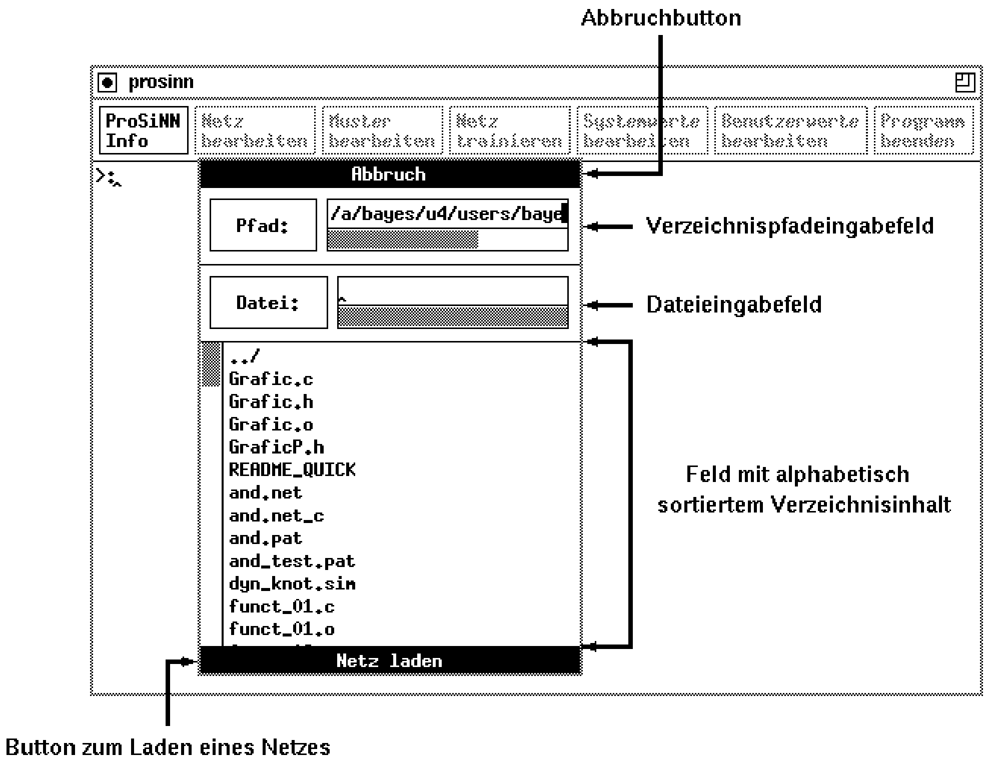

Abbildung 3.8: Die Dateiauswahlbox des Menüpunktes *Netz laden*

den Inhalt des Verzeichnisses dargestellt, läßt sich ein Wechsel des Arbeitsverzeichnisses bequemer durchführen. Dort sind alle von diesem Pfad aus zugänglichen Verzeichnisse vor den Dateien alphabetisch aufgelistet. Ein Verzeichnis hat immer die Endung '/' (z.B. steht '../' für den Vater des Arbeitsverzeichnisses). Durch Setzen des Mauszeigers auf den gewünschten Verzeichnisnamen und durch Drücken der linken Maustaste wird der Pfad umgesetzt und der entsprechende Inhalt angezeigt.

Das Dateieingabefeld dient zum Setzen des gewünschten Filenamens. Mit der Tastatur kann hier der Name der zu bearbeitenden Datei eingegeben werden (die `Return`–Taste wird ignoriert). Auch über das Feld, in dem alle Dateien alphabetisch aufgelistet sind[1], läßt sich mit der Maus ein Dateiname auswählen. Durch Drücken der linken Maustaste (während sich der Mauszeiger auf dem gewünschten Namen befindet) wird der Filename in das Dateieingabefeld übernommen. Benutzt man stattdessen die mittlere Maustaste, so wird der Name übernommen und die Datei sofort geladen, gespeichert oder ausgeführt (je nach Menüpunkt; hier wird das Netz geladen).

Mit dem Button Netz laden wird die Netzdatei geladen, deren Pfad und Name in den entsprechenden Teilfeldern angegeben ist.

Bem.:

Die Popup–Fenster von ProSiNN sind in ihrer geographischen Lage und ihrer Geometrie unabhängig vom Hauptfenster, das sich beliebig manipulieren läßt (Lage und Größe). So-

[1] Files, die mit einem Punkt beginnen, sind in der Liste nicht enthalten

lange ein Popup–Fenster aktiv ist, kann der Anwender in der Regel weder das Menüsystem noch den Kommandozeileninterpreter benutzen.

Die Abbildung 3.9 zeigt ein Beispiel, bei dem sich das Hauptfenster vor dem Popup–Fenster befindet.

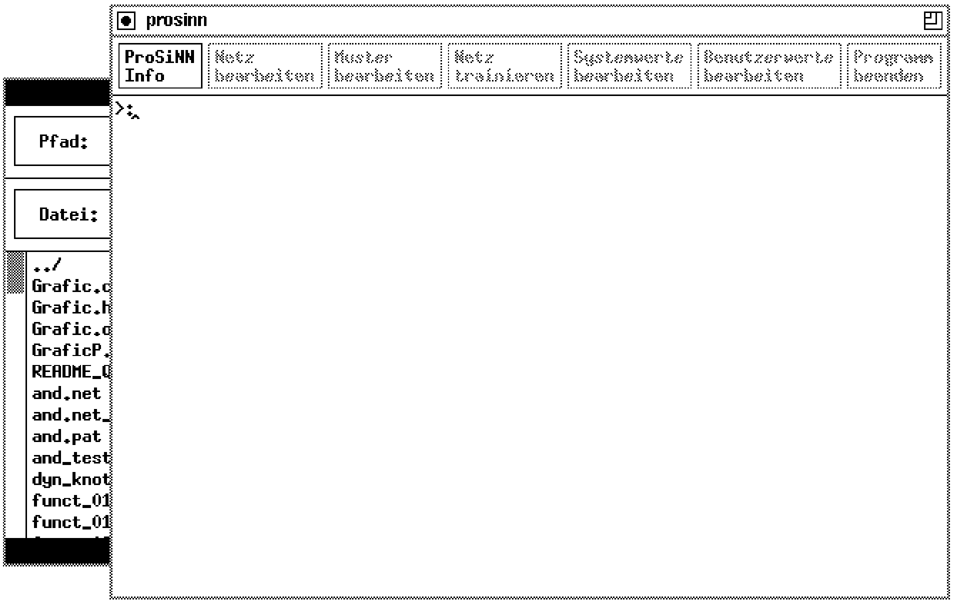

Abbildung 3.9: Das ProSiNN–Hauptfenster vor einem Popupfenster

Um zwischen Alternativen in einem Popup–Fenster auswählen zu können, bietet ProSiNN *Umschalter (Toggles)* an. Wenn ein Umschalter aus einem einzigen Button besteht, läßt sich mit ihm ein Zustand ein bzw. ausschalten. Beinhaltet ein Toggle dagegen mehrere Schalter, so ist immer nur einer von ihnen aktiv.

Bild 3.10 zeigt ein Popup–Fenster, das zum Bearbeiten der im System vorliegenden Netzstruktur erzeugt wurde (siehe Abschnitt 14.6). Der hier verwendete Umschalter besteht aus zwei Buttons: Im dargestellten Zustand können die Neuronen des Netzes bearbeitet werden. Um ein Gewicht auszuwählen, muß mit dem Mauszeiger der Button Gewichte angeklickt werden. Anschließend sind beide Schalter invertiert und die gewichtsinternen Werte können betrachtet oder geändert werden.

Die im letzten Unterabschnitt genannten Festlegungen für Editieroperationen gelten auch für Eingabefelder im Menüsystem. Es gibt allerdings einen Unterschied bei der Zahleneingabe im Menüsystem und beim Kommandozeileninterpreter (bzw. in ProSiNN–Programmen). Wenn der Anwender eine Fließkommazahl eingibt, ist er beim Interpreter an eine strenge Syntax gebunden. Er verlangt z.B. zwingend den Dezimalpunkt (1.0 ist im Gegensatz zu 1 eine Fließkommazahl). Bei Zahleneingabefeldern in Popup–Fenstern ist das Format freier. Hier wird aus dem Kontext heraus entschieden, ob ein Wert in einer korrekten Darstellung eingegeben wurde. Wird eine ganze Zahl erwartet, so akzeptiert ProSiNN z.B. die Eingabe *123.e34* und verarbeitet dann intern den Wert *123*. Das System liest hier also solange Zeichen einer Zahl ein, bis eine Verletzung der Syntax festgestellt wurde (das Menüsystem verwendet die Syntax, die durch die Programmiersprache *C* defi-

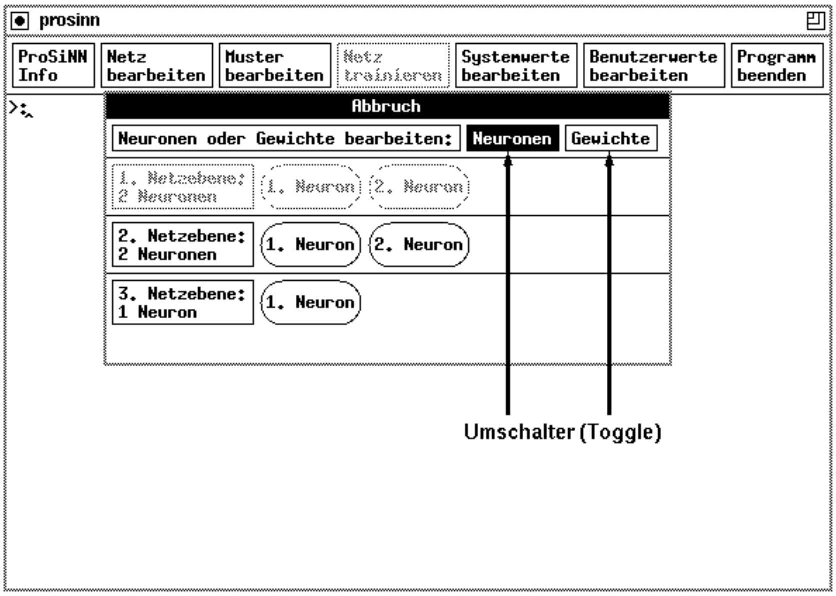

Abbildung 3.10: Der ProSiNN–Zustandsumschalter (Toggle)

niert ist [KER83]). Hält sich der Benutzer an die in Abschnitt 4.2 definierte Syntax, dann werden eingegebene Zahlen immer vom ProSiNN–Menüsystem korrekt verarbeitet.

Ensteht bei der Benutzung des Menüsystems ein Fehler, so wird über das Ein/Ausgabefenster ein invertierter (schwarzer) Button mit einer Fehlermeldung gelegt. Durch Anklicken des Buttons verschwindet die Meldung.

Der in Abbildung 3.11 angegebene Fehler entstand durch Drücken des Buttons Netz laden in der in Bild 3.8 dargestellten Situation. Da kein Filename eingegeben wurde, kann ProSiNN auch keine Netzdatei laden!

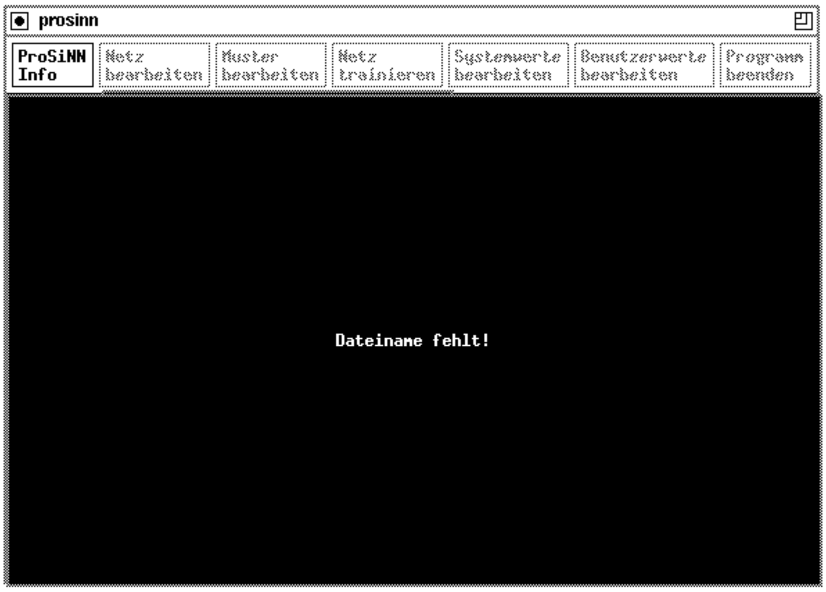

Abbildung 3.11: Die Fehlerdarstellung der ProSiNN–Menüoberfläche

Teil II

Die Programmiersprache ProSiNN

Kapitel 4

Bezeichner, Datentypen, Operatoren und Ausdrücke

Imperative Programmiersprachen (zu denen auch ProSiNN gehört) basieren auf der Verwendung von Variablen und deren Manipulation. Die von ihnen angebotenen Variablenarten gründen sich oft auf individuell verschiedenen Datentypen, die die Lösung spezieller Aufgaben erleichtern. Der überwiegende Teil dieses Kapitels beschäftigt sich deshalb mit diesen Typen, die die Grundlage aller in ProSiNN verwendeten Variablen bilden.

Im ersten Abschnitt werden alle (reservierten) Bezeichner, deren Namenskonventionen und die Schlüsselwörter (Kommandos) vorgestellt. Den Anschluß bildet die Syntax und Semantik der verwendeten Datentypen. Die angebotenen Präfixe, die den Zugriff auf (netz)interne Strukturen ermöglichen, werden im darauffolgenden Abschnitt beschrieben. Den Abschluß bilden die grundlegenden arithmetischen Operatoren und die erzeugbaren mathematischen Ausdrücke.

4.1 Bezeichner und Schlüsselwörter

In diesem Abschnitt werden zuerst die Konventionen erläutert, die ProSiNN bei der Verwendung von Bezeichnern gebraucht. Im Anschluß werden alle vordefinierten Befehle (Schlüsselwörter), Konstanten und Variablen vorgestellt.

Ein Bezeichner ist eine Zeichenfolge, die zur Identifizierung von Objekten dient. Er besteht aus einer Folge von Buchstaben, Ziffern und dem Unterstrich '_'. ProSiNN unterscheidet dabei zwischen Groß– und Kleinbuchstaben. Ein Bezeichner muß stets mit dem Unterstrich oder einem Buchstaben beginnen. Das System verwaltet beliebig lange Zeichenfolgen als Bezeichner, wobei aber nur die ersten 32 Zeichen signifikant sind.

Im folgenden werden die Schlüsselwörter aller vordefinierten Kommandos alphabetisch aufgelistet. Die Befehlsbeschreibungen befinden sich in den dazugehörigen Kapiteln.

activation
alias
axes
clear_grafic
close_all_grafics
copy
cycle_pattern
delete
derivation_of_activation_funct
double_of
fast_cycle
free_all
free_grafic_protocol
free_pattern
free_var
if
init_net
insert
layer
line
names
normal_activation
open_grafic
plot_function
print
print_function
println
pwd
read_test_pattern
readpattern
save_c_net
set
set_world
status
sys_names
time
while
add_layer
alter_unit_number
cd
close
close_grafic
cycle
define
derivation_of
dir
error_of
free
free_all_pattern
free_net
free_protocol
grafic_protocol
init
input
int_of
length_of
load
netname
open
plot
position_of
print_error_of
print_pattern
protocol
quit
readnet
reply_to
savenet
set_rand
shell
sub_layer
text
user_names

Die nachfolgenden Bezeichner verwendet ProSiNN als Konstanten:

e:

Die Konstante **e** ist vom Datentyp *double* (siehe Unterabschnitt 4.2.2) und hat den Wert *2.7182818284589998*.

pi:

Die Konstante **pi** ist vom Datentyp *double* und repräsentiert den Gleitkommawert *3.141592653589793*.

EOL:

EOL (**E**nd **O**f **L**ine) wird in einem Ausgabebefehl zur Durchführung eines Zeilenvorschubs verwendet. Diese Konstante ist vom Datentyp *string* (siehe Unterabschnitt 4.2.3).

stdout:

stdout ist ein Dateihandle (siehe Abschnitt 7.2) und lenkt Ausgaben in das Ein/Ausgabefenster um. Intern handelt es sich um eine ganze Zahl (Datentyp *int*; siehe Unterabschnitt 4.2.1), deren Wert *0* beträgt.

off und **on**:

Diese Konstanten sind die Werte, die der Variablen **floatpattern** (siehe unten) zugewiesen werden dürfen. Sie sind vom Datentyp *int*, wobei **off** den Wert *0* und **on** den Wert *1* repräsentiert.

quadratisch, **betragsmaessig**, **mittel_betrag** und **max_betrag**:

Diese Konstanten dürfen der Variablen **error_mode** zugewiesen werden (näheres siehe unter **error_mode**). Intern sind sie vom Datentyp *int*, wobei ihre Werte in obiger Reihenfolge von *0* bis *3* durchnummeriert sind.

Die folgenden Präfixe werden hier nur auf Grund der Vollständigkeit aufgeführt. Die dazugehörigen Erklärungen findet man in Abschnitt 4.3:

delta_pj	**delta_theta**
delta_w	**nr_units**
o_pj	**pattern_in**
pattern_out	**theta**
w	

ProSiNN bietet vordefinierte Funktionen, mit denen sich komplexe Ausdrücke aufgebauen lassen. In Unterabschnitt 4.4.4 werden die anschließend aufgelisteten Funktionen erläutert:

acos	**asin**
atan	**cos**
cosh	**exp**
ln	**log**

sin	**sinh**
sqrt	**tan**
tanh	

Das System verwendet eine Anzahl von globalen Variablen, auf die der Benutzer zurückgreifen kann. Durch das *set*–Kommando sind ihnen in der Regel neue Werte zuweisbar (siehe Abschnitt 5.2). In der folgenden Aufzählung werden diese Variablen eingeführt:

- Für die Ausgabe von Gleitkommazahlen verwendet ProSiNN eine Variable, die die Anzahl der Nachkommastellen beschreibt:

  precision:

  > Dieser Variablen vom Datentyp *int* kann der Anwender positive ganze Zahlen zuweisen. Der Defaultwert ist *8*.

- Wenn ProSiNN ein Neuronales Netz in wieder einlesbarer Form speichert, wird der Inhalt der folgenden Variablen mit in die entsprechende Datei geschrieben (siehe Abschnitt 9.4).

  net_name:

  > Diese Variable beinhaltet eine Zeichenkette (Datentyp *string*; siehe 4.2.3). Der Anwender kann ihr eine beliebige Zeichenfolge zuweisen. Der vorgegebene Wert ist die leere Zeichenkette.

- Zur Verwaltung von Mustern dienen Variablen, deren Bedeutungen jetzt beschrieben werden:

  floatpattern:

  > Diese Variable (Datentyp *int*) wird als boolescher Flag benutzt. Wenn ihr Wert **on** ist, erwartet das System Fließkommazahlen beim Einlesen der Muster. Ist ihr Wert dagegen **off**, wird eine symbolische Schreibweise verlangt (näheres siehe in Kapitel 10). Der Defaultwert ist **on**.

  pat_min und **pat_max**:

  > Hier handelt es sich um zwei Gleitkommavariablen vom Datentyp *double*, die das Intervall definieren, das beim Einlesen symbolischer Muster verwendet wird. **pat_min** beschreibt dabei die untere und **pat_max** die obere Grenze. Der Benutzer kann diesen Variablen beliebige Zahlen zuweisen, solange der Wert von **pat_max** größer oder gleich dem Wert von **pat_min** ist.
  > Die Vorgabewerte sind **pat_min** = *0.0* und **pat_max** = *1.0*.

  nr_pattern:

  > Diese Variable vom Datentyp *int* enthält die Anzahl der im System vorhandenen Trainingsmuster. Der Anwender darf diese Variable nicht direkt setzen (sie wird indirekt beim Einlesen von Mustern erhöht oder durch Musterentfernen erniedrigt).

