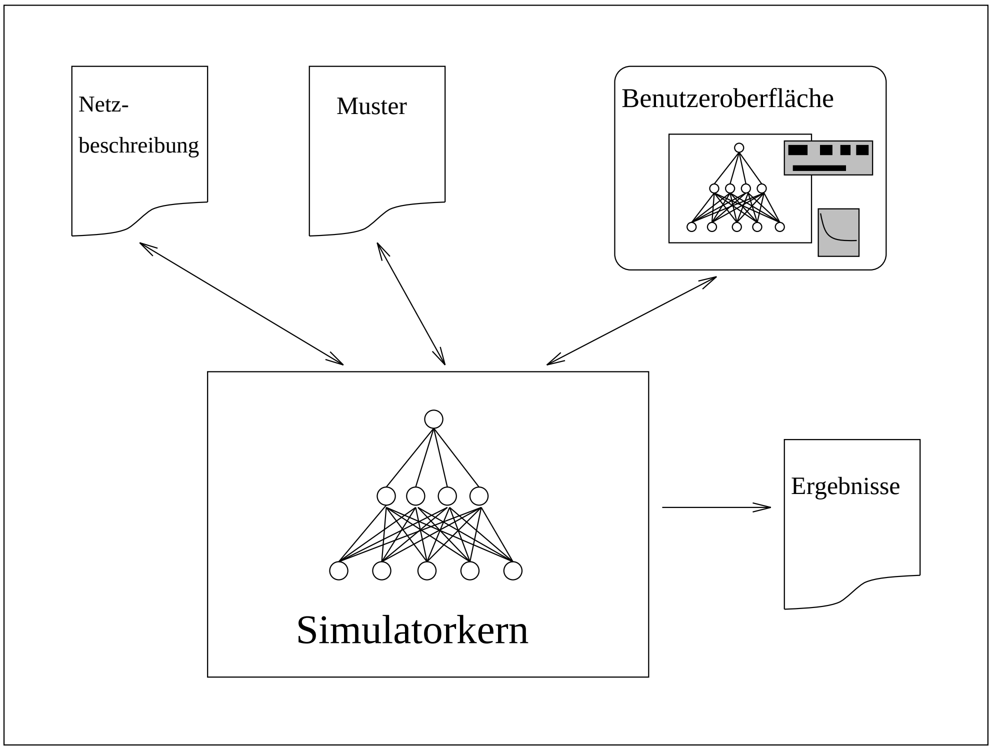

Abbildung 3.1: Aufbau von SNNS

3.2 Funktionalität und Anwendungen

SNNS bietet eine Vielzahl von Werkzeugen zum Erstellen, Trainieren, Testen und Analysieren neuronaler Netze. So können neuronale Netze sowohl interaktiv erstellt und geändert als auch durch Spezifikation ihrer Topologie automatisch generiert werden. Die Netze können in 2D– und 3D–Graphiken sowohl auf dem Bildschirm als auch als Ausdruck dargestellt werden. Für den Anwender steht eine große Auswahl an Lernfunktionen und unterstützten Architekturen zur Verfügung:

- mehrere Lernfunktionen für mehrschichtige Feedforward–Netze
- Cascade Correlation
- Counterpropagation
- Radial Basis Function Netze
- Adaptive Resonance Theory (ART) Netze
- Hopfield Netze
- Time–Delay Netze
- Selbstorganisierende Karten (Kohonennetze)
- Autoassoziative Netze
- Partiell rekurrente Netze
- Dynamic Learning Vector Quantization

- stochastische Lernfunktionen

Ferner sind Verfahren zur Optimierung der Netzarchitektur vorhanden.

Unter den Anwendungen finden sich: Rotationsinvariante Mustererkennung, Erkennung von handgeschriebenen Zeichen, Börsenkursvorhersage, Erkennung und Klassifikation von Gehirnpotentialen, Rauschverminderung in Telekommunikationssystemen für natürliche Sprache, Vorhersage der Sekundär– und Tertiärstruktur von Proteinen, Texturanalyse, Worterkennung durch Lippenlesen, Stereobilderkennung, Robotersteuerung und vieles mehr (nach [Zell et al. 93]).

Kapitel 4

Sprachen für die Batch–Simulation neuronaler Netze

Neben SNNS existieren weitere Software–Simulatoren für neuronale Netze. Manche davon besitzen eine graphische Benutzeroberfläche und eine Sprache zur Batch–Simulation, andere nur eins von beidem. Es lag also nahe, beim Entwurf der Batchsprache für SNNS das Design einiger vorhandener Batchsprachen auf eine Eignung als Vorbild zu untersuchen. Im folgenden werden einige der Simulatoren, die über die Möglichkeit einer Batchsteuerung verfügen, kurz vorgestellt.

4.1 Beschreibung ausgewählter Simulatoren und ihrer Batchsprachen

NEUROGRAPH

NEUROGRAPH [Wilke 94] ist eine Simulationsumgebung für neuronale Netze mit folgenden Komponenten: Visualisierung, Batchsteuerung, Projektmanager, Funktionsbibliothek, Parallelisierung und weiteren. Neben einer C–ähnlichen Programmiersprache zur Beschreibung von Netzen und Lernverfahren verfügt NEUROGRAPH über eine separate Batchsteuerung. Diese Pascal– oder Modula–2–ähnliche Sprache enthält skalare Datentypen, Kontrollstrukturen für Schleifen und bedingte Ausführung von Anweisungen, Befehle für Parallelisierung und Dateioperationen, sowie Unterroutinen und Prozeduren. Ferner besteht die Möglichkeit, mit sogenannten Handlern Laufzeitfehler abzufangen.

Die Sprache verfolgt ein strenges Typkonzept mit Deklarationspflicht. Ferner wurde, um die Steuerung der Prozesse einheitlich zu gestalten, der objektorientierte Ansatz des Methodenaufrufs gewählt. Zum Erlernen der Sprache werden beim Anwender demzufolge Kenntnisse in objektorientierter und paralleler Programmierung vorausgesetzt.

PlaNet

Das PlaNet–System [Miyata 91] ist ein Werkzeug zum Erstellen, Untersuchen und Betreiben von neuronalen Netzen. PlaNet bietet zum einen eine sehr mächtige Spezifikationssprache zum Beschreiben von neuronalen Netzen, zum anderen eine Kommandozeilenschnittstelle, über die sowohl eingebaute Funktionen als auch die vom Benutzer geschriebenen Funktionen der Netzbeschreibung ausgeführt werden können. Unter den mitgelieferten Bibliotheksfunktionen befinden sich auch solche zur graphischen Ausgabe von Netzen und Daten.

Interessanterweise wird nun ein Batchbetrieb solchermaßen ermöglicht, daß der Kontrollfluß für ganze Versuchsabläufe als gewöhnliches UNIX–Shellscript[1] geschrieben werden kann. So mußte keine eigene Batchsprache entworfen und implementiert werden. Auch haben viele UNIX–erfahrene Benutzerinnen und Benutzer bereits Übung im Schreiben von Shellscripts. Das Script führt die eingebauten oder benutzerdefinierten Funktionen aus, die sonst über die Kommandozeilenschnittstelle zugänglich sind. Da man in Shellscripts jedoch nicht ohne weiteres Rechenoperationen durchführen kann, ist die Mächtigkeit dieser Art der Steuerung begrenzt und die Syntax unkomfortabel.

Xerion

Bei Xerion [van Camp 93] handelt es sich nicht um einen einsatzbereiten Simulator für neuronale Netze, sondern vielmehr um eine Sammlung von C–Bibliotheken, die dazu benutzt werden können, viele verschiedene Typen von neuronalen Netzen zu implementieren. Mit den in Xerion enthaltenen C–Routinen können komplexe Netze auf einfache Weise kodiert werden. Ein Simulator, der mit Hilfe der Xerion–Bibliotheken geschrieben wurde, besitzt eine Kommandozeilenschnittstelle mit Befehlen zum Erstellen und Trainieren von Netzen, Darstellen und Ändern der Datenstrukturen sowie Hilfsprogramme. Außerdem enthält er eine graphische Schnittstelle zur Visualisierung der Netze.

Da die Anwenderin oder der Anwender den Simulator unter Einbeziehung der Xerion–Bibliotheken in C selbst schreibt, hat er oder sie viele Möglichkeiten zur Integration einer Batchsteuerung, muß diese allerdings auch selber programmieren. Denkbar sind zum Beispiel festkodierte Abläufe oder Einbinden eines vorhandenen Interpreters bis hin zum Schreiben einer eigenen Batchsprache. Auch eine Scriptsprache, die auf die Kommandozeilenschnittstelle aufsetzt, ist möglich. Selbstredend wird beim Anwender die Beherrschung der Sprache C vorausgesetzt.

CuPit

CuPit [Prechelt 94] (der Name geht auf Warren McCulloch und Walter Pitts zurück, die 1943 erstmalig eine formale Beschreibung eines Neurons veröffentlichten) ist eine parallele Programmiersprache mit zwei Hauptzielen: erstens soll eine einfache, problembezogene Beschreibung von Lernalgorithmen ermöglicht werden und zweitens die Erzeugung von effizientem Code für massiv parallele Rechner aus einem maschinenunabhängigen Programm. Entstanden ist eine vollständige parallele Programmiersprache, als deren Vorbilder C und Ada genannt werden können. In der Mächtigkeit ihrer Kontroll- und Datenstrukturen dürfte sie ihren Vorbildern kaum nachstehen. Dadurch sind, ähnlich wie bei Xerion, der Batchsteuerung praktisch keine Einschränkungen auferlegt. Allerdings ist zum Erlernen von CuPit

[1] Das PlaNet–Handbuch enthält als Beispiel ein für die C–Shell (csh) geschriebenes Script.

fortgeschrittene Kenntnis mindestens einer höheren Programmiersprache sowie Erfahrung mit paralleler Informationsverarbeitung unbedingt erforderlich.

Da es sich bei CuPit um einen Compiler handelt, müssen, wenn das Batchprogramm geändert wird, Teile des Programms neu übersetzt und neu gebunden werden. Dieses Problem tritt bei Xerion auch auf, sofern die Batchsteuerung innerhalb des C–Codes für den Simulator implementiert ist.

4.2 SNNSBAT — bisherige Batchsteuerung für SNNS

SNNSBAT [Zell et al. 95] ist ein Programm zur Batchsteuerung von SNNS. Es kann als eine zusätzliche Schnittstelle zum Simulatorkern aufgefaßt werden. In einer sogenannten Konfigurationsdatei können Kommandos und Parameter für die Steuerung des Simulatorkerns zusammengestellt werden. Im Prinzip bleibt die Leistungsfähigkeit der graphischen Benutzeroberfläche in Bezug auf das Training und den Test neuronaler Netze erhalten; das Verändern von Netztopologien und Mustern sowie alle Formen graphischer Ausgabe sind allerdings nicht möglich. Auch kann SNNSBAT nicht als eine Programmiersprache bezeichnet werden, sondern eher als eine Art Kommandosprache: es gibt keine Variablen, keine Arithmetik und keine Kontrollstrukturen. Allein der Abbruch des Trainings bei Überschreiten einer vorgegebenen Zyklenzahl oder bei Unterschreiten eines vorgegebenen Fehlers ist als Eingriff in den Ablaufplan möglich. Aktionen des Simulatorkerns werden implizit durch Angabe von Parametern ausgelöst (zum Beispiel startet die Angabe der maximalen Anzahl von Lernzyklen das Training).
Die Ergebnisse von Training oder Test der Netze werden in den von SNNS erzeugbaren Ergebnisdateien festgehalten. Zusätzlich werden Informationen über den Ablauf des Batchjobs in einer Protokolldatei aufgezeichnet.

Obwohl SNNSBAT alle SNNS–Funktionen zum Trainieren und Testen neuronaler Netze unterstützt, sind die Einsatzmöglichkeiten bei einem größeren Versuchsprojekt stark eingeschränkt. Das Fehlen von Kontrollstrukturen macht sich bei der Formulierung von Versuchsabläufen als sehr hinderlich bemerkbar. So ging der Autor dieser Arbeit bei einem früheren Projekt bald dazu über, die Konfigurationsdateien für SNNSBAT automatisch von Shellscripts erstellen zu lassen. Dadurch konnte auch ein weiterer eklatanter Mangel von SNNSBAT behoben werden, nämlich die Tatsache, daß Dateinamen nur als Konstante in der Konfigurationsdatei eingegeben werden können. Die Möglichkeit des Erzeugens von unterschiedlichen und aussagekräftigen Namen für die Dateien, in denen trainierte Netze und Ergebnisse im Verlauf des Batchjobs gespeichert werden, ist für jeden größeren Versuch unverzichtbar.

So entstand die Forderung nach einer neuen Batchsprache, die in ihrer Leistungsfähigkeit den Merkmalen von SNNS — hohe Effizienz und Flexibilität — gerecht wird.

Kapitel 5

Entwurf einer Batchsprache für SNNS

„LANGUAGE DESIGN IS DECISION MAKING
… . Let me list a few of the most frequently encountered demands.

- The language must be easy to learn and easy to use.
- It must be safe from misinterpretation and misuse.
- It must be extensible without change of existing features.
- There must be a rigorous, mathematical definition, based on axioms and withstanding the scrutiny of logicians.
- The notation must be convenient and compatible with widely used (and sometimes not so logical) standards.
- …“

Niklaus Wirth in: On The Design of Programming Languages (aus [Horowitz 83])

Nachdem im vorhergehenden Kapitel existierende Batchsprachen vorgestellt worden sind, werden im ersten Abschnitt dieses Kapitels zunächst die Forderungen an die neue Batchsprache aufgestellt und die Frage eines Neuentwurfs diskutiert. Im zweiten Abschnitt wird die entworfene Sprache detailliert beschrieben.

5.1 Entwurfskriterien, Sprachumfang

Im folgenden sind sowohl die Entwurfskriterien und Forderungen an die neue SNNS–Batchsprache als auch ihre Grenzen zusammengestellt:

1. Die Leistungsfähigkeit der graphischen Benutzeroberfläche von SNNS soll erhalten bleiben, wenn nicht sogar ausgebaut werden. Damit ist gemeint, daß alle Funktionen zum Trainieren und Testen der neuronalen Netze von der Batchsprache unterstützt werden sollen, nicht aber die graphischen Funktionen der Benutzeroberfläche. Die Möglichkeit der Ausführung von SNNSBAT–Konfigurationsdateien ist ebenfalls nicht vorgesehen.

2. Die Sprache soll trotz ihrer Leistungsfähigkeit einfach zu erlernen sein. Dies bedeutet, den Umfang auf ein sinnvolles Maß zu begrenzen und kein völlig neues oder schwerfälliges Sprachkonzept zu verfolgen.
3. Es sollen die von höheren Programmiersprachen her gewohnten Kontrollstrukturen wie Schleifen und bedingte Anweisungen vorhanden sein. Damit soll ermöglicht werden, den Abbruch des Trainings von Bedingungen abhängig zu machen, die der Programmierer oder die Programmiererin völlig frei formulieren kann.
4. Aus Punkt 3 ergibt sich fast zwangsläufig, daß die Sprache in der Lage sein muß, den Wert arithmetischer und logischer Ausdrücke berechnen zu können. Wünschenswert wären die üblichen arithmetischen Operatoren auf den Datentypen ganze Zahl und Fließkommazahl, wie sie beispielsweise in Pascal vorhanden sind, sowie logische Operatoren für Vergleichsoperationen auf den obengenannten Datentypen. Sinnvoll wäre auch der Datentyp für binäre Zustände (boolescher Datentyp).
5. Die Sprache soll durch leichte Erweiterbarkeit mit der Weiterentwicklung von SNNS schritthalten können.
6. Es muß die Möglichkeit bestehen, sinnvolle Namen für Ergebnisdateien zu erzeugen. Daraus ergibt sich die Notwendigkeit eines weiteren Datentyps Zeichenkette sowie entsprechender Operationen darauf.
7. Da der vom Simulatorkern an die Batchsteuerung zurückgegebene Netzwerkfehler allein nicht immer für die Beurteilung des Lernerfolgs ausreicht, sollen von der Batchsprache aus Analyseprogramme auf der UNIX–Kommandoebene gestartet werden können. Deren Ergebnisse sollen vom Batchprogramm zur Weiterverarbeitung eingelesen werden können. Natürlich können so auch alle anderen (nichtinteraktiven) Programme auf der UNIX–Kommandoebene ausgeführt werden.
8. Weitere Datentypen, insbesondere nicht–skalare und benutzerdefinierte Datentypen, sind nicht vorgesehen. Ebenso wird auf benutzerdefinierbare Prozeduren verzichtet, denn durch die Möglichkeit des Aufrufs von UNIX–Programmen kann auch ein weiterer Batchinterpreter gestartet werden, der ein Teilexperiment ausführt. Auf diese Weise können sich oft wiederholende Aufgaben in gesonderte Batchprogramme ausgelagert werden.

Manche der in Kapitel 4, Abschnitt 4.1 vorgestellten Batchsprachen verfügen über Mechanismen, die es erlauben, das Auftreten von Laufzeitfehlern im Programm (zum Beispiel Division durch null) abzufangen und so ein vorzeitiges Ende eines Versuchs zu verhindern. Diese Option wurde für die hier beschriebene Batchsprache auch in Betracht gezogen und in einer Testversion implementiert. Bei den Tests zeigte sich jedoch, daß der Nutzen sehr gering ist: Weiß man im voraus, wo ein Fehler auftreten kann, so wird man das Batchprogramm entsprechend ändern. Weiß man es nicht, so ist unbekannt, welche Anweisungen von dem Fehler betroffen sind und nicht zur Ausführung kommen. Ein reibungsloser Ablauf des verbleibenden Batchprogramms ist meist nicht mehr möglich.

Aufgrund dieser Erfahrungen wurde auf das Abfangen von Laufzeitfehlern bewußt verzichtet. Sinnvoller ist es, größere Versuche in einzelne Teilprogramme aufzuteilen.

Neuentwurf oder nicht?

Beim Entwurf der Sprache wurde den Forderungen des vorhergehenden Abschnitts Rechnung getragen. Für das Design der Sprache bestanden folgende Optionen:

1. Übernehmen von Syntax und Semantik einer verbreiteten Sprache, Anpassung an den Einsatzbereich als SNNS–Batchsprache.
2. Einbettung der SNNS–Batchsprache in eine vorhandene interpretierte Batchsprache, zum Beispiel Tcl [Ousterhout 93]. Tcl (Tool command language) ist eine einfache Scriptsprache, deren Interpreter in Form einer C–Bibliothek implementiert ist und einfach in Anwendungen eingebunden und erweitert werden kann. Die Einbettung hätte zur Folge, daß die etwas eigenwillige Syntax und die Semantik von Tcl (oder einer anderen Wirtssprache) übernommen werden müßten.
3. Neuentwurf.

Die Entscheidung fiel auf den Neuentwurf, und zwar aus folgenden Gründen:

1. Zur Übernahme einer vorhandenen Sprache wäre es sinnvoll, eine weitverbreitete Sprache wie zum Beispiel C, Ada oder Pascal zu wählen. Da diese Sprachen sehr mächtig sind, müßte die Batchsprache in vielerlei Hinsicht auf eine echte Untermenge des Vorbilds reduziert werden. Dabei bestünde das Problem, daß der Benutzer auf Elemente des Sprachvorbilds zugreift, die in der Batchsprache nicht implementiert sind, zum Beispiel bestimmte Datenstrukturen oder Operatoren. Damit wäre der vermeintliche Vorteil einer leichten Erlernbarkeit dahin — es bestünde die Notwendigkeit, alle Abweichungen zu kennen. Auch sind in den genannten Sprachen einige Konstruktionen aufwendiger als es für den Zweck einer Batchsprache nötig ist, man denke an die zwingende Deklaration von Variablen.
2. Die Einbettung der SNNS–Batchsprache in eine vorhandene interpretierte Batchsprache hätte zwar den Vorteil einer leichteren Implementierung, die Benutzerin oder der Benutzer müßte aber in den meisten Fällen die Wirtssprache erlernen. Die Wirtssprache müßte auf dem Rechnersystem des Anwenders vorhanden sein.
3. Beim Neuentwurf kann die Sprache so einfach wie möglich gehalten werden. Zudem können die besten Konstrukte aus anderen Sprachen in der neuen Batchsprache vereinigt werden. Gleichzeitig kann diese von unnötigem Ballast, wie er sich bei vielen mächtigeren Sprachen findet, freigehalten werden.

Als Vorbild für die neue Batchsprache ist an erster Stelle die von Alfred V. Aho, Peter J. Weinberger und Brian W. Kernighan entwickelte Sprache AWK [Aho et al. 88b] beziehungsweise GAWK[1] [Close et al. 93] zu nennen. Ferner sind Anleihen bei Pascal [Jensen et al. 74] und Modula–2 [Wirth 85] gemacht worden. Ausdrücke können in C– beziehungsweise AWK–

[1] Die Verwandtschaft bezieht sich auf die Semantik von AWK–*Aktionen*, nicht auf die AWK–Architektur als ein zeilenorientierter Patternmatcher.

und Pascal–Syntax formuliert werden, wobei allerdings nicht alle C–Operatoren unterstützt werden (genaueres siehe Abschnitt „Ausdrücke" auf Seite 19).

5.2 Sprachbeschreibung

Dieser Abschnitt enthält die Beschreibung der syntaktischen Elemente und Konstrukte der neuen Batchsprache, die Formate von Funktionsaufrufen, Ausdrücken und Anweisungen bis zum kompletten Batchprogramm.

Notation

In Anlehnung an das Standardwerk für Compilerbau [Aho et al. 88a] (siehe auch das Vorwort zu Kapitel 6 dieser Arbeit auf Seite 22), Kapitel 2 und 4, werden hier zur Spezifikation der Sprachsyntax kontextfreie Grammatiken verwendet, da solche Grammatiken auf recht anschauliche Weise die hierarchische Struktur von Programmiersprachen beschreiben können. Eine kontextfreie Grammatik enthält vier Komponenten:

1. Eine Menge von Symbolen, auch Terminale genannt.
   Beispiele sind Schlüsselworte, Strichpunkt, Plus- und Minuszeichen usw.
2. Eine Menge von Nichtterminalen.
   Nichtterminale repräsentieren Symbolfolgen und könnten auch als Platzhalter für Symbolfolgen bezeichnet werden.
3. Eine Menge von Produktionen.
   Jede Produktion besteht aus einem Nichtterminal auf der linken Seite gefolgt von einem Pfeil ($\rightarrow$) und einer Folge von Terminalen und oder Nichtterminalen auf der rechten Seite. Beispiel: $A \rightarrow B + C$.
4. Ein ausgezeichnetes Nichtterminal als Startsymbol.
   Das Startsymbol steht auf der linken Seite *der* Produktion, mit der die Ableitungen von der Grammatik beginnen. Ableiten bedeutet hier, die Nichtterminale auf den linken Seiten der Produktionen eins nach dem anderen zu den rechten Seiten der Produktionen aufzulösen, sodaß schließlich nur noch Terminale übrigbleiben: dann ist ein „Satz" aus der durch die Grammatik erzeugten Sprache gebildet beziehungsweise abgeleitet worden.

Die Grammatik kann nun durch Angabe ihrer Produktionen beschrieben werden. Das Nichtterminal auf der linken Seite der *ersten* angegebenen Produktion ist hier immer das Startsymbol der Grammatik oder der Teilgrammatik.

Eine Produktion für ein bestimmtes Nichtterminal kann auf der rechten Seite mehrere Alternativen besitzen. Diese können entweder nacheinander mit demselben Nichtterminal auf der linken Seite oder alle auf der rechten Seite der Produktion durch einen senkrechten Strich (|) getrennt erscheinen.
Ziffern, Zeichen und Operatoren wie + (natürlich außer |) sowie fettgedruckte Zeichenketten wie **while** sind Terminale. Kursivgedruckte Namen wie *statement* sind Nichtterminale.
Das Zeichen Epsilon (ϵ) steht für eine leere Produktion, das heißt, das Nichtterminal auf

der linken Seite wird durch nichts ersetzt.
Die im folgenden verwendeten Namen für Nichtterminale weichen geringfügig von den in der Implementierung verwendeten Namen ab, um eine bessere Lesbarkeit der Grammatik zu erreichen.

Kommentare

An beliebiger Stelle im Batchprogramm können Kommentare stehen, um dem Quellcode Erklärungen beizufügen. Diese Kommentare werden vom Interpreter überlesen. Die Syntax von Kommentaren wird hier nur informell eingeführt, da Kommentare nicht Teil der Grammatik sind, die ein Batchprogramm beschreibt. Die Syntax für Kommentare lautet:
Ein Kommentar muß mit einem #–Zeichen beginnen und kann dann aus keinem oder beliebig vielen beliebigen Zeichen bis zum Zeilenende bestehen.

Namen und Konstante

Die Namen, mit denen Variable bezeichnet werden können, unterliegen folgender Konvention: sie müssen mit einem Buchstaben oder einem Unterstrich (_) beginnen und dürfen dann beliebig viele Buchstaben, Ziffern und Unterstriche enthalten. Klein- und Großschreibung wird unterschieden; die Länge ist beliebig. Die Grammatik für einen Namen lautet also:

$$\begin{aligned}
&\textit{name} \rightarrow \textit{firstletter rest}\\
&\textit{firstletter} \rightarrow \mathbf{a} \mid \ldots \mid \mathbf{z} \mid \mathbf{A} \mid \ldots \mid \mathbf{Z} \mid _\\
&\textit{rest} \rightarrow \epsilon \mid \textit{rest otherletter}\\
&\textit{otherletter} \rightarrow \textit{firstletter} \mid \mathbf{0} \mid \ldots \mid \mathbf{9}
\end{aligned}$$

Als Konstante sind ganze Zahlen, Fließkommazahlen, Zeichenketten und boolesche Werte vorgesehen. Die Grammatik für ganze Zahlen lautet:

$$\begin{aligned}
&\textit{number} \rightarrow \textit{digit} \mid \textit{number digit}\\
&\textit{digit} \rightarrow \mathbf{0} \mid \ldots \mid \mathbf{9}
\end{aligned}$$

Die Syntax von Fließkommazahlen ist wie in C definiert: eine Fließkommazahl kann mit einer ganzen Zahl beginnen. Danach kann ein Punkt stehen, aber nun muß unbedingt eine ganze Zahl stehen, die mindestens einstellig ist. Nach dieser Zahl kann ein Exponent folgen, der mit e oder E eingeleitet wird. Der Exponent kann nach dem e beziehungsweise dem E ein Vorzeichen haben, muß aber auf jeden Fall mit einer mindestens einstelligen ganzen Zahl enden.
Beispiele für gültige Fließkommazahlen sind: 0.42, 3.3, 1.0, .9, 3e3, 0.7E-12, 2123E123.
Die Grammatik für Fließkommazahlen lautet (unter Einbeziehung der Grammatik für ganze Zahlen):

$$\begin{aligned}
&\textit{realnumber} \rightarrow \textit{integerPart number}\\
&\qquad\quad \mid \textit{integerPart number exponent}\\
&\qquad\quad \mid \textit{number exponent}\\
&\textit{integerPart} \rightarrow \textit{number}\ \mathbf{.} \mid \mathbf{.}\\
&\textit{exponent} \rightarrow \mathbf{e}\ \textit{sign number} \mid \mathbf{E}\ \textit{sign number}\\
&\textit{sign} \rightarrow \epsilon \mid + \mid -
\end{aligned}$$

Die Syntax für Zeichenketten legt fest, daß eine Zeichenkette aus mindestens einem Zeichen bestehen muß. Außer Anführungsstrichen oben ("), Tabulatorzeichen und Zeilenendezeichen darf sie jedes beliebige druckbare Zeichen enthalten. Die Zeichenkette ist in Anführungsstriche oben eingeschlossen, die aber nicht zur Zeichenkette gehören. Die Grammatik ist nachfolgend angegeben, wobei die Produktion für „beliebige druckbare Zeichen“ vereinfacht angegeben ist:

> *string* → "*stringcharacter*"
> *stringcharacter* → *character* | *stringcharacter character*
> *character* → *beliebigesDruckbaresZeichen*

Nach dieser Grammatik ist die Länge einer Zeichenkette beliebig. Aus technischen Gründen ist die Länge einer Zeichenkette jedoch auf eine konstante Zahl von Zeichen begrenzt. Diese Konstante kann im Quellcode des Batchinterpreters verändert werden.

