Kiel Interactive Evaluation Laboratory: Eine Arbeitsumgebung zur Visualisierung von Termersetzungssemantik

Diplomarbeit

vorgelegt von

Markus Tiedt

im Juli 1999

Betreuer: Prof. Dr. R. Berghammer

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik

Inhaltsübersicht

Kapitel 1

Einleitung

KIEL ist das Akronym zu Kiel Interactive Evaluation Laboratory. Wie der Name schon andeutet, ist es ein Anliegen, dem Benutzer eine Arbeitsumgebung bereitzustellen, in der er die Auswertung von Ausdrücken bezüglich verschiedener Auswertungsarten vornehmen kann. Dazu zeigt KIEL eingegebene Ausdrücke grafisch als einen Baum an und interagiert mit dem Benutzer, indem es verschiedene Operationen zur Verfügung stellt, die der Benutzer per Mausklick auswählen kann.

Ausdrücke sind Terme über Rechenvorschriften. Deshalb bietet es sich an, eine deklarative Programmiersprache als Grundlage zu benutzen. KIEL stützt sich in der Syntax der Ausdrücke und Rechenvorschriften auf die funktionale Programmiersprache ML [Ull94]. ML ist gegen Ende der 70er Jahre an der Universität Edinburgh ursprünglich als Implementierungssprache eines größeren Programmsystems mit dem Namen LCF entwickelt worden und bekam daher seinen Namen als Abkürzung von <u>m</u>eta <u>l</u>anguage for LCF.

Die von KIEL akzeptierte Sprache ist eine Teilsprache von ML, die im wesentlichen aus Ausdrücken über den Standardtypen `unit`[1], `bool`, `int`, `real` und `string` sowie Sequenzen über den Standardtypen besteht. Eine genaue Beschreibung durch Syntaxdiagramme findet sich im Anhang A.

Aufgrund der Tatsache, daß KIEL unter anderem eine schrittweise Auswertung der eingegebenen ML-Ausdrücke vornehmen kann, kann es auch als ein einfaches Debug-Programm für ML-Rechenvorschriften genutzt werden. Hiermit können zum Beispiel die Gründe einer Nichtterminierung wesentlich besser untersucht werden als mit dem Standard-ML-System.

Die Arbeit gliedert sich in folgender Weise: In Kapitel 2 werden neben der Auswertung von ML die Grundlagen der Semantik mittels Termersetzung behandelt. Es folgt in Kapitel 3 eine Beschreibung aller Komponenten von KIEL, mit denen der Benutzer zu tun hat. Neben sämtlichen Fenstern und Aktionen, die in ihnen ausgeführt werden können, wird hier auch auf den Aufbau und die Wirkungsweise der Ressourcedateien eingegangen. Um den Einstieg in das Arbeiten mit KIEL zu erleichtern, widmet sich das Kapitel 4 einigen Bearbeitungsbeispielen, welche unmittelbar vom Benutzer nachvollzogen werden können. Außerdem sind an dieser Stelle verschiedene Statistiken angegeben, die unter anderem die Unterschiede der Termersetzungssemantiken veranschaulichen. In Kapitel 5 wird auf Implementierungsdetails eingegangen. Zu Beginn werden die Systemkomponenten, in die sich KIEL gliedert, sowie die verwendeten Strukturen, globalen Variablen und Konstanten beschrieben, damit sich eine Person, die den über 11000 Zeilen umfassenden Quellcode vorliegen hat, in diesem leichter zurechtfinden kann. Außerdem wird in einem Abschnitt näher auf das Baumfenster eingegangen, welches in KIEL eine zentrale Rolle spielt. Schließlich werden einige ausgewählte C-Funktionen im Detail beschrieben. Wie oben bereits erwähnt, folgt im Anhang A eine Auflistung von Syntaxdiagrammen, die die von KIEL akzeptierte Sprache spezifizieren.

[1] Alle Zeichenreihen, die Eingaben von ML oder KIEL darstellen, werden in dieser Arbeit in Schreibmaschinenschrift angezeigt, um ihre Sonderstellung zu verdeutlichen.

Kapitel 2

Termmäßige Auswertung von ML

Dieses Kapiel widmet sich den Grundlagen der termmäßigen Auswertung von Ausdrücken, auf die funktionale Programmiersprachen aufgebaut sind. Da KIEL an ML orientiert ist, wird die Syntax und die Semantik von ML behandelt. Im Abschnitt 2.1 wird auf die Grundlagen von Termen eingegangen. Hierbei werden die Syntax und die Semantik von Signaturen und Termen näher betrachtet. Dadurch ist es möglich, einfachen ML-Ausdrücken, also solchen ohne Rechenvorschriftsaufrufe und Fallunterscheidungen, einen Wert zuzuordnen. Die Teile der Syntax und der Semantik von ML, die in KIEL implementiert sind, werden in 2.2 beschrieben. Neben den Standard- und den Sequenztypen werden Rechenvorschriften und Systeme von Rechenvorschriften definiert, die das zentrale Konzept funktionaler Programmiersprachen darstellen. Erst durch das Vorhandensein von Rechenvorschriften und Fallunterscheidungen wird die Semantik in funktionalen Programmiersprachen für die Programmierung interessant. Auf diese beiden Erweiterungen bezüglich 2.1 wird in dem Abschnitt 2.3 eingegangen. Hierzu ist das Konzept der allgemeinen Termersetzung zu behandeln. Es ist grundlegend für die Semantik aller funktionalen Programmiersprachen.

2.1 Signaturen und Terme

In diesem Abschnitt werden Signaturen und Terme behandelt. Der Wert von Termen über der ML-Signatur, die keine Rechenvorschriftsaufrufe und Fallunterscheidungen enthalten, kann nach diesem Konzept bestimmt werden. Zunächst folgt die Definition von Signaturen und Termen. Der zweite Teil widmet sich der Semantik von Signaturen und Termen. Hierzu wird neben dem Begriff der Σ-Algebra auch der des Wertes eines Terms definiert.

2.1.1 Syntax von Signaturen und Termen

Terme werden über Signaturen gebildet. Die Signaturen stellen die Sorten-, Konstanten- und Funktionensymbole zur Verfügung, aus denen Terme nach gewissen Regeln gebildet werden können.

Definition 2.1 (Signatur) *Eine* **Signatur** Σ *ist ein Tripel* (S, K, F) *von drei disjunkten Mengen, wobei gilt:*

a) S *ist eine nichtleere Menge von* **Sorten**.

b) K *ist eine Menge von* **Konstantensymbolen**, *und jedem* $c \in K$ *ist genau ein Element* $s \in S$ *zugeordnet.* c *heißt dann* **Konstantensymbol der Sorte** s.

c) *F ist eine Menge von* **Funktionssymbolen**, *und jedem $f \in F$ ist genau ein Element $w \in S^+$ und genau ein Element $s \in S$ zugeordnet. f heißt dann* **Funktionssymbol** *mit* **Argumentsorten** *w,* **Ergebnissorte** *s und* **Funktionalität** *$w \to s$.* □

Auf einer Signatur aufbauend, werden Terme induktiv über ihren Aufbau definiert. Neben Konstantensymbolen können in einem Term Variable und Funktionssymbole auftreten.

Definition 2.2 (Terme über Signaturen) *Es sei $\Sigma = (S, K, F)$ eine Signatur und X eine zu $S \cup K \cup F$ disjunkte Menge von* **Variablen**, *wobei jedem $x \in X$ genau eine Sorte $s \in S$ zugeordnet ist. Die Menge $\mathcal{T}^s_\Sigma(X)$ der* **Σ-Terme der Sorte s ∈ S mit Variablen aus X** *ist induktiv wie folgt definiert:*

a) *Für jedes Konstantensymbol c der Sorte s ist $c \in \mathcal{T}^s_\Sigma(X)$.*

b) *Für jede Variable x der Sorte s ist $x \in \mathcal{T}^s_\Sigma(X)$.*

c) *Für jedes Funktionssymbol f der Funktionalität $s_1 \dots s_k \to s$ und $t_i \in \mathcal{T}^{s_i}_\Sigma(X)$, $1 \le i \le k$, ist $f(t_1, \dots, t_k) \in \mathcal{T}^s_\Sigma(X)$.* □

Zur Verbesserung der Lesbarkeit gibt es in fast allen Programmiersprachen — auch in ML — vereinfachende Schreibweisen für die Darstellung von Termen. Es kommt zur Verwendung von Infix-, Präfix- und Postfixnotation. So steht $(\sim 101) + (17 * 42)$ anstelle von $plus(negativ(101), mult(17, 42))$. Außerdem gelten Vorrangsregeln, wie zum Beispiel Prioritätsregeln zur Klammerersparnis, so daß $\sim 101 + 17 * 42$ anstelle von $(\sim 101) + (17 * 42)$ geschrieben werden kann.

Definition 2.3 (gewisse Termmengen) *Es sei $\Sigma = (S, K, F)$ eine Signatur und X Variablenmenge.*

a) *$\mathcal{T}_\Sigma(X) := \bigcup_{s \in S} \mathcal{T}^s_\Sigma(X)$ ist die* **Menge aller Σ-Terme**.

b) *Ein Term $t \in \mathcal{T}_\Sigma(\emptyset)$ heißt* **geschlossen** *oder* **Σ-Grundterm**.

c) *$\mathcal{T}^s_\Sigma := \mathcal{T}^s_\Sigma(\emptyset)$ ist die* **Menge der Σ-Grundterme der Sorte s ∈ S**.

d) *$\mathcal{T}_\Sigma := \bigcup_{s \in S} \mathcal{T}^s_\Sigma$ ist die* **Menge der Σ-Grundterme**. □

2.1.2 Semantik von Signaturen und Termen

Die Semantik zu einer Signatur wird durch die entsprechende Σ-Algebra bestimmt. In ihr wird jedem Sorten-, Konstanten- und Funktionssymbol eine Sorte, Konstante bzw. Funktion zugeordnet.

Definition 2.4 (Σ-Algebra) *Sei $\Sigma = (S, K, F)$ eine Signatur. Eine* **Algebra zur Signatur Σ**, *kurz* **Σ-Algebra**, *ist ein Tripel $A = (S^A, K^A, F^A)$, wobei gilt:*

a) *$S^A := \{s^A \mid s \in S\}$, wobei jedes s^A eine nichtleere Menge ist. s^A heißt* **Träger** *oder* **Interpretation von s**.

b) *$K^A := \{c^A \mid c \in K\}$, wobei $c^A \in \bigcup_{s \in S} s^A$, und $c^A \in s^A$ genau dann, wenn s die Sorte von c ist. c^A heißt die* **Interpretation der Konstanten c**.

c) *$F^A := \{f^A \mid f \in F\}$, wobei $f^A \in \bigcup_{(s_1 \dots s_k, s) \in S^+ \times S} (s^A)^{\prod_{i=1}^{k} {s_i}^A}$, und $f^A \in (s^A)^{\prod_{i=1}^{k} {s_i}^A}$ genau dann, wenn $s_1 \dots s_k \to s$ die Funktionalität von f ist. f^A ist eine partielle Funktion und heißt die* **Interpretation von** *f.* □

Da in einem Term Variable vorhanden sein können, ist es notwendig, den Wert einer Variablen zu kennen, um den Wert des Terms bestimmen zu können. Diese Aufgabe wird von der Belegung einer Variablenmenge übernommen.

Definition 2.5 (Belegung) *Sei $\Sigma = (S, K, F)$ Signatur, $A = (S^A, K^A, F^A)$ Σ-Algebra und X Variablenmenge mit Sorten aus S. Eine* **Belegung (von X in A)** *ist eine Funktion $\nu : X \to \bigcup_{s \in S} s^A$ mit der Eigenschaft, daß für alle $x \in X$ gilt: $\nu(x) \in s^A$ genau dann, wenn s die Sorte von x ist.* □

Mit Hilfe der entsprechenden Σ-Algebra und Belegung, kann der Wert eines Terms induktiv über seinen Aufbau definiert werden.

Definition 2.6 (Wert eines Terms) *Sei $\Sigma = (S, K, F)$ Signatur, $A = (S^A, K^A, F^A)$ Σ-Algebra, X Variablenmenge und $\nu : X \to \bigcup_{s \in S} s^A$ Variablenbelegung. Der* **Wert** $\mathcal{I}[t]^A_\nu$ **eines Terms** *t ist induktiv wie folgt definiert:*

a) Für alle $c \in K$ ist $\mathcal{I}[c]^A_\nu = c^A$.

b) Für alle $x \in X$ ist $\mathcal{I}[x]^A_\nu = \nu(x)$.

c) Für alle $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(X)$ ist $\mathcal{I}[f(t_1, \ldots, t_k)]^A_\nu = f^A(\mathcal{I}[t_1]^A_\nu, \ldots, \mathcal{I}[t_k]^A_\nu)$. □

Hierbei ist zu beachten, daß die Funktionsanwendung bei *c)* strikt ist, ein undefiniertes Argument also zu einem undefinierten Ergebnis führt.

2.2 ML

Die Signatur zu ML stellt die Sorten, Konstantensymbole und Funktionssymbole zur Verfügung. Die Basissorten sind `unit`, `bool`, `int`, `real` und `string`. Eine Gliederung der Konstanten- und Funktionssymbole anhand dieser Sorten führt zu einer Klassenbildung. Diese Klassen werden als **elementare Datenstrukturen** bezeichnet. In diesem Zusammenhang werden Konstantensymbole auch Konstanten und Funktionssymbole auch Operationen genannt.

In KIEL wurde eine Teilmenge der ML-Signatur implementiert. So kennt KIEL neben den oben genannten Basissorten lediglich noch Sequenztypen. ML hat darüber hinaus noch einige weitere Sorten, Konstanten und Operationen, die die gewünschte Veranschaulichung der Semantiken jedoch nicht vergrößern. Zu der ML-Signatur gibt es eine Algebra. Diese Algebra sei A. Durch sie wird die Semantik der Sorten, Konstanten und Operationen der Signatur festgelegt. Für das Studium der in KIEL nicht implementierten Teile der ML-Signatur sei auf die komplette Zusammenstellung der ML-Signatur und -Algebra in [SML93] sowie auf die ausführlicheren Beschreibungen in [Ull94] verwiesen. In den Unterabschnitten 2.2.1 bis 2.2.6 werden die Teile der ML-Signatur und -Algebra beschrieben, mit denen in KIEL gearbeitet werden kann. Die Sorten, Konstanten und Operationen werden dabei in Klassen eingegliedert behandelt.

Der Termbegriff in ML ist im Vergleich zu 2.1 erweitert. Neben der Fallunterscheidung ist vor allem auch Rekursion durch Rechenvorschriften (siehe 2.2.7) zugelassen. Diese führt dazu, daß die Semantik komplexer wird. Die Definition des Wertes eines Terms aus 2.1.2 bezieht sich nur auf Terme ohne Rekursion. Die Erweiterung der Semantik auf (rekursive) Rechenvorschriften ist in 2.3 beschrieben.

2.2.1 Einelementiger Bereich

Signatur

Sorte	:	`unit`
Konstantensymbole	:	`()` : `unit`
Funktionssymbole	:	`=` : `unit`, `unit` $\rightarrow$ `bool`
		`<>` : `unit`, `unit` $\rightarrow$ `bool`

Algebra

`unit`	:	Einelementige Menge — $\texttt{unit}^A = \{\texttt{()}\}$
`()`	:	Einziges Element
$\mathbf{u_0}$ `=` $\mathbf{u_1}$	:	Gleichheit von $\mathbf{u_0}$ und $\mathbf{u_1}$[1]
$\mathbf{u_0}$ `<>` $\mathbf{u_1}$	:	Ungleichheit von $\mathbf{u_0}$ und $\mathbf{u_1}$[2]

ML bietet die Möglichkeit, zusammengesetzte Datenstrukturen[3] zu erzeugen. Mit dem einelementigen Bereich können solche generiert werden.

2.2.2 Wahrheitswerte

Signatur

Sorte	:	`bool`
Konstantensymbole	:	`true` : `bool`
		`false` : `bool`
Funktionssymbole	:	`not` : `bool` $\rightarrow$ `bool`
		`andalso` : `bool`, `bool` $\rightarrow$ `bool`
		`orelse` : `bool`, `bool` $\rightarrow$ `bool`
		`=` : `bool`, `bool` $\rightarrow$ `bool`
		`<>` : `bool`, `bool` $\rightarrow$ `bool`

Algebra

`bool`	:	Menge der Wahrheitswerte — $\texttt{bool}^A = \mathbb{B} = \{tt, \mathit{ff}\}$
`true`	:	logisch wahr — $\texttt{true}^A = tt$
`false`	:	logisch falsch — $\texttt{false}^A = \mathit{ff}$
`not(`$\mathbf{b}$`)`	:	logische Negation von $\mathbf{b} \in \mathbb{B}$
$\mathbf{b_0}$ `andalso` $\mathbf{b_1}$	:	Sequentielle Konjunktion von $\mathbf{b_0}$ und $\mathbf{b_1}$
$\mathbf{b_0}$ `orelse` $\mathbf{b_1}$	:	Sequentielle Disjunktion von $\mathbf{b_0}$ und $\mathbf{b_1}$
$\mathbf{b_0}$ `=` $\mathbf{b_1}$	:	Gleichheit von $\mathbf{b_0}$ und $\mathbf{b_1}$
$\mathbf{b_0}$ `<>` $\mathbf{b_1}$	:	Ungleichheit von $\mathbf{b_0}$ und $\mathbf{b_1}$

Bis auf `andalso` und `orelse` sind alle Operationen auf den elementaren Datenstrukturen in ML strikt, das bedeutet, daß sie als Ergebnis keinen definierten Wert[4] liefern, sobald eines ihrer Argumente undefiniert ist.

Die Operationen `andalso` und `orelse` werden sequentiell, also von links nach rechts, ausgewertet. Dies resultiert aus der Tatsache, daß das Ergebnis der Auswertung schon nach Ermittlung des ersten Arguments

[1] Da es nur ein Element vom Typ `unit` gibt, liefert der Test auf Gleichheit bei definierten Argumenten stets der Wert tt (siehe 2.2.2).

[2] Da es nur ein Element vom Typ `unit` gibt, liefert der Test auf Ungleichheit bei definierten Argumenten stets der Wert ff (siehe 2.2.2).

[3] In KIEL sind zusammengesetzte Datenstrukturen nicht implementiert. Neben den elementaren Datenstrukturen kennt KIEL nur noch Sequenzen über elementare Datenstrukturen (siehe 2.2.6).

[4] Undefinierte Werte sind im Baumfenster von KIEL (siehe 3.3.6) als Fehlerknoten dargestellt. Die Auswertung eines Knotens läßt diesen also — mit Ausnahme der `andalso`- und `orelse`-Knoten — zum Fehlerknoten werden, falls eines seiner Kinder im Auswertungsbaum ein Fehlerknoten ist.

feststehen kann. Hat das erste Argument von `andalso` den Wert *ff*, so hat auch das Ergebnis den Wert *ff*. Entsprechend hat das Ergebnis einer `orelse`-Auswertung den Wert *tt*, falls dies für das erste Argument gilt. Die folgenden Wertetafeln geben eine Übersicht über die Ergebnisse von `andalso`- und `orelse`-Auswertungen:

$\texttt{andalso}^A$	*tt*	*ff*	$\perp$[5]
tt	*tt*	*ff*	$\perp$
ff	*ff*	*ff*	*ff*
$\perp$	$\perp$	$\perp$	$\perp$

$\texttt{orelse}^A$	*tt*	*ff*	$\perp$
tt	*tt*	*tt*	*tt*
ff	*tt*	*ff*	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

2.2.3 Ganze Zahlen

Signatur

Sorte	:	`int`
Konstantensymbole	:	Alle vorzeichenbehafteten 31-Bit-Ganzzahlen
Funktionssymbole	:	`~` : `int` $\to$ `int`
		`+` : `int`, `int` $\to$ `int`
		`-` : `int`, `int` $\to$ `int`
		`*` : `int`, `int` $\to$ `int`
		`div` : `int`, `int` $\to$ `int`
		`mod` : `int`, `int` $\to$ `int`
		`min` : `int`, `int` $\to$ `int`
		`max` : `int`, `int` $\to$ `int`
		`abs` : `int` $\to$ `int`
		`=` : `int`, `int` $\to$ `bool`
		`<>` : `int`, `int` $\to$ `bool`
		`>` : `int`, `int` $\to$ `bool`
		`>=` : `int`, `int` $\to$ `bool`
		`<` : `int`, `int` $\to$ `bool`
		`<=` : `int`, `int` $\to$ `bool`

Algebra

`int`	:	$\texttt{int}^A = \{-2^{30}, \ldots, 2^{30}-1\}$
`~ i`	:	Negation von **i**
$\mathbf{i_0}$ `+` $\mathbf{i_1}$	:	Summe von $\mathbf{i_0}$ und $\mathbf{i_1}$
$\mathbf{i_0}$ `-` $\mathbf{i_1}$	:	Differenz von $\mathbf{i_0}$ und $\mathbf{i_1}$
$\mathbf{i_0}$ `*` $\mathbf{i_1}$	:	Produkt von $\mathbf{i_0}$ und $\mathbf{i_1}$
$\mathbf{i_0}$ `div` $\mathbf{i_1}$	:	Ganzzahliger Quotient von $\mathbf{i_0}$ und $\mathbf{i_1}$[6]
$\mathbf{i_0}$ `mod` $\mathbf{i_1}$	:	Rest bei ganzzahliger Division von $\mathbf{i_0}$ und $\mathbf{i_1}$[7]
`min(`$\mathbf{i_0}$`,`$\mathbf{i_1}$`)`	:	Minimum von $\mathbf{i_0}$ und $\mathbf{i_1}$
`max(`$\mathbf{i_0}$`,`$\mathbf{i_1}$`)`	:	Maximum von $\mathbf{i_0}$ und $\mathbf{i_1}$
`abs(i)`	:	Absoluter Wert von **i**
$\mathbf{i_0}$ `=` $\mathbf{i_1}$	:	Gleichheit von $\mathbf{i_0}$ und $\mathbf{i_1}$
$\mathbf{i_0}$ `<>` $\mathbf{i_1}$	:	Ungleichheit von $\mathbf{i_0}$ und $\mathbf{i_1}$
$\mathbf{i_0}$ `>` $\mathbf{i_1}$	:	Test, ob $\mathbf{i_0} > \mathbf{i_1}$
$\mathbf{i_0}$ `>=` $\mathbf{i_1}$	:	Test, ob $\mathbf{i_0} \geq \mathbf{i_1}$
$\mathbf{i_0}$ `<` $\mathbf{i_1}$	:	Test, ob $\mathbf{i_0} < \mathbf{i_1}$
$\mathbf{i_0}$ `<=` $\mathbf{i_1}$	:	Test, ob $\mathbf{i_0} \leq \mathbf{i_1}$

[5] Das Symbol $\perp$ steht für „undefiniert".

[6] Der Quotient von $\mathbf{i_0}$ und $\mathbf{i_1}$ wird in Richtung $-\infty$ gerundet.

[7] $\mathbf{i_0}$ `mod` $\mathbf{i_1}$ liefert als Ergebnis ein k vom Typ `int` mit der Eigenschaft, daß $0 \leq k < |\mathbf{i_1}|$ und $\mathbf{i_1} * (\mathbf{i_0}\ \texttt{mod}\ \mathbf{i_1}) + k = \mathbf{i_0}$ ist.

Die Operationen `min`, `max`, `=`, `<>`, `>`, `>=`, `<` und `<=` sind total. Ihr Ergebnis ist nur dann undefiniert, wenn mindestens eines ihrer Argumente undefiniert ist.

Es ist keiner Programmiersprache möglich, alle Ganzzahlen zu unterstützen. Aus diesem Grund wird immer eine Auswahl von ihnen in der Art getroffen, daß der überdeckte Bereich in möglichst vielen Fällen genügt. In der Regel besteht der unterstützte Zahlenbereich aus aufeinanderfolgenden ganzen Zahlen. Mit welcher Zahl dieser Bereich beginnt und mit welcher Zahl er endet, hängt sowohl von der Anzahl der zur Verfügung stehenen Bits pro Zahl ab als auch davon, ob nur natürliche Zahlen zur Verfügung gestellt werden sollen oder ob auch auf negative ganze Zahlen zugegriffen werden können soll. Werden nur natürliche Zahlen erlaubt, ist die kleinste zur Verfügung stehende Zahl Null. Die größte unterstützte Zahl hängt von der Anzahl der zur Verfügung stehenden Bits ab. Sind k Bits verfügbar, so ist die größte Zahl $2^k - 1$. Solche Zahlenbereiche werden **vorzeichenlos** genannt.

Das Gegenstück hierzu bilden die **vorzeichenbehafteten** Zahlenbereiche. Sie erstrecken sich möglichst symmetrisch um den Nullpunkt. Aufgrund der internen Bearbeitung der Bits erstrecken sich solche Zahlenbereiche, falls insgesamt k Bits für eine Zahl zur Verfügung stehen, von -2^{k-1} bis $2^{k-1} - 1$. Wird vereinfachend von k-Bit-Ganzzahlen gesprochen, ist immer der vorzeichenbehaftete Zahlenbereich gemeint, da dieser auch nichtnatürliche Zahlen enthält.

Bei den Operationen `~`, `+`, `-`, `*`, `div` und `abs` kann es dazu kommen, daß ihr Ergebnis außerhalb des zugrundeliegenden Zahlenbereiches liegt. In einem solchen Fall ist es nicht möglich, das Resultat durch eine 31-Bit-Ganzzahl darzustellen. Abhängig davon, ob das Resultat zu groß oder zu klein ist, kommt es zu einem Über- bzw. Unterlauffehler. Schließlich haben die Operationen `div` und `mod` kein definiertes Ergebnis, falls ihr zweites Argument den Wert Null hat.

2.2.4 Reelle Zahlen

Signatur

Sorte	:	`real`	
Konstantensymbole	:	Alle Gleitkommazahlen doppelter Präzision	
Funktionssymbole	:	`~`	: `real` $\rightarrow$ `real`
		`+`	: `real,real` $\rightarrow$ `real`
		`-`	: `real,real` $\rightarrow$ `real`
		`*`	: `real,real` $\rightarrow$ `real`
		`/`	: `real,real` $\rightarrow$ `real`
		`abs`	: `real` $\rightarrow$ `real`
		`sqrt`	: `real` $\rightarrow$ `real`
		`sin`	: `real` $\rightarrow$ `real`
		`cos`	: `real` $\rightarrow$ `real`
		`exp`	: `real` $\rightarrow$ `real`
		`ln`	: `real` $\rightarrow$ `real`
		`real`	: `int` $\rightarrow$ `real`
		`truncate`	: `real` $\rightarrow$ `int`
		`=`	: `real,real` $\rightarrow$ `bool`
		`<>`	: `real,real` $\rightarrow$ `bool`
		`>`	: `real,real` $\rightarrow$ `bool`
		`>=`	: `real,real` $\rightarrow$ `bool`
		`<`	: `real,real` $\rightarrow$ `bool`
		`<=`	: `real,real` $\rightarrow$ `bool`

Algebra

`real`	:	Menge der Gleitkommazahlen doppelter Präzision
`~ r`	:	Negation von **r**
$\mathbf{r_0}$ `+` $\mathbf{r_1}$	:	Summe von $\mathbf{r_0}$ und $\mathbf{r_1}$
$\mathbf{r_0}$ `-` $\mathbf{r_1}$	:	Differenz von $\mathbf{r_0}$ und $\mathbf{r_1}$
$\mathbf{r_0}$ `*` $\mathbf{r_1}$	:	Produkt von $\mathbf{r_0}$ und $\mathbf{r_1}$
$\mathbf{r_0}$ `/` $\mathbf{r_1}$	:	Quotient von $\mathbf{r_0}$ und $\mathbf{r_1}$
`abs(r)`	:	Absoluter Wert von **r**
`sqrt(r)`	:	Quadratwurzel von **r**
`sin(r)`	:	Sinus von **r**
`cos(r)`	:	Cosinus von **r**
`exp(r)`	:	Wert der Exponentialfunktion mit dem Argument **r**
`ln(r)`	:	Natürlicher Logarithmus von **r**
`real(i)`	:	Konvertierung von **i** in eine Gleitkommazahl
`truncate(r)`	:	Konvertierung von **r** in eine Ganzzahl[8]
$\mathbf{r_0}$ `=` $\mathbf{r_1}$	:	Gleichheit von $\mathbf{r_0}$ und $\mathbf{r_1}$
$\mathbf{r_0}$ `<>` $\mathbf{r_1}$	:	Ungleichheit von $\mathbf{r_0}$ und $\mathbf{r_1}$
$\mathbf{r_0}$ `>` $\mathbf{r_1}$	:	Test, ob $\mathbf{r_0} > \mathbf{r_1}$
$\mathbf{r_0}$ `>=` $\mathbf{r_1}$	:	Test, ob $\mathbf{r_0} \geq \mathbf{r_1}$
$\mathbf{r_0}$ `<` $\mathbf{r_1}$	:	Test, ob $\mathbf{r_0} < \mathbf{r_1}$
$\mathbf{r_0}$ `<=` $\mathbf{r_1}$	:	Test, ob $\mathbf{r_0} \leq \mathbf{r_1}$

Die Darstellung der Gleitkommazahlen erfolgt mit Mantisse und Exponent. Die Mantisse muß mindestens eine Nachkommastelle haben. Ihr folgen optional der Buchstabe `e`[9] und eine ganze Zahl. Zum Beispiel steht `17.42E96` für die Zahl $17.42 \cdot 10^{96}$. Wird kein Exponent angegeben, so wird als Exponent Null gesetzt.

Die Operationen `~`, `abs`, `sin`, `cos`, `real`, `=`, `<>`, `>`, `>=`, `<` und `<=` sind total.

Ähnlich wie die ganzen Zahlen können auch die reellen Zahlen nicht alle von einer Programmiersprache unterstützt werden. Die Darstellung reeller Zahlen unterscheidet sich jedoch grundsätzlich von der Darstellung ganzer Zahlen. Je mehr Bits für die Speicherung einer reellen Zahl zur Verfügung stehen, desto genauer kann diese dargestellt werden. Somit ist ein Kompromiß zwischen Speicherplatzverbrauch und Genauigkeit der Darstellung zu finden. Hierzu stehen in der Regel unter anderem reelle Zahlen mit einfacher und mit doppelter Genauigkeit zur Verfügung. ML und KIEL verwenden als Grundlage für die Darstellung reeller Zahlen solche mit doppelter Präzision.

Bei den Operationen `+`, `-`, `*`, `/`, `exp` und `truncate` kann es zu einer Bereichsüber- und -unterschreitung kommen. Das Ergebnis der Operation `/` ist des weiteren in dem Fall undefiniert, wenn der Wert des Arguments gleich Null[10] ist. Für Argumente kleiner als Null[11] hat die Operation `sqrt` keinen definierten Wert. Schließlich ist die Operation `ln` nur für Argumente mit einem Wert echt größer als Null[12] definiert.

[8]Die Konvertierung einer Gleitkommazahl in eine Ganzzahl geschieht bei `truncate` durch Abhacken der Nachkommastellen der Gleitkommazahl. Also gilt:

$$\texttt{truncate(r)} = \begin{cases} \lfloor \mathbf{r} \rfloor & : \mathbf{r} \geq 0 \\ \lceil \mathbf{r} \rceil & : \mathbf{r} < 0 \\ \perp & : \mathbf{r} = \perp \end{cases}$$

[9]Sowohl `e` als auch `E` kann zur Einleitung des Exponenten verwendet werden.

[10]Zahlen mit einem kleineren Absolutwert als die kleinste positive reelle Zahl doppelter Genauigkeit werden als Null interpretiert. Also ist die Operation `/` genaugenommen auch dann undefiniert, falls der Absolutwert des Arguments kleiner als diese Zahl ist.

[11]Entsprechend der vorigen Fußnote ist die Operation `sqrt` genau für die Werte definiert, deren Wert in doppelt präziser Darstellung größer oder gleich Null sind.

[12]Analog zu den vorigen beiden Fußnoten führt ein Argument mit einem Wert, der sich von Null bei doppelter Präzision nicht unterscheiden läßt, ebenfalls zu einem undefinierten Ergebnis.

2.2.5 Zeichenreihen

Signatur

Sorte	:	`string`
Konstantensymbole	:	Alle Sequenzen über 8-Bit-Zeichen von `"` begrenzt
Funktionssymbole	:	`size` : `string` $\rightarrow$ `int`
		`^` : `string`, `string` $\rightarrow$ `string`
		`chr` : `int` $\rightarrow$ `string`
		`ord` : `string` $\rightarrow$ `int`
		`substring` : `string`, `int`, `int` $\rightarrow$ `string`
		`=` : `string`, `string` $\rightarrow$ `bool`
		`<>` : `string`, `string` $\rightarrow$ `bool`
		`>` : `string`, `string` $\rightarrow$ `bool`
		`>=` : `string`, `string` $\rightarrow$ `bool`
		`<` : `string`, `string` $\rightarrow$ `bool`
		`<=` : `string`, `string` $\rightarrow$ `bool`

Algebra

`string`	:	$\texttt{string}^A = \{\texttt{"}a\texttt{"} \mid a \in \tilde{A}^*\}$, wobei $\tilde{A}$ die Menge aller 8-Bit-ASCII-Zeichen ist
`size(s)`	:	Anzahl der Zeichen in **s**
$\mathbf{s_0}$ `^` $\mathbf{s_1}$	:	Konkatenation von $\mathbf{s_0}$ und $\mathbf{s_1}$
`chr(i)`	:	String aus einem Zeichen, das den ASCII-Wert **i** hat
`ord(s)`	:	ASCII-Wert des ersten Zeichens von **s**
`substring(s,i,n)`	:	Teilwort von **s**, das mit dem **i**-ten Zeichen beginnt und **n** Zeichen umfaßt
$\mathbf{s_0}$ `=` $\mathbf{s_1}$	:	Gleichheit von $\mathbf{s_0}$ und $\mathbf{s_1}$
$\mathbf{s_0}$ `<>` $\mathbf{s_1}$	:	Ungleichheit von $\mathbf{s_0}$ und $\mathbf{s_1}$
$\mathbf{s_0}$ `>` $\mathbf{s_1}$	:	Test, ob $\mathbf{s_0} > \mathbf{s_1}$[13]
$\mathbf{s_0}$ `>=` $\mathbf{s_1}$	:	Test, ob $\mathbf{s_0} \geq \mathbf{s_1}$
$\mathbf{s_0}$ `<` $\mathbf{s_1}$	:	Test, ob $\mathbf{s_0} < \mathbf{s_1}$
$\mathbf{s_0}$ `<=` $\mathbf{s_1}$	:	Test, ob $\mathbf{s_0} \leq \mathbf{s_1}$

Die Operationen `=`, `<>`, `>`, `>=`, `<` und `<=` sind total. Die Tatsache, daß in ML keine Längebegrenzung für `string`-Werte vorgesehen ist, bedingt, daß auch die Operationen `size` und `^` total sind. In KIEL gibt es hingegen eine Maximallänge für `string`-Werte. Sie kann jedoch vom Benutzer selbst eingestellt werden (siehe 3.4.1). Dadurch ist die Operation `^` in KIEL nicht total. Die Operation `size` ist genau dann total, wenn die Maximallänge für `string`-Werte kleiner oder gleich der größten darstellbaren ganzen Zahl, also $2^{30} - 1$, ist. Auch bei den beiden Operationen `chr` und `ord`, die das Zeichen zu einem ASCII-Wert bzw. den ASCII-Wert zu einem Zeichen liefern, kann es zu Undefiniertheiten kommen. Da nur Werte von Null bis 255 gültige ASCII-Werte sind, ist das Ergebnis von `chr` undefiniert, falls das Argument kleiner als Null oder größer als 255 ist. Bei der Anwendung der Operation `ord` wird das erste Zeichen des Arguments in einen ASCII-Wert konvertiert. Gibt es kein erstes Zeichen im Argument, ist das Argument also die leere Zeichenreihe, so hat das Ergebnis keinen definierten Wert. Schließlich hat der Aufruf `substring(s,i,n)` als Ergebnis $\perp$, falls $\mathbf{n} < 0$, $\mathbf{i} < 0$ oder $\mathbf{i} + \mathbf{n} >$ `size(s)`.

[13] Die Vergleichsoperatoren beziehen sich beim Typ `string` auf die lexikografische Ordnung.

2.2.6 Sequenzen

Sei m ein ML-Typ aus 2.2.1 bis 2.2.5. [14]

Signatur

Sorte	:	m `list`
Konstantensymbole	:	`nil` : m `list`
Funktionssymbole	:	`::` : m, m `list` $\rightarrow m$ `list`
		`hd` : m `list` $\rightarrow m$
		`tl` : m `list` $\rightarrow m$ `list`
		`null` : m `list` $\rightarrow$ `bool`
		`length` : m `list` $\rightarrow$ `int`
		`@` : m `list`, m `list` $\rightarrow m$ `list`
		`rev` : m `list` $\rightarrow m$ `list`

Algebra

m `list`	:	Die Menge der Sequenzen von Elementen aus m^A — $(m\ \texttt{list})^A = (m^A)^*$
a `::` **l**	:	Erzeugung der Liste[15] mit erstem Element **a** und Restliste[16] **l**
`hd(`**l**`)`	:	Erstes Element von **l**
`tl(`**l**`)`	:	Restliste von **l**
`null(`**l**`)`	:	Test, ob die Liste **l** leer ist
`length(`**l**`)`	:	Anzahl der Elemente in **l**
$\mathbf{l_0}$ `@` $\mathbf{l_1}$	:	Konkatenation von $\mathbf{l_0}$ und $\mathbf{l_1}$
`rev(`**l**`)`	:	Revertierte Liste von **l**

Zum Aufbau einer Liste werden häufig die Konstante `nil` und die Operation `::` verwendet. Zur Vereinfachung der Eingabe gibt es in ML eine abkürzende Schreibweise. Anstelle von a_k`::`a_{k-1}`::`...`::`a_1`::`a_0`::nil` für ein $k \in \mathbb{N}$ und $a_0, \ldots, a_k \in m^A$, wobei m ein Basistyp ist, kann abkürzend `[`a_k`,`a_{k-1}`,`...`,`a_1`,`a_0`]` geschrieben werden.

Die Operationen `null`, `length` und `rev` sind total. Wie bei den Zeichenreichen (siehe 2.2.5) gibt es auch für die Sequenzen in ML keine Maximallänge.[17] Die Operationen `::` und `@` sind somit in ML total. In KIEL gibt es eine Maximallänge für Sequenzen. Sie kann jedoch vom Benutzer selbst eingestellt werden (siehe 3.4.1). Dadurch sind die Operationen `::` und `@` in KIEL nicht total. Die Operationen `hd` und `tl` haben nur dann ein definiertes Ergebnis, wenn ihr Argument eine nichtleere Liste ist.

In den Unterabschnitten 2.2.1 bis 2.2.6 wurden diejenigen Datenstrukturen beschrieben, die in KIEL verwendet werden können. Um mit diesen Datenstrukturen tiefgründiger arbeiten zu können, bedarf es Kontrollstrukturen, wie Rechenvorschriften und Fallunterscheidungen. Auf diese wird im folgenden in 2.2.7 eingegangen.

2.2.7 Rechenvorschriften

Rekursive Rechenvorschriften sind das zentrale Konzept funktionaler Programmiersprachen. Mit ihnen ist es möglich, eigene Operationen festzulegen. Diese Operationen eröffnen funktionalen Programmiersprachen,

[14] Die folgenden Festlegungen gelten in ML für beliebige Typen. Da in KIEL jedoch nur Sequenzen über den Basistypen implementiert sind, wird auch nur von diesen ausgegangen.

[15] Die Begriffe Liste und Sequenz werden im Hinblick auf ML zur Beschreibung desselben Konstruktes verwendet. Wird der Begriff Liste anstelle des Begriffs Sequenz benutzt, soll dem Aspekt der Implementierung mehr Gewicht verliehen werden.

[16] Die Restliste einer nichtleeren Liste l ist die Liste l ohne das erste Element.

[17] Dies ist auch dadurch sehr anschaulich, da Zeichenreihen semantisch nichts anderes sind als Sequenzen von 8-Bit-ASCII-Zeichen.

insbesondere durch die Möglichkeit der Rekursion, eine komplexe Ausdruckskraft. Die hieraus resultierende Semantik geht über die in 2.1 beschriebene Wertermittlung weit hinaus. Dieses Thema wird in 2.3 behandelt. Zunächst werden die Bestandteile einer einzelnen ML-Rechenvorschrift definiert.

Definition 2.7 (ML-Rechenvorschrift) *Eine* **ML-Rechenvorschrift** *besteht aus*

a) einem **Namen***,*

b) einer **Funktionalität***, die entsprechend der Definition 2.1 durch die Sorten der (formalen) Parameter und durch die Ergebnissorte bestimmt ist, und*

c) einem **Rumpf***, der ein Term ist und der das Ergebnis der Rechenvorschrift in Abhängigkeit von den Parametern bestimmt.* □

Syntaktisch wird eine ML-Rechenvorschrift mit Namen F, Parametern x_i der jeweiligen ML-Sorten m_i (für $0 \leq i \leq k$), Ergebnissorte n und Rumpf t wie folgt **deklariert**:

$$\texttt{fun}\ F\texttt{(}x_0\texttt{:}m_0\texttt{,}\ldots\texttt{,}x_k\texttt{:}m_k\texttt{):}n\ \texttt{=}\ t\texttt{;}$$

Das Semikolon ist das Abschlußzeichen der Deklaration. Als Kontextbedingung wird des weiteren gefordert, daß alle Parameter paarweise verschieden sind, sich von dem Rechenvorschriftsnamen unterscheiden und weder Bezeichnungen der ML-Signatur sind, noch mit im Vorfeld deklarierten Rechenvorschriftsnamen übereinstimmen. Beim Aufbau des Rumpfes sind die folgenden Konstrukte erlaubt:

a) Die **Konstanten** der Datenstrukturen aus 2.2.1 bis 2.2.6

b) Die **Parameter** der Rechenvorschrift

c) Anwendung von **Operationen** der Datenstrukturen aus 2.2.1 bis 2.2.6

d) Anwendung von zuvor deklarierten **Rechenvorschriften**

e) **Fallunterscheidungen** der Form

$$\texttt{if}\ b\ \texttt{then}\ e_0\ \texttt{else}\ e_1,$$

wobei die **Bedingung** b ein Term der Sorte `bool` sein muß und sowohl der **`then`-Zweig** e_0 als auch der **`else`-Zweig** e_1 Terme der gleichen Sorte sein müssen

f) **Rekursive Aufrufe** der Rechenvorschrift

Bei der Anwendung von Operationen und Rechenvorschriften muß die Anzahl der Argumente sowie deren Sorte der jeweiligen Signatur bzw. Deklaration entsprechen. Die Argumente sind ihrerseits wieder Terme nach obigem Aufbauprinzip. Die Sorte einer Fallunterscheidung ist gleich der gemeinsamen Sorte des `then`- und des `else`-Zweiges. Schließlich ist vorgeschrieben, daß der Rumpf ein Term der festgelegten Ergebnissorte ist.

Ein Beispiel für die Deklaration einer Rechenvorschrift ist:

```
fun FIB(x:int):int=
    if (x=0) orelse (x=1)
        then 1
        else FIB(x-1)+FIB(x-2);
```

Hierdurch wird eine Rechenvorschrift `FIB` deklariert, mit welcher der Wert der Fibonacci-Funktion für eine beliebige natürliche Zahl berechnet werden kann. Dieses Beispiel wird in Abschnitt 4.1 ausführlich behandelt.

In ML wird neben der einfachen Rekursion auch die **verschränkte Rekursion** unterstützt. Damit ist es möglich, nicht nur auf die aktuelle Rechenvorschift und vorher deklarierte Rechenvorschriften zuzugreifen, sondern auch auf einige folgende. Soll zum Beispiel eine Rechenvorschrift F aus einer Rechenvorschrift G aufgerufen werden, die ihrerseits aus F aufgerufen wird, so ist dies mit der obigen Festlegung nicht möglich. Abhilfe schaffen **Systeme von Rechenvorschriften**, in denen jede Rechenvorschrift im Rumpf zusätzlich zu den oben aufgeführten Konstrukten auch beliebige Rechenvorschriften des Systems verwenden kann.

Deklariert wird ein System von Rechenvorschriften mit den Namen F_i (für $0 \leq i \leq k \in \mathbb{N}$), den Parametern x^i_j der jeweiligen ML-Sorten m^i_j (für $0 \leq j \leq l_i \in \mathbb{N}$), den Ergebnissorten n_i und den Rümpfen t_i wie folgt:

$$
\begin{array}{llllllll}
\texttt{fun}\ F_0 & (\ x^0_0 & :m^0_0 & ,\ldots, & x^0_{l_0} & :m^0_{l_0} &):\ n_0 & =\ t_0 \\
\texttt{and}\ F_1 & (\ x^1_0 & :m^1_0 & ,\ldots, & x^1_{l_1} & :m^1_{l_1} &):\ n_1 & =\ t_1 \\
\vdots & & & & & & & \\
\texttt{and}\ F_{k-1} & (\ x^{k-1}_0 & :m^{k-1}_0 & ,\ldots, & x^{k-1}_{l_{k-1}} & :m^{k-1}_{l_{k-1}} &):\ n_{k-1} & =\ t_{k-1} \\
\texttt{and}\ F_k & (\ x^k_0 & :m^k_0 & ,\ldots, & x^k_{l_k} & :m^k_{l_k} &):\ n_k & =\ t_k \quad ;
\end{array}
$$

Ein typisches Beispiel für verschränkte Rekursion und somit für die Deklaration eines Rechenvorschriftensystems mit mehr als einer Rechenvorschrift wird durch die folgende Möglichkeit geliefert, den Test, ob eine natürliche Zahl gerade oder ungerade ist, auszuführen. Hierzu wird folgendes Rechenvorschriftensystem deklariert:

```
fun iseven(n:int):int=
    if n=0 then true
           else isodd(n-1)
and isodd(n:int):int=
    if n=0 then false
           else iseven(n-1);
```

Die verschränkte Rekursion ist deutlich daran zu erkennen, daß in der Rechenvorschrift `iseven` die Rechenvorschrift `isodd` aufgerufen wird und ebenso in der Rechenvorschrift `isodd` auf die Rechenvorschrift `iseven` zugegriffen wird.

2.3 Termersetzungssemantik in funktionalen Programmiersprachen

Die Terme, die in diesem Abschnitt behandelt werden, sind im Vergleich zu denen aus 2.1 um Rechenvorschriftsaufrufe und Fallunterscheidungen erweitert. Sie stellen die Ausdrücke dar, mit denen in KIEL und damit auch in ML gearbeitet werden kann. Zur Bestimmung des Wertes solcher Terme wird in 2.3.1 das Konzept der Termersetzung eingeführt. Auswertungsschritte[18] sind Auswertungen von Basisoperationen, Expansionen von Rechenvorschriftsaufrufen und Vereinfachungen von Fallunterscheidungen (siehe 2.3.2). Die Reihenfolge, in der Auswertungsschritte vorgenommen werden, ist nicht eindeutig bestimmt. Durch verschiedene Strategien wird das Ergebnis einer Ersetzung unter Umständen maßgeblich beeinflußt. In den letzten Teilen dieses Abschnittes werden verschiedene Auswertungsstrategien beschrieben. In 2.3.3 wird die Auswertung eines Ausdrucks durch wiederholtes Anwenden von Vereinfachung und Expansion behandelt. Dabei wird auf die Strategien eingegangen, in denen die Expansionen vorgenommen werden. Thema von

[18] In funktionalen Programmiersprachen werden Ausdrücke ausgewertet. Dazu werden sie als Terme in einem entsprechenden Termersetzungssystem betrachtet. Aus diesem Grund bezeichnen die Begriffe Auswertungsschritt und Ersetzungsschritt dieselbe Aktion.

2.3.4 ist die Auswertungsstrategie, nach der in ML Ausdrücke ausgewertet werden. Schließlich widmet sich 2.3.5 den Auswertungsstrategien, die von KIEL unterstützt werden.

Eine komplexere Bearbeitung der in 2.3.1 bis 2.3.3 behandelten Thematik findet sich in [Manna74] und [LoeSie84].

2.3.1 Termersetzung

Ziel dieses Unterabschnittes ist es, zu einer Berechnungsrelation zu gelangen, durch die der Wert eines Ausdrucks ermittelt werden kann. Dazu wird zunächst der Begriff des Termersetzungssystems benötigt.