- Nach der Erzeugung eines Neuronalen Netzes haben alle Schwellenwerte und Gewichte den Wert *0.0*. Um ein Netz trainieren zu können, sollten sie mit einem Wert ungleich *0.0* initialisiert werden. Der *init_net*–Befehl (siehe Abschnitt 9.3) erzeugt Zufallszahlen, die in einem vom Benutzer definierbaren Intervall liegen:

  theta_min und **theta_max**:

  > Diese Fließkommavariablen vom Datentyp *double* schränken den Bereich ein, aus dem die Initialisierungswerte für die Schwelle θ kommen. Sie dürfen beliebig vom Anwender gesetzt werden, solange **theta_max** größer oder gleich **theta_min** ist.
  > Die Defaultwerte sind: **theta_min** = *-0.5* und **theta_max** = *0.5*.

  w_min und **w_max**:

  > Diese Fließkommavariablen (Datentyp *double*) schränken den Bereich ein, aus dem die Initialisierungswerte für die Gewichte kommen. Sie dürfen beliebig vom Benutzer gesetzt werden, solange **w_max** größer oder gleich **w_min** ist.
  > Die Defaultwerte sind: **w_min** = *-0.5* und **w_max** = *0.5*.

- Wenn im System ein Neuronales Netz vorhanden ist, dann beschreiben die folgenden drei Variablen vom Datentyp *int* einen Teil der Netztopologie:

  nr_of_layers:

  > Diese Variable enthält die Anzahl der Netzebenen. Ihr Wert kann vom Anwender nicht geändert werden.

  nr_in_units:

  > Diese Variable beschreibt die Anzahl der Eingabeneuronen (Zellen der ersten Ebene). Sie ist nicht direkt modifizierbar.

  nr_out_units:

  > Diese Variable beinhaltet die Anzahl der Ausgabeneuronen (Zellen der letzten Schicht). Ihr Wert kann nicht direkt geändert werden.

- Die nachfolgenden Variablen beschreiben netzinterne Strukturen, die beim Training Neuronaler Netze für das Fehlermaß und die Terminierung des Lernverfahrens verwendet werden:

  act_error und **old_error**:

  > Diese beiden Fließkommavariablen (Datentyp *double*) beschreiben das aktuelle Fehlermaß, das nach dem letzten Trainingszyklus vorliegt (durch **act_error**) und das vor diesem Trainigszyklus gültige Fehlermaß (durch **old_error**). Sie können beliebig gesetzt werden, wobei ihre Werte aber größer als *0.0* sein müssen. Die vorgegebenen Werte sind *0.0*.

error_limit:

Diese Gleitkommavariable vom Datentyp *double* enthält die untere Fehlermaßgrenze, bei der das Trainingsverfahren beendet wird (siehe auch die nachfolgende Variable). Ihr dürfen beliebige Werte größer als *0.0* zugewiesen werden. Der Vorgabewert ist *0.001*.

error_mode:

Die von ProSiNN zur Verfügung gestellten Lernbefehle verwenden immer das in Gleichung 2.1 und 2.2 dargestellte Fehlermaß zur Modifikation der Gewichte und Schwellenwerte.

Mit dieser Variablen vom Datentyp *int* wird dagegen die Modalität festgelegt, wie ProSiNN das Fehlermaß zur Terminierung des Trainingsverfahrens verwenden soll. Dabei wird der Fehler je nach Modus berechnet und mit der unteren Fehlerschranke **error_limit** verglichen. Wenn der errechnete Wert darunterliegt, wird das Verfahren beendet.

An **error_mode** dürfen nur die folgenden Konstanten zugewiesen werden (p beschreibt die Anzahl der Trainingsmuster **nr_pattern**, n stellt die Anzahl der Ausgabeneuronen **nr_out_units** dar, t_{ij} ist der verlangte Zielwert des Neurons j bei Eingabe des Musters i und o_{ij} repräsentiert den real erhaltenen Ausgabewert dieses Neurons. E beschreibt in den folgenden Gleichungen den Wert, der der Variablen **act_error** vom System zugewiesen wird):

quadratisch:

Dies ist der Defaultwert der Variablen **error_mode**, das quadratische Fehlermaß:

$$E = \sum_{i=1}^{p} \sum_{j=1}^{n} (t_{ij} - o_{ij})^2$$

betragsmaessig:

Bei diesem Fehlermaß wird die absolute Differenz als Summe über alle Neuronen und Muster gebildet:

$$E = \sum_{i=1}^{p} \sum_{j=1}^{n} |t_{ij} - o_{ij}|$$

mittel_betrag:

Hierbei handelt es sich um die auf ein Neuron und ein Muster gemittelte absolute Abweichung über alle Neuronen und Muster:

$$E = \frac{1}{p} \sum_{i=1}^{p} \frac{1}{n} \sum_{j=1}^{n} |t_{ij} - o_{ij}|$$

max_betrag:

Bei diesem Fehlermaß wird die maximale Abweichung eines Ausgabeneurons über alle Muster berechnet:

$$E = \max_{\substack{i \in \{1,\dots,p\} \\ j \in \{1,\dots,n\}}} \{|t_{ij} - o_{ij}|\}$$

- Die folgenden Variablen vom Datentyp *double* beschreiben die Lernrate η und das Lernmoment α:

  eta und **act_eta**:

  Diese Fließkommavariablen repräsentieren die Lernrate η. Dabei steht **eta** für die vom System (oder Anwender) vorgegebene Lernrate. **act_eta** beschreibt die aktuelle Lernrate (η wird durch Erfolge und Fehlschläge beim Netztraining variiert; siehe Abschnitt 2.3). Der Benutzer kann ihnen beliebige Werte zuweisen, sofern sie größer als *0.0* sind. Der von beiden benutzte Vorgabewert ist *0.2*.

  alpha und **act_alpha**:

  Mit diesen Gleitkommavariablen wird das Lernmoment α beschrieben. Dabei steht **alpha** für das vom System (oder Anwender) vorgegebene Lernmoment. **act_alpha** repräsentiert das aktuelle Lernmoment (α wird durch Erfolge und Fehlschläge beim Netztraining variiert; siehe Abschnitt 2.3). Der Benutzer kann ihnen beliebige Werte zuweisen, sofern sie größer als *0.0* sind. Der von beiden benutzte Vorgabewert ist *0.9*.

- Zur Unterstützung des in Abschnitt 2.3 vorgestellten modifizierten Backpropagationalgorithmus werden noch drei Variablen vom Datentyp *double* benötigt:

  fail_limit:

  Diese Fließkommavariable gibt die in Punkt (2.b) des Abschnitts 2.3 genannte Prozentangabe wieder. Wird ihr Wert mit 100% multipliziert, dann repräsentiert das Ergebnis den oberen Grenzwert der erlaubten Abweichung. Der Defaultwert beträgt *1.04* (mit 100% multipliziert ergibt sich 104%; wenn das aktuelle Fehlermaß mehr als 4% über dem Wert des vorhergehenden Zyklus liegt, sieht das Verfahren diesen Lernzyklus als Mißerfolg an). Der Anwender darf der Variablen beliebige Werte zuweisen, sofern sie größer als *1.0* sind.

  phi:

  Diese Gleitkommavariable beschreibt den Erfolgslernratenfaktor Φ. Auch ihr darf vom Benutzer ein beliebiger Wert zugewiesen werden, sofern er größer als *1.0* ist. Defaultmäßig wird *1.05* verwendet.

  beta:

  Bei dieser Fließkommavariablen handelt es sich um den Mißerfolgslernratenfaktor β. Auch sie darf beliebig geändert werden, sofern gilt:
  $0 <$ **beta** < 1. Der Defaultwert beträgt *0.7* .

- Die beiden folgenden Variablen vom Datentyp *int* beschreiben die Anzahl der bisher ausgeführten Trainingszyklen:

  act_iterations:

  Sie beschreibt die Anzahl der beim *letzten* Trainingsbefehl ausgeführten Zyklen. Sie darf auf einen beliebigen ganzzahligen Wert größer *0* gesetzt werden. Allerdings wird sie vor Ausführung des nächsten Trainingskommandos wieder auf *0* gesetzt.

  sum_iterations:

  Diese Variable beschreibt die Gesamtanzahl *aller* bisher ausgeführten Trainingszyklen. Sie darf auf einen beliebigen ganzzahligen Wert größer *0* gesetzt werden. Beim Systemstart wird sie mit *0* initialisiert.

4.2 Datentypen und Konstanten

In diesem Abschnitt sollen die Datentypen vorgestellt werden, auf denen alle ProSiNN–Variablen basieren. Zu jedem Typ wird die Syntax und Semantik angegeben.

4.2.1 Konstanten vom Datentyp *int*

Dieser von ProSiNN verwendete Datentyp wird im folgenden *Integer* oder *int* genannt. Die Syntax einer Integerkonstanten lautet:

int_const ::= [*"+"* | *"-"*] **decimal_const.**
decimal_const ::= **decimal_digit** { **decimal_digit** }.
decimal_digit ::= *"0"* | *"1"* | *"2"* | *"3"* | *"4"* | *"5"* | *"6"* | *"7"* | *"8"* | *"9"*.

Beim vorliegenden System dürfen die Konstanten aus dem nachfolgenden Intervall stammen: $[-2147483648, \ldots, 2147483647]$.

4.2.2 Konstanten vom Datentyp *double*

Dieser Datentyp wird im weiteren als *double* bezeichnet. Die Syntax einer Konstanten ist:

double_const ::= [**int_constant**] *"."* **decimal_const** [**exponent_const**].
exponent_const ::= (*"e"* | *"E"*) **int_const.**

Bei der verwendeten Rechnerumgebung ist das Ursprungsintervall $[-1.8e308 \ldots 1.8e308]$.

Beispiel:

Gültige Konstanten sind u.a.:

Für die Null : `0.0`, `.0`, `0.0e0`, `.0e0`, `.0e-0`, `.0e+0`
Für 3.12 : `3.12`, `312.0e-2`, `0.312e1`, `.312e1`

4.2.3 Konstanten vom Datentyp *string*

Eine Stringkonstante ist eine in Anführungsstrichen eingeschlossene Zeichenfolge. Ihr Datentyp wird als *string* bezeichnet. Die Syntax einer Stringkonstanten lautet:

string_const ::= „“„ { **char_const** } „“„.
char_const ::= **„Ein Zeichen“.**

Bem.:

- **char_const** darf keine Anführungszeichen “ enthalten!
- Eine Stringkonstante darf über das Zeilenende hinausgehen: Das entsprechende *Zeilenendezeichen* wird ignoriert!

Beispiel:

Gültige Konstanten sind:

`“Ich bin eine Zeichenkette“`
`““` (hier handelt es sich um die leere Zeichenkette)
```
“Ich bin eine Zeichenkette,
 die über eine Zeile hinausgeht“
```

4.2.4 Netzebenenkonstanten vom Datentyp *layer*

Eine Netzebenenkonstante hat den Datentyp *layer* und dient zum Ansprechen einer Netzschicht. Ihre Werte stammen aus einer Teilmenge der Integerkonstanten und dürfen in Integerausdrücken verwendet werden. Die Syntax einer solchen Konstante ist:

layer_const ::= **non_zero_decimal_digit** { **decimal_digit** }.
non_zero_decimal_digit ::= *“1“* | *“2“* | *“3“* | *“4“* | *“5“* | *“6“* | *“7“* | *“8“* | *“9“*.

Bem.:

Wenn eine solche Konstante zur Referenz einer Netzebene verwendet wird, dann darf sie nicht größer sein als die Anzahl der vorhandenen Schichten. Bei einem Netz, das z.B. aus drei Ebenen besteht, muß gelten: **layer_const** $\in \{1, 2, 3\}$!

4.2.5 Neuronenkonstanten vom Datentyp *unit*

Der Datentyp einer Neuronenkonstanten wird im ProSiNN–System als *unit* bezeichnet. Mit ihr kann jedes Neuron angesprochen werden. Diese Konstanten dürfen nur verwendet werden, wenn im System ein Neuronales Netz erzeugt oder geladen wurde. Die Syntax lautet:

unit_const ::= **layer_const** *“:“* **unit_number.**
unit_number ::= **non_zero_decimal_digit** { **decimal_digit** }.

Bem.:

- **layer_const** muß eine Netzebene beschreiben, die real existiert.
- **unit_number** repräsentiert das durchnummerierte Neuron der Ebene **layer_const**. Es muß also gelten: **unit_const** $\in \{1, 2, \ldots,$ Anzahl Zellen dieser Netzschicht$\}$
- Man beachte, daß in einem Neuronalen Netz *real* keine Eingabeneuronen existieren.

Beispiel:

Im System befindet sich ein Netz, das in der Eingabeebene 3, in der mittleren Schicht 4 und in der letzen Ebene 2 Neuronen enthält.

Die Zellen der ersten Schicht sind durch `1:1` , `1:2` *und* `1:3`,
die Neuronen der zweiten Ebene durch `2:1` , `2:2` , `2:3` *und* `2:4`
und die Zellen der letzten Schicht durch `3:1` *und* `3:2` *ansprechbar.*

4.2.6 Gewichtskonstanten vom Datentyp *weight*

Der Datentyp einer Gewichtskonstanten wird *weight* genannt. Mit ihr kann jedes vorhandene Gewicht angesprochen werden. Auch diese Konstanten dürfen nur verwendet werden, wenn im ProSiNN–System ein Neuronales Netz vorhanden ist. Die Syntax einer entsprechenden Konstante ist:

weight_const ::= $\textbf{unit\_const}_{source\_layer}$ *“:“* $\textbf{unit\_const}_{destination\_layer}$.

Bem.:

- Die Neuronen $\textbf{unit\_const}_{source\_layer}$ und $\textbf{unit\_const}_{destination\_layer}$ müssen beide im Netz existieren und in *zwei aufeinanderfolgenden* Schichten liegen.
- Man beachte, daß in einem Neuronalen Netz keine Gewichte der letzten Ebene existieren.

Beispiel:

Im System befindet sich ein Netz, das in der Eingabeebene 3, in der mittleren Schicht 4 und in der letzen Ebene 2 Neuronen enthält.

- *Das Gewicht der 3. Eingabezelle zum 4. Neuron der zweiten Ebene kann durch folgende Gewichtskonstante beschrieben werden:*
  `1:3:2:4` (Startebene:Startneuron:Zielebene:Zielneuron).
- *Das Gewicht der 4. Zelle der mittleren Ebene zum 1. Neuron der Ausgabeschicht kann durch folgende Gewichtskonstante beschrieben werden:*
  `2:4:3:1` (Startebene:Startneuron:Zielebene:Zielneuron).

4.2.7 Musterkonstanten vom Datentyp *pattern*

Eine Musterkonstante hat im ProSiNN–System den Datentyp *pattern*. Die Syntax einer entsprechenden Konstante ist wie folgt definiert:

pattern_const ::= non_zero_decimal_digit { decimal_digit }.

Bem.:

- Für eine Musterkonstante muß gelten: **pattern_const** $\in \{1, 2, \ldots,$ **nr_pattern**$\}$. **nr_pattern** ist die globale Variable, die die Anzahl der Trainingsmuster beschreibt (s.o.).
- Eine Musterkonstante darf in einem Integerausdruck (und umgekehrt) verwendet werden.
- Eine derartige Konstante ist nur für ein Trainingsmuster verwendbar (Testmuster müssen einen eigenen Bezeichner besitzen). Eine Musterkonstante wird in der Regel nur beim Netztraining interessant sein. Mit ihr und einem entsprechenden Präfix (siehe Unterabschnitt 4.3.4) kann auf musterinterne Werte zugegriffen werden. Aufbauend darauf läßt sich ein eigenes Lernverfahren entwickeln.

4.3 Präfixe für interne Werte

Um auf netzinterne Strukturen (wie z.B. ein Gewichts– oder ein Schwellenwert) zugreifen zu können, bietet ProSiNN Präfixe zur Referenz dieser Werte an. Diese Präfixe werden durch vordefinierte Bezeichner realisiert, die vor dem entsprechenden Gebilde positioniert werden. Je nach Präfixtyp lassen sich bestimmte interne Werte weiter verarbeiten oder neu setzen.

4.3.1 Der Netzebenenpräfix: nr_units

Mit Hilfe des Netzebenenpräfixes **nr_units** kann von jeder Netzebene die Anzahl der Neuronen bestimmt werden. Der angesprochene Wert hat den Datentyp *int* und läßt sich nicht direkt vom Benutzer ändern. Diese Struktur darf nur verwendet werden, wenn im System ein Neuronales Netz erzeugt oder geladen wurde. Die Syntax eines solchen Wertes sieht folgendermaßen aus:

nr_units_val ::= “*nr_units*“ “*:*“ **layer_const.**

Bem.:

- **layer_const** spezifiziert die Netzebene, dessen Neuronenanzahl dieser Ausdruck beschreibt. **layer_const** kann auch durch eine Variable vom Typ *layer* ersetzt werden (siehe Unterabschnitt 5.1.6).

- Diese Struktur ist sinnvoll, wenn ein eigener Lernalgorithmus programmiert werden soll. Für die Anzahl der Zellen in jeder Schicht braucht dann keine eigene Variable generiert zu werden.

Beispiel:

Im System befindet sich ein Netz, das in der Eingabeebene 3, in der mittleren Schicht 4 und in der letzen Ebene 2 Neuronen enthält.

- *Der Wert von* `nr_units:1` *beträgt 3* (es existieren 3 Neuronen in der 1. Schicht)
- *Der Wert von* `nr_units:2` *beträgt 4* (es existieren 4 Neuronen in der 2. Schicht)
- *Der Wert von* `nr_units:3` *beträgt 2* (es existieren 2 Neuronen in der 3. Schicht)

4.3.2 Die Neuronenpräfixe: theta , o_pj , delta_pj , delta_theta

Durch Verwendung dieser Präfixe kann der Anwender auf interne Werte eines Neurons zugreifen. Der Datentyp des angesprochenen Wertes ist *double*. Diese Struktur darf nur verwendet werden, wenn im System ein Neuronales Netz erzeugt oder geladen wurde. Die Syntax lautet:

unit_val ::= **unit_prefix** *“:“* **unit_const.**
unit_prefix ::= *“theta“* | *“o_pj“* | *“delta_pj“* | *“delta_theta“*.

Die folgende Auflistung beschreibt die Bedeutung der einzelnen Präfixe:

theta:

Mit diesem Präfix läßt sich der aktuelle Schwellenwert θ_j eines Neurons bearbeiten (siehe Abschnitt 3.2).

o_pj:

Der gegenwärtige Ausgabewert o_j eines Neurons kann mit diesem Präfix angesprochen werden (siehe Abschnitt 2.2).

delta_pj:

Mit diesem Präfix hat der Benutzer Zugriff auf das Fehlersignal δ_{pj} eines Neurons, das beim Training des letzten Muster entstanden ist (siehe Unterabschnitt 2.2.3).

delta_theta:

Mit Hilfe dieses Präfixes kann der Anwender den Änderungswert $\Delta\theta_j$ eines Neurons ansprechen (dieser Wert ist die analoge Übertragung der Änderung Δw_{ij} aus Gleichung 2.19 und 2.20 auf den Schwellenwert θ).

Bem.:

- **unit_const** beschreibt dabei das Neuron, dessen innere Werte verwendet oder geändert werden sollen. **unit_const** kann auch durch eine Variable vom Typ *unit* ersetzt werden (siehe Unterabschnitt 5.1.7).
- Diese Präfixe sind sinnvoll, wenn der Benutzer ein eigenes Lernverfahren programmieren will. Die Komponenten, auf die er durch **delta_pj** und **delta_theta** Zugriff hat, können dann als Zwischenspeicher für neuroneninterne Werte verwendet werden.
- Interne Werte eines Eingabeneurons sind nicht ansprechbar, da sie real nicht existieren!

Beispiel:

Im System befindet sich ein Netz, dessen 2. Neuron der 3. Ebene folgende interne Werte enthält (j bezeichne dieses Neuron):

Schwellenwert θ_j	: *3.4637539294*
Ausgabewert o_{pj} (Aktivierungswert nach letztem verarbeiteten Muster p)	: *0.1095685285*
Fehlersignal δ_{pj} (Fehlersignal nach letztem verarbeiteten Muster p)	: *−0.00029904*
Änderungswert $\Delta\theta_j$	: *0.0001335922*

Dann können mit den Neuronenpräfixen folgende Werte erhalten werden:

`theta:3:2`	(Präfix:Ebene:Neuron) ⟶ *3.4637539294*
`o_pj:3:2`	(Präfix:Ebene:Neuron) ⟶ *0.1095685285*
`delta_pj:3:2`	(Präfix:Ebene:Neuron) ⟶ *−0.00029904*
`delta_theta:3:2`	(Präfix:Ebene:Neuron) ⟶ *0.0001335922*

4.3.3 Die Gewichtspräfixe: w , delta_w

Diese beiden Präfixe ermöglichen den Zugriff auf gewichtsinterne Werte. Der durch die Verwendung der Präfixe erhaltene Datentyp ist *double*. Diese Struktur ist nur anwendbar, wenn ein Neuronales Netz im System vorhanden ist. Die Syntax hat folgendes Format:

weight_val ::= **weight_prefix** *“:“* **weight_const.**
weight_prefix ::= *“w“* | *“delta_w“*.