Abschließend sei noch die Grammatik für boolesche Werte angegeben. Boolesche Werte können nur einen von zwei Zuständen annehmen:

> *boolean* → **TRUE** | **FALSE**

Datentypen

Als Datentypen stehen wie gefordert die skalaren (nicht zusammengesetzten) Typen ganze Zahl, Fließkommazahl, Zeichenkette und boolescher Typ zur Verfügung. Der Wertebereich für die beiden numerischen Typen ist systemabhängig. Für ganze Zahlen wird bei der Implementierung der C–Datentyp *int*, für Fließkommazahlen der C–Datentyp *double* verwendet. Beide Typen können auch anders gewählt werden.

Variable

Variable können ohne vorherige Deklaration verwendet werden. Dabei ist der Wert und der Typ einer Variablen solange undefiniert, bis ihr ein Wert zugewiesen wird. Bei jeder Zuweisung nimmt sie den Typ an, der sich auf der rechten Seite der Zuweisung ergibt. Der Typ wird bis zur nächsten Zuweisung beibehalten. Namen für Variable müssen der oben angegebenen Grammatik für Namen genügen.

Systemvariable

Als eine Sonderform von Variablen stellt die Batchsprache Systemvariable zur Verfügung. Hierbei handelt es sich um Variable, deren Wert ausschließlich vom Interpreter verändert werden kann. Sie besitzen feste Namen, die im Abschnitt 6.9 auf Seite 51 aufgeführt sind. Das Benutzerprogramm kann nur lesend auf sie zugreifen.

Ausdrücke

Mit Ausdrücken sind arithmetische und logische Rechen- oder Vergleichsoperationen gemeint. Als einzige Operation auf Zeichenketten ist die Verkettung mittels des +–Operators implementiert. Die Grammatik für Ausdrücke lautet:

$$
\begin{aligned}
&\textit{expression} \rightarrow \textit{simpleExpression}\\
&\qquad \mid \textit{unaryOperator expression}\\
&\qquad \mid \textit{expression binaryOperator expression}\\
&\textit{simpleExpression} \rightarrow \textbf{(}\ \textit{expression}\ \textbf{)}\\
&\qquad \mid \textit{identifier}\\
&\textit{unaryOperator} \rightarrow + \mid - \mid \textbf{not} \mid \textbf{!}\\
&\qquad \mid \textbf{ln} \mid \textbf{log} \mid \textbf{sqrt}\\
&\textit{binaryOperator} \rightarrow + \mid - \mid * \mid / \mid \textbf{div} \mid \% \mid \textbf{mod} \mid < \mid > \mid == \mid \textbf{!=}\\
&\qquad \mid <> \mid <= \mid =< \mid >= \mid => \mid \textbf{and} \mid \&\& \mid \textbf{or} \mid \|^{2} \mid ** \mid \hat{}\\
&\textit{identifier} \rightarrow \textit{name} \mid \textit{number} \mid \textit{realnumber} \mid \textit{string} \mid \textit{boolean}
\end{aligned}
$$

Ausdrücke können nach dieser Grammatik ineinander verschachtelt werden, wobei die Reihenfolge der Auswertung direkten Einfluß auf das Ergebnis hat. Die Reihenfolge der Auswertung ist durch obige Grammatik noch nicht festgelegt. Sie wird durch Angabe der Operator–Bindungsstärken in der Quelldatei für den Parsergenerator definiert (siehe Abschnitt 6.5 auf Seite 35).

Man erkennt ferner, daß für manche Operatoren zwei verschiedene Schreibweisen zugelassen sind, die sich an C und Pascal beziehungsweise AWK anlehnen. Dadurch sollen unnötige Syntaxfehler vermieden werden. Weiter beachte man, daß die Vorzeichen für Zahlen (+ und −) als Teil der Grammatik für Ausdrücke implementiert sind und nicht als Teil der Grammatik für ganze Zahlen oder Fließkommazahlen.

Die Teilgrammatik für Bezeichner (*identifier*) erlaubt für alle Stellen, an denen Bezeichner stehen können, sowohl Variable als auch Konstante beliebigen Typs. Ob im Einzelfall eine Variable oder eine Konstante stehen muß beziehungsweise von welchem Typ sie sein muß, wird vom Zwischencodeinterpreter überprüft.

Typ eines Ausdrucks

Um die Programmierung in der neuen SNNS–Batchsprache möglichst einfach zu halten und den Anwender nicht mit Anweisungen zur Typkonversion zu belasten, wurde auf ein strenges Typkonzept verzichtet. Die meisten Operatoren arbeiten auf verschiedenen Argumenttypen. Der Typ eines Ausdrucks (also der Typ des Ergebnisses einer Operation) wird automatisch festgelegt und ist, außer bei der Ganzzahl–Division, der Modulo–Operation, den booleschen Operatoren und den Vergleichsoperatoren, nicht durch den Operator vorbestimmt.

[2] Hier ist das Terminal „doppelter senkrechter Strich" gemeint.

So ergibt zum Beispiel die Multiplikation zweier ganzer Zahlen den Typ Ganzzahl. Die Multiplikation einer ganzen Zahl und einer Fließkommazahl hingegen ergibt als Ergebnis den Typ Fließkommazahl. Die ganze Zahl wird hierzu vor der Multiplikation automatisch in eine Fließkommazahl umgewandelt.
Ist mindestens ein Operand einer Addition eine Zeichenkette, so wird der andere Operand in eine Zeichenkette umgewandelt, und es wird eine Verkettung der beiden Operanden zu einer resultierenden Zeichenkette durchgeführt.
Der Typ der bei einer Operation *nur* als Operand beteiligten Variablen bleibt natürlich unverändert. Bei dem folgenden Programmfragment allerdings kommt die Variable `a` sowohl als Operand als auch auf der linken Seite einer Zuweisung vor:

```
a := 5
a := a + 2.3
```

Ist der Typ von `a` nach der ersten Zuweisung ganze Zahl, so ergibt sich durch die Addition mit einer Fließkommazahl in der zweiten Zuweisung für `a` der Typ Fließkommazahl.

Funktionsaufrufe

Funktionsaufrufe sind — formal gesehen — Namen, denen in einem ihnen folgenden runden Klammerpaar Parameter zugeordnet werden können. Funktionsaufrufe rufen in der Batchsprache immer eingebaute Funktionen auf, da benutzerdefinierte Funktionen nicht vorgesehen sind. Das Format von Funktionsaufrufen ist durch folgende Grammtik gegeben:

$$\begin{aligned}
&\mathit{functioncall} \rightarrow \mathit{identifier}\ (\ \mathit{parameterlist}\ )\\
&\mathit{parameterlist} \rightarrow \epsilon \mid \mathit{fullList}\\
&\mathit{fullList} \rightarrow \mathit{expression} \mid \mathit{fullList}\ ,\ \mathit{expression}
\end{aligned}$$

Diese Grammatik ermöglicht Funktionsaufrufe ohne Parameter oder mit einer beliebigen Anzahl von durch Kommata getrennten Parametern. Die Klammern nach dem Funktionsnamen müssen in jedem Fall vorhanden sein. Die Parameter können — gemäß der Grammatik für Ausdrücke — Konstante, Variable oder Ausdrücke beliebigen Typs sein.

Zuweisungen

Auch Zuweisungen können nach Pascal– oder C–Syntax geschrieben werden. Die Variable links des Zuweisungsoperators darf keine Systemvariable sein.

$$\begin{aligned}
&\mathit{assignment} \rightarrow \mathit{identifier}\ \mathit{assignmentOperator}\ \mathit{expression}\\
&\mathit{assignmentOperator} \rightarrow\ = \mid :=
\end{aligned}$$

Anweisungen

Zum Schluß soll nun die Grammatik für Anweisungen vorgestellt werden. Die Syntax ist der bekannter prozeduraler Programmiersprachen sehr ähnlich. Beachtenswert ist, daß die Anweisungsblöcke bei der for– und while–Schleife sowie bei den if–Anweisungen immer durch ein Schlüsselwort abgeschlossen werden müssen. Damit soll vermieden werden, daß Fehler im Kontrollfluß durch versehentliches Weglassen von blockumgebenden Schlüsselworten (zum Beispiel „begin" und „end") entstehen. Diese Sicherheit rechtfertigt den geringen Mehraufwand, das Ende eines Blocks hier immer durch ein Schlüsselwort kennzeichnen zu müssen.

Die nun folgende Grammatik ist eigentlich die Grammatik für Blöcke von Anweisungen. Dieser Vorgriff sei erlaubt, da die Grammatik für Blöcke von der Grammatik für Anweisungen untrennbar ist, weil beide wechselseitig aufeinander Bezug nehmen:

$$
\begin{aligned}
&\textit{block} \rightarrow \textit{statement} \\
&\qquad \mid \textit{block statementDelimiter statement} \\
&\textit{statementDelimiter} \rightarrow \textbf{;} \mid \textbf{newlineCharacter}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{statement} \rightarrow \epsilon \\
&\qquad \mid \textit{functioncall} \\
&\qquad \mid \textit{assignment} \\
&\qquad \mid \textbf{for } \textit{assignment} \textbf{ to } \textit{expression} \textbf{ do } \textit{block} \textbf{ endfor} \\
&\qquad \mid \textbf{while } \textit{expression} \textbf{ do } \textit{block} \textbf{ endwhile} \\
&\qquad \mid \textbf{repeat } \textit{block} \textbf{ until } \textit{expression} \\
&\qquad \mid \textbf{if } \textit{expression} \textbf{ then } \textit{block} \textbf{ endif} \\
&\qquad \mid \textbf{if } \textit{expression} \textbf{ then } \textit{block} \textbf{ else } \textit{block} \textbf{ endif} \\
&\qquad \mid \textbf{break} \\
&\qquad \mid \textbf{continue}
\end{aligned}
$$

Mit „newlineCharacter" ist das Zeilenendezeichen gemeint. Aus der Produktion *statementDelimiter* ergibt sich, daß Anweisungen entweder durch ein Semikolon oder durch ein Zeilenende getrennt werden können.

Programm

Jetzt kann die Grammatik für ein komplettes Batchprogramm leicht formuliert werden:

$$\textit{batchprogram} \rightarrow \textit{block}$$

Ein komplettes Batchprogramm besteht also aus einer Folge von Anweisungen. Anweisungen müssen nur dann durch Strichpunkte getrennt sein, wenn sie hintereinander in der gleichen Zeile stehen. Strichpunkte am Ende des Programms oder am Ende von Blöcken sind optional. Es gibt keinen Deklarationsteil und keine Schlüsselworte, mit denen Anfang oder Ende des Programms gekennzeichnet werden müssen. Die gesamte Grammatik ist in Anhang A zusammenhängend abgedruckt.

Viele Beispiele für Batchprogramme finden sich in Abschnitt 7.2 ab Seite 57.

Kapitel 6

Implementierung

In diesem Kapitel werden die programmiertechnischen Details der Implementierung der Batchsprache beschrieben. Nach einer Vorstellung der Programmierumgebung und der verwendeten Werkzeuge wird auf Besonderheiten des Programms eingegangen, wobei dem Parser als Kernstück und Bindeglied zwischen Quellprogramm und Codeerzeugung besonderes Augenmerk geschenkt wird. Ferner wird die Realisierung von Scanner, Symboltabelle, Zwischencodeinterpreter, den SNNS-Funktionsaufrufen und Sonderfunktionen beschrieben, und es wird auf die Fehlerbehandlung eingegangen.

Wie im vorigen Kapitel wird auch in diesem häufig auf [Aho et al. 88a] Bezug genommen. Bei diesem zweibändigen Werk beziehungsweise dem englischen Original [Aho et al. 87] handelt es sich um das absolute Standardwerk für die Theorie und Praxis des Compilerbaus (das sogenannte „Drachenbuch“). Viele der dort beschriebenen Techniken und Methoden lassen sich auch auf den Entwurf eines Interpreters übertragen. So kann und soll es nicht ausbleiben, daß dieses Werk im folgenden häufig zitiert wird. Dennoch ist es bei der Implementierung einer neuen Programmiersprache wie der hier beschriebenen Batchsprache keineswegs mit dem Übernehmen des (Pseudo–) Codes aus „Compilerbau“ getan. So ist dieses Werk trotz seines Umfangs an manchen Stellen nicht allzu detailliert. Daher wird in den folgenden Abschnitten darauf hingewiesen, welche Probleme bei der Implementierung auftraten beziehungsweise an welchen Stellen abweichend von den Vorschlägen des Drachenbuchs ein anderer Weg gegangen wurde.

6.1 Programmierumgebung

Als Implementierungssprache war ANSI–C [Schildt 90] vorgegeben. Zur Übersetzung wurde der GNU–C–Compiler gcc Version 2.6.2 verwendet. Alle Module des Programms wurden zur Sicherstellung der ANSI–C–Kompatibilität mit den gcc–Compileroptionen -ansi, -pedantic, -Wall, -Wtraditional, -Wconversion, -Wstrict-prototypes und -Wmissing-prototypes übersetzt. Bei den Modulen, die vom Autor geschrieben worden sind (im Gegensatz zu den von Werkzeugen erstellten Modulen), traten dabei keine Warnungen auf.

Zusätzlich zum ANSI–C–Standard sind die „Richtlinien für die Konvertierung von SNNS in den ANSI Standard“ der SNNS–Gruppe bei der Implementierung umgesetzt worden. Die

Meldungen des Programms, die Namen von Bezeichnern und die Kommentare im Quelltext sind in englischer Sprache abgefaßt, da das Programm als Teil der SNNS–Distribution einem internationalen Anwenderkreis zur Verfügung gestellt wird.

Für die Erstellung des Scanners und des Parsers wurden als Werkzeuge der Scannergenerator flex Version 2.4.6 und der Parsergenerator Bison Version 1.22 verwendet. Diese Werkzeuge werden in den folgenden Abschnitten kurz vorgestellt. Genaueres findet sich außer in der dort angegebenen Literatur auch in [Aho et al. 88a].
Die gesamte Programmentwicklung wurde auf einem 486er–PC unter dem Betriebssystem Linux Version 1.0.9 und der Oberfläche X Window System Version X11 R6 durchgeführt. Entstanden sind rund dreitausend Zeilen Quellcode (für Scanner und Parser sind jeweils die Zeilen der Beschreibungsdateien gezählt worden, nicht der erzeugte Code für Parser und Scanner).

Scannergenerator flex

Der Scannergenerator flex [Paxson 90] (fast lexical analyzer generator) ist ein Werkzeug zur Erzeugung des C–Codes für einen Scanner. Scanner sind Programme, die lexikalische Muster in Texten erkennen. flex liest als Eingabe eine Datei mit der Beschreibung des zu erzeugenden Scanners. Die Beschreibung hat die Form von Paaren: jedes Paar besteht aus einem regulären Ausdruck und einer Regel. Die Regel besteht aus C–Code, der jedesmal bei Erkennen des zugehörigen regulären Ausdrucks ausgeführt wird.

flex erzeugt aus dieser Beschreibung von regulären Ausdrücken und Regeln C–Code für einen Scanner, der dann vom C–Compiler übersetzt und mit der flex–Bibliothek und den anderen Modulen des Programms zusammengebunden wird.

Parsergenerator Bison

Bison [Donnelly et al. 92] ist der Name der GNU–Version von yacc [Aho et al. 74] (yet another compiler compiler), einem Standard–UNIX–Werkzeug zur automatischen Erzeugung von Parsern. Parsen oder parsing bedeutet Syntaxanalyse. Die Aufgabe und die Funktion eines Parsers sind in Abschnitt 6.5 dieses Kapitels (Seite 32) genauer beschrieben. Bison erzeugt, ähnlich wie yacc, aus einer Beschreibung eines Parsers den C–Code für einen Parser. Die Beschreibung ist etwas aufwendiger als bei flex. Vereinfacht gesagt, wird die Grammatik der zu parsenden Sprache in Bison–Syntax eingegeben. In jede Produktion der Grammatik können beliebig viele semantische Aktionen eingebaut werden, die, wenn der Parser an der entsprechenden Stelle der Produktion steht, ausgeführt werden. Die semantischen Aktionen sind C–Code, der auf Attribute der Terminale und Nichtterminale der Produktion zugreifen kann.

Bison erzeugt einen effizienten LALR–Parser. LALR(k) steht für „lookahead left–to–right scan rightmost derivation with k symbols of lookahead" und bedeutet, daß die zu parsende Eingabe von links nach rechts gelesen wird, eine Rechtsableitung der Grammatik gebildet wird und k Eingabesymbole im voraus betrachtet werden. Da die Schreibweise LALR(1)–Parser äquivalent zu der Schreibweise LALR–Parser ist, handelt es sich bei dem von Bison

erzeugten Parser also um einen Parser, der ein Eingabesymbol im voraus betrachtet. Die Mächtigkeit der Sprachklasse, die mit LALR–Parsern geparst werden kann, reicht für die Grammatiken der meisten höheren Programmiersprachen aus (nach [Aho et al. 88a] und [Hopcroft et al. 79]).
Der von Bison erzeugte C–Code wird dann vom C–Compiler übersetzt und mit den anderen Modulen des Programms zusammengebunden.

Die Verwendung von Parser- und Scannergeneratoren ist von der Konstruktion von Compilern und Interpretern nicht mehr wegzudenken. Die genannten Werkzeuge entbinden den Programmierer oder die Programmiererin von der mühseligen Arbeit des Aufstellens von Tabellen und endlichen Automaten und erzeugen effizienten Code bei vertretbarem Platzverbrauch. Die Grammatik kann so auf einer hohen Abstraktionsebene formuliert werden.

6.2 Funktionale Struktur des Interpreters

Um der Leserin oder dem Leser einen Überblick über das Zusammenwirken der in den folgenden Abschnitten erläuterten Bestandteile des Interpreters zu geben, wird hier zunächst einmal der grobe Aufbau beschrieben (nach [Aho et al. 88a]).

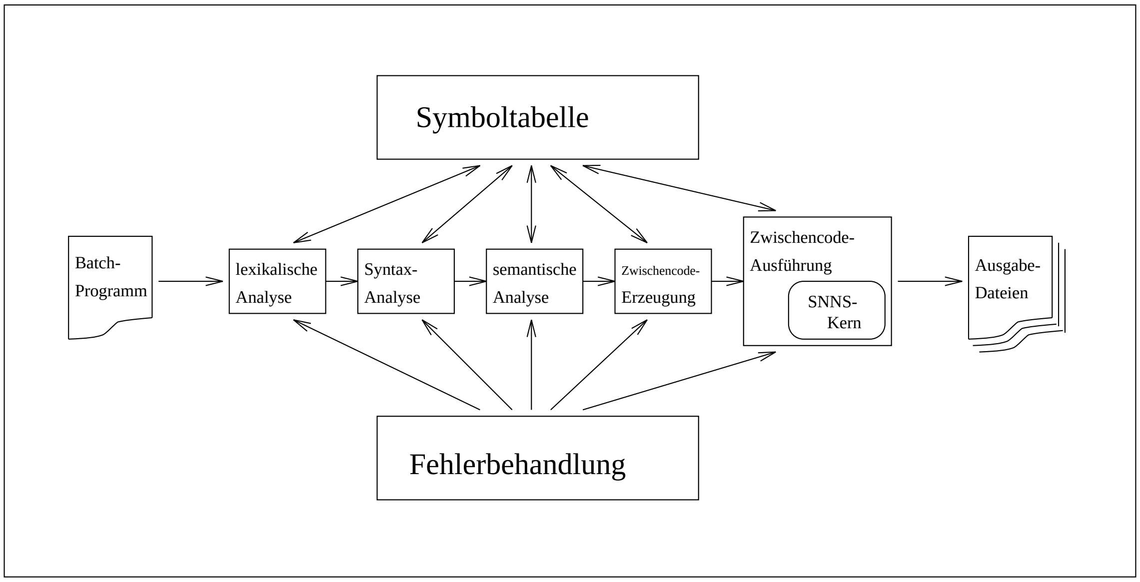

Abbildung 6.1: Aufbau des Interpreters

Abbildung 6.1 zeigt nahezu die klassische Struktur eines Compilers, es fehlen lediglich die Phasen der Codeoptimierung und -erzeugung. Dafür gibt es einen Zwischencodeinterpreter. Trotz der Ähnlichkeit enthält die Implementierung dieses Interpreters einige Abweichungen vom klassischen Modell, die an den entsprechenden Stellen in den nachfolgenden Abschnitten erläutert werden.

Symboltabelle

Zunächst ist die Schlüsselrolle der Symboltabelle zu erkennen. Die Aufgabe der Symboltabelle ist es, die im Batchprogramm[1] vorkommenden Bezeichner zu speichern und die dazugehörigen Attribute, zum Beispiel den Wert und den Typ einer Variablen, zu sammeln. Auch die Namen der Systemvariablen und der eingebauten Funktionen des Interpreters sind in der Symboltabelle eingetragen. Von allen Phasen des Interpreters wird auf die Symboltabelle zugegriffen.

Der Eintrag der im Batchprogramm vorkommenden Bezeichner geschieht während der lexikalischen Analyse. Auch konstante Werte wie Zahlen, Zeichenketten und boolesche Werte werden in die Symboltabelle eingetragen. Dadurch wird die Bison–Beschreibung der Grammatik übersichtlicher und die Implementierung der Zwischencodefunktionen wesentlich eleganter, da deren Parameter immer Symboltabelleneinträge sind.
Der Name einer Konstante ist die Konstante selber (zum Beispiel hat die Konstante 5 den Namen „5“).

Analysephasen

Zurück zu Abbildung 6.1. Die Verarbeitung des Batchprogramms beginnt mit der lexikalischen Analyse. Die zunächst als bedeutungslos erscheinende Folge von Buchstaben, Zeichen und Ziffern wird hier in eine Folge von Symbolen umgewandelt. Jedes Symbol ist eine sinnvolle Einheit eines oder mehrerer Zeichen, wie etwa eine ganze Zahl oder ein Name. Der für die Erkennung von Symbolen zuständige Teil des Programms wird Scanner genannt.

Es folgt die Phase der Syntaxanalyse. Hier wird die Folge von Symbolen, welche die lexikalische Analyse liefert, in eine hierarchische Struktur gebracht. Diese Struktur muß aus der Grammatik der Batchsprache ableitbar sein. Diese und andere Aufgaben, wie Zwischencodeerzeugung, werden vom Parser erledigt.
Angeschlossen ist die Phase der semantischen Analyse. Hier werden zum Beispiel die Verträglichkeit von Operanden in Ausdrücken oder der Typ von Schleifenbedingungen überprüft. Die Aufgaben und die Funktion des Parsers werden in Abschnitt 6.5 auf Seite 32 genau beschrieben.

Zwischencodeerzeugung und -ausführung

Die Phase der Zwischencodeerzeugung ist eng mit den Phasen syntaktische und semantische Analyse verknüpft. Die Art des hier verwendeten Zwischencodes könnte als „Mehradreßcode“ bezeichnet werden (in Anlehnung an den 1–, 2– oder 3–Adreßcode, der in Compilern oft Verwendung findet), da die Zwischencode–Anweisungen mit einer unterschiedlichen Zahl von Parametern arbeiten. Dies ist möglich, da ja nicht, wie bei einem Compiler, Maschinencode erzeugt werden muß, dessen Format an eine Prozessorarchitektur gebunden ist.

Wenn bis zu dieser Phase kein Fehler aufgetreten ist, wird der Zwischencode vom Zwischencodeinterpreter ausgeführt. Er führt Berechnungen für arithmetische und logische Ausdrücke

[1] Mit „Batchprogramm“ ist hier und im folgenden das vom Anwender geschriebene Programm gemeint.

durch, führt Zuweisungen und Sprünge aus und ruft die Funktionen des SNNS–Kerns auf. Außerdem werden Informationen über den Zustand des SNNS–Kerns verwaltet wie zum Beispiel, ob ein Netz initialisiert worden ist oder ob das Batchprogramm Parameter geändert hat.

6.3 Verwendete Datenstrukturen

In diesem Abschnitt werden die wichtigen und komplexen Datenstrukturen für die Symboltabelle, die Zwischencodetabelle und die Parameterliste beschrieben.

Symboltabelle

Der Aufbau der Symboltabelle ist in Tabelle 6.1 wiedergegeben.

Index	Name	Status	Typ	Wert
0	FALSE	ro	BOOL	FALSE
1	print	ro	FUNCTION	$\uparrow$print()
2	i	rw	INTEGER	5
3	encoder.net	ro	STRING	encoder.net
4	net1	rw	STRING	encoder.net
5	34.456	ro	REAL	34.456
6	test	rw	UNKNOWN	
⋮	...	...	...	...

Tabelle 6.1: Symboltabelle

Die Symboltabelle enthält bereits einige Beispieleinträge, anhand derer ihre Verwendung verdeutlicht werden soll. Zunächst wird die Bedeutung der einzelnen Spalten erklärt.

Die Spalte „Index“ gehört eigentlich nicht zur Datenstruktur der Symboltabelle. Technisch gesehen ist die Symboltabelle als ein Feld von C–Strukturen implementiert. Der Zugriff auf einzelne Elemente dieses Felds (ein Element entspricht einem einzelnen Symboltabelleneintrag) geschieht über Zeiger. Da C den Zugriff auf Elemente eines Felds mittels einer indizierten Zeigervariablen zuläßt, kann hier dieser Index als Möglichkeit des Zugriffs auf einen Symboltabelleneintrag gesehen werden.