Definition 2.8 (Termersetzungssystem) *Ein* **Termersetzungssystem** *ist ein Tripel* $(\Sigma, X, \mathcal{R})$ *mit*

a) einer Signatur $\Sigma = (S, K, F)$,

b) einer Variablenmenge X*, wobei die Sorten der Variablen aus* S *sind, und*

c) einer endlichen Menge $\mathcal{R}$ *von Regeln* $l \to r$*, wobei* $l, r \in \mathcal{T}_\Sigma(X)$ *so, daß* l *und* r *die gleiche Sorte haben und jede Variable von* r *auch in* l *vorkommt.* □

Durch die Signatur wird in einem Termersetzungssystem festgelegt, nach welchen Kriterien Terme bzw. Ausdrücke gebildet sein müssen, um mit dem Termersetzungssystem bearbeitet werden zu können.

In der Regelmenge eines Termersetzungssystems sind Regeln angegeben, anhand welcher Ersetzungen in dem zu bearbeitenden Ausdruck vorgenommen werden können. In diesen Regeln können Variable der Variablenmenge vorkommen. Die Variablen stehen für gewisse geschlossene Terme. Um mit ihnen arbeiten zu können, benötigt man Substitutionen.

Definition 2.9 (Substitutionen)

a) Zu einem Termersetzungssystem $(\Sigma, X, \mathcal{R})$ *heißt eine Funktion* $\sigma : X \to \mathcal{T}_\Sigma(X)$ *eine* **Variablensubstitution***, falls für alle* $x \in X$ *die Sorte von* $\sigma(x)$ *gleich der Sorte von* x *ist.*

b) Eine Variablensubstitution σ *heißt* **Grundsubstitution***, falls* $\sigma(x) \in \mathcal{T}_\Sigma$ *für alle* $x \in X$. □

Variablensubstitutionen können auf Terme angewendet werden. Dies geschieht bei der Termersetzung beim Prüfen, ob eine Regel angewendet werden kann, und beim Anwenden von Regeln.

Definition 2.10 (Anwendung einer Variablensubstitution) *Sei* $\Sigma = (S, K, F)$ *Signatur und* X *Variablenmenge. Gegeben sei weiter ein Term* $t \in \mathcal{T}_\Sigma(X)$ *und eine Variablensubstitution* $\sigma : X \to \mathcal{T}_\Sigma(X)$. $t\sigma$ *bezeichnet die* **Anwendung der Variablensubstitution** σ **auf den Term t** *und ist wie folgt definiert:*

a) Für alle $c \in K$ *gilt* $c\sigma = c$.

b) Für alle $x \in X$ *gilt* $x\sigma = \sigma(x)$.

c) Für alle $t = f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(X)$ *gilt* $t\sigma = f(t_1\sigma, \ldots, t_k\sigma)$. □

Um die Position festzulegen, an der eine Regel während einer Termersetzung angewendet werden kann, sind einige weitere Definitionen erforderlich. Zunächst werden die Stellen eines Terms definiert.

Definition 2.11 (Stellen eines Terms) *Sei* $\Sigma = (S, K, F)$ *Signatur und* X *Variablenmenge. Die Funktion* $Occ : \mathcal{T}_\Sigma(X) \to \wp(\mathbb{N}^*)$ *ist definiert durch* *a) Für alle $c \in K$ gilt $Occ(c) = \{\varepsilon\}$.*

b) Für alle $x \in X$ gilt $Occ(x) = \{\varepsilon\}$.

c) Für alle $t = f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(X)$ gilt $Occ(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{k} \{i\} \times Occ(t_i)$.

Sei $t \in \mathcal{T}_\Sigma(X)$. Dann bezeichnet $Occ(t)$ die **Stellen des Terms** *t.* □

Jeder Term kann als ein Auswertungsbaum dargestellt werden. Näheres hierzu steht in Definition 2.22. Betrachtet man denjenigen Teilbaum, der durch einen bestimmten Knoten aufgespannt wird, erhält man einen Teilterm. Jede Stelle eines Terms bestimmt eindeutig einen Knoten des zugehörigen Baumes. Demnach wird durch jede Stelle eine Terms ein induzierter Teilbaum und somit eindeutig ein Teilterm festgelegt.

Definition 2.12 (Teilterm) *Sei $\Sigma = (S, K, F)$ Signatur, X Variablenmenge, $t \in \mathcal{T}_\Sigma(X)$ und $z \in Occ(t)$. Dann bezeichnet t/z den* **Teilterm von t an der Stelle z** *und es gilt:*

a) Für alle $\hat{t} \in \mathcal{T}_\Sigma(X)$ ist $\hat{t}/\varepsilon = \hat{t}$.

b) Für alle $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(X)$ und alle Stellen $i_1 \ldots i_j$ ist $f(t_1, \ldots, t_k)/i_1 \ldots i_j = t_{i_1}/i_2 \ldots i_j$. □

Wie es der Name schon andeutet, ist es Ziel der Termersetzung, Terme durch andere zu ersetzen. Genaugenommen wird ein bestimmter Teilterm durch einen neuen Term ersetzt.

Definition 2.13 (Ersetzung) *Sei $\Sigma = (S, K, F)$ Signatur, X Variablenmenge, $t \in \mathcal{T}_\Sigma(X)$ und $z \in Occ(t)$. Weiter sei $t_1 \in \mathcal{T}_\Sigma(X)$ so, daß t/z und t_1 die gleiche Sorte haben. Dann bezeichnet $t[t_1/z]$ die* **Ersetzung von t/z in t durch $\mathbf{t_1}$** *und es gilt:*

a) Für alle $\hat{t} \in \mathcal{T}_\Sigma(X)$ ist $\hat{t}[t/\varepsilon] = t$.

b) Für alle $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(X)$ und alle Stellen $i_1 \ldots i_j$ ist

$$f(t_1, \ldots, t_k)[t/i_1 \ldots i_j] = f(t_1, \ldots, t_{i_1 - 1}, t_{i_1}[t/i_2 \ldots i_j], t_{i_1+1}, \ldots, t_k).$$ □

Die Regelmenge eines Termersetzungssystems bestimmt die Art der Ersetzungen, die in diesem Termersetzungssystem ausgeführt werden können. Aus diesem Grund ist es notwendig zu definieren, wann eine Regel auf einen Term anwendbar ist. Des weiteren wird definiert, daß das geordnete Paar zweier Terme genau dann in der Einschritt-Ersetzungsrelation liegt, wenn eine Regel auf den ersten Term anwendbar ist und diese Anwendung den zweiten Term liefert.

Definition 2.14 (Berechnungsrelation) *Sei $(\Sigma, X, \mathcal{R})$ ein Termersetzungssystem. Dann gilt:*

a) Eine Regel $l \to r$ ist **anwendbar** *auf $t \in \mathcal{T}_\Sigma$, falls es $z \in Occ(t)$ und $\sigma : X \to \mathcal{T}_\Sigma$ so gibt, daß $l\sigma = t/z$.*

b) Die **Einschritt-Ersetzungsrelation** *$\Longrightarrow \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ ist definiert durch:*

$$\begin{array}{ll} t_1 \Longrightarrow t_2 & \text{genau dann, wenn} \\ & \exists\ l \to r \in \mathcal{R},\ z \in Occ(t),\ \sigma : X \to \mathcal{T}_\Sigma : l\sigma = t_1/z \wedge t_2 = t_1[r\sigma/z] \end{array}$$

c) Die **Berechnungsrelation** *$\stackrel{*}{\Longrightarrow} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ ist die reflexiv transitive Hülle von $\Longrightarrow$.* □

Die Berechnungsrelation beinhaltet alle die Paare von Termen, bei denen sich aus dem ersten Term in endlich vielen Ersetzungen der zweite ergibt. Einige der zweiten Komponenten nehmen eine Sonderstellung ein. In ihnen kann keine weitere Ersetzung vorgenommen werden. Sie werden als terminal bezeichnet. Dies führt zur Definition der Normalform.

Definition 2.15 (Normalform) *Sei $(\Sigma, X, \mathcal{R})$ ein Termersetzungssystem.*

a) Ein Term $t \in \mathcal{T}_\Sigma$ heißt **terminal**, *falls es keinen Term $t' \in \mathcal{T}_\Sigma$ so gibt, daß $t \Longrightarrow t'$.*

b) Sei $t \in \mathcal{T}_\Sigma$. Ein Term $t' \in \mathcal{T}_\Sigma$ ist **Normalform zu t bezüglich** $(\Sigma, X, \mathcal{R})$, *falls $t \overset{*}{\Longrightarrow} t'$ und t' terminal ist.* □

Alle Normalformen zu einem Term bezüglich eines Termersetzungssystems sind Ergebnisse einer Ersetzung, die dem Ursprungsterm einen gewissen Wert zuordnen.

Definition 2.16 (Ergebnis, Wert) *Sei $(\Sigma, X, \mathcal{R})$ ein Termersetzungssystem.*
Sei $t \in \mathcal{T}_\Sigma$. Sei t' Normalform zu t bezüglich $(\Sigma, X, \mathcal{R})$.

a) Dann ist t' ein **Ergebnis** *von t bezüglich der Termersetzung in $(\Sigma, X, \mathcal{R})$.*

b) Der **Wert** *von t bezüglich der Ersetzung in $(\Sigma, X, \mathcal{R})$, die zu dem Ergebnis t' führte, ist der Wert von t' nach Definition 2.6.* □

Eine Ersetzungsfolge muß nicht terminieren. Nicht terminierende Ersetzungen liefern keinen definierten Wert.

Definition 2.17 (Terminierung) *Sei $(\Sigma, X, \mathcal{R})$ ein Termersetzungssystem.*
$(\Sigma, X, \mathcal{R})$ heißt **terminierend**, *falls es keine unendlich langen Ersetzungsketten der Form*

$$t_0 \Longrightarrow t_1 \Longrightarrow t_2 \Longrightarrow \ldots$$

gibt, wobei $t_i \in \mathcal{T}_\Sigma$ für alle $i \in \mathbb{N}$. □

In der Regel führen in einem Termersetzungssystem nicht alle Ersetzungsfolgen zu dem gleichen Ergebnis. In Termersetzungssystemen, die die Eigenschaft der Konfluenz haben, ist dies jedoch stets der Fall.

Definition 2.18 (Konfluenz) *Sei $(\Sigma, X, \mathcal{R})$ ein Termersetzungssystem.*
$(\Sigma, X, \mathcal{R})$ heißt **konfluent**, *falls die folgende Bedingungs erfüllt ist:*

$$\forall t_0, t_1, t_2 \in \mathcal{T}_\Sigma : ((t_0 \overset{*}{\Longrightarrow} t_1) \wedge (t_0 \overset{*}{\Longrightarrow} t_2)) \rightarrow \exists t' \in \mathcal{T}_\Sigma : (t_1 \overset{*}{\Longrightarrow} t') \wedge (t_2 \overset{*}{\Longrightarrow} t')$$ □

2.3.2 Auswertungsschritt und Auswertungsbaum

In funktionalen Programmiersprachen geschieht die Auswertung eines Ausdrucks analog zu gewissen Ersetzungsfolgen in einem Termersetzungssystem in einer Folge von Auswertungsschritten. Um in den weiteren Abschnitten dieses Kapitels auf verschiedene Auswertungsarten eingehen zu können, werden in diesem Unterabschnitt zunächst die Begriffe des Auswertungsschrittes und des Auswertungsbaumes eingeführt. Sei in diesem Unterabschnitt $(\Sigma, X, \mathcal{R})$ das Termersetzungssystem von ML und $A = (S^A, K^A, F^A)$ die entsprechende Σ-Algebra. Alle in 2.2 beschriebenen Basisoperationen von ML können zu einem Auswertungsschritt führen. Sind genügend Informationen über die Werte der Argumente vorhanden, kann eine solche Operation ausgewertet werden.

Sei $\odot$ eine ML-Operation aus 2.2.1 bis 2.2.6. $\odot$ kann ausgewertet werden, falls alle Argumente komplett ausgewertet sind. Dies entspricht einem Berechnungsschritt (siehe Definition 2.14) im ML-Termersetzungssystem. Wie in 2.2.2 beschrieben, stellen die Operationen `andalso` und `orelse` Ausnahmen dar. Bei ihnen kann eine Auswertung unter Umständen schon stattfinden, obwohl das zweite Argument noch nicht komplett ausgewertet ist.[19] Da mit Ausnahme der eben genannten Operationen alle Operationen von ML strikt sind, wird ihr Ergebnis undefiniert, sobald eines ihrer Argumente undefiniert ist. Das Ergebnis einer Operation ist ansonsten durch die in 2.2 angegebene Semantik bestimmt.

Im Vergleich zu 2.2 sind auch Rechenvorschriftsaufrufe und Fallunterscheidungen in ML erlaubt. Ein Rechenvorschriftsaufruf kann zu jedem beliebigen Zeitpunkt expandiert werden.

Definition 2.19 (Expandieren eines Rechenvorschriftsaufrufs) *Sei eine Rechenvorschrift F durch folgende Deklaration gegeben:*

$$\texttt{fun } F\texttt{(}x_0\texttt{:}m_0\texttt{,}\ldots\texttt{,}x_k\texttt{:}m_k\texttt{)}\texttt{:}n\texttt{=}t\texttt{;}$$

Sei t' ein Term, der einen Rechenvorschriftsaufruf

$$F\texttt{(}e_0\texttt{,}\ldots\texttt{,}e_k\texttt{)}$$

an der Stelle z enthält. Dann ist die **Expansion des Aufrufs von F an der Stelle z in t′** *der Term*

$$t'[t[e_0 \leftarrow x_0][e_1 \leftarrow x_1]\ldots[e_k \leftarrow x_k]/z],$$

wobei $a[e \leftarrow x]$ die simultane syntaktische Ersetzung aller Vorkommen der Variablen x in a durch e beschreibt. □

Um eine Fallunterscheidung zu vereinfachen, ist es notwendig, daß der boolesche Ausdruck, genauer das erste Argument der Fallunterscheidung, bereits ausgewertet ist.

Definition 2.20 (Vereinfachung einer Fallunterscheidung) *Gegeben sei der Term t durch*

$$t := \texttt{if } \beta \texttt{ then } e_0 \texttt{ else } e_1,$$

wobei $\beta \in \mathcal{T}_\Sigma^{\texttt{bool}}$ und $e_0, e_1 \in \mathcal{T}_\Sigma^m$ für eine beliebige Sorte m. Falls β komplett ausgewertet ist, kann t vereinfacht werden. Sei also β komplett ausgewertet. Nun ist der Wert von β undefiniert, oder β ist `true` *oder* `false`*. Die* **Vereinfachung der Fallunterscheidung t** *ist gegeben durch:*

$$\begin{cases} e_0 & ,falls \quad \beta = \texttt{true} \\ e_1 & ,falls \quad \beta = \texttt{false} \\ undefiniert & ,falls \quad \beta \ keinen\ definierten\ Wert\ hat \end{cases}$$ □

Zusammen mit der Auswertung einer ML-Operation bilden die in Definition 2.19 und Definition 2.20 festgelegten Arten von Auswertungen genau die Menge der möglichen Auswertungsschritte.

Definition 2.21 (Auswertungsschritt) *Ein* **Auswertungsschritt** *ist eine der folgenden drei Aktionen:*

- *Auswertung einer ML-Operation*
- *Expandieren eines Rechenvorschriftsaufrufs — siehe Definition 2.19*

[19] Die Möglichkeit der frühzeitigen Auswertung von `andalso` und `orelse` hängt von dem Wert des ersten Arguments ab. Darauf wurde in 2.2.2 eingegangen.

- *Vereinfachung einer Fallunterscheidung — siehe Definition 2.20*

Auf der Menge der Grundterme in ML sei die Relation $\leadsto$ definiert durch:

$$t_0 \leadsto t_1 \text{ genau dann, wenn } t_1 \text{ in einem Auswertungsschritt aus } t_0 \text{ entsteht.} \qquad \square$$

Grafisch kann ein ML-Ausdruck als Auswertungsbaum dargestellt werden.[20]

Definition 2.22 (Auswertungsbaum) *Sei t ein ML-Ausdruck. Er besteht somit aus*

- *Konstanten der ML-Datenstrukturen,*
- *ML-Operationsanwendungen,*
- *Anwendungen von Rechenvorschriften und*
- *Fallunterscheidungen.*

Die ML-Operationen und die Fallunterscheidungen haben mindestens ein Argument. Zu t existiert ein eindeutig bestimmter **Auswertungsbaum**. *Dies ist ein Baum, in dem die Kinder eines Knotens in einer festen Reihenfolge stehen. Die Knoten des Baumes sind die vier oben genannten Bestandteile von t. Die Kinder eines Knotens sind die Argumente des entsprechenden Bestandteils in gleicher Reihenfolge.*[21] □

2.3.3 Auswertung durch Vereinfachung und Expansion

Wie in 2.3.1 bemerkt, bedingt die Wahl der Reihenfolge der Ersetzungen in einem Termersetzungssystem unter Umständen das Ergebnis. Neben der Auswertung von elementaren Operationen sind bei der Ersetzung jetzt auch das Expandieren von Rechenvorschriftsaufrufen und das Vereinfachen von Fallunterscheidungen zu berücksichtigen.

In diesem Unterabschnitt wird ein Auswertungmechanismus, also eine Termersetzungssemantik, beschrieben, mit der in der Theorie häufig gearbeitet wird. Zum Beispiel wird in [Manna74] diese Termersetzungssemantik detailliert beschrieben. In der Praxis, also in funktionalen Programmiersprachen, wird die Auswertung durch Vereinfachung und Expansion im allgemeinen nicht verwendet. Die Unterabschnitte 2.3.4 und 2.3.5 gehen auf die Auswertungsmechanismen von ML bzw. KIEL ein.

Definition 2.23 (Auswertung durch Vereinfachung und Expansion) *Sei t ein ML-Ausdruck. Konstruiere eine Folge von ML-Ausdrücken $t_0, t_1, \ldots$ wie folgt:*

	$t_0 := t$; $i := 0$
(Vereinfachungsschritt)	*Bilde t_{i+1} durch Vereinfachung von t_i so weit wie möglich.*[22]
	Falls keine Rechenvorschriftsaufrufe in t_{i+1} vorhanden sind, Ende.
(Expansionsschritt)	*Bilde t_{i+2} durch simultane Expansion von Rechenvorschriftsaufrufen in t_{i+1}.*[23]
	$i := i + 2$; *gehe zu* (Vereinfachungsschritt)

[20] Als Auswertungsbaum werden alle Ausdrücke intern in KIEL gespeichert.

[21] In ML-Ausdrücken können Teile beliebig geklammert werden. Alle Klammern, die ohne Veränderung der Semantik in einem ML-Ausdruck weggelassen werden können, werden beim Auswertungsbaum nicht mit berücksichtigt. Alle anderen Klammern, wie zum Beispiel in `3*(4+1)`, tauchen im Auswertungsbaum ebenfalls nicht auf, da sie durch die Baumstruktur repräsentiert werden. Im Beispiel hätte die Wurzel des Auswertungsbaumes, ein `*`-Knoten, als erstes Kind einen Knoten mit dem Wert Drei und als zweites Kind einen `+`-Knoten, der selbst Kinder mit den Werten Vier und Eins hätte.

Ist die gebildete Folge **endlich**, *ist das letzte Folgenglied das Ergebnis der Berechnung; anderenfalls ist das Resultat undefiniert.* □

Die im Expansionsschritt zu expandierenden Rechenvorschriftsaufrufe werden von der gewählten Ersetzungsart bestimmt. Es gibt mehrere **Ersetzungsarten** (oder **Ersetzungsstrategien**), von denen im folgenden die drei gängigsten behandelt werden. Diese sind in der folgenden Aufzählung hervorgehoben:

- **Leftmost-Innermost-Ersetzung**
- Parallel-Innermost-Ersetzung
- **Leftmost-Outermost-Ersetzung**
- Parallel-Outermost-Ersetzung
- Free-Argument-Ersetzung
- **Full-Ersetzung**

Sei t ein ML-Ausdruck. Sei b der Auswertungsbaum zu t. In der Tabelle 2.1 ist die Menge der simultan bei einem Expansionsschritt zu expandierenden Knoten in Abhängigkeit der Ersetzungsstrategie angegeben.

Ersetzunsart	zu expandierende Rechenvorschriftsaufrufe
Leftmost-Innermost	$\{r \mid r$ ist bei einem Tiefensuche-Durchlauf in b der erste Rechenvorschriftsknoten, der keine Rechenvorschriftsknoten als Nachkommen hat $\}$
Parallel-Innermost	$\{r \mid r$ ist ein Rechenvorschriftsknoten in b, der keine Rechenvorschriftsknoten als Nachkommen hat $\}$
Leftmost-Outermost	$\{r \mid r$ ist bei einem Tiefensuche-Durchlauf in b der erste Rechenvorschriftsknoten $\}$
Parallel-Outermost	$\{r \mid r$ ist ein Rechenvorschriftsknoten in b, der keine Rechenvorschriftsknoten als Vorfahren hat $\}$
Free-Argument	$\{r \mid r$ ist ein Rechenvorschriftsknoten in b, der mindestens ein Argument besitzt, in dem keine Rechenvorschriftsknoten vorkommen $\}$
Full	$\{r \mid r$ ist Rechenvorschriftsknoten in $b\}$

Tabelle 2.1: Zu expandierende Rechenvorschriftsaufrufe in Abhängigkeit der Ersetzungsstrategie

Im folgenden wird auf die Parallel-Innermost-, die Parallel-Outermost- und die Free-Argument-Ersetzung nicht weiter eingegangen.

Ein Tiefensuche-Durchlauf besucht die Kinder eines Knotens von links nach rechts. Deshalb handelt es sich bei der Leftmost-Innermost- und Leftmost-Outermost-Ersetzung um Leftmost-Ersetzungen. Bei der Leftmost-Innermost-Ersetzung wird der erste Rechenvorschriftsknoten gewählt, der keine Rechenvorschriftsknoten als Nachkommen hat. Dieser ist somit ein innerster Knoten. Bei der Leftmost-Outermost-Ersetzung wird der erste Rechenvorschriftsknoten bei einem Tiefensuche-Durchlauf gewählt. Dieser hat also keine Rechenvorschriftsknoten als Vorfahren und ist somit ein äußerster Rechenvorschriftsknoten.

[22] Vereinfachung bedeutet hier Auswertung von ML-Basisoperationen und Fallunterscheidungen, ohne dabei Rechenvorschriftsaufrufe bearbeiten zu müssen.

[23] Im Expansionsschritt wird keine Aussage darüber getroffen, welche Rechenvorschriftsaufrufe zu expandieren sind. Für die Ermittlung des Ergebnisses ist dies jedoch ein sehr wichtiger Aspekt. Er wird im Anschluß an die Definition behandelt.

Der Wert eines ML-Ausdrucks t ist analog zu Definition 2.16 der Wert der Normalform zu t bezüglich des nun erweiterten Termersetzungssystems von ML. Die Wahl der Ersetzungsart bestimmt maßgeblich den Wert einer Termauswertung. Als Beispiel werden eine Rechenvorschrift sowie ein Term angegeben, dessen Auswertung nach Leftmost-Innermost-Ersetzung zu einem anderen Ergebnis führt als die Auswertung nach Leftmost-Outermost-Ersetzung.

Bei der Funktion handelt es sich um die Morris-Funktion, die als Funktion `F` gegeben sei durch:

```
fun F(x:int,y:int):int = if x<=0 then 1 else F(x-1,F(x-y,y));
```

Die Auswertung des Terms `F(1,0)` liefert mit Leftmost-Innermost-Ersetzung keinen definierten Wert, da die Auswertung nicht endlich ist:

`F(1,0)`	$\leadsto$	`F(1,0)`	(keine) Vereinfachung
	$\leadsto$	`if 1<=0 then 1 else F(1-1,F(1-0,0))`	Expansion von `F`
	$\leadsto$	`F(0,F(1,0))`	Auswertung von `<=`, Vereinfachung der Fallunterscheidung und zwei Auswertungen von `-`
	$\leadsto$	`F(0,if 1<=0 then 1 else F(1-1,F(1-0,0)))`	Expansion des inneren Rechenvorschriftsaufrufs
	$\leadsto$	`F(0,F(0,F(1,0)))`	Auswertung von `<=`, Vereinfachung der Fallunterscheidung und zwei Auswertungen von `-`
	$\leadsto^2$	`F(0,F(0,F(0,F(1,0))))`	
	$\leadsto^2$	`F(0,F(0,F(0,F(0,F(1,0)))))`	
	...		

Die Auswertung des Terms `F(1,0)` nach Leftmost-Outermost-Ersetzung ist:

`F(1,0)`	$\leadsto$	`F(1,0)`	(keine) Vereinfachung
	$\leadsto$	`if 1<=0 then 1 else F(1-1,F(1-0,0))`	Expansion von `F`
	$\leadsto$	`F(0,F(1,0))`	Vereinfachung des Ausdrucks
	$\leadsto$	`if 0<=0 then 1 else F(0-1,F(0-F(1,0),F(1,0)))`	Expansion des äußeren Rechenvorschriftsaufrufs
	$\leadsto$	`1`	Auswertung von `<=`, Vereinfachung der Fallunterscheidung

Die Ersetzung ist endlich, und das Ergebnis der Auswertung ist `1`.

2.3.4 Auswertung in ML

Die Termersetzung in ML erfolgt nach der Leftmost-Innermost-Strategie. Dies ist auch die Ersetzungsart, die bei vielen bekannten Programmiersprachen wie PASCAL oder C verwendet wird.

Im Gegensatz zur Auswertung durch Vereinfachung und Expansion ist die Auswertung in ML nicht in zwei Schritte unterteilt. Stattdessen wird zur Ermittlung des nächsten Auswertungsschrittes ein Tiefensuche-Durchlauf im Auswertungsbaum gestartet. Der erste Knoten eines solchen Durchlaufs, der bearbeitet werden kann, wird ausgewertet, vereinfacht bzw. expandiert. In der Regel kann ein Knoten bearbeitet werden, wenn er selbst nicht ausgewertet ist, alle seine Kinder hingegen ausgewertet sind. Neben den in 2.2.2 beschriebenen Operationen `andalso` und `orelse` bilden die Fallunterscheidungsknoten die einzige Ausnahme. Sie können bereits vereinfacht werden, wenn das erste Kind, also der boolesche Ausdruck, ausgewertet ist.

2.3.5 Auswertung in KIEL

In KIEL kann der Benutzer selbst bestimmen, nach welcher Ersetzungsstrategie er einen eingegebenen Ausdruck auswerten möchte. Dabei kann er einerseits eine der vorgegebenen Ersetzungsarten

- Leftmost-Innermost-Ersetzung,
- Leftmost-Outermost-Ersetzung oder
- Full-Ersetzung

auswählen (siehe 3.3.8), andererseits kann er auch die Reihenfolge der Termersetzung interaktiv beliebig steuern, so daß er jede beliebige Ersetzungsstrategie verfolgen kann.

Die Auswertung mit vorgegebenen Ersetzungsarten kann der Benutzer

- durch Vereinfachung und Expansion (siehe 2.3.3) vornehmen oder
- durch eine Folge von Auswertungsschritten, die durch Tiefensuche-Durchläufe gebildet wird.

Geschieht die Auswertung analog zur ML-Auswertung in einer Folge von Auswertungsschritten, ist noch zu klären, welche Kriterien ein Knoten im Tiefensuche-Durchlauf erfüllen muß, um bearbeitet werden zu können.

- Basisoperations- und Fallunterscheidungsknoten werden unabhängig von der Ersetzungsart, wie in 2.3.4 beschrieben, behandelt.
- Die Behandlung der Rechenvorschriftsknoten geschieht in Abhängigkeit der gewählten Ersetzungsart.
  - Wie in 2.3.4 beschrieben, wird bei Auswertung mit der Leftmost-Innermost-Ersetzung ein Rechenvorschriftsknoten, auf den bei einem Tiefensuche-Durchlauf gestoßen wird, dann expandiert, wenn alle seine Kinder ausgewertet sind.
  - Ist die Leftmost-Outermost-Ersetzung gewählt und erreicht ein Tiefensuche-Durchlauf einen Rechenvorschriftsknoten, dann wird dieser ohne Bedingungen für die folgende Expansion ausgewählt.
  - Trifft ein Tiefensuche-Durchlauf während einer Auswertung mit der Full-Ersetzung auf einen Rechenvorschriftsknoten, werden simultan alle Rechenvorschriftsknoten im Auswertungsbaum expandiert.

Eine Beschreibung der Vorgehensweise bei den verschiedenen Auswertungsmöglichkeiten, die KIEL bietet, findet sich in Kapitel 3, wo auf alle Komponenten des Systems eingegangen wird. Besonders der Abschnitt 3.3.6, in dem unter anderem die Funktionen des Baumfensters beschrieben werden, ist in diesem Zusammenhang von Bedeutung.

Kapitel 3

Komponenten des Systems und deren Bedienung

3.1 Gesamtübersicht

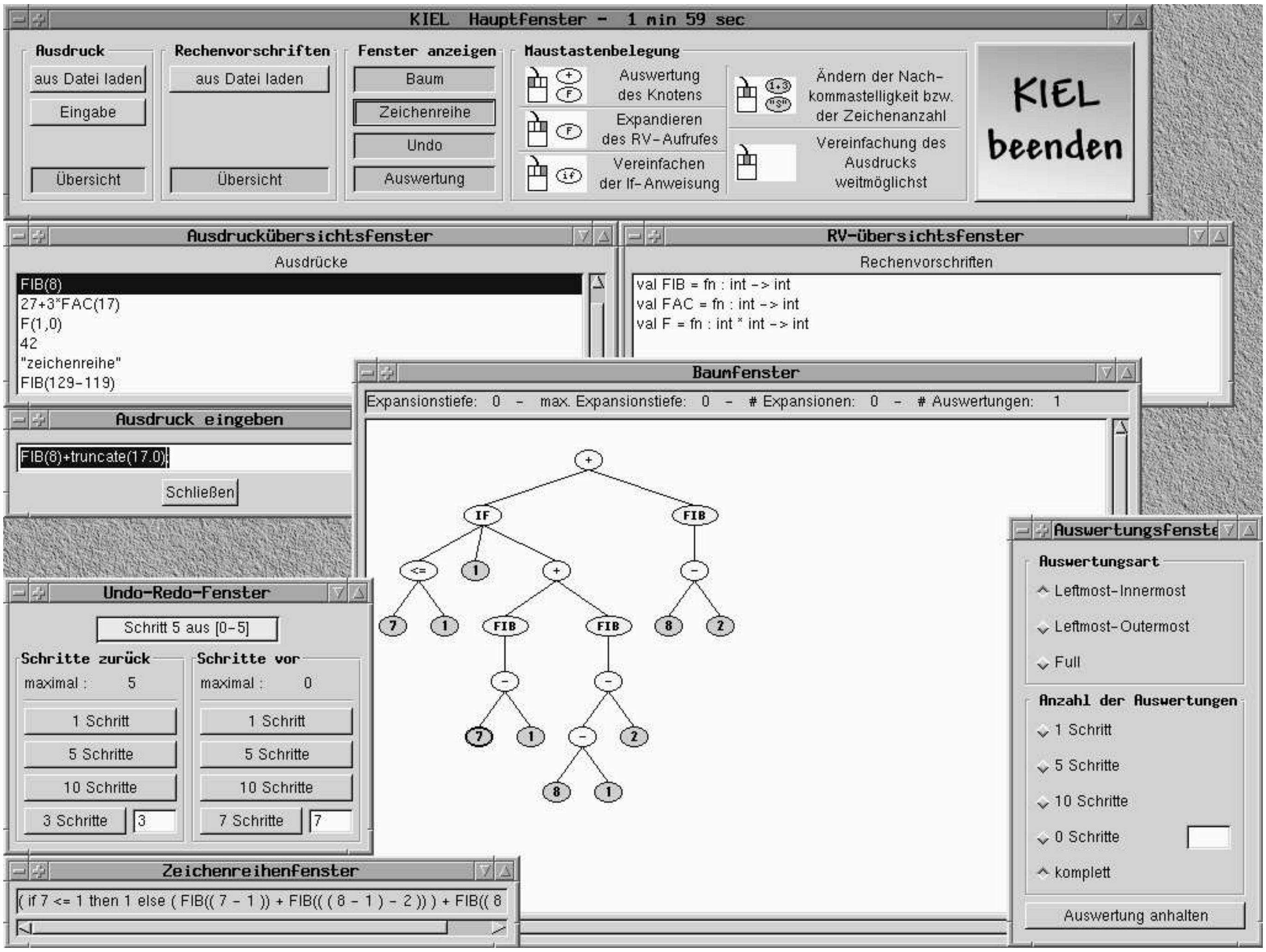


Abbildung 3.1: Gesamtübersicht über die Oberfläche von KIEL

In Abbildung 3.1 sind die Fenster von KIEL zu sehen. Es sind fast alle Fenster dargestellt, die während der Arbeit mit KIEL benötigt werden. Nur die Dateiauswahl- und mögliche Warnungsfenster sind nicht abgebildet.

3.2 Start des Programms

Es existieren zwei Versionen von KIEL. Die eine besitzt eine deutschsprachige, die andere eine englischsprachige Oberfläche. Die Funktionalität hinter den verschiedenen Darstellungen ist identisch. Da diese Arbeit in deutscher Sprache verfaßt ist, sind auch alle Abbildungen der deutschen Version entnommen.

Somit hat der Benutzer Wahl zwischen dem Aufruf der deutschen Version mit

`kiel_deutsch`

und der englischen Version mit

`kiel_englisch`

Als Erleichterung wird beim Kompilieren automatisch ein symbolischer Link erzeugt, so daß mit

`kiel`

die deutsche Version gestartet werden kann.

3.3 Die Benutzschnittstelle

Die Fenster stellen die Schnittstelle von KIEL zum Benutzer dar. Über sie werden Daten eingegeben, und sie zeigen Daten an. Eine kurze Übersicht über die Fenster wird in Tabelle 3.1 gegeben. In ihr sind die Fenster bezüglich ihrer Größenflexibilität geordnet und mit den Kapitelnummern, in denen sie beschrieben werden, angegeben.

Fenster mit fester Größe	Befehlsfenster	3.3.1
	Auswertungsfenster	3.3.8
	Undo-Redo-Fenster	3.3.9
Fenster mit Mindestgröße	Ausdruckeingabefenster	3.3.2
	Dateifenster	3.3.3
	Ausdruckübersichtsfenster	3.3.4
	Rechenvorschrifts-Übersichtsfenster	3.3.5
	Warnungsfenster	3.3.10
Beliebig in ihrer Größe veränderbare Fenster	Baumfenster	3.3.6
	Zeichenreihenfenster	3.3.7

Tabelle 3.1: Fenster bezüglich ihrer Größenflexibilität geordnet

Im folgenden wird jedes Fenster mit seinen Funktionalitäten beschrieben. Die Reihenfolge ist so gewählt, daß die Zusammenhänge unter den Fenstern besonders gut erkannt werden können. Nach dem Hauptfenster werden zunächst die Eingabefenster, dann die Übersichtsfenster und die Anzeigefenster beschrieben. Es folgen die Manipulationsfenster und schließlich die Warnungsfenster.

3.3.1 Befehlsfenster

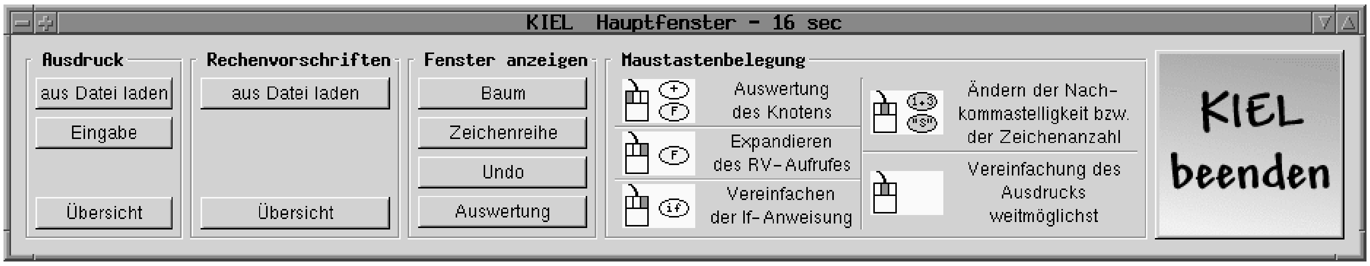


Abbildung 3.2: Das Befehlsfenster

Das Befehlsfenster ist in Abbildung 3.2 dargestellt. Es ist das Hauptfenster von KIEL und koordiniert die Anzeige anderer Fenster, gibt einen Überblick über die Funktionen, die im Baumfenster (siehe 3.3.6) mit Mausklicks ausgeführt werden können, und dient zum Beenden von KIEL.

3.3.1.1 Der Timer

In der Titelzeile des Befehlsfensters wird die seit dem Programmstart vergangene Zeit angezeigt. Diese Anzeige wird einmal pro Sekunde durch einen Timer aktualisiert.

3.3.1.2 Der Ausdruckrahmen

Der Ausdruckrahmen stellt drei Buttons zur Verfügung:

- `aus Datei laden`
- `Eingabe`
- `Übersicht`

Der Button `aus Datei laden` öffnet ein Dateifenster (siehe 3.3.3) zum Laden eines ML-Ausdrucks. Dieser Ausdruck sollte ein Element der Teilsprache sein, die in KIEL implementiert ist. Der Aufbau eines solchen Ausdrucks ist durch Syntaxdiagramme in A.2 beschrieben. Diese Art der Eingabe von Ausdrücken in das System ist sinnvoll, wenn es sich um längere Ausdrücke handelt, auf häufig benutzte Ausdrücke zugegriffen werden soll oder mehrere Ausdrücke auf einmal eingegeben werden sollen.

Beabsichtigt der Benutzer hingegen, einen kurzen Ausdruck, wie zum Beispiel `FIB(17)`, einzugeben, so wählt er der Einfachheit halber den Button `Eingabe`. Hiermit öffnet er das Ausdruckeingabefenster (siehe 3.3.2), in dem die Eingabe einzelner Ausdrücke möglich ist.

Der Button `Übersicht` ist ein Toggle-Button. Mit seiner Hilfe kann der Benutzer bestimmen, ob das Ausdruckübersichtsfenster (siehe 3.3.4) angezeigt wird.

3.3.1.3 Der Rechenvorschriftenrahmen

Der Rechenvorschriftenrahmen stellt zwei Buttons zur Verfügung:

- `aus Datei laden`
- `Übersicht`

Der Button `aus Datei laden` öffnet ein Dateifenster (siehe 3.3.3) zum Laden eines ML-Programms. Dieses Prgramm sollte ein Element der Teilsprache sein, die in KIEL implementiert ist. Der Aufbau eines solchen Programms ist durch Syntaxdiagramme in A.1 beschrieben.

Der Button `Übersicht` ist ein Toggle-Button. Mit seiner Hilfe kann der Benutzer bestimmen, ob das Rechenvorschrifts-Übersichtsfenster (siehe 3.3.5) angezeigt wird.

3.3.1.4 Der Fensterrahmen

In dem Fensterrahmen befinden sich vier Toggle-Buttons:

- `Baum`
- `Zeichenreihe`
- `Undo`
- `Auswertung`

Mit ihrer Hilfe kann der Benutzer bestimmen, ob das Baumfenster (siehe 3.3.6), das Zeichenreihenfenster (siehe 3.3.7), das Undo-Redo-Fenster (siehe 3.3.9) und das Auswertungsfenster (siehe 3.3.8) angezeigt werden.

3.3.1.5 Der Maustastenbelegungsrahmen

Der Maustastenbelegungsrahmen stellt dem Benutzer eine Übersicht über die Belegung der Maustasten im Baumfenster (siehe 3.3.6) zur Verfügung.

3.3.1.6 Der Quitbutton

Mit dem Drücken dieses Buttons beendet der Benutzer KIEL.

3.3.2 Ausdruckeingabefenster

Abbildung 3.3: Das Ausdruckeingabefenster

In dem Ausdruckeingabefenster (siehe Abbildung 3.3) kann der Benutzer einen Ausdruck eingeben. Es besteht aus einem Eingabefeld und einem Button. Ist dieses Fenster das aktive Bildschirmfenster, so wird jeder Tastendruck als Eingabe für das Eingabefeld interpretiert.

Das Ausdruckeingabefenster wird durch Drücken des Buttons `Eingabe` aus dem Ausdruckrahmen des Befehlsfensters (siehe 3.3.1) geöffnet. Es kann mit dem `Schließen`-Button geschlossen werden.

Das Drücken der Eingabetaste ist für KIEL das Zeichen zum Übernehmen des Ausdrucks. Ist ein korrekter Ausdruck eingegeben worden, so wird dieser im Ausdruckübersichtsfenster (siehe 3.3.4) angezeigt. Sollte das Ausdruckübersichtsfenster nicht geöffnet sein, so wird es automatisch geöffnet. Bei Eingabe eines inkorrekten Ausdrucks wird eine Warnung ausgegeben (siehe 3.3.10). In jedem Fall wird nach Betätigung der Eingabetaste der eingegebene Ausdruck im Eingabefeld markiert dargestellt, so wie es in der Abbildung 3.3 zu sehen ist. Dies bedeutet für den Benutzer, daß er einen weiteren Ausdruck auf zwei Arten eingeben kann:

- Der Benutzer kann den nächsten Ausdruck komplett neu eingegeben. Dazu beginnt er mit der Eingabe, wenn der vorige Ausdruck noch markiert ist. Dann wird mit dem ersten Druck einer Zeichentaste der alte Ausdruck automatisch gelöscht. Hierbei ist zu beachten, daß die Markierung aufgehoben wird, wenn der Benutzer mit der Maus in das Eingabefeld klickt. Hat diese Handlung den Zweck, das Ausdruckeingabefenster wieder zum aktiven Bildschirmfenster werden zu lassen, ist ein Anklicken der Titelleiste des Ausdruckeingabefensters vorzuziehen, da so die Markierung erhalten bleibt.
- Außerdem hat der Benutzer die Möglichkeit, den im Eingabefeld dargestellten Ausdruck als Vorlage für den nächsten zu nutzen. Hierzu muß er die Markierung des alten Ausdrucks aufheben. Dies kann er erreichen, indem er
  - entweder mit der Maus auf das Eingabefeld klickt
  - oder sich mit den Cursortasten im Eingabefeld bewegt.

  Nun kann der Benutzer einen neuen Ausdruck eingeben, indem er auf Teile des vorigen Ausdrucks zurückgreift. Verständlicherweise wird diese Methode meist dann angewendet, wenn der alte und der neue Ausdruck ähnlich sind.

Die Markierung des Ausdrucks oder von Teilen des Ausdrucks kann ebenso jederzeit in gewohnter Weise mit Hilfe der Maus vorgenommen werden. Das Drücken des `Schließen`-Buttons bewirkt neben dem Schließen des Ausdruckeingabefensters auch das Löschen des Eingabefeldes. Wird das Ausdruckeingabefenster also nach dem Schließen wieder geöffnet, so ist das Eingabefeld leer.

3.3.3 Dateifenster

Es gibt zwei Dateifenster in KIEL. Das eine dient zum Laden von Ausdrücken, das andere zum Laden von Rechenvorschriften. Sie werden durch Drücken der Buttons `aus Datei laden` aus dem Ausdruck- bzw. Rechenvorschriftenrahmen des Befehlsfensters (siehe 3.3.1) geöffnet. Exemplarisch ist in Abbildung 3.4 das Dateifenster zum Laden von Ausdrücken zu sehen.

Nach dem Öffnen eines Dateifensters werden alle Ausdruck- bzw. Rechenvorschriftendateien, die sich im Vorgabeverzeichnis befinden, in der rechten Liste angezeigt. Aus dieser kann der Benutzer durch Markieren seine Auswahl treffen. Die Vorgabeverzeichnisse werden durch Einträge in der Ressourcedatei `kiel.rc` (siehe 3.4.1) festgelegt. Das Wechseln in andere Verzeichnisse kann in der linken Liste oder in dem oben liegenden Popup-Button vollzogen werden. Außerdem kann der Name der zu ladenden Datei direkt per Tastatur in das unten liegende Eingabefeld geschrieben werden. Um eine Datei zu laden, bestätigt der Benutzer seine Auswahl durch einen Doppelklick auf den entsprechenden Dateinamen oder drückt den `OK`-Button. Das Laden kann durch Druck auf den `Cancel`-Button abgebrochen werden.

Nach dem Laden einer Ausdruck- bzw. Rechenvorschriftendatei prüft KIEL, ob ihr Inhalt konform zu der unterstützten Sprache (siehe A.2 bzw. A.1) ist. Nur wenn dies der Fall ist, werden die Daten ins System übernommen, anderenfalls wird ein Warnungsfenster (siehe 3.3.10) geöffnet und dort der erkannte Syntaxfehler angezeigt.

KIEL gibt ebenso eine Warnung aus, falls beim Überprüfen einer Ausdruck- oder Rechenvorschriftendatei ein `int`-, `real`-, `string`- oder `list`-Typ auftaucht, der die im System festgesetzten Bereichsgrenzen (siehe 5.3)

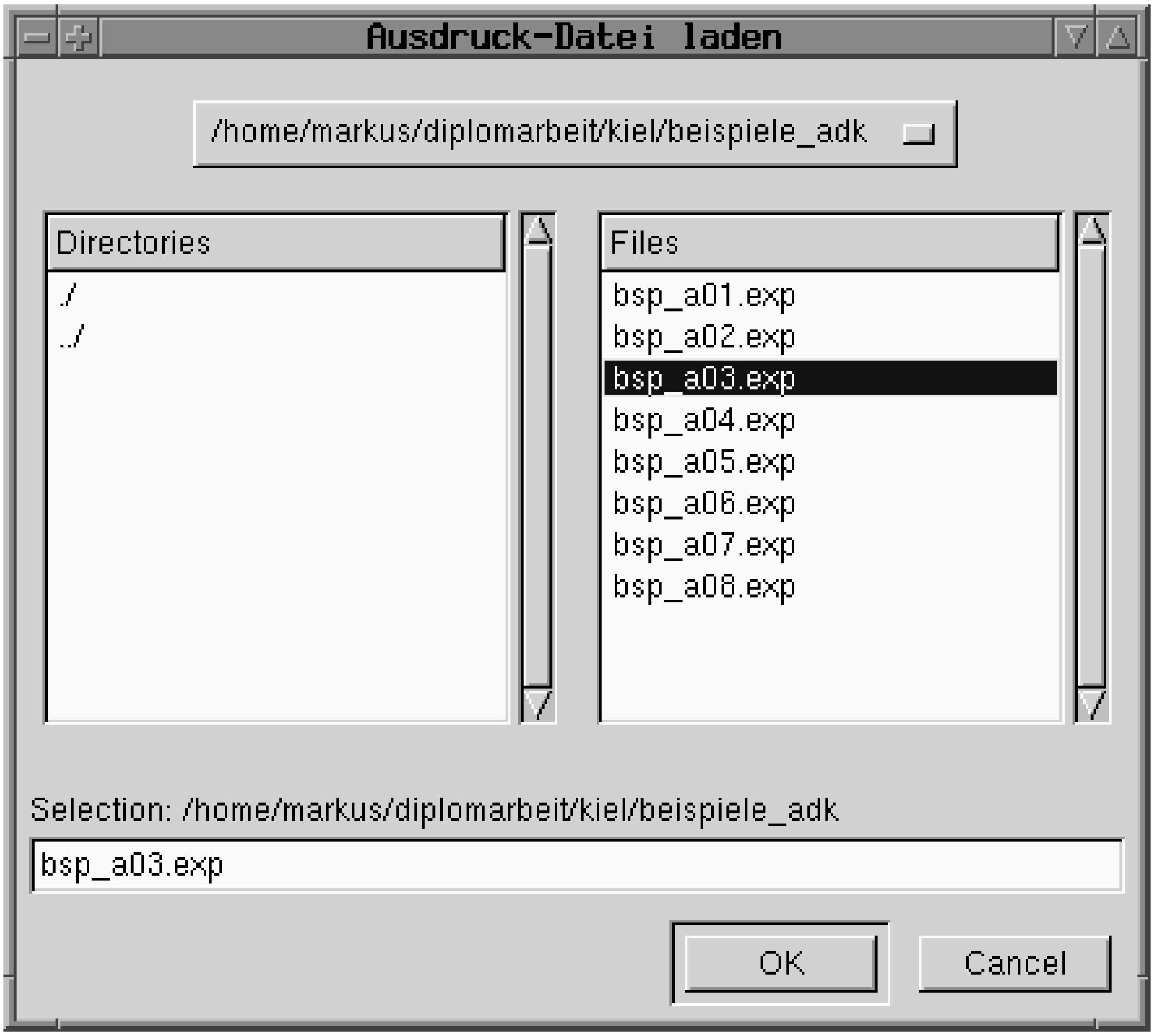


Abbildung 3.4: Das Dateifenster zum Laden von Ausdrücken

überschreitet. Entspricht der Inhalt der Datei der Syntax, werden auch in diesem Fall die Daten in das System übernommen. Die Knoten, die einen bereichsüberschreitenden Wert darstellen, werden zu Fehlerknoten (siehe 3.3.6).