Die nachfolgende Auflistung beschreibt die Bedeutung der einzelnen Präfixe:

w:

Mit diesem Präfix läßt sich der aktuelle Gewichtswert w_{ij} einer Verbindung bearbeiten (siehe Abschnitt 2.2).

delta_w:

Durch diesen Präfix hat der Anwender Zugriff auf den Änderungswert eines Gewichtes Δw_{ij} (siehe Gleichung 2.19 und 2.20).

Bem.:

- **weight_const** beschreibt dabei das Gewicht, dessen innere Werte verwendet oder geändert werden sollen. **weight_const** kann auch durch eine Variable vom Typ *weight* ersetzt werden (siehe Unterabschnitt 5.1.8).
- Diese Präfixe sind sinnvoll, wenn der Benutzer ein eigenes Lernverfahren programmieren will. Die Komponente **delta_w** kann dann als Zwischenspeicher für gewichtsinterne Werte verwendet werden.
- Interne Werte eines Gewichtes, dessen Startebene die letzte Schicht ist, existieren nicht!

Beispiel:

Im System befindet sich ein Netz, dessen Gewicht vom 3. Neuron der 1. Ebene zur 6. Zelle der 2. Schicht folgende interne Werte enthält (w_{ij} bezeichne dieses Gewicht):

Aktuelles Gewicht w_{ij} : 3.6360925
Änderungswert Δw_{ij} : 8.842e-05

Nun können unter Zuhilfenahme der Präfixe folgende Werte erhalten werden:

`w:1:3:2:6` (Präfix:Startebene:Startneuron:Zielebene:Zielneuron) $\longrightarrow$ *3.6360925*
`delta_w:1:3:2:6` (Präfix:Startebene:Startneuron:Zielebene:Zielneuron) $\longrightarrow$ *8.842e-05*

4.3.4 Die Musterpräfixe: pattern_in , pattern_out

Mit diesen Präfixen lassen sich die internen Werte eines Musters ansprechen (sie sind unveränderbar[1]). Der Datentyp einer solchen Struktur ist *double*. Sie können aber nur verwendet werden, wenn im System Muster vorhanden sind. Die Syntax lautet folgendermaßen:

pattern_val ::= **pattern_prefix** *":"* **pattern_const** *":"* **pattern_index.**
pattern_prefix ::= *"pattern_in"* | *"pattern_out"*.
pattern_index ::= **non_zero_decimal_digit** { **decimal_digit** }.

Die folgende Auflistung beschreibt die Bedeutung der einzelnen Präfixe:

[1] Mit dem Menüsystem können allerdings interne Werte eines Musters verändert werden

pattern_in:

Mit diesem Präfix lassen sich die Netzeingabewerte eines Musters erhalten. Er kann bei Test– und Trainingsmuster benutzt werden (bei einem Testmuster muß dann für **pattern_const** der entsprechende Bezeichner des Musters stehen; siehe Abschnitt 10.2).

pattern_out:

Durch diesen Präfix hat der Anwender Zugriff auf die Netzausgabewerte eines Trainingsmusters (d.h. auf die Werte, die das Netz als Antwort auf ein Muster liefern soll).

Bem.:

- **pattern_out** darf nicht bei Testmustern verwendet werden, da sie keine vordefinierten Ausgabewerte enthalten. Wenn mit **pattern_const** ein Trainingsmuster angesprochen wird, dann muß für **pattern_index** gelten:
  pattern_index $\in \{1, 2, \ldots,$ **nr_out_units**$\}$. **nr_out_units** ist die globale Variable, die die Anzahl der Netzausgabezellen enthält.
- Wenn als Präfix **pattern_in** gewählt wurde, dann muß für **pattern_index** gelten: **pattern_index** $\in \{1, 2, \ldots,$ **nr_in_units**$\}$. **nr_in_units** ist die globale Variable, die die Anzahl der Netzeingabeneuronen enthält.
- Diese Präfixe sind sinnvoll, wenn der Benutzer ein eigenes Lernverfahren programmieren will. Er hat damit die Möglichkeit auf jedes Bestandteil eines Musters zuzugreifen.

Beispiel:

Im System befindet sich ein Netz, das zwei Eingabe– und ein Ausgabeneuron enthält. Das 3. im System vorhandene Muster definiert die Netzein– und Netzausgabewerte folgendermaßen:

1. Eingabewert : 0.2
2. Eingabewert : 0.7
1. Ausgabewert : 0.9

Durch Verwendung der Musterpräfixe kann nun auf diese Werte zugegriffen werden:

`pattern_in:3:1` (Eingabepräfix:Musternummer:Eingabeneuronen–Nummer) $\longrightarrow$ *0.2*
`pattern_in:3:2` (Eingabepräfix:Musternummer:Eingabeneuronen–Nummer) $\longrightarrow$ *0.7*
`pattern_out:3:1` (Ausgabepräfix:Musternummer:Ausgabeneuronen–Nummer) $\longrightarrow$ *0.9*

Bei allen anderen Werten von **pattern_index** *und Vertauschungen der beiden Bezeichner für* **pattern_prefix** *(die noch nicht in den Beispielen oben enthalten sind) erhält man Fehlermeldungen!*

4.4 Arithmetische Operatoren und vordefinierte Funktionen

In diesem Abschnitt soll auf die Verwendung der von ProSiNN unterstützten arithmetischen Operatoren und der vordefinierten Funktionen eingegangen werden. Zu jedem Operator werden die erlaubten Datentypen angegeben. Einzelheiten zu der praktischen Anwendung werden bei den ProSiNN–Kommandos erläutert, die diese Operatoren benutzen.

4.4.1 Addition und Subtraktion

Hier handelt es sich um die binären Operatoren "+" und "–", mit denen man Werte addieren und subtrahieren kann. Die Synax eines solchen Ausdrucks lautet:

add_exp ::= val_1 ("+" | "–") val_2.
val_1 , val_2 ::= „2 typkonforme Werte; siehe die nachfolgenden Bemerkungen".

Bem.:

1. **val_1** und **val_2** seien vom Datentyp *int*. Dann handelt es sich um die normalen Additions– und Subtraktionsoperatoren. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *int* (*layer* oder *pattern*), sowie eine Struktur, die den Netzebenenpräfix enthält.

2. **val_1** und **val_2** seien vom Datentyp *double*. Es handelt sich um die gewöhnlichen mathematischen Operatoren. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *double*, sowie eine Struktur, die die Neuronen–, Gewichts– oder Musterpräfixe benutzt.

3. Beim Datentyp *string* entspricht der "+"–Operator einer Konkatenation der beiden Zeichenketten (siehe Unterabschnitt 5.2.2 und 5.5.1).

4. Wenn einer der beiden Werte vom Typ *unit* ist, muß der andere vom Datentyp *int* sein (d.h. ein Ausdruck, der in der Bemerkung 1 genannt wurde). Durch diese Operation ergibt sich evtl. ein neues Neuron in derselben Ebene.

   "+"–**Operator:**

   Der Integerwert und die Neuronennummer (sie wird von der 1. Zelle bis zur letzten einer Ebene durchgezählt) werden addiert und ergeben die neue Neuronennummer in dieser Schicht (sofern diese Zelle existiert).

"– "–Operator:

Der Integerwert und die Neuronennummer werden in der angegebenen Reihenfolge subtrahiert und ergeben die neue Zellennummer in dieser Netzebene (sofern dieses Neuron existiert).

5. Wenn einer der beiden Werte vom Typ *weight* ist, muß der andere vom Datentyp *int* sein (d.h. ein Ausdruck, der in der Bemerkung 1 genannt wurde). Durch diese Operation ergibt sich evtl. ein neues Gewicht.

   "+ "–Operator:

   Der Integerwert und die Neuronennummer, bei der die Gewichtsverbindung endet, werden addiert und ergeben die neue Zellennummer in der Zielebene (sofern dieses Neuron existiert).

   "– "–Operator:

   Der Integerwert und die Neuronennummer, bei der die Gewichtsverbindung endet, werden in der angegebenen Reihenfolge subtrahiert und ergeben die neue Zellennummer in der Zielebene (sofern dieses Neuron existiert).

Beispiele:

Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, drei innere und ein Ausgabeneuron enthält (es handelt sich also um ein Netz mit drei Ebenen). Das Gewicht, das das 3. Neuron der mittleren Ebene mit der Ausgabezelle verbindet, hat den Wert 1.234. Zu jeder Alternative, die in den Bemerkungen genannt wurde, folgt jetzt mindestens ein Beispiel:

1. `33 + nr_in_units` ⟶ *35*
`nr_of_layers - 5` ⟶ *-2*
`nr_units:1 + 9` ⟶ *11*

2. `pi + 1.0` ⟶ *4.141592653589793*
`w:2:3:3:1 - 1.0` ⟶ *0.1234*

3. "`Dies ist eine` " + "`Konkatenation`" ⟶ *"Dies ist eine Konkatenation"*

4. `2:1 + 2` ⟶ *2:3*
`2:3 - 2` ⟶ *2:1*
`2 - 2:1` ⟶ *2:1*
`2 - 2:2` ⟶ *Fehler!*

5. `2:1:3:1 + 1` ⟶ *2:1:3:2*
`2:1:3:2 - 1` ⟶ *2:1:3:1*
`3 - 1:2:2:1` ⟶ *1:2:2:2*

4.4.2 Multiplikation und Division

Die binären Operationen "∗" und "/" werden für die Multiplikation und Division von Werten verwendet. Die Syntax einer solchen Operation ist folgendermaßen definiert:

mult_exp ::= val_1 ("∗" | "/") val_2.
val_1 , val_2 ::= „2 typkonforme Werte; siehe die nachfolgenden Bemerkungen".

Bem.:

1. **val_1** und **val_2** seien vom Datentyp *int*. Bei dem Operator "∗" handelt es sich um die normale Multiplikation. "/" führt eine ganzzahlige Division aus, bei der der Divisionsrest des Ergebnisses unterdrückt wird. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *int* (*layer* oder *pattern*), sowie eine Struktur, die den Netzebenenpräfix enthält.

2. **val_1** und **val_2** seien vom Datentyp *double*. Es handelt sich um die gewöhnlichen mathematischen Operatoren. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *double*, sowie eine Struktur, die die Neuronen–, Gewichts– oder Musterpräfixe benutzt.

Beispiele:

Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, drei innere und ein Ausgabeneuron enthält (es handelt sich also um ein Netz mit drei Ebenen). Das Gewicht, das das 3. Neuron der mittleren Ebene mit der Ausgabezelle verbindet, hat den Wert 1.234. Zu jeder Alternative, die in den Bemerkungen genannt wurde, folgen nun mindestens zwei Beispiele:

1. `33 * nr_in_units` ⟶ *66*
`10 / nr_of_layers` ⟶ *3*
`nr_units:1 * 9` ⟶ *18*

2. `pi * 10.0` ⟶ *31.415926535589793*
`w:2:3:3:1` */ 2.0* ⟶ *0.617*

4.4.3 Potenzen

Die binäre Operation „ˆ" wird zur Berechnung von Potenzen verwendet. Die Syntax lautet:

power_exp ::= val_1 „ˆ" val_2.
val_1 , val_2 ::= „2 typkonforme Werte; siehe die nachfolgenden Bemerkungen".

Bem.:

1. **val_1** und **val_2** seien vom Datentyp *int*. „^" führt die Berechnung der Potenz $\mathbf{val\_1}^{\mathbf{val\_2}}$ durch. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *int* (*layer* oder *pattern*), sowie eine Struktur, die den Netzebenenpräfix enthält.

2. **val_1** und **val_2** seien vom Datentyp *double*. Es handelt sich wieder um die gewöhnliche mathematische Operation. Für beide Werte können auch die folgenden Ausdrücke verwendet werden: Alle (vordefinierten) Konstanten und Variablen vom Datentyp *double*, sowie eine Struktur, die die Neuronen–, Gewichts– oder Musterpräfixe benutzt.

Beispiele:

Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, drei innere und ein Ausgabeneuron enthält (es handelt sich also um ein Netz mit drei Ebenen). Das Gewicht, das das 3. Neuron der mittleren Ebene mit der Ausgabezelle verbindet, soll den Wert 1.2 haben. Zu jeder Alternative, die in den Bemerkungen genannt wurde, folgen nun Beispiele:

1. `3 ^ nr_in_units` ⟶ *9*
`10 ^ nr_of_layers` ⟶ *1000*
`nr_units:1 ^ 4` ⟶ *16*

2. `pi ^ 0.0` ⟶ *1.0*
`w:2:3:3:1 ^ 2.0` ⟶ *1.44*

4.4.4 Vordefinierte mathematische Funktionen

In diesem Unterabschnitt sollen alle vordefinierten mathematischen Funktionen vorgestellt werden. Wenn der Definitions– oder der Wertebereich einer Funktion eingeschränkt ist, wird er im folgenden mit angegeben.

Bem.:

Alle Funktionen können als Argument einen Ausdruck vom Datentyp *int* oder *double* verarbeiten. Das Funktionsergebnis ist aber immer vom Typ *double*.

Quadratwurzel–Funktion:

sqrt():

Berechnet die Quadratwurzel eines Argumentes x, für das gelten muß: $x > 0.0$.

Trigonometrische Funktionen:

sin():

Berechnet den Sinus eines Argumentes, das im Bogenmaß angegeben ist.

cos():

Berechnet den Kosinus eines Argumentes, das im Bogenmaß angegeben ist.

tan():

Berechnet den Tangens eines Argumentes, das im Bogenmaß angegeben ist.

asin():

Berechnet den Arcussinus eines Argumentes x, für das gelten muß:
$x \in [-1.0, \ldots, 1.0]$. Der Funktionswert liegt im Intervall $[-\frac{\pi}{2}, \ldots, \frac{\pi}{2}]$.

acos():

Berechnet den Arcuskosinus eines Argumentes x, für das gelten muß:
$x \in [-1.0, \ldots, 1.0]$. Der Funktionswert liegt im Intervall $[0.0, \ldots, \pi]$.

atan():

Berechnet den Arcustangens eines Argumentes.
Der Funktionswert liegt im Intervall $[-\frac{\pi}{2}, \ldots, \frac{\pi}{2}]$.

Exponential– und logarithmische Funktionen:

exp():

Berechnet die Potenz eines Argumentes zur Basis e (Exponentialfunktion).

ln():

Berechnet den natürlichen Logarithmus eines Argumentes x, für das gelten muß: $x > 0.0$.

log():

Berechnet den dekadischen Logarithmus eines Argumentes x, für das gelten muß: $x > 0.0$.

Hyperbolische Funktionen:

sinh():

Berechnet den Sinushyperbolikus.

cosh():

Berechnet den Kosinushyperbolikus.

tanh():

Berechnet den Tangenshyperbolikus.

4.5 Mathematische Ausdrücke

ProSiNN bietet die Möglichkeit komplexe mathematische Ausdrücke zu formulieren. Diese entstehen durch Kombination der Operatoren, die in diesem Kapitel eingeführt worden sind. Mit dieser Erweiterung ist es möglich, zusammenhängende Ausdrücke prägnanter zu formulieren und neue Funktionen zu definieren.

Innerhalb eines zusammengesetzten Ausdrucks müssen die verwendeten Elemente in der Regel typkonform sein (Funktionsdefinitionen bilden hier eine Ausnahme; siehe Unterabschnitt 5.1.4). Es existieren zwei Datentypen für komplexe Ausdrücke: *int* und *double*.

Die allgemeine Syntax eines zusammengesetzten Ausdrucks lautet:

expression	::=	**term** { (“+“ | “–“) **term** }.	
term	::=	**factor** { (“*“ | “/“) **factor** }.	
factor	::=	**power** { „ˆ “ **power** }.	
power	::=	[“–“] **scalar**.	
scalar	::=	*“(“* **expression** *“)“* | **function**.	
function	::=	*“sin(“* **expression** *“)“*	| *“cos(“* **expression** *“)“* |
		“tan(“ **expression** *“)“*	| *“asin(“* **expression** *“)“* |
		“acos(“ **expression** *“)“*	| *“atan(“* **expression** *“)“* |
		“sinh(“ **expression** *“)“*	| *“cosh(“* **expression** *“)“* |
		“tanh(“ **expression** *“)“*	| *“sqrt(“* **expression** *“)“* |
		“ln(“ **expression** *“)“*	| *“log(“* **expression** *“)“* |
		var_or_const	| **own_function.**
var_or_const	::=	**„Eine typkonforme Konstante oder Variable (siehe Bem.)“**	
own_function	::=	**„Eine vom Benutzer definierte Funktion (siehe Bem.)“**	

Bem.:

- Für **var_or_const** sind Konstanten oder Variablen verwendbar, die alle den gleichen Datentyp haben. **own_function** muß eine Funktion sein, die ein Ergebnis vom selben Datentyp wie **var_or_const** liefert.

- Ein solcher zusammengesetzter Ausdruck darf nur benutzt werden, wenn ein Befehl von ProSiNN dies ausdrücklich gestattet. Da einige Kommandos nur Ausdrücke vom Typ *int* oder *double* gestatten, werden für die restlichen Kapitel folgende Konventionen getroffen:

  int_expression:
  Ein **int_expression** ist ein **expression**, bei dem alle Konstanten, Variablen und Funktionen vom Datentyp *int* sind.

  double_expression:
  Ein **double_expression** ist ein **expression**, bei dem alle Konstanten, Variablen und Funktionen vom Datentyp *double* sind.

Beispiele:

Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, drei innere und ein Ausgabeneuron enthält (es handelt sich also um ein Netz mit drei Ebenen). Das Gewicht, das das 3. Neuron der mittleren Ebene mit der Ausgabezelle verbindet, soll den Wert 1.2 haben.

int_expression:

```
((22 - 2) * 2 + 2^ 5) / 7                                              ⟶ 10
nr_of_layers^ 2 + (nr_in_units - 42) * 3                               ⟶ -111
(3 + 2) * 10 - nr_units:2 * (nr_units:1 + (nr_out_units + 1)^ 4)       ⟶ -4
```

double_expression:

```
ln(e) + cos(pi)                             ⟶ 0.0
cos(acos(1.0)) * (exp(1.0) - e + 4.0)       ⟶ 4.0
((w:2:3:3:1 + 0.8)^ 4.0 - 10.0)/2.0         ⟶ 3.0
```

Kapitel 5

Variablen und deren Bearbeitung

Dieses Kapitel beschäftigt sich mit Variablen, die der Benutzer direkt generieren und bearbeiten kann.

Im ersten Abschnitt wird der Befehl vorgestellt, der die verschiedenen Variablentypen erzeugt. Die Anweisung, die eine Modifikation des Variableninhaltes ermöglicht, wird danach beschrieben. Der dritte Abschnitt ist der Typumwandlung zwischen *int*– und *double*–Werten gewidmet. Danach wird erläutert, wie man eine Funktion(svariable) ändert oder ableitet. Kommandos zur Bearbeitung von *string*–Variablen werden anschließend eingeführt. Der vorletzte Abschnitt beschäftigt sich mit der Zusammenfassung einer Kommandosequenz zu einem Makro. Der Befehl, der eine Variable wieder freigibt, wird abschließend präsentiert.

Bem.:

Alle im System erzeugbaren Variablen sind **global**!

5.1 Variablendefinition: define

Das *define*–Kommando dient zur Generierung einer globalen Variablen. Die erlaubten Datentypen sind *int*, *double*, *string*, *layer*, *unit* und *weight*. Darüberhinaus können noch eindimensionale Felder und Funktionen auf den Datentypen *int* oder *double* definiert werden. Die allgemeine Syntax des Befehls lautet folgendermaßen:

define_command	::=	*“define“* **type** (**name** [*“=“* **init_val**]) | (**name** *“[“* **nr_elements** *“]“*) | (**name** *“(“* **var_list** *“)“* *“=“* **init_val**).
type	::=	*“int“* | *“double“* | *“array int“* | *“array double“* | *“function int“* | *“function double“* | *“string“* | *“layer“* | *“unit“* | *“weight“*.
name	::=	**„Bezeichner, der noch nicht definiert ist“.**
init_val	::=	**„Initialisierungswert“.**
nr_elements	::=	**non_zero_decimal_digit { decimal_digit }.**

var_list ::= **var_name** { “,“ **var_name** }.
var_name ::= **„Argumentname (gültiger Bezeichner)“.**

Bem.:

- **type** charakterisiert den der Variablen zugrundeliegenden Datentyp.
- **name** steht für den Bezeichner, mit dem auf die Variablen zugegriffen werden kann. Diese Zeichenfolge muß die in Abschnitt 4.1 genannten Konventionen einhalten.
- **init_val** darf oder muß bei einigen Variablentypen stehen und versorgt die Variable mit einem Initialisierungswert.
- **nr_elements** spezifiziert die Anzahl der Elemente bei der Definition einer Feldvariablen.
- **var_list** steht für eine durch Kommata getrennte Liste von Bezeichnern, die die Argumente bei einer Funktionsdefinition festlegen.