Die Spalte „Name“ enthält den Namen einer Variablen, einer Funktion oder einer Konstanten. Über diesen Namen wird sie identifiziert, wenn im Verlauf des Batchprogramms wieder auf sie zugegriffen wird.

Ob eine Anweisung des Batchprogramms einen Symboltabelleneintrag verändern darf, wird durch den Eintrag in der Spalte „Status“ festgelegt. Es sind zwei Einträge möglich: ro für „read–only“, was bedeutet, daß das Batchprogramm den Symboltabelleneintrag nicht ändern darf, und rw für „read–write“, für den Fall, daß das Batchprogramm den Symboltabelleneintrag ändern darf. Alle Variable des Batchprogramms haben den Status rw; Systemvariable, Konstante und Funktionen haben den Status ro.

Der Typ eines Symboltabelleneintrags wird durch den Eintrag in der Spalte „Typ“ festgelegt. Mögliche Typen sind INTEGER, REAL, STRING, BOOL, FUNCTION und UNKNOWN entsprechend den Datentypen ganze Zahl, Fließkommazahl, Zeichenkette, boolescher Wert, dem Typ Funktion und dem unbekannten Typ für Variable, denen noch kein Wert und Typ zugewiesen worden ist.

Schließlich wird der Wert eines Symboltabelleneintrags in der Spalte „Wert“ eingetragen. Dieser Wert ist eine Konstante von dem Typ, der in der Spalte „Typ“ desselben Symboltabelleneintrags steht, also zum Beispiel eine Zeichenkette oder eine Zahl. Ist der Typ UNKNOWN, so ist der zugehörige Wert undefiniert. Ist der Typ FUNCTION, so ist der Wert ein Zeiger auf eine Zwischencodefunktion, die an anderer Stelle im Programm ausgeführt ist.

In der Symboltabelle, die oben als Beispiel abgebildet ist, haben die Einträge demnach folgende Bedeutung:

- Eintrag 0 ist eigentlich eine boolesche Konstante. Die beiden booleschen Werte TRUE und FALSE sind aber als „Systemkonstante“ schon fest in der Symboltabelle eingetragen;
- Eintrag 1 ist der Name einer eingebauten Funktion;
- Eintrag 2 ist eine ganzzahlige Variable des Batchprogramms namens „i“ mit dem Wert 5;
- Eintrag 3 ist eine Zeichenkettenkonstante mit dem Wert „encoder.net“;
- Eintrag 4 dagegen ist eine Variable namens „net1“ vom Typ Zeichenkette mit dem gleichen Wert „encoder.net“;
- Eintrag 5 ist eine Fließkommakonstante und
- Eintrag 6 eine Variable namens „test“, deren Wert und Typ noch nicht bekannt sind.

Zugriffsoperationen auf der Symboltabelle

Auf der Symboltabelle sind folgende Zugriffsoperationen definiert:

- Initialisieren. Speicher für die Symboltabelle wird vom Betriebssystem angefordert, die Namen von Systemvariablen und eingebauten Funktionen werden eingetragen.
- Eintragen. Es wird nachgeschaut, ob ein Name schon in der Symboltabelle vorhanden ist. Ist dies nicht der Fall, wird der Name eingetragen. In jedem Fall wird ein Zeiger auf seine Position in der Symboltabelle zurückgegeben.

- Nachschauen. Die Symboltabelle wird nach einem Namen durchsucht, ein Zeiger auf seine Position in der Symboltabelle zurückgegeben.
- Wert und Typ eines Symboltabelleneintrags lesen.
- Wert und Typ eines Symboltabelleneintrags ändern.
- Status lesen. Dient dazu festzustellen, ob ein Symboltabelleneintrag geändert werden darf.
- Status auf ro setzen. Diese Operation wird vom Scanner bei der Eintragung von Konstanten verwendet.

Zwischencodetabelle

Die Zwischencodetabelle enthält den erwähnten „Mehradreßcode", im folgenden Zwischencode genannt. Er ist die interne Darstellung des Batchprogramms im Interpreter. Der Zwischencode enthält drei Arten von Befehlen:

- Sprungbefehle mit zwei Parametern: Sprungziel und Sprungbedingung.
- Arithmetisch–logische Operationen und die Zuweisung im 3–Adreß–Format (Ergebnis, Operand 1 und Operand 2).
- Mantelfunktionen mit einem Parameter, nämlich einem Zeiger auf eine Liste, die Zeiger auf die Symboltabelleneinträge der effektiven Parameter der Mantelfunktion enthält (siehe auch Abbildung 6.3 auf Seite 30). Die Mantelfunktionen dienen einerseits dazu, den zusätzlichen Code bereitzustellen, der notwendig ist, um eine oder mehrere Funktionen des SNNS–Kerns aufzurufen. Andererseits kann eine Mantelfunktion Aufgaben ausführen, die einen oder mehrere Funktionsparameter benötigen wie zum Beispiel die `print`–Funktion, welche eine beliebige Anzahl von Parametern druckt.

Die Zwischencodetabelle ist als Feld implementiert. Die erste, mit „Idx" bezeichnete Spalte, enthält den Index eines Eintrags in die Zwischencodetabelle. Der Index kann als die „Zeilennummer" eines Zwischencodebefehls angesehen werden. Tatsächlich verwendet der Zwischencodeinterpreter diesen Index als Programmzähler.

Die Zwischencodetabelle enthält noch zwei weitere Spalten:

Zum einen die Spalte „Funktionstyp". Sie enthält einen der drei möglichen Typen von Zwischencodefunktionen: Sprungbefehl, arithmetisch–logische Operation beziehungsweise Mantelfunktion. Zum anderen die Spalte „Zwischencodefunktion". Sie ist ihrerseits bereits eine komplexe Datenstruktur: Sie enthält für jeden der drei Funktionstypen einen Eintrag in einem Verbund, wobei jeder Eintrag wiederum aus einer Struktur besteht. Der Aufbau dieser drei Strukturen ist unterschiedlich. Die Struktur für den Funktionstyp „Sprungbefehl" enthält drei Elemente:

1. Einen Zeiger auf die Zwischencode–Sprungfunktion, die ausgeführt werden soll.
2. Den Index des Zwischencodebefehls, zu dem gesprungen werden soll.

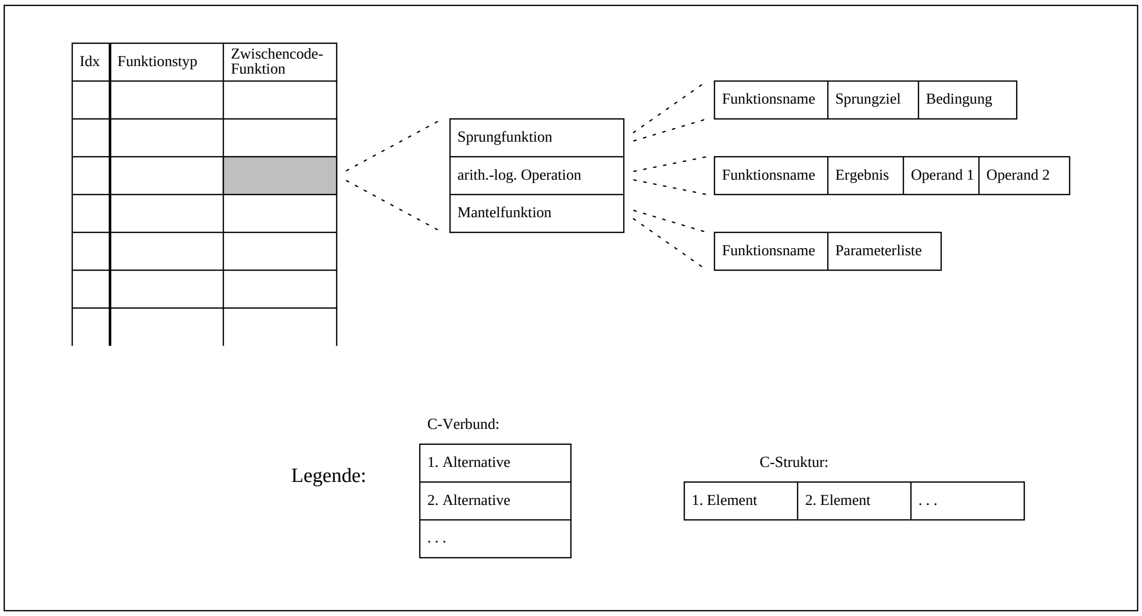

Abbildung 6.2: Aufbau der Zwischencodetabelle

3. Einen Zeiger auf einen Eintrag in der Symboltabelle. Es hängt vom Wert der dort befindlichen Konstante oder Variable ab, ob bei bedingten Sprüngen der Sprung ausgeführt wird oder nicht.

Die Struktur für den Funktionstyp „arithmetisch–logische Operation“ enthält vier Elemente:

1. Einen Zeiger auf die arithmetisch–logische Zwischencodefunktion, die ausgeführt werden soll.
2. Einen Symboltabellenzeiger auf die Variable, in der das Ergebnis der Operation gespeichert werden soll.
3. Einen Symboltabellenzeiger auf die Variable oder Konstante, die als einziger Operand (bei einstelligen Operationen) oder als erster Operand einer zweistelligen Operation verwendet werden soll.
4. Einen Symboltabellenzeiger auf die Variable oder Konstante, die als zweiter Operand einer zweistelligen Operation verwendet werden soll.

Der Mechanismus der Ausführung von Mantelfunktionen weicht von dem der beiden vorigen Funktionstypen ab. Die Struktur für den Funktionstyp „Mantelfunktion“ hat die folgenden zwei Elemente:

1. Einen Symboltabellenzeiger, der auf den Eintrag zeigt, dessen Name gleich dem Namen der Mantelfunktion im Batchprogramm ist. Der Wert dieses Symboltabelleneintrags ist ein Zeiger auf die Mantelfunktion im Code des Interpreters. Der Name der Mantelfunktion im Interpreter muß mit dem Namen der Mantelfunktion im Batchprogramm nicht übereinstimmen.

2. Einen Zeiger auf die Parameterliste der Mantelfunktion. Die Parameterliste besteht aus Zeigern auf die Symboltabelleneinträge, deren Werte als Funktionsargumente bei der Ausführung der Mantelfunktion verwendet werden.
   Diese Konstruktion ermöglicht es, im Batchprogramm Funktionsaufrufe mit einer beliebigen Zahl von Parametern beliebigen Typs zuzulassen.

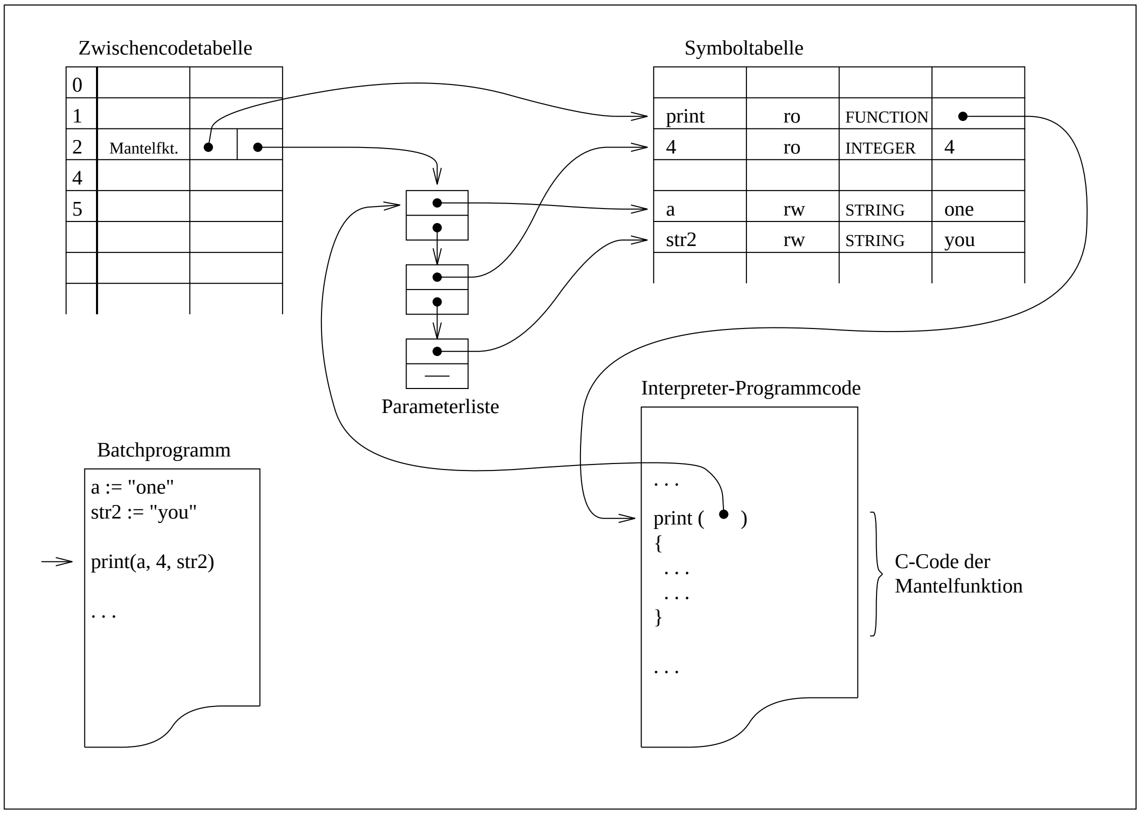

Abbildung 6.3: Zusammenspiel von Symboltabelle, Zwischencodetabelle und Parameterliste

Das Zusammenspiel von Symboltabelle, Zwischencodetabelle und Parameterliste ist in Abbildung 6.3 illustriert. In der Abbildung wird folgender Fall dargestellt: Der Zwischencodeinterpreter ist an einer `print`–Funktion angelangt. Sie ist vom Typ „Mantelfunktion“. Der Symboltabellenzeiger der Zwischencodetabelle zeigt auf den Namen der fest eingebauten Funktion `print` in der Symboltabelle. Der Wert dieses Symboltabelleneintrags ist ein Zeiger auf das Interpreter–Programmfragment, welches den Code für die `print`–Funktion enthält. Der zweite Zeiger der Zwischencodetabelle zeigt auf den Anfang der Parameterliste der `print`–Funktion. Die Listenelemente der Parameterliste beinhalten Zeiger auf die Symboltabelleneinträge der Argumente für die `print`–Funktion. Die `print`–Funktion im C–Code des Interpreters erhält als Argument einen Zeiger, der ebenfalls auf den Anfang der Parameterliste zeigt. Die Reihenfolge der Zeiger der Parameterliste entspricht der Reihenfolge der Parameter beim Aufruf der `print`–Funktion im Batchprogramm. Für jeden Funktionsaufruf im Batchprogramm, der Parameter benötigt, existiert eine eigene Parameterliste.
Das Batchprogramm ist links unten dargestellt.

Zugriffsoperationen auf der Zwischencodetabelle

Auf der Zwischencodetabelle sind folgende Zugriffsoperationen definiert:

- Eintragen einer Sprungfunktion in die Zwischencodetabelle.
- Eintragen einer arithmetisch–logischen Operation.
- Eintragen einer Mantelfunktion.

Das Eintragen in die Zwischencodetabelle geschieht folgendermaßen: Der Funktionstyp wird eingetragen. Sodann werden der Name der Funktion und ihre Parameter eingetragen und am Schluß der Zwischencode–Programmzähler um eins erhöht.

6.4 Scanner

Der Scanner bildet die erste Phase des Interpreters. Seine Hauptaufgabe besteht darin, die Zeichen des Quellprogramms in Symbole zu segmentieren, sogenannte Token. Weitere Aufgaben des Scanners sind: Eintragen von Bezeichnern in die Symboltabelle, Entfernen von Leerzeichen und Tabulatorzeichen aus dem Quellprogramm, Überlesen von Kommentaren und das Zählen von Zeilen. Das Zählen der Zeilen dient dazu, dem Programmierer nötigenfalls die Position von Fehlern angeben zu können.

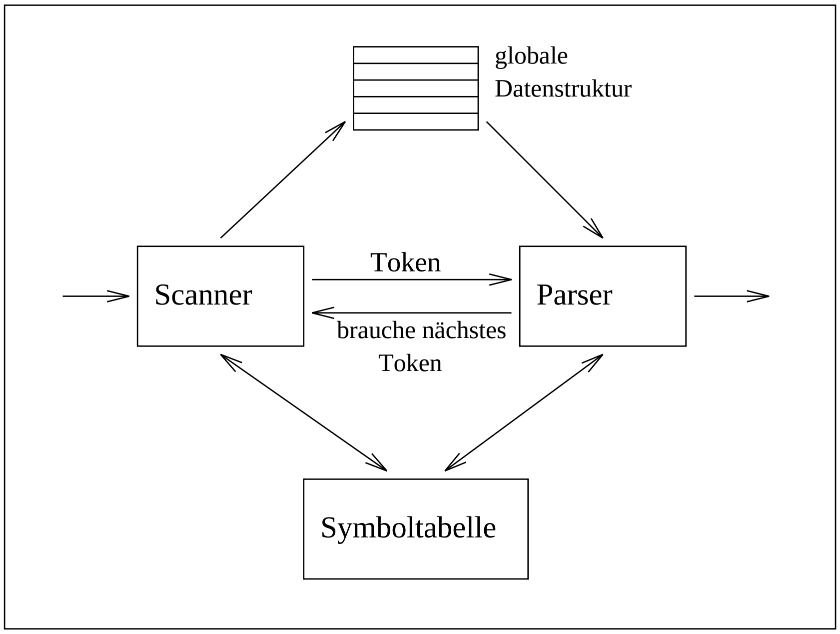

Abbildung 6.4: Zusammenarbeit von Scanner und Parser

Im Scanner wird der Teil der Grammatik realisiert, der die Syntax für Bezeichner enthält: ganze Zahlen, Fließkommazahlen, boolesche Werte, Namen von Variablen und Zeichenketten.

Befindet sich im Batchprogramm ein Symbol, das kein Schlüsselwort oder Operator ist und dessen Syntax zu keinem der möglichen Bezeichner paßt, so wird eine Fehlermeldung ausgegeben.

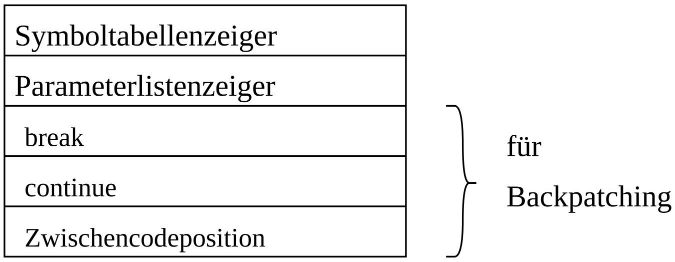

Abbildung 6.5: Die Datenstruktur Tokentyp

Der Scanner wird vom Parser aufgerufen und liefert bei jedem Aufruf ein Token zurück. Für Bezeichner beispielsweise wird immer das Token „Identifier“ zurückgegeben, Typ und Wert des Bezeichners werden in die Symboltabelle eingetragen, soweit bekannt. Dem Parser wird über eine globale Datenstruktur ein Zeiger auf den Symboltabelleneintrag zur Verfügung gestellt. Bei Schlüsselworten wie etwa „for“ und bei Operatoren wie „+“ oder „mod“ wird ein gleichnamiges Token zurückgegeben.

Die Zusammenarbeit von Scanner und Parser ist in Abbildung 6.4 dargestellt (nach [Aho et al. 88a], geändert).

Die globale Datenstruktur, die unter anderem zur Übergabe von Symboltabellenzeigern dient, wird als „Tokentyp“ bezeichnet. Sie wird zwar im Parser definiert, soll aber hier schon kurz vorgestellt werden, da der Scanner sie auch benutzt. Es handelt sich um eine Struktur mit fünf Elementen. Abbildung 6.5 zeigt die Bedeutung der einzelnen Elemente von Tokentyp. Das Element „Symboltabellenzeiger“ wurde schon erwähnt. Im Element „Parameterlistenzeiger“ kann ein Zeiger auf eine Parameterliste übergeben werden. Die übrigen Elemente werden vom Parser für das sogenannte „Backpatching“, das nachträgliche Eintragen von Sprungzielen in den Zwischencode, benötigt. Sie werden in Abschnitt 6.5 auf Seite 42 beschrieben.

6.5 Parser und Grammatik, Zwischencodeerzeugung

Der Parser spielt als Bindeglied zwischen Batchprogramm und Zwischencode eine Schlüsselrolle. Natürlich kann auf kein Modul, ja keine einzelne Zeile des Interpretercodes verzichtet werden. Jedoch findet in der Phase der Syntaxanalyse, die durch den Parser realisiert ist, die Umsetzung der rekursiven Struktur der Batchsprachen–Grammatik in den linearen Zwischencode statt, in welchem sich die Steuerung des Kontrollflusses nur noch in absoluten Sprunganweisungen wiederfindet.

Parser für Grammatiken lassen sich in drei Kategorien einteilen [Aho et al. 88a]. Universelle Parsemethoden wie zum Beispiel die Algorithmen von Cocke, Younger, Kasami (CYK–

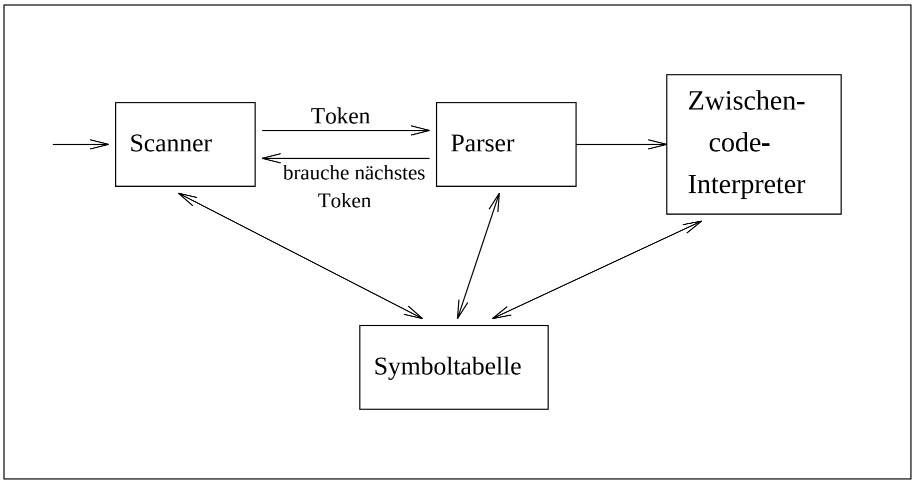

Abbildung 6.6: Die Position des Parsers im Interpreter

Algorithmus) und Early funktionieren für alle kontextfreien Grammatiken. Für den praktischen Einsatz sind diese Methoden jedoch zu ineffizient. Die in Compilern beziehungsweise Interpretern überwiegend verwendeten Methoden lassen sich als „top–down“– oder „bottom–up“–Methoden klassifizieren. „Top–down“ oder „von–oben–nach–unten“ bedeutet, daß sich der Parser bei der Konstruktion der internen Darstellung des Quellprogramms vom Startsymbol der Grammatik („top“) zu immer tiefer liegenden Produktionen („down“) durcharbeitet. Ein „bottom–up“–Parser hingegen beginnt seine Arbeit bei den Terminalen der Grammatik („bottom“) und arbeitet sich „rückwärts“ durch die Produktionen bis zum Startsymbol der Grammatik hinauf („up“). Beide Parsertypen lesen die Symbole der Eingabe von links nach rechts.

Die effizienten top–down– und bottom–up–Methoden sind auf Teilklassen von Grammatiken beschränkt. Genauer gesagt bezieht sich die Beschränkung auf Teilklassen kontextfreier Grammatiken, nämlich jene Teilklassen der sogenannten deterministisch kontextfreien Sprachen. Erkenner beziehungsweise Parser für solche Sprachen sind deterministische Kellerautomaten[2]. Einige dieser Teilklassen, wie zum Beispiel LL– und LR–Grammatiken, sind jedoch mächtig genug, um die meisten syntaktischen Konstrukte in Programmiersprachen damit beschreiben zu können. Von Hand geschriebene Parser arbeiten überwiegend mit LL–Grammatiken. Werkzeuge wie der hier verwendete Parsergenerator Bison generieren in der Regel Parser für die größere Klasse der LR–Grammatiken.

Bottom–up–Syntaxanalyse

In diesem Abschnitt soll die vom Parsergenerator Bison verwendete sogenannte „shift–reduce–Syntaxanalyse“ erläutert werden. Bison verwendet eine sehr allgemeine shift–reduce–Syntaxanalyse, die LR–Syntaxanalyse. Bei der shift–reduce–Syntaxanalyse versucht der Parser, wie bei bottom–up–Syntaxanalysemethoden üblich, die interne Darstellung für die

[2] Zur Einführung in die Theorie der Automaten und formalen Sprachen sei [Hopcroft et al. 79] empfohlen.

Eingabe von den Terminalen der Grammatik ausgehend aufzubauen. Shift–reduce bedeutet schiebe–reduziere. Jedesmal, wenn ein Eingabesymbol gelesen wird, wird in der Grammatik eine passende Produktion gesucht, die das Symbol auf ihrer rechten Seite enthält. Das Eingabesymbol wird sodann auf das Symbol, welches auf der linken Seite der Produktion steht, *reduziert*. Man betrachte als Beispiel die einfache Grammatik:

satz → *satz wort* | *wort*
wort → **dies** | **ist** | **ein** | **Satz**

Eine Eingabe wie „dies ist ein Satz" würde wie folgt reduziert:

1	**dies ist ein Satz**	
2	*wort* **ist ein Satz**	(**dies** wurde zu *wort* reduziert)
3	*satz* **ist ein Satz**	(*wort* wurde zu *satz* reduziert)
4	*satz wort* **ein Satz**	(**ist** wurde zu *wort* reduziert)
5	*satz* **ein Satz**	(*satz wort* wurde zu *satz* reduziert)
6	*satz wort* **Satz**	(**ein** wurde zu *wort* reduziert)
7	*satz* **Satz**	(*satz wort* wurde zu *satz* reduziert)
8	*satz wort*	(**Satz** wurde zu *wort* reduziert)
9	*satz*	(*satz wort* wurde zu *satz* reduziert)

Der Parser sucht hier immer nach der Produktion, die in der Eingabe so weit links wie möglich angewandt werden kann. In der dritten Zeile kommt nun auch das Schieben zum Einsatz: das Nichtterminal *satz* kann nicht allein reduziert werden, da es keine Produktion der Grammatik gibt, die nur *satz* auf der rechten Seite enthält. Da die Eingabe noch nicht zu Ende ist, wird das Nichtterminal *satz* also zunächst auf den Stapel geschoben, bis sich durch weitere Reduzieraktionen wieder eine anwendbare Produktion finden läßt. Dies ist, nachdem in der vierten Zeile das Terminal **ist** zu *wort* reduziert worden ist, der Fall. In der fünften Zeile kann dann *satz wort* zu *satz* reduziert werden.