3.3.3.1 Laden von Ausdruckdateien

Ausdruckdateien haben die Endung `exp` für Ausdruck (englisch: expression). Entspricht der Inhalt einer Ausdruckdatei der Syntax in KIEL, werden die Ausdrücke, die in dieser Datei stehen, der Reihe nach zu den vorher in KIEL vorhandenen Ausdrücken hinzugefügt. Das bedeutet, daß sie oben im Ausdruckübersichtsfenster (siehe 3.3.4) angezeigt werden.

3.3.3.2 Laden von Rechenvorschriftendateien

Rechenvorschriftendateien haben die Endung `prg` für Programm, da eine Folge von Rechenvorschriften als Programm bezeichnet wird. Entspricht der Inhalt einer Rechenvorschriftendatei der Syntax in KIEL, werden die bisherigen Rechenvorschriften gelöscht und durch die in dieser Datei stehenden ersetzt. Dies ist sinnvoll, da es in einem abgeschlossenen Programm keine Abhängigkeiten zu anderen Rechenvorschriften geben sollte. Des weiteren wird hierdurch verhindert, daß mehrere Rechenvorschriften gleichen Namens in KIEL zu demselben Zeitpunkt vorhanden sind. Die geladenen Rechenvorschriften werden in das Rechenvorschrifts-Übersichtsfenster (siehe 3.3.5) geschrieben. Außerdem werden alle vorhandenen Ausdrücke in KIEL gelöscht, da sich Ausdrücke im allgemeinen auf Rechenvorschriften eines speziellen Programms beziehen und es somit nach dem Laden eines neuen Programms zu ungewollten Bindungen zwischen Ausdrücken und Rechenvorschriften kommen könnte.

3.3.4 Ausdruckübersichtsfenster

Abbildung 3.5: Das Ausdruckübersichtsfenster

Das Ausdruckübersichtsfenster, abgebildet im Abbildung 3.5, besteht im wesentlichen aus einer (ggf. scrollbaren) Auswahlliste. In dieser sind alle Ausdrücke aufgelistet, die in KIEL zum jeweiligen Zeitpunkt eingegeben sind.

Die Auswahl eines Ausdrucks aus dieser Liste geschieht durch Klicken mit der linken Maustaste auf den gewünschten Eintrag. Nach der Auswahl eines Ausdrucks wird dieser markiert. Im Baum- und im Zeichenreihenfenster (siehe 3.3.6 und 3.3.7) wird immer der in der Auswahlliste markierte Ausdruck bearbeitet bzw. angezeigt.

Die Position der Ausdrücke in der Liste wird durch den Zeitpunkt der Übernahme bestimmt. Wird ein Ausdruck über das Ausdruckeingabefenster (siehe 3.3.2) eingegeben oder aus einer Datei geladen (siehe 3.3.3), so wird er, falls er von KIEL akzeptiert wird, in das System übernommen und an den Anfang der Liste gestellt. Er bleibt so lange in der Liste, bis das System beendet oder eine neue Rechenvorschriftendatei geladen wird (siehe 3.3.3). Im letzeren Fall werden alle vorhandenen Ausdrücke gelöscht.

Bei der Arbeit mit KIEL kommt es mitunter vor, daß ein gewisser Ausdruck, der bereits im Ausdruckübersichtsfenster aufgelistet ist, mit einen Ausdruck, der neu eingegeben werden soll, so gut übereinstimmt, daß er idealerweise als Vorlage dienen sollte. Solange der Ausdruck, der als Vorlage dienen soll, der zuletzt eingegebene ist, das Ausdruckeingabefenster zwischenzeitlich nicht geschlossen wurde und in ihm keine Änderungen vorgenommen wurden, steht der Ausdruck noch im Ausdruckeingabefenster und kann dort editiert werden. In der Regel sind diese Voraussetzungen jedoch nicht erfüllt. Die Aufgabe, einen Ausdruck, der im Ausdruckübersichtsfenster aufgelistet ist, in das Ausdruckeingabefenster zu kopieren, um ihn dort als Vorlage für weitere Ausdrücke zu benutzen, übernimmt die rechte Maustaste.

Um einen Ausdruck auf die eben beschriebene Art zu kopieren, muß der Benutzer diesen Ausdruck im Ausdruckübersichtsfenster auswählen. Ein Klick mit der rechten Maustaste auf einen beliebigen Eintrag im Ausdruckübersichtsfenster bewirkt, daß der markierte Ausdruck in das Ausdruckeingabefenster kopiert wird. Da für die Eingabe ein abschließendes Semikolon erforderlich ist, wird ein solches automatisch beim Kopieren an den Ausdruck gehängt. Falls das Ausdruckeingabefenster zu dem Zeitpunkt geschlossen ist, an dem die rechte Maustaste auf einem Eintrag im Ausdruckübersichtsfenster betätigt wird und dort ein Ausdruck markiert ist, wird das Ausdruckeingabefenster geöffnet.

Überschreitet ein Wert eines Ausdrucks in KIEL die Bereichsgrenzen seines Typs (siehe 5.3), wurde der Benutzer beim Laden oder bei der Eingabe gewarnt. Im Ausdruckübersichtsfenster wird der Wert trotz seiner Bereichsüberschreitung angezeigt; im Baumfenster (siehe 3.3.6) und im Zeichenreihenfenster (siehe 3.3.7) wird hingegen ein Fehler angezeigt.

3.3.5 Rechenvorschrifts-Übersichtsfenster

Abbildung 3.6: Das Rechenvorschrifts-Übersichtsfenster

Wie das Ausdruckübersichtsfenster (siehe 3.3.4) besteht auch das Rechenvorschrifts-Übersichtsfenster, dargestellt in Abbildung 3.6, im wesentlichen aus einer (ggf. scrollbaren) Auswahlliste. In dieser sind alle Rechenvorschriften aufgelistet, die in KIEL zum jeweiligen Zeitpunkt eingegeben sind.

In diesem Fenster stehen zu jeder im System befindlichen Rechenvorschrift Name und Funktionalität. Die Auswahl einer Rechenvorschrift aus der Auswahlliste zieht das Anzeigen der gesamten Rechenvorschrift auf der Standardeingabe nach sich. So kann sich der Benutzer die Rümpfe der Rechenvorschriften ansehen und damit deren Wirkungsweise erkennen.

Im Ausdruckübersichtsfenster kann mit Hilfe der rechten Maustaste ein Ausdruck in das Ausdruckeingabefenster kopiert werden. Mit dem gleichen, in 3.3.4 beschriebenen, Mechanismus können auch Daten aus dem Rechenvorschrifts-Übersichtsfenster in das Ausdruckeingabefenster übernommen werden.

Wird die rechte Maustaste über einem Eintrag im Rechenvorschrifts-Übersichtsfenster betätigt, wird an der Cursorposition im Ausdruckeingabefenster der Name der Rechenvorschrift eingefügt, die im Rechenvorschrifts-Übersichtsfenster markiert ist. Des weiteren werden eine öffnende und eine schließende Klammer hinter den Rechenvorschriftsnamen gesetzt und der Cursor zwischen diesen Klammern plaziert. Auf diese Art kann auf das Eintippen von Rechenvorschriftsnamen verzichtet werden. Außerdem wird die Gefahr, eine schließende Klammer bei der Eingabe zu vergessen, verringert.

3.3.6 Baumfenster

Das Baumfenster, in Abbildung 3.7 dargestellt, ist das Fenster, in welchem hauptsächlich die Interaktion zwischen dem Benutzer und KIEL stattfindet. Es besteht aus einem Statusbalken und einer Zeichenfläche. Ist im Ausdruckübersichtsfenster (siehe 3.3.4) ein Ausdruck markiert, so wird er auf der Zeichenfläche grafisch als Baum angezeigt. Es gibt vier verschiedene Arten von Knoten in diesem Baum:

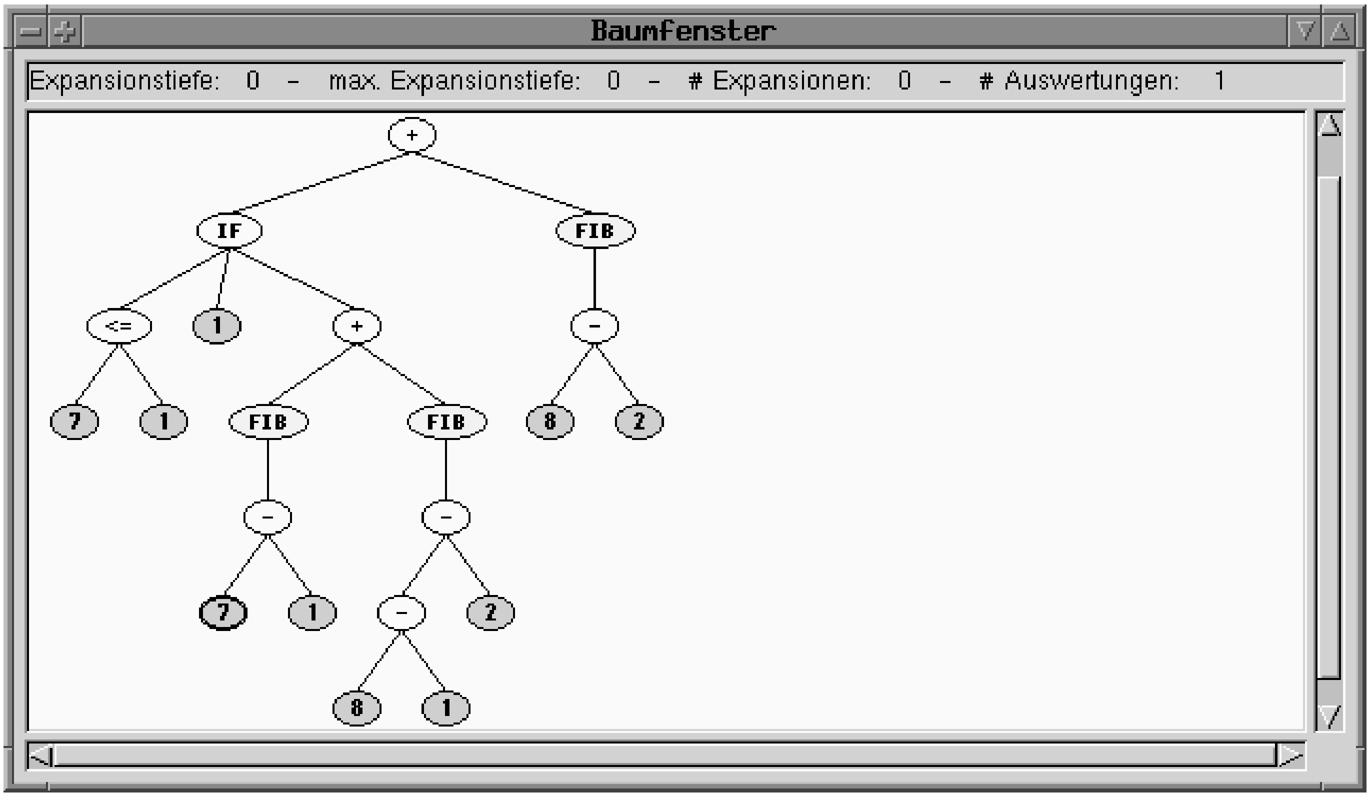
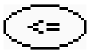
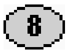


Abbildung 3.7: Das Baumfenster



Nicht ausgewertete Knoten

Mit Ausnahme von Rechenvorschriftsaufrufen werden alle Knoten, die nicht ausgewertet sind, auf weißem Hintergrund dargestellt.



Ausgewertete Knoten

Alle Knoten, die einen Wert haben, werden auf hellblauem Hintergrund angezeigt.

Rechenvorschriftsaufrufe

Wegen ihrer besonderen Bedeutung bei der Auswertung haben Rechenvorschriftsaufrufe eine eigene Darstellung. Ihr Hintergrund ist hellgelb.

Fehlerknoten

Überschreitet ein Wert im aktuellen Ausdruck eine Bereichsgrenze seines Typs (siehe 5.3), oder ist das Ergebnis einer Operation bei einer Auswertung nicht definiert, wird der entsprechende Knoten zum Fehlerknoten. Fehlerknoten weichen in ihrer Form von den anderen Knoten ab; sie sind rechteckig, und ihr Hintergrund ist rot. Dadurch wird ein deutliches Auffallen ermöglicht.

Der Benutzer hat die Möglichkeit, verschiedene Operationen auf diesem Baum ausführen zu lassen. Grundsätzlich wird eine Operation, die sich auf einen bestimmten Knoten bezieht, durch einen Mausklick auf den zu bearbeitenden Knoten bestimmt. Eine Operation, die sich auf mehrere Knoten bezieht, wird durch das Drücken einer Maustaste auf einen freien Platz der Zeichenfläche initiiert. Je nach Art des Knotens und abhängig von der gedrückten Maustaste werden unterschiedliche Operationen ausgeführt. Ein Überblick über die Operationen und die jeweilige Maustaste werden während der Ausführung von KIEL ständig im Maustastenbelegungsrahmen des Befehlsfensters (siehe 3.3.1) angezeigt. Dadurch kann sich der Benutzer, während er mit KIEL arbeitet, jederzeit über die Maustastenbelegung im klaren sein.

Es stehen insgesamt fünf Operationen zur Verfügung, die auf die oben beschriebene Weise auszuwählen sind:

- Auswerten eines Knotens

  Wird mit der **linken** Maustaste ein nicht ausgewerteter Knoten ausgewählt, so versucht KIEL, diesen auszuwerten. Die Auswertung erfolgt nach der im Rahmen `Auswertungsart` des Auswertungsfensters (siehe 3.3.8) festgelegten Termersetzungssemantik (siehe 2.3.3). Die Maximalanzahl der Auswertungsschritte, die für diesen Knoten ausgeführt werden, wird anhand der Auswahl im Rahmen `Anzahl der Auswertungen` des Auswertungsfensters (siehe 3.3.8) bestimmt.

  Während einer Auswertung werden folgende Daten über die Auswertung in dem Statusbalken angezeigt:

  - die aktuelle Expansionstiefe
  - die bisher maximale Expansionstiefe
  - die Anzahl der bisherigen Expansionen
  - die Anzahl der bisherigen Auswertungen von ML-Basisoperationen

  Anhand dieser Daten, die bei der Auswertung ständig aktualisiert werden, kann sich der Benutzer ein Bild vom Fortschritt der Auswertung machen. Dies ist insofern sehr nützlich, da (ggf. unendlich) lange Auswertungen vom Benutzer frühzeitig erkannt und abgebrochen werden können (siehe zum Abbruch einer Auswertung 3.3.8). KIEL führt so lange Auswertungsschritte durch, bis entweder

  - der ausgewählte Knoten komplett ausgewertet wurde,
  - die vorgegebene Anzahl an Auswertungsschritten durchgeführt wurde oder
  - die Expansionstiefe die Abbruch-Expansionstiefe erreicht hat.

  Die Abbruch-Expansionstiefe ist eine Konstante in KIEL, die der Benutzer in der Ressourcedatei `kiel.rc` (siehe 3.4.1) setzen kann. Sie gibt dem System ein Entscheidungskriterium für die Terminierung einer Auswertung. Ist die Abbruch-Expansionstiefe bei einer Auswertung erreicht, hat der Benutzer die Möglichkeit, diese Auswertung trotzdem fortzuführen. Dazu klickt er denselben Knoten wiederholt mit der linken Maustaste an. Der interne Expansionstiefezähler wird damit wieder auf Null zurückgesetzt.

- Expandieren eines Rechenvorschriftsaufrufs

  Durch einen Klick mit der **rechten** Maustaste auf einen Rechenvorschriftsknoten wird dieser expandiert (siehe Definition 2.19).

- Vereinfachen einer `if`-Anweisung

  Das Vereinfachen einer `if`-Anweisung ist ebenfalls mit einem Klick der **rechten** Maustaste auszuführen (siehe Definition 2.20).

- Ändern der Anzeige

  Mit der **mittleren** Maustaste kann bei folgenden Knoten ein Popup-Menü zum Ändern der Anzeige geöffnet werden:

  - ausgewertete Knoten vom Typ `real`

    Mit dem Einstellrad kann hier die Anzahl der Nachkommastellen eingestellt werden.
  - ausgewertete Knoten vom Typ `string`

    Das Popup-Menü dient in diesem Fall zur Einstellung der maximal anzuzeigenden Zeichen im String.

  - ausgewertete Knoten von Sequenztypen

    Bei beliebigen Sequenztypen wird mit dem Einstellrad die maximale Anzahl der anzuzeigenden Komponenten festgelegt.

  Nach dem Druck auf den Knoten öffnet sich das Popup-Menü. Nun ist zur Änderung der Anzeige mit einer beliebigen Maustaste auf den Zeiger des Einstellrades zu drücken, diese gedrückt zu halten, die gewünschte Einstellung durch Bewegen der Maus auszuwählen und abschließend die Maustaste loszulassen. Mit dem Loslassen wird die Änderung übernommen und das Popup-Menü geschlossen.

- Vereinfachung so weit wie möglich

  Das Drücken der **mittleren** Maustaste über einem **freien** Punkt[1] führt dazu, daß im gesamten Baum alle elementaren Operationen und Fallunterscheidungen, bei deren Auswertung kein Rechenvorschriftsknoten zu expandieren ist, ausgewertet bzw. vereinfacht werden.

Um die Termersetzung durch Vereinfachung und Expansion, wie sie in 2.3.3 beschrieben ist, in KIEL einfach nachzuvollziehen, sollte die Anzahl der auszuführenden Auswertungsschritte im Auswertungsfenster (siehe 3.3.8) auf einen Schritt gesetzt werden. Anschließend ist immer abwechselnd die mittlere Maustaste über einem freien Punkt der Zeichenfläche zu drücken und die Wurzel des Baumes mit der linken Maustaste anzuklicken.

Beim Auswerten, Expandieren und Vereinfachen eines Knotens ändert sich im allgemeinen die Größe der Zeichenfläche für den Baum. Damit der Benutzer effektiv arbeiten kann, ist es von Vorteil, den Ergebnisknoten der letzten Bearbeitung schnell erkennen zu können. Aus diesem Grund stellt KIEL den zuletzt bearbeiteten Knoten dick umrandet dar. Gerade bei Auswertungsabbrüchen und nicht kompletten Auswertungen von Knoten geschieht es häufig, daß die Position des angeklickten Knotens weit von seiner ursprünglichen Position entfernt ist. In der Regel würde somit der zuletzt bearbeitete Knoten nicht sichtbar sein und müßte zur weiteren Bearbeitung zunächst mit Hilfe der Scrollbalken gesucht werden. Um dieses uneffektive Suchen zu umgehen, führt KIEL nach jeder Bearbeitung eine Umpositionierung der Zeichenfläche so durch, daß die Position eines Knotens vor und nach seiner Bearbeitung möglichst identisch ist. Für den Benutzer bedeutet dies, daß er sofort mit diesem Knoten weiterarbeiten kann. Eine solche Umpositionierung ist nicht in allen Fällen optimal möglich. Diese Fälle treten genau dann auf, wenn der Knoten nach seiner Bearbeitung näher an einem Rand liegt als vorher. In diesen Fällen führt KIEL die Umpositionierung so weit wie möglich aus. Dadurch wird erreicht, daß der zuletzt bearbeitete Knoten immer zumindest auf dem sichtbaren Teil der Zeichenfläche liegt.

Wenn der bearbeitete Ausdruck im Laufe der Bearbeitung mehr und mehr Platz einnimmt, wird der Benutzer mitunter das Baumfenster vergrößern wollen. Er kann das Baumfenster in der gewohnten Weise beliebig in seiner Größe ändern. Während eines Bearbeitungsschrittes ist diese Möglichkeit allerdings nicht vorhanden. Da die Expansion eines Rechenvorschriftsaufrufs sehr schnell geschieht, kann diese Einschränkung nur bei einer lange andauernden Auswertung oder der Vereinfachung einer `if`-Anweisung bemerkt werden; in dem zweiten Fall auch nur dann, wenn die Auswertung des booleschen Ausdrucks genügend lange dauert. Diese Einschränkung ist in KIEL eingebaut, da bei jeder Größenänderung des Baumfensters der Ausdrucksbaum neu gezeichnet wird und zu dem Zeitpunkt, an dem KIEL die Aufforderung zum Neuzeichnen bekommt, kein vollständig definierter Baum im System vorliegen muß. Außerdem bedeutet diese Einschränkung einen Geschwindigkeitsgewinn.

Aus denselben Gründen, die im vorigen Absatz genannt wurden, wird ein Verschieben der Scrollbalken im Baumfenster erst nach Beendigung des Bearbeitungsschrittes auf der Zeichenfläche realisiert. Dies ist aufgrund des besonderen Aufbaus des Baumfensters möglich. Weitere Ausführungen sind in 5.6.3 zu finden.

---

[1] Um die Vereinfachung auszuführen, kann jeder Bereich der Zeichenfläche angeklickt werden, der nicht Teil eines einen Knoten umschließenden Rechtecks ist.

Weitere Operationen, die auf dem Baum ausgeführt werden können, wie etwa das Rückgängigmachen von Auswertungsschritten (siehe 3.3.9), werden zusammen mit den jeweiligen Fenstern beschrieben, aus denen sie aufgerufen werden können.

3.3.7 Zeichenreihenfenster

Abbildung 3.8: Das Zeichenreihenfenster

Das Zeichenreihenfenster (siehe Abbildung 3.8) dient zum Anzeigen des aktuell bearbeiteten Ausdrucks als Zeichenreihe. Es besteht lediglich aus einem Label, der ggf. gescrolled werden kann. Durch einen in 3.3.1 beschriebenen Button im Fensterrahmen des Befehlsfensters wird das Zeichenreihenfenster geöffnet und geschlossen.

Im Zeichenreihenfenster ist immer genau der Ausdruck als Zeichenreihe angezeigt, der im Baumfenster (siehe 3.3.6) als Baum zu sehen ist. Im Normalfall läßt das Baumfenster einen besseren Überblick über den Ausdruck zu als das Zeichenreihenfenster. Aber dennoch ist das Zeichenreihenfenster als Alternative oder Ergänzung zum Baumfenster anzusehen. Wird im Baumfenster ein Fehlerknoten angezeigt, so steht an der entsprechenden Stelle im Zeichenreihenfenster der Text `FEHLER`.

3.3.8 Auswertungsfenster

Das in Abbildung 3.9 dargestellte Auswertungsfenster dient zur Festlegung von Auswertungsparametern sowie zum Anhalten einer Auswertung. Es besteht aus den folgenden drei Teilen:

- dem Auswertungsartrahmen,
- dem Anzahl-der-Auswertungen-Rahmen und
- dem Button `Auswertung anhalten`.

3.3.8.1 Auswertungsartrahmen

Im Auswertungsartrahmen wird die Art der Auswertung festgelegt. Dem Benutzer stehen folgende Termersetzungssemantiken (siehe 2.3.3) als Auswahl zur Verfügung:

- Leftmost-Innermost-Ersetzung
- Leftmost-Outermost-Ersetzung
- Full-Ersetzung

Die Voreinstellung beim Start von KIEL wird durch Einträge in der Ressourcedatei `kiel.rc` (siehe 3.4.1) bestimmt.

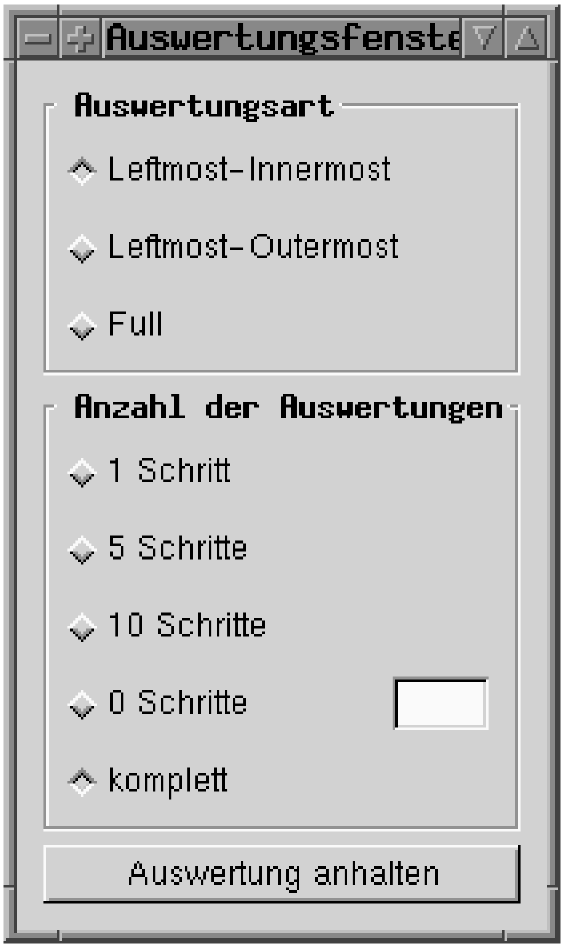


Abbildung 3.9: Das Auswertungsfenster

3.3.8.2 Anzahl-der-Auswertungen-Rahmen

Im Anzahl-der-Auswertungen-Rahmen legt der Benutzer die maximale Anzahl der bei einer Auswertung auszuführenden Auswertungsschritte fest. Ihm bieten sich folgende Optionen:

- `1 Schritt`

  Es wird bei jedem Klick mit der linken Maustaste auf einen nicht ausgewerteten Knoten im Baumfenster (siehe 3.3.6) nur ein Auswertungsschritt ausgeführt. Mit dieser Einstellung kann eine Auswertung Schritt für Schritt verfolgt werden.

- `5 Schritte`, `10 Schritte`

  Sollen mehrere Auswertungsschritte ausgeführt werden, so kann der Benutzer eine dieser Optionen wählen. Es werden bei einer Auswertung maximal fünf bzw. zehn Auswertungsschritte durchgeführt. Dies beschleunigt die Auswertung. Ist ein Knoten nach weniger als der Maximalanzahl von fünf bzw. zehn Auswertungsschritten komplett ausgewertet, so werden keine weiteren Auswertungsschritte ausgeführt.

- `...Schritte`

  Dieser Punkt ermöglicht es dem Benutzer, eine beliebige Maximalanzahl an Auswertungsschritten festzusetzen. In dem Eingabefeld im rechten Teil kann eine beliebige Zahl von Null bis 9999 eingegeben werden. Bei jeder Eingabe wird die Beschriftung dieses Auswahlpunktes aktualisiert. Gibt der Benutzer ein Zeichen ein, das keine Ziffer ist, so werden keine Auswertungsschritte ausgeführt, wenn dieser Punkt ausgewählt ist.

- `komplett`

  Ist dieser Punkt bei einer Auswertung aktiv, werden an dem jeweiligen Knoten so lange Auswertungsschritte vorgenommen, bis dieser ausgewertet ist. Lediglich ein Abbruch vom Benutzer (siehe unten) oder ein Erreichen der maximalen Expansionstiefe führen zu einem vorzeitigen Auswertungsabbruch.

Ebenso wie im vorigen Punkt wird die Voreinstellung beim Start von KIEL durch Einträge in der Ressourcedatei `kiel.rc` (siehe 3.4.1) bestimmt.

3.3.8.3 Button `Auswertung anhalten`

Mit diesem Button hat der Benutzer die Möglichkeit, eine laufende Auswertung jederzeit anzuhalten. Ein Betätigen dieses Buttons während einer Auswertung veranlaßt KIEL, sofort mit der Auswertung aufzuhören und das momentane Zwischenergebnis als Ergebnis der Auswertung anzusehen. Dies ist der gleiche Effekt wie ein Erreichen der maximalen Auswertungsanzahl.

Durch diesen Button kann der Benutzer insbesondere nicht terminierende Auswertungen und solche, die nach seinem Verständnis wesentlich schneller terminieren sollten, frühzeitig anhalten. Dies ist bei der Verwendung von KIEL bei der Fehlersuche von großer Bedeutung. Mit Hilfe des Undo-Redo-Fensters (siehe 3.3.9) ist in einem solchen Fall die letzte Auswertung einfach rückgängig zu machen. Anschließend kann dann der Fehler mit einer geringeren Maximalanzahl von Auswertungsschritten aufgespürt werden.

Eine angehaltene Auswertung kann durch nochmaliges Klicken mit der linken Maustaste auf denselben Knoten fortgesetzt werden. Dabei werden die Zähler für die Anzahl der Auswertungsschritte und für die Expansionstiefe zu Beginn auf Null zurückgesetzt.

3.3.9 Undo-Redo-Fenster

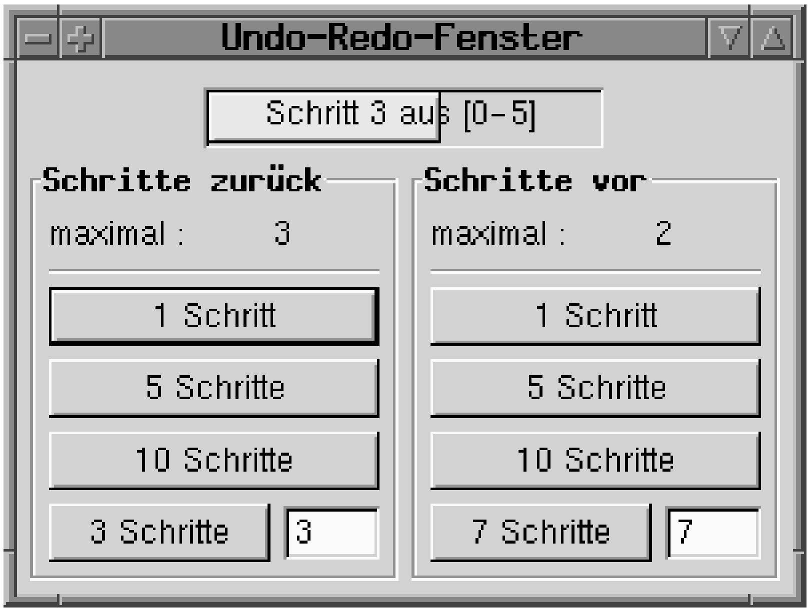


Abbildung 3.10: Das Undo-Redo-Fenster

Das Undo-Redo-Fenster (siehe Abbildung 3.10) stellt für die Arbeit mit KIEL wesentliche Funktionen bereit. Es ist hiermit möglich, in der Folge der vorgenommenen Bearbeitungen eines Ausdrucks zu springen, insbesondere können Auswertungen rückgängig gemacht werden. Das Undo-Redo-Fenster besteht aus den folgenden drei Teilen:

- einer Fortschrittsanzeige,
- einem Rahmen zum Rückwärtsgehen in der Bearbeitungsfolge und
- einem Rahmen zum Vorwärtsgehen in der Bearbeitungsfolge.

KIEL protokolliert jede Bearbeitung eines Knotens in einem Ausdruck, also jede Auswertung, jede Expandierung und jede Vereinfachung. Der obere Wert der Fortschrittsanzeige ist stets gleich der Anzahl der Protokolleinträge, die für den im Ausdruckübersichtsfenster (siehe 3.3.4) markierten Ausdruck vorhanden sind. Der Balken in der Fortschrittsanzeige zeigt an, an welcher Position im Protokoll sich KIEL gerade befindet. Zur Änderung der Position im Auswertungsprotokoll dienen die beiden Rahmen. Im oberen Teil wird in beiden Rahmen die aktuelle Maximalanzahl an Positionen angezeigt, um die sich der Benutzer im Protokoll zurück- bzw. vorbewegen kann. Außerdem haben beide Rahmen die folgenden vier Buttons zum Bewegen innerhalb des Protokolls:

- `1 Schritt`, `5 Schritte`, `10 Schritte`

  Durch Betätigung eines dieser Buttons wird die Position im Protokoll um eine, fünf bzw. zehn Positionen in die jeweilige Richtung verändert. Ist die ausgewählte Anzahl an Positionen größer als die mögliche Maximalanzahl, wird die Position im Protokoll nur um die maximal mögliche Anzahl verändert.

- `...Schritte`

  Dieser Punkt ermöglicht es dem Benutzer, eine beliebige Maximalanzahl festzusetzen, um die die Position im Protokoll geändert werden soll. In dem Eingabefeld im rechten Teil kann eine beliebige Zahl von Null bis 9999 eingegeben werden. Bei jeder Eingabe wird die Beschriftung des entsprechenden Buttons aktualisiert. Gibt der Benutzer ein Zeichen ein, das keine Ziffer ist, so wird die Position im Protokoll nicht geändert, wenn dieser Punkt ausgewählt wird. Soll sich die Position im Protokoll um die eingegebene Anzahl verändern, so muß der Benutzer dies KIEL durch Drücken des entsprechenden Buttons mitteilen.

Der Benutzer hat also die Möglichkeit, Bearbeitungen von Knoten rückgängig zu machen und ebenso die zurückgenommenen Bearbeitungen wiederherzustellen. Er kann sich somit beliebig im Protokoll des im Ausdruckübersichtsfenster (siehe 3.3.4) markierten Ausdrucks bewegen.

Wird nach dem Rückgängigmachen von Bearbeitungen der Ausdruck wieder bearbeitet, so werden im Protokoll des Ausdrucks alle Positionen nach der aktuellen gelöscht; bis zur aktuellen Position bleibt das Protokoll erhalten. Diese Handlungsweise ist sinnvoll, da der Benutzer durch Bearbeitung eines Knotens eine neue Folge von Auswertungen, Expansionen und Vereinfachungen beginnt und da das Protokoll zu jedem Ausdruck in KIEL eine lineare Liste ist.

3.3.10 Warnungsfenster

Bei jeder Anfrage an eine Operation, die KIEL nicht ausführen kann, wird ein Warnungsfenster geöffnet. Exemplarisch ist in Abbildung 3.11 das Warnungsfenster gezeigt, das ein nicht korrekt eingegebener Ausdruck bedingte.

Wird vom System ein Warnungsfenster geöffnet, so ertönt ein akkustisches Signal. Solange der Benutzer dieses Warnungsfenster nicht durch Drücken des `OK`-Buttons schließt, kann er keine Funktion von KIEL

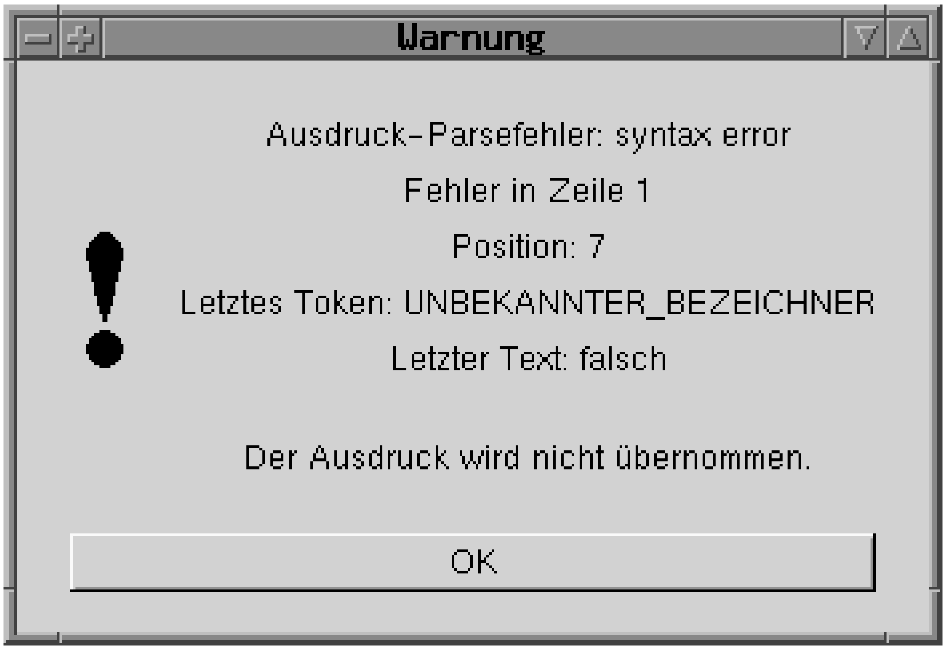


Abbildung 3.11: Ein Warnungsfenster

nutzen. Die einzige Ausnahme bildet der Quitbutton, mit dem der Benutzer jederzeit, also auch bei geöffneten Warnungsfenstern, KIEL verlassen kann.

Es folgt eine Liste von möglichen Warnungen:

- bei der Eingabe von Ausdrücken und Rechenvorschriften
  - Parsefehler
    Entsprechen Ausdrücke oder Rechenvorschriften nicht der von KIEL unterstützten Syntax, wie sie in A.1 und A.2 angegeben ist, wird ein Parsefehler entdeckt. Zusammen mit der Position seines Auftretens wird dieser in einem Warnungsfenster angezeigt. Die Ausdrücke oder Rechenvorschriften werden nicht in das System übernommen.
  - Fehler bei der Strukturüberprüfung
    Vor dem Parsen einer Rechenvorschriftendatei findet eine Strukturüberprüfung des Dateiinhaltes statt. Wenn der Dateiinhalt nicht der Struktur

    (`fun` <Name> `( ... ) :` <Sorte> `=` . . . (`and` <Name> `( ... ) :` <Sorte> `=` . . .)* `;`)*

    entspricht, erkennt KIEL dies als Fehler und gibt eine dementsprechende Fehlermeldung aus. Die fehlerhafte Datei wird nicht geparst und somit auch nicht ins System übernommen.
  - Fehler bei einem Rechenvorschriftsaufruf
    Separat von den im ersten Punkt beschriebenen Parsefehlern prüft KIEL jeden vorkommenden Rechenvorschriftsaufruf auf folgende Punkte:
    - falsche Ergebnissorte
      Dieser Fehler tritt auf, wenn die erwartete Ergebnissorte eines Rechenvorschriftsaufrufs von der Ergebnissorte, die in der entsprechenden Rechenvorschrift steht, abweicht.
    - falsche Anzahl an Argumenten
      Wird eine Rechenvorschrift mit einer anderen Anzahl an Argumenten aufgerufen, als in der entsprechenden Rechenvorschrift angegeben ist, kommt es zu diesem Fehler.

    * falsche Sorte
      Unterscheidet sich die Sorte eines Arguments von der in der entsprechenden Rechenvorschrift angegebenen, handelt es sich ebenfalls um eine fehlerhafte Eingabe.

  Kommt es zu einem Fehler, wird eine Warnung ausgegeben. In allen drei Fällen wird der Ausdruck, in dem der fehlerhafte Rechenvorschriftsaufruf steht, nicht übernommen.

  - Bereichsüberschreitung von Werten

    Für Werte der Typen `int`, `real` und `string` sowie für Listentypen gibt es in KIEL Bereichsgrenzen. Diese sind in 5.3 beschrieben. Findet eine Bereichsüberschreitung durch einen Wert statt, wird der dazugehörende Knoten im Auswertungsbaum zu einem Fehlerknoten, und der Benutzer wird entsprechend gewarnt.

  - Fehler beim Laden einer Eingabedatei

    Kann KIEL eine zu ladende Datei nicht fehlerfrei einlesen, können die Daten dieser Datei nicht ins System übernommen werden.

- beim Auswerten eines Knotens

  Bei der Auswertung eines Knotens kann es zu Undefiniertheiten und zu Konflikten mit den Bereichsgrenzen kommen. In jedem Fall wird der Ergebnisknoten der Auswertung zu einem Fehlerknoten.

  - Bereichsüberschreitungen

    Die genauen Bereichsgrenzen aller Typen sind in 5.3 aufgeführt. Es sind folgende Überschreitungen möglich:

    * `int`-Überlauf
      Eine Operation hat als Ergebnis einen `int`-Wert, der den maximal zulässigen `int`-Wert überschreitet.
    * `int`-Unterlauf
      Eine Operation hat als Ergebnis einen `int`-Wert, der den minimal zulässigen `int`-Wert unterschreitet.
    * `real`-Überlauf
      Eine Operation hat als Ergebnis einen `real`-Wert, der den maximal zulässigen `real`-Wert überschreitet.
    * `real`-Unterlauf
      Eine Operation hat als Ergebnis einen `real`-Wert, der den minimal zulässigen `real`-Wert unterschreitet.
    * Längenüberschreitung bei Zeichenreihen
      Beim Aneinanderhängen zweier `string`-Werte entsteht ein `string`-Wert, der mehr Zeichen hat, als von KIEL zugelassen sind.
    * Längenüberschreitung bei Sequenzen
      Eine Operation liefert eine Sequenz, die mehr Elemente besitzt, als von KIEL zugelassen sind.

  - Undefiniertheiten

    Während einer Auswertung kann es zu folgenden Undefiniertheiten der Basisoperationen kommen:

    * bei Operationen auf Werten des Typs `int`
      · i_0 `mod` i_1, falls $i_1 = 0$
      · i_0 `div` i_1, falls $i_1 = 0$
    * bei Operationen auf Werten des Typs `real`
      · r_0 `/` r_1, falls $r_1 = 0$
      · `sqrt(`r`)`, falls $r < 0$

- · `ln(`r`)`, falls $r \leq 0$[2]
- ∗ bei Operationen auf Werten des Typs `string`
  - · `ord(`s`)`, falls s die leere Zeichenreihe
  - · `chr(`i`)`, falls $i < 0$ oder $i > 255$
  - · `substring(`s`,`i`,`n`)`, falls $i < 0$, $n < 0$ oder $i + n > |s|$
- ∗ bei Operationen auf Werten von Sequenztypen
  - · `hd(`l`)`, falls l die leere Liste
  - · `tl(`l`)`, falls l die leere Liste

- bei der Anzeige des Auswertungsbaumes

  In KIEL ist die Maximalanzahl der anzuzeigenden Ebenen eines Auswertungsbaumes in einer Konstanten festgelegt. Der Benutzer hat die Möglichkeit, den Wert dieser Konstanten durch die Ressourcedatei `kiel.rc` (siehe 3.4.1) zu bestimmen. Ist beim Versuch der Darstellung des Baumes dieser Wert überschritten, wird eine Warnung ausgegeben, und der Auswertungsbaum wird gelöscht.

- beim Öffnen eines Laden-Fensters

  Es ist festgelegt, daß jeweils nur ein Ausdruckdatei-Laden-Fenster und ein Rechenvorschriftendatei-Laden-Fenster zur Zeit geöffnet sein dürfen. Versucht der Benutzer, ein weiteres Fenster durch Drücken des entsprechenden Buttons im Befehlsfenster zu öffnen, wird ihm mitgeteilt, daß bereits ein Fenster zum Laden einer Ausdruckdatei bzw. einer Rechenvorschriftendatei geöffnet ist.

3.4 Ressourcedateien

Ressourcedateien dienen zum Einstellen von Konstanten eines Programms. Sie werden zur Laufzeit eingelesen und bestimmen mitunter maßgeblich die Ausführung eines Programms.

KIEL arbeitet mit zwei Arten von Ressourcedateien. Es gibt

- Standard-Ressourcedateien, in denen den Konstanten typische Werte zugewiesen werden, und
- benutzereigene Ressourcedateien, in denen der Benutzer selbst Änderungen an den Belegungen der Konstanten vornehmen kann, um KIEL seinen Bedürfnissen entsprechend anzupassen.

Es gibt die beiden Standard-Ressourcedateien

- `kiel.rc` (siehe 3.4.1) zum Festlegen von Fensterpositionen und -größen sowie verschiedenen Voreinstellungen und
- `kiel.rc.gtk` (siehe 3.4.2) zum Einstellen von Farben und Schriften.

Sie befinden sich in demselben Verzeichnis wie das System selbst.

Der Benutzer hat die Möglichkeit, seine eigenen Voreinstellungen in KIEL durch Änderung der Konstantenbelegungen vorzunehmen. Dazu muß der in seinem Heimatverzeichnis die Dateien

`.kiel.rc`

und/oder

[2]Hierbei kann ≤ 0 nicht exakt mathematisch behandelt werden. Nicht alle reellen Zahlen können im Computer exakt dargestellt werden. Deshalb wird hier genaugenommen überprüft, ob das Argument kleiner als die kleinste positive reelle Zahl doppelter Genauigkeit ist.

`.kiel.rc.gtk`

erstellen. Bei Vorhandensein zum Zeitpunkt des Programmstarts werden sie anstelle der entsprechenden Standard-Ressourcedateien nach KIEL geladen. Es empfiehlt sich daher, die Standard-Ressourcedateien in das Heimatverzeichnis zu kopieren und dort entsprechend umzubenennen.

Existiert zum Zeitpunkt des Programmstarts die Datei `.kiel.rc` im Heimatverzeichnis des Benutzers nicht, so wird die Standard-Ressourcedatei `kiel.rc` eingelesen. Analoges gilt für die zweite Ressourcedatei.

3.4.1 Die Dateien `kiel.rc` und `.kiel.rc`

Über die Ressourcedatei `kiel.rc` bzw. `.kiel.rc` werden in KIEL die in der Tabelle 3.2 aufgeführten Konstanten mit Werten belegt. Durch Abändern der Datei `.kiel.rc` kann der Benutzer diese Werte selbst bestimmen.

Konstantenname	Typ	Bedeutung
	Positionsangaben	
POSITION_BEFEHLSFENSTER	$\mathbb{N}^a, \mathbb{N}$	für das Befehlsfenster
POSITION_BAUMFENSTER		für das Baumfenster
POSITION_ZEICHENREIHENFENSTER		für das Zeichenreihenfenster
POSITION_ADKUEBERSICHTSFENSTER		für das Ausdruckübersichtsfenster
POSITION_RVUEBERSICHTSFENSTER		für das Rechenvorschrifts-Übersichtsfenster
POSITION_UNDOREDOFENSTER		für das Undo-Redo-Fenster
POSITION_ADKFILESELECTION		für das Ausdruckdatei-Laden-Fenster
POSITION_PRGFILESELECTION		für das Rechenvorschriftendatei-Laden-Fenster
POSITION_AUSDRUCKEINGABEFENSTER		für das Ausdruckeingabefenster
POSITION_AUSWERTUNGSFENSTER		für das Auswertungsfenster
	Ausmaße von Komponenten	
BREITE_DRAWING_AREA_BAUM	$\mathbb{N}$	Breite der Zeichenfläche im Baumfenster
HOEHE_DRAWING_AREA_BAUM		Höhe der Zeichenfläche im Baumfenster
BREITE_SCROLLEDWINDOW_ZR		Breite des Zeichenreihen-Scrollfensters
BREITE_AUSDRUCKEINGABEFENSTER_ENTRY		Breite des Ausdruckeingabefeldes
BREITE_UEBERSICHTS_SCROLLED_WIN		Breite der Übersichts-Scrollfenster
HOEHE_UEBERSICHTS_SCROLLED_WIN		Höhe der Übersichts-Scrollfenster
DURCHMESSER_NACHKOMMASTELLEN_DIAL		Durchmesser der Einstell-Drehscheibe
	Angaben über anzuzeigende Stellen bzw. Komponenten	
VORGABE_NACHKOMMASTELLEN	$\mathbb{N}$	Vorgabe der Anzahl der Nachkommastellen
MAXIMALE_NACHKOMMASTELLEN		Maximalanzahl der Nachkommastellen
VORGABE_ANZAHLZEICHENIMSTRING		Vorgabe der Anzahl der Zeichen je String
MAXIMALE_ANZAHLZEICHENIMSTRING		Maximalanzahl der Zeichen je String
VORGABE_ANZAHLELEMENTEINLISTE		Vorgabe der Anzahl der Elemente je Liste
MAXIMALE_ANZAHLELEMENTEINLISTE		Maximalanzahl der Elemente je Liste
ABBRUCH_EXPANSIONS_TIEFE	$\mathbb{N}$	Expansionstiefe für Auswertungsabbruch
MAX_ANZUZEIGENDE_EBENEN_ANZ		Maximalanzahl der anzuzeigenden Ebenen
	Vorgaben für Verzeichnisse	
WINDOW_AUSDRUCK_DATEIAUSWAHL_DIRECTORY	ZR[b]	für Ausdrücke
WINDOW_PROGRAMM_DATEIAUSWAHL_DIRECTORY		für Programme

Anfangsstatus der Togglebuttons im Befehlsfenster

BUTTON_MLAUSDRUCK_UEBERSICHT_STATUS	$\mathbb{B}$[c]	Ausdrucksübersicht-Button
BUTTON_RV_UEBERSICHT_STATUS		Rechenvorschriften-Übersicht-Button
BUTTON_AUSDRUCKBAUM_FENSTER_STATUS		Ausdrucksbaum-Button
BUTTON_ZEICHENREIHEN_FENSTER_STATUS		Zeichenreihenfenster-Button
BUTTON_UNDOREDO_FENSTER_STATUS		Undo-Redo-Fenster-Button
BUTTON_AUSWERTUNGS_FENSTER_STATUS		Auswertungsfenster-Button

Anfangsstatus der Radiobuttons zur Ersetzungsauswahl im Auswertungsfenster

BUTTON_LI_STATUS	$\mathbb{B}$	Leftmost-Innermost-Button
BUTTON_LO_STATUS		Leftmost-Outermost-Button
BUTTON_FULL_STATUS		Full-Button

Anfangsstatus der Radiobuttons zur Auswertungsschrittauswahl im Auswertungsfenster

BUTTON_1_SCHRITT_STATUS	$\mathbb{B}$	1-Auswertungsschritt-Button
BUTTON_5_SCHRITTE_STATUS		5-Auswertungsschritte-Button
BUTTON_10_SCHRITTE_STATUS		10-Auswertungsschritte-Button
BUTTON_VAR_SCHRITTE_STATUS		Variabel-Auswertungsschritte-Button
BUTTON_BIS_FERTIG_STATUS		Bis-fertig-Auswertung-Button

Grafik-Konstante

GK_OFFSET_H	$\mathbb{N}$	Horizontaler Offset
GK_OFFSET_V		Vertikaler Offset
GK_ZEILENHOEHE		Zeilenhöhe
GK_KNOTEN_MINDESTABSTAND		Horizontaler Mindestabstand zweier Knoten
GK_KNOTENBREITE_ZU_SCHRIFTBREITE	$\mathbb{R}$[d]	Verhältnis der Knotenbreite zur Schriftbreite
GK_KNOTENHOEHE_ZU_SCHRIFTHOEHE		Verhältnis der Knotenhöhe zur Schrifthöhe

[a] $\mathbb{N}$ steht für die Menge der natürlichen Zahlen.
[b] ZR steht für die Menge der Zeichenreihen.
[c] $\mathbb{B}$ steht für die Menge der booleschen Werte.
[d] $\mathbb{R}$ steht für die Menge der reellen Zahlen.