Die nach Variablentypen gegliederten Unterabschnitte beschreiben nun das Kommando syntaktisch und sematisch näher.

Bem.:

Für alle Beispiele in diesem Abschnitt sind folgende Voraussetzungen gültig:
Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, drei innere und ein Ausgabeneuron enthält. Das Gewicht, das vom 3. Neuron der mittleren Ebene zum 1. Neuron der letzten Schicht führt, hat den Wert 1.2.

5.1.1 Ganzzahlige Variablen: int

Eine *Integer*–Variable besitzt den Datentyp *int*. Die Syntax der Definition lautet:

define_int ::= *“define“ “int“* **name** [*“=“* **int_expression**].

Bem.:

Für **int_expression** kann ein beliebiger Integerausdruck (siehe Abschnitt 4.5) verwendet werden. Fehlt dieser optionale Parameter, so wird die Variable **name** mit dem Wert 0 initialisiert.

Beispiele:

- `define int i`

  Diese Zeile definiert eine Integervariable `i` *mit dem Initialisierungswert* 0
- `define int j = 2 * nr_units:2 + 1`

  Hier wird eine Integervariable `j` *mit dem Wert* 7 *erzeugt*

5.1.2 Gleitkomma–Variablen: double

Eine *Double*–Variable besitzt den Datentyp *double*. Die entsprechende Befehlssyntax ist:

define_double ::= *"define"* *"double"* **name** [*"="* **double_expression**].

Bem.:

Für **double_expression** kann ein beliebiger Gleitkommaausdruck (siehe Abschnitt 4.5) verwendet werden. Fehlt diese optionale Angabe, so wird die Variable **name** mit dem Wert 0.0 initialisiert.

Beispiele:

- `define double x`

  Diese Zeile generiert eine Doublevariable `x` *mit dem Initialisierungswert* 0.0

- `define double y = 2.0 * w:2:3:3:1 + 0.6`

  Hier wird eine Doublevariable `y` *mit dem Wert* 3.0 *erzeugt*

5.1.3 Felder: array int und array double

ProSiNN bietet die Möglichkeit, Felder auf den Datentypen *int* und *double* zu definieren. Jedes dieser Elemente kann in einem typkonformen Ausdruck verwendet werden. Die Syntax einer Feldvariablendefininition lautet:

define_array ::= *"define"* *"array"* **type name** *"["* **nr_elements** *"]"*.
type ::= *"int"* | *"double"*.
nr_elements ::= **int_expression.**

Bem.:

- **type** charakterisiert den der Feldvariablen zugrundeliegenden Datentyp.
- **nr_elements** legt die Anzahl der Feldelemente fest.
- Alle Elemente eines Feldes werden mit 0 bzw. 0.0 initialisiert.
- Will man auf ein Feldelement zugreifen, so muß es folgendermaßen angesprochen werden:

  array_val ::= **name** *"["* **index** *"]"*.
  index ::= **int_expression.**

  name ist der Bezeichner, mit dem die Feldvariable erzeugt wurde. **index** spezifiziert den Zugriffsschlüssel eines Feldelementes. Für **index** muß gelten:
  $\mathbf{index} \in \{0, \ldots, \mathbf{nr\_elements} - 1\}$.

Beispiele:

- *Durch das folgende Kommando wird ein Array mit 7 Integerelementen generiert:*

  ```
  define array int f[7]
  ```

  Jedes dieser Elemente kann in einem Integerausdruck verwendet werden:
  `f[0]`, `f[1]`, ..., `f[5]` *und* `f[6]` *repräsentieren die Werte (und die interne Integerstruktur), die die Elemente mit dem Index* 0, 1, 2, 3, 4, 5 *und* 6 *enthalten.*

- *Der nachfolgende Befehl erzeugt ein Fließkommaarray mit 11 Elementen, die alle mit* 0.0 *initialisiert werden:*

  ```
  define array double f[3 * nr_of_layers + 2]
  ```

  Jedes Element läßt sich ändern oder in einem Gleitkommaausdruck verwenden.

5.1.4 Funktionen: function int und function double

Das *define*–Kommando ermöglicht dem Systembenutzer neue Funktionen zu generieren. Eine Funktion ist dabei eine Abbildung einer oder mehrerer Veränderlicher vom Datentyp *int* oder *double* auf einen Wert vom Typ *int* oder *double*. Die Syntax einer solchen Definition ist:

define_function	::=	*"define"* **type name** *"("* **var_list** *")"* *"="* **funct_expression.**
type	::=	*"function int"* | *"function double"*.
name	::=	**„Bezeichner, der noch nicht definiert ist".**
var_list	::=	**var_name** { *","* **var_name** }.
var_name	::=	**„Name (Funktionsvariable)".**
funct_expression	::=	**term** { ("+" | "–") **term** }.
term	::=	**factor** { ("*" | "/") **factor** }.
factor	::=	**power** { „^ " **power** }.
power	::=	["–"] **scalar.**
scalar	::=	*"("* **funct_expression** *")"* | **function.**
function	::=	*"sin("* **funct_expression** *")"* |
		"cos(" **funct_expression** *")"* |
		"tan(" **funct_expression** *")"* |
		"asin(" **funct_expression** *")"* |
		"acos(" **funct_expression** *")"* |
		"atan(" **funct_expression** *")"* |
		"sinh(" **funct_expression** *")"* |
		"cosh(" **funct_expression** *")"* |
		"tanh(" **funct_expression** *")"* |
		"sqrt(" **funct_expression** *")"* |
		"ln(" **funct_expression** *")"* |

		"log(" **funct_expression** *")"* |
		var_or_const |
		own_function |
		var_name.
var_or_const	::=	**„Eine Konstante oder Variable (int oder double)"**
own_function	::=	**„Eine vom Benutzer definierte Funktion (vom Typ int oder double). Siehe die nachfolgenden Bemerkungen."**

Bem.:

- **type** charakterisiert den Datentyp der Funktionswerte.
- **var_list** ist die durch Kommata getrennte Liste der Funktionsargumente. Diese Namen sind nur lokal bei der Definition bekannt. D.h. es können beliebige — auch vordefinierte — Bezeichner verwendet werden; die ursprüngliche Bedeutung wird kurzfristig verdeckt!
- Bei der Definition können die beiden Datentypen *int* und *double* beliebig miteinander vermischt werden.
- Indem z.B. Operationen auf Konstanten zusammengefaßt werden, versucht ProSiNN **funct_expression** zu vereinfachen. Dabei werden aber Teilausdrücke nicht permutiert, so daß eine lokale und keine globale Vereinfachung stattfindet.
- Bei der Erzeugung einer Integerfunktion wird von allen vorkommenden Konstanten vom Typ *double* der Nachkommaanteil abgeschnitten. Analog gilt bei einer Funktionsauswertung innerhalb der Definition, daß alle Teilfunktionen vom Typ *double* und deren Werte intern auf den nächsten kleineren Integer abgebildet werden (bei negativen Zahlen werden sie auf den nächsten größeren Integer transformiert); siehe die nachfolgenden Beispiele.
- Die Funktionsauswertung an einer beliebigen Stelle muß folgender Syntax genügen:

  funct_val ::= **name** *"("* **value_list** *")"*.
  value_list ::= **expression** { *","* **expression** }.

  Bei **expression** handelt es sich um den in Abschnitt 4.5 definierten Ausdruck, der etwas modifiziert benutzt wird. Hier brauchen keine typkonformen Variablen verwendet zu werden; es lassen sich also beliebige *int*– und *double*–Werte miteinander vermischen.
- Für **own_function** gilt die gleiche Syntax wie bei einer Funktionsauswertung; es dürfen aber zusätzlich noch alle in der Definition vorkommenden Funktionsargumente verwendet werden!

Beispiele:

- *Um die 2. binomische Formel als Integerfunktion* `bin` *zu definieren, muß man folgendes Kommando eingeben:*

```
define function int bin(a,b) = (a - b)^2
```

Zur Funktionsauswertung an der Stelle (0,1) muß `bin(0,1)` *verwendet werden. Wenn der Benutzer den Ausdruck* `bin(3.99,-.01)` *benutzt, erhält er als Ergebnis* `9`. *Der Grund liegt in der Äquivalenz von* `bin(3.99,-.01)` *und* `bin(3,0)`*! Wäre die Funktion dagegen als* double *definiert worden, so würde das Ergebnis* `16` *lauten!*

- *Zwei Besonderheiten zeigt auch die nächste Funktionsdefinition:*

```
define function int cos2(stdout) = cos(stdout)
```

Zum einen wurde hier ein bereits vordefinierter Bezeichner verwendet. Die ursprüngliche Bedeutung der Konstanten **stdout** *ist bis nach der Definition aufgehoben!*
Die andere Besonderheit tritt bei der Auswertung der Funktion `cos2` *auf. Nur die Funktionsargumente aus dem Bereich* $(-1, \ldots, 1)$ *liefern als Funktionswert* 1. *Dieses Phänomen tritt durch das Abschneiden der Nachkommastellen auf.* cos2 *verwendet ganze Zahlen als Eingabewerte; folglich werden nur Werte im oben genannten Intervall als Funktionswert* 1 *liefern (das System betrachtet z.B.* `cos2(2`π`)` *als* `cos2(6)`*).*

- *Die folgende Definition erzeugt eine zweidimensionale Funktion, die die Länge eines Vektors im entsprechenden Raum berechnet:*

```
define function double length(a,b) = sqrt(a^2 + b^2)
```

Im System existieren außerdem noch zwei Felder `A` *und* `B` *beliebigen Typs (int oder double), deren Elementanzahl jeweils größer als 2 ist. Betrachtet man die beiden Felder als Vektoren, so kann ihr auf die xy–Ebene projizierter Summenvektor durch den folgenden Ausdruck berechnet werden:*
`length(A[0] + B[0],A[1] + B[1])`.

- *Die vollständige Substitution in einer Funktionsdefinition zeigt das letzte Beispiel:*

```
define function double f(x) = length(x + 1,0.0) + cos2(x)
```

Durch diesen Befehl ist die folgende Funktion definiert worden:
$f(x) = \sqrt{(x+1)^2} + \cos(x)$.
Zu bemerken ist noch, daß die Substitution real ausgeführt wird. Bei einer Auswertung von `f` *mit beliebigem Argument findet kein Aufruf der Funktionen* `length` *und* `cos2` *statt (in* `f` *ist* `cos2` *zu der reellen Funktion* `cos` *geworden)!*

5.1.5 String–Variablen: string

Eine *String*–Variable besitzt den Datentyp *string*. Die Syntax der Definition lautet:

define_string ::= *“define“ “string“* **name** [*“=“* **string_val**].

Bem.:

Für **string_val** kann eine beliebige Konstante (siehe Unterabschnitt 4.2.3) oder Variable vom Typ *string* verwendet werden. Fehlt dieser optionale Parameter, so wird die Variable **name** mit dem *leeren String* initialisiert.

Beispiele:

- `define string str1`

  Diese Zeile generiert die leere Stringvariable `str1`

- `define string str2 = "Ich bin der Wert einer Stringvariablen"`

  Hier wird eine Stringvariable `str2` *erzeugt, die folgenden Inhalt enthält:*
  ″`Ich bin der Wert einer Stringvariablen`″

5.1.6 Netzebenen–Variablen: layer

Eine *Layer*–Variable basiert auf dem Datentyp *layer*. Die Befehlssyntax ist:

define_layer ::= *"define"* *"layer"* **name** *"="* **int_expression.**

Bem.:

- Eine *layer*–Variable kann nur dann definiert werden, wenn intern ein Neuronales Netz vorhanden ist.
- *layer*–Variablen können simultan bei der Erzeugung Neuronaler Netze generiert werden; siehe Abschnitt 9.1.
- Für den Wert von **int_expression** muß gelten:
  int_expression $\in \{1, 2, \ldots,$ **nr_of_layers**$\}$. Dabei bezeichnet **nr_of_layers** die globale Variable, die die Anzahl der Netzebenen enthält.

Beispiele:

- `define layer l1 = 1 + 2 - 4 * 7 + 26`

  Dieser Befehl erzeugt eine Layervariable, die auf die 1. Netzschicht verweist.

- `define layer l2 = nr_of_layers`

  Diese Variable bezeichnet die letzte Ebene im vorliegenden Netz. Hier ist das die 3. Schicht.

5.1.7 Neuronen–Variablen: unit

Eine *Unit*–Variable besitzt den Datentyp *unit*. Die Syntax der Definition lautet:

define_unit ::= *"define"* *"unit"* **name** *"="* **unit_val.**

Bem.:

- Eine *unit*–Variable kann nur dann definiert werden, wenn intern ein Neuronales Netz existiert.
- **unit_val** muß eine Konstante oder Variable vom Datentyp *unit* sein; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!
- Es kann keine Variable erzeugt werden, deren Neuron in der ersten Ebene liegt!

Beispiele:

- `define unit u1 = 2:2`

  Dieses Kommando erzeugt eine Unit–Variable, die auf das 2. Neuron in der mittleren Schicht verweist.
- `define unit u2 = nr_of_layers:1`

  Diese Variable bietet anschließend eine Referenz auf das 1. Neuron der letzten Ebene.

5.1.8 Gewichts–Variablen: weight

Eine *Weight*–Variable basiert auf dem Datentyp *weight*. Die Kommandosyntax ist:

define_weight ::= *"define"* *"weight"* **name** *"="* **weight_val.**

Bem.:

- Eine *weight*–Variable kann nur dann definiert werden, wenn intern ein Neuronales Netz vorhanden ist.
- **weight_val** muß eine Konstante oder Variable vom Datentyp *weight* sein; dabei müssen Konstanten die in Unterabschnitt 4.2.6 auf Seite 58 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole $\textbf{unit\_const}_{start\_layer}$ (für die Startebene) und $\textbf{unit\_const}_{destination\_layer}$ (für die Zielebene) können auch Variablen vom Typ *unit* sein. Anstatt eine *unit*–Variable zu benutzen, kann auch eine Form gewählt werden, die im letzten Unterabschnitt erläutert wurde. Diese Komponenten dürfen aber nicht aus zusammengesetzten Ausdrücken aufgebaut sein!

- Es kann keine Variable erzeugt werden, dessen Startneuron in der letzten Ebene liegt!

Beispiele:

- `define weight w1 = 2:2:3:1`
  `define weight w2 = 2:2:nr_of_layers:1`

  Beide Befehle erzeugen Weight–Variablen, die dasselbe Gewicht bezeichnen.

- *Es seien folgende Variablen gegeben:* `l` *sei eine Layer–Variable, die auf die 2. Netzschicht verweist,* `i` *sei eine Integervariable, deren Wert* 2 *beträgt und* `u` *sei der Bezeichner einer Unit–Variablen, mit der das einzige Ausgabeneuron ansprechbar ist.*

  `define weight w3 = l:i:u`

  Diese Variable bietet nun eine Referenz zu dem gleichen Gewicht, auf das `w1` *und* `w2` *verweisen.*

5.2 Neuen Wert an eine Variable zuweisen: set

Das *set*–Kommando wird benötigt, um Variablen vom Typ *int*, *double*, *array int*, *array double*, *string*, *layer*, *unit* und *weight* neue Werte zuzuweisen. Die allgemeine Syntax des Befehls lautet:

set_command	::=	*“set“* **var_name** [*“[“* **index** *“]“*] *“=“* **val.**
var_name	::=	**„Name einer Variablen, die neu gesetzt werden soll“.**
index	::=	**int_expression.**
val	::=	**expression** |
		string_val [*“+“* **string_val**] |
		unit_expression |
		weight_expression.

Bem.:

- Bei der Zuweisung eines neuen Wertes an ein Feldelement wird mit **index** diejenige Komponente ausgewählt, die einen neuen Wert erhalten soll.
- **val** repräsentiert den neuen Wert der Variablen:
  - **expression** steht für einen typkonformen Ausdruck, der in Abschnitt 4.5 definiert wurde.
  - Mit **string_val** wird eine Konstante oder Variable vom Typ *string* spezifiziert.
  - Durch **unit_expression** und **weight_expression** wird eine Additions– oder Subtraktionsoperation mit *unit*– oder *weight*–Operanden definiert; siehe Bemerkung 4 und 5 in Unterabschnitt 4.4.1.

In den nachfolgenden Unterabschnitten werden die einzelnen Zuweisungsbefehle — nach Variablentypen gegliedert — vorgestellt.

5.2.1 Variablen vom Typ: int , double , array int und array double

Um einer *int*– oder *double*–Variablen einen neuen Wert zuzuweisen, muß man folgende Syntax einhalten:

set_int_double ::= *"set"* **var_name** *"="* **expression.**
var_name ::= **„Name der Variablen, die neu gesetzt werden soll".**
expression ::= **int_expression | double_expression.**

Bem.:

- Der Ausdruck muß typkonform sein. D.h. nur ein **int_expression** darf einer Integervariablen und nur ein **double_expression** einer Doublevariablen zugewiesen werden (siehe Abschnitt 4.5).
- Mit diesem Kommando dürfen auch den in Abschnitt 4.1 genannten vordefinierten Variablen neue Werte zugewiesen werden. Der *set*–Befehl bricht mit einer Fehlermeldung ab, wenn der neue Wert den dort genannten Bedingungen widerspricht (z.B. wenn **error_limit** auf einen Wert kleiner 0 gesetzt werden soll).

Um einem Element einer *int*– oder *double*–Feldvariablen einen neuen Wert zuzuweisen, muß der *set*–Befehl folgender Syntax genügen:

set_int_double_array ::= *"set"* **var_name** *"["* **index** *"]"* *"="* **expression.**
var_name ::= **„Name der Feldvariablen, die neu gesetzt werden soll".**
index ::= **int_expression.**
expression ::= **int_expression | double_expression.**

Bem.:

- **index** spezifiziert das Element, das einen neuen Wert bekommen soll.
- Der Ausdruck muß typkonform sein. Einem Integerfeld darf nur ein **int_expression** und einem Doublefeld nur ein **double_expression** zugewiesen werden (siehe Abschnitt 4.5).