In der betrachteten Grammatik ist es immer eindeutig, welche Produktion zur Anwendung kommen soll. Bei komplexeren Grammatiken ist diese Frage aber nicht immer so einfach zu lösen. So gibt es kontextfreie Grammatiken, für die die shift–reduce–Syntaxanalyse nicht eingesetzt werden kann. Jeder shift–reduce–Parser kann einen Zustand erreichen, in welchem er nicht erkennen kann, ob er schieben oder reduzieren soll (ein schiebe–reduziere–Konflikt), oder einen Zustand, in welchem er nicht entscheiden kann, nach welcher Produktion er reduzieren soll (ein reduziere–reduziere–Konflikt). Technisch gesehen sind solche Grammatiken nicht in der Klasse der LR(k)–Grammatiken (das k bezieht sich auf die Zahl der im voraus betrachteten Eingabesymbole). Glücklicherweise kann ein schiebe–reduziere–Konflikt oft dadurch gelöst werden, daß das Schieben im Zweifelsfall bevorzugt wird. Reduziere–reduziere–Konflikte können für die in Programmiersprachen auftretenden Konstrukte durch Umschreiben der Grammatik vermieden werden.

Auflösung des if–then–else–Konflikts

Ein Fall, der — wie in den meisten Programmiersprachen — auch in der hier beschriebenen Batchsprache vorkommt, ist der if–then–else–Konflikt. Die Grammatik für die zwei möglichen if–then–Anweisungen lautet:

$$\begin{aligned} &\mathit{statement} \rightarrow \ldots \\ &\qquad \mid \textbf{if}\ \mathit{expression}\ \textbf{then}\ \mathit{block}\ \textbf{endif} \\ &\qquad \mid \textbf{if}\ \mathit{expression}\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \mathit{block}\ \textbf{endif} \end{aligned}$$

Das Problem der Mehrdeutigkeit scheint hier zwar durch das Schlüsselwort **endif** behoben zu sein, vorausgesetzt, man bevorzugt das Schieben. Nachdem der Parser **if** *expression* **then** *block* gelesen hat, kann er aufgrund des nächsten Symbols entscheiden, welche Produktion gemeint ist. Leider sind aber in der Implementierung die Produktionen der Grammatik noch durch die in Abschnitt 6.1 erwähnten semantischen Aktionen ergänzt, sodaß die Grammatik folgende Form hat (die semantischen Aktionen sind durch geschweifte Klammern angedeutet):

$$\begin{aligned} &\mathit{statement} \rightarrow \ldots \\ &\qquad \mid \textbf{if}\ \mathit{expression}\ \{\ \ldots\ \}\ \textbf{then}\ \mathit{block}\ \textbf{endif}\ \{\ \ldots\ \} \\ &\qquad \mid \textbf{if}\ \mathit{expression}\ \{\ \ldots\ \}\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \{\ \ldots\ \}\ \mathit{block}\ \textbf{endif}\ \{\ \ldots\ \} \end{aligned}$$

Da Bison den Inhalt der Aktionen nicht versteht, betrachtet er selbst zwei Aktionen identischen Inhalts als unterschiedlich. So weiß er nun nach dem Lesen von **if** *expression* { ... } nicht, in welcher Produktion er sich befindet. Erst das Erreichen von **endif** beziehungsweise **else** würde Klarheit bringen. Weil ein LR(1)–Parser aber nur ein Eingabesymbol im voraus betrachtet, entsteht ein nicht auflösbarer reduziere–reduziere–Konflikt. Abhilfe schafft eine Änderung der Grammatik. Nach Einfügen einer *ifExpr*–Produktion ist die Grammatik wieder in LR(1):

$$\begin{aligned} &\mathit{statement} \rightarrow \ldots \\ &\qquad \mid \mathit{ifExpr}\ \textbf{then}\ \mathit{block}\ \textbf{endif}\ \{\ \ldots\ \} \\ &\qquad \mid \mathit{ifExpr}\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \{\ \ldots\ \}\ \mathit{block}\ \textbf{endif}\ \{\ \ldots\ \} \\ &\mathit{ifExpr} \rightarrow \textbf{if}\ \mathit{expression}\ \{\ \ldots\ \} \end{aligned}$$

Auswertung von Ausdrücken

Wie schon in Abschnitt 5.2 auf Seite 19 angedeutet, legt die hier verwendete Grammatik die Reihenfolge der Auswertung von Ausdrücken noch nicht fest. Ohne weiteres Eingreifen würde dies, neben zahllosen schiebe–reduziere–Konflikten im Parser, zu einer mehr oder weniger undefinierten Auswertungsreihenfolge führen. Die Reihenfolge wird deshalb durch Angabe von Operatorpräzedenzen in der Quelldatei für den Parsergenerator Bison angegeben. Diese Angaben legen fest, wie stark ein- und zweistellige Operatoren an ihre Operanden gebunden werden. So werden auch ungeklammerte Ausdrücke in der Reihenfolge ausgewertet, wie es den Erwartungen der meisten Programmiererinnen und Programmierer entsprechen dürfte (zum Beispiel Punkt- vor Strichrechnung).
Bei der Angabe der Operatorpräzedenzen wurde dieselbe Auswertungsreihenfolge zugrunde gelegt, wie sie in der Programmiersprache C verwirklicht worden ist, sofern der jeweilige Operator der Batchsprache in C vorhanden ist. Die Reihenfolge ist der Tabelle 6.2 zu entnehmen. Der oberste Operator hat die höchste, der unterste die niedrigste Präzedenz.

Operator	Name	Assoziativität	Präzedenz
$+, -$	Vorzeichen	rechts	9
not, !	logische Negation	rechts	8
sqrt	Quadratwurzel	rechts	7
ln	natürlicher Logarithmus	rechts	7
log	Logarithmus zur Basis 10	rechts	7
$**$, ˆ	Exponentialfunktion	links	6
$*$	Multiplikation	links	5
/	Division	links	5
div	Ganzzahl-Division	links	5
mod, %	Restwert	links	5
$+$	Addition	links	4
$-$	Subtraktion	links	4
$<$	kleiner als	links	3
$<=, =<$	kleiner gleich	links	3
$>$	größer als	links	3
$>=, =>$	größer gleich	links	3
$==$	gleich	links	2
$<>, != $	ungleich	links	2
and, &&	logisches UND	links	1
or, $\vert\vert$	logisches ODER	links	1

Tabelle 6.2: Operatorpräzedenzen der Batchsprache

Operatoren im gleichen durch waagrechte Linien begrenzten Feld haben die gleiche Präzedenz.

Die Spalte „Assoziativität“ gibt an, ob die Teilausdrücke von rechts nach links („rechts“) oder von links nach rechts („links“) ausgewertet werden, für den Fall, daß mehrere gleichrangige Operatoren in einem Ausdruck stehen. Zum Beispiel ergibt der Ausdruck $5 - 4 - 3$ hier -2, da zuerst $5 - 4$ und dann der Rest ausgewertet wird. Wäre die Assoziativität „rechts“, so wäre das Ergebnis 4, da dann in der Reihenfolge $5 - (4 - 3)$ ausgewertet würde.

Die Vorzeichen $+$ und $-$ haben die höchste Präzedenz, sind also am stärksten an den Bezeichner rechts neben ihnen gebunden (sinnvollerweise eine Zahl oder eine Variable mit numerischem Wert). Man beachte, daß sie nie an einen Ausdruck binden können, sofern er nicht geklammert ist, da sie ja die höchste Präzedenz aller Operatoren besitzen.

Neben den Vorzeichen gibt es drei weitere rechtsassoziative unäre Operatoren: ln, log und sqrt. Ihre Präzedenz liegt unmittelbar unter der der Vorzeichen (und der der logischen Negation, die aber nur auf booleschen Operanden definiert ist). Man beachte, daß diese Funktionen (natürlicher Logarithmus, Logarithmus zur Basis 10 und Quadratwurzel) als

Operatoren und nicht, wie in vielen Programmiersprachen, als Funktionsaufrufe implementiert sind. Dies bedeutet, daß ihre Argumente nicht in Klammern zu stehen brauchen, solange es sich nicht um Ausdrücke handelt.

Alle anderen Operatoren sind linksassoziativ, werden also von links nach rechts ausgewertet, wenn mehrere gleiche Operatoren in einer Zeile stehen. Die logischen Verknüpfungsoperatoren UND und ODER haben die niedrigste Präzedenz. Alle Teilausdrücke rechts und links von ihnen werden ausgewertet, bevor UND beziehungsweise ODER ausgewertet werden.

Modell eines LR–Parsers

Abschließend sei noch das Modell eines LR–Parsers in Abbildung 6.7 abgebildet. Im unteren Teil der Abbildung ist die Syntaxanalysetabelle angedeutet. Sie besteht aus zwei Teilen (Aktion und Sprung). Das Treiberprogramm ist für alle Parser das gleiche, nur die Syntaxanalysetabelle wechselt, je nach Grammatik. Das aktuelle Eingabesymbol und der Inhalt des Stapels werden dazu benutzt, die Syntaxanalysetabelle zu indizieren. Aus ihr geht hervor, ob ein Sprung zu einem neuen Zustand oder eine Aktion (schieben oder reduzieren und neuen Zustand auf den Stapel legen) gemacht werden soll.

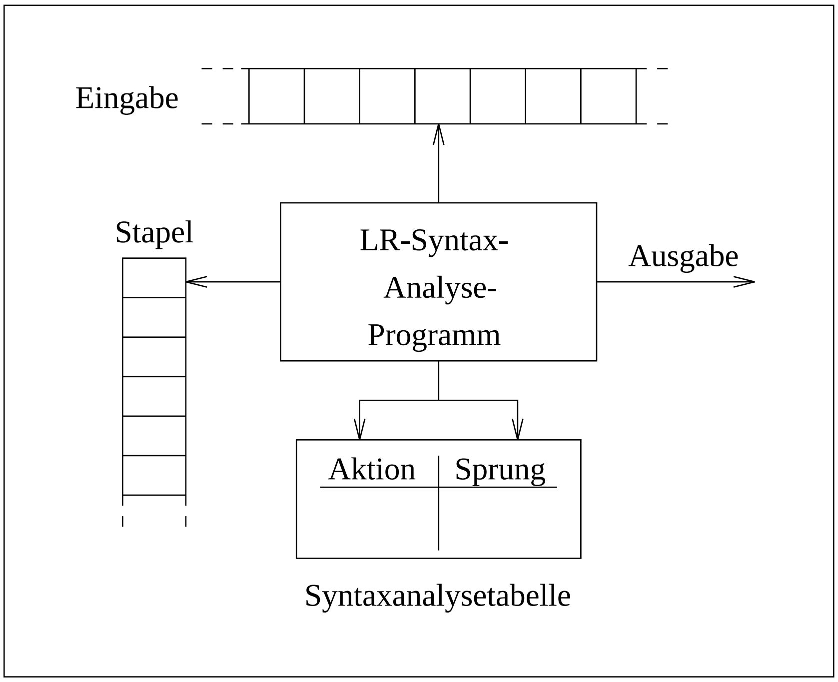

Abbildung 6.7: Modell eines LR–Parsers

Zwischencodeerzeugung

Der bisherige Teil des Kapitels schilderte die Aufgabe des Parsers, zu überprüfen, ob das Batchprogramm der Grammatik für die Batchsprache genügt. Die interessanteste Aufgabe des Parsers ist aber die der Zwischencodeerzeugung, bei der die rekursive Struktur der Grammatik in eine lineare Abfolge von Zwischencodebefehlen umgesetzt wird. Dabei muß natürlich die Semantik des Batchprogramms erhalten bleiben. Genaugenommen treffen sich in der Steuerdatei für den Parsergenerator Bison vier verschiedene Sprachen: die Syntax für die Steuerdatei selber, die Grammatik der Batchsprache, die Sprache C, welche für die semantischen Aktionen verwendet wird und die Syntax des zu erzeugenden Zwischencodes.

Was semantische Aktionen sind und wie sie eingesetzt werden, wird im nächsten Abschnitt erläutert.

Semantische Aktionen

Semantische Aktionen sind C–Code, der in die rechten Seiten von Produktionen der Batchsprachen–Grammatik eingestreut wird. Ohne die semantischen Aktionen könnte der Parser nur die Syntax des Batchprogramms überprüfen. Der C–Code der Aktionen wird ausgeführt, wenn der Parser beim Vergleich der Eingabe mit der entsprechenden Produktion der Grammatik an der semantischen Aktion angelangt ist. Man könnte sagen, daß der Parser semantische Aktionen als Nichtterminale betrachtet, die zu nichts reduziert werden (leere Produktionen) und als Seiteneffekt den in ihnen enthaltenen C-Code ausführt.

Ein einfaches Beispiel für eine Produktion mit einer semantischen Aktion ist die folgende Produktion für den +–Operator:

$$expression \rightarrow expression + expression \qquad \{\ \$\$ = \$1 + \$3;\ \}$$

Die semantische Aktion (in den geschweiften Klammern) bedeutet nun folgendes: Addiere den Wert des ersten Ausdrucks auf der rechten Seite der Produktion zu dem Wert des zweiten Ausdrucks auf der rechten Seite der Produktion und weise das Ergebnis dem Nichtterminal *expression* auf der linken Seite der Produktion zu. Die Dollar–Zeichen bezeichnen den Wert der Symbole einer Produktion. Der Wert von $$ wird dem Nichtterminal auf der linken Seite der Produktion zugewiesen; $1 wird durch den Wert des ersten Symbols auf der rechten Seite der Produktion ersetzt, $2 durch den Wert des zweiten Symbols (hier das +–Zeichen) und so weiter.

Betrachtet man nun einen größeren Ausdruck aus mehreren Teilausdrücken, zum Beispiel $2 + 3 + 4$, so werden zwei Ausdrücke unter Beachtung der Auswertungsreihenfolge ineinander geschachtelt: zunächst wird $2 + 3$ ausgewertet und dann $(2 + 3) + 4$. Dabei entstehen beim Durchlaufen der semantischen Aktionen und nach dem Ersetzen der $–Variablen zwei C–Anweisungen:

$$\begin{aligned} &expression1 = 2 + 3; \\ &expression2 = expression1 + 4; \end{aligned}$$

expression1 und *expression2* können hier als die Namen temporärer Variablen angesehen werden, die vom Parser zum Zwischenspeichern der Ergebnisse von Ausdrücken bereitgestellt werden. Die Variable *expression2* enthält nun das Ergebnis des Gesamtausdrucks. Sie kann in später ausgewerteten (oder anders gesagt, in der Hierarchie der Grammatik höher liegenden) Produktionen weiterverwendet werden.

Nun scheint es naheliegend zu sein, für jede Produktion der Batchsprachen–Grammatik den ihrer Semantik entsprechenden C–Code in die semantischen Aktionen zu schreiben und C–Anweisungen als Zwischencode zu benutzen. Dies funktioniert auch für die Teilgrammatik der Ausdrücke, nicht jedoch für die Teilgrammatiken der Kontrollstrukturen wie Schleifen. Warum nicht, sei am Beispiel der while–Schleife erläutert.

Man betrachte zunächst die Produktion der Grammatik für eine while–Schleife:

$$
\begin{aligned}
&\textit{statement} \rightarrow \ldots \\
&\qquad \mid \textbf{while } \textit{expression } \textbf{do } \textit{block } \textbf{endwhile} \\
&\textit{block} \rightarrow \textit{statement} \\
&\qquad \mid \textit{block statementDelimiter statement} \\
&\textit{statementDelimiter} \rightarrow \textbf{;} \mid \textbf{newlineCharacter}
\end{aligned}
$$

Würde der *block* immer nur aus einer einzigen Anweisung bestehen, so wäre es noch denkbar, die zugehörige C–Anweisung oder –Funktion über einen Zeiger, der auf sie zeigt, innerhalb einer C–while–Schleife auszuführen. In diesem Fall wären aber geschachtelte Schleifen nicht möglich. Die Forderung des Entwurfs, Kontrollstrukturen wie aus C oder Pascal zur Verfügung zu stellen, wäre damit nicht erfüllt. Es sollten sowohl geschachtelte Schleifen möglich sein als auch Schleifenblöcke, die mehr als eine Anweisung enthalten. So wurde die Verwendung eines Zwischencodes und des Zwischencodeinterpreters nötig.

Befehle des Zwischencodes

Der Zwischencode, der in dem Batchinterpreter verwendet wird, enthält drei Arten von Befehlen, die schon im Abschnitt „Zwischencodetabelle“ auf Seite 28 vorgestellt wurden. Es sind dies:

- Sprungbefehle mit zwei Parametern: Sprungziel und Sprungbedingung.
- Arithmetisch–logische Operationen und die Zuweisung im 3–Adreß–Format (Ergebnis, Operand 1 und Operand 2).
- Mantelfunktionen mit einem Parameter.

Im einzelnen sind folgende Zwischencodebefehle vorhanden:

1. Sprungbefehle:

Name	Zweck
jmp	unbedingter Sprung
jmp_true	Sprung, wenn die Bedingung den Wert TRUE hat
jmp_false	Sprung, wenn die Bedingung den Wert FALSE hat

2. Arithmetisch–logische Operationen und Zuweisung:

Name	Zweck
assign	Zuweisung
sq_rt	Quadratwurzel
ln	natürlicher Logarithmus
lg	Logarithmus zur Basis 10
add	Addition, Verkettung von Zeichenketten
sub	Subtraktion
mult	Multiplikation
dvde	Division (ganze Zahlen und Fließkommazahlen)
intdiv	ganzzahlige Division
mod	Modulo–Operation
raise	Exponentialfunktion
not	logische Negation
and	logische UND–Verknüpfung
or	logische ODER–Verknüpfung
less	kleiner als
greater	größer als
less_eq	kleiner gleich
great_eq	größer gleich
eq	gleich (äquivalent)
not_eq	ungleich

Die Inkrement–Operation wird im Zwischencode durch Addition von 1 realisiert, die arithmetische Negation durch Subtraktion des Operanden von 0.

3. Mantelfunktionen:

Name	Zweck
print	Drucken von Argumenten
execute	Ausführen von UNIX–Befehlen
exit	Verlassen des Batchinterpreters
setInitFunc	Initialisierungsfunktion wählen
setLearnFunc	Lernfunktion wählen
setUpdateFunc	Updatefunktion wählen
setPruningFunc	Pruningfunktionen wählen
setSubPattern	Subpattern shifting scheme definieren
setShuffle	Shuffle–Modus ändern
setSubShuffle	Subpattern–Shuffle–Modus ändern
setParallelMode	Parallele Verarbeitung ein– oder ausschalten
initNet	Netz initialisieren
loadNet	Netz laden
saveNet	Netz speichern
saveResult	Ergebnisdatei speichern
trainNet	Netz trainieren
testNet	Netz testen
pruneNet	Netz „prunen“
loadPattern	Musterdatei laden
setPattern	aktuelle Musterdatei setzen
delPattern	Musterdatei löschen

Zu jeder Art von Zwischencodebefehl gibt es eine Funktion, die den Zwischencodebefehl mit seinen Parametern in die Zwischencodetabelle einträgt. Diese Funktionen werden in den semantischen Aktionen der Grammatik aufgerufen: dies ist die sogenannte Erzeugung des Zwischencodes.

Ein Zwischencodebefehl führt typischerweise folgende Aufgaben aus:

1. Die Argumente des Zwischencodebefehls werden aus der Symboltabelle geholt. Argumente des Zwischencodebefehls sind immer Symboltabelleneinträge.
2. Der Typ der Argumente wird überprüft, gegebenenfalls auch ihr Wert (zu groß, zu klein). Die Anzahl der Argumente (zu viele, zu wenige) wird ebenfalls überprüft.
3. Die gewünschte Funktion wird ausgeführt, zum Beispiel eine Addition oder ein SNNS–Kern–Funktionsaufruf. Da arithmetische Operatoren meist überladen sind, muß der Typ der Operation ermittelt werden (zum Beispiel führt der +–Operator eine ganzzahlige Addition, Fließkommaaddition und Verkettung von Zeichenketten durch).
4. Wenn die Funktion den Wert von Systemvariablen beeinflussen soll, müssen die entsprechenden Symboltabelleneinträge geändert werden. Bei SNNS–Kern–

Funktionsaufrufen wird eine Meldung über den Erfolg der Funktion ausgegeben. Interne Variable des Zwischencodeinterpreters werden aktualisiert.

Welche Parameter ein Zwischencodebefehl erwartet und mit welchen Typen von Argumenten er arbeitet, ist in Abschnitt 6.6 auf Seite 45 beschrieben.

Der folgende Abschnitt beschreibt das Backpatching, eine weitere wichtige Aufgabe des Parsers.

Backpatching

Als Teil der Zwischencodeerzeugung ist das Backpatching der Vorgang, bei dem die Ziele für die Sprunganweisungen des Zwischencodes nachträglich in die Sprunganweisungen eingetragen werden. Das Wort „Backpatching“ kann nur mit „nachträgliches Einsetzen von Sprungzielen“ übersetzt werden, weshalb die englische Form hier ausnahmsweise beibehalten wird und, schlimmer noch, rigoros verbalisiert wird.

Beim Erzeugen des Zwischencodes für Kontrollstrukturen wie etwa eine while–Schleife müssen mehrere Sprunganweisungen in die Zwischencodetabelle eingetragen werden. Im Fall der while–Schleife sind es zwei: Ein erster, bedingter Sprung befindet sich unmittelbar nach der letzten der Zwischencodeanweisungen, welche die Schleifenbedingung abprüfen (die Schleifenbedingung kann ein größerer Ausdruck sein) und führt bis hinter die letzte Anweisung des Blocks („aus der Schleife heraus“). Ein zweiter, unbedingter Sprung führt vom Ende des Blocks wieder zu der ersten der Zwischencodeanweisungen, welche die Schleifenbedingung abprüfen. Der erste Sprung wird ausgeführt, wenn die Schleifenbedingung den Wert FALSE annimmt. Bei den anderen Kontrollstrukturen (if–then–endif, if–then–else–endif, for–to–do–endfor, repeat–until) werden entsprechende Sprünge benötigt.

Wird nun der Zwischencode für eine while–Schleife erzeugt, so ist beim Eintragen des ersten Sprungs das Sprungziel, nämlich die Position der Zwischencodeanweisung nach der letzten Anweisung des Blocks, noch nicht bekannt. Es wird eine Sprunganweisung in die Zwischencodetabelle eingetragen, bei der das Sprungziel noch offen ist. Umgekehrt ist bei dem zweiten Sprung zwar das Sprungziel bekannt, nämlich der Beginn der Zwischencodeanweisungen, welche die Schleifenbedingung abprüfen, die Sprunganweisung kann aber erst viel später eingetragen werden, nämlich nach der letzten Zwischencodeanweisung des Blocks. Hier wird das Sprungziel für das spätere Eintragen in die Sprunganweisung zwischengespeichert.

Nachdem der Parser an dem Schlüsselwort „endwhile“ (für das Beispiel while–Schleife) angelangt ist, kann die offene Sprunganweisung „gebackpatcht“ werden, also nachträglich mit ihrem Sprungziel versorgt werden. Das gespeicherte Sprungziel wird beim Eintragen des zweiten Sprungs miteingetragen.

Die break– und die continue–Anweisung werden jeweils als unbedingte Sprünge in die Zwischencodetabelle eingetragen. Die break–Anweisung verläßt eine Schleife; die continue–Anweisung veranlaßt den Zwischencodeinterpreter, sofort wieder zur Schleifenbedingung zu springen und diese auszuwerten beziehungsweise im Fall einer for–Schleife den Schleifenzähler zu erhöhen. Beide Sprünge müssen ebenfalls gebackpatcht werden. Da ein Schleifenblock beliebig viele break– und continue–Anweisungen besitzen kann, ist die Anzahl der Sprunganweisungen, die gebackpatcht werden müssen, unbekannt. Daher werden die Positionen dieser Sprunganweisungen in Listen gespeichert: eine Liste für die Sprünge, die zur Schleifenbedingung führen und eine Liste für die Sprünge, die hinter das Schleifenende führen (siehe auch Abbildung 6.5 auf Seite 32).

Das hier verwendete Verfahren für das Backpatching wurde für den Zwischencodeinterpreter neu entwickelt. Das in dem Standardwerk [Aho et al. 88a] vorgeschlagene Verfahren ist für die Erzeugung von maschinennahem Zwischencode gedacht und für den hier verwendeten, vergleichsweise mächtigen Zwischencode ungeeignet. Außerdem sind darin weder break– noch continue–Anweisungen vorgesehen.

Funktionen für Backpatching

Das Programm-Modul für Backpatching enthält folgende Funktionen:

- Eine Funktion `makelist` zum Erstellen einer Sprunganweisungsliste.
- Eine Funktion `merge` zum Verketten zweier Sprunganweisungslisten.
- Die Funktion `bp_backpatch`, die alle Sprunganweisungen einer Liste backpatcht.

Im Programm–Modul für die Zwischencodetabelle gibt es noch eine Schnittstellenfunktion zu dem Backpatching–Modul namens `backpatch`, deren Aufgabe es ist, eine einzelne Sprunganweisung zu backpatchen.

Mit den angegebenen Zwischencodefunktionen können alle Konstrukte der Batchsprache realisiert werden. In den meisten Fällen kann ein Konstrukt der Batchsprache direkt auf eine einzige Zwischencodefunktion abgebildet werden (zum Beispiel wird der +–Operator auf die Zwischencodefunktion `add` abgebildet). Die einzigen Ausnahmen sind die while– und die for–Schleife sowie die if–then–else–endif–Anweisung, die jeweils zwei oder mehr Zwischencodeanweisungen benötigen. Abschließend sei hier noch in Abbildung 6.8 am Beispiel der while–Schleife in natürlichsprachlicher Notation wiedergegeben, welche semantischen Aktionen der Parser ausführt und welcher Zwischencode entsteht.