Tabelle 3.2: Die über `kiel.rc` festzulegenden Konstanten

Die Syntax zum Ändern der Belegung einer der Konstanten ist:

`#set` <Konstantenname> <Wert>

Dabei ist

- <Konstantenname> einer der Konstantennamen aus Tabelle 3.2 und
- <Wert> ein Wert mit dem der Tabelle 3.2 zu entnehmenden Typ.

Als Beispiel folgen zwei Zeilen der Datei `kiel.rc`:

```
#set POSITION_AUSWERTUNGSFENSTER     833, 414
#set BREITE_DRAWING_AREA_BAUM             600
```

Durch diese Zeilen wird

- die Position der linken oberen Ecke der Auswertungsfensters auf den Punkt (833,414) gesetzt, sowie
- die Breite der Zeichenfläche im Baumfenster auf den Wert 600 festgelegt.

Alle Zeilen, die nicht mit `#set` beginnen, werden von KIEL als Kommentar interpretiert.

Hält sich der Benutzer nicht an die vorgegebene Syntax, oder gibt er als Konstantennamen einen nicht in der Tabelle 3.2 stehenden an, bricht das Programm mit einer Fehlermeldung ab. Da das Einlesen der Ressourcedateien zu Beginn der Programmausführung geschieht, gehen dabei keine Daten verloren. Der Benutzer kann im Anschluß die Datei `.kiel.rc` editieren und KIEL erneut starten.

3.4.2 Die Dateien `kiel.rc.gtk` und `.kiel.rc.gtk`

Grundsätzlich kann der Benutzer durch Editieren der Datei `.kiel.rc.gtk` die Farben von sämtlichen Grafikelementen von KIEL, wie zum Beispiel Buttons, Rahmen oder Fenster, festlegen. Relevant sind zum einen die Einstellung der Schrift und Farben der Zeichenflächen, die in KIEL benutzt werden. Dies sind

- die Zeichenfläche im Baumfenster und
- die Zeichenflächen im Befehlsfenster, in denen die Maustastenbelegung angezeigt wird.

Zum anderen kann für alle Grafikelemente die Schriftart beliebig gesetzt werden. In der Datei `kiel.rc.gtk` ist vorgesehen, daß die Überschriften aller Rahmen in KIEL in der Schrift `7x13bold`, also fett, angezeigt werden. Außerdem sind Vorlagen für Schrifteinstellungen der Labels und Eingabefelder, sowie aller Fortschrittsbalken vorhanden. Da in KIEL teilweise die Größe der Eingabefelder auf die Größe der Labels abgestimmt ist, sollten die Einstellungen für diese Elemente einander immer gleichen. Im folgenden ist die Datei `kiel.rc.gtk` aufgelistet:

```
style "fg_farben"
{
#                  rot gruen blau
  fg[NORMAL]=     {0.8, 0.8, 1.0}  # 0: hellblau fuer ausgewertete Knoten
  fg[ACTIVE]=     {1.0, 1.0, 0.7}  # 1: hellgelb fuer RV-Aufrufe
  fg[PRELIGHT]=   {1.0, 0.5, 0.5}  # 2: rot fuer Fehler-Knoten
  fg[SELECTED]=   {0.0, 0.0, 0.0}  # 3:
  fg[INSENSITIVE]={0.0, 0.0, 0.0}  # 4:
}

style "mausbelegung"="fg_farben"
{
  font="5x8"
}

style "baum"="fg_farben"
{
  font="-schumacher-*-*-*-*-*-*-100-*-*-*-*-*-*"
}

style "framestyle"
{
  font="7x13bold"
}
```

```
style "labelstyle"
{
#  font="7x13"
}

style "progressbarstyle"
{
#  font="10x20"
}

widget "Befehlsfenster.*GtkDrawingArea*" style "mausbelegung"
widget "Baumfenster.*GtkDrawingArea*" style "baum"
widget "*GtkFrame" style "framestyle"
widget "*GtkLabel" style "labelstyle"
widget "*GtkEntry" style "labelstyle"
widget "*GtkProgressBar" style "progressbarstyle"
```

Die Vordergrundfarbe `NORMAL` bestimmt die Farbe, in der ausgewertete Knoten in den Zeichenflächen angezeigt werden. Rechenvorschriftsaufrufe werden in der Vordergrundfarbe `ACTIVE` dargestellt. Der Hintergrund eines Fehlerknotens hat die Vordergrundfarbe `PRELIGHT`. In der Ureinstellung sind dies hellblau, hellgelb bzw. rot.

Die zu verwendende Schriftart ist für das Baum- und das Befehlsfenster getrennt einstellbar. Für das Baumfenster ist eine etwas größere und dickere Schriftart vorgesehen. Änderungen dieser Ressourcen sind in der Datei `.kiel.rc.gtk` vorzunehmen. Das Zeichen `#` signalisiert dieser Ressourcedatei, daß der Rest der aktuellen Zeile ein Kommentar ist. Dies ist bei der Definition der Stile `labelstyle` und `progressbarstyle` zu erkennen. Somit kann sich der Benutzer mehrere alternative Schriftarten und -größen in jedem beliebigen Stil der Datei `.kiel.rc.gtk` definieren. Dabei sollte darauf geachtet werden, daß höchstens eine Definition nicht auskommentiert ist. Soll eine alternative Schrift gewählt werden, kann die vorige Auswahl auskommentiert werden und das Kommentarzeichen am Anfang der Zeile, in der die Alternative definiert wird, gelöscht werden.

Kapitel 4

Beispiele

Nach der Beschreibung der Komponenten von KIEL in Kapitel 3 werden nun einige Beispiele für die praktische Arbeit mit KIEL gegeben. Für Benutzer, die das erste Mal Kontakt mit KIEL haben, empfiehlt es sich, diese Beispiele am Rechner nachzuvollziehen, um mit dem System vertraut zu werden. Neben der Anzeige von Bildschirmausdrucken werden an interessanten Stellen auch Angaben über die Anzahl der erfolgten Bearbeitungsschritte in Abhängigkeit der verwendeten Ersetzungsstrategie angegeben. Insgesamt werden drei Beispiele behandelt, die zusammen eine guten Überblick über die Möglichkeiten von KIEL geben.

Im ersten Beispiel wird die Verwendung von Rechenvorschriften behandelt. Dazu werden exemplarisch Rechenvorschriften für die **Fakultätsfunktion** und die **Fibonacci-Zahlen** betrachtet. Ein einfacher Ausdruck, der Aufrufe dieser beiden Rechenvorschriften beinhaltet, wird Schritt für Schritt bearbeitet. Dabei wird auf

- das Laden von Programmen,
- das Eingeben von Ausdrücken und
- die Funktionalität der rechten und linken Maustaste

eingegangen.

Im zweiten Beispiel steht die Verwendung des Auswertungsfensters und des Undo-Redo-Fensters im Vordergrund. Außerdem wird die Auswertung durch Vereinfachung und Expansion aus 2.3.3 angewendet. Im einzelnen werden

- die Begrenzung einer Auswertung auf beliebig viele Auswertungsschritte,
- die Verwendung verschiedener Termersetzungssemantiken,
- das Rückgängigmachen von Bearbeitungsschritten,
- das Wiederherstellen von Bearbeitungsschritten und
- die Funktionalität der mittleren Maustaste auf einem freien Teil der Zeichenfläche

behandelt. Dazu wird die **Morris-Funktion** sowohl nach der Leftmost-Innermost-, als auch nach der Leftmost-Outermost-Strategie ausgewertet. Das besondere an dieser Funktion ist, daß bestimmte Argumente bewirken, daß sie je nach Ersetzungsart unterschiedliche Ergebnisse liefert (siehe 2.3.3).

Das dritte Beispiel widmet sich

- der Verwendung der mittleren Maustaste auf einem Knoten.

In diesem Beispiel werden die Datentypen der Listen, der Gleitkommazahlen und der Zeichenreihen verwendet. Dadurch ist die mittlere Maustaste im Gegensatz zu den ersten beiden Beispielen, in denen nur mit ganzen Zahlen gearbeitet wird, beim Klicken auf Knoten des Baumes von Bedeutung. Mit ihrer Hilfe kann hier die Anzeige verschiedener Inhalte von Knoten variiert werden.

4.1 Elementare Programme auf ganzen Zahlen

Das erste Beispiel verwendet zwei Rechenvorschriften, die vor der Bearbeitung in einer Datei zu sichern sind.

- Die erste Rechenvorschrift berechnet zu einer ganzen Zahl das Ergebnis der Fibonacci-Funktion, die wie folgt rekursiv definiert ist:

$$\begin{array}{rcl} \mathcal{FIB} & : & \mathbb{Z} \to \mathbb{Z} \\ x & \longmapsto & \begin{cases} 1 & : \text{ falls } x \in \{0,1\} \\ \mathcal{FIB}(x-1) + \mathcal{FIB}(x-2) & : \text{ falls } x \geq 2 \\ \perp & : \text{ sonst} \end{cases} \end{array}$$

  Sie lautet:

```
fun FIB(x:int):int=
    if (x=0) orelse (x=1)
        then 1
        else FIB(x-1)+FIB(x-2);
```

- Die zweite Rechenvorschrift bestimmt zu einer ganzen Zahl ihre Fakultät, welche wie folgt rekursiv definiert wird:

$$\begin{array}{rcl} \mathcal{FAC} & : & \mathbb{Z} \to \mathbb{Z} \\ x & \longmapsto & \begin{cases} 1 & : \text{ falls } x = 0 \\ x \cdot \mathcal{FAC}(x-1) & : \text{ falls } x \geq 1 \\ \perp & : \text{ sonst} \end{cases} \end{array}$$

  Diese Rekursion wurde in der Deklaration der Rechenvorschrift übernommen.

```
fun FAC(x:int):int=
    if x=0
        then 1
        else x*FAC(x-1);
```

Es soll nun der Ausdruck

`FIB(FAC(3)+7)`

in KIEL bearbeitet werden.

Zunächst muß die Datei, in der die beiden obigen Rechenvorschriften definiert sind, geladen werden. Dazu wird im Befehlsfenster der Button `aus Datei laden` im Rechenvorschriftenrahmen angeklickt. In dem sich dadurch öffnenden Dateiauswahlfenster ist die zu ladende Datei auszuwählen. Das Drücken des `OK`-Buttons führt zum Laden der Rechenvorschriften.

Nun muß der zu bearbeitende Ausdruck eingegeben werden. An dieser Stelle hat man zwei Möglichkeiten. Entweder kann man den Ausdruck in eine Datei schreiben und diese laden, oder man kann ihn direkt eingeben. Da das Laden bereits eben gezeigt wurde, wird hier der direkte Weg verfolgt. Es ist also der Button

`Eingabe` aus dem Ausdruckrahmen des Befehlsfensters zu drücken. In dem nun geöffneten Ausdruckeingabefenster ist

```
FIB(FAC(3)+7);
```

einzugeben. Durch Drücken der Returntaste wird der Ausdruck in das Ausdruckübersichtsfenster übernommen.

Durch Anklicken des eben eingegebenen Ausdrucks im Ausdruckübersichtsfenster wird das Baumfenster geöffnet und dort der Ausdruck als Baum dargestellt. Zu diesem Zeitpunkt sollte die in Abbildung 4.1 dargestellte Anzeige zu sehen sein.

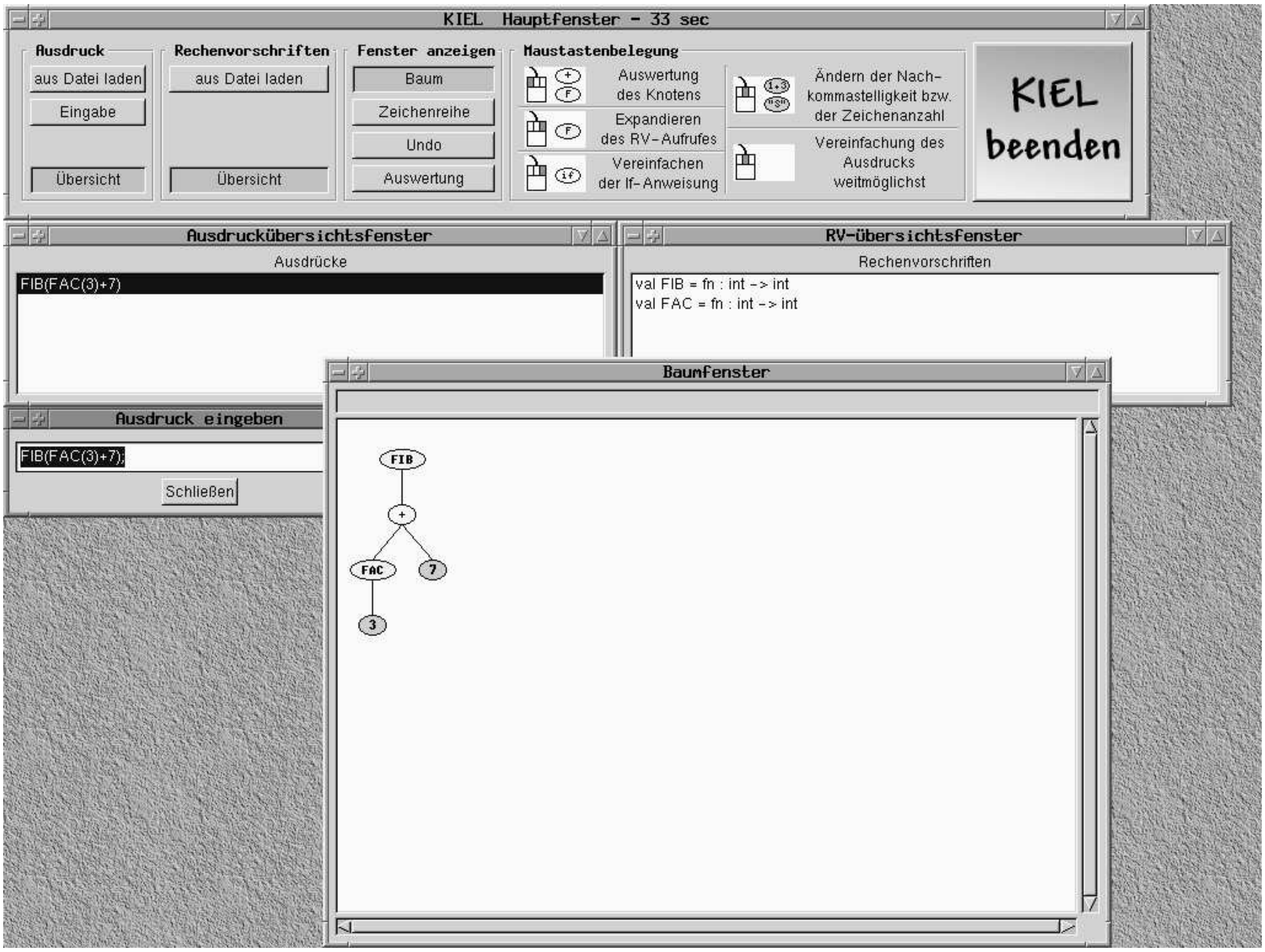


Abbildung 4.1: Der Bildschirm nach Laden eines Programms und Eingabe eines Ausdrucks

Im Baumfenster ist ein Baum mit fünf Knoten zu sehen. Es gibt zwei Rechenvorschriftsknoten, die durch ihren gelben Hintergrund zu erkennen sind. Sie entsprechen dem Aufruf der beiden im Ausdruck auftretenden Funktionsanwendungen. Es gibt eine Basisoperation. Diese Addition auf ganzen Zahlen ist durch den Knoten mit weißem Hintergrund repräsentiert. Die beiden Blätter des Baumes entsprechen den beiden Zahlen im Ausdruck. Ihr Hintergrund ist blau, woran zu erkennen ist, daß sie ausgewertet sind.

Nun kann die eingentliche Bearbeitung des Ausdrucks beginnen. Diese vollzieht sich im Baumfenster durch Anklicken gewisser Knoten mit gewissen Maustasten. Im folgenden wird nur noch das Baumfenster dargestellt. Der Inhalt aller anderen Fenster in Abbildung 4.1 bleibt unverändert.[1]

Zunächst soll der Rechenvorschriftsaufruf der Fakultätsfunktion expandiert werden. Dazu ist der entsprechende Knoten mit der **rechten** Maustaste anzuklicken. Der entstehende Baum ist in Abbildung 4.2 dargestellt.

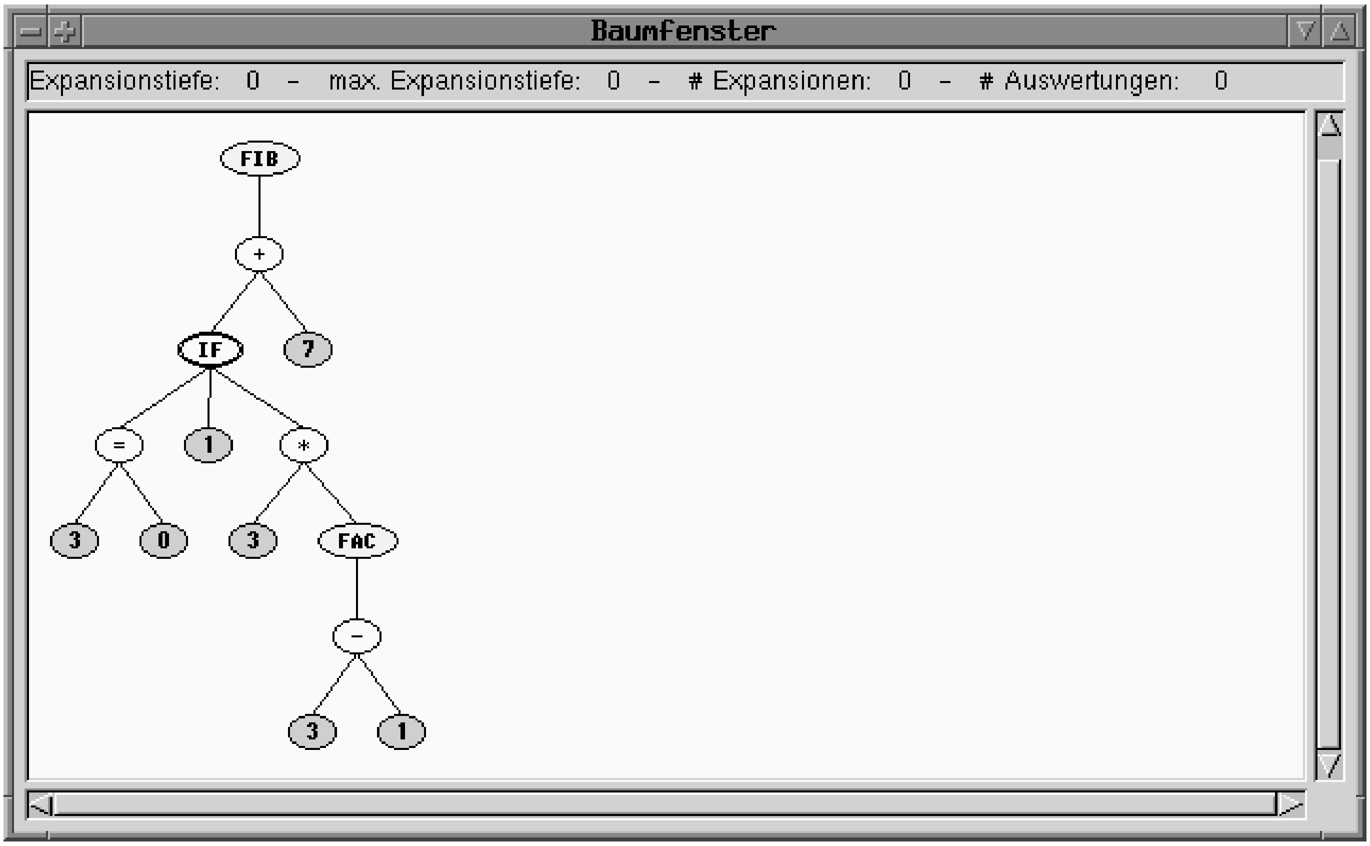


Abbildung 4.2: Das Baumfenster nach Expansion des Aufrufs der Fakultätsfunktion

Der Knoten, in dem der Text `FAC` stand und der ein Kind mit der Beschriftung `3` hatte, ist verschwunden. An seiner Stelle ist nun ein Fallunterscheidungsknoten zu sehen, dessen drei Unterbäume entsprechend der Funktion $\mathcal{FAC}$ gebildet wurden. Der Fallunterscheidungsknoten hat einen dicken Rahmen. Dies verdeutlicht, daß der Knoten, der vorher an dieser Stelle stand, bearbeitet worden ist. Der erste Unterbaum des Fallunterscheidungsknotens stellt den booleschen Ausdruck dar, dessen Wert im weiteren bestimmt, welcher der anderen beiden Unterbäume für die Bestimmung des Wertes des Fallunterscheidungsknotens verwendet werden soll.

Das nächste Ziel ist es, die Fallunterscheidung zu vereinfachen. Zum Vereinfachen von Fallunterscheidungen dient die **rechte** Maustaste. Ein Klick auf den Fallunterscheidungsknoten würde zunächst die Auswertung des ersten Unterbaumes und anschließend in Abhängigkeit des Ergebnisses die Ersetzung des Knotens durch einen der anderen beiden Unterbäume bewirken.

Der Wert des booleschen Ausdrucks ist das Ergebnis des Tests auf Gleichheit von den Werten Drei und Null. Da diese Gleichheit nicht vorliegt, soll der erste Unterbaum des Fallunterscheidungsknotens zu `FALSE` ausgewertet werden. Anschließend soll also der dritte Unterbaum an die Stelle des Fallunterscheidungsknotens treten.

[1] Die einzige Ausnahme bildet die Titelzeile des Befehlsfensters. In ihr wird sekundengenau die Zeit angezeigt, die seit dem Start von KIEL verstrichen ist.

Um diesen Sachverhalt nachzuvollziehen, soll nun der boolesche Ausdruck ausgewertet werden. Dazu ist mit der **linken** Maustaste auf den Knoten zu klicken, der dem Test auf Gleichheit entspricht. In diesem steht ein =-Zeichen. Das sich ergebende Baumfenster ist in Abbildung 4.3 zu sehen.

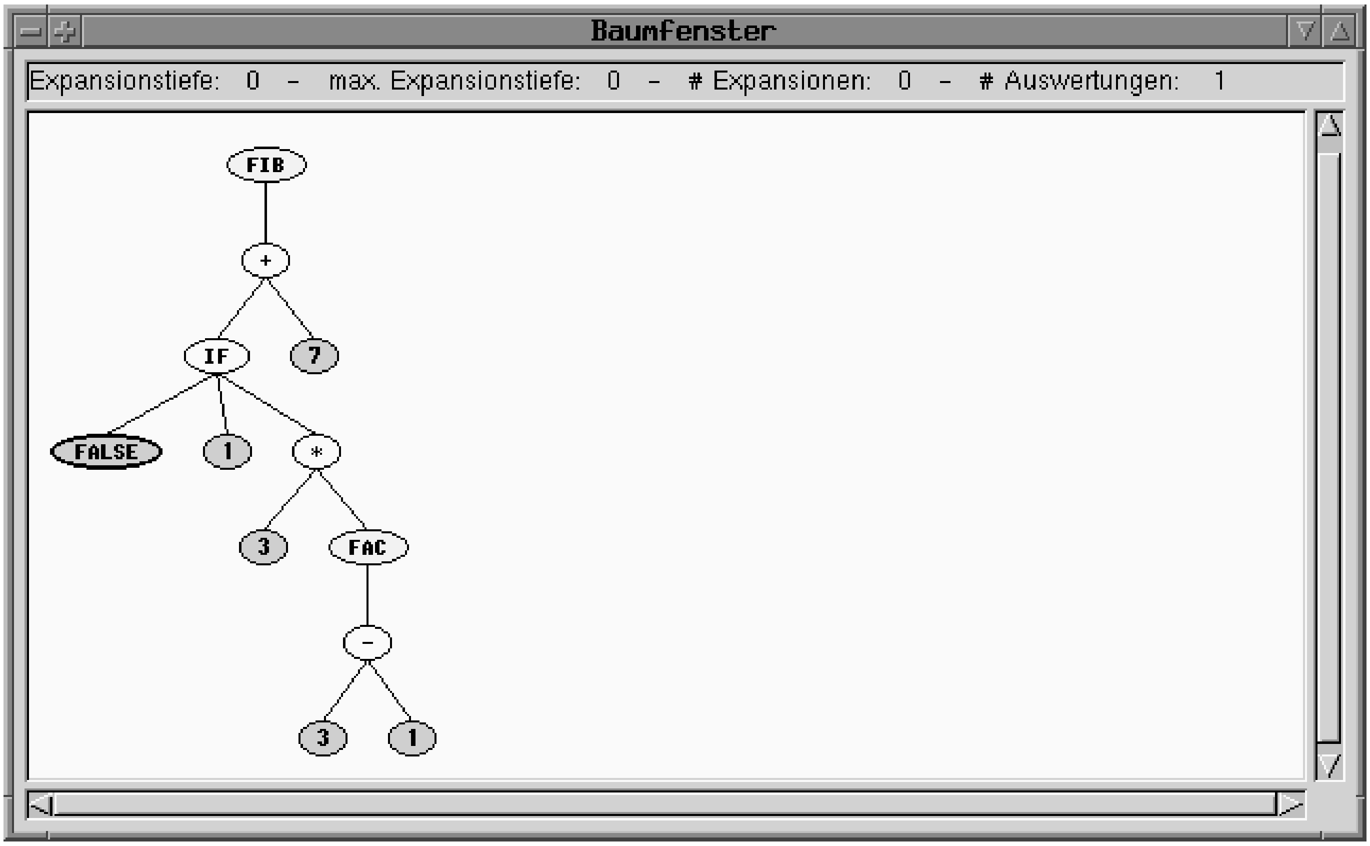


Abbildung 4.3: Das Baumfenster nach Auswertung eines Gleichheitstests

Wie oben erörtert besteht nun der erste Unterbaum des Fallunterscheidungsknotens aus einem ausgewerteten Knoten, der für den booleschen Wert *ff* steht. In dem restlichen Baum hat es keine weiteren Änderungen gegeben.

Hier hat zum ersten Mal in diesem Beispiel eine Auswertung stattgefunden. Bei allen Auswertungen werden in dem Statusbalken, der sich ganz oben im Baumfenster befindet, Informationen über die Auswertung angezeigt. Wie in 3.3.6 beschrieben werden im Statusbalken

- die aktuelle Expansionstiefe,
- die bisher maximale Expansionstiefe,
- die Anzahl der bisherigen Expansionen und
- die Anzahl der bisherigen eigentlichen Auswertungen

zu jedem Zeitpunkt einer Auswertung dargestellt.

Die ersten beiden Informationen beziehen sich auf die Expansionstiefe. Diese Angaben sind in der Regel nur dann von Interesse, wenn bei der Auswertung der Aufruf einer rekursiven Rechenvorschrift vorkommt. Da in funktionalen Programmiersprachen, also insbesondere in ML und somit in KIEL, Funktionen sehr häufig durch rekursive Rechenvorschriften dargestellt werden, dienen die ersten beiden Komponenten des Statusbalkens dem Benutzer nicht selten dazu, festzustellen, ob die Auswertung terminiert.

Die anderen beiden Komponenten des Statusbalkens geben Auskunft über die Anzahl verschiedener Bearbeitungsschritte. Es wird die Anzahl der Expansionen und die Anzahl der elementaren Auswertungen angegeben. Lediglich die Anzahl der erfolgten Vereinfachungen einer Fallunterscheidung wird aus Platzgründen nicht angezeigt.

Die soeben vorgenommene Auswertung ist von der denkbar einfachsten Art. Es wurde lediglich eine Basisoperation ausgeführt, und beide Unterbäume des auszuwertenden Knotens waren bereits ausgewertet. Da in diesem Teilausdruck demnach keine Rechenvorschriftsaufrufe standen, sind die ersten drei Informationen im Statusbalken unverändert auf dem Wert Null. Es wurde jedoch eine Auswertung, nämlich der Test auf Gleichheit, ausgeführt. Diese Auswertung wird in der vierten Komponente im Statusbalken protokolliert. Wie im weiteren Verlauf dieses Beispiels noch zu sehen sein wird, liefert der Statusbalken bei komplizierteren Auswertungen wesentlich interessantere Informationen. Diese werden dann während der Auswertung aktualisiert und geben Aufschluß über den Fortschritt der Auswertung.

An dieser Stelle ist klar zu erkennen, was bei der Vereinfachung des Fallunterscheidungsknotens geschehen wird. Aufgrund des ausgewerteten ersten Unterbaumes, der aus einem Knoten mit dem Inhalt `FALSE` besteht, muß die Vereinfachung den Teilbaum, dessen Wurzel der Fallunterscheidungsknoten bildet, durch den dritten Unterbaum dieses Knotens ersetzen.

Jetzt soll diese Vereinfachung ausgeführt werden. Dazu ist mit der **rechten** Maustaste auf den Fallunterscheidungsknoten zu klicken. Die Abblidung 4.4 stellt die hieraus resultierende Anzeige im Baumfenster dar.

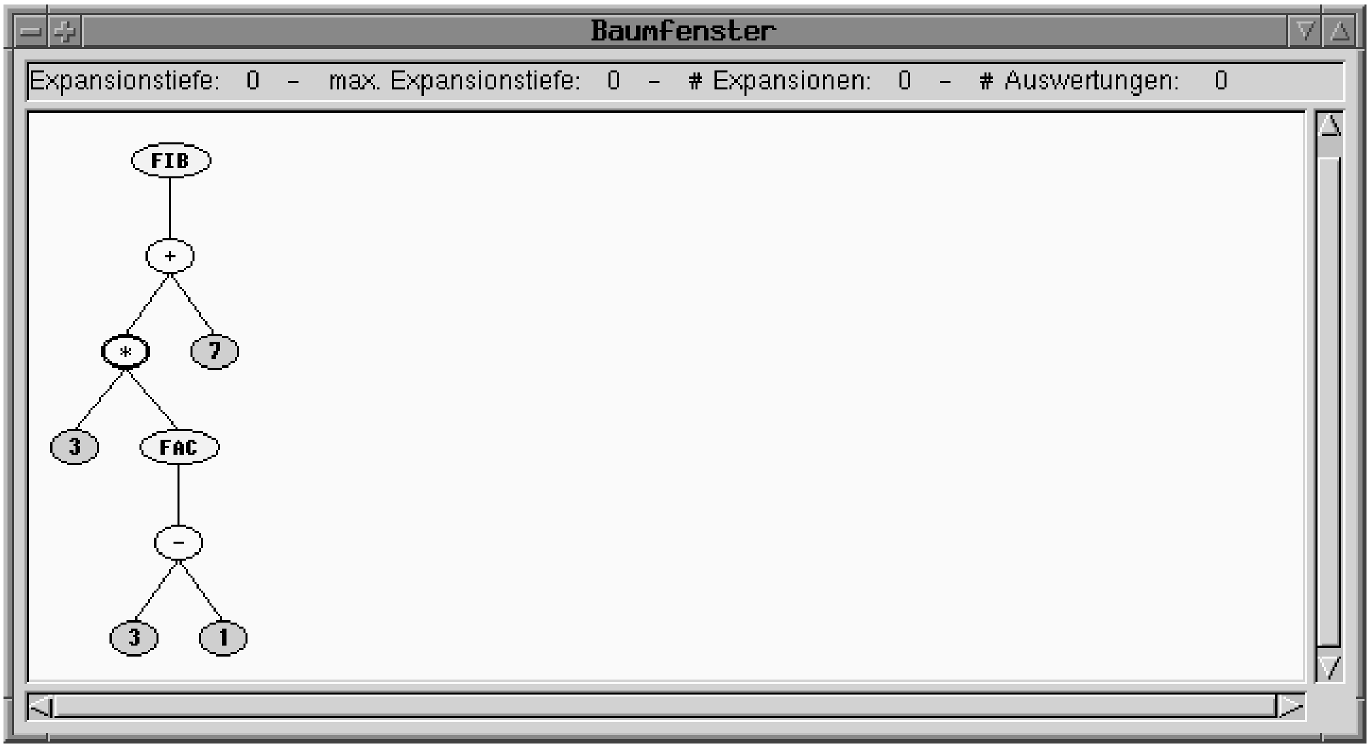


Abbildung 4.4: Das Baumfenster nach Vereinfachung einer Fallunterscheidung

Nach dieser Vereinfachung ist ein Baum entstanden, der dem Ausdruck

```
FIB(3*FAC(3-1)+7)
```

entspricht.

Zu Beginn dieses Beispiels wurde bereits ein Rechenvorschriftsaufruf expandiert. Die Tatsache, daß der entsprechende Knoten keine unausgewerteten Kinder besaß, hat zu einem recht übersichtlichen Ergebnis geführt. Daß dies im allgemeinen nicht immer so ist, wird im folgenden verdeutlicht. Dazu soll der Aufruf der Fibonacci-Funktion expandiert werden. Das Kind des entsprechenden Knotens stellt das Argument der Funktion dar. In diesem Fall handelt es sich nicht um ein Blatt. Es ist also noch nicht ausgewertet. Dies führt dazu, daß bei der Expansion jedes Vorkommen des Parameters in einer Kopie des Baumes, der die Rechenvorschrift darstellt, durch den gesamten Teilbaum des Ausdrucks ersetzt wird, dessen Wurzel das Kind des Rechenvorschriftsaufrufs ist.

Die Expansion wird durch Klicken mit der **rechten** Maustaste auf die Wurzel des im Baumfenster dargestellten Baumes bewirkt. Das Ergebnis der Expansion ist in Abbildung 4.5 zu sehen.

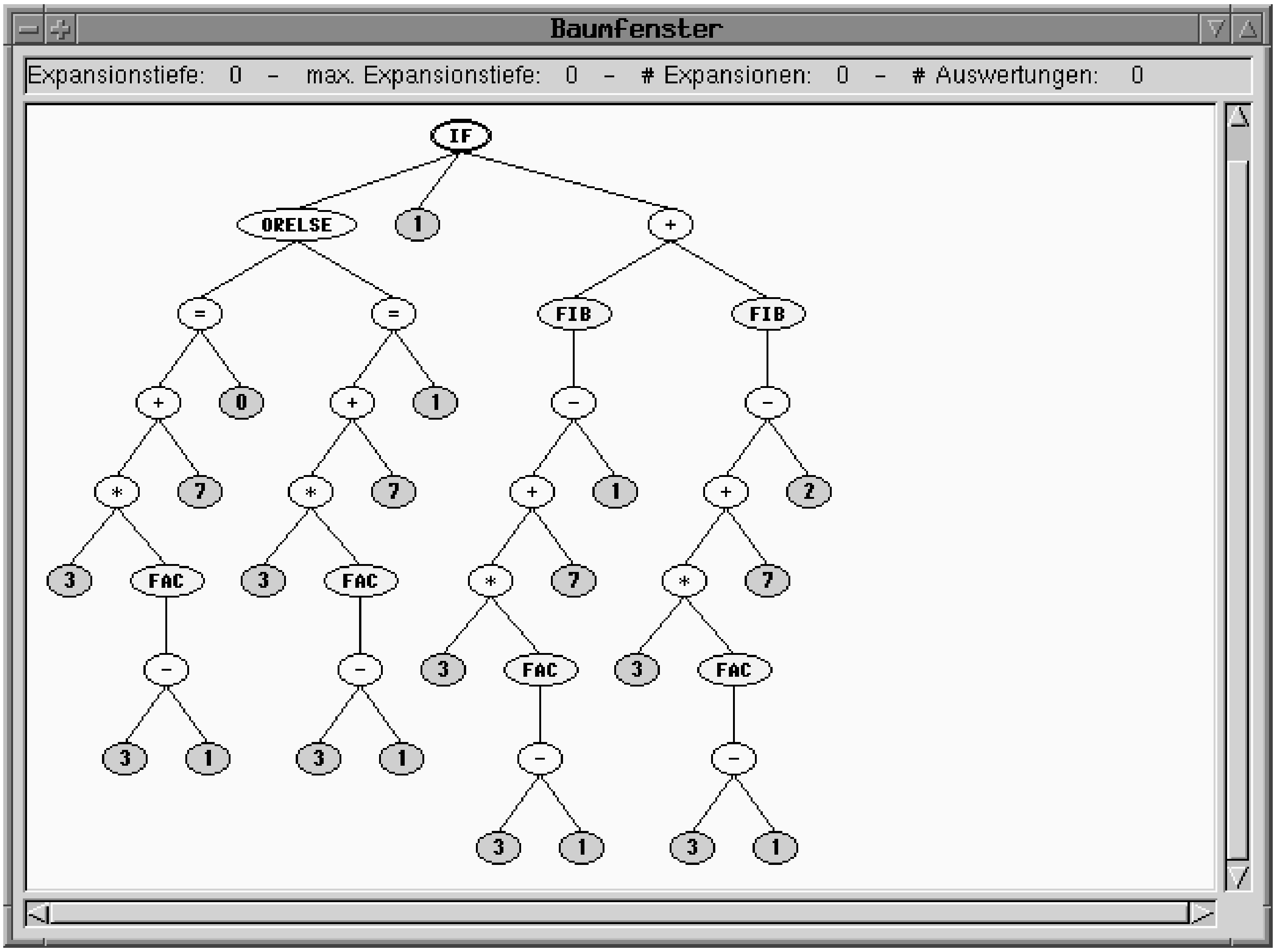


Abbildung 4.5: Das Baumfenster nach Expansion des Aufrufs der Fibonacci-Funktion

In diesem Baum sind die vier identischen Teilbäume, die zu dem Ausdruck `3*FAC(3-1)+7` korrespondieren, deutlich zu erkennen.

Die Expansion des einen Aufrufs der Funktion $\mathcal{FIB}$ hat bewirkt, daß zwei neue Aufrufe derselben Funktion im Baum auftauchen. Entsprechend der Definition von $\mathcal{FIB}$ ist das Argument um den Wert Eins bzw. um den Wert Zwei verringert. Da jedoch bei Expansionen keine Auswertungen vorgenommen werden, sind die Verringerungen durch entsprechendes Einfügen von Subtraktionsknoten und Wertknoten erreicht worden.

Im folgenden Schritt soll der Wert des Arguments des ersten Aufrufs[2] der Fibonacci-Funktion bestimmt werden. Der auszuwertende Knoten ist somit der Subtraktionsknoten, der direkt unter dem ersten Rechen-

[2]Erster Aufruf ist in diesem Zusammenhang derjenige Aufruf, der bei einem Durchlaufen des Baumes nach der Tiefensuche-Strategie zuerst gefunden wird.

vorschriftsknoten mit der Beschriftung FIB gezeichnet ist. Das Klicken mit der **linken** Maustaste auf diesen Knoten führt zu dem in Abbildung 4.6 dargestellten Baum.

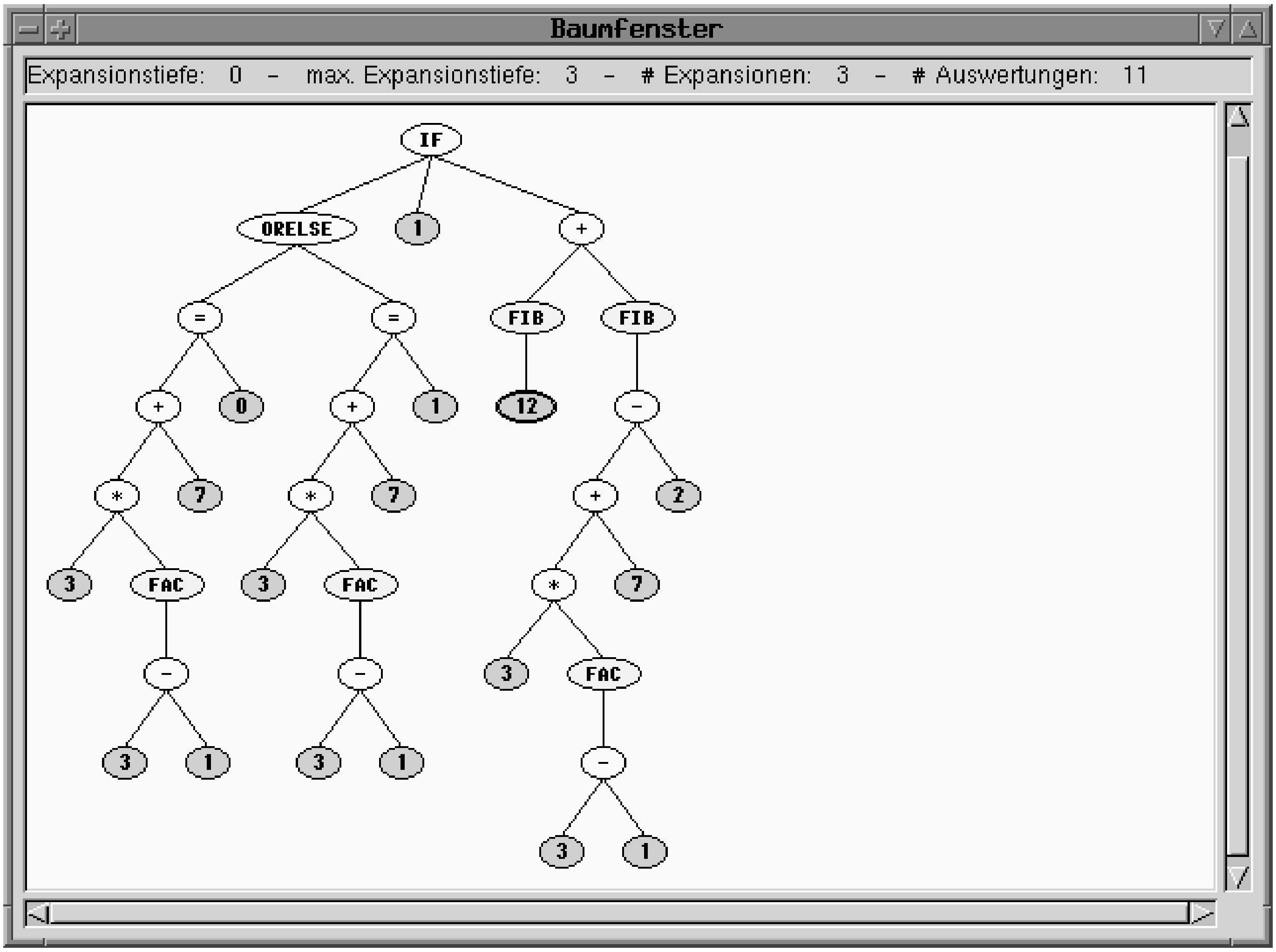


Abbildung 4.6: Das Baumfenster nach Auswertung eines Subtraktionsknotens

Wie gewünscht ist nun das Argument des ersten Aufrufs der Fibonacci-Funktion ausgewertet. An die Position des angeklickten Subtraktionsknotens ist ein ausgewerteter Knoten getreten, der die Beschriftung 12 trägt. Während der eben erfolgten Auswertung wurden mehrere Bearbeitungsschritte durchgeführt. Das Auftreten eines Aufrufs der Fakultätsfunktion in dem nun ausgewerteten Teilbaum hat zu drei Expansionen geführt. Dieser Sachverhalt kann der dritten Komponente des Statusbalkens entnommen werden. Aus der vierten Komponente ist abzulesen, daß ingesamt elf Auswertungen elementarer Funktionen benötigt wurden.

An dieser Stelle ist anzumerken, daß in diesem Beispiel alle Arbeitsschritte genau beschrieben wurden. Es werden zum Beispiel die Einstellungen im Auswertungsfenster nicht geändert. Das bedingt, daß während des gesamten Beispiels Auswertungen stets nach der Leftmost-Innermost-Strategie stattfinden. Diese Tatsache hat zum ersten Mal im gerade ausgeführten Schritt eine große Bedeutung. Wäre anstelle der Leftmost-Innermost-Auswertung eine Leftmost-Outermost- oder eine Full-Auswertung gestartet worden, so wäre die Auswertung anders verlaufen und hätte mehr Auswertungsschritte benötigt.

Die erste Vereinfachung einer Fallunterscheidung in diesem Beispiel wurde in zwei Schritten ausgeführt. Zuerst wurde der boolesche Ausdruck ausgewertet und anschließend die eigentliche Vereinfachung durchgeführt. Daß die Auswertung des booleschen Ausdruck auch automatisch bei der Vereinfachung einer Fallunterscheidung geschehen kann, wird im nächsten Schritt des Beispiels gezeigt.

Der einzige Fallunterscheidungsknoten im aktuellen Baum ist die Wurzel. Der erste Unterbaum, der bekanntlich den booleschen Ausdruck repräsentiert, ist in diesem Fall noch nicht ausgewertet, sondern nimmt mit seinen 21 Knoten einen Großteil des Auswertungsbaums ein. Um die Fallunterscheidung direkt zu vereinfachen, muß die Wurzel des dargestellten Baumes mit der **rechten** Maustaste angeklickt werden. Daß der Baum durch diese Aktion wirklich vereinfacht wurde, dokumentiert das in Abbildung 4.7 dargestellte Ergebnis dieses Arbeitsschrittes. Der entstandene Auswertungsbaum besteht genau aus dem `else`-Zweig der vereinfachten Fallunterscheidung, woraus folgt, daß die Auswertung des booleschen Ausdrucks zu dem booleschen Wert *ff* geführt hat.