Beispiele:
Im System befinden sich zwei Feldvariablen mit jeweils 10 Elementen: `ia` *sei ein Integer– und* `da` *ein Doublearray; außerdem existiert eine vom Anwender definierte Variable* `d` *vom Typ double.*

- `set d = cos(pi/2.0)^2.0 + sin(pi/2.0)^2.0`

  Dieser Befehl setzt den Wert der Variablen `d` *auf* 1.0.

- ```
  set ia[5] = (2^5 - 16)/2 -2
  set da[ (ia[5] - 3) * 3 ] = (d / 2.0) + 7.5
  ```

  Die erste Anweisung setzt das 6. Element der Feldvariablen `ia` *auf den Wert* 6. *Das darauf folgende Kommando weist der 10. Komponente von* `da` 8.0 *zu.*
  ```

5.2.2 Variablen vom Typ: string

Um einer *String*–Variablen einen neuen Wert zuzuweisen, muß die folgende Syntax eingehalten werden:

set_string ::= *“set“* **var_name** *“=“* **string_val [** *“+“* **string_val].**
var_name ::= „Name der Variablen, die neu gesetzt werden soll“.

Bem.:

- **string_val** bezeichnet eine Konstante oder Variable vom Typ *string*.
- Bei der Verwendung des Operators *“+“* wird der Variablen **var_name** die Konkatenation der beiden Operandenstrings zugewiesen.

Beispiele:
Im System befinden sich 2 Stringvariablen `str1` *und* `str2`.

- `set str1 = "Hallo!"`
  `set str2 = str1`

  Diese beiden Befehle setzen die Variablen `str1` *und* `str2` *jeweils auf die Zeichenkette ”*`Hallo!`*”.*

- `set str1 = str1 + str2`

  Das Kommando konkateniert die beiden Stringvariablen `str1` *und* `str2` *und weist das Ergebnis der Variablen* `str1` *zu.* `str1` *enthält anschließend ”*`Hallo!Hallo!`*”.*

5.2.3 Variablen vom Typ: layer

Um einer *Layer*–Variablen einen neuen Wert zuzuweisen, ist die folgende Syntax einzuhalten:

set_layer ::= *“set“* **var_name** *“=“* **int_expression.**
var_name::= „Name der Variablen, die neu gesetzt werden soll“.

Bem.:

int_expression spezifiziert die Nummer der Netzschicht, die **var_name** zugewiesen werden soll. Für **int_expression** muß gelten: **int_expression** $\in \{1, 2, \ldots,$ **nr_of_layers**$\}$. Dabei bezeichnet **nr_of_layers** die globale Variable, die die Anzahl der Netzebenen enthält.

Beispiele:
Im System befindet sich ein Neuronales Netz mit drei Ebenen und zwei Layervariablen `l1` *und* `l2`.

- `set l1 = 2 * 2 - 3`
  `set l2 = nr_of_layers`

  `l1` *verweist nun auf die erste und* `l2` *auf die letzte Netzschicht.*

- `set l1 = (l1 + (l2 - l1)) / 2 + 1`

  Nun bildet `l1` *eine Referenz zu der mittleren Netzebene.*

5.2.4 Variablen vom Typ: unit

Um einer *Unit*–Variablen einen neuen Wert zuzuweisen, muß das *set*–Kommando folgende Syntax befolgen:

set_unit	::= “*set*“ **var_name** “=“ **unit_expression.**
var_name	::= **„Name der Variablen, die neu gesetzt werden soll“.**
unit_expression	::= **unit_val** | (**unit_val op int_val**) | (**int_val op unit_val**).
op	::= “+“ | “–“.

Bem.:

- **unit_val** spezifiziert eine Konstante oder Variable vom Typ *unit*; dabei müssen Konstanten die in Unterabschnitt 4.2.5 auf Seite 57 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole **layer_const** und **unit_number** können auch Variablen vom Typ *layer* oder *int* sein. Es dürfen aber keine zusammengesetzten Ausdrücke für **layer_const** und **unit_number** verwendet werden!

- **int_val** bezeichnet eine Konstante oder Variable vom Datentyp *int*.

- Kommt der Operator **op** nicht im Zuweisungsbefehl vor, so wird **unit_val** der Variablen **var_name** zugewiesen.

- Wird der Operator **op** verwendet, dann ergibt sich durch diese Operation ein neues Neuron, das in derselben Ebene liegt, in der sich auch **unit_val** befindet. Die Bedeutung ist folgendermaßen:

  “+“**–Operator:**

  > **int_val** und die Neuronennummer (sie wird von der ersten bis zur letzten Zelle einer Ebene durchgezählt) von **unit_val** werden addiert und ergeben die neue Neuronennummer in dieser Schicht. Auf dieses Neuron verweist nach der Operation **var_name**.

  “–“**–Operator:**

  > **int_val** und die Neuronennummer von **unit_val** werden in der angegebenen Reihenfolge subtrahiert und ergeben die neue Zellennummer in dieser Netzebene. Nach der Operation bildet **var_name** eine Referenz auf dieses Neuron.

Beispiele:
Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, 3 innere und ein Ausgabeneuron enthält; außerdem existieren zwei Unitvariablen `u1` *und* `u2`*, sowie eine Layervariable* `l`*, die auf die 2. Netzschicht verweist.*

- `set u1 = 2:1`
  `set u2 = l:2`

  Nach der Zuweisung bildet `u1` *eine Referenz auf das erste Neuron der 2. Ebene und* `u2` *verweist auf die 2. Zelle in derselben Schicht.*

- `set u1 = u1 + 2`
  `set u2 = 3 - u2`

  `u1` *zeigt nun auf das 3. Neuron der mittleren Ebene und* `u2` *auf die erste Zelle der gleichen Schicht.*

5.2.5 Variablen vom Typ: weight

Um einer *Weight*–Variablen einen neuen Wert zuzuweisen, muß das *set*–Kommando die folgende Syntax einhalten:

set_weight	::=	"*set*" **var_name** "=" **weight_expression.**
var_name	::=	**„Name der Variablen, die neu gesetzt werden soll".**
weight_expression	::=	**weight_val** | (**weight_val op int_val**)
		| (**int_val op weight_val**).
op	::=	"+" | "–".

Bem.:

- **weight_val** spezifiziert eine Konstante oder Variable vom Typ *weight*; dabei müssen Konstanten die in Unterabschnitt 4.2.6 auf Seite 58 beschriebenen Bedingungen erfüllen. Die dort genannten Nichtterminalsymbole $\textbf{unit\_const}_{start\_layer}$ (für die Startebene) und $\textbf{unit\_const}_{destination\_layer}$ (für die Zielebene) können auch Variablen vom Typ *unit* sein. Anstatt einer *unit*–Variablen ist auch eine zusammengesetzte Form aus *layer*– oder *int*–Parametern verwendbar. Diese dürfen dann aber nur noch aus Konstanten oder Variablen bestehen!

- **int_val** bezeichnet eine Konstante oder Variable vom Datentyp *int*.

- Kommt der Operator **op** nicht im Zuweisungsbefehl vor, so wird **weight_val** der Variablen **var_name** zugewiesen.

- Wird der Operator **op** verwendet, dann ergibt sich durch diese Operation ein neues Gewicht, das dieselbe Startebene, aber evtl. ein neues Zielneuron hat. Die Semantik lautet folgendermaßen:

"+"–Operator:

> **int_val** und die Neuronennummer von **weight_val**, bei der die Verbindung endet, werden addiert und ergeben die neue Zellennummer in der Zielschicht. Auf dieses Gewicht verweist nach der Operation **var_name**.

"–"–Operator:

> **int_val** und die Neuronennummer von **weight_val**, bei der die Verbindung endet, werden in der angegebenen Reihenfolge subtrahiert und ergeben die neue Zellennummer in der Zielebene. Nach der Operation bildet **var_name** eine Referenz auf dieses Gewicht.

Beispiele:
Im System befindet sich ein Neuronales Netz, das zwei Eingabe–, 3 innere und ein Ausgabeneuron enthält. Weiterhin sind zwei Gewichtsvariablen `w1` *und* `w2` *definiert. Eine Layervariable* `l` *verweise auf die 2. Netzschicht und eine Unitvariable* `u` *bilde eine Referenz zur letzten Zelle der mittleren Ebene.*

- ```
  set w1 = 2:1:3:1
  set w2 = 1:2:1:3
  ```
  Nach der Zuweisung bildet `w1` *eine Referenz auf das Gewicht, das vom 1. Neuron der 2. Ebene zur Ausgabezelle verläuft.* `w2` *bezeichnet die Verbindung, die von der 2. Eingabezelle zum letzten Neuron in der mittleren Schicht geht.*

- ```
  set w1 = 1:1:2:1 + 2
  set w2 = 4 - 1:1:u
  ```
  `w1` *verweist nun auf das Gewicht, das vom 1. Eingabeneuron zur 3. Zelle der 2. Ebene verläuft, und* `w2` *repräsentiert im Anschluß die Verbindung vom 1. Eingabeneuron zur 1. Zelle der mittleren Schicht.*

5.3 Typumwandlungen

In diesem Abschnitt sollen zwei Kommandos vorgestellt werden, die Umwandlungen zwischen den Datentypen *int* und *double* durchführen. Der umgewandelte Wert wird dann einer Variablen zugewiesen.

5.3.1 double → int : int_of

Der Befehl *int_of* wandelt einen Ausdruck vom Typ *double* in einen Integerwert um und weist ihn einer *int*–Variablen zu. Die Syntax des Befehls lautet:

int_of_command	::= *"int_of("* **double_expression** *")"* *"to"* **int_var.**
int_var	::= **„Name einer Integervariablen".**

Bem.:

- **int_var** darf auch ein Element eines Integerfeldes sein.
- Von **double_expression** wird bei der Umwandlung der Nachkommaanteil abgeschnitten. Er wird intern auf den nächsten kleineren Integer abgebildet (bei negativen Zahlen wird der Wert auf den nächsten größeren Integer transformiert).
- Man beachte, daß es zu einer Bereichsüberschreitung kommen kann. In diesem Fall wird die Anweisung mit einem Fehler abgebrochen!
- Der Anwendungsbereich dieses Kommandos ist die Typkonvertierung in einem Simulationsprogramm.

Beispiele:
Im System befinden sich eine Integerfeldvariable `ia` *mit 10 Elementen und eine Integervariable* `i`.

- ```
  int_of(cos(pi/2.0)^2.0 + sin(pi/2.0)^2.0) to i
  int_of(pi) to ia[i]
  ```
  Der erste Befehl weist der Variablen `i` *den Wert* 1 *zu. Da* `i` 1 *enthält, wird* `ia[1]` *auf* 3 *gesetzt.*
- ```
  int_of(1.0e300) to i
  ```
  Dieses Kommando führt zu einer Fehlermeldung, da der Integerwertebereich verlassen wird.

5.3.2 int → double : double_of

Der Befehl *double_of* wandelt einen Ausdruck vom Typ *int* in einen Gleitkommawert um und weist ihn einer *double*–Variablen zu. Die Syntax des Befehls ist:

double_of_command ::= *"double_of("* **int_expression** *")"* *"to"* **double_var.**
int_var ::= **„Name einer Doublevariablen".**

Bem.:

- **double_var** darf auch ein Element eines Doublefeldes sein.
- Der Anwendungsbereich dieses Befehls liegt in der Verwendung zur Typkonvertierung in einem ProSiNN–Programm.

Beispiele:
Im System befinden sich eine Doublefeldvariable `da` *mit 10 Elementen und eine Doublevariable* `d`.

```
double_of( (on + 3) * 7 + 2 ) to d
double_of(2^10) to da[5]
```

Der erste Befehl weist der Variablen `d` *den Wert* 30.0 *zu.* `da[5]` *enthält nun* 1024.0.

5.4 Funktionsvariablen bearbeiten

In diesem Abschnitt soll beschrieben werden, wie eine Funktionsvariable geändert oder abgeleitet werden kann.

5.4.1 Funktion ändern

Zur Modifikation einer Funktionsvariablen gibt es kein eigenständiges Kommando, aber zwei Möglichkeiten, mit denen dies durchgeführt werden kann:

1. Der bequemste und einfachste Weg führt über das Menüsystem. In der Menüleiste *Benutzerwerte bearbeiten* existiert der Menüpunkt *Variablen bearbeiten*. Dieser Auswahlpunkt bietet einen speziellen Editor an, der das Ändern einer Funktion erlaubt; siehe Abschnitt 18.4.

2. Die andere Alternative gliedert sich in zwei Schritte (`funct_name` sei der Name der zu modifizierenden Funktion):

   - Man lösche zuerst die Funktion `funct_name` mit dem Befehl *free_var* (siehe Abschnitt 5.7):

     ```
     free_var funct_name
     ```

   - Im Anschluß definiere man die *geänderte* Funktion unter dem gleichen Namen:

     ```
     define function funct_name(...
     ```

5.4.2 Funktion ableiten: derivation_of

ProSiNN besitzt die Fähigkeit, mit dem Befehl *derivation_of* eine Funktion nach einer beliebigen Variablen abzuleiten und als neue Funktion des gleichen Typs zu speichern; mit diesem Kommando sind also alle *partiellen Ableitungen* generierbar. Die Syntax des Befehls lautet:

derivation_of_command	::=	*“derivation_of“* **funct_name** *“in“* **int_expression** *“to“* **derivated_funct_name.**
funct_name	::=	**„Name der abzuleitenden Funktion“.**
derivated_funct_name	::=	**„Bezeichner der abgeleiteten Funktion“.**

Bem.:

- **funct_name** ist der Bezeichner der abzuleitenden Funktion.

- **int_expression** spezifiziert die Nummer der Funktionsvariablen, nach der abgeleitet werden soll. Die einzelnen Funktionsvariablen werden dabei von 1 bis zur *Anzahl der Funktionsvariablen* durchnummeriert. Folglich muß für **int_expression** gelten: **int_expression** $\in \{1, \ldots, \text{Anzahl der Funktionsvariablen}\}$.

- Das Kommando speichert die abgeleitete Funktion immer in einer neu erzeugten Variablen. Der angegebene Bezeichner **derivated_funct_name** darf deshalb noch nicht definiert worden sein!

- Der Befehl erzeugt eine Funktion vom gleichen Datentyp.

- Nachdem ProSiNN die Funktion abgeleitet hat, wird ein Algorithmus zur Vereinfachung der entstandenen Terme ausgeführt. Er ist aber nicht in der Lage, einzelne Terme so zu permutieren, daß weitere Verbesserungen ermöglicht werden. Der Benutzer sollte deshalb u.U. *manuell* eine Vereinfachung der abgeleiteten Funktion durchführen (z.B. durch das Menüsystem; siehe Punkt 1 im letzten Unterabschnitt). Die Gründe, die für eine solche Vorgehensweise sprechen, sind:
  - Eine vereinfachte Funktion hat ein besseres Laufzeitverhalten. Beim Auswerten eines Funktionswertes brauchen weniger Berechnungen durchgeführt zu werden.
  - Es können real nicht vorhandene singuläre Stellen auftreten. Wird z.B. die Ableitung der Funktion $f(x) = x^2$ generiert, so erzeugt ProSiNN intern die Funktion $f'(x) = x^2 * 1/x * 2$ anstatt der vereinfachten Lösung $f'(x) = 2x$. Hier gibt es bei der Funktionsauswertung an der Stelle $x = 0$ eine real nicht vorhandene singuläre Stelle, die zu einer Fehlermeldung des Systems führen wird.

Beispiele:

- *Im System befindet sich die zweidimensionale Funktion $f(x, y) = 2xy + \ln(x)$:*

```
derivation_of f in 1 to f_x
derivation_of f in 22 - 2 * 10 to f_y
```

  ProSiNN erzeugt dann die Ableitungen `f_x` *als $f_x(x, y) = 2y + 1/x$ und* `f_y` *als $f_y(x, y) = 2x$.*

- *Es seien zwei Funktionen $i(x) = d(x) = \cos(2x) + 2^x$ definiert worden. Dabei wurde* `i` *als Integer– und d als Doublefunktion generiert:*

```
derivation_of i in 1 to i_x
derivation_of d in 1 to d_y
```

  ProSiNN erzeugt die Ableitung `d_x` *als $d_x(x) = 2 * (-\sin(2x)) + 2^x * \ln 2$. Die Ableitung* `i_x` *der Integerfunktion* `i` *ergibt $i_x(x) = 2 * (-\sin(2x))$; da hier intern alle Konstanten Integer sein müssen, wird* $\ln 2$ *und somit auch der zweite Term in d_x zu* 0.

5.5 Stringbefehle

In diesem Abschnitt sollen die Befehle vorgestellt werden, die ProSiNN zur Bearbeitung von *String*–Variablen anbietet. Dazu werden in den folgenden Unterabschnitten Kommandos eingeführt, die Strings oder Teile davon konkatenieren, Teilstrings löschen oder einfügen, Musterzeichenfolgen in einem String suchen oder die Länge einer Zeichenkette liefern.

Bem.:

Der Vorteil dieser Operationen liegt in der dynamischen Verwaltung von Programmen und Variablen. Der Benutzer kann mit String–Befehlen Variablennamen und Kommandos zusammensetzen und sie anschließend in eine Datei schreiben. Mit den erzeugten Strukturen ist er dann in der Lage, dynamisch entwickelte Definitionen und Programme auszuführen.

5.5.1 Strings konkatenieren

Auf Grund der Vollständigkeit sei hier noch einmal auf den entsprechenden Befehl hingewiesen. In Unterabschnitt 5.2.2 auf Seite 81 wurde die Variante des *set*–Kommandos präsentiert, die eine Konkatenation zweier Stringparameter und die Zuweisung an eine entsprechende Variable realisiert.

5.5.2 Teilstrings kopieren: copy

Diese Anweisung gibt dem Anwender die Möglichkeit, aus einem String eine beliebige Teilzeichenfolge (Teilstring) zu duplizieren und in einer Stringvariablen abzulegen. Die Syntax lautet:

copy_command	::=	*"copy"* **nr_char** *"at"* **start_pos** *"in"* **source_string** *"to"* **destination_string.**
nr_char	::=	**int_expression.**
start_pos	::=	**int_expression.**
source_string	::=	**string_val.**
destination_string	::=	**„Bezeichner einer Stringvariablen".**

Bem.:

- **nr_char** spezifiziert die Anzahl der zu kopierenden Zeichen. Es muß gelten: **nr_char** > 0. Wenn durch **nr_char** mehr Zeichen kopiert werden sollen, als real vorhanden sind, wird ab **start_pos** nur der Rest des Quellstrings kopiert.
- **start_pos** ist die Position, ab der die Zeichenfolge im Quellstring kopiert wird. Es sollte gelten: **start_pos** $\in \{1, \ldots,$ Länge des Quellstrings$\}$. Wenn **start_pos** größer als die Länge des Quellstrings ist, wird in **destination_string** der leere String abgelegt.
- **source_string** ist der Quellstring, aus dem die Zeichenfolge herauskopiert wird. Hier darf eine Konstante oder Variable vom Typ *string* verwendet werden.
- **destination_string** ist die Zielvariable, deren ursprünglicher Inhalt verloren geht.

Beispiele:
Im System befinden sich die zwei Stringvariablen `str1` *und* `str2`.

- `copy 2 at 5 in "1234567890" to str1`
  `copy 2 ^ 2 at 1 + 1 in "ProSiNN" to str2`

  Durch diese Kommandos wird in der Variablen `str1` *"*`56`*" und in* `str2` *"*`roSi`*" abgelegt.*

- `copy 100 at 2 in str2 to str1`
  `copy 10 at 100 in "Hallo" to str2`

  Der erste Befehl erzeugt in der Variablen `str1` *die Zeichenkette "*`oSi`*".* `str2` *enthält anschließend den leeren String.*

5.5.3 Teilstrings löschen: delete

Mit dieser Anweisung kann ein Teilstring aus einer Stringvariablen entfernt werden. Die Syntax des Kommandos ist:

delete_command ::= *"delete"* **nr_char** *"in"* **string_var** *"at"* **start_pos.**
nr_char ::= **int_expression.**
string_var ::= **„Bezeichner einer Stringvariablen".**
start_pos ::= **int_expression.**

Bem.:

- **nr_char** spezifiziert die Anzahl der zu löschenden Zeichen. Es muß gelten: **nr_char** > 0. Wenn durch **nr_char** mehr Zeichen entfernt werden sollen, als real vorhanden sind, wird ab **start_pos** nur der Rest des Quellstrings gelöscht.
- **string_var** ist der Bezeichner der Stringvariablen, aus der Zeichen entfernt werden sollen.
- **start_pos** ist die Position, ab der die Zeichenfolge in **string_var** gelöscht wird. Es sollte gelten: **start_pos** $\in \{1, \ldots,$ Länge von **string_var**$\}$. Ist **start_pos** größer als die Stringlänge der Variablen, so bleibt **string_var** unverändert.

Beispiele:
Im System befinden sich die zwei Stringvariablen `str1` *und* `str2`. `str1` *enthalte die Zeichenfolge "*`1234567890`*" und* `str2` *die Zeichenkette "*`Terminal`*".*

- `delete 2 in str1 at 5`
  `delete 22 - 20 in str2 at 55 - 5 * 10`

  Nach Ausführung dieser Kommandos enthält `str1` *die Zeichenkette "*`12347890`*" und* `str2` *"*`Termal`*".*

- `delete 100 in str1 at 4`
  `delete 10 in str2 at 100`

  Der erste Befehl erzeugt in der Variablen `str1` *die Zeichenkette "*`123`*".* `str2` *behält dagegen ihren ursprünglichen Wert bei.*

5.5.4 Teilstrings einfügen: insert

Mit dem *insert*–Befehl kann der Benutzer einen String in eine Variable vom gleichen Typ einfügen. Die Syntax lautet:

insert_command	::=	*"insert"* **source_string** *"to"* **destination_string** *"at"* **start_pos.**
source_string	::=	**string_val.**
destination_string	::=	**„Bezeichner einer Stringvariablen".**
start_pos	::=	**int_expression.**

Bem.:

- **source_string** ist der Quellstring, der in den Zielstring eingefügt wird. Hier darf eine Konstante oder Variable vom Typ *string* verwendet werden.
- **destination_string** ist die Zielvariable, in der die gewünschte Zeichenfolge eingefügt wird.
- **start_pos** ist die Position, ab der die Zeichenkette in **destination_string** eingefügt wird. Es sollte gelten: **start_pos** $\in \{1, \ldots,$ Länge von **destination_string**$\}$. Ist **start_pos** größer als die Stringlänge der Variablen, so wird die einzufügende Zeichenfolge an das Ende angehängt.

Beispiele:
Im System befinden sich die zwei Stringvariablen `str1` *und* `str2`. `str1` *enthalte die Zeichenfolge* "`1234567890`" *und* `str2` *die Zeichen* "`Hallo!`".

- `insert str2 to str1 at 5`
  `insert "li-Hal" to str2 at (1 + 1) * 2`

  Bei der Ausführung dieser Kommandos wird die Zeichenkette "`1234Hallo!567890`" *der Variablen* `str1` *und* "`Halli-Hallo!`" `str2` *zugewiesen.*

- `insert " Die Beispiele werden immer kreativer!" to str2 at 1000`

  Dieser Befehl weist der Variablen `str2` *die folgende Zeichenfolge zu:*
  "`Halli-Hallo! Die Beispiele werden immer kreativer!`"

5.5.5 Position eines Teilstrings bestimmen: position_of

Das Kommando gibt dem Anwender die Möglichkeit, einen String in einem anderen zu suchen und deren Position in einer Integervariablen abzulegen. Der Befehl muß die folgende Syntax einhalten:

position_of_command	::=	*"position_of"* **pattern_string** *"in"* **search_string** *"to"* **int_var.**
pattern_string	::=	**string_val.**
search_string	::=	**string_val.**
int_var	::=	**„Bezeichner einer Integervariablen".**

Bem.:

- **pattern_string** ist die zu suchende Zeichenkette (Muster). Für sie darf eine Konstante oder Variable vom Typ *string* verwendet werden.
- **search_string** ist die nach dem Muster zu untersuchende Zeichenkette. Hier ist eine Konstante oder Variable vom Typ *string* erlaubt.
- **int_var** ist eine Integervariable, der die Position (des ersten Auftretens des Musters) zugewiesen wird. Auch ein Integerfeldelement darf verwendet werden.
- Wird **pattern_string** nicht in **search_string** gefunden, so bekommt die Variable **int_var** den Wert 0; ansonsten wird ihr die Startposition zugewiesen.

Beispiele:
Im System befinden sich die zwei Stringvariablen `str1` *und* `str2`. `str1` *enthalte die Zeichenfolge "*`1234Hallo!567890`*" und* `str2` *die Zeichen "*`34`*". Außerdem existieren noch die zwei Integervariablen* `i` *und* `j`.