Zu Abbildung 6.8: Der Parser arbeitet, ausgehend vom Nichtterminal *statement*, den rechten Teil der Produktion für die while–Schleife mit den semantischen Aktionen (in geschweiften Klammern) ab. Mit „Label1, Label2“ sind Platzhalter gemeint, die den Stand des Zwischencode–Programmzählers speichern. Die umrandeten Befehle `jmp` und `jmp_false` sind Zwischencodeanweisungen. Für die while–Schleife selber werden nur diese beiden Anweisungen erzeugt, der Rest sind C–Anweisungen für den Parser. Der Rest des Zwischencodes für die while–Schleife wird in den semantischen Aktionen für *expression* und *block* erzeugt. Man sieht, daß bei der ersten Sprunganweisung `jmp_false` das Sprungziel noch offen ist (bei der Anweisung wurde der zweite Parameter, die Sprungbedingung, zur Vereinfachung weggelassen). Bei der zweiten Sprunganweisung kann das Sprungziel direkt ein-

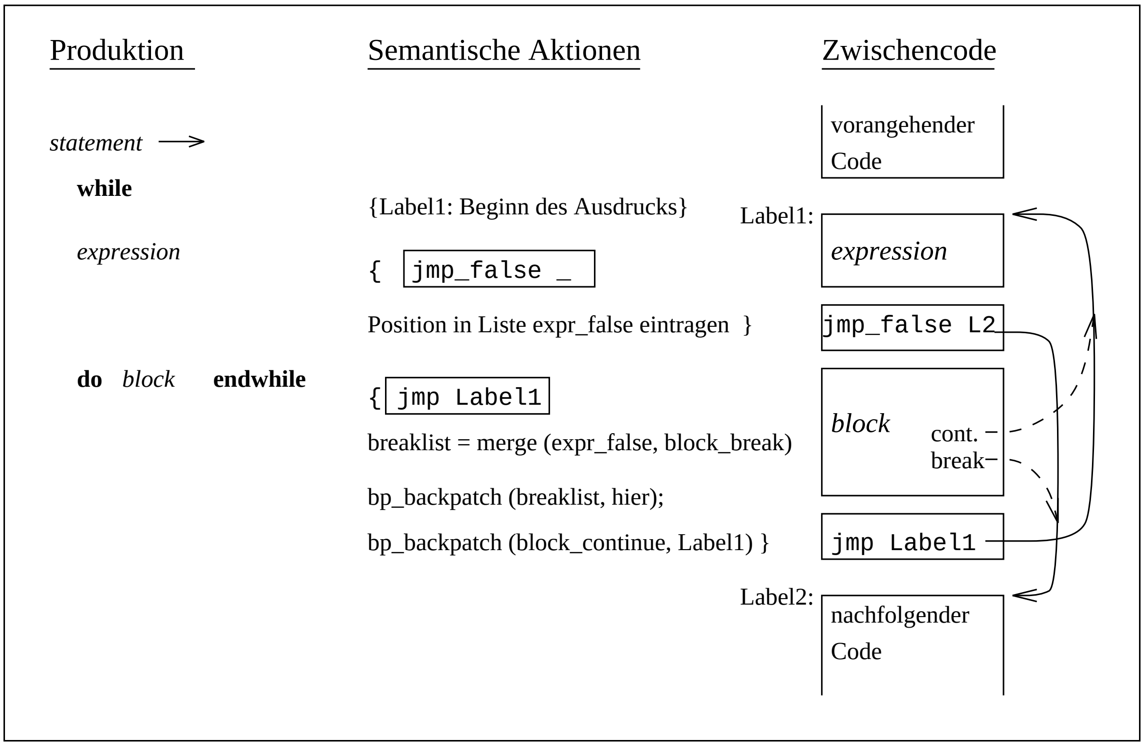

Abbildung 6.8: Zwischencodeerzeugung im Parser

getragen werden, da es seit der ersten semantischen Aktion bekannt ist.

Die Funktionen `merge` und `bp_backpatch` wurden bereits auf Seite 43 erwähnt. `block_break` und `block_continue`[3] sind die Backpatch– beziehungsweise Sprunganweisungslisten für die `break`– und `continue`–Anweisungen des Schleifenblocks *block*. Die Sprunganweisungen der `block_continue`–Liste werden direkt mit dem Sprungziel Label1 gebackpatcht. Die `block_break`–Liste hingegen wird vor dem Backpatchen mit der Liste `expr_false` verbunden, welche die Position der `jmp_false`–Anweisung enthält. Sodann wird die neue Liste `breaklist` mit dem augenblicklichen Stand des Zwischencode–Programmzählers („hier“) gebackpatcht.

Die rechte Seite des Bildes zeigt vereinfacht den Aufbau des entstandenen Zwischencodes, eingeschlossen in den vorangehenden und den nachfolgenden Code, der durch andere Produktionen der Grammatik erzeugt wurde. Die Verzweigungen des Kontrollflusses durch die Sprunganweisungen sind durch Pfeile kenntlich gemacht.

[3] Zur Veranschaulichung sind hier andere Namen verwendet worden als in der Implementierung, bei der die unübersichtlichere $-Notation von Bison verwendet werden muß.

6.6 Zwischencodeinterpreter

Die Aufgabe des Zwischencodeinterpreters ist es, die Befehle der Zwischencodetabelle der Reihe nach auszuführen. Die Funktion, welche die Interpreterschleife beinhaltet, heißt `run`. Sie besteht nur aus einer Schleife, die Befehl für Befehl aus der Zwischencodetabelle holt und den dazugehörigen Zeiger auf die Zwischencodefunktion ausführt. Die Funktion `run` ist beendet, wenn der Zwischencode–Programmzähler hinter dem letzten Befehl des Zwischencodes steht. Weitere Funktionen für den Zwischencodeinterpreter sind:

- `set_ic_pc`: setzt den Programmzähler auf einen neuen Wert (wird von den Sprunganweisungen benötigt)
- `get_ic_pc`: gibt den Stand des Programmzählers zurück
- `get_ic_pos`: gibt den Index des letzten Zwischencodebefehls zurück

Zum Zwischencodeinterpreter gehören natürlich auch die auf Seite 39ff aufgelisteten Zwischencodefunktionen. Die einzelnen Funktionen arbeiten oft mit mehreren möglichen Parametertypen. Sprungbefehle erhalten als Parameter das Sprungziel und den Symboltabelleneintrag der Variable, von deren Wert es abhängt, ob der Sprung ausgeführt wird oder nicht. Mantelfunktionen erhalten aus der Sicht des Zwischencodeinterpreters alle den gleichen Parameter als Argument: einen Zeiger auf ihre Parameterliste. Die Funktionen, die SNNS–Kern–Aufrufe durchführen, sind in Abschnitt 6.7 auf Seite 47 näher beschrieben. Die Funktionen `execute`, `exit` und `print` sind in Abschnitt 6.8 erläutert. Die Parameter der arithmetischen und logischen Operatoren sind im folgenden zusammengefaßt.

Parameter der arithmetischen und logischen Operatoren

In den folgenden Tabellen sind die zulässigen Parametertypen der arithmetischen Operatoren aufgeführt. Am Schnittpunkt, an dem sich die Zeile des ersten Operanden und die Spalte des zweiten Operanden kreuzen, steht der Typ des Ergebnisses der Operation. Zum Beispiel ergibt die Addition zweier ganzer Zahlen (Typ INTEGER) wieder eine ganze Zahl, die Addition einer ganzen Zahl und einer Zeichenkette (Typ STRING) eine Zeichenkette. Links oben in jeder Tabelle steht der Name der Zwischencodefunktion.

Die Zwischencodefunktionen `not`, `and` und `or` sind nur für Parameter von booleschem Typ (BOOL) definiert. Ihr Ergebnis ist immer von booleschem Typ.

Die Vergleichsfunktionen `less`, `greater`, `less_eq`, `great_eq`, `eq` und `not_eq` sind jeweils für Parameter von booleschem Typ, für ganze Zahlen und für Fließkommazahlen definiert. Ganze Zahlen und Fließkommazahlen können gemischt als Operanden auftreten. Ist ein Operand von booleschem Typ, so muß der andere auch von booleschem Typ sein. Das Ergebnis aller Vergleichsfunktionen ist immer von booleschem Typ.

add		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	REAL	—	STRING
Operand 1	REAL	REAL	REAL	—	STRING
	BOOL	—	—	—	STRING
	STRING	STRING	STRING	STRING	STRING

sub		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	REAL	—	—
Operand 1	REAL	REAL	REAL	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

mult		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	REAL	—	—
Operand 1	REAL	REAL	REAL	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

dvde		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	REAL	REAL	—	—
Operand 1	REAL	REAL	REAL	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

intdiv		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	—	—	—
Operand 1	REAL	—	—	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

mod		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	—	—	—
Operand 1	REAL	—	—	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

raise		Operand 2			
		INTEGER	REAL	BOOL	STRING
	INTEGER	INTEGER	REAL	—	—
Operand 1	REAL	REAL	REAL	—	—
	BOOL	—	—	—	—
	STRING	—	—	—	—

6.7 SNNS-Funktionsschnittstelle

Die meisten der sogenannten Mantelfunktionen des Batchinterpreters dienen dazu, den Code für den Aufruf von Funktionen des SNNS–Kerns bereitzustellen (die einzigen Ausnahmen sind die `print`–, die `exit`– und die `execute`–Funktion). Die Funktionen des SNNS–Kerns direkt aus dem Batchprogramm aufzurufen wäre aus mehreren Gründen sehr unzweckmäßig:

- Die Funktionen des SNNS–Kerns bilden die Kommandos der graphischen Benutzeroberfläche nicht eins zu eins ab. Für manche Kommandos sind mehrere SNNS–Kern–Funktionsaufrufe notwendig, ein Kommando hingegen ändert nur den Wert von Variablen, die erst bei späteren Funktionsaufrufen benötigt werden. Die Semantik der Kommandos der Benutzeroberfläche sollte aber erhalten bleiben.
- Einige Funktionen des SNNS–Kerns benötigen Parameter, die normalerweise nicht vom Benutzer der Oberfläche beziehungsweise vom Programmierer des Batchprogramms verändert werden. Auf diese Parameter sollte der Programmierer keinen Zugriff haben.
- Die Fehlermeldungen des SNNS–Kerns müssen aufbereitet werden, da sie im gleichen Format erscheinen sollen wie Fehlermeldungen des Batchinterpreters.
- Manche Funktionen des SNNS–Kerns reagieren auf falsche Parameter (zu wenige, falscher Typ) mit einem Absturz des Kerns. Um dies zu vermeiden, ist eine genaue Prüfung aller Parameter notwendig.

Daher wurden die Funktionen des SNNS–Kerns in Mantelfunktionen eingebettet. Wie bei allen Mantelfunktionen kann der Funktionsname, mit dem die Mantelfunktion im Batchprogramm aufgerufen werden muß, unabhängig vom Namen der Mantelfunktion im C–Code des Batchinterpreters gewählt werden.

In der folgenden Tabelle sind nun noch einmal alle Funktionen des Batchinterpreters aufgeführt, die SNNS–Funktionen ausführen. In der mittleren Spalte findet sich eine kurze Erklärung der Funktionen und rechts sind die Namen der SNNS–Kern–Funktionsaufrufe angegeben, die zur Ausführung kommen.

Name	Zweck	SNNS–Kern– Funktionsaufruf(e)
setInitFunc	Initialisierungsfunktion wählen	krui_setInitialisationFunc
setLearnFunc	Lernfunktion wählen	krui_setLearnFunc
setUpdateFunc	Updatefunktion wählen	krui_setUpdateFunc
setPruningFunc	Pruningfunktionen wählen	krui_setPrunFunc krui_setFFLearnFunc
setSubPattern	Subpattern shifting scheme definieren	—
setShuffle	Shuffle–Modus ändern	krui_shufflePatterns
setSubShuffle setParallelMode	Subpattern–Shuffle–Modus ändern Parallele Verarbeitung ein– oder ausschalten	krui_shuffleSubPatterns krui_setSpecialNetworkType
initNet	Netz initialisieren	krui_initializeNet
loadNet	Netz laden	krui_loadNet
saveNet	Netz speichern	krui_saveNet
saveResult	Ergebnisdatei speichern	krui_saveResultParam krui_DefTrainSubPat
trainNet	Netz trainieren	krui_learnAllPatterns krui_DefTrainSubPat
testNet	Netz testen	krui_testAllPatterns
pruneNet	Netz „prunen“	krui_learnAllPatternsFF pr_callPrunFunc pr_calcMeanDeviation krui_DefTrainSubPat
loadPattern	Musterdatei laden	krui_loadNewPatterns
setPattern	aktuelle Musterdatei setzen	kr_npui_setCurrPatSet
delPattern	Musterdatei löschen	kr_npui_deletePatSet

Anleitung zum Schreiben neuer Funktionen

Der Batchinterpreter kann leicht um neue Mantelfunktionen erweitert werden. Dies könnte nötig sein, um auch zukünftige Erweiterungen von SNNS in der Batchsprache bereitzustellen.

Der grundsätzliche Aufbau einer Mantelfunktion ist im Abschnitt „Befehle des Zwischencodes“ (Seite 41) schon beschrieben worden. Zur Erinnerung hier noch einmal der Aufbau einer Mantelfunktion in Kurzform:

1. Argumente der Mantelfunktion aus der Symboltabelle holen.
2. Typ und Anzahl der Argumente überprüfen.
3. SNNS–Kern–Funktionsaufruf(e) ausführen.

4. Symboltabelleneinträge der Systemvariablen aktualisieren, die beeinflußt werden sollen.

Folgende interne Funktionen des Batchinterpreters stehen für die Konstruktion von Mantelfunktionen zur Verfügung:

- `err_prt`: gibt einen Fehlertext aus und beendet den Batchinterpreter
- `warn_prt`: gibt eine Warnung aus
- `st_get_val_type`: holt Wert und Typ eines Symboltabelleneintrags aus der Symboltabelle
- `chck_type`: prüft den Typ eines Symboltabelleneintrags und ruft gegebenenfalls `err_prt` auf
- `kernel_error`: gibt eine Fehlermeldung des SNNS–Kerns im Format des Batchinterpreters aus; die betreffende Zeile des Batchprogramms, in welcher der fehlerhafte Funktionsaufruf steht, wird angegeben

Zum Einbinden einer neuen Mantelfunktion sind nun folgende Schritte notwendig:

1. Neue Mantelfunktion schreiben und in das Modul `ic_snns.c` kopieren.
2. Den Prototyp für die neue Mantelfunktion in die Header–Dateien `ic_snns.h` und `ic_snns.ph` eintragen.
3. Das Modul `snns_intfce.c` um eine Schnittstellenfunktion erweitern, welche die neue Mantelfunktion aufruft. Das Modul `snns_intfce` ist für zukünftige Erweiterungen des Batchinterpreters notwendig. Dabei bekommt die Schnittstellenfunktion den gleichen Namen wie die Mantelfunktion, allerdings wird vor den im Modul `ic_snns.c` verwendeten Funktionsnamen „`int_`“ geschrieben.
4. Den Prototyp für die neue Schnittstellenfunktion in die Header–Dateien `snns_intfce.h` und `snns_intfce.ph` eintragen.
5. Den Namen der neuen Mantelfunktion in die Symboltabelle eintragen. Die Symboltabelle befindet sich im Modul `symtab.c`. Der Code zur Eintragung des neuen Bezeichners wird in den Code der Initialisierungsfunktion `st_init` eingefügt. Die Eintragung erfolgt am besten mit folgendem Codefragment:

   ```
   tmp = st_insert("BatchprogFunctionName");
   val.fct_val = JacketFunctionName;
   st_set_val_type(tmp, FKT, val);
   st_set_ro(tmp);
   ```

   Hierbei ist „BatchprogFunctionName“ durch den Namen zu ersetzen, mit dem die neue Funktion im Batchprogramm aufgerufen werden soll. „JacketFunctionName“ ist durch den Namen der Schnittstellenfunktion zu ersetzen, wie er im Modul `snns_intfce.c` steht.
6. Das Makefile aufrufen.

6.8 Sonderfunktionen

Als Sonderfunktionen werden hier Funktionsaufrufe der Batchsprache bezeichnet, die entweder in der graphischen Benutzeroberfläche des SNNS nicht verfügbar sind oder keinen SNNS–Kern–Funktionsaufruf beinhalten.

Die `print`–Funktion

Die Funktion `print` hat die Aufgabe, alle ihre Parameter auszudrucken. Sie ist vom Typ Mantelfunktion. Bei jedem Argument wird der Typ festgestellt und eine entsprechende Druckanweisung ausgeführt. Alle Parameter werden in einer Zeile direkt hintereinander ausgegeben. Ist kein Parameter gegeben, so wird ein Zeilenwechsel vollzogen. Ist ein Parameter vom Typ UNKNOWN, wurde ihm also noch kein Wert zugewiesen, so wird das Symbol ⟨`undef`⟩ ausgegeben.

Die Funktion `execute`

Die Mantelfunktion `execute` dient dazu, Programme auf der UNIX–Kommandoebene auszuführen und deren Ausgabe gegebenenfalls ins Batchprogramm einzulesen. Da auf diese Weise unter anderem auch der gesamte Vorrat von UNIX–Hilfsprogrammen zur Verfügung steht, ist die Funktion `execute` sehr vielseitig. Denkbare Anwendungen sind zum Beispiel der Aufruf von Analyseprogrammen für SNNS–Ergebnisdateien, Abfrage von verfügbarem Plattenplatz, Formatierung oder Weiterverarbeitung der Ausgaben des Batchprogramms und vieles mehr.

Technisch ist der Aufruf der UNIX–Programme mittels der C–Routinen `popen` und `pclose` implementiert. `popen` öffnet einen Prozeß durch Starten einer sh–Shell. Das UNIX–Programm wird von dieser Shell ausgeführt. Die Routine `pclose` und damit der Batchinterpreter warten auf die Beendigung des UNIX–Programms.

Die Ausgaben des UNIX–Programms können vom Batchinterpreter eingelesen werden, sofern sie vom UNIX–Programm auf die Standardausgabe geschrieben werden. Zum Einlesen besitzt der Batchinterpreter einen zweiten Scanner. Die Ausgabe des UNIX–Programms sollte aus durch Leerzeichen getrennten Symbolen bestehen und von einem Zeilenendezeichen abgeschlossen sein. Der Scanner ordnet den Symbolen der Ausgabe nun, soweit möglich, einen der vier Typen STRING, INTEGER, REAL oder BOOL zu. Die Symbole werden sodann nach der Reihenfolge ihres Erscheinens auf der Ausgabezeile in Variablen des Batchprogramms gespeichert, die der `execute`–Funktion als Parameter übergeben worden sind. Es können so viele Symbole gespeichert werden, wie Variable angegeben wurden. Umgekehrt werden nur so viele Variable mit dem Wert von Symbolen belegt, wie Symbole auf der Ausgabezeile erscheinen.
Zusätzlich wird der Exit–Status des UNIX–Programms in der Systemvariablen EXIT_CODE gespeichert.

Die Funktion `exit`

Die Funktion `exit` beendet den Batchinterpreter sofort. Sie dient in erster Linie zum Verlassen des Batchprogramms. In zweiter Linie kann sie dazu verwendet werden, den Exit–Status des Batchinterpreters zu setzen. Wird die Funktion `exit` ohne Parameter aufgerufen, so wird ein Exit–Status von null angenommen. Ist ein ganzzahliges Argument als Parameter angegeben, so wird dieses als Exit–Status des Batchinterpreters verwendet. Der Exit–Status kann dazu dienen, anderen Programmen oder Shellscripts, die den Batchinterpreter aufrufen, Mitteilungen über den Programmablauf zu machen. Der Wertebereich des Exit–Status ist allerdings auf das Intervall $[-128; +127]$ begrenzt. Argumente, die außerhalb dieses Wertebereichs liegen, werden vom Betriebssystem auf den Wertebereich abgebildet. Der Batchinterpreter gibt hier eine Warnung aus.

6.9 Systemvariable

Der Batchinterpreter stellt dem Batchprogramm mehrere Systemvariable zur Verfügung. Diese beinhalten Informationen über den Ablauf des Trainings, die Anzahl der Muster in der aktuellen Musterdatei sowie den Exit–Status von auf der UNIX–Kommandoebene ausgeführten Programmen. Für das Batchprogramm erfolgt der Zugriff auf die Systemvariablen genauso wie auf normale Variable, Zuweisungen an Systemvariable führen allerdings zu einer Fehlermeldung. Es existieren folgende Systemvariable:

- SSE (summed squared error): Summe der quadrierten Abweichungen jedes Ausgabeneurons von seiner Soll–Ausgabe
- MSE (mean squared error): SSE geteilt durch die Anzahl der Muster, mit denen trainiert wird
- SSEPU (summed squared error per unit): SSE geteilt durch die Anzahl von Ausgabeneuronen des Netzes
- PAT (pattern): Anzahl der Muster in der aktuellen Musterdatei
- CYCLES: Anzahl der bisher trainierten Zyklen
- EXIT_CODE: Exit–Status von UNIX–Programmen

Welche Funktionen des Batchprogramms welche Systemvariable verändern, ist in der Benutzeranleitung (Kapitel 7) beschrieben.

6.10 Fehlerbehandlung

Die Erkennung von Fehlern im Batchprogramm ist eine wichtige Aufgabe des Batchinterpreters. Sie ist umfangreicher, als man auf den ersten Blick vermuten könnte: Müsste der Batchinterpreter nur korrekte Programme verarbeiten, könnte wahrscheinlich fast die Hälfte des Gesamtcodes eingespart werden. Die Fehlererkennung ist Bestandteil aller Phasen des Batchinterpreters. Analog zu den Phasen des Batchinterpreters kann zwischen syntaktischen und semantischen Fehlern sowie Laufzeitfehlern unterschieden werden. Ein Syntaxfehler wäre zum Beispiel das Weglassen des Schlüsselworts „do“ in einer for–Schleife; ein semantischer Fehler wäre eine Zuweisung mit einer undefinierten Variablen auf der rechten Seite. Ein Standardbeispiel für einen Laufzeitfehler ist ein Ausdruck, bei dem eine Division durch null ausgeführt werden soll. Weitere Fehler werden vom SNNS–Kern beim Ausführen der SNNS–Kern–Funktionsaufrufe erkannt, ferner werden Fehlermeldungen vom Betriebssystem übernommen, wenn Ein– oder Ausgabeoperationen oder Speicheranforderungen fehlschlagen.

In diesem Batchinterpreter wird bei Erkennen des ersten Fehlers im Batchprogramm die Verarbeitung der Eingabe beendet. Es wird eine Fehlermeldung ausgegeben, welche die Art des Fehlers sowie die Programmzeile des Batchprogramms angibt, in welcher der Fehler auftrat. Eine Liste aller möglichen Fehlermeldungen findet sich in Anhang D.

Bei Fehlern, die die weitere Ausführung des Batchprogramms nicht unbedingt verhindern, wird eine Warnung ausgegeben und die Verarbeitung der Eingabe fortgesetzt. Unter Umständen werden weggelassene Parameter oder ähnliches durch Standardwerte ergänzt. Die Programmiererin oder der Programmierer des Batchprogramms wird durch die Warnung davon in Kenntnis gesetzt, daß er oder sie vermutlich eine Anweisung oder einen Parameter vergessen haben.

Durch das beim Entwurf dieser Batchsprache verfolgte Konzept sind manche Fehlermeldungen, wie sie in Programmiersprachen wie C oder Pascal denkbar sind, nicht möglich. Beispielsweise kann ein „falsch“ geschriebener Variablenname nicht erkannt werden: da Variable in der Batchsprache vor ihrer Verwendung nicht deklariert werden müssen, ist jeder neu auftauchende Name (also auch ein falsch geschriebener) eine neue Variable. Umgekehrt kann auch nicht festgestellt werden, ob ein und dieselbe Variable fälschlicherweise für zwei oder mehr unterschiedliche Zwecke eingesetzt wird, da sie bei jeder Zuweisung einen neuen Typ annehmen kann.
Die Verantwortung für diese Fälle muß also der Programmiererin beziehungsweise dem Programmierer überlassen werden.

6.11 Dateioperationen

Der Batchinterpreter kann folgende Dateioperationen durchführen:

- Netzdatei laden
- Netzdatei speichern
- Musterdatei laden
- Ergebnisdatei schreiben
- Protokolldatei schreiben
- Batchprogramm aus Datei lesen

Die ersten vier Dateioperationen werden vom SNNS–Kern zur Verfügung gestellt. Wird kein Name für eine Protokolldatei angegeben, so werden Fehlermeldungen, Warnungen, Meldungen des Batchinterpreters und die Ausgaben des `print`–Befehls auf die Standardausgabe des Terminals geschrieben. Ist kein Dateiname für das Batchprogramm angegeben, so liest der Batchinterpreter das Programm von der Standardeingabe des Terminals. Die letzten zwei Dateioperationen werden mittels Kommandozeilenschaltern eingestellt (siehe auch Abschnitt 6.12).

6.12 Kommandozeilenschalter

Zur Steuerung des Batchinterpreters gibt es folgende Kommandozeilenschalter:

- f: Dateinamen des Batchprogramms angeben
- l: Dateinamen der Protokolldatei angeben
- p: Batchprogramm nur parsen (zur Fehlersuche), aber nicht ausführen
- q: Keine Meldungen ausgeben außer den Ausgaben der `print()`–Funktion
- s: Keine Warnungen ausgeben
- h: Hilfetext ausgeben

Die Schalter werden beim Aufruf des Batchinterpreters auf der UNIX–Kommandoebene eingegeben, zum Beispiel wie folgt:

```
batchman -f testprog -l batch.log
```

Dieser Befehl wird den Batchinterpreter starten, das Batchprogramm aus der Datei „testprog“ lesen und die Ausgabe in die Protokolldatei „batch.log“ schreiben.

Kapitel 7

Benutzeranleitung

Dieses Kapitel enthält die Benutzeranleitung für die Batchsprache. Das Schreiben der Benutzeranleitung ist Teil der Aufgabenstellung dieser Diplomarbeit. Die Benutzeranleitung wird später in das SNNS–Handbuch aufgenommen. Sie richtet sich an die Anwenderinnen und Anwender von SNNS, welche die Batchsprache zur Steuerung von SNNS einsetzen möchten. Die Benutzeranleitung ist im Gegensatz zum vorangegangenen Teil dieser Arbeit, der sich an den Informatiker wendet, so gehalten, daß sie auch für Anwender verständlich ist, die nicht über die Kenntnis einer höheren Programmiersprache verfügen.

7.1 Einführung

Dies ist die Benutzer– oder Programmieranleitung für die neue SNNS–Batchsprache, im folgenden auch kurz Batchsprache genannt. Die neue Batchsprache löst die bisherige Batchsteuerung von SNNS, das Programm SNNSBAT, ab. Programme, die für SNNSBAT geschrieben worden sind, können vom Batchinterpreter der neuen Batchsprache nicht ausgeführt werden. Die Batchsprache unterstützt alle Funktionen zum Trainieren und Testen neuronaler Netze, wie sie in der graphischen Benutzeroberfläche XGUI zur Verfügung stehen.

Diese Anleitung richtet sich an Anwenderinnen und Anwender von SNNS. Zum Erlernen der Batchsprache ist die Kenntnis einer höheren Programmiersprache zwar von Vorteil, aber nicht unbedingt erforderlich. Erfahrungen mit der graphischen Benutzeroberfläche von SNNS werden allerdings von großer Hilfe sein, obgleich die Batchsprache im Prinzip auch unabhängig von der Beherrschung der graphischen Benutzeroberfläche erlernt und eingesetzt werden kann. In bezug auf das Betriebssystem UNIX sollten allerdings Kenntnisse auf Anwenderebene wie zum Beispiel Starten von Programmen und Arbeiten mit Dateien vorliegen, um sinnvoll mit der Batchsteuerung arbeiten zu können.