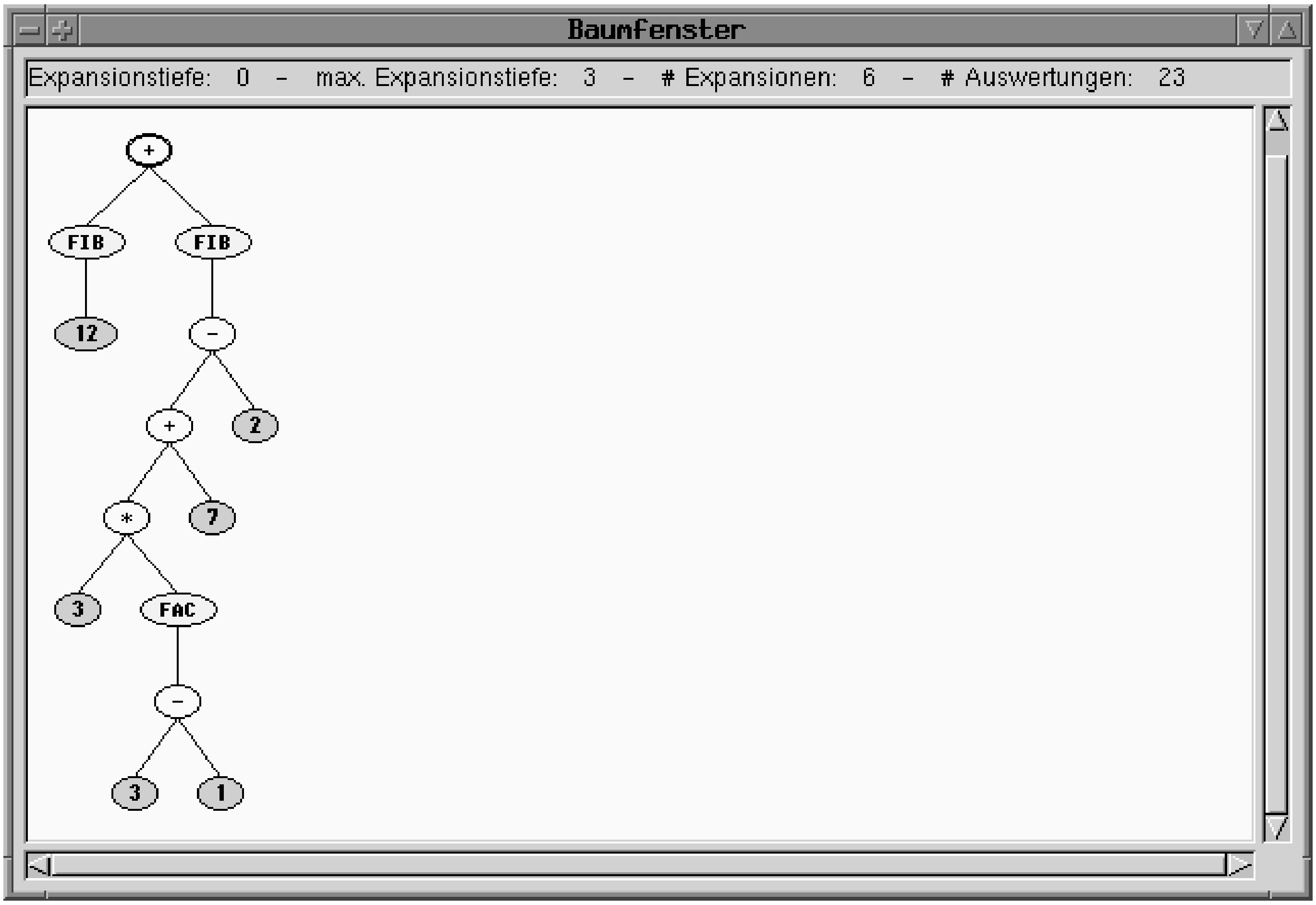


Abbildung 4.7: Das Baumfenster nach direkter Vereinfachung einer Fallunterscheidung

Die Auswertung des booleschen Ausdrucks war etwa doppelt so aufwendig wie die zuvor ausgeführte Auswertung des Arguments des ersten Aufrufs der Fibonacci-Funktion. Die dabei vollzogenen Auswertungsschritte sind fast[3] mit denen identisch, die beim Auswerten des booleschen Ausdrucks für jedes der beiden Argumente der logischen Oder-Verknüpfung erforderlich waren. Mit Kenntnis dieser Tatsache sind nun die Informationen in dem Statusbalken im Vergleich zu den Angaben nach dem vorletzten Auswertungsschritt leicht nachvollziehbar.

Bis zu dieser Stelle wurden in diesem Beispiel noch keine Rechenvorschriftsaufrufe explizit ausgewertet. Dies soll nun geschehen. Dazu soll der zweite Rechenvorschriftsaufruf im aktuellen Baum komplett ausgewertet werden. Wie alle bisher vorgenommenen Auswertungen wird auch diese durch Klicken der **linken** Maustaste gestartet. Im Gegensatz zu den vorigen Auswertungen verstreicht nun jedoch eine kurze Zeit, bevor das Ergebnis angezeigt wird. Das Bearbeiten der Auswertung kann über die Informationen des Statusbalkens

[3] Der einzige Unterschied besteht darin, daß an die Stelle der Subtraktion ein logischer Vergleich getreten ist.

verfolgt werden. Der resultierende Baum ist in Abbildung 4.8 zu sehen.

Abbildung 4.8: Das Baumfenster nach einer aufwendigeren Auswertung

Im Vergleich zu dem vorigen Baum ist der nun vorliegende besser überschaubar. Um dies zu erreichen, mußte von KIEL einige Arbeit geleistet werden. Die soeben vollzogene Auswertung hat als Ergebnis einen Baum geliefert, der dem Ausdruck

`FIB(12)+144`

entspricht.

Anders als bei allen anderen Inhalten des Statusbalkens ist die Expansionstiefe nach der Auswertung nicht null, sondern eins. Dies spiegelt die Tatsache wider, daß ein Rechenvorschriftsaufruf direkt ausgewertet wurde. Es wurden keine Auswertungsschritte vorgenommen, die mit dem Ergebnis der ersten Expansion arbeiten. Da der Ergebnisknoten, der die Beschriftung `144` trägt, an die Stelle des Rechenvorschriftsknotens getreten ist, liegt er in einer logischen Ebene mit diesem, also in einer Expansionstiefe von eins.

Aus der Information, daß 290 Expansionen durchgeführt wurden, die maximale Expansionstiefe jedoch nur elf beträgt, läßt sich erkennen, daß bei Vereinfachungen von Fallunterscheidungen, die aus der Expansion eines Aufrufs der Fibonacci-Funktion entstanden, recht häufig der `else`-Zweig gewählt wurde, der seinerseits zwei neue Rechenvorschriftsaufrufe von $\mathcal{FIB}$ beinhaltet. Dies hat mit der Art und Weise zu tun, wie die Fibonacci-Funktion berechnet werden soll. Um $\mathcal{FIB}(n)$ zu berechnen, wird das Problem auf die Berechnung von $\mathcal{FIB}(n-1)$ und $\mathcal{FIB}(n-2)$ zurückgeführt. Solche Ansätze werden als Teile-und-Herrsche-Algorithmen bezeichnet. In diesem Beispiel führt der Ansatz jedoch zu einer exponentiellen Laufzeit des Algorithmus. Ein algorithmischer Ansatz, der bezüglich der Laufzeit bei der Fibonacci-Funktion besser ist, ist die Methode der Dynamischen Programmierung, mit der man lineare Laufzeit erreicht. Dabei wird die Strategie verfolgt, zuerst Ergebnisse für kleine Argumente zu finden und mit deren Hilfe Ergebnisse für größere Argumente zu bestimmen.

Die im Vergleich zu den vorigen Bearbeitungsschritten dieses Beispiels lange Auswertungszeit spiegelt sich auch in der vierten Komponente des Statusbalkens wider. Es wurden während der Auswertung des Rechenvorschriftsaufrufs insgesamt 1246 elementare Auswertungen vorgenommen.

Um dieses Beispiel abzuschließen, soll nun das Endergebnis der Auswertung des ursprünglichen Ausdrucks bestimmt werden. Die Hauptarbeit hierbei ist die Auswertung des Teilbaumes, der dem Ausdruck `FIB(12)` entspricht. Die Auswertung, die im vorigen Schritt vorgenommen wurde, entsprach im wesentlichen der Wertbestimmung des Ausdrucks `FIB(11)`. Da die Auswertung des eben angesprochenen Ausdrucks `FIB(12)` fast genau so aufwendig ist wie die Auswertung der Ausdrücke `FIB(11)` und `FIB(10)` zusammen, ist zu erwarten, daß im Vergleich zu der vorigen Auswertung die Anzahl der Auswertungsschritte um einen Faktor zunimmt, der etwas größer als 1.5 ist.[4] Um diese Überlegung zu testen, soll nun die Auswertung durchgeführt werden. Hierzu ist die Wurzel des aktuellen Baumes, ein Additionsknoten, mit der **linken** Maustaste anzuklicken. Das in Abbildung 4.9 dargestellte Ergebnis ist ein Baum, der aus nur einem Knoten besteht.

Abbildung 4.9: Das Baumfenster nach abschießender Auswertung

Wie erwartet wurden etwa 1.6 mal so viele Expansionen und elementare Auswertungen vorgenommen wie in der zuvor erfolgten Auswertung.

Nun liegt das Endergebnis der Auswertung des Ausdrucks

```
FIB(FAC(3)+7)
```

vor. Es wird durch einen Baum dargestellt, der aus einem Knoten besteht. Dieser Knoten trägt die Beschriftung

```
377.
```

Der Wert des bearbeiteten Ausdrucks bezüglich der gewählten Semantik ist somit 377. Die Semantik wurde durch die angewendeten Bearbeitungsschritte bestimmt. Offensichtlich wurde keine der in 2.3.3 beschriebenen Ersetzungsarten konsequent angewendet. Es ist durchaus möglich, eine Folge von Bearbeitungsschritten durchzuführen, die nicht zu dem Wert 377 führt. Man könnte beispielsweise immer das erste Auftreten eines Rechenvorschriftsknotens im aktuellen Baum expandieren.

Die gängigen Auswertungsarten, die Leftmost-Innermost- und die Leftmost-Outermost-Ersetzung, liefern jedoch jeweils das Ergebnis 377. Die Unterschiede machen sich in dem Ablauf der Auswertung bemerkbar. Wird der Ausdruck `FIB(FAC(3)+7)` direkt nach seiner Eingabe durch Klicken mit der **linken** Maustaste auf die Wurzel des entsprechenden Baumes ausgewertet, ist der Rechenaufwand stark von der gewählten Ersetzungsart abhängig, wie die Tabelle 4.1 verdeutlicht.

verwendete Ersetzungsstrategie	benötigte Expansionen	benötigte Auswertungen
Leftmost-Innermost	757	3254
Leftmost-Outermost	6201	36273

Tabelle 4.1: Arbeitsaufwand in Abhängigkeit der Ersetzungsart im ersten Beispiel

Eine Auswertung mit der Full-Ersetzung ist in diesem Beispiel wenig sinnvoll. Die Anzahl der auszuführenden Expansionen pro Auswertungsschritt, der eine Expansion erfordert, wächst sehr schnell. Dies führt zu sehr großen Bäumen, die der Benutzer bei einer Einzelschrittauswertung betrachten kann.

[4] Der Faktor 1.5 würde dann erreicht werden, wenn die Auswertung von `FIB(10)` genau halb so aufwendig wäre wie die von `FIB(11)`. Da die Auswertung von `FIB(11)` im wesentlichen jedoch den Auswertungen von `FIB(10)` und `FIB(9)` entspricht und die Auswertung von `FIB(10)` aufwendiger ist, als die von `FIB(9)`, muß der Faktor etwas größer sein als 1.5.

4.2 Unterschiede der Ersetzungsarten

Nachdem in 4.1 die grundlegenden Funktionen des Baumfensters behandelt wurden, soll in diesem Beispiel die Verwendung des Auswertungsfensters und das Arbeiten mit der mittleren Maustaste auf einem freien Teil der Zeichenfläche im Vordergrund stehen. Außerdem wird gezeigt, wie mit dem Undo-Redo-Fenster in der Bearbeitungsfolge einer Auswertung nach vorne und im Anschluß wieder nach hinten gesprungen werden kann.

Wie am Ende von 4.1 gesehen, kann die Wahl der Ersetzungsart den Arbeitsaufwand zur Berechnung eines Ausdrucks erheblich beeinflussen. Bei einer Auswertung des dort behandelten Ausdrucks mit der Leftmost-Outermost-Strategie müssen mehr als elfmal so viele elementare Auswertungen getätigt werden als bei einer Auswertung mit der Leftmost-Innermost-Strategie.

Die nun behandelte Funktion ist die **Morris-Funktion** $\mathcal{F}$, die durch

$$\begin{array}{rcl} \mathcal{F} \quad : \quad \mathbb{Z} \times \mathbb{Z} & \rightarrow & \mathbb{Z} \\ (x,y) & \longmapsto & \begin{cases} 1 & : \text{ falls } x \leq 0 \\ \mathcal{F}(x-1, \mathcal{F}(x-y,y)) & : \text{ sonst} \end{cases} \end{array}$$

definiert ist.

Die Besonderheit dieser Funktion wurde bereits in 2.3.3 bei der Behandlung der Auswertung durch Vereinfachung und Expansion aufgezeigt. Es gibt Argumente, die bei einer Auswertung mit der Leftmost-Innermost-Strategie ein anderes Ergebnis liefern als bei der Auswertung mit der Leftmost-Outermost-Strategie. In 2.3.3 sind die jeweiligen Auswertungen angegeben. Diese sollen hier exakt nachvollzogen werden. Die Überführung der Morris-Funktion in eine ML-Rechenvorschrift liefert:

```
fun F(x:int,y:int):int=
    if x<=0
        then 1
        else F(x-1,F(x-y,y));
```

Zur Eingabe dieser Rechenvorschrift in KIEL muß sie zunächst in einer Eingabedatei abgelegt werden. Danach ist sie analog zum ersten Beispiel zu laden. Es ist nicht notwendig, KIEL zu beenden und neu zu starten. Das Laden einer Rechenvorschriftendatei bewirkt, daß alle bis dahin eingelesenen Rechenvorschriften und Ausdrücke aus dem Speicher gelöscht werden. In bezug auf diese Informationen befindet sich KIEL also in dem gleichen Zustand wie nach einem Neustart.[5]

Der auszuwertende Ausdruck lautet

`F(1,0).`

Also ist im Ausdruckeingabefenster

`F(1,0);`

einzugeben. Alternativ zu der Möglichkeit, alle Zeichen über die Tastatur einzugeben, kann zur Eingabe auch teilweise das Rechenvorschrifts-Übersichtsfenster (siehe 3.3.5) benutzt werden.[6] Hierzu wird mit der linken Maustaste die Rechenvorschrift `F` im Rechenvorschrifts-Übersichtsfenster markiert. Anschließend wird die rechte Maustaste über einer beliebigen Rechenvorschrift im Rechenvorschrifts-Übersichtsfenster gedrückt. Diese Aktionen haben bewirkt, daß im Ausdruckeingabefenster der Text `F()` steht. Zusätzlich wurde das Ausdruckeingabefenster geöffnet, falls dies nicht bereits vorher der Fall war. Die Argumente und das abschließende Semikolon müssen noch über die Tastatur eingegeben werden, um den Ausdruck zu vervollständigen.

[5] Natürlich können beispielsweise Fenster umplaziert worden oder Einstellungen im Undo-Redo- oder im Auswertungsfenster vorgenommen worden sein. Diese Einstellungen bleiben nach dem Laden unverändert.

[6] Für Ausdrücke, die wie in diesem Beispiel sehr kurz sind, wird die beschriebene Methode in der Regel nicht genutzt, da weder die Zeitersparnis sehr groß ist, noch das Risiko, schließende Klammern zu vergessen, erheblich vermindert wird. Dennoch kann hier der Mechanismus, der bei komplizierteren Ausdrücken einige Arbeit ersparen kann, gut nachvollzogen werden.

Die Auswahl dieses Ausdrucks im Ausdruckübersichtsfenster bewirkt die Anzeige des korrespondierenden Baumes im Baumfenster.

Die Auswertung soll durch Vereinfachung und Expansion erfolgen. Also muß die Anzahl der auszuführenden Auswertungsschritte auf Eins gesetzt werden. Hierzu wird das Auswertungsfenster durch Klicken auf den Button mit der Aufschrift **`Auswertung`** im Fensterrahmen des Befehlsfensters (siehe 3.3.1) geöffnet und im Rahmen **`Anzahl der Auswertungen`** der erste Punkt ausgewählt.

Aus Platzgründen wurden in diesem Beispiel das Auswertungsfenster und das Undo-Redo-Fenster gegebenenfalls über einen im jeweiligen Fall nicht aussagekräftigen Teil des Baumfensters geschoben. In der Abbildung 4.10 sind das Baum- und das Auswertungsfenster nach Ausführung der soeben beschriebenen Aktionen dargestellt.

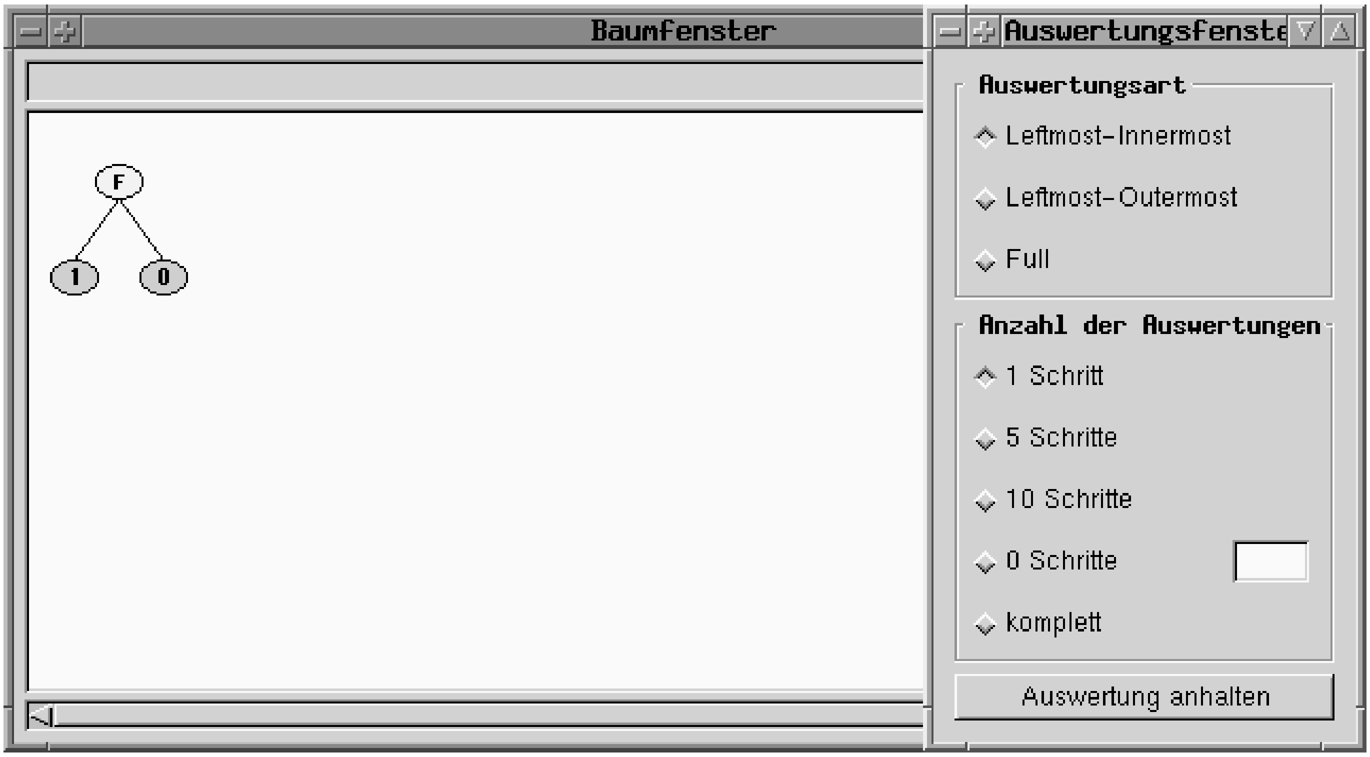


Abbildung 4.10: Das Baum- und Auswertungsfenster nach Eingabe von `F(1,0)`

Im ersten Teil dieses Beispiels soll die Auswertung des Ausdrucks mit der Leftmost-Innermost-Ersetzung erfolgen. Somit muß die Voreinstellung im Auswertungsartrahmen des Auswertungsfensters nicht geändert werden. Da die Auswertung in den Schritten erfolgen soll, die in 2.3.3 angegeben sind, werden in diesem Beispiel zur Auswertung des Ausdrucks abwechselnd die **mittlere** Maustaste auf einem freien Teil der Zeichenfläche betätigt und mit der **linken** Maustaste auf die Wurzel des angezeigten Baumes gedrückt.

In dem Ausdruck

```
F(1,0)
```

tauchen weder ML-Basisoperationen noch Fallunterscheidungen auf. Er kann somit nicht weiter vereinfacht werden. Das Drücken der **mittleren** Maustaste über einer freien Stelle der Zeichenfläche läßt den Ausdruck unverändert. Da keine Veränderung des Baumes im Baumfenster stattgefunden hat, wurde diese Aktion vom Undo-Redo-Fenster nicht protokolliert. Ansonsten würde häufiges, aufeinanderfolgendes Drücken der mittleren Maustaste mit dem Ziel der Vereinfachung des Ausdrucks das Protokoll wachsen lassen. Damit würde die Orientierung innerhalb dieses Protokolls erschwert werden.

Um den folgenden Expansionsschritt auszuführen, ist mit der **linken** Maustaste auf die Wurzel des Baumes zu klicken. Es wird genau ein Expansionsschritt vorgenommen. Da es nur einen Rechenvorschriftsknoten im aktuellen Baum gibt, hat die Einstellung der Auswertungsart im Auswertungsfenster an dieser Stelle keine Bedeutung. Es wird der einzige Rechenvorschriftsknoten expandiert. Das Ergebnis ist in Abbildung 4.11 zu sehen.

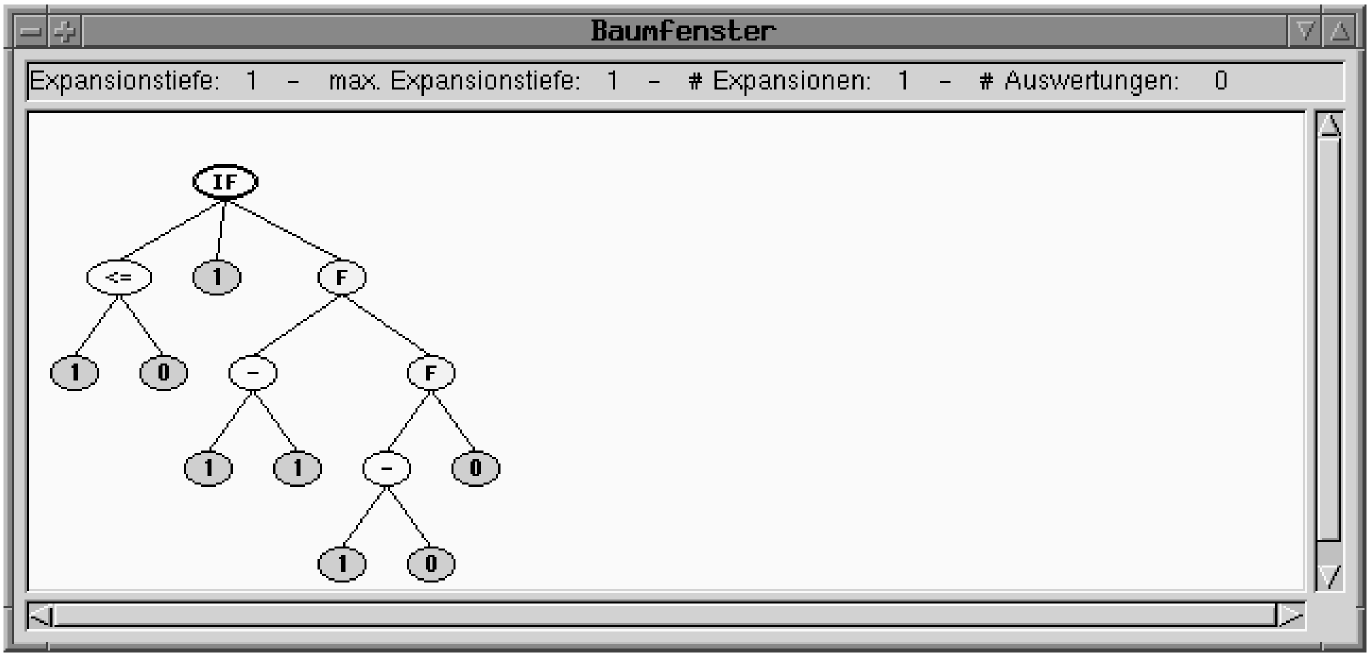


Abbildung 4.11: Das Baumfenster nach der Expansion des einzigen Rechenvorschriftsaufrufs

Auf den Expansionsschritt folgt ein Vereinfachungsschritt. Nun kann an einigen Stellen im Ausdruck eine Vereinfachung vorgenommen werden. Neben den beiden Subtraktionsknoten wird auch der Knoten ausgewertet, der dem Test entspricht, ob Eins kleiner oder gleich Null ist. Schließlich wird der Fallunterscheidungsknoten vereinfacht.

All diese Aktionen werden ausgeführt, nachdem die **mittlere** Maustaste über einem freien Punkt der Zeichenfläche gedrückt wurde. Das Baumfenster zeigt, wie die Abbildung 4.12 verdeutlicht, einen Baum an, der so weit wie möglich vereinfacht wurde.

Abbildung 4.12: Das Baumfenster nach dem ersten Vereinfachungsschritt

Der Auswertungsbaum entspricht dem Ausdruck

```
F(0,F(1,0)).
```

Nach der soeben erfolgten Vereinfachung findet im nächsten Schritt wieder eine Expansion statt. Zu diesem Zeitpunkt gibt es zwei Rechenvorschriftsknoten im aktuellen Baum. Somit bestimmt an dieser Stelle die Einstellung der Auswertungsart im Auswertungsfenster den Ort der Expansion. Da die Leftmost-Innermost-Strategie ausgewählt ist, wird der innere Aufruf der Morris-Funktion expandiert. Die Expansion wird nach dem Anklicken der Wurzel mit der **linken** Maustaste vorgenommen. Der entstehende Baum ist in 4.13 dargestellt.

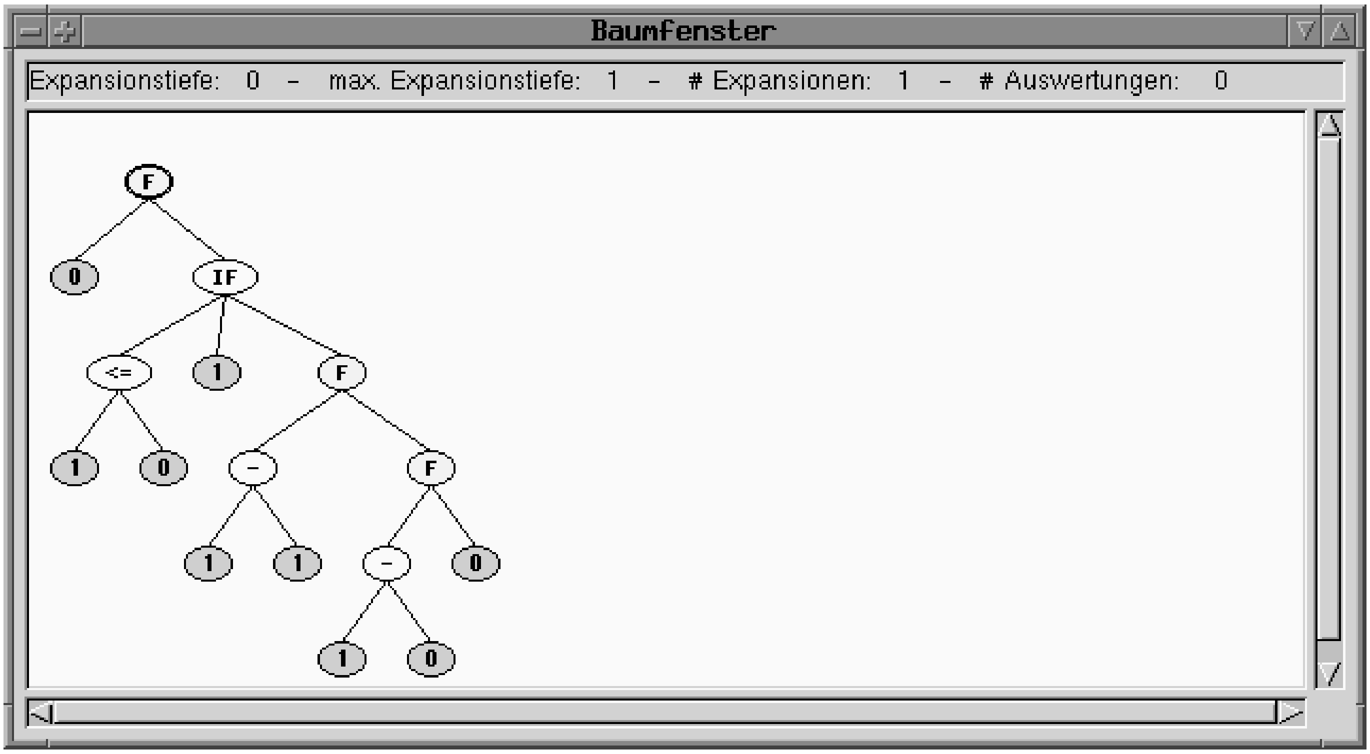


Abbildung 4.13: Das Baumfenster nach der Expansion des inneren Rechenvorschriftsaufrufs

An dieser Stelle ist ein Vergleich der Abbildung 4.13 mit der Abbildung 4.11 interessant. Der in Abbildung 4.13 dargestellte Baum entstand aus dem Baum von Abbildung 4.11 durch Anwenden eines Vereinfachungs- und eines Expansionsschrittes. Der gesamte Baum aus Abbildung 4.11 ist als Teilbaum in Abbildung 4.13 wiederzufinden. Er entspricht dem Ausdruck

```
if 1<=0 then 1 else F(1-1,F(1-0,0))
```

und taucht als zweites Argument des äußersten Aufrufs der Morris-Funktion in dem Baum aus Abbildung 4.13 wieder auf. Da momentan eine Leftmost-Innermost-Ersetzung durchgeführt wird, wird im folgenden zunächst mit den Knoten gearbeitet werden, die durch die soeben ausgeführte Expansion zum Ausdrucksbaum hinzugekommen sind. Es ist also zu erwarten, daß der Auswertungsbaum im Laufe der folgenden Vereinfachungen und Expansionen stetig wachsen wird. Diese Beobachtung zeigt, wie wichtig die ausgewählte Ersetzungsart für den Verlauf einer Auswertung sein kann.

Die anstehenden Vereinfachungen sollen im folgenden Schritt vollzogen werden. Wie bei dem ersten eigentlichen Vereinfachungsschritt gibt es auch hier wieder drei ML-Basisoperationen zum Auswerten und eine Fallunterscheidung, die vereinfacht werden kann, weil der Wert ihres booleschen Ausdrucks nach der Auswertung des Tests, ob Eins kleiner oder gleich Null ist, bekannt wird.

Es wird nun mit der **mittleren** Maustaste auf einen freien Punkt der Zeichenfläche geklickt, um die Vereinfachung auszuführen. In dem entstehenden Baum, der in Abbildung 4.14 dargestellt ist, sind drei Rechenvorschriftsknoten zu sehen.

Abbildung 4.14: Das Baumfenster nach Vereinfachung des Ausdrucks

Es werden nun ein Expansions- und ein Vereinfachungsschritt nacheinander ausgeführt. Anschließend wird das Undo-Redo-Fenster durch Drücken des Buttons mit der Beschriftung `Undo` aus dem Fenster-anzeigen-Rahmen des Befehlsfensters geöffnet. Der entstandene Auswertungsbaum im Baumfenster und das Undo-Redo-Fenster sind in Abbildung 4.15 zu sehen.

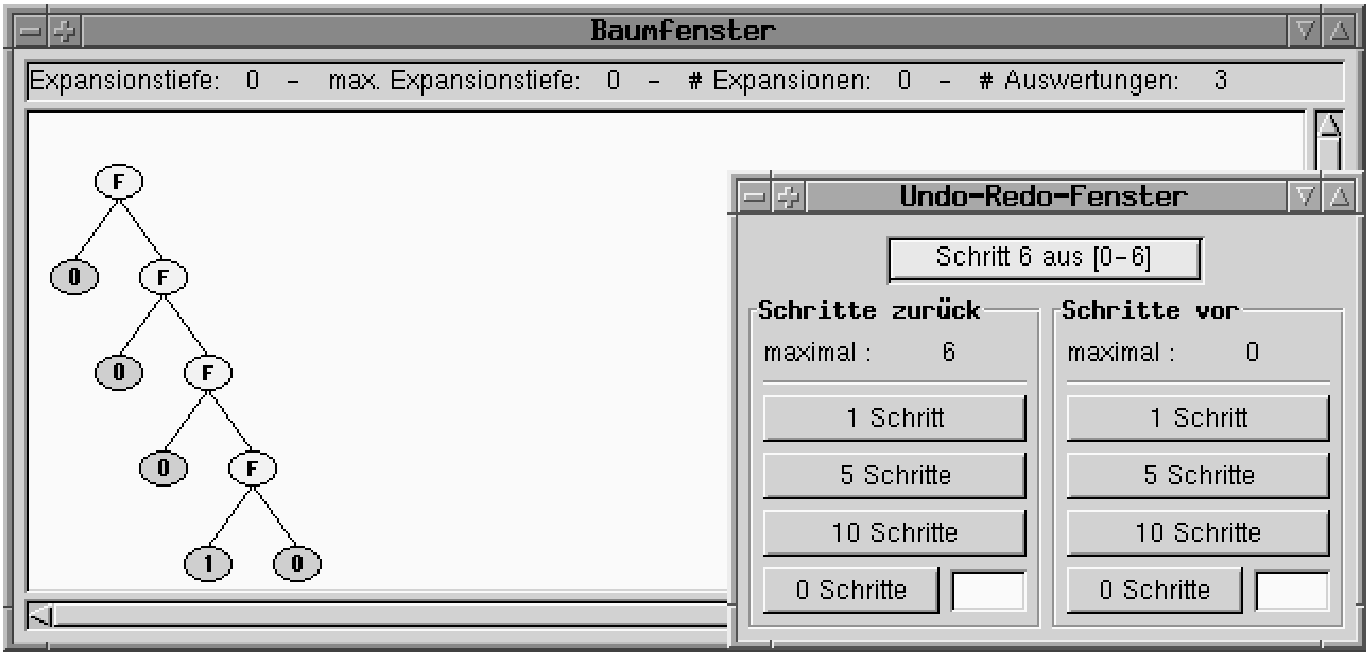


Abbildung 4.15: Das Baum- und das Undo-Redo-Fenster nach sechs Auswertungsschritten

Der im Baumfenster dargestellt Baum repräsentiert den Ausdruck

```
F(0,F(0,F(0,F(1,0)))).
```

Dieses Ergebnis zeigt, daß jeweils ein Expansions- und ein Vereinfachungssschritt zu einem Hinzufügen eines Aufrufs der Morris-Funktion im zu bearbeitenden Ausdruck führen. An dieser Stelle ist ein Punkt erreicht, der offenbart, daß die Auswertung des Ausdrucks nach Leftmost-Innermost-Strategie nicht terminiert und somit kein definiertes Endergebnis liefert.

Es soll nun derselbe Ausdruck mit der Leftmost-Outermost-Strategie bearbeitet werden. Es gibt zwei Möglichkeiten, die Auswertung erneut zu beginnen.

- Zum einen kann der Ausdruck `F(1,0)` abermals in dem Ausdruckeingabefenster eingegeben werden. Dadurch wird ein neuer Eintrag im Ausdruckübersichtsfenster erstellt. Ein Anklicken dieses Ausdrucks läßt einen Baum im Baumfenster erscheinen, der zu dem Baum, mit dem die Bearbeitung am Anfang dieses Beispiels begonnen wurde, identisch ist. Diese Vorgehensweise wird hier nicht verfolgt; es wird stattdessen die folgende Alternative gewählt.

- Im ersten Teil des Beispiels, der sich mit der Auswertung nach der Leftmost-Innermost-Strategie befaßte, wurden insgesamt sechs Auswertungsschritte vorgenommen. Daraus resultieren sieben Bäume, die von KIEL protokolliert wurden. Das Undo-Redo-Fenster dient zum Bewegen innerhalb dieses Protokolls. Mit Hilfe der Funktionen, die das Undo-Redo-Fenster zur Verfügung stellt, kann an den Anfang der Auswertung zurückgegangen werden.

Die Wahl der zweiten Alternative ermöglicht es, Auswertungsschritte aus der bisherigen Auswertung, die bei der Auswertung nach der Leftmost-Innermost- und Leftmost-Outermost-Strategie identisch sind, zu übernehmen. Sie müssen nicht abermals berechnet werden. Dieser Sachverhalt fällt in diesem Beispiel nicht sonderlich ins Gewicht, da die Expansionen und Vereinfachung recht schnell ausgeführt werden; bei komplexeren Ausdrücken kann diese Vorgehensweise jedoch eine erhebliche Zeitersparnis zur Folge haben.

Um zu dem ursprünglichen Ausdruck zurückzukehren, müssen alle sechs protokollierten Schritte rückgängig gemacht werden. In dem Schritte-zurück-Rahmen des Undo-Redo-Fensters ist daher im Eingabefeld die Zahl Sechs einzutragen. Ein Klick auf den nebenstehenden Button führt zur Anzeige des Ursprungsbaumes im Baumfenster. Die Inhalte des Baumfensters und des Undo-Redo-Fensters nach dieser Aktion sind in Abbildung 4.16 zu sehen.

Wie aus den in 2.3.3 aufgeführten Zwischenergebnissen der beiden in diesem Beispiel behandelten Auswertungen zu erkennen ist, sind neben dem ersten Vereinfachungsschritt, der, wie in diesem Beispiel bereits beschrieben, nicht protokolliert wurde, der erste Expansionsschritt und der zweite Vereinfachungsschritt nach der Leftmost-Innermost- und der Leftmost-Outermost-Strategie identisch.

Die ersten beide Auswertungen, die zu diesem Zeitpunkt im Auswertungsprotokoll von KIEL festgehalten sind, können somit direkt übernommen werden. Zur Wiederherstellung von zurückgenommenen Auswertungen dient der Schritte-vor-Rahmen des Undo-Redo-Fensters. Die Eingabe der Zahl Zwei im dort befindlichen Eingabefeld und ein anschließendes Drücken des nebenstehenden Buttons stellt im Baumfenster den gewünschten Zustand her, der bereits im ersten Teil dieses Beispiels zu sehen war. In Abbildung 4.17 sind das aktuelle Baum- und Undo-Redo-Fenster zu sehen.

Spätestens zu diesem Zeitpunkt muß die Auswertungsart in KIEL auf Leftmost-Outermost gesetzt werden. Hierzu ist die entsprechende Auswahl im Auswertungsartrahmen des Auswertungsfensters zu treffen. Da im folgenden weiterhin eine Auswertung durch Vereinfachung und Expansion erfolgt, ist die Auswahl im Anzahl-der-Auswertungen-Rahmen beizubehalten. Das resultierende Auswertungsfenster ist zusammen mit dem Baum, der als nächstes erzeugt wird, in Abbildung 4.18 zu sehen.

Nun wird der nächste Expansionsschritt durch Anklicken der Wurzel des Baumes im Baumfenster mit der **linken** Maustaste ausgeführt. Im Gegensatz zur Expansion des inneren Rechenvorschriftsaufrufs, wie es bei der Leftmost-Innermost-Strategie der Fall war, findet nach der Änderung der Auswertungsart nun eine Expansion des äußeren Rechenvorschriftsaufrufs statt.[7]

[7] An dieser Stelle sei auf die Tabelle 2.1 hingewiesen, in der die zu expandierenden Rechenvorschriftsaufrufe in Abhängigkeit der Ersetzungsart aufgeführt sind.

Der Ausdruck

```
F(0,F(1,0))
```

wird somit zum Ausdruck

```
if 0<=0 then 1 else F(0-1,F(0-F(1,0),F(1,0)))
```

expandiert.

Deutlich zu erkennen ist, daß der `else`-Zweig komplizierter ist, als dies bei der Expansion im ersten Teil dieses Beispiels der Fall war. Dort erschien der Fallunterscheidungsknoten, den die Expansion der Morris-Funktion zwingend nach sich zieht, im Inneren des Baumes. Ein großer Unterschied zu dem Ergebnis der Expansion nach der Leftmost-Innermost-Strategie ist, daß nun der Fallunterscheidungsknoten die Wurzel des Baumes im Baumfenster bildet.

Das Ergebnis der soeben vorgenommenen Expansion ist in Abbildung 4.18 zu sehen. Außerdem ist dort das aktuelle Auswertungsfenster abgebildet.

Das eben erzeugte Zwischenergebnis wird zwar durch einen relativ zu den vorigen Bäumen in diesem Beispiel großen Baum repräsentiert, jedoch fällt auf, daß die Wurzel ein Fallunterscheidungsknoten ist, an dessen Stelle beim folgenden Vereinfachungsschritt der nur aus einem Knoten bestehende `then`-Zweig tritt. Das Ergebnis des Vereinfachungsschrittes ist in Abbildung 4.19 zu sehen.

Der zu diesem Zeitpunkt im Baumfenster angezeigte Baum besteht nur noch aus einem einzigen Knoten. Die Auswertung des Ausdrucks

```
F(1,0)
```

nach der Leftmost-Outermost-Strategie hat somit zu einem definierten Ergebnis geführt.

4.3 Weitere Möglichkeiten in KIEL

In diesem Beispiel wird die Funktionalität der mittleren Maustaste auf Knoten des Auswertungsbaumes behandelt. Mit ihrer Hilfe kann das Erscheinungsbild von

- reellen Knoten,
- Zeichenreihenknoten und
- Sequenzknoten

geändert werden, um es den jeweiligen Bedürfnissen anzupassen.

Um dies zu verdeutlichen, wird der Ausdruck

```
[1,~2,~3,3-4,4,8,7,~6,5,~19,17,15,19,17,111,~4,4,3]@
  (truncate(sin(5.7)/cos(5.7))::
   [size("Alle meine Entchen schwimmen auf dem See"),truncate(13.9 / 0.03) div ~1]
  )
```

bearbeitet. Wie leicht zu erkennen ist, werden keine Rechenvorschriften benötigt; es werden lediglich Basisoperationen verwendet.

Nach der Eingabe dieses Ausdrucks, erzeugt KIEL einen Baum, der aufgrund seiner Größe nicht komplett im Baumfenster angezeigt werden kann. Zunächst wird mit dem rechten unteren Teil des Baumes gearbeitet. Mit Hilfe der Scrollbalken wird dieser Ausschnitt des Baumes, wie in Abbildung 4.20 zu sehen ist, im Baumfenster eingestellt.

In dem in der Mitte des Baumfensters befindlichen Zeichenreihenknoten, durch den die Zeichenreihe

`Alle meine Entchen schwimmen auf dem See`

repräsentiert wird, steht

`"Alle meine Entchen s..."`.

Es wurden insgesamt nur zwanzig Zeichen der erwarteten Zeichenreihe dargestellt. Ihnen folgen drei Punkte, die andeuten, daß die komplette Zeichenreihe länger als zwanzig Zeichen ist und daher verkürzt dargestellt wird.

Die Anzahl der Zeichen, die in diesem Zeichenreihenknoten dargestellt werden, soll jetzt so variiert werden, daß die Zeichenreihe komplett zu sehen ist. Dazu wird mit der **mittleren** Maustaste kurz auf den Zeichenreihenknoten geklickt. Daraufhin erscheint eine Wählscheibe, die momentan auf dem Wert 20 steht. Eine beliebige Maustaste ist nun über dem Zeiger der Wählscheibe zu drücken und gedrückt zu halten. Durch Bewegen der Maus kann nun die Anzahl der anzuzeigenden Zeichen in diesem Zeichenreihenknoten geändert werden. Da die komplette Zeichenreihe 40 Zeichen lang ist, wird nun der Zeiger auf einen beliebigen Wert gedreht, der größer oder gleich 40 ist. Das Loslassen der Maustaste führt zur Übernahme der Auswahl und damit zu einer neuen Darstellung des Baumes im Baumfenster.

Da der Zeichenreihenknoten nun mehr Platz einnimmt, ist der Baum insgesamt breiter geworden. Nachdem mit den Scrollbalken der Zeichenreihenknoten wieder in der Mitte des Baumfensters plaziert wurde, ist das in Abbildung 4.21 dargestellte Bild zu sehen.

Von Bedeutung sind bei der eben erfolgten Änderung der Darstellungsart eines Knotens

- die voreingestellte Anzahl an Stellen, Zeichen bzw. Komponenten und
- die maximale Anzahl an Stellen, Zeichen bzw. Komponenten.

Diese Werte werden beim Start von KIEL entsprechend der Einträge in der Ressourcedatei `kiel.rc` (siehe 3.4.1) gesetzt. In der Tabelle 4.2 sind die standardmäßigen Werte der sechs maßgeblichen Konstanten aufgeführt.

Art der Knoten	voreingestellte Anzahl	maximale Anzahl
reelle Knoten	2	10
Zeichenreihenknoten	20	100
Sequenzknoten	20	100

Tabelle 4.2: Standardmäßige Werte der Konstanten zur Festlegung der Knotendarstellung

Der folgende Teil dieses Beispiels geht auf reelle Knoten ein. Genau rechts neben dem eben betrachteten Zeichenreihenknoten befindet sich im Baumfenster ein `truncate`-Knoten. Sein Argument, dessen Wert das Ergebnis der Berechnung von

`13.9 / 0.03`

ist, steht im Mittelpunkt der nächsten Betrachtungen.

Zunächst wird dieses Argument ausgewertet. Dazu wird es mit der **linken** Maustaste angeklickt. Somit erzeugt KIEL einen Baum, von dem der momentan interessante Ausschnitt im rechten Teil der Abbildung 4.22 zu sehen ist.

Das Ergebnis der soeben ausgeführten Auswertung ist ein Knoten, der den Wert

$$463\frac{1}{3}$$

repräsentiert. Als Dezimalzahl kann dieser nicht exakt im Rechner dargestellt werden. Da die Anzahl der darzustellenden Nachkommastellen auf zwei gesetzt ist (siehe Tabelle 4.2), ist die Beschriftung des Ergebnisknotens

```
4.63e+02.
```

Diese Anzeige soll so geändert werden, daß möglichst viele Nachkommastellen zu sehen sind. Da die maximale Anzahl an Nachkommastellen standardmäßig auf zehn gesetzt ist, können in der Wählscheibe, die durch Drücken der **mittleren** Maustaste aufgerufen werden kann, Werte zwischen Null und Zehn ausgewählt werden.

Wie oben erwähnt, wird nun die maximale Anzahl ausgewählt und dazu der Zeiger der Wählscheibe bis zum Anschlag nach rechts gedreht. Nachdem die dazu verwendete Maustaste losgelassen wurde, ist im Baumfenster das in Abbildung 4.23 dargestellte Bild zu sehen.

Wie erwartet werden nun zehn Nachkommastellen im rellen Knoten angezeigt. Die dargestellte Knotenbeschriftung stellt demnach eine besser Annäherung an den Knotenwert dar.

Die dritte und letzte Art von Knoten, deren Darstellungsart mit Hilfe der **mittleren** Maustaste geändert werden kann, sind die Sequenzknoten. Die Änderung der Anzahl der in ihnen angezeigten Komponenten wird im letzten Teil dieses Beispiels behandelt.

Um einen Sequenzknoten zu erhalten, der mehr als die standardmäßig angezeigten 20 Komponenten enthält, ist die Wurzel des im Baumfenster angezeigten Baumes auszuwerten. Der entstehende Baum hat genau einen Knoten und ist in Abbildung 4.24 zu sehen.

Die Tatsache, daß sich in der durch den Knoten repräsentierten Sequenz mehr Komponenten befinden als momentan dargestellt sind, wird durch die drei Punkte vor der schließenden eckigen Klammer am Ende der Knotenbeschriftung verdeutlicht.

Bei einer Betrachtung des in Abbildung 4.24 dargestellten Knotens fällt auf, daß der Knoten erheblich breiter ist als seine Beschriftung. Sein Erscheinungsbild wird maßgeblich durch die Werte der Grafikkonstanten bestimmt. Auf ihre Wirkung wird in 5.6.1 genauer eingegangen. Durch Änderung der Ressourcedatei `kiel.rc` (siehe 3.4.1) können die Werte der Grafikkonstanten beliebigen Situationen angepaßt werden.

Um dieses Beispiel abzuschließen, soll die Darstellung des angezeigten Knotens dahingehend geändert werden, daß die komplette Sequenz aus der Beschriftung des Knotens erkennbar ist. Da die repräsentierte Sequenz 21 Komponenten enthält, muß die Anzahl der anzuzeigenden Komponenten nur geringfügig erhöht werden.

Diese Änderung wird entsprechend der zuvor vorgenommenen Darstellungsänderungen ausgeführt. Der daraufhin im Baumfenster angezeigte Baum ist in Abbildung 4.25 dargestellt.

Jetzt ist die komplette Liste im Baumfenster zu erkennen. Der Wert der vorher nicht dargestellten Komponente ist das Ergebnis der Auswertung des Ausdrucks

```
truncate(13.9 / 0.03) div ~1,
```

auf dessen reellwertige Division weiter oben eingegangen wurde.