- ```
  position_of "Hallo" in str1 to i
  position_of str2 in str1 to j
  ```
  Nach der Ausführung der Kommandos ist `i` *gleich* 5 *und* `j` *hat den Wert* 3 *erhalten.*
- ```
  position_of "Nicht enthalten" in str2 to i
  ```
  Weil der gesuchte String nicht in der Variablen `str2` *enthalten ist, wird* `i` *der Wert* 0 *zugewiesen.*

5.5.6 Stringlänge bestimmen: length_of

Der Befehl *length_of* bestimmt die Länge eines Strings und weist ihn einer Integervariablen zu. Die Syntax lautet:

length_of_command ::= *"length_of"* **string_val** *"to"* **int_var.**
int_var ::= **„Bezeichner einer Integervariablen".**

Bem.:

- **string_val** ist der String, dessen Länge bestimmt werden soll. Hier darf eine Konstante oder Variable vom Typ *string* verwendet werden.
- **int_var** ist die Integervariable, der die Länge zugewiesen wird. Auch ein Integerfeldelement ist erlaubt.
- Handelt es sich um den leeren String, so wird **int_var** der Wert 0 zugewiesen.

Beispiele:
Im System befinden sich die zwei Stringvariablen `str1` *und* `str2`. `str1` *enthalte die Zeichenfolge "*`1234567890`*" und* `str2` *beinhalte den leeren String. Weiterhin seien noch die drei Integervariablen* `i`, `j` *und* `k` *definiert.*

```
length_of str1 to i
length_of str2 to j
length_of "Ich bin 24 Zeichen lang!" to k
```

Nach der Ausführung dieser Befehle hat die Variable `i` *den Wert* 10*,* `j` *wurde auf* 0 *gesetzt und* `k` *beinhaltet* 24*.*

5.6 Kommandomakros definieren: alias

ProSiNN bietet die Möglichkeit, mehrere Anweisungen als *Makro* zu definieren. Bei Aufruf des Makronamens werden die vorher festgelegten Kommandos ausgeführt. Die Syntax des entsprechenden Befehls ist:

alias_command ::= "*alias*" **macro_name**
"{" **command** ";" { **command** ";" } "}".
macro_name ::= **„Noch nicht definierter Bezeichner".**
command ::= **„Ein gültiger ProSiNN–Befehl".**

Bem.:

- **macro_name** ist der im System noch nicht definierte Bezeichner, mit dem das Makro ausgeführt werden kann.
- **command** bezeichnet ein beliebiges von ProSiNN unterstütztes Kommando. Auch der Bezeichner eines anderen bereits generierten Makros ist zulässig.
- Die Korrektheit der Anweisungen wird erst beim ersten Aufruf des Makros überprüft!
- Alle im Anweisungsblock neu erzeugten Variablen sind auch nach Ausführung des Makros noch definiert. Es können also keine lokalen Variablen generiert werden!

Beispiele:
Die folgenden Beispiele zeigen jeweils Auschnitte aus dem Ein/Ausgabefenster, die durch Kommandoeingaben entstanden sind:

- *Das Ziel des ersten Makros ist die Definition einer Anweisung, die eine Begrüßungsfloskel ausgibt:*

  ```
  >:alias p_hallo  { print "Hallo!"; }
  >:p_hallo
  Hallo!
  >:p_hallo
  Hallo!
  >:
  ```

  Zuerst wurde das Makro definiert und anschließend zweimal aufgerufen (näheres zum print*–Befehl siehe in Unterabschnitt 7.3.1).*

- *Im zweiten Beispiel soll dieselbe Aufgabe gelöst werden. Zur Verwendung kommt jetzt aber eine Stringvariable, die den Begrüßungstext beinhaltet.*

```
>:alias p_hallo2 { define string str = "Hallo!";
                   print str;
                 }
>:p_hallo2
Hallo!
>:p_hallo2

Fehler 60:
Bezeichner ist schon definiert: "str"
Der Fehler trat in einem durch 'alias' definierten Befehl auf:
"p_hallo2"

>:
```

Innerhalb der Makrodefinition befinden sich zwei ProSiNN–Befehle: Der erste erzeugt eine Stringvariable (mit dem gewünschten Inhalt) und der zweite gibt sie anschließend aus. Als das Makro zum zweiten Mal aufgerufen wurde, ist eine Fehlermeldung deshalb ausgegeben worden, weil die erste Anweisung erneut die Variable `str` *zu definieren versuchte.*

- *Der folgende Ausschnitt aus dem Ein/Ausgabefenster zeigt die korrigierte Fassung des zweiten Beispiels (der Name* `_str` *darf aber vorher nicht definiert sein).* free_var *ist der Befehl, der eine Variable löscht und den Namen wieder zur Verwendung freigibt (siehe nächsten Abschnitt).*

```
>:alias p_hallo3 { define string _str = "Hallo!";
                   print _str;
                   free_var _str;
                 }
>:p_hallo3
Hallo!
>:p_hallo3
Hallo!
>
```

Hier wird bei jedem Aufruf die Variable `_str` *neu erzeugt und anschließend wieder entfernt!*

5.7 Eine Variable freigeben: free_var

Der Befehl *free_var* entfernt eine Variable aus dem Speicher. Anschließend ist der Bezeichner der Variablen wieder verwendbar. Die Syntax lautet:

free_var_command ::= *"free_var"* **var_name.**
var_name ::= **„Bezeichner der zu entfernenden Variablen".**

Bem.:

var_name kann der Bezeichner einer Variablen der folgenden Datentypen sein:

- *int* oder *double*:

  Die Datenstruktur und der Bezeichner einer vom Benutzer definierten Integer– oder Doublevariablen werden freigegeben.

- *array int* oder *array double*:

  Die Datenstrukturen und der Bezeichner einer Integer– oder Doublefeldvariablen werden freigegeben.

- *function int* oder *function double*:

  Die Datenstrukturen und der Bezeichner einer vom Anwender definierten Integer– oder Doublefunktion werden freigegeben.

- *string*:

  Die Datenstrukturen und der Bezeichner einer vom Benutzer definierten Stringvariablen werden freigegeben.

- *layer*, *unit* oder *weight*:

  Die Datenstrukturen und der Bezeichner einer vom Benutzer definierten Netzebenen–, Neuronen– oder Gewichtsvariablen werden freigegeben.

- *pattern*:

  – Bei einem Trainingsmuster wird nur der zugehörige Bezeichner entfernt. Die Musterdaten bleiben erhalten.
  – Bei einem Testmuster werden die Musterdaten und der Bezeichner freigegeben (d.h. vollständiges Entfernen aus dem System).

- Makros:

  Die Datenstrukturen und der Bezeichner eines vom Anwender definierten Makros werden freigegeben.

Beispiele:
Im System existieren eine Integervariable `i`*, eine Funktion* `f`*, das im letzten Abschnitt definierte Makro* `p_hallo` *und eine Gewichtsvariable* `weight`*. Durch die folgenden Kommandos werden die zugehörigen Datenstrukturen und Bezeichner wieder aus dem System entfernt:*

```
free_var i
free_var f
free_var p_hallo
free_var weight
```

Kapitel 6

Kontrollstrukturen und ProSiNN–Programme

In diesem Kapitel werden Befehle präsentiert, die ProSiNN–Programme starten und deren Abläufe kontrollieren. Nachdem im ersten Abschnitt die Vergleichsoperatoren und die darauf aufgebauten Ausdrücke vorgestellt worden sind, werden die vorhandenen Kontrollstrukturen — die *if*–Anweisung und die *while*–Schleife — eingeführt. Der zweite Abschnitt beschäftigt sich mit dem Aufbau eines ProSiNN–Programms und dem Befehl *load*, der ein Programm lädt und ausführt.

6.1 Kontrollstrukturen

Jede imperative Programmiersprache benötigt Kontrollstrukturen, die die Reihenfolge der Anweisungen steuert. Im ersten Unterabschnitt werden die vorhandenen Vergleichsoperatoren und die darauf basierenden Ausdrücke vorgestellt. Um zwischen Alternativen auswählen zu können, bietet der anschließend präsentierte *if*–Befehl die Möglichkeit der *bedingten Auswertung*. Im letzten Unterabschnitt wird die Realisierung von Programmschleifen mit Hilfe der *while*–Anweisung dargestellt.

6.1.1 Vergleichsoperatoren

Das ProSiNN–System bietet dem Programmierer Operatoren zum arithmetischen und lexikographischen Vergleich. Ist ein Vergleich erfüllt, so hat ein solcher Ausdruck den Wert 1; ansonsten ist er 0. Die Syntax eines Vergleichsausdrucks lautet:

compare_expression	::=	**operand compare_operator operand.**
compare_operator	::=	“=“ | “#“ | “<“ | “<=“ | “>“ | “>=“.
operand	::=	**int_expression** | **double_expression** |
		string_val | **unit_val** | **weight_val.**

Bem.:

Die beiden Operanden, die hier durch das Nichtterminalsymbol **operand** repräsentiert werden, müssen typkonform sein. Es ist z.B. nicht erlaubt, eine Integervariable mit einer Gewichtsangabe zu vergleichen.

Für die einzelnen Operandentypen gilt:

- **int_expression** (auch Netzebenenvariablen sind zulässig):
  Bei jeder Operation wird der arithmetische Vergleich durchgeführt.
- **double_expression**:
  Bei jeder Operation wird der arithmetische Vergleich durchgeführt.
- **string_val**:
  Hier wird der übliche lexikographische Vergleich der beiden Stringparameter durchgeführt. Dabei sind Zahlen lexikographisch kleiner als Großbuchstaben, die wiederum kleiner als Kleinbuchstaben sind!
- **unit_val**:
  Zuerst werden die Netzebenen der beiden Neuronen miteinander numerisch verglichen. Im Fall der Gleichheit ist das Ergebnis der arithmetische Vergleich der schichtinternen Neuronennummern; ansonsten wird das erste Resultat verwendet.
  Es dürfen keine Neuronenwerte der ersten Schicht miteinander verknüpft werden!
- **weight_val**:
  Zuerst werden die Startneuronen der Gewichte miteinander verglichen (siehe die letzte Bemerkung). Im Fall der Gleichheit ist das Ergebnis der Vergleich der beiden Zielzellen der Gewichte; ansonsten wird das erste Resultat verwendet.

Beispiele:
Im System befindet sich eine Integervariable `i` *(mit dem Wert* 3*), eine Stringvariable* `s` *(mit dem Inhalt "*`Hallo!`*"), eine Layervariable* `l` *(sie liefert eine Referenz zur 2. Ebene eines dreischichtigen Netzes), eine Unitvariable* `u` *(die das 2. Neuron der 3. Schicht charakterisiert) und eine Gewichtsvariable* `weight`*, die die Verbindung des ersten Eingabeneurons und des 2. Neurons der mittleren Ebene bezeichnet:*

- *Folgende Ausdrücke liefern eine* 0 *als Ergebnis:*

```
3 = 4 , 9 - 2 * 3 # i , i < 3 , 2 ^ 10 <= i + 10 , -100 * 3 >= i / 10
2.0 * pi = 4.0 , cos(pi) # -1.0 , cos(2.4 * pi) >= 10.0
s = "ProSiNN" , "Hallo!" # s , "abc" < "ABC" , "1" > "a"
u # 3:2 , 3:1 < 2:1 , 3:1 > u
weight = 1:1:2:1 , 1:1:2:2 < 1:1:2:1
```

- *Die nachfolgenden Ausdrücke liefern eine* 1 *als Rsultat:*

```
3 # 4 , 9 - 2 * 3 = i , i <= 3 , 2 ^ 10 >= i + 10
2.0 * pi # 4.0 , cos(pi) = -1.0 , cos(2.4 * pi) <= 10.0
s # "ProSiNN" , "Hallo!" = s , "abc" > "ABC" , "1" <= "a"
u = 3:2 , 3:1 > 2:1 , 3:1 < u
weight # 1:1:2:1 , 1:1:2:2 > 1:1:2:1
```

6.1.2 Die bedingte Verzweigung: if

Der *if*–Befehl dient zur Ablaufsteuerung in einem Programm. Ist ein Vergleichsausdruck erfüllt, so wird ein Anweisungsblock ausgeführt; im anderen Fall wird dieser Block übersprungen und evtl. ein optionaler Kommandoblock interpretiert. Die Syntax des Befehls ist:

if_command	::=	*"if"* *"("* **compare_expression** *")"*
		"{" **statement_block**$_1$ "}"
		[*";"* *"else"* "{" **statement_block**$_2$ "}"].
statement_block$_1$	::=	**statement_block.**
statement_block$_2$	::=	**statement_block.**
statement_block	::=	{ **statement** *";"* }.
statement	::=	**„Ein ProSiNN–Befehl".**

Bem.:

- **compare_expression** ist der in Unterabschnitt 6.1.1 definierte Vergleichsausdruck. Ist dessen Wert gleich 1 (Vergleich ist erfüllt), so wird der **statement_block**$_1$ ausgeführt (der Rest des Befehls wird ignoriert). Ist der Wert gleich 0 (Vergleich ist nicht erfüllt), so wird der optionale **statement_block**$_2$ — wenn vorhanden — interpretiert; ansonsten wird die Anweisung nach dem *if*–Befehl bearbeitet.
- Ein **statement_block** ist eine sequentielle Aneinanderreihung von ProSiNN–Kommandos, die jeweils durch ein Semikolon getrennt werden. Erlaubt sind hier auch weitere *if*–oder *while*–Befehle (d.h. weitere Schachtelungen sind möglich).
- Ein leerer **statement_block** ist zulässig.

Beispiele:
Im System befindet sich eine Integervariable `i` *und eine Doublevariable* `d`. *Die folgenden Beispiele zeigen jeweils Auschnitte aus einem ProSiNN–Programm:*

- ```
  if( i = 100 )
    { set i = 0; }
  ```
  Ist der Wert der Variablen `i` *gleich* 100, *so wird* `i` *durch diese Befehle auf* 0 *gesetzt.*

- ```
  if( i <= 100 )
  { set d = cos(pi / 2.0);
    set i = 0;
  };
  else
  { set d = cos(pi); set i = -1; }
  ```
  Diese Kommandos setzen die Variablen `i` *und* `d` *auf* 0 *(bzw.* 0.0*), wenn der Wert von* `i` *vorher kleiner oder gleich* 100 *war. Ansonsten werden beide auf* -1 *bzw.* -1.0 *gesetzt.*

- ```
  if( i < 0 )
  { set d = -1.0; };
  else
  { if( i = 0 ) { set d = 0.0; };
    else        { set d = 1.0; };
  }
  ```

  Diese Anweisungen weisen der Variablen `d` *das Vorzeichen von* `i` *zu. Ist* `i` *kleiner als* 0*, dann wird* `d` *auf* -1.0 *gesetzt; ist der Wert gleich* 0*, so erhält* `d` 0.0*. Ansonsten wird* `d` *auf* 1.0 *gesetzt.*

6.1.3 Die Schleife: while

Zur Ausführung von Schleifen stellt ProSiNN den *while*–Befehl zur Verfügung. Dieses Kommando führt solange einen Anweisungsblock aus, bis der angegebene Vergleichsausdruck nicht mehr erfüllt ist. Die Syntax des Befehls lautet:

while_command ::= *"while" "("* **compare_expression** *")"*
"{" **statement_block** *"}"*.
statement_block ::= { **statement** *";"* }.
statement ::= **„Ein ProSiNN–Befehl"**.

Bem.:

- **compare_expression** ist der in Unterabschnitt 6.1.1 definierte Vergleichsausdruck. Solange dessen Wert gleich 1 ist (Vergleich ist erfüllt), wird der **statement_block** ausgeführt. Ist der Wert gleich 0 (Vergleich ist nicht erfüllt), so wird der Befehl nach dem *while*–Kommando bearbeitet.
- Ein **statement_block** ist eine sequentielle Aneinanderreihung von ProSiNN–Kommandos, die jeweils durch ein Semikolon getrennt werden. Erlaubt sind hier auch weitere *if*–oder *while*–Befehle (d.h. weitere Schachtelungen sind möglich).
- Ein leerer **statement_block** ist zulässig.
- Zur Beendigung von (Endlos–) Schleifen kann der Benutzer die Ausführung abbrechen lassen. Dazu muß der Mauszeiger in das Ein/Ausgabefenster gesetzt und die Tastenkombination `Control-C` gedrückt werden.

Beispiel:
Im System befindet sich eine Integervariable `i`*, deren Wert* 0 *beträgt.*

```
while( i < 100 )
{ print "Hallo";
  print EOL;
  set i = i + 1;
}
```
```

Diese Schleife druckt 100 mal den Text "`Hallo!`*" in das Ein/Ausgabefenster.* `print EOL` *führt einen Zeilenvorschub aus.*
Zwei weitere Beispiele findet man im nächsten Abschnitt.

6.2 ProSiNN–Programme ausführen: load

Mit dem *load*–Befehl kann der Benutzer ein ProSiNN–Programm laden und starten. Die Syntax des Kommandos ist:

load_command ::= *"load"* **file_name.**
file_name ::= **„Der Pfad und Dateiname des Programms".**

Bem.:

- **file_name** ist der Pfad und Dateiname des Programms, das ausgeführt werden soll. Die genaue Definition findet man in Unterabschnitt 7.2.1.
- Der syntaktische Aufbau eines Simulationsprogramms muß den folgenden Bedingungen genügen:
  - Das auszuführende Programm besteht aus einem **statement_block** beliebiger Größe. Jedes Kommando muß mit einem Semikolon abgeschlossen werden. Innerhalb dieser Anweisungen dürfen auch weitere *load*–Befehle auftreten. So kann z.B. ein Programm in Abhängigkeit einer Bedingung geladen und ausgeführt werden.
  - Zur Dokumentation des Programms können Kommentare verwendet werden. Ein Kommentar ist folgendermaßen definiert:

    comment ::= *"/*"* { **char_const** } *"*/"*.

    In einem Kommentar dürfen sich weitere Kommentare befinden. Damit ist es möglich, während eines Testlaufs kommentierte Befehlssequenzen von der Interpretierung auszuschließen.
- Zur vorzeitigen Beendigung eines Programms kann der Benutzer die Ausführung abbrechen lassen. Dazu muß der Mauszeiger in das Ein/Ausgabefenster gesetzt und die Tastenkombination `Control-C` gedrückt werden.
- Von einem Programm generierte Variablen sind nach dessen Ausführung noch im System vorhanden — sofern sie nicht explizit wieder freigegeben werden. Das Simulationsprogramm selbst wird aber aus dem Speicher entfernt. Um es erneut zu starten, muß der entsprechende *load*–Befehl wieder eingegeben werden.