Vorbilder der neuen Batchsprache sind die Sprachen AWK, Pascal, Modula–2 und C. Wer einer dieser Sprachen mächtig ist, wird sich in der Batchsprache schnell zurechtfinden. Der Sprachentwurf zielt bewußt darauf ab, die Programmiererin oder den Programmierer nicht mit syntaktischen Feinheiten und Spitzfindigkeiten von ihrem oder seinem eigentlichen Vorhaben abzulenken. So sind zum Beispiel für die meisten Operatoren mehrere Schreibweisen möglich; Variable brauchen vor ihrer ersten Verwendung nicht deklariert zu werden. Auf

Fehler im Batchprogramm wird unter Angabe der Zeilennummer und der vermutlichen Ursache mit einer aussagekräftigen Fehlermeldung oder Warnung hingewiesen.

Aufbau der Benutzeranleitung

Nach diesem Abschnitt, der außer der Einführung noch einige Hinweise zum Aufruf des Batchinterpreters enthält, folgt zunächst ein Abschnitt mit einer Einführung in die Batchsprache. Vorerst unabhängig von SNNS wird die Programmierung in der Batchsprache erläutert. Leserinnen und Leser, die bereits einer höheren Programmiersprache mächtig sind, können diesen Abschnitt überfliegen. Wichtig für sie sind nur das Konzept der undeklarierten Variablen, die vorhandenen Datentypen und die Namen der Systemvariablen. Der darauffolgende Abschnitt enhält die SNNS–Funktionsaufrufe mit ihren Parametern. Im vorletzten Abschnitt sind Sonderfunktionen beschrieben und es werden Anregungen für fortgeschrittene Anwender gegeben. Der letzte Abschnitt enthält einige Beispielprogramme.

Schreibweisen

Zunächst jedoch einige Hinweise zur verwendeten Schreibweise:

Eingaben, die auf der UNIX–Kommandoebene gemacht werden sollen oder Teil eines Batchprogramms sind, werden in Schreibmaschinenschrift gesetzt. Diese Eingaben sollten ohne Veränderung auf dem Anwendersystem eingegeben werden. Beispiel:

```
~/unix> batchman -h
```

Dies ist ein Befehl, der auf der UNIX–Kommandoebene eingegeben werden soll. Die Zeichen `~/unix>` deuten die UNIX–Eingabeaufforderung an; sie kann je nach Installation auch anders aussehen und beispielsweise das aktuelle Verzeichnis oder den Benutzernamen enthalten. Das Kommando `batchman -h` startet den Batchinterpreter mit dem Kommandozeilenschalter `-h`, der den Batchinterpreter veranlaßt, einen Hilfetext auszugeben. Alle Eingaben dieser Art sind mit der Eingabetaste (Enter, Return oder ähnliche) abzuschließen, was hier nicht extra dargestellt ist. Mögliche Ausgaben des Batchinterpreters sind in der gleichen Schriftart dargestellt und werden im Begleittext erläutert.

Batchprogramme oder Teile daraus werden ebenfalls in Schreibmaschinenschrift gesetzt, allerdings ohne die Eingabeaufforderung. Sie können entweder in der interaktiven Betriebsart des Batchinterpreters auf der Kommandoebene eingegeben werden oder mit einem Texteditor in einer Datei gespeichert werden. Der Name dieser Datei muß dann beim Starten des Batchinterpreters mit angegeben werden (siehe nachfolgenden Abschnitt).
Für Platzhalter in Batchprogrammen wird geneigte Schreibmaschinenschrift verwendet (*`so wie hier`*). Diese Platzhalter, zum Beispiel Namen von Variablen, müssen bei der Eingabe durch reale Namen ersetzt werden.

Aufruf des Batchinterpreters

Der Batchinterpreter kann auf der UNIX–Kommandoebene entweder interaktiv benutzt oder dazu veranlaßt werden, das Batchprogramm aus einer Datei zu lesen und keine weiteren Eingaben von der Tastatur entgegenzunehmen. Zum Starten des Batchinterpreters im interaktiven Modus wird einfach der Name des Batchinterpreters eingegeben:

```
~/unix> batchman
```

Daraufhin erfolgt die Ausgabe:

```
SNNS Batch Interpreter V1.0. Type batchman -h for help.
No input file specified, reading input from stdin.
batchman>
```

Nun können Befehle über die Tastatur eingegeben werden. `batchman>` ist die Eingabeaufforderung des Batchinterpreters. Ist die Eingabe beendet, so kann mit dem Dateiendezeichen (Ctrl–D) der Interpreter gestartet werden. Mit Ctrl–C kann der Batchinterpreter abgebrochen werden. Solange kein Dateiendezeichen eingegeben worden ist, werden die eingegebenen Befehle nicht ausgeführt.

Um die Eingabe, also das Batchprogramm, aus einer Datei zu lesen, muß ein Kommandozeilenschalter angegeben werden. Der Schalter für „Lesen aus einer Datei" ist `-f`. Hat man sein Programm unter dem Dateinamen `myprog.bat` abgespeichert, so kann der Batchinterpreter wie folgt gestartet werden:

```
~/unix> batchman -f myprog.bat
```

Auf diesen Befehl hin wird der Batchinterpreter das Programm, welches in der Datei `myprog.bat` gespeichert ist, ausführen und danach terminieren.

Ruft man den Batchinterpreter auf diese Weise auf, so werden alle Ausgaben des Batchinterpreters auf die Standardausgabe des Terminals, also normalerweise den Bildschirm, ausgegeben. Der Schalter `-l` bewirkt, daß die Ausgaben in eine Datei geschrieben werden (eine sogenannte Protokolldatei). Der Name der Datei muß dem Schalter, durch ein Leerzeichen getrennt, folgen:

```
~/unix> batchman -l logfile
```

Meistens wird man nur dann eine Protokolldatei angeben, wenn der Batchinterpreter die Eingabe auch aus einer Datei liest. Der Aufruf sieht dann folgendermaßen aus:

```
~/unix> batchman -f myprog.bat -l logfile
```

Die Reihenfolge, in der die Schalter angegeben werden, spielt keine Rolle. Sie müssen natürlich trotzdem von dem zugehörigen Dateinamen gefolgt sein.

Weitere Kommandozeilenschalter sind:

`-p`: Batchprogramm nur parsen (zur Fehlersuche), aber nicht ausführen. Die Syntax der Eingabe (Datei oder interaktiv) wird überprüft, die Befehle werden aber nicht ausgeführt. Fehler, die erst zur Laufzeit entstehen, können natürlich nicht erkannt werden; dies sind insbesondere auch fehlerhafte SNNS–Funktionsaufrufe.

`-q`: Keine Meldungen ausgeben außer den Ausgaben der `print()`–Funktion.

`-s`: Keine Warnungen ausgeben.

`-h`: Hilfetext ausgeben. Ein kurzer Text mit der Aufzählung der Schalter und ihrer Bedeutung wird ausgegeben.

In den folgenden Abschnitten ist es der Programmiererin und dem Programmierer freigestellt, die Eingaben in eine Datei zu schreiben oder sie im interaktiven Modus direkt einzugeben. Im Text werden alle Eingaben ohne die Eingabeaufforderung des Batchinterpreters abgedruckt.

7.2 Beschreibung der Batchsprache

In diesem Abschnitt wird zuerst der allgemeine Aufbau eines Batchprogramms erläutert. Danach werden die vorhandenen Datentypen eingeführt und die Benutzung von Variablen erklärt. Es wird die Funktion `print` vorgestellt und es wird die Benutzung von Operatoren in Ausdrücken veranschaulicht. Ein weiterer Abschnitt führt in die Verwendung von Kontrollstrukturen ein.

Aufbau eines Batchprogramms

Bezüglich des Aufbaus eines Batchprogramms bestehen keine Vorgaben. Es gibt keinen Deklarationsteil, die Befehle werden in der Reihenfolge, in der sie ausgeführt werden sollen, in die Programmdatei geschrieben. Zwischen den Befehlen können beliebig viele Leerzeichen und Leerzeilen eingestreut sein. Solange eindeutig ist, was gemeint ist, kann andererseits auf Leerzeichen auch weitgehend verzichtet werden.

Wie in der Sprache AWK brauchen Befehle, die allein in einer Zeile stehen, *nicht* mit einem Semikolon (`;`) abgeschlossen zu werden. Das Zeilenendezeichen wirkt hier als Trennzeichen zwischen zwei aufeinanderfolgenden Befehlen. Auch Befehle, die vor einem Schlüsselwort stehen, welches einen Anweisungsblock beendet (`endwhile`, `endif`, `endfor`, `until` und `else`), brauchen nicht mit einem Semikolon abgeschlossen zu werden. Es dürfen aber beliebig viele Semikola zwischen die Befehle gesetzt werden. Nur bei mehreren Befehlen pro Zeile ist das Semikolon als Trennzeichen unbedingt notwendig.

In das Batchprogramm können, wie in wohl allen Programmiersprachen, Kommentare eingefügt werden. Diese werden vom Batchinterpreter nicht als Befehle betrachtet. Kommentare dienen einerseits dazu, das Programm mit Erläuterungen zu versehen, andererseits sind sie oft ein willkommenes Mittel, einzelne Befehle vorübergehend „auszuschalten“ (in Programmdateien). Kommentare werden in der Batchsprache durch ein Doppelkreuz (`#`) eingeleitet. Der Rest der Zeile, in der das Doppelkreuz steht, wird ab dem Doppelkreuz einschließlich nicht mehr beachtet. Kommentare können zum Beispiel so aussehen:

```
# Dies ist ein Kommentar
a := 4    # Dies auch
```

Die zweite Zeile beginnt mit einer korrekten Zuweisung, die noch ausgeführt wird, und endet mit einem Kommentar.

Datentypen und Variable

Die Batchsprache kennt die folgenden Datentypen:

- Ganze Zahlen
- Fließkommazahlen
- Die Wahrheitswerte „TRUE“ und „FALSE“ (auch boolescher Typ genannt)
- Zeichenketten

Für die Darstellung von Fließkommazahlen kann eine Exponentialdarstellung wie in der Sprache C verwendet werden. Beispiele für gültige Fließkommazahlen sind: 0.42, 3.3, 1.0, .7, 3e3, 0.7E-12, 2123E123. Der Wert einer solchen Exponentialzahl ergibt sich aus der Mantisse (der Zahl vor dem kleinen oder großen e) multipliziert mit dem Exponenten, welcher als Potenz von 10 in diese Rechnung eingeht. So errechnet sich der Wert der obigen Zahl 3e3 mit $3 * 10^3$ zu 3000; der Wert von 0.7E-12 ist $0.7 * 10^{-12}$ und der Wert von 2123E123 ist $2123 * 10^{123}$.

Die Wahrheitswerte „TRUE“ und „FALSE“ werden genauso eingegeben, wie sie hier abgedruckt sind, also in Großbuchstaben. Zeichenketten müssen in Anführungsstriche oben (") eingeschlossen sein und dürfen alle Zeichen außer dem Tabulatorzeichen enthalten. Zeichenketten müssen mindestens ein Zeichen enthalten und dürfen nicht länger als eine Zeile sein. Zum Beispiel enthalten die ersten beiden der folgenden Zeilen gültige Zeichenketten:

```
"Dies ist eine Zeichenkette"
"# Ich bin auch eine Zeichenkette (0.7E-12)"
"Aber das hier
               ist keine."
```

Variable

Zur Speicherung von Werten können in der Batchsprache Variable verwendet werden. Eine Variable wird dem Interpreter einfach durch ihre erste Verwendung bekanntgemacht, sie braucht nicht vorher deklariert zu werden. Als Namen für Variable kommen alle Zeichenketten in Betracht, die mit einem Buchstaben oder einem Unterstrich (_) beginnen und dann aus Buchstaben, Ziffern oder weiteren Unterstrichen bestehen, wie beispielsweise:

`a`, `num1`, `_test`, `first_net_name`, `error_flag`, `k17_u`, `Test_Buffer_1`

Zwischen Groß– und Kleinschreibung wird unterschieden: die Namen `alpha` und `Alpha` beispielsweise bezeichnen verschiedene Variable. Es können alle Namen verwendet werden, die nicht vom Batchinterpreter selber verwendet werden, wie zum Beispiel „TRUE“ und „FALSE“, die Namen von Funktionen oder Schlüsselworte wie „else“ oder „for“. Ferner können die Namen der Systemvariablen, die weiter unten beschrieben werden, vom Programmierer nicht für andere Zwecke als für die, für welche sie vorgesehen sind, verwendet werden.

Da die Variablen ohne vorherige Deklaration verwendet werden, ist der Wert und der Typ einer Variablen solange undefiniert, bis ihr ein Wert zugewiesen wird. Bei jeder Zuweisung nimmt sie den Typ und den Wert an, der sich auf der rechten Seite der Zuweisung ergibt. Typ und Wert werden bis zur nächsten Zuweisung beibehalten. Folgende Beispiele sind Zuweisungen eines Werts (und des dazugehörigen Typs) an eine Variable (es wird jeweils der rechts des Zuweisungsoperators stehende Wert der Variablen links des Zuweisungsoperators zugewiesen):

```
a = 5
filename := "first.net"
init_flag := TRUE
NET_ERR = 4.7e+11
a := init_flag
```

Als Zuweisungsoperator kann sowohl das Zeichen `=` als auch die Zeichenfolge `:=` verwendet werden. Die Variable a in der ersten Zeile hat nach der Zuweisung den Typ ganze Zahl. Nach der Zuweisung in der fünften Zeile nimmt sie Wert und Typ der Variablen `init_flag` an, also ist sie dann von booleschem Typ. Die Variable `filename` ist vom Typ Zeichenkette und `NET_ERR` vom Typ Fließkommazahl.

Operatoren und Ausdrücke

Als „Ausdruck“ wird allgemein eine Rechenvorschrift bezeichnet, die zu einem einzigen Wert ausgewertet werden kann. Ein Ausdruck kann eine einzelne Zahl, aber auch eine komplizierte mathematische Formel sein. Beispiele für Ausdrücke sind:

```
3
TRUE
3 + 6
17 - 4 * a + (2 * ln 5) / 0.3
```

Der Wert oder das Ergebnis eines Ausdrucks kann einer Variablen zugewiesen werden, in einem Funktionsaufruf als Argument oder aber direkt bei der Auswertung von Bedingungen verwendet werden.

In Ausdrücken, die sich aus mehr als einem Einzelwert zusammensetzen, werden Operatoren zur Verknüpfung der Einzelwerte benötigt. Einige Operatoren sind schon in obigem Beispiel zu sehen. Insgesamt stellt die Batchsprache folgende Operatoren zur Verfügung:

Operator	Funktion
+, −	Vorzeichen für Zahlen
not, !	logische Negation für Wahrheitswerte
sqrt	Quadratwurzel
ln	natürlicher Logarithmus zur Basis e
log	Logarithmus zur Basis 10
**, ^	Exponentialfunktion
*	Multiplikation
/	Division
div	Ganzzahl-Division mit ganzzahligem Ergebnis
mod, %	Restwert bei der Ganzzahl-Division
+	Addition
−	Subtraktion
<	kleiner als
<=, =<	kleiner gleich
>	größer als
>=, =>	größer gleich
==	gleich
<>, ! =	ungleich
and, &&	logisches UND für Wahrheitswerte
or, ||	logisches ODER für Wahrheitswerte

Die Reihenfolge, in der die einzelnen Teilausdrücke größerer Ausdrücke ausgewertet werden, ist folgende:

1. Die Teilausdrücke mit Operatoren, die in der Tabelle weiter oben stehen, werden *vor* Teilausdrücken ausgewertet, deren Operatoren weiter unten in der Tabelle stehen. Manche Operatoren haben die gleiche Auswertungspriorität, unter anderem + und −. So ist zum Beispiel Punkt– vor Strichrechnung gewährleistet.
2. Stehen mehrere gleiche Operatoren in einem Ausdruck, so wird *von links nach rechts* ausgewertet.

Die Reihenfolge der Auswertung kann durch runde Klammern (()) geändert werden. Teilausdrücke, die in runden Klammern stehen, werden komplett ausgewertet, bevor ihr Ergebnis nach der normalen Reihenfolge weiterverwendet wird.

Die meisten Operatoren arbeiten auf verschiedenen Argumenttypen. Der Typ eines Ausdrucks (also der Typ des Ergebnisses einer Operation) wird automatisch festgelegt und ist, außer bei der Ganzzahl–Division, der Modulo–Operation, den booleschen Operatoren und den Vergleichsoperatoren, nicht durch den Operator vorbestimmt.

So ergibt zum Beispiel die Multiplikation zweier ganzer Zahlen den Typ Ganzzahl. Die Multiplikation einer ganzen Zahl und einer Fließkommazahl hingegen ergibt als Ergebnis den Typ Fließkommazahl. Die ganze Zahl wird hierzu automatisch in eine Fließkommazahl

umgewandelt.
Ist mindestens ein Operand einer Addition eine Zeichenkette, so werden alle beteiligten Operanden in Zeichenketten umgewandelt, und es wird eine Verkettung aller Operanden zu einer resultierenden Zeichenkette durchgeführt. Beispielsweise hat die Variable `a` nach der Zuweisung:

```
a := 5 + " plus " + 4 + " ist " + (8 + 1)
```

den Wert `5 plus 4 ist 9` und ist vom Typ Zeichenkette.

Zu beachten ist, daß in der Batchsprache die mathematischen Funktionen Quadratwurzel, natürlicher Logarithmus zur Basis e, Logarithmus zur Basis 10 und die Exponentialfunktion als Operatoren betrachtet werden und nicht, wie oft, als Funktionsaufrufe. Der Unterschied für den Programmierer ist nur, daß nach dem Operator *keine* runden Klammern gesetzt zu werden brauchen. Die genannten mathematischen Funktionen werden also folgendermaßen aufgerufen:

Quadratwurzel:	`sqrt 9`
natürlicher Logarithmus:	`ln 2`
Logarithmus zur Basis 10:	`log alpha`
Exponentialfunktion:	`10 ** 4` oder `a^b`

Klammern dürfen natürlich trotzdem verwendet werden; sie sind sogar erforderlich, wenn der Operator auf einen Teilausdruck angewandt werden soll:

```
sqrt (9 + 16)
ln (2^16)
log (alpha * sqrt _tau)
```

Die Funktion `print`

Mit dem bisher Gelernten können nun zwar Variable mit Werten belegt und Ausdrücke formuliert werden, die Ergebnisse bleiben aber ein Geheimnis des Batchinterpreters. Um den Wert von Ausdrücken, Variablen oder einfach von Konstanten wie Zahlen und Zeichenketten darzustellen, gibt es daher die Funktion `print`. Sie ist ein „echter“ Funktionsaufruf des Batchinterpreters und druckt ihre Argumente hintereinander in die Ausgabedatei (falls keine angegeben ist, wird auf die Standardausgabe des Terminals gedruckt). Es können beliebig viele Argumente beim Aufruf der `print`–Funktion angegeben werden. Ist kein Argument angegeben, so wird lediglich ein Zeilenwechsel vorgenommen (die `print`–Funktion nimmt bei *jedem* Aufruf einen Zeilenwechsel vor). Dadurch, daß beliebig viele Argumente beliebigen Typs angegeben werden können, ist die `print`–Funktion recht vielseitig, wie folgende Beispiele verdeutlichen sollen:

Befehl:	erzeugt die Ausgabe:
`print(5)`	`5`
`print(3*4)`	`12`
`print("Dies ist ein Text")`	`Dies ist ein Text`
`print("Dies sind Text und Zahlen:",1,2,3)`	`Dies sind Text und Zahlen:123`
`print("Oder so: ",1," ",2," ",3)`	`Oder so: 1 2 3`
`print(ln (2^16))`	`11.0904`
`print(FALSE)`	`FALSE`
`print(25e-2)`	`0.25`

Da es vorkommen kann, daß in einem `print`–Befehl Variable gedruckt werden sollen, die noch nicht mit einem Wert belegt sind, wird in diesem Fall der Wert `<undef>` ausgegeben.

Kontrollstrukturen

Kontrollstrukturen sind ein wesentliches Merkmal von Programmiersprachen. Sie ermöglichen die wiederholte Ausführung von einem oder mehreren Befehlen, abhängig von einer Bedingung oder von einer vorgegebenen Anzahl. Ferner sind Verzweigungen des Programms mit bedingten Anweisungen möglich. Hätte man keine Kontrollstrukturen zur Verfügung, so würden immer alle Befehle des Batchprogramms nacheinander abgearbeitet, Wiederholungen oder Schleifen wären nicht möglich. In der Batchsprache stehen folgende Kontrollstrukturen zur Verfügung:

```
if Ausdruck then Block endif
if Ausdruck then Block else Block endif

for Zuweisung to Ausdruck do Block endfor
while Ausdruck do Block endwhile
repeat Block until Ausdruck
```

Die kursiv geschriebenen Bestandteile der Kontrollstrukturen sind durch entsprechende Befehle zu ersetzen, also *`Ausdruck`* durch einen Ausdruck, wie oben beschrieben, *`Zuweisung`* durch eine Zuweisung und *`Block`* durch eine Folge von Befehlen, die selber auch Kontrollstrukturen enthalten darf. Die einzelnen Kontrollstrukturen werden im folgenden genauer erläutert.

Die `if`–Anweisung

Die `if`–Anweisung existiert in zwei Varianten. Die erste ist eine einfache bedingte Anweisung:

```
if Ausdruck then Block endif
```

Der *`Block`* wird hier genau dann ausgeführt, wenn der *`Ausdruck`* ein Wahrheitswert ist *und* als Ergebnis den Wert „TRUE“ hat. So erzeugt die Anweisung:

```
if 3<5 then print("3 ist kleiner als 5!") endif
```

die Ausgabe:

```
3 ist kleiner als 5!
```

Hingegen erzeugt die Anweisung:

```
if 5<5 then print("3 ist kleiner als 5!") endif
```

keine Ausgabe, da der `print`–Befehl nicht ausgeführt wird.

Für ***Ausdruck*** können beliebig komplizierte Ausdrücke stehen, immer unter der Voraussetzung, daß das Ergebnis ein Wahrheitswert ist:

```
if sqrt 9 - 5 < 0 and TRUE <> FALSE then print("hello world") endif
```

erzeugt die Ausgabe:

```
hello world
```

Man beachte hier, daß der logische Verknüpfungsoperator „and“ als letztes ausgewertet wird, da er die niedrigste Priorität bei der Auswertung besitzt. Wenn man sich über die genaue Auswertungsreihenfolge unsicher ist, sollten runde Klammern um die Teilausdrücke gesetzt werden.

Die zweite `if`–Anweisung besitzt einen zweiten Block, der alternativ zum ersten ausgeführt wird. Der Aufbau der zweiten `if`–Anweisung ist also:

```
if Ausdruck then Block1 else Block2 endif
```

Der erste Block, hier als ***Block1*** bezeichnet, wird wie bei der oben beschriebenen `if`–Anweisung ausgeführt, wenn der Wert des Ausdrucks ***Ausdruck*** „TRUE“ ist. Der zweite Block ***Block2*** wird jedoch ausgeführt, wenn der Wert von ***Ausdruck*** „FALSE“ ist. So können Anweisungen der folgenden Art formuliert werden:

```
if a<3 then
  print("a ist kleiner als 3")
else
  print("a ist nicht kleiner als 3")
endif
```

Die Anweisung erzeugt die Ausgabe:

```
a ist kleiner als 3
```

wenn der Wert der Variablen a kleiner als 3 ist oder

```
a ist nicht kleiner als 3
```

wenn nicht. Der Variablen a muß natürlich vor der `if`–Anweisung ein Wert zugewiesen werden.

Die `for`–Schleife

Die `for`–Schleife ist eine Kontrollstruktur, die einen Programmblock eine vorgegebene Anzahl von Durchläufen ausführt. Die allgemeine Form der `for`–Schleife sieht so aus:

```
for Zuweisung to Ausdruck do Block endfor
```

Für die `for`–Schleife wird ein sogenannter Schleifenzähler benötigt, das ist eine Variable, welche die Durchläufe zählt. Sie wird in der *`Zuweisung`* mit einem Anfangswert belegt. Der Wert dieses Schleifenzählers wird nach jedem Durchlauf des Blocks um eins erhöht. Übersteigt der Wert des Schleifenzählers den Wert von *`Ausdruck`*, so wird der Block nicht mehr durchlaufen und die `for`–Schleife wird verlassen. Ist der Wert des Schleifenzählers schon von Beginn an größer als der Wert von *`Ausdruck`*, so wird der Block gar nicht durchlaufen. Beim Schleifenzähler handelt es sich um eine gewöhnliche Variable, die auch in der `for`–Schleife verwendet werden darf. Eine einfache `for`–Schleife wäre zum Beispiel:

```
for i := 2 to 5 do print("here we are: ",i) endfor
```

Sie erzeugt die Ausgabe:

```
here we are: 2
here we are: 3
here we are: 4
here we are: 5
```

Man kann die Anzahl der Schleifendurchläufe beeinflussen, indem man entweder im *`Block`* den Wert des Schleifenzählers mittels einer Zuweisung verändert oder eine der Anweisungen `break` und `continue` verwendet. Die Anweisung `break` verläßt die `for`–Schleife sofort, hingegen wird bei der `continue`–Anweisung die Ausführung des Blocks abgebrochen und ein Sprung zum *`Ausdruck`* durchgeführt, wobei auch der Wert des Schleifenzählers um eins erhöht wird. Nun wird wieder geprüft, ob der Wert des Schleifenzählers den Wert von *`Ausdruck`* übersteigt: wenn ja, wird die Schleife verlassen. In der Praxis kann das zum Beispiel so aussehen:

```
for zaehler := 1 to 200 do
  a := a * zaehler
  c := c + 1
  if test == TRUE then break endif
endfor
```

Hier wird die boolesche Variable `test` dazu verwendet, die `for`–Schleife gegebenenfalls vorzeitig zu beenden. Hat `test` den Wert „TRUE“, so wird der Schleifenblock nur ein einziges Mal durchlaufen.