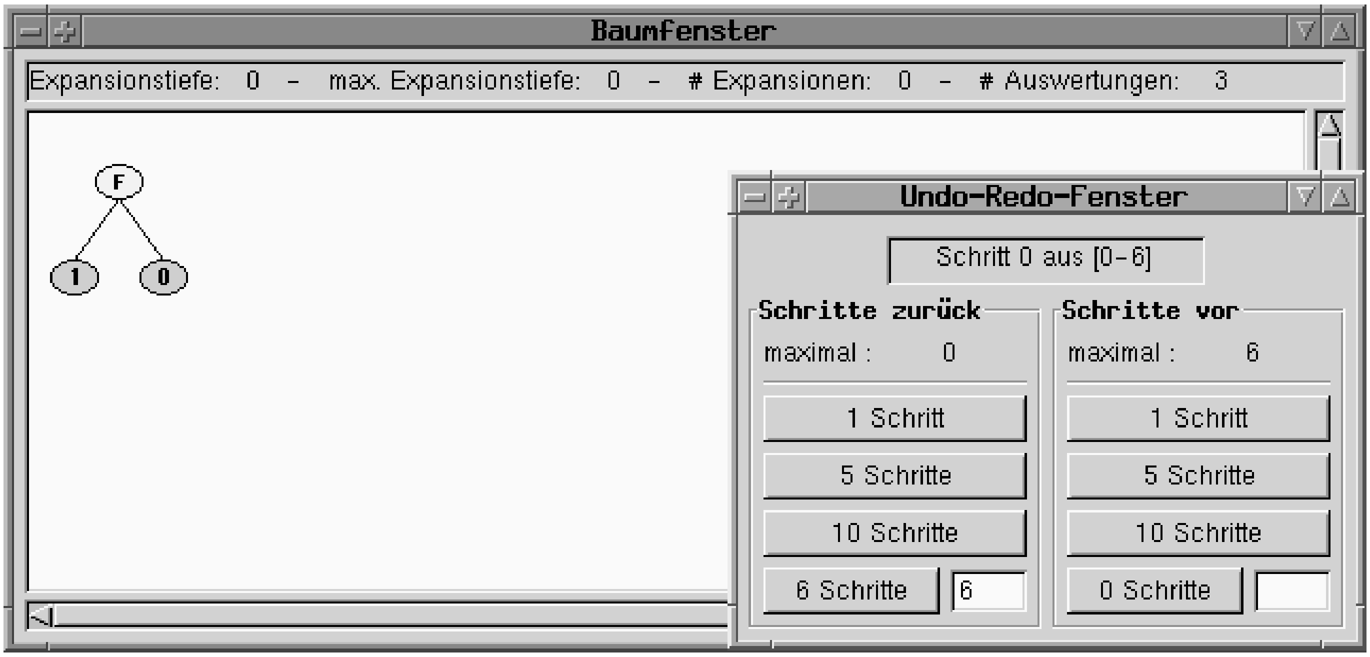


Abbildung 4.16: Das Baumfenster nach Wiederherstellung des ursprünglichen Ausdrucks

Abbildung 4.17: Das Baum- und das Undo-Redo-Fenster nach Wiederherstellen von zwei Auswertungsschritten

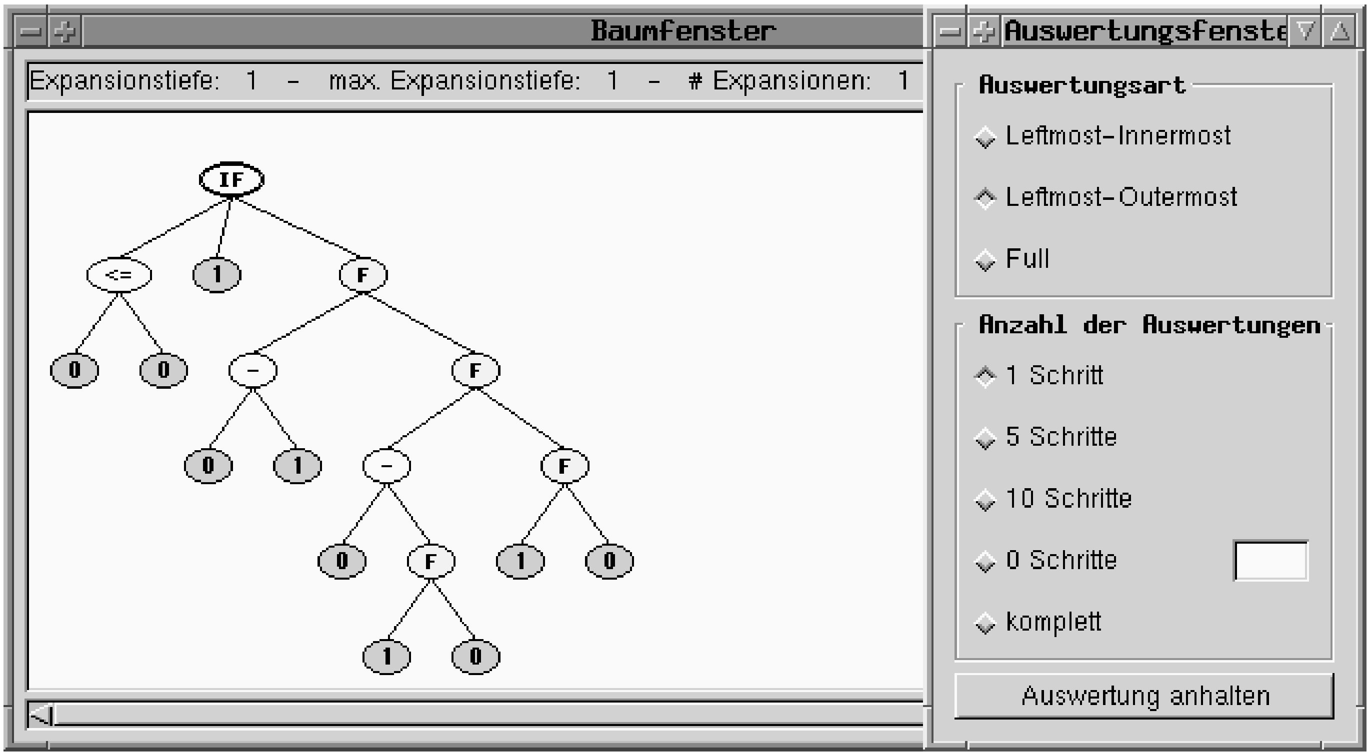


Abbildung 4.18: Das Baum- und das Auswertungsfenster nach einer Leftmost-Outermost-Expansion

Abbildung 4.19: Das Baumfenster nach Beendigung der Auswertung

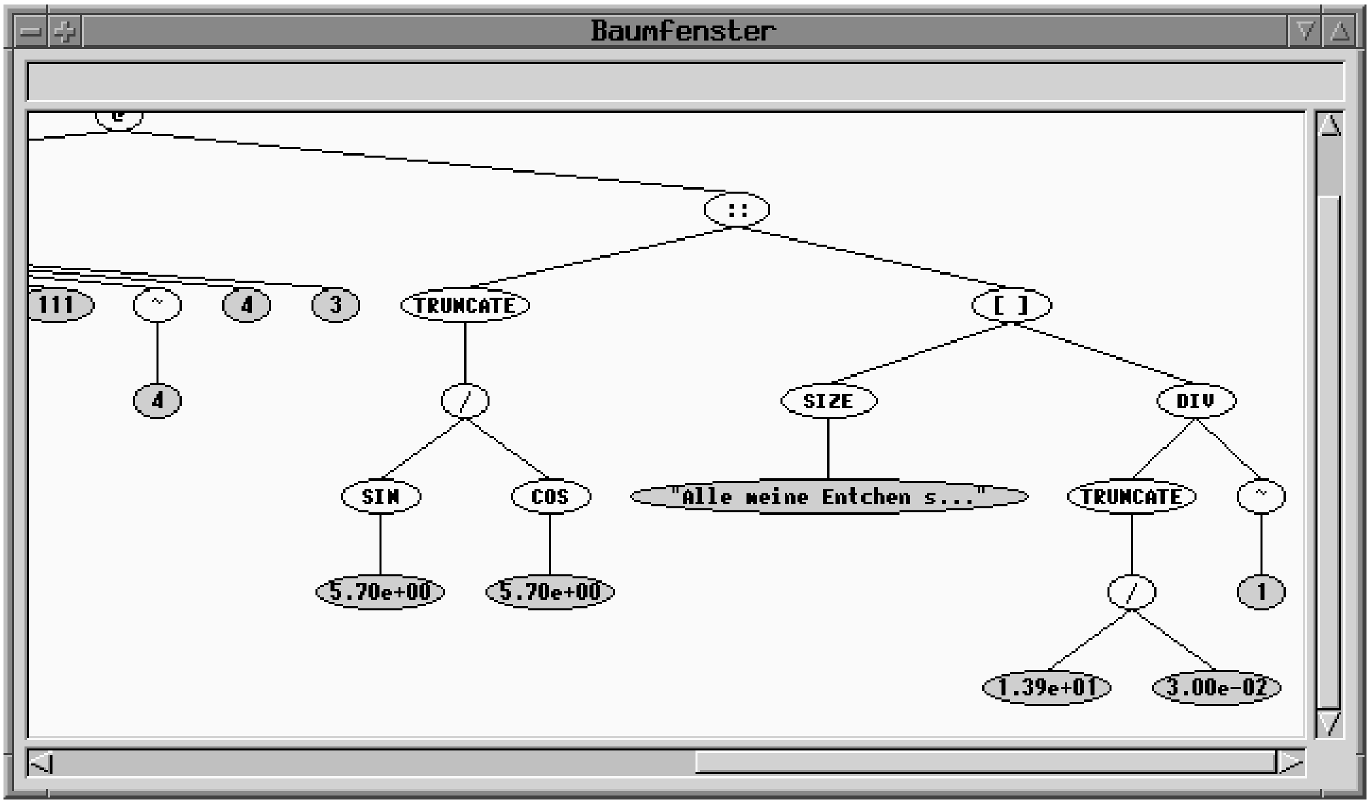


Abbildung 4.20: Der rechte untere Teil eines komplexeren Baumes

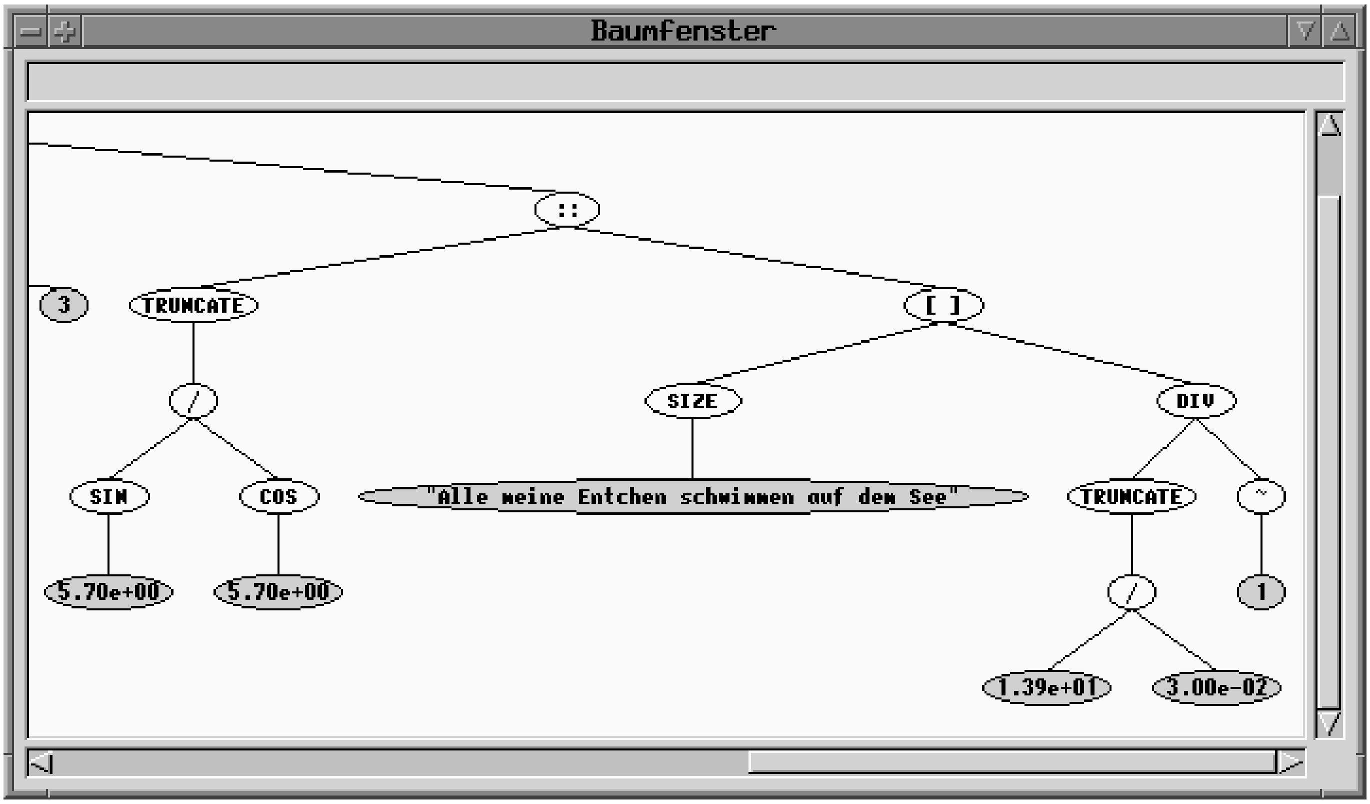


Abbildung 4.21: Der Baum nach Änderung der Darstellungsart für einen Zeichenreihenknoten

Abbildung 4.22: Das Baumfenster nach Ausführung einer Division

Abbildung 4.23: Das Baumfenster mit einem reellen Knoten mit zehn Nachkommastellen

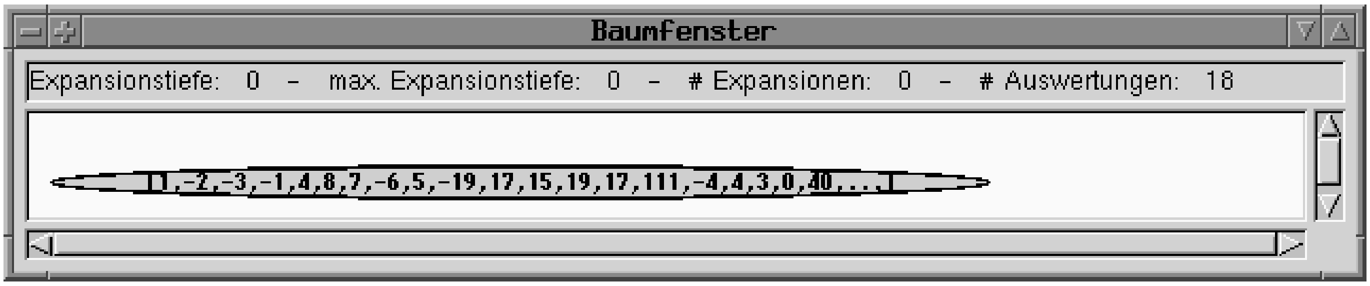


Abbildung 4.24: Ein Sequenzknoten im Baumfenster in verkürzter Darstellung

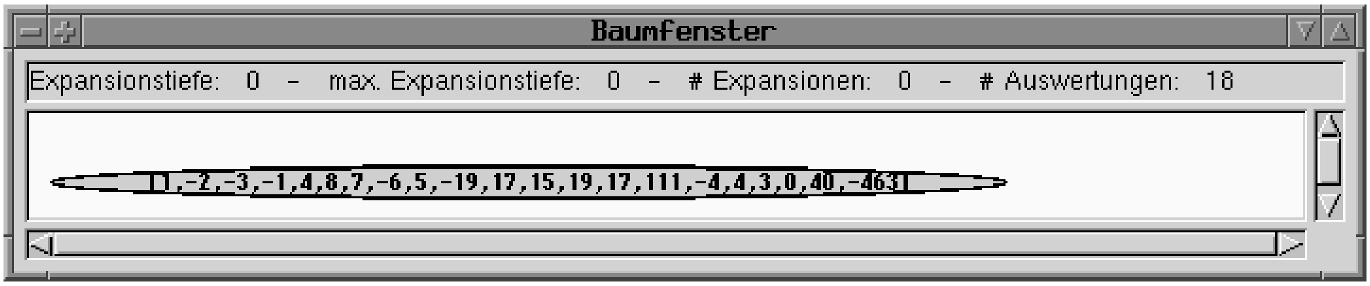


Abbildung 4.25: Ein komplett dargestellter Sequenzknoten im Baumfenster

Kapitel 5

Implementierungsdetails

Nachdem in Kapitel 3 die Bedienung von KIEL beschrieben wurde und in Kapitel 4 einige Arbeitsbeispiele folgten, wird in diesem Kapitel auf Details der Implementierung eingegangen.

Für das Arbeiten mit KIEL sollten die folgenden Ausführungen nicht unbedingt erforderlich sein. Sie dienen jedoch dem Verständnis und zeigen, wie einige Programmteile verwirklicht wurden.

5.1 Verwendete Werkzeuge: GTK, GDK und GLib

GTK, GDK und GLib sind drei C-Bibliotheken, auf die aufbauend KIEL programmiert wurde.

GTK [GTK99] ist eine Bibliothek zur Erstellung von grafischen Benutzerschnittstellen. Ursprünglich wurde GTK zur Entwicklung des „General Image Manipulation Program“, kurz GIMP, erstellt. Daher rührt auch der Name „GIMP Toolkit“, kurz GTK. Ein typischer Bildschirm, der bei der Arbeit mit GIMP zu sehen ist, wird in Abbildung 5.1 dargestellt.

Mittlerweile wird GTK in einer Vielzahl von Softwareprojekten eingesetzt, wie zum Beispiel dem „GNU Network Object Model Environment“-, kurz GNOME-, Projekt. In Abbildung 5.2 ist ein Bildschirm zu sehen, der typisch für die Arbeit mit GNOME ist.

Bei der Entwicklung von GTK wurde besonders auf eine geringe Größe sowie auf hohe Effizenz geachtet. Dennoch sollte GTK flexibel genug sein, um dem Programmierer möglichst große Freiheiten in der Gestaltung der Schnittstellen zu geben. GTK erlaubt das Arbeiten mit den Standard-Benutzerschnittstellen, wie Push-, Radio- und Checkbuttons sowie Menüs, Listen und Rahmen. Des weiteren stellt GTK einige Container-Elemente zur Verfügung, mit deren Hilfe das Layout der Schnittstellen einfach festgelegt werden kann. Hinzu kommt die große Flexibilität von GTK. So muß ein Button nicht notwendigerweise ein Label als Beschriftung enthalten, sondern kann ein beliebiges grafisches Element aufnehmen, das der Programmierer nach seinen Wünschen erstellt.

GTK baut auf dem „GIMP Drawing Kit“, kurz GDK [GDK99] auf. GDK ist ein sogenannter Wrapper, also eine Verpackung, um die Basisfunktionen zur Grafikerzeugung des zugrundeliegenden Systems zur Fensterverwaltung anzusprechen. Bei KIEL ist dieses System das „X Window System“, und deshalb werden die Befehle der X-Bibliothek, kurz Xlib, angesprochen. Inzwischen gibt es aber auch GTK und GDK für Windows-Betriebssysteme, so daß GDK dort auf die Windows-Grafikfunktionen zurückgreift. Durch GDK bietet GTK somit die Möglichkeit, einfach Fenster zu erzeugen und in ihnen zu zeichnen, und zwar in gleicher Weise auf verschiedenen Betriebssystemen.

Die dritte Bibliothek ist GLib [GLib99]. Sie wird sowohl von GTK wie auch von GDK benutzt. Die Bibliothek GLib wurde entworfen, um Portabilitätsproblemen vorzubeugen und um Funktionen bereitzustellen, die von den meisten größeren Softwareprojekten benötigt werden.

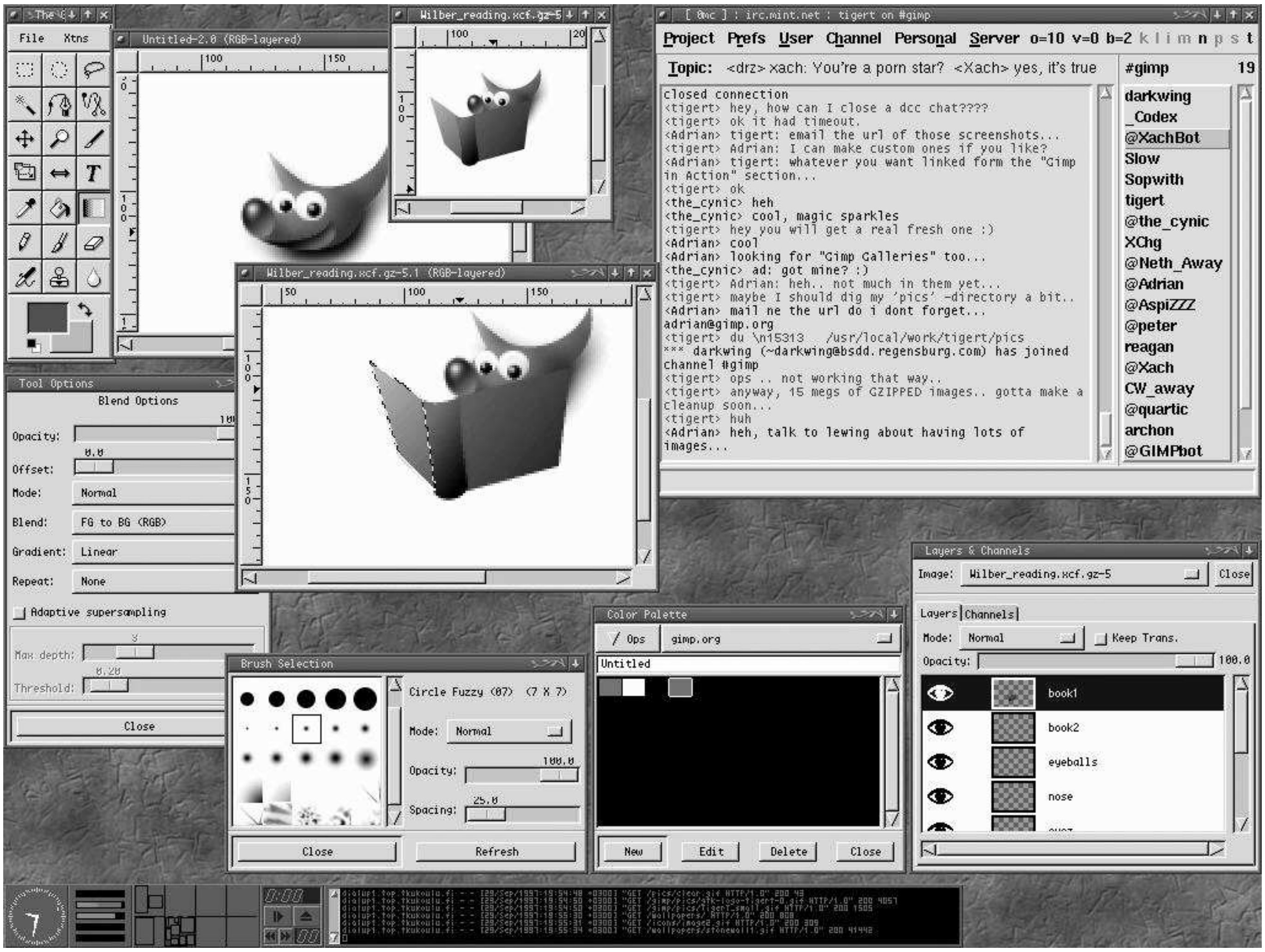


Abbildung 5.1: Ein typischer Bildschirm bei der Arbeit mit GIMP

GLib enthält neben der Ersetzung einiger Standardaufrufe viele Funktionen, die zum Beispiel das Arbeiten mit einfach und doppelt verketteten Listen, mit regulären und ausbalancierten Bäumen und mit Zeichenreihen sehr vereinfachen. Für alle eben genannten Strukturen stellt GLib Typen, Basisoperationen und viele weiterführende Funktionen zur Verfügung, so daß solche Mechanismen nicht immer wieder neu implementiert werden müssen, sondern als vorhanden angesehen werden dürfen.

5.2 Systemkomponenten

Bei jedem größeren Programmierprojekt ist die Aufteilung der Funktionen und Daten in Systemkomponenten von entscheidender Bedeutung für die Übersichtlichkeit, die Wartung und die Weiterentwicklung.

KIEL besteht aus Systemkomponenten

- zur Verarbeitung von **Ausdrücken**,
- zur Verarbeitung von **Programmen**,
- zur **Auswertung** von Ausdrücken,
- zum Bearbeiten von **Bäumen**,

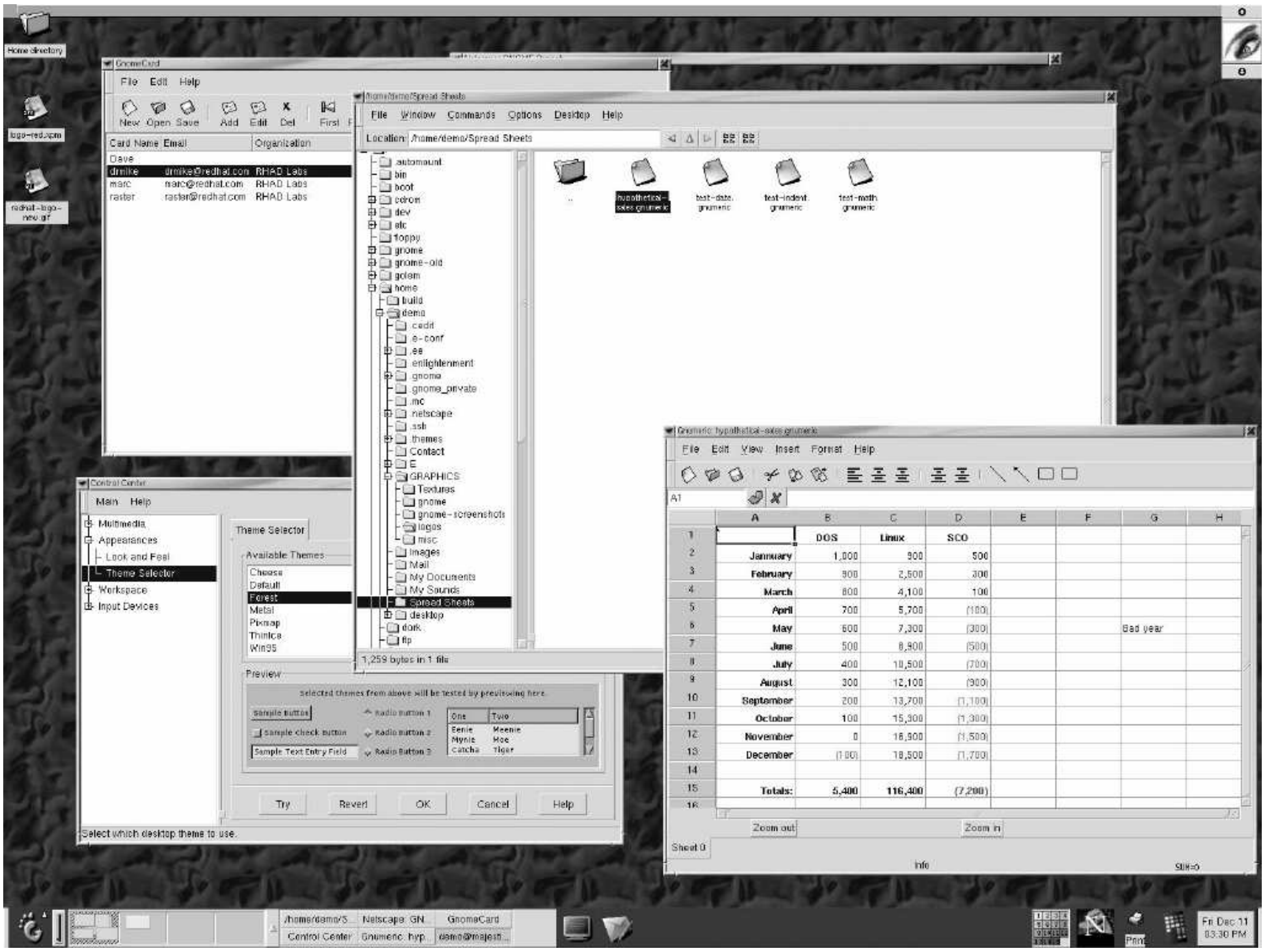


Abbildung 5.2: Ein typischer Bildschirm bei der Arbeit mit GNOME

- zur **Grafikausgabe** auf dem Bildschirm,
- zur Bereitstellung vom **Fenstern** und deren Funktionalität,
- zum **Protokollieren** von Auswertungen,
- zur Verwaltung des **Speichers**,
- zur Bereitstellung des **Hauptprogramms** und
- zur Bereitstellung von **Hilfsfunktionen** sowie
- den **Ressourcedateien** und
- dem **Makefile**.

Die Bestandteile der oben aufgeführten Systemkomponenten sowie deren Aufgaben werden in 5.2.1 bis 5.2.10 kurz beschrieben. Die Ressourcedateien werden in 5.2.11 und das Makefile in 5.2.12 behandelt.

Mit Ausnahme der letzten beinhaltet jede Systemkomponente mindestens eine `.c`-Datei, in der die jeweiligen Funktionen stehen. Ihr Name beginnt mit `kiel_`, gefolgt von einer Abkürzung der Systemkomponente.

Zu jeder `.c`-Datei gehört eine Headerdatei gleichen Namens mit der Endung `.h`.

In vielen Fällen sind in der .c-Datei neben den Funktionen, die Schnittstellen zu den übrigen Systemkomponenten darstellen, auch interne Funktionen verwirklicht. Diese werden nur innerhalb der Datei verwendet. Sie werden den anderen Systemkomponenten von KIEL nicht über die .h-Datei bekannt gemacht, so wie es bei den Schnittstellen-Funktionen der Fall ist. Es gibt eine spezielle private Headerdatei, in der die internen Funktionenköpfe deklariert werden. Anstelle der Endung .h endet sie bei sonst gleichem Namen mit _priv.h.

5.2.1 Verarbeitung von Ausdrücken

Diese Systemkomponente besteht aus den Dateien

- ausdruck.l,
- ausdruck.y,
- kiel_adk.c,
- kiel_adk.h und
- kiel_adk_priv.h.

In den Dateien ausdruck.l und ausdruck.y sind ein **Scanner** und ein **Parser** definiert [LexYacc97], die zum Einlesen von Ausdrücken in dem zugrundeliegenden Format dienen. Dieses Format ist in den Syntaxdiagrammen in A.2 dargestellt.

Mit Hilfe der Funktionen dieser Systemkomponente können verschiedene Eigenschaften von Ausdrücken abgefragt werden. Außerdem bieten sie die Möglichkeit zum Laden einer Ausdruckdatei und zum Prüfen des Ausdrucks.

5.2.2 Verarbeitung von Programmen

Die Dateien

- programm.l,
- programm.y,
- kiel_prg.c,
- kiel_prg.h,
- kiel_prg_priv.h,
- kiel_vorprg.c,
- kiel_vorprg.h und
- kiel_vorprg_priv.h

bilden die Systemkomponente zur Verarbeitung von Programmen. Analog der Verarbeitung von Ausdrücken definieren die Dateien programm.l und programm.y einen eigenen **Scanner** und **Parser** [LexYacc97].

Die Dateien, die mit kiel_prg beginnen, beinhalten ähnlich der vorigen Systemkomponente Funktionen zum Laden von Rechenvorschriftendateien und zum Arbeiten mit Rechenvorschriften.

In den Dateien `kiel_vorprg.c`, `kiel_vorprg.h` und `kiel_vorprg_priv.h` ist als einzige Schnittstellen-Methode `Vorprogramm_Name_Sorte_Liste_holen` realisiert. Sie liefert vor dem Scannen und Parsen einer Eingabedatei eine Liste, in der die Namen aller in dieser Datei definierten Rechenvorschriften zusammen mit der jeweiligen Ergebnissorte stehen. Diese Information wird vom Parser benötigt, um festzustellen, ob die Typen in den Rümpfen der Rechenvorschriften zueinander passen.

5.2.3 Auswertung von Ausdrücken

Wenn ein Ausdruck in KIEL eingegeben ist und die benötigten Rechenvorschriften geladen sind, kann dieser Ausdruck ausgewertet werden. Die Auswertung wird über Funktionen gesteuert, die in den Dateien

- `kiel_ausw.c`,
- `kiel_ausw.h` und
- `kiel_ausw_priv.h`

stehen. Als Schnittstellen-Funktionen stehen Funktionen zum Übernehmen von `int`-, `real`- und `string`-Typen sowie zum Ausführen eines Auswertungsschrittes zur Verfügung. Exemplarisch ist die Funktion zum Übernehmen einer `int`-Variablen in 5.7.8 beschrieben.

Die drei verschiedenen Arten an Auswertungsschritten bedingen die Existenz der drei Funktionen

- `Knoten_auswerten`,
- `Knoten_expandieren` und
- `Knoten_vereinfachen`.

Von den internen Funktionen dieser Systemkomponente ist die Funktion `auszuwertender_knoten` interessant, die in Abhängigkeit der ausgewählten Termersetzungssemantik den als nächstes auszuwertenden Knoten eines Ausdrucks bestimmt.

5.2.4 Bearbeiten von Bäumen

Das Verarbeiten der Baumdaten wird von dieser Systemkomponente übernommen. Sie besteht aus den Dateien

- `kiel_baum.c`,
- `kiel_baum.h` und
- `kiel_baum_priv.h`.

Es gibt zwei Schnittstellen-Funktionen.

Die Funktion `erstelle_Baum` dient zum Erstellen eines Baumes. Genaugenommen werden für die Knoten einer vorhandenen Baumstruktur die Zeilen- und die Spaltennummer ermittelt, was Voraussetzung zum grafischen Darstellen des Baumes ist. Die genaue Vorgehensweise ist in 5.6 beschrieben.

Die Funktion `zeichne_Baum` gibt eine Baumstruktur, in der die Knoten positioniert sind, im Baumfenster (siehe 3.3.6) aus.

5.2.5 Grafikausgabe auf dem Bildschirm

Die aus den Dateien

- `kiel_zei.c` und
- `kiel_zei.h`

bestehende Systemkomponente zur Grafikausgabe auf dem Bildschirm stellt verschiedene Funktionen zur Verfügung, um

- grafische Elemente auf einer Zeichenfläche anzuzeigen und
- Positionen und Ausmaße von Knoten auf der Zeichenfläche des Baumfensters zu bestimmen.

Es können Knoten und Kanten zwischen beliebigen Knoten gezeichnet werden. Die Positionen sind durch die Inhalte der jeweiligen Knotenstrukturen (siehe 5.4.1.3) und die Grafikkonstanten (siehe 5.6.1) festgelegt. Bei dem Zeichnen eines Knotens kann zusätzlich noch festgelegt werden, ob der Knoten fett oder normal gezeichnet werden soll. Fett soll immer der zuletzt vom Benutzer angeklickte Knoten gezeichnet werden (siehe 3.3.6).

Durch kurze Funktionen wie `rechte_untere_Ecke_des_Knotens_y` lassen sich alle benötigten Daten über die Position eines Knotens auf der Zeichenfläche des Baumfensters einholen. Bei der Positionierung der Knoten auf der Zeichenfläche ist insbesondere die Funktion `breite_des_Knotens` von Bedeutung, die die horizontale Ausdehnung eines Knotens in Abhängigkeit der Grafikkonstanten liefert. Ihre Anwendung ist in 5.6.2 bei der Behandlung des Positionierungsalgorithmus zu sehen.

5.2.6 Bereitstellung von Fenstern und deren Funktionalität

Die grafische Interaktion zwischen KIEL und dem Benutzer ist ein Hauptbestandteil des Programmpaketes. Aufgrund der Komplexität wurden die Funktionen dieser Systemkomponente in mehrere Teilkomponenten aufgeteilt. Da alle Teilkomponenten hauptsächlich über GTK-Funktionen realisiert sind, werden die Buchstaben `gtk` als Abkürzung verwendet. Die Teilkomponenten sind nach Fensterarten aufgeteilt. In der Tabelle 5.1 sind die Abkürzungen der Teilkomponenten mit den dazugehörigen Fenstern aufgeführt.

Abkürzung der Teilkomponente	Fenster, die in der Teilkomponente behandelt werden
`aef`	Ausdruckeingabefenster
`awf`	Auswertungsfenster
`bf`	Befehlsfenster
`bmf`	Baumfenster
`df`	Dateifenster
`uef`	Übersichtsfenster
`urf`	Undo-Redo-Fenster
`wf`	Warnungsfenster
`zrf`	Zeichenreihenfenster

Tabelle 5.1: Abkürzungen der grafischen Teilkomponenten

Diese Aufteilung führt zu einer größeren Anzahl an Dateien, die zu dieser Systemkomponente gehören. Dies sind

- kiel_gtk.c,
- kiel_gtk.h,
- kiel_gtk_priv.h,
- kiel_gtk_fenster.h,
- kiel_gtk_aef.c,
- kiel_gtk_aef.h,
- kiel_gtk_aef_priv.h,
- kiel_gtk_awf.c,
- kiel_gtk_awf.h,
- kiel_gtk_awf_priv.h,
- kiel_gtk_bf.c,
- kiel_gtk_bf.h,
- kiel_gtk_bf_priv.h,
- kiel_gtk_bmf.c,
- kiel_gtk_bmf.h,
- kiel_gtk_bmf_priv.h,
- kiel_gtk_df.c,
- kiel_gtk_df.h,
- kiel_gtk_df_priv.h,
- kiel_gtk_uef.c,
- kiel_gtk_uef.h,
- kiel_gtk_uef_priv.h,
- kiel_gtk_urf.c,
- kiel_gtk_urf.h,
- kiel_gtk_urf_priv.h,
- kiel_gtk_wf.c,
- kiel_gtk_wf.h,
- kiel_gtk_wf_priv.h,
- kiel_gtk_zrf.c und
- kiel_gtk_zrf.h.

In den Dateien der Teilkomponenten stehen Funktionen zur Darstellung der jeweiligen Fenster sowie zur Verarbeitung von Ereignissen, die in den Fenstern stattfinden. Zum Beispiel gibt es in der Datei `kiel_gtk_bf.c` eine Funktion, die den Timer (siehe 3.3.1) realisiert.

In den Dateien `kiel_gtk.c`, `kiel_gtk.h` und `kiel_gtk_priv.h` sind neben der in 5.2.9 erwähnten Funktion `GTK_Schleife` auch Funktionen verwirklicht, die die Kommunikation unter den Teilkomponenten verwirklichen. Zu den Headerdateien `kiel_gtk.h` und `kiel_gtk_priv.h`, die die Funktionen allen anderen bzw. dem eigenen `.c`-Programm zugänglich machen, kommt noch die Headerdatei `kiel_fenster.h`, die Funktionen bereitstellt, welche genau von allen Teilkomponenten genutzt werden können.

5.2.7 Protokollieren von Auswertungen

Die Dateien

- `kiel_pkl.c` und
- `kiel_pkl.h`

bilden die Systemkomponente zum Protokollieren von Auswertungen.

Mit der Funktion `Protokoll_Eintrag_hinzufuegen` kann ein Protokolleintrag, also ein Auswertungsbaum, an ein Protokoll angehängt werden. Die Funktion `Protokoll_Eintrag_holen` gibt den durch den zweiten Parameter bestimmten Eintrag eines Protokolls zurück. Mit der Funktion `Protokoll_Anzahl_Eintraege` kann die Anzahl der in einem Protokoll vorhandenen Einträge ermittelt werden. Um einen bestimmten Zustand, der in einem Protokoll festgehalten wurde, wiederherzustellen, wurde die Funktion `Protokoll_ausfuehren` programmiert. Diese ist detailliert in 5.7.7 beschrieben.

5.2.8 Verwaltung des Speichers

Alle Funktionen zur Verwaltung von Speicherbereichen sind in den Dateien

- `kiel_mem.c`,
- `kiel_mem.h` und
- `kiel_mem_priv.h`

zusammengefaßt. Die Schnittstellen-Funktionen lassen sich in fünf Gruppen einteilen:

- Mit Hilfe der Funktionen `unitelm2retval`, `gboolean2retval`, `gint2retval`, `gdouble2retval`, `gstring2retval`, `emptylist2retval` und `null2retval` wird der aufrufenden Funktion jeweils ein Zeiger auf eine Struktur vom Typ `RETVAL_STRUCT` (siehe 5.4.1.1) zurückgegeben. Sie werden beim Parsen eines Ausdrucks oder einer Rechenvorschrift aufgerufen und erzeugen aus einem Wert eines bestimmten ML-Typs eine solche Struktur, in der der entsprechende Wert gesetzt ist.

  Die Funktion `null2retval` wird benutzt, falls der zugehörige Knoten nicht ausgewertet ist. In diesem Fall soll die Struktur keinen Wert repräsentieren, was durch das Setzen der Komponente `typ` auf den Wert `NO_TYP` erreicht wird.

- Die Funktionen, deren Name mit `create_` beginnt, bilden die zweite Gruppe. Sie dienen zum Erzeugen von gewissen Strukturen. Über die Parameter werden die Werte übergeben, die den Komponenten der Strukturen zugewiesen werden sollen. Auf diese Art können beispielsweise Teilbaum- und Rechenvorschriftsstrukturen erzeugt werden.

- Die dritte Gruppe besteht aus einigen Funktionen zum Freigeben von Speicherbereichen, die durch verschiedene Strukturen belegt wurden. Bei der Programmierung von KIEL wurde Wert darauf gelegt, daß die Speicherbereichsbehandlung möglichst genau betrieben wird. So wird in der Regel jeder Speicherbereich, der nicht mehr benötigt wird, freigegeben.

- Während einer Auswertung ist es häufig erforderlich, gewisse Strukturen zu duplizieren. Dies ist zum Beispiel bei Ganzzahlvariablen durch einfaches Zuweisen möglich. In KIEL wird jedoch mit komplexeren, selbstdefinierten Strukturen gearbeitet, die durch Zeiger angesprochen werden. Ein einfaches Kopieren des Wertes würde in diesen Fällen der neuen Variablen einen Zeiger zuweisen, der auf denselben Speicherbereich zeigt wie die zu kopierene Variable. Das Arbeiten mit der neuen Variablen würde somit immer den Inhalt der zu kopierenden Variablen verändern. Damit das Duplizieren in der gewünschten Art geschehen kann, ist es notwendig, neuen Speicherbereich zu allokieren und in diesen die Werte der Ursprungsvariablen zu übernehmen. Diese Aufgabe wird durch die Funktionen `retval_dup`, `ausdruckbaum_dup`, `ausdruck_dup`, `rechenvorschrift_dup` und `retval_liste_dup` erfüllt.

  Zusätzlich gibt es noch die Funktion `Ausdruck_initialisieren`, die eine Ausdrucksstruktur erzeugt und deren Komponenten mit neutralen Werten vorbelegt.

- Die letzte Gruppe bilden die mt-Funktionen, die die interne Speicherbereichsverwaltung übernehmen. Sie werden in 5.7.6 genauer behandelt.

5.2.9 Bereitstellung des Hauptprogramms

Die Systemkomponente für das Hauptprogramm ist die kleinste Systemkomponente von KIEL. Sie besteht aus den Dateien

- `kiel_main.c` und
- `kiel_main.h`

und stellt lediglich die Funktion `main` zur Verfügung. In ihr werden

- der Programmaufruf getestet,
- die mt-Funktionen (siehe 5.7.6) initialisiert und
- die Funktion `GTK_Schleife` aufgerufen, die KIEL bis zum Programmende ausführt und die alle Ereignisse verarbeitet.

5.2.10 Bereitstellung von Hilfsfunktionen

Einige Funktionen, die nicht direkt in andere Systemkomponenten passen, sind in den Dateien

- `kiel_fkt.c`,
- `kiel_fkt.h` und
- `kiel_fkt_priv.h`

untergebracht. Im folgenden sind alle Schnittstellen-Funktionen aufgeführt.

- Die `kiel_debug`-Funktionen, die zur Entwicklungszeit dem Programmierer eine gute Orientierung während einer Programmausführung geben, werden detailliert in 5.7.4 beschrieben.

- Die Funktion `Liste_aus_Ausdruckbaeumen` wird während des Parsens benutzt, um aus einer variablen Anzahl an Teilbäumen eine Liste zu erstellen, in der genau diese Teilbäume stehen. Ihr Aufbau wird in 5.7.3 genau behandelt.

- Die Funktion `string_teil` hat eine Zeichenreihe, eine Ganzzahl und ein einzelnes Zeichen als Argumente. Ihre Aufgabe ist es, die Zeichenreihe als Folge von Zeichenreihen zu interpretieren, die durch das übergebene Trennzeichen voneinander getrennt sind. Falls es die Anzahl der Folgeglieder zuläßt, gibt sie als Ergebnis das Folgeglied zurück, das an der durch das zweite Argument spezifizierten Stelle steht.

  Da eine einzulesende Ausdruckdatei aus Ausdrücken besteht, die durch ein Semikolon voneinander getrennt sind, ist der Nutzen dieser Funktion für KIEL klar erkennbar.

- Als Warnzeichen gibt die Funktion `beep` das Zeichen mit dem ASCII-Code 7 auf der Konsole aus, von der KIEL aufgerufen wurde.

- Die Ressourcedaten der Datei `kiel.rc` sind in 3.4.1 beschrieben. Die Informationen aus dieser Datei werden mit Hilfe der Funktion `ressourcen_einlesen` in die dafür vorgesehenen globalen Variablen (siehe 5.5.1.3) eingelesen.

5.2.11 Die Ressourcedateien

Über Ressourcedateien haben der Benutzer und der Entwickler die Möglichkeit, die Programmausführung ihren Vorstellungen gemäß anzupassen, ohne Änderungen in den Funktionen von KIEL vorzunehmen.[1] Die Ressoucedaten sind auf die Dateien

- `kiel.rc`,
- `kiel.rc.gtk` und
- `kiel_ressource.h`

verteilt.

Die Dateien `kiel.rc` und `kiel.rc.gtk` sind dem Benutzer von KIEL zugänglich. Eine genaue Beschreibung von ihnen befindet sich in 3.4.1 bzw. 3.4.2.

Die Datei `kiel_ressource.h` enthält die Definitionen aller Texte, die bei der Programmausführung in den Fenstern erscheinen können. Sie stehen in einer Datei, die beim Kompilieren in das Programmpaket eingebunden wird. Das Ändern von hier aufgeführten Definitionen ist dem Benutzer daher in der Regel nicht möglich. Bei der Definition der Zeichenreihen gibt es zwei Blöcke. In dem einen stehen alle Texte in deutscher und in dem anderen in englischer Sprache. Die Tatsache, daß in der Datei `kiel_ressource.h` alle relevanten Zeichenreihen definiert werden, führt dazu, daß es sehr unkompliziert ist, KIEL auf andere Sprachen zu erweitern.

In der Datei `kiel_ressource.h` sind des weiteren globale Variable deklariert, welche die durch die Ressourcedatei `kiel.rc` bestimmten Werte während der Programmausführung aufnehmen und allen Systemkomponenten zugänglich machen.

[1] Das Ändern von Funktionen sollte dem Benutzer grundsätzlich nicht erlaubt sein.

5.2.12 Das Makefile

Das Makefile von KIEL hat den Standardnamen `Makefile`. In ihm stehen Informationen, wie das Programm `make` aus den Quelldateien von KIEL die ausführbaren Programme erstellen soll.

Nach dem Definitionsblock, in dem alle benötigten Variablen definiert werden, widmen sich die ersten Anweisungen der Erzeugung des C-Codes für den Scanner und Parser. Die so erzeugten Quelldateien werden beim Kompilieren in die Ausdruck- und Programm-Systemkomponente eingegliedert.

Die zu erzeugenden Objektdateien des Programmpaketes können in die folgenden drei Gruppen gegliedert werden:

- sprachunabhängige Dateien
- Dateien für die deutschsprachige Version
- Dateien für die englischsprachige Version

Diese Gruppierungen werden in verschiedenen Variablen festgehalten. Für das Erstellen der deutschsprachigen Version von KIEL müssen zum Beispiel nur die sprachunabhängigen und deutschsprachigen Objektdateien erzeugt werden. Die Quelldateien für die Objektdateien der deutsch- und der englischsprachigen Version sind dieselben. Der einzige Unterschied besteht darin, daß bei der Erzeugung der englischsprachigen Objektdateien das Makro `ENGLISCH` definiert wird. Dies veranlaßt den Kompiler, anstelle der deutschen Texte die englischen in den Objektdateien zu verwenden. Nach der Erzeugung aller Objektdateien, werden diese zu der jeweiligen ausführbaren Datei gebunden. Die Dateien sind `kiel_deutsch` und `kiel_englisch`. Abschließend wird noch ein symbolischer Link auf die Datei `kiel_deutsch` erzeugt, so daß diese direkt durch die Eingabe von `kiel` aufgerufen werden kann. In dem `clean`-Teil des Makefiles stehen alle Objektdateien, die Scanner- und Parserdateien, die ausführbaren Dateien und der symbolische Link. Eine Ausführung von `make clean` bewirkt, daß bis auf die Quelldateien alle Dateien von KIEL gelöscht werden.

5.3 Bereichsgrenzen

In diesem Abschnitt werden die Bereiche der Typen in KIEL aufgeführt. Im wesentlichen entsprechen sie denen von ML.