Beispiele:

- *Das folgende Beispiel zeigt ein Programm, das nach dem Start 100 mal "*`Hallo!`*" in das Ein/Ausgabefenster schreibt:*

```
define int i;
while( i < 100 )
{ print "Hallo!";
  print EOL;
  set i = i + 1;
  /* Dies ist ein /* geschachtelter /* Kommentar */ ! */ */
};
```

Dieser Text ist in der Datei mit dem Namen `test` *gespeichert. Wenn sich ProSiNN in demselben Verzeichnis wie die Datei* `test` *befindet, kann das Simulationsprogramm durch den nachfolgenden Befehl gestartet werden:*

```
load test
```

- *Das nächste Beispiel zeigt ein Programm, das alle Primzahlen zwischen 2 und 100 berechnet und ausgibt:*

```
define int i = 2; /* Zaehlvariable ueber alle zu testenden Zahlen */
define int j; /* zum Testen */
while( i < 101 )
{ set j = 2; /* Divisor zum Testen */
  while( j < i )
  { /* Teilbarkeitstest bei allen Zahlen, die kleiner als i sind */
    if( ( i / j ) * j # i )
    { /* es gibt einen Rest; j teilt i nicht */
      set j = j + 1; /* auf naechsten moeglichen Teiler setzen */
      if( j = i )
      { /* alle Zahlen getestet ==> i ist prim! */
        print i; print " ist eine Primzahl";
        print EOL;
      };
    };

    else
    { /* j teilt i ==> i ist nicht prim! */
      set j = i; /* zum Beenden der Ueberpruefung */
    };
  };
  set i = i + 1; /* naechste Zahl untersuchen */
};
```

Dieses Programm ist in der Datei `prim_test` *gespeichert, die mit ProSiNN im gleichen Verzeichnis liegt. Die folgende Auflistung zeigt den Befehl zum Ausführen und die Ausgaben, die erzeugt werden:*

```
>:load prim_test
3 ist eine Primzahl
5 ist eine Primzahl
7 ist eine Primzahl
11 ist eine Primzahl
13 ist eine Primzahl
17 ist eine Primzahl
19 ist eine Primzahl
23 ist eine Primzahl
29 ist eine Primzahl
31 ist eine Primzahl
37 ist eine Primzahl
41 ist eine Primzahl
43 ist eine Primzahl
47 ist eine Primzahl
53 ist eine Primzahl
59 ist eine Primzahl
61 ist eine Primzahl
67 ist eine Primzahl
71 ist eine Primzahl
73 ist eine Primzahl
79 ist eine Primzahl
83 ist eine Primzahl
89 ist eine Primzahl
97 ist eine Primzahl

>:
```

- *Weitere Beispiele befinden sich in Kapitel 20 und 21.*

Kapitel 7

Ein– und Ausgabebefehle

Jede Programmiersprache benötigt Anweisungen, die von der Tastatur (oder aus Files) Daten einlesen oder auf ein Medium ausgeben. ProSiNN bietet daher mehrere Befehle zur E/A–Verarbeitung an.

Im ersten Abschnitt wird das Kommando vorgestellt, das vom Anwender am Bildschirm interaktiv Eingaben anfordert. Anschließend wird die Dateiverwaltung beschrieben, die zum Speichern von Daten verwendet wird. Der letzte Teil dieses Kapitels führt die wichtigsten Anweisungen ein, mit denen Werte in einer Datei ablegt werden können; weitere Ausgabebefehle werden später bei der Darstellung anderer Befehlsgruppen präsentiert.

7.1 Werte einlesen: input

Der *input*–Befehl bietet die Möglichkeit, während einer Programmausführung Eingaben über die Tastatur vom Benutzer anzufordern. Das Kommando verlangt solange ein neues Datum, bis dessen Format mit dem der Variablen, in der der Wert gespeichert werden soll, übereinstimmt. Die Syntax des Befehls lautet:

input_command	::=	*“input“* **var_parameter.**
var_parameter	::=	**int_var | double_var | string_var |**
		layer_var | unit_var | weight_var.
int_var	::=	**„Integervariable oder Integerfeldelement“.**
double_var	::=	**„Doublevariable oder Doublefeldelement“.**
string_var	::=	**„Stringvariable“.**
layer_var	::=	**„Layervariable“.**
unit_var	::=	**„Unitvariable“.**
weight_var	::=	**„Weightvariable“.**

Bem.:

- Dieser Befehl ist auch direkt durch den Kommandozeileninterpreter ausführbar.
- Wartet das System auf Eingaben vom Anwender, so erscheint das Prompt “**?:**“.

- Eine Eingabe wird mit `Return`–Taste übernommen und ausgewertet.
- Das System verlangt solange einen neuen Eingabewert, bis dieser korrekt ist.
- In der Regel dürfen bei Eingaben nur Konstanten verwendet werden. Die Ausnahmen bilden Netzebenen–, Neuronen– und Gewichtseingaben. Hier dürfen typgleiche Variablen verwendet werden (auch zusammengesetzte Ausdrücke sind erlaubt; siehe Beispiele).
- Beim Einlesen eines Gleitkommawertes ist der Dezimalpunkt nicht zwingend vorgeschrieben.
- Bei einfachen Zahleneingaben (Integer, Double oder Layer) werden solange Zeichen verarbeitet, bis eins auftritt, das nicht einer Zahl vom entsprechenden Typ zugeordnet werden kann.
- Bei Eingaben, die an vordefinierte Variablen zugewiesen werden sollen, liest der Befehl solange Werte ein, bis die jeweiligen Bereichsbeschränkungen erfüllt sind. Wenn z.B. **eta** mit dem *input*–Befehl gesetzt werden soll, wird solange ein neuer Wert verlangt, bis dieser größer als 0.0 ist.
- Soll eine Stringvariable über die Tastatur gesetzt werden, so wird jedes vom Benutzer eingegebene Zeichen — ohne das abschließende *Return* — in der Variablen abgelegt. Es werden keine Anführungszeichen bei der Eingabe benötigt. Diese Anweisung stellt die einzige Möglichkeit in Programmen dar, Anführungszeichen in einem String zu erzeugen!

Beispiele:
Im System befindet sich eine Integervariable `i`*, eine Layervariable* `l`*, eine Unitvariable* `u` *und zwei Gewichtsvariablen* `w1` *und* `w2`*. Das vorliegende Neuronale Netz enthält drei Ebenen mit 2 Eingabeneuronen, 2 Zellen in der mittleren Schicht und ein Ausgabeneuron. Die folgenden Beispiele zeigen jeweils einen Ausschnitt des Ein/Ausgabefensters:*

- *Das erste Kommando fordert eine Eingabe für die Integervariable* `i` *an:*

  ```
  >:input i

  ?:2
  >:
  ```

  Hier wurde `i` *durch den Anwender auf* 2 *gesetzt.*

- *Das nächste Beispiel zeigt, daß nur so viele Eingabezeichen verarbeitet werden, bis das erste die Regeln des verlangten Datentyps verletzt. Der* print*–Befehl gibt anschließend den aktuellen Wert der Variablen aus (siehe Unterabschnitt 7.3.1):*

  ```
  >:input i

  ?:2.3
  ```

```
>:print i
2
>:
```

- ```
  >:input act_error

  ?:hallo
  ?:-100
  ?:-22.3
  ?:13
  >:
  ```

  Bei diesem Beispiel liest das System solange neue Werte ein, bis eine korrekte Eingabe vorliegt. Im Anschluß hat **act_error** *den Wert* 13.0.

- *Der nächste Einlesebefehl, der die Netzebenenvariable* `u` *setzen soll, wird solange wiederholt, bis ein korrekter Wert vorliegt:*

  ```
  >:input l

  ?:-1
  ?:0
  ?:4
  ?:1
  >:
  ```

- *Das letzte Beispiel zeigt, wie Unit– und Gewichtsvariablen über die Tastatur gesetzt werden können:*

  ```
  >:input u

  ?:1:2
  ?:2:1
  >:input w1

  ?:u:3:1
  >:input w2

  ?:w1
  >:
  ```

  Die Neuronenvariable `u` *wird vom Benutzer auf das 1. Neuron der 2. Ebene gesetzt.* `w1` *bezeichnet nun die Verbindung, die von* `u` *zum Ausgabeneuron verläuft. Der Anwender setzt abschließend die Variable* `w2` *auf das gleiche Gewicht, das* `w1` *beschreibt.*

7.2 Dateiverwaltung

In diesem Abschnitt werden die Mechanismen vorgestellt, die zur Verwaltung von Dateien verwendet werden. Nach einer grundlegenden Einführung, beschäftigen sich die nachfol-
  ```

genden Unterabschnitte mit dem Öffnen und Schließen einzelner Dateien.

7.2.1 Dateinamen und Dateihandles

Will der Anwender eine Datei laden (z.B. ein Simulationsprogramm oder ein File, das ein Neuronales Netz beinhaltet) oder zum Schreiben öffnen, so muß er sich beim Dateinamen an die folgende Konvention halten:

file_name ::= string_const | string_var | any_char_sequence.
any_char_sequence ::= char_const { char_const }.

Bem.:

- **file_name** ist der evtl. mit dem Pfad versehene Dateiname. Der Pfad braucht nur angegeben zu werden, wenn er von dem des aktuellen Arbeitsverzeichnisses differiert.
- **string_const** ist eine beliebige Stringkonstante (ohne innere Anführungszeichen), die den Pfad und den Dateinamen beschreibt.
- **string_var** beschreibt eine Stringvariable, die den Pfad und den Dateinamen enthält.
- Der Pfad und der Dateiname können auch durch eine beliebige Zeichenfolge, die nicht in Anführungszeichen steht, angegeben werden. Diese Zeichenkette darf dann aber nicht mit einem systeminternen Bezeichner kollidieren. Außerdem dürfen in der Zeichenfolge keine Blanks, Tabulatoren und Zeilenendezeichen vorkommen; ansonsten betrachtet ProSiNN nur den ersten zusammenhängenden Teil als Dateinamen. In dem Verzeichnis `/a/bayes/u4/users/bayes/karsten/work` befinde sich z.B. eine Datei mit dem Namen `test`. Sie kann mit dem folgenden Dateinamen angesprochen werden: `/a/bayes/u4/users/bayes/karsten/work/test`.

Um Daten in eine Datei zu schreiben, muß der Benutzer (oder ein Programm) ein File explizit zum Schreiben öffnen. Dazu muß der gewünschte Dateiname angegeben werden. Wenn ProSiNN diese Datei dann geöffnet hat, wird ein *Dateihandle* (*Filedeskriptor*) an den Anwender zurückgegeben. Dieses *Handle* ist dann bei jedem Ausgabebefehl anzugeben; es dient dabei zur eindeutigen Identifizierung eines Files. Nachdem alle Ausgaben vorgenommen worden sind, muß die Datei mit Hilfe des Handles wieder geschlossen werden; wird dies nicht gemacht, könnten alle bisher ausgegebenen Daten *verloren* sein!

ProSiNN erlaubt maximal 10 Dateihandles[1], d.h. das System gestattet es, bis zu 10 Dateien gleichzeitig zu verwalten. Zusätzlich existiert die Konstante **stdout**, die das Handle zum normalen Ein/Ausgabefenster darstellt; damit lassen sich Ausgabebefehle in einem Simulationsprogramm allgemeingültiger formulieren. Die Syntax eines Dateihandles lautet:

file_handle ::= int_const.

[1] Dieser Wert kann bei der Systeminstallation geändert werden

Bem.:

Es handelt sich also um einen Integerwert, der den Zugriff auf eine zum Schreiben geöffnete Datei erlaubt. Bei der aktuellen Systemkonfiguration stammen die Handles aus dem Intervall $[0, \ldots, 11]$, wobei 0 der Wert von **stdout** ist und nicht mehr von ProSiNN vergeben wird. Das System gibt immer den ersten freien Filedeskriptor beim Öffnen einer neuen Datei zurück.

7.2.2 Datei zum Schreiben öffnen: open

Der *open*–Befehl dient zum Öffnen einer Datei, in die Daten vom Benutzer oder einem Programm geschrieben werden können. Die Syntax des Befehls ist:

open_command ::= *"open"* **file_name int_var.**
int_var ::= **„Integervariable oder Integerfeldelement".**

Bem.:

- **file_name** ist der Pfad und Name der Datei, die zum Schreiben geöffnet werden soll (siehe letzter Unterabschnitt).
- Das System legt das mit der Datei korrespondierende Handle in der Variablen **int_var** ab.
- Ist die angegebene Datei noch nicht vorhanden, so erzeugt ProSiNN sie neu; existiert sie dagegen bereits, wird ihr ursprünglicher Inhalt gelöscht.

Beispiele:
ProSiNN wird in dem Verzeichnis `work` *gestartet.* `work` *beinhaltet ein Unterverzeichnis mit dem Namen* `prog`*, dessen einziger Inhalt die Datei* `test` *ist. Seit dem Systemstart wurde kein Verzeichniswechsel vorgenommem. ProSiNN verwaltet außerdem die Integervariablen* `i` *und* `j`*, sowie eine Stringvariable* `s`*. Der Anwender möchte nun in das File* `test` *Daten schreiben.*

- *Der folgende Befehl öffnet die Datei* `test` *und weist das Handle der Variablen* `i` *zu:*

  ```
  open prog/test i
  ```

- *Zu einer Fehlermeldung würde das nachfolgende Kommando führen, da ProSiNN* `prog/` *als Dateinamen und* `test` *als die benötigte Integervariable betrachtet:*

  ```
  open prog/ test i
  ```

- *Mit einer Stringkonstanten könnte der Dateiname folgendermaßen angegeben werden:*

  ```
  open "prog/test" i
  ```

- *Mit der Stringvariablen* `s` *läßt sich das Öffnen wie folgt durchführen:*

  ```
  set s = "prog/test"
  open s i
  ```

7.2.3 Datei schließen: close

Der *close*–Befehl schließt eine zum Schreiben geöffnete Datei wieder. Seine Syntax lautet:

close_command ::= *"close"* **handle.**
handle ::= **int_expression.**

Bem.:

- **handle** muß ein Dateideskriptor sein, der eine vom Anwender geöffnete Datei spezifiziert.
- Nach der Ausführung des Kommandos korrespondiert das Handle nicht mehr mit dem File und ein weiterer Zugriff ist nicht mehr möglich; es sei denn, sie wird wieder explizit mit dem *open*–Kommando geöffnet, wobei ihr Inhalt aber verlorengeht!

Beispiele:
ProSiNN habe eine Datei zum Schreiben geöffnet und dem Anwender das zugehörige Handle 1 *in der Integervariablen* `i` *übergeben:*

- *Der folgende Befehl schließt die Datei wieder:*

```
close i
```

- *Eine weitere Möglichkeit besteht darin, das Handle direkt mit dem Wert* 1 *anzugeben (wobei ein beliebiger Integerausdruck erlaubt ist):*

```
close 10 - 9
```

7.3 Werte ausgeben

Dieser Abschnitt ist den wichtigsten Ausgabebefehlen gewidmet. Nach der Einführung des Standardbefehls wird ein Kommando vorgestellt, das einen Zeilenvorschub durchführt. Den Abschluß bildet der Befehl, der eine beliebige Funktion ausgeben kann.

7.3.1 Standardausgabebefehl: print

Mit dem *print*–Kommando können eine Vielzahl von Parametern unterschiedlichen Typs in das Ein/Ausgabefenster bzw. eine Datei geschrieben werden. Die Syntax des Befehls ist:

print_command ::= *"print"* **parameter [file_handle].**
parameter ::= **int_val | double_val | function_val | string_val | layer_val | unit_val | weight_val.**
file_handle ::= **int_expression.**
function_val ::= **function_name** *"("* **expression {** *","* **expression }** *")"*.
function_name ::= **„Name einer Integer– oder Doublefunktion".**

Bem.:

- **parameter** beschreibt den Wert, der ausgegeben werden soll:
  - **int_val**:
    Hier darf eine Integerkonstante, eine entsprechende Variable oder ein Integerfeldelement verwendet werden.
  - **double_val**:
    Hier darf eine Doublekonstante, eine entsprechende Variable oder ein Doublefeldelement benutzt werden.
  - **function_val**:
    Es muß eine Integer– oder Doublefunktion und deren Funktionsargumente, die aus komplexen typkonformen Ausdrücken bestehen können, angegeben werden.
  - **string_val**:
    Hier darf eine Stringkonstante oder eine entsprechende Variable verwendet werden.
  - **layer_val**, **unit_val** und **weight_val**:
    Sie beschreiben eine Konstante oder Variable des entsprechenden Typs. Dabei dürfen die einzelnen Komponenten auch aus Variablen zusammengesetzt sein. Eine Gewichtsangabe kann z.B. aus zwei Unitvariablen gebildet werden.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird die Ausgabe in das Ein/Ausgabefenster gelenkt.

Beispiele:
Es sei eine Datei geöffnet worden, deren Handle 4 *ist. Die folgenden Beispiele zeigen jeweils einen Ausschnitt aus dem Ein/Ausgabefenster:*

- *Im System existiere eine Integervariable* `i` *(mit dem Wert* 3*) und ein Doublefeld* `da` *mit 10 Elementen (jedes hat den Wert* 3.41*). Der Wert der Integervariablen und des 9. Feldelementes sollen in das Ein/Ausgabefenster geschrieben werden. Anschließend soll* $\cos(\pi^2 - e)$ *berechnet werden. Der letzte Befehl zeigt die Ausgabe einer Stringkonstanten:*

  ```
  >:print i stdout
  3
  >:print da[ 2 ^ 3 ]
  3.4100000
  >:print cos( pi ^ 2.0 - e )
  0.64624917
  >:print "Hallo!"
  Hallo!
  >:
  ```

- *Im System sei nun ein dreischichtiges Neuronales Netz. In der ersten Ebene befinden sich 4, in der mittleren 2 und in der Ausgabeschicht 3 Neuronen. Es sei eine*

Unitvariable `u` *vorhanden, die das 3. Ausgabeneuron bezeichnet. Weiterhin befinde sich im System noch die Gewichtsvariable* `weight`.