Die `while`– und die `repeat`–Schleife

Außer der `for`–Schleife stehen noch zwei weitere Schleifen zur Verfügung, nämlich die `while`– und die `repeat`–Schleife. Sie besitzen im Gegensatz zur `for`–Schleife keinen Schleifenzähler, sondern werden immer wieder durchlaufen, *solange* eine Bedingung erfüllt ist (`while`–Schleife) beziehungsweise *bis* eine Bedingung erfüllt ist (`repeat`–Schleife). Die Bedingung muß, wie bei der `if`–Anweisung, ein Ausdruck sein, der einen Wahrheitswert als Ergebnis hat. Die Formate der `while`– und der `repeat`–Schleife sind:

```
while Ausdruck do Block endwhile
repeat Block until Ausdruck
```

Im Block der beiden Schleifen muß also jeweils dafür gesorgt werden, daß die Schleifenbedingung `Ausdruck` nach endlicher Zeit einmal den Wert „FALSE“ (im Fall der `while`–Schleife) beziehungsweise den Wert „TRUE“ (im Fall der `repeat`–Schleife) annimmt, da die Schleifenblöcke `Block` sonst endlos durchlaufen werden. Die `for`–Schleife aus dem vorhergehenden Abschnitt könnte also durch eine `while`–Schleife wie folgt nachgebildet werden:

```
i := 2
while i <= 5 do
  print("here we are: ",i)
  i := i + 1
endwhile
```

Zu beachten ist hier, daß die Variable `i`, die hier als Schleifenzähler verwendet wird, *vor* der `while`–Schleife auf den Wert 2 gesetzt wird. Der wesentliche Unterschied zwischen der `while`– und der `repeat`–Schleife ist, daß die `repeat`–Schleife mindestens einmal durchlaufen wird, bevor es zum Prüfen der Schleifenbedingung kommt. So sähe das obige Beispiel als `repeat`–Schleife aus:

```
i := 2
repeat
  print("here we are: ",i)
  i := i + 1
until i > 5
```

Man beachte die unterschiedliche Formulierung der Schleifenbedingungen.
Auch in den Schleifenblöcken der `while`– und der `repeat`–Schleife sind die Anweisungen `break` und `continue` zulässig. Wieder bewirkt eine `break`–Anweisung, daß die Schleife sofort verlassen wird, eine `continue`–Anweisung bewirkt einen Sprung zur Schleifenbedingung, worauf diese dann ausgewertet wird.

7.3 SNNS–Funktionsaufrufe

Nachdem nun in die Programmierung in der Batchsprache eingeführt worden ist, werden jetzt die SNNS–Funktionsaufrufe beschrieben, mit denen der Simulatorkern gesteuert wird. Die SNNS–Funktionsaufrufe lassen sich grob in drei Gruppen einteilen:

1. Funktionsaufrufe, mit denen Parameter eingestellt werden
2. Funktionsaufrufe, die sich auf Netze beziehen
3. Funktionsaufrufe, die sich auf Muster beziehen

Das Format von Funktionsaufrufen lautet:

```
Funktionsname (Parameter1,Parameter2 ...)
```

Nach dem Funktionsnamen kann kein Parameter, ein Parameter, oder es können, durch Kommata getrennt, mehrere Parameter stehen. Die runden Klammern müssen in jedem

Fall angegeben werden.

Zunächst werden die Funktionsaufrufe der ersten Gruppe, die zur Einstellung von Parametern dienen, erläutert.

Funktionsaufrufe zur Einstellung von Parametern

Folgende Funktionsaufrufe stehen zur Einstellung von Parametern zur Verfügung:

`setInitFunc()`	Initialisierungsfunktion und ihre Parameter wählen
`setLearnFunc()`	Lernfunktion und ihre Parameter wählen
`setUpdateFunc()`	Updatefunktion und ihre Parameter wählen
`setPruningFunc()`	Pruningfunktionen und ihre Parameter wählen
`setSubPattern()`	Subpattern shifting scheme definieren
`setShuffle()`	Shuffle–Modus ändern
`setSubShuffle()`	Subpattern–Shuffle–Modus ändern
`setParallelMode()`	Parallele Verarbeitung ein– oder ausschalten

Das Format der Funktionsaufrufe und ihre Verwendung soll nun im einzelnen erläutert werden. Erfahrungen mit der graphischen Benutzeroberfläche von SNNS sind für das Erlernen der Funktionsaufrufe von großem Vorteil. Deshalb soll die Leserin oder der Leser ermuntert werden, sich durch Benutzen der graphischen Oberfläche einen Überblick über den Ablauf des Trainings neuronaler Netze mit SNNS zu verschaffen. Dringend zu empfehlen ist die Lektüre der folgenden Kapitel im SNNS–Handbuch: Kapitel 4.3 (Parameters of the learning functions), 4.4 (Update functions), 4.5 (Initialisation functions), 5 (Handling patterns with SNNS), 9 (Neural network models and functions) sowie bei Bedarf Kapitel 10 (Pruning algorithms).

Der Funktionsaufruf `setInitFunc()`

Dieser Funktionsaufruf wählt die Funktion, mit der das Netz initialisiert wird. Das Format des Funktionsaufrufs ist:

```
setInitFunc (Funktionsname, Parameter ...)
```

`Funktionsname` ist der Name der Initialisierungsfunktion. Er muß unbedingt angegeben werden und genauso geschrieben sein, wie unten angegeben, allerdings in Anführungsstriche oben eingeschlossen werden. Zur Auswahl stehen:

`ART1_Weights`	`DLVQ_Weights`	`Random_Weights_Perc`
`ART2_Weights`	`Hebb`	`Randomize_Weights`
`ARTMAP_Weights`	`Hebb_Fixed_Act`	`RBF_Weights`
`CC_Weights`	`JE_Weights`	`RBF_Weights_Kohonen`
`ClippHebb`	`Kohonen_Rand_Pat`	`RBF_Weights_Redo`
`CPN_Weights_v3.2`	`Kohonen_Weights_v3.2`	`RCC_Weights`
`CPN_Weights_v3.3`	`Kohonen_Const`	`RM_Random_Weights`
`CPN_Rand_Pat`	`PseudoInv`	

Nach dem Namen der Initialisierungsfunktion können Parameter für die Initialisierungsfunktion angegeben werden. Werden keine Parameter angegeben, so werden Standardwerte angenommen. Die Parameter müssen vom Datentyp „Fließkommazahl“ sein. Eine genaue Beschreibung der Initialisierungsfunktionen und ihrer Parameter findet sich in Kapitel 4.5 (Initialisation functions) des SNNS–Handbuchs. Beispiele für den Aufruf von `setInitFunc()` sind:

```
setInitFunc ("Randomize_Weights")
setInitFunc ("Randomize_Weights", 1.0, -1.0)
```

Der erste Aufruf legt nur die Initialisierungsfunktion auf „Randomize_Weights“ fest, der zweite Aufruf gibt auch noch zwei Parameter mit an. Der Batchinterpreter gibt darauf den folgenden Text aus:

```
Init function is now Randomize_Weights
Parameters are: 1.0 -1.0
```

In der zweiten Zeile werden die angegebenen Parameter der Initialisierungsfunktion ausgegeben.

Der Funktionsaufruf `setLearnFunc()`

Der Funktionsaufruf `setLearnFunc()` ist dem Funktionsaufruf `setInitFunc()` recht ähnlich. `setLearnFunc()` wählt die Lernfunktion, mit der das Netz trainiert wird. Das Format des Funktionsaufrufs ist:

`setLearnFunc (`*`Funktionsname, Parameter`*` ...)`

`Funktionsname` ist der Name der Lernfunktion. Er muß unbedingt angegeben werden und genauso geschrieben sein, wie unten angegeben und in Anführungsstriche oben eingeschlossen sein. Zur Auswahl stehen:

```
ART1                   Dynamic_LVQ           RadialBasisLearning
ART2                   Hebbian               RBF-DDA
ARTMAP                 JE_BP                 RCC
BackPercolation        JE_BP_Momentum        RM_delta
BackpropBatch          JE_Quickprop          Rprop
BackpropMomentum       JE_Rprop              Sim_Ann_SS
BackpropWeightDecay    Kohonen               Sim_Ann_WTA
BPTT                   Monte-Carlo           Sim_Ann_WWTA
BBPTT                  PruningFeedForward    Std_Backpropagation
CC                     QPTT                  TimeDelayBackprop
Counterpropagation     Quickprop
```

Nach dem Namen der Lernfunktion können Parameter für die Lernfunktion angegeben werden. Werden keine Parameter angegeben, so werden Standardwerte angenommen. Die Parameter müssen vom Datentyp „Fließkommazahl“ sein. Eine genaue Beschreibung der Lernfunktionen und ihrer Parameter findet sich in Kapitel 4.3 (Parameters of the learning functions) des SNNS–Handbuchs. Beispiele für den Aufruf von `setLearnFunc()` sind:

```
setLearnFunc ("Std_Backpropagation")
setLearnFunc ("Std_Backpropagation", 0.1)
```

Der erste Aufruf legt nur die Lernfunktion auf „Std_Backpropagation" fest, der zweite Aufruf gibt noch einen Lernparameter mit an. Der Batchinterpreter gibt darauf den folgenden Text aus:

```
Learning function is now Std_Backpropagation
Parameters are: 0.1
```

Der Funktionsaufruf `setUpdateFunc()`

Auch der Funktionsaufruf `setUpdateFunc()` ist dem Funktionsaufruf `setInitFunc()` recht ähnlich. `setUpdateFunc()` wählt die Funktion, nach der die Erregung der Neuronen durch das Netz propagiert wird. Das Format des Funktionsaufrufs ist:

setUpdateFunc (*Funktionsname*, *Parameter* ...)

Funktionsname ist der Name der Updatefunktion. Er muß unbedingt angegeben werden und genauso geschrieben sein, wie unten angegeben und in Anführungsstriche oben eingeschlossen sein. Zur Auswahl stehen folgende Updatefunktionen:

Topological_Order	BAM_Order	JE_Special
ART1_Stable	BPTT_Order	Kohonen_Order
ART1_Synchronous	CC_Order	Random_Order
ART2_Stable	CounterPropagation	Random_Permutation
ART2_Synchronous	Dynamic_LVQ	RCC_Order
ARTMAP_Stable	Hopfield_Fixed_Act	Serial_Order
ARTMAP_Synchronous	Hopfield_Synchronous	Synchonous_Order
Auto_Synchronous	JE_Order	TimeDelay_Order

Nach dem Namen der Updatefunktion können Parameter für die Updatefunktion angegeben werden. Werden keine Parameter angegeben, so werden Standardwerte angenommen. Die Parameter müssen vom Datentyp „Fließkommazahl" sein. Eine genaue Beschreibung der Updatefunktionen und ihrer Parameter findet sich in Kapitel 4.4 (Update functions) des SNNS–Handbuchs. Ein Beispiel für den Aufruf von `setUpdateFunc()` ist:

```
setUpdateFunc ("Topological_Order")
```

Der Aufruf legt die Updatefunktion auf „Topological_Order" fest. Diese Updatefunktion erwartet keine Parameter. Der Batchinterpreter gibt darauf den folgenden Text aus:

```
Update function is now Topological_Order
```

Der Funktionsaufruf `setPruningFunc()`

Mit dem Funktionsaufruf `setPruningFunc()` werden die Funktionen für das Pruning von neuronalen Netzen gewählt (siehe auch Kapitel 10 des SNNS–Handbuchs: Pruning Algorithms und [Schreiner 94]). Das Format des Funktionsaufrufs ist:

setPruningFunc (*Funktionsname1*, *Funktionsname2*, *Parameter* ...)

Funktionsname1 ist der Name der Pruningfunktion. Er muß unbedingt angegeben werden. Zur Auswahl stehen folgende Pruningfunktionen:

```
MagPruning                OptimalBrainSurgeon
Noncontributing_Units     Skeletonization
OptimalBrainDamage
```

Funktionsname2 ist der Name der untergeordneten Lernfunktion. Er muß unbedingt angegeben werden. Zur Auswahl stehen folgende Lernfunktionen:

```
BackpropBatch           Quickprop
BackpropMomentum        Rprop
BackpropWeightDecay     Std_Backpropagation
```

Die folgenden Parameter können nach Bedarf angegeben werden. Die Parameter, die nicht angegeben sind, werden durch die Standardwerte ersetzt, die im Pruning–Fenster der graphischen Benutzeroberfläche erscheinen. Allerdings müssen, soll beispielsweise der dritte Parameter geändert werden, auch die beiden ersten Parameter in den Funktionsaufruf eingetragen werden. Die den Funktionsnamen folgenden Parameter haben dieselbe Reihenfolge wie die Parameter im Pruning–Fenster der graphischen Benutzeroberfläche (der erwartete Datentyp steht jeweils in Klammern):

1. Maximum error increase in % (Fließkommazahl)
2. Accepted error (Fließkommazahl)
3. Recreate last pruned element (Wahrheitswert)
4. Learn cycles for first training (ganze Zahl)
5. Learn cycles for retraining (ganze Zahl)
6. Minimum error to stop (Fließkommazahl)
7. Initial value for matrix (Fließkommazahl)
8. Input pruning (Wahrheitswert)
9. Output pruning (Wahrheitswert)

Beispiele für den Aufruf von `setPruningFunc()` sind:

```
setPruningFunc ("OptimalBrainDamage", "Std_Backpropagation")
setPruningFunc ("MagPruning","Rprop",15.0,3.5,FALSE,500,100,1.0,1e-6)
```

Der erste Aufruf legt nur die Pruningfunktion und die untergeordnete Lernfunktion fest, der zweite Aufruf hingegen gibt fast alle Parameter mit an. Zu beachten ist, daß ein Funktionsaufruf keinen Zeilenwechsel enthalten darf. Auch lange Funktionsaufrufe mit vielen Parametern müssen also in eine Zeile geschrieben werden. Der Batchinterpreter gibt für den zweiten Funktionsaufruf den folgenden Text aus:

```
Pruning function is now MagPruning
Subordinate learning function is now Rprop
Parameters are: 15.0 3.5 FALSE 500 100 1.0 1e-6 TRUE TRUE
```

Für das Pruning muß als „normale" Lernfunktion `PruningFeedForward` angegeben werden (mittels des `setLearnFunc()`–Funktionsaufrufs), falls diese Lernfunktion nicht schon in der Netzdatei gespeichert ist.

Der Funktionsaufruf `setSubPattern()`

Der Funktionsaufruf `setSubPattern()` definiert das „Subpattern–Shifting–Scheme“, wie es in Kapitel 5 (Handling patterns with SNNS), Abschnitt 3 (Variable size patterns) des SNNS–Handbuchs beschrieben ist. Die Definition des Subpattern–Shifting–Scheme muß zu der verwendeten Musterdatei und der Architektur des Netzes passen. Das Format des Funktionsaufrufs ist:

`setSubPattern (`*`InputSize1, InputStep1, OutputSize1, OutputStep1`* `...)`

Die vier Parameter *`InputSize1`*, *`InputStep1`*, *`OutputSize1`* und *`OutputStep1`* beschreiben die erste Dimension des Subpatterns. Sie haben dieselbe Reihenfolge wie im Fenster „Sub Pattern Handling“ der graphischen Benutzeroberfläche des SNNS. Es müssen immer alle vier Parameter einer Dimension angegeben werden. Gibt es eine zweite Dimension, so werden die vier Parameter *`InputSize2`*, *`InputStep2`*, *`OutputSize2`* und *`OutputStep2`* hinter die vier Parameter der ersten Dimension geschrieben und so fort für jede weitere Dimension.

Beispiele für den Funktionsaufruf `setSubPattern()` sind:

```
setSubPattern (3, 5, 1, 5)
setSubPattern (3, 5, 1, 5, 3, 5, 1, 5)
```

Im ersten Funktionsaufruf wird ein eindimensionales Subpattern definiert mit InputSize 3, InputStep 5, OutputSize 1 und OutputStep 5. Im zweiten Funktionsaufruf wird ein zweidimensionales Subpattern für das mitgelieferte Netz „watch.net“ definiert. Der Batchinterpreter gibt den Text:

```
Subpattern shifting scheme (re)defined
Parameters are: 3 5 1 5 3 5 1 5
```

aus. Die Parameter für den Funktionsaufruf `setSubPattern()` müssen immer ganze Zahlen sein.

Die Funktionsaufrufe `setShuffle()` und `setSubShuffle()`

Die beiden Funktionsaufrufe `setShuffle()` und `setSubShuffle()` dienen dazu, die Shuffle–Betriebsart des SNNS (zufällige Wahl des nächsten Trainingsmusters) ein– oder auszuschalten. Die Formate der Funktionsaufrufe sind:

`setShuffle (`*`Modus`*`)`
`setSubShuffle (`*`Modus`*`)`

Hierbei ist der Parameter *`Modus`* ein Wahrheitswert. Hat *`Modus`* den Wert TRUE, so arbeitet SNNS in der Shuffle–Betriebsart, hat *`Modus`* den Wert FALSE, so arbeitet SNNS nicht in der Shuffle–Betriebsart. `setShuffle()` bezieht sich auf „normale“ Muster und `setSubShuffle` auf Subpattern. Auf den Funktionsaufruf:

```
setSubShuffle(TRUE)
```

erfolgt die Meldung:

```
Subpattern shuffling enabled
```

Der Funktionsaufruf `setShuffle()` wird analog eingesetzt.

Der Funktionsaufruf `setParallelMode()`

Der Funktionsaufruf `setParallelMode()` schaltet zwischen dem seriellen und dem parallelen Simulatorkern hin und her. Dies funktioniert nur, wenn ein Parallelrechner vorhanden ist *und* Batchinterpreter und Simulatorkern für diesen Parallelrechner übersetzt worden sind. Der Funktionsaufruf hat das Format:

```
setParallelMode (Modus)
```

Der Parameter ***Modus*** ist hier, wie bei den `setShuffle()`–Funktionsaufrufen, ein Wahrheitswert. Hat ***Modus*** den Wert „TRUE", so werden die folgenden Anweisungen auf dem parallelen Simulatorkern ausgeführt. Hat ***Modus*** den Wert „FALSE", so werden die folgenden Anweisungen auf dem seriellen Simulatorkern ausgeführt.

Funktionsaufrufe, die sich auf Netze beziehen

In diesem Abschnitt wird die zweite Gruppe von SNNS–Funktionsaufrufen beschrieben. Es handelt sich dabei um jene Funktionsaufrufe, die sich auf Netze beziehungsweise Netzdateien beziehen. Die zweite Gruppe enthält folgende Funktionsaufrufe:

`loadNet()`	Netz laden
`saveNet()`	Netz speichern
`saveResult()`	Ergebnisdatei speichern
`initNet()`	Netz initialisieren
`trainNet()`	Netz trainieren
`testNet()`	Netz testen

Die ersten zwei Funktionsaufrufe dienen zum Laden und Speichern der Netze. Der dritte Funktionsaufruf speichert eine SNNS–Ergebnisdatei, die übrigen Funktionsaufrufe beziehen sich auf das Training von Netzen.

Die Funktionsaufrufe `loadNet()` und `saveNet()`

Diese beiden Funktionsaufrufe haben das gleiche Format, nämlich:

```
loadNet (Dateiname)
saveNet (Dateiname)
```

Für ***Dateiname*** muß ein gültiger UNIX–Dateiname einer SNNS–Netzdatei (in Anführungsstriche oben eingeschlossen) eingesetzt werden. `loadNet()` lädt ein Netz in den Simulatorkern, `saveNet()` speichert ein Netz, welches sich im Simulatorkern befindet, in der Netzdatei ab. Dies ist notwendig, um trainierte Netze für den späteren Einsatz zu erhalten.

Der Funktionsaufruf `loadNet()` setzt die Systemvariable `CYCLES` auf null. Diese Systemvariable enthält die Anzahl der Trainingszyklen, mit denen das geladene Netz trainiert worden ist.

Beispiele für den Aufruf der beiden Funktionen sind:

```
loadNet ("encoder.net")
. . .
saveNet ("encoder.net")
```

Es erfolgen entsprechend die Ausgaben:

```
Net encoder.net loaded
Network file encoder.net written
```

Der Funktionsaufruf `saveResult()`

Der Funktionsaufruf `saveResult()` speichert eine SNNS–Ergebnisdatei ab. Er hat folgendes Format:

saveResult (*Dateiname*, *Start*, *Ende*, *inclIn*, *inclOut*, *FileMode*)

Der erste Parameter ***Dateiname*** muß unbedingt angegeben werden. Er muß ein gültiger UNIX–Dateiname (in Anführungsstriche oben eingeschlossen) sein. Die weiteren Parameter können nach Bedarf angegeben werden, wobei wieder alle Parameter vor dem letzten, der angegeben werden soll, eingegeben werden müssen.
Der Parameter ***Start*** ist die Nummer des ersten Musters, das ausgegeben werden soll. ***End*** ist die Nummer des letzten Musters, das ausgegeben werden soll. Sollen alle Muster ausgegeben werden, kann hier die Systemvariable `PAT`, welche die Anzahl der Muster in der aktuellen Musterdatei enthält, eingesetzt werden. ***inclIn*** und ***inclOut*** sind Wahrheitswerte, mit denen festgelegt wird, ob die Eingabemuster in der Ergebnisdatei enthalten sein sollen (wenn ja, ***inclIn*** auf den Wert „TRUE“ setzen, sonst auf „FALSE“) beziehungsweise ob die Ausgabemuster in der Ergebnisdatei enthalten sein sollen (wenn ja, ***inclOut*** auf den Wert „TRUE“ setzen, sonst auf „FALSE“). Der Parameter ***FileMode*** vom Typ Zeichenkette entscheidet darüber, ob eine neue Ergebnisdatei angelegt wird (***FileMode*** hat den Wert „create“) oder ob die Daten an eine bestehende Datei hinten angehängt werden (***FileMode*** hat den Wert „append“). Dieser Parameter ist nur dann interessant, wenn schon eine Datei mit dem Namen, der für ***Dateiname*** eingesetzt wurde, existiert.
Beispiele für den Aufruf der Funktion `saveResult()` sind:

```
saveResult ("encoder.res")
saveResult ("encoder.res", 1, PAT, FALSE, TRUE, "create")
```

Es erfolgt in beiden Fällen die Ausgabe:

```
Result file encoder.res written
```

Im zweiten Beispiel wurde die Ergebnisdatei mit dem Namen `encoder.res` angelegt. Sie enthält von allen Mustern jeweils die Ausgabemuster.

Die Funktionsaufrufe `initNet()`, `trainNet()` und `testNet()`

Die Funktionsaufrufe `initNet()`, `trainNet()` und `testNet()` sind eng miteinander verknüpft. Alle werden ohne Parameter aufgerufen:

```
initNet()
trainNet()
testNet()
```

`initNet()` initialisiert ein Netz mit der durch den Funktionsaufruf `setInitFunc` eingestellten Initialisierungsfunktion und den eventuell angegebenen Parametern. Dabei wird die Systemvariable `CYCLES` auf null gesetzt. Der Funktionsaufruf `initNet()` ist immer erforderlich, wenn ein neu erzeugtes Netz das erste Mal trainiert werden soll oder wenn ein bereits trainiertes Netz wieder in den untrainierten Zustand versetzt werden soll. `initNet()` wird so aufgerufen:

```
initNet()
```

Es erzeugt die Ausgabe:

```
Net initialized
```

Der Funktionsaufruf `trainNet()` führt *einen* Trainingszyklus durch. Danach wird der Inhalt der Systemvariablen `SSE`, `MSE`, `SSEPU` und `CYCLES` aktualisiert. Die ersten drei Systemvariablen sind die gleichen Fehlerwerte, die beim Trainieren mit der graphischen Benutzeroberfläche auf der Standardausgabe des Terminals erscheinen. Die Systemvariablen haben folgenden Inhalt:

- `SSE` (summed squared error): Summe der quadrierten Abweichungen jedes Ausgabeneurons von seiner Soll–Ausgabe
- `MSE` (mean squared error): SSE geteilt durch die Anzahl der Muster, mit denen trainiert wird
- `SSEPU` (summed squared error per unit): SSE geteilt durch die Anzahl von Ausgabeneuronen des Netzes
- `CYCLES`: Anzahl der bisher trainierten Zyklen

Der Funktionsaufruf `testNet()` dient dazu, den Fehler eines trainierten Netzes zu bestimmen, ohne es weiter zu trainieren. Er macht also nahezu das gleiche wie `trainNet()`, verändert aber nur die Systemvariablen `SSE`, `MSE` und `SSEPU`, nicht das Netz. Er ist dazu gedacht, den Erfolg eines Trainings (den Netzwerkfehler) auf einer anderen Musterdatei zu messen als auf jener, mit der das Netz trainiert wurde. Die Funktionsaufrufe `trainNet()` und `testNet()` werden mit

```
trainNet()
testNet()
```

aufgerufen und erzeugen keine Ausgabe. In den allermeisten Fällen wird der Funktionsaufruf `trainNet()` innerhalb einer Schleife verwendet werden. Nachdem im folgenden Abschnitt noch die Funktionsaufrufe, die sich auf Muster beziehen, besprochen werden, folgen dann einige Beispiele für größere Batchprogramme. Zu beachten ist noch, daß die Systemvariablen mit den Fehlerwerten (`SSE`, `MSE` und `SSEPU`) nach dem Start des Batchinterpreters mit sehr hohen Werten vorbelegt werden. Dies ist keine Fehlfunktion. Nach dem ersten Aufruf von `trainNet` oder `testNet()` werden die Variablen mit den realen Fehlerwerten belegt.