Ausnahmen sind lediglich die Zeichenreihen und die Sequenzen. Für diese ist in ML keine Längenbegrenzung vorgesehen. Sie können theoretisch beliebig lang sein. Der von der Hardware zur Verfügung gestellte Speicherplatz kann aber nicht beliebig lange Zeichenreihen oder Sequenzen aufnehmen. Im Vergleich zu früher erscheint er sehr groß. Aber selbst heutzutage unvorstellbar große Speichermedien werden nichts an der Tatsache ändern, daß er immer endlich sein wird. Aus diesem Grund gibt es in KIEL Längenbegrenzungen für Zeichenreihen und Sequenzen.

Im folgenden sind alle Typen, die KIEL unterstützt, mit den dazugehörigen Bereichen aufgeführt:

`unit`	:	Das einzige Element ist `()`.
`bool`	:	Es gibt zwei Elemente: `TRUE` und `FALSE`.
`int`	:	Alle 31-Bit-Ganzzahlen, also alle Ganzzahlen von -2^{30} bis $2^{30}-1$
`real`	:	Alle Gleitkommazahlen mit doppelter Präzision
`string`	:	Alle Folgen von 8-Bit-ASCII-Zeichen, deren Länge die Obergrenze `MAXIMALE_ANZAHLZEICHENIMSTRING` nicht übersteigt
Sequenzen	:	Alle Sequenzen über dem jeweiligen Typ, die nicht die Länge `MAXIMALE_ANZAHLELEMENTEINLISTE` überschreiten.

Die Konstanten `MAXIMALE_ANZAHLZEICHENIMSTRING` und `MAXIMALE_ANZAHLELEMENTEINLISTE` werden durch die Ressourcedatei `kiel.rc` (siehe 3.4.1) mit Werten belegt. Der Benutzer hat also selbst die Möglichkeit, diese Obergrenzen abzuändern und auf seine Bedürfnisse anzupassen.

5.4 Verwendete Strukturen

Die in KIEL verwendeten Strukturen gliedern sich in zwei Kategorien, und zwar in

- globale Strukturen, die in der Datei `kiel_global.h` stehen und aus allen Include-Dateien ansprechbar sind, und in
- lokale Strukturen, die in jeweils nur einer Include-Datei verwendet werden.

5.4.1 Globale Strukturen

Die folgenden Strukturdefinitionen stehen in der Datei `kiel_global.h`. Sie bestimmen die grundlegende Interpretation der Speicherbereiche, mit denen KIEL arbeitet.

5.4.1.1 Struktur `RETVAL_STRUCT`

```
typedef struct RetValStruct
{
  gint      typ;

  gpointer *_unit;
  gboolean  _bool;
  gint      _int;
  gdouble   _real;
  gchar    *_string;
  GList    *_list;
} RETVAL_STRUCT;
typedef RETVAL_STRUCT  *RETVAL_TYP;
```

Die Struktur `RETVAL_STRUCT` dient zur Aufnahme eines Wertes eines Ausdrucks. In der Komponente `typ` wird festgehalten, welche der anderen Komponenten einen Wert enthält. Dies ist höchstens eine. Werte aller in KIEL verwendeten Typen können in einer Variablen des Strukturtyps `RETVAL_STRUCT` aufgenommen werden:

- `unit`-Typen

  Es gibt nur ein Element vom Typ `unit`: `()`. Dieses wird in KIEL intern durch den Null-Pointer dargestellt.

- `bool`-Typen

  Die Wahrheitswerte werden durch die Werte des GLib-Typs `gboolean` repräsentiert.

- `int`-Typen

  Für Ganzzahlwerte in KIEL wird der 32-Bit-GLib-Typ `gint` verwendet. In Variablen dieses Typs können Werte von -2^{31} bis $2^{31}-1$ verwaltet werden — eine Obermenge der in KIEL unterstützten Ganzzahlmenge (siehe 5.3).

- `real`-Typen

  Die reellen Zahlen werden in KIEL durch Gleitkommazahlen mit doppelter Präzision repräsentiert. Dies entspricht genau dem GLib-Typ `gdouble`.

- `string`-Typen

  Zeichenreihen sind Sequenzen von 8-Bit-ASCII-Zeichen. In der Struktur `RETVAL_STRUCT` wird ein Zeiger zum ersten Zeichen einer Zeichenreihe in der Komponente `_string` aufgenommen.

- Listen-Typen

  Listen-Typen werden durch einen Zeiger auf eine Struktur vom GLib-Typ `GList` verwaltet. In der entsprechenden Liste sind die Komponenten eingetragen. Dies sind wieder Variable der Struktur `RETVAL_STRUCT`.

Des weiteren wird der Typ `RETVAL_TYP` als Zeiger auf eine Struktur von Typ `RETVAL_STRUCT` definiert.

5.4.1.2 Struktur `WetShaTyp`

```
typedef struct WetShaTypStruct
{
  gdouble x;
  gdouble modifier;
} WetShaTyp;
```

Die Struktur `WetShaTyp` ist die einzige Struktur in KIEL, die nicht über Zeiger angesprochen wird. Sie wird nur in der Struktur `AUSDRUCKBAUM_STRUCT`, die im Anschluß aufgeführt ist, verwendet und dient zur Aufnahme von Werten zur Positionierung des jeweiligen Knotens im gesamten Ausdrucksbaum. Diese Positionierung geschieht nach einem Algorithmus, der auf dem Baumzeichenalgorithmus von Wetherell und Shannon [WetSha79] aufbaut. Der verwendete Algorithmus wird in 5.6.2 beschrieben. Dort wird auch die Verwendung der Komponenten `x` und `modifier` deutlich gemacht.

5.4.1.3 Struktur `AUSDRUCKBAUM_STRUCT`

```
typedef struct ausdruckBaumStruct
{
  RETVAL_TYP                wert;            /* Wert des Knotens            */
  gint                      anz_stellen;     /* Nachkommastellen bei real-  */
                                             /* Ausdruecken, Zeichenanzahl  */
                                             /* bei string-Ausdruecken und  */
                                             /* Komponentenanzahl bei Se-   */
                                             /* quenzausdruecken            */
  gint                      nodeIdentifier;  /* Art des Knotens             */
  gboolean                  ausgewertet;     /* Ist Knoten ausgewertet?     */
  GList                    *kinder;          /* Kinder des Knotens          */
  gint                      zeile;
  gdouble                   spalte;          /* Zeile und Spalte in der     */
                                             /* Baumdarstellung (abhaengig  */
                                             /* vom Baumzeichenalgorithmus) */
  WetShaTyp                 WetSha;          /* fuer Positionierung nach    */
                                             /* Wetherell und Shannon       */
  gint                      expansionstiefe; /* Expansionstiefe             */
  gboolean                  full_expansion;  /* Hier koennen RV-Aufrufe     */
                                             /* markiert werden, die zu     */
```

```
                                                /* expandieren sind.           */
} AUSDRUCKBAUM_STRUCT;
typedef AUSDRUCKBAUM_STRUCT         *AUSDRUCKBAUM_TYP;
```

In dem zu einem Ausdruck gehörenden Baum spannt jeder Knoten einen eigenen Teilbaum auf. Zur internen Darstellung ist also eine Struktur erforderlich, die die Daten eines Knotens aufnimmt und Verweise auf die Kinder enthält. Dabei sind die Kinder Elemente der gleichen Struktur. Die Struktur, die diese Aufgabe übernimmt, ist `AUSDRUCKBAUM_STRUCT`.

Im folgenden sind ihre Komponenten aufgelistet:

- Die Komponente `wert` steht zur Aufnahme des Knotenwertes zur Verfügung und wird gefüllt, wenn der Knoten ausgewertet wird.
- In der Komponente `anz_stellen` wird für reelle Knoten die Anzahl der Nachkommastellen, für Zeichenreihenknoten die Anzahl der anzuzeigenden Zeichen und für Listenknoten die Anzahl der anzuzeigenden Elemente gespeichert.
- Die Art des Knotens steht in der Komponente `nodeIdentifier`.
- Die boolesche Komponente `ausgewertet` enthält die Information, ob der Knoten bereits ausgewertet ist.
- Die Komponente `kinder` ist ein Zeiger auf eine GLib-Liste. In dieser Liste stehen die Speicheradressen, an denen die Teilbaumstrukturen der Kinder abgelegt sind.
- Die Komponente `zeile` legt die Zeile fest, in der der Knoten im Baumfenster angezeigt wird.
- Die Spalte im Baumfenster, in der der Knoten angezeigt wird, steht in der Komponente `spalte`. Näheres dazu ist in 5.6 zu finden.
- Die Komponente `WetSha` wird für die Bestimmung der Positionen des Knotens im Baumfenster benötigt. Sie wird während des in 5.6.2 beschriebenen Baumzeichenalgorithmus gefüllt.
- Um den automatischen Abbruch einer Auswertung nach einer festgelegten Anzahl an Expansionen realisieren zu können, hält die Komponente `expansionstiefe` die aktuelle Expansionstiefe des Knotens bezüglich der momentan andauernden Auswertung fest.
- Bei einer Full-Expansion werden alle Rechenvorschriftsknoten, die in dem aktuellen Baum vorhanden sind, simultan expandiert. Die Komponente `full_expansion` dient zur Markierung der Knoten, die in einem solchen Auswertungsschritt zu expandieren sind.

Da nie direkt mit der Struktur `AUSDRUCKBAUM_STRUCT`, sondern immer mit Verweisen auf Speicherbereiche, in denen eine solche Struktur steht, in KIEL gearbeitet wird, wird der Typ `AUSDRUCKBAUM_TYP` als Abkürzung für einen Zeiger auf eine `AUSDRUCKBAUM_STRUCT`-Struktur definiert.

5.4.1.4 Struktur `PIXMAP_DRAWINGAREA_DATEN_STRUCT`

```
typedef struct pixmap_drawingarea_daten_typ
{
  GdkPixmap              *pixm;          /* Adresse der Pixmap             */
  GtkWidget              *draw_a;        /* Adresse des Drawing Area       */
} PIXMAP_DRAWINGAREA_DATEN_STRUCT;
typedef PIXMAP_DRAWINGAREA_DATEN_STRUCT   *PIXMAP_DRAWINGAREA_DATEN_TYP;
```

Zeichenflächen werden mit Hilfe von GDK und GTK in der Art verwirklicht, daß in einem Zeichenbereich, genannt **drawing area**, eine Pixmap-Grafik angezeigt wird. Da zum Zeichnen von Elementen auf der Zeichenfläche beide Daten erforderlich sind und in KIEL mehrere Zeichenflächen verwendet werden, beinhaltet die Struktur `PIXMAP_DRAWINGAREA_DATEN_STRUCT` einen Zeiger zu einer `GdkPixmap`-Struktur und einen Zeiger zu einer `GtkWidget`-Struktur. Selbst in diesem Fall verwendet KIEL die Struktur nicht direkt, obwohl in ihr nur zwei Zeiger stehen. Auch hier wird ein Typ `PIXMAP_DRAWINGAREA_DATEN_TYP` zum Arbeiten mit der Adresse einer solchen Struktur definiert.

5.4.1.5 Struktur PROTOKOLL_STRUCT

```
typedef struct ProtokollStruct
{
  GList                         *liste;         /* Liste des Auswertungsproto-*/
                                                /* kolls fuer Undo und Redo   */
  gint                           position;      /* Position im Protokoll      */
} PROTOKOLL_STRUCT;
typedef PROTOKOLL_STRUCT *PROTOKOLL_TYP;
```

Die Struktur `PROTOKOLL_STRUCT` dient zum Protokollieren der Auswertungsschritte, die ein Benutzer an einem Ausdruck ausführt. In 3.3.9 ist dieser Mechanismus beschrieben.

Die Struktur benötigt lediglich zwei Komponenten:

- Die Komponente `liste` ist ein Zeiger auf eine GLib-Liste, in der die Auswertungsbäume nach jedem Auswertungsschritt festgehalten werden.
- Die Position im Protokoll, an der sich KIEL aktuell befindet, wird in der Komponente `position` gespeichert.

Die Struktur wird unter anderem bei der Ausführung eines Protokolls verwendet, welche detailliert in 5.7.7 beschrieben ist. Dort und an allen anderen Stellen wird der Typ `PROTOKOLL_TYP`, der als Zeiger auf die hier behandelte Struktur `PROTOKOLL_STRUCT` definiert ist, benutzt.

5.4.1.6 Struktur AUSDRUCK_STRUCT

```
typedef struct ausdruck_typ
{
  gchar                         *datei_name;      /* Dateiname der Adk.-Datei */
                                                  /* bei RV's "".             */
  AUSDRUCKBAUM_TYP               baum;            /* Ausdrucksbaum            */
  AUSDRUCKBAUM_TYP               ursprungsbaum;   /* Um Undo machen zu koennen*/
  gboolean                       Zeichnen_erlaubt;/* darf Adk. gezeichnet     */
                                                  /* werden?                  */
  gchar                         *eingabe_string;  /* Eingabe-String           */
  GList                         *knotenliste;     /* Knotenliste zum          */
                                                  /* einfacheren Ansprechen   */
                                                  /* der Knoten im Baum       */
  PROTOKOLL_TYP                  protokoll;       /* Auswertungsprotokoll fuer*/
                                                  /* Undo und Redo            */
} AUSDRUCK_STRUCT;
typedef AUSDRUCK_STRUCT  *AUSDRUCK_TYP;
```

In KIEL sind Ausdrücke von zentraler Bedeutung. Da der Benutzer mehrere Ausdrücke gleichzeitig in KIEL bearbeiten kann, ist es erforderlich, einige Angaben zu jedem Ausdruck festzuhalten. Dazu dient die Struktur `AUSDRUCK_STRUCT`.

Folgende Komponenten bestimmen einen Ausdruck:

- Falls der Ausdruck aus einer Datei geladen wurde, wird der Name der Datei in der Zeichenreihenkomponente `datei_name` festgehalten, aus der der Ausdruck stammt. Bei direkter Eingabe des Ausdrucks steht in dieser Komponente die Zeichenreihe `Eingabe`.
- Der Ausdrucksbaum wird in der Komponente `baum` festgehalten.
- Für das Protokollieren ist neben dem Speichern der Bäume nach jedem Auswertungsschritt auch der ursprüngliche Ausdrucksbaum von Interesse. Dieser wird in der Komponente `ursprungsbaum` gespeichert.
- Anhand des Wertes der Komponente `Zeichnen_erlaubt` kann KIEL feststellen, ob der entsprechende Ausdruck in einem konsistenten Zustand im Speicher vorliegt. Dies ist eine notwendige Bedingung dafür, ob im Baumfenster gezeichnet werden darf.
- Die Zeichenreihe, durch die der ursprüngliche Ausdruck beschrieben wird, wird in der Komponente `eingabe_string` gespeichert. Dies ist diejenige Zeichenreihe, die der Benutzer im Ausdruckeingabefenster (siehe 3.3.2) eingegeben hat oder die in der geladenen Ausdruckdatei steht.
- Bei der Ausführung von KIEL kommt es teilweise nicht auf die Baumstruktur eines Ausdrucks an, sondern nur auf den Inhalt der Knoten. Um die Arbeit in solchen Fällen zu beschleunigen, wird aus den Baumknoten eine lineare Liste gebildet, in der die Adressen aller Knoten stehen. Die Komponente `knotenliste` enthält einen Zeiger auf diese Liste.
- Das Protokoll zum Ausdruck wird in der Komponente `protokoll` verwaltet.

Der Typ `AUSDRUCK_TYP` dient als Abkürzung für einen Zeiger auf die hier behandelte Struktur.

5.4.1.7 Struktur `ARGUMENT_STRUCT`

```
typedef struct argument_typ
{
  gchar                  *name;          /* Name                           */
  gint                    sorte;         /* Sorte                          */
} ARGUMENT_STRUCT;
typedef ARGUMENT_STRUCT    *ARGUMENT_TYP;
```

Rechenvorschriften werden erst durch die Benutzung von Argumenten vielseitig. Für das Arbeiten mit Argumenten gibt es die Struktur `ARGUMENT_STRUCT`, in der der Name des Arguments als Zeichenreihe in der Komponente `name`, sowie die Sorte des Arguments in der Komponente `sorte` gespeichert wird. Durch die Typdefinition von `ARGUMENT_TYP` wird das Arbeiten mit Zeigern auf die Struktur vereinfacht.

5.4.1.8 Struktur `RECHENVORSCHRIFT_STRUCT`

```
typedef struct rechenvorschrift_struct
{
  gchar                  *datei_name;    /* Dateiname                      */
  gchar                  *name;          /* Name                           */
```

```
  GList                  *argumente;     /* Argumente                      */
  gint                    ergebnissorte; /* Ergebnissorte                  */
  AUSDRUCK_TYP            rumpf;         /* Rumpf                          */
  gchar                  *eingabe_string;/* Eingabe-String                 */
} RECHENVORSCHRIFT_STRUCT;
typedef RECHENVORSCHRIFT_STRUCT         *RECHENVORSCHRIFT_TYP;
```

Die Daten für eine Rechenvorschrift werden in Speicherbereichen festgehalten, die entsprechend der Struktur `RECHENVORSCHRIFT_STRUCT` interpretiert werden.

In der Komponente `datei_name` wird der Name der Datei gespeichert, aus der die Rechenvorschrift gelesen wurde. Zum Aufnehmen des Namens der Rechenvorschrift dient die Komponente `name`. Die Argumente einer Rechenvorschrift werden in der Liste `argumente` festgehalten. Für die Ergebnissorte steht die Komponente `ergebnissorte` zur Verfügung. Der Rumpf einer Rechenvorschrift ist ein Ausdruck, in dem zusätzlich die Argumentbezeichner der Rechenvorschrift auftauchen dürfen. Also wird dieser erweiterte Ausdruck in der Komponente `rumpf` gespeichert. Der Typ `AUSDRUCK_TYP` braucht hierzu nicht erweitert zu werden, da aus Programmiersicht lediglich ein weiterer Knotentyp beachtet werden muß. In der Komponente `eingabe_string` wird schließlich die Rechenvorschrift als Zeichenreihe genau so festgehalten, wie sie in der Rechenvorschriftendatei spezifiziert wurde. Der Typ `RECHENVORSCHRIFT_TYP`, der abschließend definiert wird, beschreibt einen Zeiger auf die Struktur `RECHENVORSCHRIFT_STRUCT`.

5.4.1.9 Struktur `RV_NAME_SORTE_STRUCT`

```
typedef struct rv_name_sorte_struct
{
  gchar                  *name;          /* Name                           */
  gint                    sorte;         /* Sorte                          */
  gchar                  *eingabe_string;/* Eingabe-String                 */
} RV_NAME_SORTE_STRUCT;
typedef RV_NAME_SORTE_STRUCT         *RV_NAME_SORTE_TYP;
```

Um beim Parsen einer Rechenvorschriftendatei bestimmen zu können, ob auftretende Rechenvorschriftsaufrufe Ergebnisse der richtigen Sorte liefern, gibt es in der Systemkomponente zur Verarbeitung von Programmen (siehe 5.2.2) die Dateien, deren Name mit `kiel_vorprg` beginnt. Sie erzeugen eine Liste von Elementen, die Auskunft über eine Rechenvorschrift, deren Namen und deren Ergebnissorte liefern. Diese Informationen sind in der Struktur `RV_NAME_SORTE_STRUCT` zusammengefaßt. Zur Ansprechung der Daten über Zeiger wird der Typ `RV_NAME_SORTE_TYP` als Zeigertyp auf die Struktur `RV_NAME_SORTE_STRUCT` definiert.

5.4.1.10 Strukturen für Fenster

Exemplarisch wird hier die Struktur für das Befehlsfenster aufgeführt und beschrieben. Für alle anderen Fensterarten existieren ähnliche Strukturen, die den Bestandteilen der jeweiligen Fensterart angepaßt sind.

```
#define BUTTON_MLAUSDRUCK_LADEN      0x01
#define BUTTON_MLAUSDRUCK_EINGEBEN   0x02
#define BUTTON_MLAUSDRUCK_UEBERSICHT 0x03
#define BUTTON_MLPROGRAMM_LADEN      0x04
#define BUTTON_RV_UEBERSICHT         0x05
#define BUTTON_AUSDRUCKBAUM_FENSTER  0x06
#define BUTTON_ZEICHENREIHEN_FENSTER 0x07
#define BUTTON_UNDOREDO_FENSTER      0x08
#define BUTTON_AUSWERTUNGS_FENSTER   0x09
```

```
typedef struct befehlsfenster_daten_struct
{
  GtkWidget                  *fenster;
  gint                        timer_id;
  GtkWidget                  *button_ausdruckbaum,
                             *button_zeichenreihe,
                             *button_adkuebersicht,
                             *button_rvuebersicht,
                             *button_undoredo,
                             *button_auswertung;
  PIXMAP_DRAWINGAREA_DATEN_TYP  pixdraw[5];
} BEFEHLSFENSTER_STRUCT;
typedef BEFEHLSFENSTER_STRUCT *BEFEHLSFENSTER_TYP;
```

Jedes Drücken eines Buttons im Befehlsfenster[2] wird über dieselbe, in der Datei `kiel_gtk_bf.c` stehende, Callback-Funktion `befehlsfenster_button_press_callback` gehandhabt. Dazu ist es notwendig, der Funktion die Information zu geben, welcher Button gedrückt wurde, damit die entsprechende Aktion ausgeführt werden kann. Aus diesem Grund wird zunächst für alle betroffenen Button eine Definition notiert, über die der jeweilige Button identifiziert werden kann.

In der Struktur `BEFEHLSFENSTER_STRUCT` werden alle Komponenten des Befehlsfensters zusammengefaßt, auf die beim Programmablauf zugegriffen werden können soll. Dies sind neben dem eigentlichen Fenster der Timer und alle Toggle-Button, deren Zustand nicht nur vom Benutzer, sondern auch von KIEL geändert werden kann. Beispielsweise wird nach der Eingabe eines korrekten Ausdrucks das Ausdruckübersichtsfenster automatisch geöffnet. Zusätzlich ist in der Struktur noch ein Feld von fünf Zeichenflächen vorgesehen. Es dient zur Aufnahme der fünf Bilder, die die Maustastenbelegung anzeigen. Um nicht die gesamte Struktur von einer Funktion einer anderen übergeben zu müssen, wird der Typ `BEFEHLSFENSTER_TYP` als Zeiger auf diese Struktur definiert, so daß mit seiner Hilfe nur noch ein Zeiger übergeben zu werden braucht.

5.4.1.11 Struktur `POZRAREUO_RUECKGABE_STRUCT`

```
#define POZRAREUO_KEIN_FEHLER               0x00
#define POZRAREUO_FEHLER_FALSCHE_ERGSORTE 0x01
#define POZRAREUO_FEHLER_ANZ_ARG            0x02
#define POZRAREUO_FEHLER_FALSCHE_SORTE     0x03
typedef struct pozRAReuo_RueckgabeStruct
{
  gint   fehler;
  gchar *sorte0,*sorte1;
  gchar *rv_funk;
  gint   anz0,anz1;
  gint   nr;
} POZRAREUO_RUECKGABE_STRUCT;
typedef POZRAREUO_RUECKGABE_STRUCT *POZRAREUO_RUECKGABE_TYP;
```

In der Datei `kiel_adk.c` gibt es die Funktion `pruefe_ob_zu_RV_Aufruf_RV_ex_und_ok`, die nach dem Laden eines Ausdrucks eine Überprüfung durchführt, ob ein Rechenvorschriftsaufruf zu einer geladenen Rechenvorschrift paßt. Sie teilt der aufrufenden Funktion mit, ob die Überprüfung fehlerfrei erfolgte, bzw. welcher Fehler im Fehlerfall aufgetreten ist. Fehler können dabei

- unterschiedliche Ergebnissorten,

[2] Die einzige Ausnahme bildet der Quit-Button, der das Programmende direkt über die Funktion `programm_verlassen_event` einleitet.

- eine voneinander abweichende Anzahl an Argumenten oder
- sich unterscheidende Sorten zweier entsprechender Argumente

sein. Beim Parsen wurde bereits darauf geachtet, daß zu dem Rechenvorschriftsaufruf eine entsprechende Rechenvorschrift existiert. Außerdem wurde bereits ermittelt, welche Sorte die Rechenvorschift liefern muß. Der erste Teilname der Struktur `POZRAREUO_RUECKGABE_STRUCT` bildet sich aus den Anfangsbuchstaben der Wörter im Funktionsnamen `pruefe_ob_zu_RV_Aufruf_RV_ex_und_ok`.

In die Komponente `fehler` wird die Information geschrieben, ob die Überprüfung fehlerfrei geschah, oder welcher Fehler aufgetreten ist. Die übrigen Komponenten übergeben im Fehlerfall der aufrufenden Funktion zusätzliche Daten über die Umstände, die den Fehler verursachten. Mit diesen Daten können dem Benutzer von KIEL über ein Warnungsfenster die Datails des Fehlers mitgeteilt werden. So werden in die Komponenten `sorte0` und `sorte1` gegebenenfalls die jeweiligen Sorten geschrieben, deren Abweichen zu einem Fehler führte. Die Komponente `rv_name` dient zum Aufnehmen des Rechenvorschriftsnamens sowie der Funktionalität der Rechenvorschrift. Sollte sich die Anzahl der Parameter beim Rechenvorschriftsaufruf von der Anzahl der Argumente der entsprechenden Rechenvorschrift unterscheiden, so werden beide in den Komponenten `anz0` und `anz1` festgehalten. Die Komponente `nr` ist für die Aufnahme der Nummer des Arguments vorgesehen, welches beim Vergleich auf Sortenentsprechung einen Fehler verursachte. Auch die Daten einer Struktur vom Typ `POZRAREUO_RUECKGABE_STRUCT` werden über Zeigervariable verarbeitet. Der Typ `POZRAREUO_RUECKGABE_TYP` wird für das Ansprechen solcher Zeigervariablen definiert.

5.4.2 Lokale Strukturen

Die lokalen Strukturen werden jeweils nur von den Funktionen einer Systemkomponente benötigt. Bezüglich dieser sind sie global; bezüglich des gesamten Programms sind sie jedoch lokal.

5.4.2.1 Struktur `ExpansionAuswertungsTyp`

```
typedef struct ExpansionAuswertungsStruct
{
  gint aktuelle_exp_tiefe;
  gint maximale_exp_tiefe;
  gint exp_anzahl;
  gint ausw_anzahl;
} ExpansionAuswertungsTyp;
```

Die Struktur `ExpansionAuswertungsTyp` wird in der Datei `kiel_ausw_priv.h` definiert. Da es nur eine einzige Variable dieser Struktur gibt (siehe 5.5.2), wird hier auf das Ansprechen über eine Zeigervariable verzichtet. In der eben angesprochenen Strukturvariablen werden Daten über die aktuelle Auswertung gespeichert. Neben der aktuellen Tiefe der Expansion, wird auch die größte Expansionstiefe verwaltet, die während der Auswertung bisher auftrat. Außerdem werden die Anzahl der Expansionen und die Anzahl der Auswertungsschritte, die bisher in der aktuellen Auswertung ausgeführt wurden, protokolliert.

5.4.2.2 Struktur `WARNUNGRUECKGABE_STRUCT`

```
typedef struct WarnungRueckgabeStruct
{
  gint nr;
  GtkWidget *fenster;
} WARNUNGRUECKGABE_STRUCT;
typedef WARNUNGRUECKGABE_STRUCT *WARNUNGRUECKGABE_TYP;
```

Es kann der Fall eintreten, daß zu einem Zeitpunkt mehrere Warnungsfenster geöffnet sind. Aus diesem Grund ist es notwendig, zwischen ihnen unterscheiden zu können. Hierzu würde eine Variable vom GTK-Typ `GtkWidget *` genügen. Des weiteren ist in KIEL die Möglichkeit vorgesehen, Warnungsfenster mit einer beliebigen Anzahl an Button zu erzeugen, die nicht als einfache Bestätigung dienen, sondern dem Benutzer einen Eingriff in die Fehlerbehandlung ermöglichen. Warnungsfenster mit mehr als einem Button haben anstelle des anderenfalls vorgesehenen Ausrufungszeichens ein Fragezeichen in ihrem linken Teil abgebildet. Damit das Programm auf die Auswahl des Benutzers reagieren kann, ist es notwendig, die Nummer der Button ebenfalls zu verwalten.

In der Datei `kiel_gtk_wf_priv.h` wird für die Verwaltung des Warnungsfensters und der darin enthaltenen Button die Struktur `WARNUNGRUECKGABE_STRUCT` definiert, die genau diese beiden Informationen in ihren Komponenten aufnehmen kann. Die Tatsache, daß diese Daten über Callback-Funktionen übergeben werden, erfordert das Ansprechen der Struktur über Zeigervariable. Der Typ `WARNUNGRUECKGABE_TYP` wird zu diesem Zweck als Zeigertyp auf eine Struktur des Typs `WARNUNGRUECKGABE_STRUCT` definiert.

5.5 Globale Variable

In dem Programmcode von KIEL sind mehrere globale Variable deklariert. Dabei muß zwischen zwei Arten von ihnen unterschieden werden. Es gibt

- globale Variable **bezüglich des gesamten Programmpaketes**, die von allen Systemkomponenten benutzt werden können, die die jeweilige `.h`-Datei includiert haben, und
- globale Varibale **bezüglich einer Systemkomponente**, von denen alle Funktionen dieser Systemkomponente Gebrauch machen können.

5.5.1 Globale Variable bezüglich des gesamten Programmpaketes

Die globalen Variablen bezüglich des gesamten Programmpaketes können in drei Kategorien eingeteilt werden:

- Fenstervariable,
- Variable zur Festhaltung eines Systemzustandes und
- Ressourcedaten-Variable.

5.5.1.1 Fenstervariable

Die Fenstervariablen sind Zeiger auf speziell für das jeweilige Fenster angelegte Strukturen (siehe 5.4.1.10). Im folgenden sind diese zusammen mit dem Namen der Datei, in der sie deklariert werden, aufgeführt.

- in `kiel_gtk_aef.h`:

  ```
  AUSDRUCKEINGABEFENSTER_TYP ausdruckeingabefenster;
  ```

- in `kiel_gtk_awf.h`:

  ```
  AUSWERTUNGSFENSTER_TYP auswertungsfenster;
  ```

- in `kiel_gtk_bf.h`:

```
BEFEHLSFENSTER_TYP befehlsfenster;
```

- in kiel_gtk_bmf.h:

```
BAUMFENSTER_TYP baumfenster;
```

- in kiel_gtk_uef.h:

```
UEBERSICHTSFENSTER_TYP rvuebersichtsfenster;
UEBERSICHTSFENSTER_TYP adkuebersichtsfenster;
```

- in kiel_gtk_urf.h:

```
UNDOREDOFENSTER_TYP undoredofenster;
```

- in kiel_gtk_zrf.h:

```
ZEICHENREIHENFENSTER_TYP zeichenreihenfenster;
```

5.5.1.2 Variable zur Festhaltung eines Systemzustandes

In der Datei kiel_gtk_wf.h steht die folgende Deklaration:

```
gint radio_auswahl_bei_warnung_ausw_art,radio_auswahl_bei_warnung_schritte;
```

Die beiden Variablen dienen zum Festhalten der getroffenen Auswahl in dem Auswertungsfenster (siehe 3.3.8) bezüglich der Art der Auswertung und der Anzahl der auszuführenden Auswertungsschritte. In sie werden diese Informationen vor dem Öffnen eines Warnungsfensters geschrieben, um diese Werte nach dem Schließen des Warnungsfensters sofort wieder übernehmen zu können.

5.5.1.3 Ressourcedaten-Variable

Die letzte Kategorie bilden die globalen Variablen, die die Daten aus der Ressourcedatei kiel.rc aufnehmen. Sie sind in der Datei kiel_ressource.h deklariert.

Auszug aus der Datei kiel_ressource.h — Anfang

```
/* globale Variable, die durch die Ressourcedatei kiel.rc zur Laufzeit */
/* gesetzt werden */

/* Fensterpositionen */
gint POSITION_BEFEHLSFENSTER_X,POSITION_BEFEHLSFENSTER_Y,
     POSITION_BAUMFENSTER_X,POSITION_BAUMFENSTER_Y,
     POSITION_ZEICHENREIHENFENSTER_X,POSITION_ZEICHENREIHENFENSTER_Y,
     POSITION_ADKUEBERSICHTSFENSTER_X,POSITION_ADKUEBERSICHTSFENSTER_Y,
     POSITION_RVUEBERSICHTSFENSTER_X,POSITION_RVUEBERSICHTSFENSTER_Y,
     POSITION_UNDOREDOFENSTER_X,POSITION_UNDOREDOFENSTER_Y,
     POSITION_ADKFILESELECTION_X,POSITION_ADKFILESELECTION_Y,
     POSITION_PRGFILESELECTION_X,POSITION_PRGFILESELECTION_Y,
     POSITION_AUSDRUCKEINGABEFENSTER_X,POSITION_AUSDRUCKEINGABEFENSTER_Y,
     POSITION_AUSWERTUNGSFENSTER_X,POSITION_AUSWERTUNGSFENSTER_Y;

/* Groessen */
gint BREITE_DRAWING_AREA_BAUM,
     HOEHE_DRAWING_AREA_BAUM,
```

```
     BREITE_SCROLLEDWINDOW_ZR,
     BREITE_AUSDRUCKEINGABEFENSTER_ENTRY,
     BREITE_UEBERSICHTS_SCROLLED_WIN,
     HOEHE_UEBERSICHTS_SCROLLED_WIN,
     DURCHMESSER_NACHKOMMASTELLEN_DIAL;

/* Nachkommastellen */
gint VORGABE_NACHKOMMASTELLEN,
     MAXIMALE_NACHKOMMASTELLEN,
     VORGABE_ANZAHLZEICHENIMSTRING,
     MAXIMALE_ANZAHLZEICHENIMSTRING,
     VORGABE_ANZAHLELEMENTEINLISTE,
     MAXIMALE_ANZAHLELEMENTEINLISTE;

/* Begrenzungen */
gint ABBRUCH_EXPANSIONS_TIEFE,
     MAX_ANZUZEIGENDE_EBENEN_ANZ;

/* Verzeichnisse */
gchar *WINDOW_AUSDRUCK_DATEIAUSWAHL_DIRECTORY,
      *WINDOW_PROGRAMM_DATEIAUSWAHL_DIRECTORY;

/* Status der Toggle-Buttons */
gboolean BUTTON_MLAUSDRUCK_UEBERSICHT_STATUS,
         BUTTON_RV_UEBERSICHT_STATUS,
         BUTTON_AUSDRUCKBAUM_FENSTER_STATUS,
         BUTTON_ZEICHENREIHEN_FENSTER_STATUS,
         BUTTON_UNDOREDO_FENSTER_STATUS,
         BUTTON_AUSWERTUNGS_FENSTER_STATUS;

/* Status der Radio-Buttons */
gboolean BUTTON_LI_STATUS,
         BUTTON_LO_STATUS,
         BUTTON_FULL_STATUS;
gboolean BUTTON_1_SCHRITT_STATUS,
         BUTTON_5_SCHRITTE_STATUS,
         BUTTON_10_SCHRITTE_STATUS,
         BUTTON_VAR_SCHRITTE_STATUS,
         BUTTON_BIS_FERTIG_STATUS;

/* Grafik-Konstanten */
gint K_OffsetH,
     K_OffsetV,
     K_Zeilenhoehe,
     K_KnotenMindestabstand;
gdouble K_KnotenBreite_zu_SchriftBreite,
        K_KnotenHoehe_zu_SchriftHoehe;
```

Auszug aus der Datei `kiel_ressource.h` — Ende

Alle diese Variablen können über Einträge in der Ressourcedatei `kiel.rc` (siehe 3.4.1) festgelegt werden.[3] Die Funktion `ressourcen_einlesen`, die zu Beginn der Funktion `GTK_Schleife` (siehe 5.2.6) aufgerufen wird, weist den globalen Variablen die vom Benutzer gesetzten Werte zu, so daß während der weiteren Ausführung von KIEL diese Vorgaben benutzt werden können.

[3] Das Ändern der Ressourcedatei `kiel.rc` ist dem Benutzer nicht möglich. Stattdessen kann er in der privaten Ressourcedatei `.kiel.rc` beliebig Veränderungen vornehmen und so auf die Werte der hier behandelten Variablen Einfluß nehmen. Näheres hierzu findet sich in 3.4.1.

5.5.2 Globale Variable bezüglich einer Systemkomponente

In den privaten Headerdateien einiger Systemkomponenten bzw. Teilkomponenten werden globale Variable bezüglich der jeweiligen Systemkomponente bzw. Teilkomponente deklariert.

- In der Datei `kiel_adk_priv.h` steht die folgende Deklaration:

  ```
  gboolean letzter_ausdruck_fehlerhaft;
  ```

  Kommt es beim Parsen von Ausdrücken zu einem Fehler, wird dieses Auftreten durch die Variable `letzter_ausdruck_fehlerhaft` der Funktion zum Einlesen von Ausdrücken mitgeteilt.

- Während jeder Auswertung protokolliert KIEL einige wichtige Informationen zur aktuellen Auswertung. Die Variable `expansionauswertungs_daten` wird in der Datei `kiel_ausw_priv.h` deklariert:

  ```
  ExpansionAuswertungsTyp expansionauswertungs_daten;
  ```

  Die protokollierten Informationen sind in 5.4.2.1 beschrieben.

- Die `kiel_debug`-Funktionen, welche in 5.7.4 genauer beschrieben sind, benutzen die in der Headerdatei `kiel_fkt_priv.h` durch

  ```
  gint einrueck_debug_puts=0;
  ```

  deklarierte Variable `einrueck_debug_puts` zum Festhalten der aktuellen Ebene während der Programmausführung. Initial hat diese den Wert Null, da vor Beginn der Programmausführung mit der Abarbeitung noch keiner Funktion begonnen wurde.
  Zusätzlich sind in der Datei `kiel_fkt.c`[4] die Variablen

  ```
  gint kiel_debug_erwuenscht_ab_ebene=100;
  gboolean kiel_debug_flex_erwuenscht=FALSE;
  gboolean kiel_debug_ausw_erwuenscht=FALSE;
  ```

  deklariert. Durch sie kann der Entwickler dem System mitteilen, ob und in welcher Art Debugausgaben erwünscht sind.

- In der Teilkomponente zur Verwaltung des Baumfensters sind die folgenden Variablen in der Datei `kiel_gtk_bmf_priv.h` deklariert:

  ```
  AUSDRUCKBAUM_TYP abaum_fuer_geoeffnetes_popup=NULL;
  AUSDRUCK_TYP adk_fuer_geoeffnetes_popup=NULL;
  AUSDRUCKBAUM_TYP zuletzt_bearbeitet_abaum=NULL;
  gint zuletzt_bearbeitet_x,zuletzt_bearbeitet_y;
  gboolean baumfenster_groessenaenderung_zulassen=TRUE;
  gboolean bin_in_adjustment_value_changed_callback=FALSE;
  ```

  Die Variablen `abaum_fuer_geoeffnetes_popup` und `adk_fuer_geoeffnetes_popup` werden für die Einstellung der Anzahl der Nachkommastellen bei reellen Knoten, der Anzahl der Zeichen bei Zeichenreihenknoten bzw. der Anzahl der Elemente bei Sequenzknoten benötigt. Die Einstellungen, die in der Wählscheibe getroffen werden, können mit ihrer Hilfe dem richtigen Knoten des richtigen Ausdrucks zugeordnet werden.

[4]Hier wird von der Vorgabe abgewichen, globale Variable bezüglich einer Systemkomponente in den `_priv.h`-Dateien zu deklarieren. Der Grund für diese Maßnahme ist, daß diese Variablen sehr eng mit den im Anschluß definierten Funktionen zusammenhängen.

Der zuletzt bearbeitete Knoten soll im Baumfenster hervorgehoben werden (siehe 3.3.6). Daher muß KIEL beim Zeichnen eines Baumes ermitteln können, welcher Knoten als letzter bearbeitet wurde. Die Variable `zuletzt_bearbeitet_abaum` speichert genau diese Information.

Der gleiche Knoten soll zusätzlich möglichst an derselben Stelle im Baumfenster erscheinen, an der er vor der Auswertung stand. Deshalb werden die Koordinaten seiner Ursprungsposition in den Variablen `zuletzt_bearbeitet_x` und `zuletzt_bearbeitet_y` festgehalten. Mit ihrer Hilfe kann später die gewünschte Stellung der Scrollbalken im Baumfenster eingestellt werden.

Während einer Auswertung darf die Größe des Baumfensters nicht geändert werden. Ob eine Größenänderung des Baumfensters erlaubt ist, wird den Funktionen, die diese Information benötigen, durch die Variable `baumfenster_groessenaenderung_zulassen` mitgeteilt.

Die Funktion `adjustment_value_changed_callback` ist eine Callback-Funktion, die immer dann aufgerufen wird, wenn sich der Wert eines Scrollbalkens im Baumfenster geändert hat. Sie veranlaßt KIEL, den Inhalt des Baumfensters neu darzustellen. In der Funktion `baumfenster_configure_event`, die hierfür zuständig ist, werden normalerweise die Einstellungen der beiden Scrollbalken vorgenommen. Dies würde wiederum zu einem Aufruf der oben erwähnten Callback-Funktion führen, woraus ein ungewolltes, andauernd sich wiederholendes Neuzeichnen des Baumfensters resultierte. Aus diesem Grund werden die Einstellungen an den Scrollbalken im Baumfenster nur dann vorgenommen, wenn der Aufruf der Funktion `baumfenster_configure_event` nicht in der Callback-Funktion `adjustment_value_changed_callback` stattgefunden hat. Um diese Konstellation erkennen zu können, gibt es die boolesche Variable `bin_in_adjustment_value_changed_callback`, die zu Beginn der Ausführung von `adjustment_value_changed_callback` auf den Wert `TRUE` und an ihrem Ende auf den Wert `FALSE` gesetzt wird. Sie ermöglicht es der Funktion `baumfenster_configure_event` festzustellen, ob sie aus dieser Callback-Funktion aufgerufen wurde.

- Es gibt ein Dateiauswahlfenster zum Laden von Ausdruckdateien und eines zum Laden von Rechenvorschriftendateien. Da es von ihnen je nur eins gibt, beide aber zur selben Zeit geöffnet sein können, benötigen beide genau eine Variable vom GTK-Typ `GtkWidget *`. In der Datei `kiel_gtk_df_priv.h` steht daher die folgende Deklaration.

```
GtkWidget *adk_file_selection_widget;
GtkWidget *prg_file_selection_widget;
```

- Die `gtk`-Systemkomponente verwaltet das Zusammenspiel ihrer einzelnen Teilkomponenten genauso, wie sie auch zwei für KIEL sehr wichtige Variable zur Verfügung stellt. Der Datei `kiel_gtk_priv.h` sind die Zeilen

```
GList *MLAusdruecke;
GList *MLRechenvorschriften;
gboolean ist_auswertungs_abbruch_erwuenscht=FALSE;
gint anz_warnungs_fenster_geoeffnet=0;
gboolean trotz_warnung_=FALSE;
```

entnommen. `MLAusdruecke` und `MLRechenvorschriften` sind die eben angesprochenen Variablen. Beide sind Zeiger auf den GLib-Listentyp `GList`. In ihnen werden alle dem System zum jeweiligen Zeitpunkt bekannten Ausdrücke bzw. Rechenvorschriften gespeichert.

Der Wert der booleschen Variablen `ist_auswertungs_abbruch_erwuenscht` wird `TRUE`, sobald der Benutzer während einer Auswertung auf den Button zum Abbruch der Auswertung im Auswertungsfenster (siehe 3.3.8) klickt. Ein solcher Auswertungsabbruch soll nämlich nicht sofort, sondern erst nach Ende der Ausführung des aktuellen Auswertungsschrittes vollzogen werden. So wird vor jeder Ausführung eines Auswertungsschrittes geprüft, ob diese Variable einen geforderten Abbruch signalisiert.

In der Variablen `anz_warnungs_fenster_geoeffnet` wird die Anzahl der insgesamt geöffneten Warnungsfenster mitgezählt.

Wenn mindestens ein Warnungsfenster geöffnet ist, kann der Benutzer nicht mit KIEL weiterarbeiten. Er muß zunächst alle Warnungen bestätigen. Unter gewissen Umständen sollen dennoch einige Funktionen ausführbar sein. Wird zum Beispiel beim Laden einer Ausdruckdatei ein Fehler erkannt, dann soll das Ausdruckübersichtsfenster ggf. geöffnet werden, damit der Benutzer einen Überblick über die bisher im System befindlichen Ausdrücke hat. Dieser Effekt kann durch Arbeiten mit der booleschen Variablen `trotz_warnung_` erreicht werden.

- Die Datei `kiel_gtk_wf_priv.h` enthält die Deklarationen

```
static gchar *ausrufungszeichen[] = {
/* Breite Hoehe anz_Farben Zeichen_je_Pixel */
"    16    52      2             1",
/* Farben */
"       c None",
"#      c #000000000000",
/* Pixel */
"                ",
"      ####      ",
"     ######     ",
"    ########    ",
"   ##########   ",
"  ############  ",
"  ############  ",
" ############## ",
" ############## ",
" ############## ",
" ############## ",
" ############## ",
"  ############  ",
"  ############  ",
"  ############  ",
"  ############  ",
"   ##########   ",
"   ##########   ",
"   ##########   ",
"   ##########   ",
"    ########    ",
"    ########    ",
"    ########    ",
"    ########    ",
"    ########    ",
"     ######     ",
"     ######     ",
"     ######     ",
"     ######     ",
"      ####      ",
"      ####      ",
"      ####      ",
"       ##       ",
"       ##       ",
"       ##       ",
"                ",
"                ",
"                ",
```

```
"      ####      ",
"    ########    ",
"   ##########   ",
"  ############  ",
"  ############  ",
" ############## ",
" ############## ",
" ############## ",
" ############## ",
"  ############  ",
"  ############  ",
"   ##########   ",
"    ########    ",
"      ####      ",
"                "
};
static gchar *fragezeichen[] =  {
/* Breite Hoehe anz_Farben Zeichen_je_Pixel */
"    33    52      2             1",
/* Farben */
"       c None",
"#      c #000000000000",
/* Pixel */
"                                 ",
"         ############            ",
"       ################          ",
"      ###################        ",
"     ####        ##########      ",
"    #####         ###########    ",
"   #######        ############   ",
"  #########        ###########   ",
" ###########        ###########  ",
" ###########        ###########  ",
" ###########         ########### ",
" ###########         ########### ",
"  #########          ########### ",
"   #######            ########## ",
"    #####             ########## ",
"                      ########## ",
"                     ##########  ",
"                     ##########  ",
"                     #########   ",
"                    #########    ",
"                    ########     ",
"                   ########      ",
"                  ########       ",
"                 ########        ",
"                ########         ",
"               ########          ",
"              #######            ",
"             ######              ",
"            ######               ",
"           #######               ",
"          ######                 ",
"         ######                  ",
"         ######                  ",
```

```
"         ######                  ",
"                                 ",
"                                 ",
"                                 ",
"          ####                   ",
"        ########                 ",
"       ##########                ",
"      ############               ",
"      ############               ",
"     ##############              ",
"     ##############              ",
"     ##############              ",
"     ##############              ",
"      ############               ",
"      ############               ",
"       ##########                ",
"        ########                 ",
"          ####                   ",
"                                 "
};
gint letzte_warnung_auswahl;
```

Die beiden Felder **ausrufungszeichen** und **fragezeichen** beinhalten Pixmapdefinitionen für ein 16 mal 52 Pixel großes Ausrufungszeichen bzw. ein 33 mal 52 Pixel großes Fragezeichen. In Abhängigkeit der Anzahl der darzustellenden Buttons in einem Warnungsfenster wird eines der beiden Pixmaps im linken Teil des Warnungsfensters dargestellt.

In der Variablen `letzte_warnung_auswahl` wird die Nummer des zuletzt gedrückten Buttons eines Warnungsfensters festgehalten. Mit dieser Information kann KIEL gemäß den Wünschen des Benutzers den weiteren Programmablauf gestalten.

- Für die Systemkomponente zur Speicherverwaltung werden in der Datei `kiel_mem_priv.h` die Deklarationen

```
gint belegter_Speicher_durch_mtmalloc=0;
gboolean mtmem_belegt[max_anz_speicherplaetze];
voidp mtmem_adresse[max_anz_speicherplaetze];
gint mtmem_groesse[max_anz_speicherplaetze];
```

vorgenommen. Sie sind alle Bestandteil der mt-Funktionen und werden mit diesen zusammen in 5.7.6 detailliert beschrieben.