```
>:print u
3:3
>:set weight = 2:1:u
>:print weight
2:1:3:3
>: print u 4; print EOL 3 * 4 - 2 * 4; print weight 2 ^ 2
```

Der erste Befehl druckt die Unitvariable in das normale Ein/Ausgabefenster. Anschließend wird die Gewichtsvariable neu gesetzt und in die Standardausgabe geschrieben. Das letzte Kommando druckt beide Werte in die Datei, dessen Handle 4 *ist. In der Datei steht nun u.a.:*

```
3:3
2:1:3:3
```

7.3.2 Zeilenvorschub ausführen: println

Der *println*–Befehl führt einen Zeilenvorschub in einem angegebenen Medium aus. Die Syntax ist:

println_command ::= *"println"* [**file_handle**].
file_handle ::= **int_expression.**

Bem.:

- Dieser Befehl ist gleichwertig zu dem folgenden Kommando:
  "print EOL" [**file_handle**].
  Der *println*–Befehl bietet allerdings den Vorteil der schnelleren Verarbeitung!
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird **EOL** in das Ein/Ausgabefenster geschrieben.

Beispiele:
Es sei eine Datei geöffnet worden, deren Handle 4 *ist. Die folgenden Beispiele zeigen jeweils einen Ausschnitt aus dem Ein/Ausgabefenster:*

- *Der nachfolgende Befehl führt einen Zeilenvorschub im Ein/Ausgabefenster durch:*

  ```
  >:println


  >:
  ```

- *Wie in die Datei mit dem Handle 4 ein Zeilenvorschub ausgegeben wird, zeigt das letzte Beispiel:*

```
>:println 4
>:
```

7.3.3 Funktion ausgeben: print_function

Zur Ausgabe einer Funktionsdefinition, muß der Benutzer den *print_function*–Befehl verwenden. Seine Syntax lautet:

print_function_command	::=	*"print_function"* **function_name [file_handle].**
function_name	::=	**„Name einer definierten Funktion".**
file_handle	::=	**int_expression.**

Bem.:

- **name** ist der im System existierende Bezeichner einer Integer– oder Doublefunktion.
- Der optionale Parameter **file_handle** gibt — sofern vorhanden — den Dateideskriptor an, der das Ausgabemedium repräsentiert. Fehlt diese Angabe, so wird die Funktion in das Ein/Ausgabefenster geschrieben.
- Die Ausgabe einer Funktion hat das folgendes Format:
function_name*"("* **x_i** { *","* **x_i** } *") = "* **function_definition.**
x_i ist ein Funktionsargument, wobei **i** die natürlichen Zahlen durchläuft; es gilt: **x_i** $\in$ [*"x_1"*,*"x_2"*,*"x_3"*, ...). **function_definition** ist der Definitionsrumpf der Funktion.

Beispiele:
Die folgenden Beispiele zeigen jeweils einen Ausschnitt aus dem Ein/Ausgabefenster:

- *Der nachfolgende Befehl gibt die vordefinierte Kosinusfunktion aus:*

```
>:print_function cos
cos(x_1) = cos(x_1)
>:
```

- *Hier definiert der Anwender eine Doublefunktion* `f` *und schreibt sie anschließend in das Ein/Ausgabefenster:*

```
>:define function double f(x) = 2 * x^2 + cos(pi * x)
>:print_function f
f(x_1) = 2.0000000*x_1^2.0000000+cos(3.1415927*x_1)
>:
```

- *Als letztes soll noch die Ausgabe einer Funktion mit mehreren Funktionsvariablen vorgeführt werden:*

```
>:define function int g(x,y,z) = x * y * z + ln(x)
>:print_function g
g(x_1,x_2,x_3) = x_1*x_2*x_3+ln(x_1)
>:
```

Kapitel 8

Grafikbefehle

ProSiNN bietet dem Benutzer Möglichkeiten zur Generierung von Grafiken. Es können neue Fenster geöffnet werden, in denen sich durch Befehle Grafiken erzeugen lassen. Es handelt sich hier um Grafiken vom Typ *Bitmap*, d.h. ein Grafikpunkt ist entweder gesetzt oder nicht. Durch Kommandos lassen sich Punkte, Linien, Texte und Funktionen in ein Fenster einzeichnen. Eine weiteres Merkmal ist die grafische Protokollierung von internen Werten parallel zum Netztraining (siehe Abschnitt 12.2).

Der erste Abschnitt beschäftigt sich mit der Verwaltung von Grafikfenstern. Im Anschluß wird erklärt, wie man ein Weltkoordinatensystem setzt und ein Achsenkreuz einzeichnet. Die Ausgabebefehle werden in den nachfolgenden Unterabschnitten eingeführt. Im letzten Abschnitt wird das Kommando dargestellt, das zum Löschen eines Grafikfensterinhaltes verwendet werden muß.

8.1 Grafikfenster bearbeiten

Dieser Abschnitt ist den Mechanismen gewidmet, die zur Verwaltung von Grafikfenstern benötigt werden. Nach der Einleitung werden die Anweisungen vorgestellt, die ein Fenster öffnen und ein (oder alle) Fenster wieder schließen.

8.1.1 Grafikhandles

Um Grafikbefehle ausführen zu können, muß der Benutzer ein Grafikfenster explizit erzeugen. Nachdem ProSiNN ein neues Fenster generiert hat, wird ein *Grafikhandle* (*Grafikdeskriptor*) an den Anwender zurückgegeben. Dieses *Handle* ist bei jedem grafischen Ausgabebefehl anzugeben; es dient dabei zur eindeutigen Identifikation eines Grafikfensters. Nachdem alle gewünschten Zeichenbefehle ausgeführt worden sind, kann das Fenster nur mit Hilfe des Handles wieder geschlossen werden (die Ausnahme bildet der *close_all_grafics*–Befehl, der *alle* geöffneten Fenster wieder schließt; siehe Unterabschnitt 8.1.4).

ProSiNN erlaubt maximal 10 Grafikhandles[1], d.h. das System gestattet es, bis zu 10 Grafikfenster gleichzeitig zu verwalten. Die Syntax eines Grafikhandles lautet:

grafic_handle ::= int_const.

Bem.:

Es handelt sich also um einen Integerwert, der den Zugriff auf ein geöffnetes Grafikfenster erlaubt. Bei der aktuellen Systemkonfiguration stammen die Handles aus dem Intervall $[0, \ldots, 9]$. Das System vergibt immer das erste freie Grafikhandle beim Öffnen eines neuen Fensters.

8.1.2 Grafikfenster erzeugen: open_grafic

Der *open_grafic*–Befehl dient zum Erzeugen eines Grafikfensters. Die Syntax des Kommandos ist:

open_grafic_command	::= *"open_grafic"* **width** *","* **height** *"to"* **int_var.**
width	::= **int_expression.**
height	::= **int_expression.**
int_var	::= **„Integervariable oder Integerfeldelement".**

Bem.:

- **width** ist die Anzahl der Pixel, die die Grafikfensterbreite spezifiziert. Für diesen Wert muß gelten: **width** $\in [1, \ldots, 2000]$[2]. Die Mindestbreite, die ProSiNN immer erzeugt, ist 100 Pixel.
- **height** ist die Anzahl der Pixel, die die Grafikfensterhöhe spezifiziert. Für diesen Wert muß gelten: **height** $\in [1, \ldots, 2000]$[3]. Die Mindesthöhe, die ProSiNN immer erzeugt, ist 100 Pixel.
- Das System legt das mit dem Grafikfenster korrespondierende Handle in der Variablen **int_var** ab.
- Während der Bearbeitung des Befehls muß der Benutzer das neu generierte Fenster mit der Maus positionieren. Solange dies nicht ausgeführt wird, wartet das System.
- Jedes Grafikfenster enthält in der oberen Leiste einen *Namen*. Dieser Titel beschreibt das dazugehörige Handle (siehe Beispiel).
- Ein Fenster kann nachträglich verkleinert oder vergrößert werden. Intern bleiben aber immer die maximalen Abmesungen erhalten; d.h. wenn der Anwender z.B. ein Fenster nach dessen Erzeugung erst vergrößert und anschließend wieder verkleinert, bleiben dessen maximale Abmessungen, die nach der Vergrößerung gültig waren, erhalten. Bei der Verkleinerung eines Fensters wird immer das Zentrum des intern vorliegenden Gesamtbildes dargestellt.

[1] Dieser Wert kann bei der Systeminstallation geändert werden
[2] Der Wert für die maximale Breite eines Fensters kann bei der Systeminstallation geändert werden
[3] Der Wert für die maximale Höhe eines Fensters kann bei der Systeminstallation geändert werden

Beispiel:
Abbildung 8.1 zeigt einen möglichen Ausschnitt des Bildschirms:

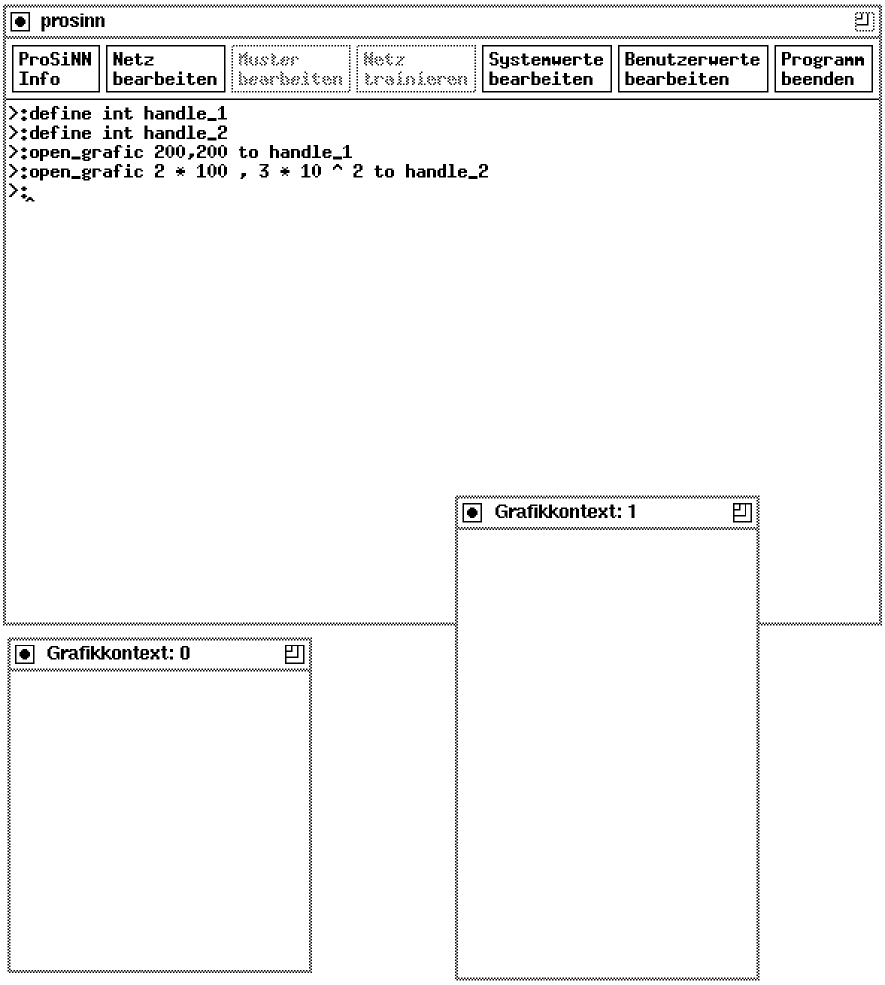

Abbildung 8.1: Erzeugung von Grafikfenstern

Der Benutzer definierte zuerst die beiden Integervariablen `handle_1` *und* `handle_2`*, die zur Erzeugung der Grafikfenster benötigt werden. Danach wurde ein Grafikfenster generiert, dessen Höhe und Breite* 200 *Pixel beträgt. Dieses Fenster hat das Handle* 0 *(der Deskriptor ist in der oberen Leiste des Fensters ablesbar:* `Grafikkontext: 0`*). Anschließend erzeugte der Anwender ein zweites Grafikfenster, dessen Breite* 200 *und dessen Höhe* 300 *Pixel beträgt. Das dazugehörige Handle* (1) *wurde in der Variablen* `handle_2` *abgelegt.*

8.1.3 Ein Grafikfenster beenden: close_grafic

Um ein bestimmtes Grafikfenster wieder zu beenden, muß der Benutzer den *close_grafic*–Befehl verwenden. Seine Syntax lautet:

close_grafic_command ::= *"close_grafic"* **grafic_handle.**

Bem.:

- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- Nach der Beendigung eines Grafikfensters geht dessen Inhalt verloren. Um eine erzeugte Grafik zu speichern, muß der Anwender auf *X Window Utilities* zurückgreifen.
- Nach Ausführung dieses Kommandos ist das Handle nicht mehr belegt. Ein weiterer Grafikbefehl, der dieses Handle verwendet, würde zu einer Fehlermeldung führen.

8.1.4 Alle Grafikfenster beenden: close_all_grafics

Um *alle* Grafikfenster wieder zu beenden, muß der Anwender die *close_all_grafics*–Anweisung benutzen. Die Syntax ist:

close_all_grafics_command ::= *"close_all_grafics"*.

Bem.:

- Nach der Beendigung der Grafikfenster gehen deren Inhalte verloren. Um eine erzeugte Grafik zu speichern, muß der Anwender auf *X Window Utilities* zurückgreifen.
- Nach Ausführung dieses Kommandos ist kein Handle mehr belegt. Ein weiterer Grafikbefehl, der ein vorher definiertes Handle verwendet, würde zu einer Fehlermeldung führen.

8.2 Weltkoordinatensystem setzen: set_world

Bei grafischen Ausgabebefehlen verwendet ProSiNN Doublewerte zur Angabe der Koordinaten. Der Benutzer muß deshalb vor dem ersten Zeichenbefehl ein Weltkoordinatensystem definieren, mit dem ProSiNN berechnen kann, welcher Pixel eine Double–Positionsangabe beschreibt. Das Koordinatensystem wird mit dem *set_world*–Befehl gesetzt, dessen Syntax folgendermaßen lautet:

set_world_command	::=	*"set_world"* **x_min** *","* **x_max** *","* **y_min** *","* **y_max** *"to"* **grafic_handle.**
x_min	::=	**double_expression.**
x_max	::=	**double_expression.**
y_min	::=	**double_expression.**
y_max	::=	**double_expression.**
grafic_handle	::=	**int_expression.**

Bem.:

- **x_min** und **x_max** sind die Werte des linken und rechten Randes des Weltkoordinatensystems. Es muß gelten: **x_min** < **x_max**.
- **y_min** und **y_max** sind die Werte des unteren und oberen Randes des Weltkoordinatensystems. Es muß gelten: **y_min** < **y_max**.
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- Das System transformiert das Weltkoordinatensystem so in die Grafik, daß an jedem Rand noch 20 Pixel Leerraum verbleibt.

Beispiel:
Im System existiere eine Grafik, deren Deskriptor 0 *ist:*

```
set_world -10.0 , 10.0 , 0.0 , 3.0 to 0
```

Dieser Befehl definiert das Weltkoordinatensystem so, daß 20 Pixel rechts vom linke Rand -10.0*, 20 Pixel links vom rechten Rand* 10.0*, 20 Pixel über dem unteren Rand* 0.0 *und 20 Pixel unterhalb des oberen Randes* 3.0 *ist.*

8.3 Koordinatenachsen einzeichnen: axes

Zum Einzeichnen vom beschrifteten Koordinatenachsen bietet ProSiNN dem Benutzer den *axes*–Befehl an. Seine Syntax ist:

axes_command	::=	*“axes“* **x_min** *“,“* **x_max** *“,“* **y_min** *“,“* **y_max** *“,“* **x_center** *“,“* **y_center** *“to“* **grafic_handle.**
x_min	::=	**double_expression.**
x_max	::=	**double_expression.**
y_min	::=	**double_expression.**
y_max	::=	**double_expression.**
x_center	::=	**double_expression.**
y_center	::=	**double_expression.**
grafic_handle	::=	**int_expression.**

Bem.:

- **x_min** und **x_max** sind die Werte des linken und rechten Randes der Koordinatenachsen. Es muß gelten: **x_min** < **x_max**.
- **y_min** und **y_max** sind die Werte des unteren und oberen Randes der Koordinatenachsen. Es muß gelten: **y_min** < **y_max**.

- **x_center** und **y_center** bilden die Position des Koordinatenachsenursprungs. Diese Werte werden in der Nähe des Ursprungs in Klammern eingezeichnet (siehe Beispiel). Sie dürfen beliebig gesetzt werden; verlassen sie aber den Bereich, der durch die oben angegebenen Koordinatenwerte gesetzt wurde, wird der Ursprung an den entsprechenden Rand verschoben.
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.
- Liegt der Bereich, der durch **x_min**, **x_max**, **y_min** und **y_max** angegeben wurde, nicht im Weltkoordinatensystem, so ist das Achsenkreuz nicht sichtbar.

Beispiel:
Im System seien vier Grafikfenster vorhanden (mit den Handles $0, 1, 2$ *und* 3*). In jedem Fenster wurde das Weltkoordinatensystem mit dem folgenden Befehl gesetzt (i entspricht dem jeweiligen Handle):*

```
set_world -10.0 , 10.0 , -5.0 , 10.0 to i
```

Die anschließenden Kommandos zum Einzeichnen der Koordinatenachsen erzeugen die in Abbildung 8.2 dargestellten Grafiken:

```
axes -10.0 , 10.0 , -5.0 , 10.0 ,    0.0 ,    2.5 to 0
axes -10.0 , 10.0 , -5.0 , 10.0 ,    0.0 ,    0.0 to 1
axes -10.0 , 10.0 , -5.0 , 10.0 , -100.0 , -100.0 to 2
axes -10.0 , 10.0 , -5.0 , 10.0 ,    0.0 ,  100.0 to 3
```

8.4 Strichlierungsangaben

ProSiNN bietet dem Benutzer eine zusätzliche Option zur Unterscheidung eingezeichneter Grafikelemente. Er kann angeben, ob beim Zeichnen einer Linie, einer Funktion oder beim grafischen Protokollieren (parallel zum Netztraining) einzelne Ausgaben *gestrichelt* erfolgen sollen. In diesem Abschnitt wird die Spezifikation einer solchen Strichlierung definiert:

dotted_specification	::=	*"[*" **block_1** *","* **gap_1** *","* **block_2** *","* **gap_2** *"]*".
block_1	::=	**int_expression.**
gap_1	::=	**int_expression.**
block_2	::=	**int_expression.**
gap_2	::=	**int_expression.**

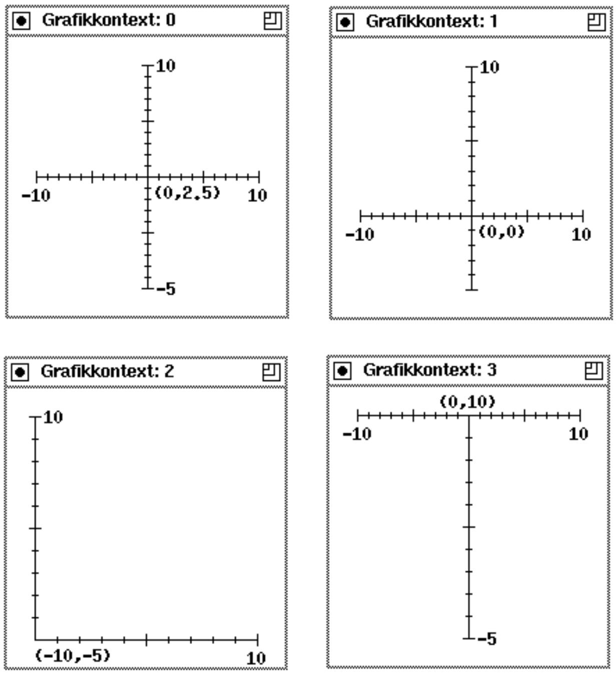

Abbildung 8.2: Einzeichnen von Koordinatenachsen

Bem.:

- Bei dieser Spezifikation werden Intervallangaben verwendet. Muß ProSiNN z.B. beim Zeichnen einer gestrichelten Linie entscheiden, ob ein Pixel zu setzen ist, überprüft das System in welchem Intervall der Pixel liegt. Befindet er sich in einem Bereich, der einzuzeichnen ist, wird der Pixel gesetzt; ansonsten wird der Pixel nicht eingezeichnet. Dabei wird das Gesamtintervall immer wieder periodisch durchlaufen.
- **block_1** ist die Länge des ersten Teilintervalls in Pixel. Die Pixel in diesem Bereich werden eingezeichnet. Es muß gelten: **block_1** ≥ 0.
- **gap_1** ist die Länge des zweiten Teilintervalls in Pixel. Die Pixel in diesem Bereich werden nicht eingezeichnet. Es muß gelten: **gap_1** ≥ 0.
- **block_2** ist die Länge des dritten Intervalls in Pixel. Die Pixel in diesem Bereich werden eingezeichnet. Es muß gelten: **block_2** ≥ 0.
- **gap_2** ist die Länge des letzten Teilintervalls in Pixel. Die Pixel in diesem Bereich werden nicht eingezeichnet. Es muß gelten: **gap_2** ≥ 0.

Beispiele:

- *Der Benutzer möchte eine gestrichelte Linie in ein Grafikfenster einzeichnen; er verwendet dazu die Strichlierungsangabe* `[ 5 , 10 , 2 , 7 ]`. *Wenn das System die Linie zeichnet, werden die Pixel in folgender Reihenfolge gesetzt (oder nicht gesetzt):*

5 Pixel werden gesetzt
10 Pixel werden nicht gesetzt
2 Pixel werden gesetzt
7 Pixel werden nicht gesetzt
Wiederholung:
5 Pixel werden gesetzt
10 Pixel werden nicht gesetzt
2 Pixel werden gesetzt
7 Pixel werden nicht gesetzt
...Wiederholung...

- *Konkrete Beispiele findet man u.a. in Bild 8.4 des Unterabschnitts 8.5.2 und in Abbildung 8.6 des Abschnitts 8.6.*

8.5 Zeichenbefehle

In diesem Abschnitt werden die wichtigsten grafischen Ausgabekommandos vorgestellt. Dies sind die Befehle, die Punkte, Linien und Texte in ein Grafikfenster einzeichnen.

8.5.1 Punkt ausgeben: plot

Der *plot*–Befehl setzt einen Punkt in einem Grafikfenster. Die Syntax des Kommandos lautet:

plot_command	::=	*"plot"* **x** *","* **y** *"to"* **grafic_handle.**
x	::=	**double_expression.**
y	::=	**double_expression.**
grafic_handle	::=	**int_expression.**

Bem.:

- **x** beschreibt die x-Position im Weltkoordinatensystem.
- **y** beschreibt die y-Position im Weltkoordinatensystem.
- **grafic_handle** muß ein Grafikdeskriptor sein, der ein vom Anwender geöffnetes Grafikfenster spezifiziert.

Beispiel:
Bild 8.3 zeigt ein Beispiel, bei dem die Identitätsfunktion mit dem plot*–Befehl eingezeichnet wurde. Das nach der Abbildung dargestellte Programm generierte diese Grafikausgabe.*