Funktionsaufrufe, die sich auf Muster beziehen

Die folgenden Funktionsaufrufe beziehen sich auf Muster beziehungsweise Musterdateien:

`loadPattern()`	Musterdatei laden
`setPattern()`	aktuelle Musterdatei setzen
`delPattern()`	Musterdatei löschen

Der Simulatorkern kann mehrere Musterdateien gleichzeitig enthalten (zur Zeit fünf). Zwischen diesen Musterdateien kann mit dem Funktionsaufruf `setPattern()` umgeschaltet werden; `delPattern()` dient zum Löschen einer Musterdatei aus dem Simulatorkern. Alle drei Funktionsaufrufe haben als einzigen Parameter den Dateinamen der Musterdatei:

```
loadPattern (Dateiname)
setPattern (Dateiname)
delPattern (Dateiname)
```

Alle drei Funktionsaufrufe setzen den Wert der Systemvariablen `PAT` auf die Anzahl der Muster in der gerade aktuellen Musterdatei. Die Verwaltung der Musterdateien erfolgt genauso wie in der graphischen Benutzeroberfläche: Die zuletzt geladene Musterdatei wird zur aktuellen. Mit `setPattern()` (entsprechend dem „USE"–Button im Fenster „SNNS–control" der graphischen Benutzeroberfläche) wird eine der geladenen Musterdateien zur aktuellen gemacht. Wird mit `delPattern()` die aktuelle Musterdatei aus dem Simulatorkern gelöscht, so wird die zum frühesten Zeitpunkt geladene und noch nicht gelöschte Musterdatei zur aktuellen. Die Funktionsaufrufe:

```
loadPattern ("encoder.pat")
loadPattern ("encoder1.pat")
setPattern ("encoder.pat")
delPattern ("encoder.pat")
```

erzeugen folgende Ausgaben:

```
Patternset encoder.pat loaded; 1 patternset(s) in memory
Patternset encoder1.pat loaded; 2 patternset(s) in memory
Patternset is now encoder.pat
Patternset encoder.pat deleted; 1 patternset(s) in memory
```

7.4 Sonderfunktionen

Der Funktionsaufruf `pruneNet()`

Der Funktionsaufruf `pruneNet()` führt ein „Pruning" eines Netzes durch, analog zu dem im Fenster „Pruning" der graphischen Benutzeroberfläche implementierten Verfahren (siehe Kapitel 10 des SNNS–Handbuchs: Pruning algorithms und [Schreiner 94]). Nachdem mit dem Funktionsaufruf `setPruningFunc` die Funktionen und Parameter eingestellt worden sind, wird das Pruning mit `pruneNet()` gestartet. Der Funktionsaufruf erwartet keine Parameter:

```
pruneNet()
```

Der Funktionsaufruf `execute()`

Mit dem Funktionsaufruf `execute()` wird eine Schnittstelle zum Betriebssystem UNIX eröffnet. `execute()` erlaubt das Starten von Programmen auf der UNIX–Kommandoebene und das Einlesen ihrer Ausgaben in das Batchprogramm. Da auf diese Weise unter anderem auch der gesamte Vorrat von UNIX–Hilfsprogrammen zur Verfügung steht, ist `execute()` sehr vielseitig. Denkbare Anwendungen sind zum Beispiel der Aufruf von Analyseprogrammen für SNNS–Ergebnisdateien wie zum Beispiel das mitgelieferte Programm `analyze`, Abfrage von verfügbarem Plattenplatz, Formatierung oder Weiterverarbeitung der Ausgaben des Batchprogramms und vieles mehr. Das Format des Funktionsaufrufs ist wie folgt:

```
execute (Befehl, Variable1, Variable2 ...)
```

Für ***Befehl*** muß eine Zeichenkette mit der gesamten UNIX–Kommandozeile für das UNIX–Programm, das ausgeführt werden soll, eingesetzt werden. Ausgaben, die das UNIX–Programm auf die Standardausgabe macht, können nun vom Batchinterpreter eingelesen werden. Dazu muß das UNIX–Programm seine Ausgaben, durch Leerzeichen getrennt, hintereinander auf einer Zeile ausgeben. Ist dies nicht möglich, so kann die Ausgabe in den meisten Fällen mit UNIX–Programmen wie AWK oder grep (siehe UNIX–Dokumentation) auf das geforderte Format gebracht werden. Die einzelnen Felder der Ausgabe werden vom Batchinterpreter in der gleichen Reihenfolge, wie sie auf der Standardausgabe erscheinen, den Variablen ***Variable1***, ***Variable2*** und so weiter zugewiesen, wobei die eingelesenen Werte automatisch einem der vier Typen des Batchinterpreters zugeordnet werden. Es können soviele Felder der Ausgabe gespeichert werden, wie Variable angegeben worden sind. Umgekehrt werden nur soviele Variable mit dem Wert von Feldern der Ausgabe belegt, wie Felder auf der Standardausgabe erscheinen.
Zusätzlich wird der Exit–Status des UNIX–Programms in der Systemvariablen `EXIT_CODE` gespeichert. Beispiele für den Aufruf von `execute()` sind:

```
execute ("date", one, two, three, four)
print ("Es ist ", four, " Uhr.")
```

Dieser Funktionsaufruf startet den Befehl `date` auf der UNIX–Kommandozeile und liest die Ausgabe `Fri May 19 16:28:29 GMT 1995` in die vier Variablen `one`, `two`, `three` und `four` ein. Davon wird nur die vierte, welche die Uhrzeit enthält, `four`, ausgegeben. Der Batchinterpreter gibt also aus:

```
Es ist 16:28:29 Uhr.
```

Ein weiteres Beispiel wäre die Bestimmung von freiem Plattenspeicherplatz. Mit dem Funktionsaufruf:

```
execute ("df .| grep dev", dmy, dmy, dmy, freeblocks)
print ("Es sind ", freeblocks, " Blocks frei.")
```

kann die Anzahl freier Plattenspeicherblöcke bestimmt werden. Der UNIX–Befehl `df .` gibt zunächst folgendes aus:

```
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda4             256536  197608    46102     81%   /
```

Über eine UNIX–Pipe wird diese Ausgabe mit dem Programm `grep` auf die interessierende zweite Zeile reduziert (es werden alle Zeilen herausgefiltert, die die Zeichenkette `dev` enthalten). Diese zweite Zeile wird vom Batchinterpreter eingelesen. Dabei werden die ersten drei Felder der Ausgabe (nacheinander) in der Variablen `dmy` gespeichert und nur das vierte, interessierende Feld mit dem Plattenspeicherplatz in der Variablen `freeblocks`. Es erfolgt die Ausgabe:

```
Es sind 46102 Blocks frei.
```

Natürlich kann der Inhalt von `freeblocks` auch anderweitig verwendet werden. Zu beachten ist, daß die Ausgaben der UNIX–Befehle je nach UNIX–Version und Rechnertyp auch ein abweichendes Format haben können.

Diese Beispiele sollen nur als Anregung für Anwendungen des Funktionsaufrufs `execute()` dienen. Denkbar ist auch der Aufruf weiterer Batchinterpreter, die auf diese Weise Teilaufgaben verrichten könnten. Hierzu können mit dem Kommandozeilenschalter `-q` alle Ausgaben des Batchinterpreters *außer* der Ausgabe des `print()`–Funktionsaufrufs unterbunden werden, um das Einlesen der Ausgabe zu erleichtern.

Der Funktionsaufruf `exit()`

Als letztes sei der Funktionsaufruf `exit()` beschrieben. Er verläßt das Batchprogramm sofort und beendet den Batchinterpreter. Als Parameter kann der Exit–Status angegeben werden, den der Batchinterpreter an das aufrufende Programm (im Normalfall die UNIX–Shell) zurückgibt. Ist kein Parameter angegeben, gibt der Batchinterpreter den Exit–Status null zurück. Das Format des Funktionsaufrufs ist:

```
exit (Status)
```

Die ganze Zahl *`Status`* kann zwischen -128 und $+127$ liegen. Tut sie das nicht, so wird sie auf dieses Intervall abgebildet und es wird eine Warnung ausgegeben. Denkbar wäre unter vielen anderen folgende Anwendung (anknüpfend an das Beispiel zur Bestimmung des freien Plattenspeicherplatzes im vorigen Abschnitt):

```
if freeblocks < 1000 then
  print ("Nicht genug Platz auf der Platte!")
  exit (1)
endif
```

7.5 Beispielprogramme

Ein typisches Programmstück für einen Trainingslauf kann folgendermaßen aussehen:

```
loadNet("encoder.net")
loadPattern("encoder.pat")
setInitFunc("Randomize_Weights", 1.0, -1.0)
initNet()

while SSE > 6.9 and CYCLES < 1000 do
```

```
 if CYCLES mod 10 == 0 then
   print ("cycles = ", CYCLES, "  SSE = ", SSE) endif
 trainNet()
endwhile

saveResult("encoder.res", 1, PAT, TRUE, TRUE, "create")
saveNet("encoder.trained.net")

print ("Cycles trained: ", CYCLES)
print ("Training stopped at error: ", SSE)
```

Das Batchprogramm lädt die mitgelieferte Netzdatei `encoder.net` sowie die dazugehörige Musterdatei, initialisiert das Netz und trainiert es, bis der Fehler SSE kleiner oder gleich 6.9 ist. Das trainierte Netz und eine Ergebnisdatei werden abgespeichert. Hierbei werden die folgenden Ausgaben erzeugt, wobei Meldungen des Batchinterpreters und Ausgaben der `print`–Funktion gemischt vorkommen:

```
Net encoder.net loaded
Patternset encoder.pat loaded; 1 patternset(s) in memory
Init function is now Randomize_Weights
Net initialised
cycles = 0  SSE = 3.40282e+38
cycles = 10  SSE = 7.68288
cycles = 20  SSE = 7.08139
cycles = 30  SSE = 6.95443
Result file encoder.res written
Network file encoder.trained.net written
Cycles trained: 40
Training stopped at error: 6.89944
```

Bei dem folgenden Beispiel wird die Ausgabe des Analyseprogramms `analyze` eingelesen. Zu diesem Zweck wird seine Ausgabe mit dem mitgelieferten AWK–Programm `analyze.gawk` in eine Zeile gebracht. Beachtenswert ist auch, wie die UNIX–Kommandozeile durch Verkettung mittels des +–Operators zusammengesetzt wird:

```
loadNet ("encoder.net")
loadPattern ("encoder.pat")
initNet ()

for i := 1 to 500 do
 trainNet ()
endfor

resfile := "test.res"
saveResult (resfile, 1, PAT, FALSE, TRUE, "create")
saveNet ("enc1.net")

command := "analyze -s -e WTA -i " + resfile + " | analyze.gawk"
execute (command, w, r, u, e)
print ("wrong: ", w, "  right: ", r, "  unknown: ", u, "  error: ", e)
```

Es erfolgt die Ausgabe:

```
Net encoder.net loaded
Patternset encoder.pat loaded; 1 patternset(s) in memory
-> Batchman warning at line 3:
   Init function and params not specified; using defaults
Net initialised
Result file test.res written
Network file enc1.net written
wrong: 87.5  right: 12.5  unknown: 0  error: 7
```

Das nun folgende Programm ist ein Beispiel für die Validierung eines Trainings mit einer zweiten Musterdatei. Hierbei wird das Netz mit einer Trainingsmusterdatei trainiert; der Fehler, der als Abbruchkriterium für das Training herangezogen werden soll, wird jedoch auf einer anderen Musterdatei gemessen. So kann beurteilt werden, ob das Netz in der Lage ist, ihm unbekannte Muster richtig zu klassifizieren.

```
loadNet ("test.net")
loadPattern ("validate.pat")
loadPattern ("training.pat")
initNet ()

repeat
  for i := 1 to 20 do
   trainNet ()
  endfor
  saveNet ("test." + CYCLES + "cycles.net")
  setPattern ("validate.pat")
  testNet ()
  valid_error := SSE
  setPattern ("training.pat")
until valid_error < 2.5

saveResult ("test.res")
```

Das Programm trainiert das Netz immer je 20 Zyklen lang, speichert es unter einem für jeden Durchlauf der repeat–Schleife neu generierten Dateinamen wie `test.20cycles.net` ab und testet es mit der Validierungsmustermenge. Der Vorgang wiederholt sich so lange, bis der Fehler auf der Validierungsmustermenge kleiner als 2.5 ist.

Kapitel 8

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit sollte eine Sprache zur Batchsimulation neuronaler Netze für den Stuttgarter Neuronale Netze Simulator SNNS entworfen werden. Hierfür wurden bestehende Sprachen zur Batchsimulation auf ihre Eignung als Vorbilder hin untersucht. Die Entscheidung, eine vorhandene Sprache (oder zumindest deren Syntax und Semantik) zu übernehmen oder eine neue Programmiersprache zu entwerfen, fiel aufgrund mehrerer Kriterien zugunsten des Neuentwurfs einer Batchsprache aus.
Die Hauptaspekte bei der Entscheidung waren:

- Die Batchsprache soll die Leistungsfähigkeit der graphischen Benutzeroberfläche nicht nur erhalten, sondern, wenn möglich, ausbauen.
- Die Batchsprache soll einfach zu erlernen und gleichzeitig mächtig und flexibel genug sein, um anspruchsvolle Aufgaben aus dem Bereich der Batch–Simulation neuronaler Netze erfüllen zu können.
- Die Batchsprache soll für zukünftige Funktionen von SNNS leicht erweiterbar sein, um mit der ständigen Weiterentwicklung dieses Projekts Schritt halten zu können.

Die Grammatik der Sprache wurde — unter Berücksichtigung von Erfahrungen mit der Anwendung anderer höherer Programmiersprachen — formal spezifiziert, wobei immer auf leichte Erlernbarkeit Wert gelegt wurde. Es wurde ein Batchinterpreter für die neue Batchsprache implementiert, der auf die SNNS–Kern–Funktionsschnittstelle aufsetzt. Die Implementierung erfolgte in ANSI–C unter Beachtung der Programmierrichtlinien der SNNS–Gruppe.

Die Implementierung erfüllt die Anforderungen in allen Punkten. Entstanden ist eine interpretierte Batchsprache mit vier Datentypen, Variablen und Kontrollstrukturen wie in den Sprachen Pascal oder C. Alle Funktionen der graphischen Benutzeroberfläche zum Training und Test neuronaler Netze werden unterstützt, UNIX–Programme können zur Analyse der Trainingsergebnisse und zu anderen Zwecken aufgerufen werden und die Ausgabe dieser Programme kann in den Batchinterpreter eingelesen werden. Mittels einer vielseitigen Druckanweisung besteht die Möglichkeit, aussagekräftige Protokolldateien über die Trainingsläufe zu erstellen.

Gemäß der Aufgabenstellung wurde eine ausführliche Benutzeranleitung für die Programmierung in der neuen Batchsprache erstellt, die Anwenderinnen und Anwendern mit und ohne Programmiererfahrung Zugang zur Batchsimulation neuronaler Netze mit SNNS verschaffen soll. Die Benutzeranleitung wird in das SNNS–Handbuch übernommen.

Als Ausblick kann die Möglichkeit eröffnet werden, die graphische Benutzeroberfläche von SNNS für Demonstrations– und Testzwecke über Kommandos der Batchsprache „fernzusteuern“. Spezielle Kommandos hierfür sind nicht implementiert worden, aber die Erweiterung ist in Form einer Schnittstellendatei vorbereitet worden.

Anhang

A. Grammatik

$name \rightarrow firstletter\ rest$
$firstletter \rightarrow \mathbf{a} \mid \ldots \mid \mathbf{z} \mid \mathbf{A} \mid \ldots \mid \mathbf{Z} \mid \mathbf{\_}$
$rest \rightarrow \epsilon \mid rest\ otherletter$
$otherletter \rightarrow firstletter \mid \mathbf{0} \mid \ldots \mid \mathbf{9}$

$number \rightarrow digit \mid number\ digit$
$digit \rightarrow \mathbf{0} \mid \ldots \mid \mathbf{9}$

$$
\begin{aligned}
realnumber \rightarrow\ & integerPart\ number \\
& \mid integerPart\ number\ exponent \\
& \mid number\ exponent
\end{aligned}
$$

$integerPart \rightarrow number\ \mathbf{.} \mid \mathbf{.}$
$exponent \rightarrow \mathbf{e}\ sign\ number \mid \mathbf{E}\ sign\ number$
$sign \rightarrow \epsilon \mid + \mid -$

$string \rightarrow \mathbf{"}stringcharacter\mathbf{"}$
$stringcharacter \rightarrow character \mid stringcharacter\ character$
$character \rightarrow beliebigesDruckbaresZeichen$

$boolean \rightarrow \mathbf{TRUE} \mid \mathbf{FALSE}$

$$
\begin{aligned}
expression \rightarrow\ & simpleExpression \\
& \mid unaryOperator\ expression \\
& \mid expression\ binaryOperator\ expression
\end{aligned}
$$

$$
\begin{aligned}
simpleExpression \rightarrow\ & \mathbf{(}\ expression\ \mathbf{)} \\
& \mid identifier
\end{aligned}
$$

$$
\begin{aligned}
unaryOperator \rightarrow\ & + \mid - \mid \mathbf{not} \mid \mathbf{!} \\
& \mid \mathbf{ln} \mid \mathbf{log} \mid \mathbf{sqrt}
\end{aligned}
$$

$\mathit{binaryOperator} \rightarrow + \mid - \mid * \mid / \mid \textbf{div} \mid \% \mid \textbf{mod} \mid < \mid > \mid \textbf{==} \mid \textbf{!=}$
$\quad \mid <> \mid <= \mid =< \mid >= \mid => \mid \textbf{and} \mid \&\& \mid \textbf{or} \mid \|$[1] $\mid ** \mid \hat{}$

$\mathit{identifier} \rightarrow \mathit{name} \mid \mathit{number} \mid \mathit{realnumber} \mid \mathit{string} \mid \mathit{boolean}$

$\mathit{functioncall} \rightarrow \mathit{identifier}\ \textbf{(}\ \mathit{parameterlist}\ \textbf{)}$
$\mathit{parameterlist} \rightarrow \epsilon \mid \mathit{fullList}$
$\mathit{fullList} \rightarrow \mathit{expression} \mid \mathit{fullList}\ \textbf{,}\ \mathit{expression}$

$\mathit{assignment} \rightarrow \mathit{identifier}\ \mathit{assignmentOperator}\ \mathit{expression}$
$\mathit{assignmentOperator} \rightarrow = \mid :=$

$\mathit{block} \rightarrow \mathit{statement}$
$\quad \mid \mathit{block}\ \mathit{statementDelimiter}\ \mathit{statement}$
$\mathit{statementDelimiter} \rightarrow \textbf{;} \mid \textbf{newlineCharacter}$

$\mathit{statement} \rightarrow \epsilon$
$\quad \mid \mathit{functioncall}$
$\quad \mid \mathit{assignment}$
$\quad \mid \textbf{for}\ \mathit{assignment}\ \textbf{to}\ \mathit{expression}\ \textbf{do}\ \mathit{block}\ \textbf{endfor}$
$\quad \mid \textbf{while}\ \mathit{expression}\ \textbf{do}\ \mathit{block}\ \textbf{endwhile}$
$\quad \mid \textbf{repeat}\ \mathit{block}\ \textbf{until}\ \mathit{expression}$
$\quad \mid \textbf{if}\ \mathit{expression}\ \textbf{then}\ \mathit{block}\ \textbf{endif}$
$\quad \mid \textbf{if}\ \mathit{expression}\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \mathit{block}\ \textbf{endif}$
$\quad \mid \textbf{break}$
$\quad \mid \textbf{continue}$

$\mathit{batchprogram} \rightarrow \mathit{block}$

[1] Hier ist das Terminal „doppelter senkrechter Strich“ gemeint.

B. Operatoren und Schlüsselworte

In Ausdrücken sind folgende Operatoren zulässig:

Operator	Name
+, −	Vorzeichen
not, !	logische Negation
sqrt	Quadratwurzel
ln	natürlicher Logarithmus
log	Logarithmus zur Basis 10
**, ^	Exponentialfunktion
*	Multiplikation
/	Division
div	Ganzzahl-Division
mod, %	Restwert
+	Addition
−	Subtraktion
<	kleiner als
<=, =<	kleiner gleich
>	größer als
>=, =>	größer gleich
==	gleich
<>, !=	ungleich
and, &&	logisches UND
or, $\vert\vert$	logisches ODER

Die Zuweisungsoperatoren sind:

:=, =

Schlüsselworte sind:
for, to, do, endfor, while, endwhile, repeat, until, if, then, else, endif, break, continue.

C. Namen der Funktionsaufrufe

Folgende Namen sind reserviert für Funktionsaufrufe:

Name	Zweck
print	Drucken von Argumenten
execute	Ausführen von UNIX–Befehlen
exit	Verlassen des Batchinterpreters
setInitFunc	Initialisierungsfunktion wählen
setLearnFunc	Lernfunktion wählen
setUpdateFunc	Updatefunktion wählen
setPruningFunc	Pruningfunktionen wählen
setSubPattern	Subpattern shifting scheme definieren
setShuffle	Shuffle–Modus ändern
setSubShuffle	Subpattern–Shuffle–Modus ändern
setParallelMode	Parallele Verarbeitung ein– oder ausschalten
initNet	Netz initialisieren
loadNet	Netz laden
saveNet	Netz speichern
saveResult	Ergebnisdatei speichern
trainNet	Netz trainieren
testNet	Netz testen
pruneNet	Netz „prunen“
loadPattern	Musterdatei laden
setPattern	aktuelle Musterdatei setzen
delPattern	Musterdatei löschen

D. Fehlermeldungen

Die folgenden Fehlermeldungen und Warnungen können vom Batchinterpreter ausgegeben werden. Nicht eingeschlossen sind Fehlermeldungen des SNNS–Kerns sowie der Teil der Fehlermeldungen, der vom Parsergenerator Bison zur Verfügung gestellt wird.

Fehlermeldungen:

```
Pattern file not yet loaded
String argument expected
Real value argument expected
Integer value argument expected
Boolean value argument expected
Cannot print this
Shell command line string missing
Execution of shell command failed
Constants are not allowed as parameters here
Error parsing shell command's output
Unknown init function name
Unknown learning function name
Unknown update function name
At least four arguments expected
Missing network filename
Missing result filename
Missing pattern filename
Assignment to constant or built-in variable
Right side of assignment is undefined
Invalid operand type for 'sqrt'
Argument for 'sqrt' has negative value
Invalid operand type for 'ln'
Argument for 'ln' is zero or negative
Invalid operand type for 'log'
Argument for 'log' is zero or negative
Invalid operand type(s) for '+'
Invalid operand type(s) for '-'
Invalid operand type(s) for '*'
Invalid operand type(s) for '/'
Division by zero attempted within '/'
Invalid operand type(s) for 'div'
Division by zero attempted within 'div'
Invalid operand type(s) for 'mod'
Division by zero attempted within 'mod'
Invalid operand type(s) for '**' resp. '^'
Arg 2 of '**' resp. '^' must be of integral type if arg 1 is negative
Invalid operand type for 'not'
```

```
Invalid operand type(s) for 'and'
Invalid operand type(s) for 'or'
Invalid operand type(s) for '<'
Invalid operand type(s) for '>'
Invalid operand type(s) for '<='
Invalid operand type(s) for '>='
Invalid operand type(s) for '=='
Invalid operand type(s) for '!=' resp. '<>'
Boolean type expression expected
At least one argument expected
Number of arguments must match an integer multiple of 4
Argument TRUE or FALSE expected
Memory allocation failed
```

Warnungen:

```
Init function and params not specified; using defaults
Continue found outside of a block; ignored
Found more arguments than expected; ignored
No function arguments expected; ignored
Exit status out of range
```

Abbildungsverzeichnis

Literaturverzeichnis

[Aho et al. 74] Aho, Alfred V.; Johnson, S. C.: *Programming Utilities and Libraries*
Computing Surveys, June, 1974

[Aho et al. 87] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.:
COMPILERS, Principles, Techniques and Tools
Bell Telephone Laboratories, Inc., 1987

[Aho et al. 88a] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: *Compilerbau*
Teile 1 und 2, Addison Wesley Publishing Company, Bonn u. a., 1988

[Aho et al. 88b] Aho, Alfred V.; Kernighan, Brian W.; Weinberger, Peter J.:
The AWK Programming Language
Addison Wesley Publishing Company, Reading, Massachusetts u. a., 1988

[van Camp 93] van Camp, Drew: *Xerion Neural Network Simulator V 3.1*
Department of Computer Science, University of Toronto, April 14, 1993

[Close et al. 93] Close, Diane Barlow; Robbins, Arnold D.; Rubin, Paul H.; Stallman, Richard: *The GAWK Manual, Edition 0.15, April 1993*
Free Software Foundation, Inc., Cambridge, Massachusetts, 1993

[Donnelly et al. 92] Donnelly, Charles; Stallman, Richard:
Bison, The YACC–compatible Parser Generator, December 1992
Free Software Foundation, Inc., Cambridge, Massachusetts, 1992

[Hopcroft et al. 79] Hopcroft, John E.; Ullman, Jeffrey D.:
Introduction to Automata Theory, Languages, and Computation
Kapitel 10, Addison Wesley Publishing Company, Reading, Massachusetts u. a., 1979

[Horowitz 83] Horowitz, Ellis (Hrsg.): *Programming Languages: A Grand Tour*
Springer–Verlag, Berlin, Heidelberg, New York, 1983

[Jensen et al. 74] Jensen, Kathleen; Wirth, Niklaus: *PASCAL — User Manual and Report*
Springer–Verlag, Berlin, Heidelberg, New York, 1974

[Krüger 91] Krüger, Guido:
C Programmieren, Ein Kurs zum Selbststudium mit Musterlösungen
Addison Wesley Publishing Company, Bonn u. a., 1991

[Miyata 91] Miyata, Yoshiro: *A User's guide to PlaNet, Version 5.6, A Tool for Constructing, Running and Looking into a PDP Network*
Computer Science Department, University of Colorado, Boulder; January 29, 1991

[Ousterhout 93] Ousterhout, John K.: *Tcl and the Tk Toolkit*
Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley 1993

[Paxson 90] Paxson, Vern: *flex*
Computer Systems Engineering, Lawrence Berkeley Laboratory, University of California, Berkeley 1990

[Prechelt 94] Prechelt, Lutz: *CuPit — A Parallel Language for Neural Algorithms: Language Reference and Tutorial*
Technical Report 4/94, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, January 20, 1994

[Rojas 93] Rojas, Raúl: *Theorie der neuronalen Netze*
Kapitel 7, Springer Verlag, Berlin, Heidelberg u. a., 1993

[Rumelhart et al. 86] Rumelhart, D. E.; McClelland, J. L.: *Parallel Distributed Processing*
Volume 1, MIT Press, Massachusetts, 1986

[Schildt 90] Schildt, Herbert: *The annotated ANSI C standard: American national standard for programming languages — C; ANSI–ISO 9899–1990*
Osborne McGraw-Hill, Berkeley, California, USA, 1990

[Schreiner 94] Schreiner, Tobias: *Ausdünnungsverfahren für neuronale Netze*
Diplomarbeit Nr. 1140, Institut für Parallele und Verteilte Höchstleistungsrechner, Fakultät Informatik, Universität Stuttgart 1994

[Wilke 94] Wilke, Peter:
NEUROGRAPH Bedienhandbuch, Version 3.0, Revision 4, August 1994
Erlangen 1994

[Wirth 85] Wirth, Niklaus: *Programmieren in Modula–2*
Springer–Verlag, Berlin, Heidelberg, New York, 1985

[Zell 92] Zell, Andreas (Hrsg.): *Workshop: Simulation neuronaler Netze mit SNNS*
Bericht Nr. 10/92, IPVR, Universität Stuttgart 1992

[Zell et al. 93] Zell, Andreas, Mamier, Günter; Hübner, Ralf; Mache, Niels; Schmalzl, Michael; Sommer, Tilman; Vogt, Michael:
SNNS: An Efficient Simulator for Neural Nets
S. 343–346, in: Schwetman, Herb; Walrand, Jean; Bagchi, Kallol; DeGroot, Doug (Hrsg.): *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '93*
Simulation Series Volume 25 Number 1, Simulation Councils, Inc., San Diego, California, 1993

[Zell 94] Zell, Andreas: *Simulation Neuronaler Netze*
Addison Wesley Publishing Company, Bonn u. a., 1994

[Zell et al. 95] Zell, Andreas; Mamier, Günter; Vogt, Michael; Mache, Niels; Hübner, Ralf; Döring, Sven; Herrmann, Kai-Uwe; Soyez, Tobias; Schmalzl, Michael; Sommer, Tilman; Hatzigeorgiou, Artemis; Posselt, Dietmar; Schreiner, Tobias; Kett, Bernward; Clemente, Gianfranco;
external contributions by: Reczko, Martin; Riedmiller, Martin; Seemann, Mark; Ritt, Marcus; DeCoster, Jamie; Biedermann, Jochen; Danz, Joachim; Wehrfritz, Christian; Werner, Randolf; Berthold, Michael:
SNNS, Stuttgart Neural Network Simulator, User Manual, Version 4.0, Report No. 3/95
Fakultät Informatik, IPVR, Universität Stuttgart 1995