- Im Vergleich zum Einlesen von Ausdrücken gestaltet sich das Einlesen von Rechenvorschriften etwas schwieriger. Daher werden auch vergleichsweise viele globale Variable bezüglich der Systemkomponente zur Verarbeitung von Programmen in der Datei `kiel_prg_priv.h` deklariert.

```
gboolean  letztes_programm_fehlerhaft;
GList    *letzte_argumentliste;
gchar    *programm_aktueller_rv_name;
gint      programm_letzte_sorte;
GList    *bisherige_RVs;
gboolean  beim_parsen_einer_argumentliste;
GList    *rv_name_sorte_liste_liste;
```

Falls während des Parsens einer Rechenvorschriftendatei ein Fehler entdeckt wird, kann dies anderen Funktionen durch die Variable `letztes_programm_fehlerhaft` mitgeteilt werden.

Da im Rumpf einer Rechenvorschrift auch die Bezeichner der Parameter auftauchen dürfen, ist es nötig, beim Parsen des Rumpfes jederzeit auf eine Liste der Parameter zugreifen zu können. Nur so kann bestimmt werden, ob ein Bezeichner zulässig ist. Beim Parsen einer Argumentenliste wird die Listenvariable `letzte_argumentliste` gefüllt und stellt beim Parsen des folgenden Rumpfes die nötigen Informationen zur Verfügung.

Der Bezeichner der aktuellen Rechenvorschrift darf ebenfalls im Rumpf erscheinen — genau dies ist direkte Rekursion. Damit diese für funktionale Programmiersprachen bedeutende Eigenschaft vom Parser kontrolliert werden kann, wird in der Variablen `programm_aktueller_rv_name` der Name der Rechenvorschrift, die zum jeweiligen Zeitpunkt bearbeitet wird, festgehalten.

Außer der Überprüfung des Namens von Bezeichnern, ist es ebenfalls wichtig zu wissen, welcher Sorte die Bezeichner angehören. Nur so kann der Parsevorgang korrekt weitergeführt werden. Die Variable `programm_letzte_sorte` speichert die Sorte der aktuellen Rechenvorschrift. Diese wird nämlich erst nach Beendigung des Parsens in eine Liste von Rechenvorschriften[5] geschrieben. Durch die Variable `programm_letzte_sorte` wird somit die Sortenprüfung für direkte Rekursion ermöglicht.

In die eben schon angesprochenen Liste `bisherige_RVs` werden die Rechenvorschriften geschrieben, sobald sie komplett geparst wurden. Da im Rumpf einer Rechenvorschrift auf alle bereits deklarierten Rechenvorschriften zurückgegriffen werden kann, ist diese Liste wichtig, um im Rumpf auftretende Bezeichner zu überprüfen und um auf ihre Sorte zugreifen zu können. Im Gegensatz zu der weiter unten beschriebenen Variablen `rv_name_sorte_liste_liste` sind in dieser Liste alle Informationen der Rechenvorschriften verfügbar.

Während des Parsens einer Argumentenliste stößt der Parser in der Regel auf unbekannte Bezeichner. Diese Tatsache ist erwünscht, da dies die Bezeichner der Parameter sein können. Vorkommen solcher unbekannten Bezeichner sind hier im Gegensatz zum Scannen und Parsen eines Rechenvorschriftsrumpfes ganz anders zu behandeln. Durch den Inhalt der Variablen `beim_parsen_einer_argumentliste` wird dem Scanner mitgeteilt, daß sich der Parser bei der Bearbeitung einer Argumentenliste befindet.

In die Variable `rv_name_sorte_liste_liste` wird eine Liste von Listen von Strukturen geschrieben, die einen Rechenvorschriftsnamen und die dazugehörige Ergebnissorte aufnehmen. In sie werden vor dem Parsen durch Funktionen der `kiel_vorprg`-Dateien alle Rechenvorschriftensysteme eingetragen. Da diese Systeme aus einer Liste von Rechenvorschriften bestehen, kommt es zu dem komplexeren Typ dieser Variablen. Mit ihrer Hilfe können auch Bezeichner von Rechenvorschriften akzeptiert werden, die in dem gleichen Rechenvorschriftensystem wie die aktuelle Rechenvorschrift stehen, jedoch noch nicht geparst wurden, da sie hinter ihr notiert sind.

5.6 Das Baumfenster

Das Baumfenster ist in KIEL von erheblicher Bedeutung. Es dient sowohl als Hauptanzeigefenster als auch als Schnittstelle, über die der Benutzer die Auswertung eines Ausdrucks vornimmt. In diesem Abschnitt wird auf die Darstellung eines Baumes, sowie auf den internen Aufbau des Baumfensters eingegangen. Das Reagieren auf Benutzereingaben über das Baumfenster ist Thema von 5.7.5. Dort wird die Funktion `untersuche_mausklick` aufgeführt und im Detail beschrieben.

Zur Darstellung des Baumes wird die Zeichenfläche intern in Zeilen eingeteilt, in denen die Knoten später plaziert werden. Hierdurch wird es ermöglicht, die Ebenen eines Baumes übersichtlich darzustellen. Für die horizontale Einteilung wird als Einheit ein Bildpunkt genommen. So kann auf die verschiedenen Breiten der Knoten optimal eingegangen werden.

[5] Diese Liste wird durch die Variable `bisherige_RVs` repräsentiert.

5.6.1 Grafikkonstante

Zur Festlegung des Erscheinungsbildes eines Knotens im Baumfenster (siehe 3.3.6) gibt es in KIEL die Konstanten

- `K_OffsetH`,
- `K_OffsetV`,
- `K_Zeilenhoehe`,
- `K_KnotenMindestabstand`,
- `K_KnotenBreite_zu_SchriftBreite` und
- `K_KnotenHoehe_zu_SchriftHoehe`.

Ihr Wert wird durch die Werte von

- `GK_OFFSET_H`,
- `GK_OFFSET_V`,
- `GK_ZEILENHOEHE`,
- `GK_KNOTEN_MINDESTABSTAND`,
- `GK_KNOTENBREITE_ZU_SCHRIFTBREITE` und
- `GK_KNOTENHOEHE_ZU_SCHRIFTHOEHE`

aus der Ressourcedatei `kiel.rc` (siehe 3.4.1) festgelegt.

In der Abbildung 5.3 ist die Bedeutung der oben aufgeführten Konstanten in bezug auf das Erscheinungsbild im Baumfenster verdeutlicht.

5.6.2 Der verwendete Baumzeichenalgorithmus

Nach der Festlegung des Erscheinungsbildes der Knoten in 5.6.1 wird nun auf die Positionierung der Knoten im Baumfenster eingegangen. Es werden einige Anforderungen an den Positionierungsalgorithmus gestellt. Er soll die Knoten eines beliebigen Baumes so auf der Zeichenfläche plazieren, daß

1) Knoten, die denselben Abstand[6] von der Wurzel haben, in einer Zeile dargestellt werden,

2) ein Knoten stets mittig über seine Kinder gesetzt wird und

3) die Ausgabe möglichst wenig Platz einnimmt.

[6] Der Abstand eines Knotens a von einem Knoten b ist die Länge eines (bei einem Baum eindeutigen) kürzesten Weges im Graphen von a nach b.

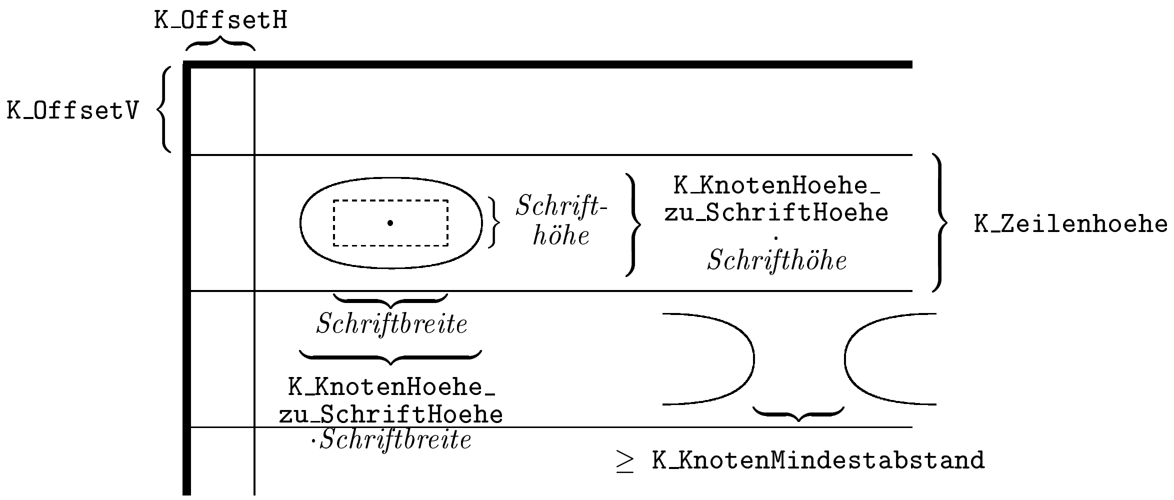


Abbildung 5.3: Die Grafikkonstanten

Die Zuordnung der Knoten zu den Zeilen geschieht auf kanonische Art und Weise. Die Funktion *zeile* von der Menge der Knoten eines Baumes in die Menge der natürlichen Zahlen hat folgende Gestalt:

$$zeile(knoten) \quad = \quad \begin{cases} 0 & : \text{ falls } knoten \text{ die Wurzel des Baumes ist} \\ zeile(vater)+1 & : \text{ sonst, } \quad \text{wobei } vater \text{ der eindeutig bestimmte} \\ & \qquad\qquad\quad \text{Vaterknoten von } knoten \text{ ist.} \end{cases}$$

Die horizontale Zuordnung der Knoten ist komplizierter. In der Systemkomponente zur Grafikausgabe auf dem Bildschirm (siehe 5.2.5) wird eine Funktion definiert, die die Breite eines Knotens ermittelt. Dieses Hilfsmittel ist zur Feststellung der horizontalen Ausrichtung sehr nützlich.

Die Anforderungen 2) und 3) sind für sich gesehen recht einfach zu erfüllen. Für 2) wird die Zeichenfläche in gleichgroße Spalten aufgeteilt, die den breitesten Knoten des Baumes aufnehmen können. Nun wird für jeden Knoten eine Spalte vorgesehen. Bei einem Tiefensuche-Durchlauf durch den Baum werden den Knoten reelle Zahlen zugeordnet, anhand welcher die Zuordnung der Knoten zu den Spalten festgelegt wird. Um 3) zu genügen, werden alle Knoten einer Zeile, am linken Bildrand beginnend, direkt aufeinanderfolgend gesetzt.

Erwünscht ist ein Plazierungsalgorithmus, der 2) und 3) erfüllt. Hierzu wird der Algorithmus von Wetherell und Shannon [WetSha79] herangezogen, der dieses Problem für binäre Bäume in einem festen Raster behandelt, wobei jeder Knoten eine Einheitsbreite besitzt. Um ein akzeptables Ergebnis für KIEL zu erzielen, wurde dieser Algorithmus dahingehend verbessert, daß er auf beliebige Bäume angewendet werden kann und jeder Knoten eine individuelle Breite besitzt.

Die Breite der Darstellung wird offensichtlich dann minimal, wenn in einer Zeile benachbarte Knoten direkt nebeneinander auf die Zeichenfläche gesetzt werden. Da dies zu keiner übersichtlichen Baumdarstellung führt, wird die Anforderung 3) dahingehend angepaßt. Die Knoten des Baumes sollen so plaziert werden, daß neben 1) und 2) auch

3') die Ausgabe möglichst wenig Platz einnimmt, zwei in einer Zeile benachbarte Knoten jedoch einen festen Minimalabstand voneinander haben.

Wie in 5.6.1 dargestellt, ist der horizontale Mindestabstand zweier Knoten durch die Grafikkonstante `K_KnotenMindestabstand` festgelegt. Der Benutzer hat die Möglichkeit, den Mindestabstand durch Ändern des Wertes `GK_KNOTEN_MINDESTABSTAND` in der Ressourcedatei `kiel.rc` (siehe 3.4.1) selbst zu setzen.

Im folgenden wird der modifizierte Algorithmus, der 1), 2) und 3') genügt, vorgestellt. Hierzu wird angenommen, daß ein Knoten durch eine Struktur repräsentiert wird, in der die Komponenten

- `x`,
- `modifier` und
- `kinder`

vorhanden sind.[7] Die Komponenten `x` und `modifier` sind reellwertig, und die Komponente `kinder` ist eine Liste, die beliebig viele Knoten aufnehmen kann.

Sei t ein beliebiger Ausdrucksbaum, der in obiger Struktur vorliegt. Sei *anz_ebenen* die Anzahl der Ebenen in t. Seien *modifier* und *next_pos* zwei Felder aus je *anz_ebenen* reellen Variablen, die initial alle den Wert Null haben.

Der eigentliche horizontale Positionierungsalgorithmus gliedert sich in den **first_pass** und den **modifying_pass** und setzt in der Eingabe t die Komponenten `x` und `modifier`. Der Algorithmus zur horizontalen Positionierung der Knoten hat folgendes Aussehen:[8]

```
horizontale_Positionierung
   first_pass(t);
   modifying_pass(t);
```

Der **first_pass** übernimmt die Hauptarbeit des Positionierungsalgorithmus. Hier werden die Positionen aller Knoten festgelegt, die jedoch unter Umständen im **modifying_pass** noch modifiziert werden müssen.

In dem **first_pass** gibt es eine lokale Ganzzahlvariable *zeile* und eine reelle Variable *place*.

```
   first_pass(knoten)
1:    for all knoten.kinder do first_pass[9];
      ebene:=Zeile von knoten;
      if |knoten.kinder| = 0 then
2:       knoten.x := next_pos[ebene] + Breite von knoten/2;
      else
3:       place := arithmetische Mittelwert der x-Komponenten aller Kinder von knoten;
         modifier[ebene] := max(modifier[ebene], next_pos[ebene] - place + Breite von knoten/2);
         knoten.x := place + modifier[ebene];
      fi
4:    next_pos[ebene] := knoten.x + Breite von knoten/2 + Knotenmindestabstand;
      knoten.modifier := modifier[ebene];
```

[7] Die in KIEL verwendete Struktur für die Knoten ist `AUSDRUCKBAUM_STRUCT` (siehe 5.4.1.3). In ihr sind die drei angegebenen Komponenten ebenfalls vorhanden.

[8] In den Algorithmen dieses Unterabschnittes sind die Algorithmusnamen **fett**, Variablen- und Parameternamen *kursiv* und umgangssprachliche Formulierungen serifenlos. Bezeichner der gewählten Pseudoprogrammiersprache sowie Komponentennamen sind in `Schreibmaschinenschrift` gesetzt.

[9] Das Argument dieses Aufrufs von **first_pass** ist das jeweilige Listenelement des Schleifendurchlaufs.

ad 1: In dem **first_pass**-Teil des horizontalen Positionierungsalgorithmus werden alle Knoten des Baumes nach der Tiefensuche-Strategie besucht und bearbeitet. Aus diesem Grund stellt die erste Anweisung von **first_pass** den Rekursionsschritt dar.

ad 2: Ist der aktuelle Knoten ein Blatt, so wird er an der nächstmöglichen Stelle in der entsprechenden Ebene plaziert. In der `x`-Komponente des Knotens soll die Position des Knotenmittelpunktes festgehalten werden. Deshalb wird die halbe Knotenbreite addiert.

ad 3: Ist der aktuelle Knoten kein Blatt, so hat er mindestens ein Kind. Er soll mittig über seinen Kindern plaziert werden. Dazu wird in der Variablen *place* der arithmetische Mittelwert der Positionen seiner Kinder gespeichert, die aufgrund des Tiefensuche-Durchlaufs bereits bekannt sind.

In dem Variablenfeld *modifier* wird zu jeder Ebene gespeichert, um wie viele Pixel[10] die Kinder des aktuellen Knotens nach rechts verschoben werden müssen, um keine Überschneidung des aktuellen Knotens mit Knoten derselben Ebene hervorzurufen. Zu Beginn des Algorithmus haben alle Komponenten den Wert Null, da noch keine Positionsmodifizierungen zu beachten sind.

Der aktuelle Knoten wird mittig über seinen Kindern plaziert, wobei die Positionsmodifizierung in die Berechnung einzubeziehen ist. Könnte der Knoten aufgrund der Positionen seiner Kinder weiter links plaziert werden als die nächste freie Stelle der aktuellen Ebenen, wird das entsprechende Feld in *modifier* so erhöht, daß die Positionierung an der nächsten freien Stelle stattfindet.

ad 4: Die nächste freie Stelle in der aktuellen Ebene ergibt sich als Summe des soeben ermittelten Mittelpunktes des aktuellen Knotens, seiner halben Breite sowie dem Mindestabstand, den zwei Knoten horizontal voneinander haben sollen.

In der Komponente `modifier` des aktuellen Knotens wird festgehalten, um wie viele Pixel seine Position modifiziert wurde.

Der **modifying_pass** dient zum Modifizieren der Knotenpositionen in solchen Fällen, wo Vorfahren wegen Platzmangels nicht mittig über den Kindern plaziert werden konnten, sondern nach rechts verschoben werden mußten. Jeder Knoten wird um die Summe der Modifizierungen aller seiner Vorfahren nach rechts verschoben.

Für den **modifying_pass** wird eine globale reelle Variable *modifier_sum* deklariert, die zum Aufsummieren der `modifier`-Komponente gewisser Knoten benutzt wird.

```
   modifying_pass(knoten)
1:    knoten.x := knoten.x + modifier_sum;
      modifier_sum := modifier_sum + knoten.modifier;
2:    for all knoten.kinder do modifying_pass[11];
      modifier_sum := modifier_sum - knoten.modifier;
```

ad 1: Die Komponente `x` des aktuellen Knotens wird um den Wert der Variablen *modifier_sum* erhöht.

ad 2: In dem Rekursionsteil von **modifying_pass** wird bei allen Kindern der Modifikationsschritt durchgeführt. Vor Beginn der Rekursion wird die Variable *modifier_sum* um den Wert der Komponente `modifier` des aktuellen Knotens erhöht; nach Beendigung der Rekursion wird diese Variable um denselben Wert verringert. Dies führt dazu, daß bei der Rekursion in der Variablen *modifier_sum* stets die Summe der `modifier`-Komponenten aller Vorfahren des jeweiligen Knotens steht.

[10] In dem Variablenfeld *modifier* wird die Anzahl der Pixel reellwertig festgehalten. Dies ist a priori ein Widerspruch, da die Anzahl stets ganzzahlig sein sollte. Da sich jedoch sofort vorgenommene Rundungen am Ende des Algorithmus zu größeren Fehlern addieren können, wird zunächst reellwertig gerechnet und die Rundung erst an der Anwendungsstelle vorgenommen.

[11] Das Argument dieses Aufrufs von **modifying_pass** ist das jeweilige Listenelement des Schleifendurchlaufs.

5.6.3 Das Scrolled-Window im Baumfenster

GTK bietet die Möglichkeit, mit einem einzigen Befehl ein Scrolled-Window zu erstellen. Von dieser Möglichkeit wird im Zeichenreihenfenster Gebrauch gemacht. In ein erzeugtes Scrolled-Window wird hier einfach ein Label gesetzt, in das später die gewünschte Zeichenreihe geschrieben wird. Dabei kümmert sich GTK um die Einstellung des Scrollbalkens. Verschiebt der Benutzer den Scrollbalken, so regelt GTK automatisch das Neuzeichnen des Labels.

Diese weitgehende Arbeitsabnahme seitens GTK hat aber auch einen Nachteil, der beim Scrolled-Window im Baumfenster deutlich wird. Es ist möglich, GTK ein Scrolled-Window erzeugen zu lassen und dort hinein eine Zeichenfläche zu setzen, auf der im Anschluß ein Baum dargestellt werden kann. Der Programmierer braucht sich in diesem Fall fast keine Gedanken zu machen, da alle gängigen Funktionalitäten von GTK gehandhabt werden.

Dieser Ansatz funktioniert aber nur, solange die benötigte Zeichenfläche nicht zu groß wird. Was genau „zu groß" in diesem Zusammenhang bedeutet, hängt stark von der verwendeten Hardware, insbesondere von den Systemressourcen und der Rechnergeschwindigkeit ab.

Unabhängig von der Größe der Zeichenfläche wird maximal nur ein Ausschnitt von ihr, der etwas kleiner ist als das Baumfenster, angezeigt. Anstelle einer Zeichenfläche, die etliche Megabyte an Systemressourcen verbraucht, benötigt KIEL nur einen Ausschnitt von ihr, der viele hundert Male kleiner sein kann. Ein solcher Ansatz wird in KIEL verfolgt. Er bedarf eines erheblich höheren Programmieraufwands, trägt allerdings auch maßgebend zu einer mitunter extrem hohen Ersparnis an Systemressourcen bei. Für den Benutzer viel interessanter ist, daß zudem

a) die Programmausführung wesentlich beschleunigt wird und

b) es hierdurch möglich ist, weit größere Auswertungsbäume zu erzeugen und zu bearbeiten.

Zu diesem Zweck wird das Scrolled-Window im Baumfenster „per Hand" erzeugt. Im wesentlichen werden neben der Zeichenfläche noch ein horizontaler und ein vertikaler Scrollbalken generiert und unter der Zeichenfläche bzw. rechts von ihr angeordnet. Die Größe der Zeichenfläche ist nicht statisch. Sie richtet sich nach den Benutzerwünschen, die dieser durch eine Größenänderung des Baumfensters kundtun kann.

In jedem Fall ist die im Speicher befindliche Zeichenfläche genau so groß, wie sie auf dem Bildschirm angezeigt wird. Diese Tatsache erfordert eine Anpassung der Zeichenflächengröße an die Größe des Baumfensters bei jeder Ausführung der Funktion `baumfenster_configure_event` aus der Datei `kiel_gtk_bmf.c`.

Ebenfalls muß sich der Programmierer selbst um den auf der Zeichenfläche darzustellenden Ausschnitt des Baumes kümmern. Dieser hängt neben den Ausmaßen der Zeichenfläche wesentlich von der Stellung der Scrollbalken ab. Dies wurde in den Funktionen `zeichne_knoten` und `zeichne_linie` aus der Datei `kiel_zei.c` zum Zeichnen eines Knotens bzw. einer Kante berücksichtigt.

Das Setzen der Scrollbalken nach einer Auswertung (siehe 3.3.6), und damit die Festlegung des anzuzeigenden Baumausschnittes, geschieht in der Funktion `baumfenster_expose_event` aus der Datei `kiel_gtk_bmf.c`. Den Scrollbalken, auf die dank des verwendeten Ansatzes relativ unkompliziert zugegriffen werden kann, werden Werte in Abhängigkeit

- von der Position des Mittelpunktes des zuletzt ausgewerteten Knotens,
- von dem zuletzt angezeigten Baumausschnitt und
- von der Stelle, an die der Benutzer geklickt hat, um die Auswertung zu beginnen,

zugewiesen.

5.7 Einige ausgewählte C-Funktionen

Zum Abschluß des Kapitels sollen nun einige Funktionen von KIEL im Detail beschrieben werden. Diese Darstellung stellt allerdings nur eine sehr kleine Auswahl von den in KIEL verwendeten Funktionen dar.

5.7.1 Erzeugung eines Teilbaumes

Jeder Ausdruck in KIEL wird intern als Baum verarbeitet. Jeder Teilbaum stellt einen Teilausdruck dar. Die wesentliche Struktur zum Aufbau des Baumes zu einem Ausdruck ist `AUSDRUCKBAUM_TYP` (siehe 5.4.1.3).

Während des Parsens eines Ausdrucks wird der zugehörige Baum generiert. Dabei werden vor der Erzeugung eines Knotens alle Teilbäume für seine Kinder aufgebaut. Um den Baum zu einem Knoten zu bilden, benötigt man:

- seinen Wert,
- seine Art und
- die Bäume für seine Kinder.

Die Funktion `create_ausdruckbaum` erzeugt aus genau diesen Angaben den entsprechenden Teilbaum.

Auszug aus der Datei `kiel_mem.c` — Anfang

```
AUSDRUCKBAUM_TYP create_ausdruckbaum(RETVAL_TYP rval,gint case_identifier,
                                     GList *kinder)
{
  AUSDRUCKBAUM_TYP abaum;

#ifdef KIEL_DEBUG
  kiel_debug_in(8,"create_ausdruckbaum");
#endif

  abaum=(AUSDRUCKBAUM_TYP)mtmalloc(sizeof(AUSDRUCKBAUM_STRUCT));
  abaum->wert=rval;
  abaum->nodeIdentifier=case_identifier;
  abaum->kinder=kinder;
  switch(knoten_sorte(abaum)){
    case(REAL_TYP)       :abaum->anz_stellen=VORGABE_NACHKOMMASTELLEN;
                          break;
    case(STRING_TYP)     :abaum->anz_stellen=VORGABE_ANZAHLZEICHENIMSTRING;
                          break;
    case(EMPTY_LIST_TYP) :
    case(UNIT_LIST_TYP)  :
    case(BOOL_LIST_TYP)  :
    case(INT_LIST_TYP)   :
    case(REAL_LIST_TYP)  :
    case(STRING_LIST_TYP):abaum->anz_stellen=VORGABE_ANZAHLELEMENTEINLISTE;
                          break;
    default:         abaum->anz_stellen=0;
  }
  abaum->ausgewertet=Knoten_ist_Value(abaum);
  abaum->full_expansion=FALSE;

#ifdef KIEL_DEBUG
  kiel_debug_out(8,"create_ausdruckbaum");
```

```
     #endif
16:    return(abaum);
}
```

Auszug aus der Datei `kiel_mem.c` — Ende

`ad 1:` Die vorliegende Funktion erzeugt eine Struktur, in die sie den durch die Argumente spezifizierten Teilbaum schreibt, und gibt die Adresse dieser Teilbaumstruktur, also eine Variable vom Typ `AUSDRUCKBAUM_TYP`, zurück.

`ad 2:` `abaum` ist die Variable, in der die gewünschte Struktur erstellt wird.

`ad 3:` Den `kiel_debug`-Funktionen (siehe 5.7.4) wird mitgeteilt, daß KIEL mit der Ausführung der Funktion `create_ausdruckbaum` begonnen hat.

`ad 4:` Zunächst wird ein passender Speicherbereich für `abaum` allokiert.

`ad 5:` Der Komponente `wert` des Baumes wird das Wert-Argument zugewiesen.

`ad 6:` In die Komponente `nodeIdentifier` wird die übergebene Sorte des Knotens geschrieben.

`ad 7:` Die Komponente `kinder` bekommt als Wert die Adresse der Liste, in der die Teilbäume aller Kinder des Knotens stehen.

`ad 8:` Die Komponente `anz_stellen` bestimmt die Art der Anzeige des Knotens mit. Ihr Inhalt ist abhängig von der Sorte des Knotens.

`ad 9:` Wird der Baum für einen reellwertigen Knoten erstellt, wird in `anz_stellen` die Anzahl der für diesen Knotentyp anzuzeigenden Nachkommastellen geschrieben. Der hierfür vorgesehene Vorgabewert `VORGABE_NACHKOMMASTELLEN` ist in der Ressourcedatei `kiel.rc` (siehe 3.4.1) festgelegt.

`ad 10:` Entsprechend wird `anz_stellen` der Wert von `VORGABE_ANZAHLZEICHENIMSTRING` zugewiesen, falls ein Zeichenreihenknoten zu bilden ist.

`ad 11:` Bei einem Knoten, der eine Sequenz darstellt, wird `anz_stellen` dem ebenfalls in `kiel.rc` festgelegten Wert von `VORGABE_ANZAHLELEMENTEINLISTE` gleichgesetzt.

`ad 12:` In allen anderen Fällen wird in die Komponente `anz_stellen` der Wert Null geschrieben.

`ad 13:` Der Knoten ist genau dann ausgewertet, wenn in ihm eine ML-Konstante, also ein Wert, steht.

`ad 14:` Der kreierte Knoten gehört zu einem Baum, der momentan nicht ausgewertet wird. Insbesondere findet zu diesem Zeitpunkt keine Full-Expansion statt.

`ad 15:` Den `kiel_debug`-Funktionen (siehe 5.7.4) wird mitgeteilt, daß KIEL zu diesem Zeitpunkt die letzte Anweisung der Funktion `create_ausdruckbaum` ausführt.

`ad 16:` Der so erzeugte Teilbaum wird der aufrufenden Funktion, in der Regel einem Parser, zurückgegeben.

5.7.2 Teilbaum duplizieren

Beim Arbeiten mit KIEL ist des häufig notwendig, Teilbäume zu duplizieren. Wird zum Beispiel ein Rechenvorschriftsaufruf expandiert (siehe 2.19), wird der Knoten, der den Aufruf darstellt, durch ein Duplikat des Baumes der entsprechenden Rechenvorschrift ersetzt. Des weiteren werden alle Vorkommen von Parametern durch die jeweiligen Teilbäume ersetzt, die den Argumenten des Rechenvorschriftsaufrufs entsprechen.

Um die Daten für die Rechenvorschrift nicht zu verlieren, genügt es nicht, einfach dem zu ersetzenden Teilbaum den Wert des neuen Teilbaumes zuzuweisen. Bei dieser Vorgehensweise würde nämlich nur die Adresse, an der die Teilbaumstruktur im Speicher steht, kopiert. Beide Zeiger würden also auf denselben Speicherbereich zeigen. Eine Änderung im neu entstandenen Ausdruck, also zum Beispiel ein weiterer Auswertungsschritt, würde nicht nur die Daten im Ausdrucksbaum, sondern auch die Daten der Rechenvorschrift ändern, was mit Sicherheit nicht geschehen soll.

Die Funktion `ausdruckbaum_dup` wurde für das Duplizieren eines Teilbaumes erstellt. Sie erzeugt einen Teilbaum, dessen Daten mit einem als Argument spezifizierten Teilbaum identisch sind.

Auszug aus der Datei `kiel_mem.c` — Anfang

```
 1:  AUSDRUCKBAUM_TYP ausdruckbaum_dup(AUSDRUCKBAUM_TYP abaum)
     {
 2:    AUSDRUCKBAUM_TYP erg;
 3:    GList *kinder;
       gint i;
       AUSDRUCKBAUM_TYP abaumkind;

     #ifdef KIEL_DEBUG
       kiel_debug_in(8,"ausdruckbaum_dup");
     #endif
 4:    if (!abaum) erg=NULL;
       else {
         /* Rekursion, d.h. kinder fuellen */
 5:      kinder=NULL;
 6:      for(i=0;i<g_list_length(abaum->kinder);i++) {
 7:        abaumkind=(AUSDRUCKBAUM_TYP)g_list_nth_data(abaum->kinder,i);
 8:        kinder=g_list_append(kinder,ausdruckbaum_dup(abaumkind));
         }
         /* Ergebnis fuellen */
 9:      erg=(AUSDRUCKBAUM_TYP)mtmalloc(sizeof(AUSDRUCKBAUM_STRUCT));
10:      erg->wert=retval_dup(abaum->wert);
         erg->anz_stellen=abaum->anz_stellen;
         erg->nodeIdentifier=abaum->nodeIdentifier;
         erg->ausgewertet=abaum->ausgewertet;
         erg->kinder=kinder;
         erg->zeile=abaum->zeile;
         erg->spalte=abaum->spalte;
         erg->WetSha.x=abaum->WetSha.x;
         erg->WetSha.modifier=abaum->WetSha.modifier;
         erg->expansionstiefe=abaum->expansionstiefe;
         erg->full_expansion=abaum->full_expansion;
       }

     #ifdef KIEL_DEBUG
       kiel_debug_out(8,"ausdruckbaum_dup");
     #endif
11:    return(erg);
}
```

Auszug aus der Datei `kiel_mem.c` — Ende

`ad 1:` Der Funktion `ausdruckbaum_dup` wird über den Parameter `abaum` der Teilbaum übergeben, der dupliziert werden soll. Das Duplizieren geschieht rekursiv. Zunächst wird eine Liste erstellt, die Duplikate aller Kinder von `abaum` enthält. Im Anschluß werden alle Komponenten einer neu erzeugten Struktur vom Typ `AUSDRUCKBAUM_TYP` mit den geforderten Werten belegt.

ad 2: Die Variable `erg` dient zur Aufnahme der neuen Struktur, die in dieser Funktion erstellt wird.

ad 3: Wie im ersten Punkt erwähnt, wird beim Ablauf dieser Funktion eine Liste der Teilbäume aller Kinder erstellt. Diese wird in der Variablen `kinder` festgehalten.

ad 4: Ist der zu duplizierende Baum leer, so ist auch das Ergebnis ein leerer Baum.

ad 5: Zum Erstellen der Kinderliste wird der Variablen `kinder` die leere Liste zugewiesen.

ad 6: In dieser Schleife werden alle Kinder der Wurzel von `abaum` bearbeitet.

ad 7: Dazu wird in die Variable `abaumkind` die Adresse des Teilbaumes geschrieben, der dem gerade zu bearbeitenden Kind entspricht.

ad 8: An die Liste `kinder` wird dann ein Duplikat des Teilbaumes gehängt, der durch `abaumkind` bestimmt ist.

ad 9: Zu diesem Zeitpunkt in der Ausführung der Funktion ist die Kinderliste fertiggestellt. Für die Ergebnisvariable `erg` wird ein passender Speicherbereich allokiert.

ad 10: Allen Komponenten von `erg` werden nun die durch `abaum` festgelegten Werte zugewiesen. Dabei werden bei Nicht-Zeiger-Komponenten die Werte schlicht kopiert. Bei den Komponenten `wert` und `kinder` muß ein Duplikat der Inhalte weitergegeben werden. Für die Komponente `kinder` wurde dieses bereits zu Beginn der Funktion `ausdruckbaum_dup` erstellt. Das Duplikat der Komponente `wert` wird mit Hilfe der Funktion `retval_dup` erzeugt. Diese ähnelt der vorliegenden Funktion, dupliziert aber keine Teilbäume, sondern `RETVAL_STRUCT`-Strukturen.

ad 11: Schließlich wird die Adresse des erzeugten Teilbaumes, die in der Ergebnisvariablen `erg` steht, dem Aufrufer zurückgegeben.

5.7.3 Erzeugung einer Liste von Teilbäumen

Ein Teilbaum besteht in der Regel aus mehreren Strukturen, in denen Knoteneigenschaften sowie Verkettungen unter diesen beschrieben sind. Der Aufbau eines jeden Teilbaumes geschieht in KIEL von den Blättern zur Wurzel. Dies impliziert, daß eine so konstruierte Struktur nach ihrem Setzen sofort konsistent ist. Würde ein Knoten mit Werten belegt werden, ohne daß dies vorher mit all seinen Nachkommen geschehen ist, würde man zwar den Knoten korrekt ansprechen, sich aber nicht auf die Daten verlassen können, die den von ihm induzierten Teilbaum betreffen.

Das Aufbauen der in KIEL verwalteten Bäume geschieht mit Hilfe der beiden Parser — dem Ausdruckparser und dem Rechenvorschriftenparser. Jedesmal, wenn einer von ihnen dem Baum, der sich gerade in der Entstehung befindet, einen Knoten hinzufügt, muß eine Liste aller Kinder des Knotens gebildet werden (siehe 5.4.1.6).

Die Verwendung von einer Liste an dieser Stelle resultiert aus der Tatsache, daß sich die Anzahl der Kinder von Knoten zu Knoten unterscheiden kann. Die Funktion `Liste_aus_Ausdruckbaeumen`, die für die Aufgabe geschaffen wurde, eine Liste aus Kindern zu erstellen, muß daher in der Lage sein, eine variable Anzahl an Argumenten zu übernehmen und zu verarbeiten. Als Zeichen, daß keine weiteren Argumente zu bearbeiten sind, wird von dieser Funktion das Auftauchen eines Arguments gewertet, das den Wert `NULL` enthält.

Auszug aus der Datei `kiel_fkt.c` — Anfang

```
 1:  GList *Liste_aus_Ausdruckbaeumen(AUSDRUCKBAUM_TYP abaum1,...)
     {
 2:    GList *erg;
       AUSDRUCKBAUM_TYP abaum;
```

```
 3:    va_list args;

     #ifdef KIEL_DEBUG
       kiel_debug_in(8,"Liste_aus_Ausdruckbaeumen");
     #endif

 4:    erg=NULL;
 5:    if (abaum1) {
 6:      erg=g_list_append(erg,abaum1);

 7:      va_start(args,abaum1);
 8:      abaum=va_arg(args,AUSDRUCKBAUM_TYP);
 9:      while(abaum) {
10:        erg=g_list_append(erg,abaum);
11:        abaum=va_arg(args,AUSDRUCKBAUM_TYP);
         }
12:      va_end(args);
       }

     #ifdef KIEL_DEBUG
       kiel_debug_out(8,"Liste_aus_Ausdruckbaeumen");
     #endif
13:    return(erg);
     }
```

Auszug aus der Datei `kiel_fkt.c` — Ende

`ad 1:` Das Ergebnis der Funktion `Liste_aus_Ausdruckbaeumen` ist ein Zeiger auf eine `GList`-Struktur, in welcher die Liste der Kinder erzeugt werden soll. Das erste Argument der Funktion hat zwingend den Typ `AUSDRUCKBAUM_TYP`. Alle weiteren Argumente, von denen es beliebig viele geben kann, unterliegen a priori keiner Typvorgabe. Während der Abarbeitung der Funktion wird allerdings sichergestellt, daß Argumente, die verarbeitet werden, den Typ `AUSDRUCKBAUM_TYP` haben.

`ad 2:` In einer Struktur, auf die die Ergebnisvariable `erg` zeigt, wird während der Ausführung der Funktion die gewünschte Liste gebildet.

`ad 3:` Die Variable `args` dient zur Verarbeitung der Argumente.

`ad 4:` Zu Beginn ist die Kinderliste noch leer. Deshalb wird der Ergebnisvariablen der für die leere Liste stehende Wert `NULL` zugewiesen.

`ad 5:` Hat schon der erste Parameter den Wert `NULL`, so steht in `erg` bereits das gewünschte Ergebnis.

`ad 6:` Anderenfalls wird zunächst das durch `abaum1` beschriebene erste Kind mit Hilfe des GLib-Befehls `g_list_append` hinten an die Ergebnisliste gehängt.

`ad 7:` Um die Verarbeitung der variablen Argumentenliste zu initialisieren, erfolgt ein Aufruf des Makros `va_start`, das anhand des ersten Arguments die Position der weiteren Argumente bestimmt.

`ad 8:` Der Variablen `abaum` wird der Inhalt des folgenden Arguments zugewiesen. An dieser Stelle findet zum ersten Mal die oben erwähnte Typprüfung statt.

`ad 9:` Solange in der Variablen `abaum` ein Wert steht, der nicht gleich `NULL` ist, soll dieser als Zeiger auf die Struktur eines Kindes interpretiert werden.

`ad 10:` Das aktuelle Kind wird hinten an die Ergebnisliste gehängt.

`ad 11:` Als letzte Schleifenanweisung wird der Variablen `abaum` der Inhalt des nächsten Arguments zugewiesen.

`ad 12:` An dieser Stelle ist die Abarbeitung der variablen Argumentenliste beendet. Das Arbeiten mit dieser wird mit dem Aufruf des Makros `va_end` abgeschlossen.

`ad 13:` Die Ergebnisvariable zeigt nun auf eine Listenstruktur, in der alle Argumente, die einen Wert ungleich `NULL` hatten, aufgeführt sind. Sie wird als Ergebnis der Funktion dem entsprechenden Parser zur weiteren Bearbeitung übergeben.

5.7.4 `kiel_debug`-Funktionen

Durch die grafische Verarbeitung und die komplexen Baumstrukturen, die KIEL unterstützt, kommt es vor, daß viele Funktionsaufrufe auf dem internen Aufrufstack liegen. Kommt es in der Entwicklungs- bzw. Weiterentwicklungsphase zu einem Fehler, wie zum Beispiel ein beim Programmieren mit C recht häufig auftretender Segmentierungsfehler[12], ist es wichtig, daß das genaue Auftreten des Fehlers festgestellt werden kann. Zu diesem Zweck wurden die `kiel_debug`-Funktionen geschrieben, die während der Entwicklungsphase den Programmierer genau über die Position informieren können, an der sich KIEL im Augenblick befindet. Dazu wird zu Beginn **jeder** anderen Funktion in KIEL die Funktion `kiel_debug_in` und an deren Ende die Funktion `kiel_debug_out` aufgerufen.

Mit Hilfe der `kiel_debug`-Funktionen wird jede Funktion von KIEL einer Ebene zugeordnet, welche bestimmt, wie wichtig die Funktion zur Orientierung während des Programmablaufes ist. Zum Beispiel gehört die Funktion `warnungs_fenster_geoeffnet` zur Ebene Null. Ihr Aufruf wird nur dann protokolliert, wenn der Entwickler Informationen über alle Ebenen größer gleich Null anfordert. Andere Funktionen, wie zum Beispiel `untersuche_mausklick`, sind einer höheren Ebene zugeordnet, so daß sie fast immer angezeigt werden, wenn der Programmierer Debugausgaben wünscht. Das Protokollieren geschieht auf der Standardausgabe, in der Regel also dem Fenster, aus dem KIEL gestartet wurde. Die Funktion `kiel_debug_in` zeigt jedes Eintreten in eine Funktion an, deren Ebene der Entwickler zur Anzeige ausgewählt hat. Diese Auswahl geschieht durch die Variable `kiel_debug_erwuenscht_ab_ebene` (siehe 5.5.2).

Auszug aus der Datei `kiel_fkt.c` — Anfang

```
 1:  void kiel_debug_in(gint ebene,gchar *s)
     {
       gint i;

 2:    if (kiel_debug_erwuenscht_ab_ebene<=ebene) {

 3:      for (i=0;i<einrueck_debug_puts;i++)
           if (i%2==0) printf("%d",i%10); else printf(" ");
 4:      printf("-> %s\n",s);
 5:      einrueck_debug_puts++;
       }
     }
```

Auszug aus der Datei `kiel_fkt.c` — Ende

`ad 1:` Eine beliebige aufrufende Funktion übergibt neben der Ebene, der sie zugeordnet ist, der Funktion `kiel_debug_in` noch eine Zeichenreihe, welche theoretisch jeden beliebigen Wert annehmen kann.

[12] Ein Segmentierungsfehler tritt auf, wenn das Programm einen Speicherbereich anspricht, dessen Ansprechen ihm nicht erlaubt ist. Die große Freizügigkeit, die C dem Programmierer in bezug auf den Speicherplatzzugriff läßt (siehe hierzu auch 5.7.6), hat mitunter zur Folge, daß auf vorher nicht allokierte Speicherbereiche zugegriffen werden kann, ohne daß ein Fehler angezeigt wird. Sind diese Speicherbereiche aus welchen Gründen auch immer bei der nächsten Programmausführung dem Programm nicht zugänglich, meldet das System einen Segmentierungsfehler.

In KIEL wird es jedoch so gehandhabt, daß diese Zeichenreihe immer den Namen der aufrufenden Funktion enthält.

`ad 2:` Nur wenn die Ebene der aufrufenden Funktion größer oder gleich der vom Programmierer eingestellten Mindestebene ist, werden die folgenden Anweisungen ausgeführt.

`ad 3:` Zur Verdeutlichung der Aufruftiefe, also der Größe des programmweiten Aufrufstacks, wird die eigentliche Ausgabe entsprechend eingerückt. An jeder zweiten Position wird jeweils die letzte Ziffer der Zahl notiert, die die Einrücktiefe beschreibt.

`ad 4:` Nach der Einrückung wird ein Pfeil nach rechts ausgegeben, der das Betreten der Funktion symbolisiert. Im Anschluß wird der Name der aufrufenden Funktion angezeigt und die Ausgabezeile mit einem Zeilenvorschub abgeschlossen.

`ad 5:` Da nun mit der Ausführung einer neuen Funktion begonnen wurde, wird der globale Zähler, der für die Einrückung verantwortlich ist, (siehe 5.5.2) um Eins erhöht.

Die Datei `kiel_debug_out` protokolliert das Verlassen all der Funktionen, die der Entwickler überwachen möchte. Ihr Inhalt ist dem von `kiel_debug_in` nicht unähnlich. Aufgrund der Kürze der Funktion wird sie dennoch aufgeführt.

Auszug aus der Datei `kiel_fkt.c` — Anfang

```
 1:  void kiel_debug_out(gint ebene,gchar *s)
     {
       gint i;

 2:    if (kiel_debug_erwuenscht_ab_ebene<=ebene) {

 3:      einrueck_debug_puts--;
 4:      for (i=0;i<einrueck_debug_puts;i++)
           if (i%2==0) printf("%d",i%10); else printf(" ");
 5:      printf("<- %s\n",s);
       }
     }
```

Auszug aus der Datei `kiel_fkt.c` — Ende

`ad 1:` Die Parameter der Funktion `kiel_debug_out` entsprechen genau den Parametern der Funktion `kiel_debug_in`. Die übergebene Zeichenreihe enthält wiederum den Namen der Funktion, die nun verlassen werden soll. Die Ebenen, also die Werte der ersten Parameter, beider Funktionen müssen für dieselbe Funktion identisch sein, um die gewünschte Ausgabe zu erhalten.

`ad 2:` Da das Verlassen einer Funktion genau dann protokolliert werden soll, wenn auch das Eintreten in diese Funktion protokolliert wurde, wird auch hier die Funktionenebene mit der Variablen `kiel_debug_erwuenscht_ab_ebene` verglichen.

`ad 3:` Die Anzahl der Einrückungen wird um den Wert Eins verringert, da alle Anweisungen[13] der entsprechenden Funktion ausgeführt wurden. Das Verringern geschieht vor der Ausgabeanweisung, damit diese genau so weit eingerückt ist wie die der korrespondierenden Anweisung in `kiel_debug_in`.

`ad 4:` Um dem Entwickler eine gute Zuweisung zu ermöglichen, ist das Einrücken der eigentlichen Ausgabe identisch zum Einrücken in der Funktion `kiel_debug_in`.

`ad 5:` Zur Kennzeichnung der Beendigung der Funktion wird anstelle des Pfeiles nach rechts nun ein Pfeil nach links vor dem Funktionennamen ausgegeben.

[13] Die einzige Anweisung in einer Funktion, die erst nach der Abarbeitung der Funktion `kiel_debug_out` ausgeführt wird, ist eine in Funktionen mit Rückgabewert erforderliche `return`-Anweisung. Sie beendet die Ausführung einer solchen Funktion, so daß der Aufruf von `kiel_debug_out` ignoriert werden würde, falls er erst hinter ihr notiert wäre.

5.7.5 Reagieren auf Mausklick

Das eigentliche Arbeiten mit eingegebenen Ausdrücken vollzieht sich im Baumfenster (siehe 3.3.6). Dort wählt der Benutzer von KIEL einen Knoten aus, den er bearbeiten möchte. Im einzelnen kann er folgende Aktionen ausführen:

- Durch Anklicken eines nicht ausgewerteten Knotens mit der **linken** Maustaste wird dieser Knoten **ausgewertet**.
- Durch Anklicken eines ausgewerteten Knotens mit der **mittleren** Maustaste kann
  - bei **reellen Knoten** die Anzahl der anzuzeigenden **Nachkommastellen**,
  - bei **Zeichenreihenknoten** die Anzahl der maximal **anzuzeigenden Zeichen** und
  - bei **Sequenzknoten** die Anzahl der maximal **anzuzeigenden Komponenten**

  geändert werden.
- Durch Anklicken mit der **rechten** Maustaste wird
  - ein **Rechenvorschriftsknoten expandiert** und
  - ein **Fallunterscheidungsknoten vereinfacht**.
- Durch Drücken der **mittleren** Maustaste auf einem leeren Teil der Zeichenfläche wird der dargestellte Baum so weit wie möglich vereinfacht.

Die Funktion `untersuche_mausklick` aus der Datei `kiel_gtk_bmf.c` wird nach jedem Mausklick im Baumfenster von der entsprechenden Callback-Funktion `button_press_event` aufgerufen. Sie steuert die Ausführung der gewünschten Aktion für den ausgewählten Knoten.

Auszug aus der Datei `kiel_gtk_bmf.c` — Anfang

```
 1:  void untersuche_mausklick(GdkEventButton *event,gint maustaste,
                               AUSDRUCK_TYP adk)
     {
       gdouble x,y;
       AUSDRUCKBAUM_TYP abaum;
       gint abaum_nr;
       gint x1,y1,x2,y2;
       gboolean neupositionierung_noetig;
       GList **MLRVsP;

     #ifdef KIEL_DEBUG
       kiel_debug_in(10,"untersuche_mausklick");
     #endif

 2:    if ((adk)&&(adk->baum)&&(adk->Zeichnen_erlaubt)) {
 3:      x=event->x+baumfenster->adjustmenth->value;
         y=event->y+baumfenster->adjustmentv->value;

         /* Finde erste Knoten, in dem der Mausklick stattgefunden hat.         */
         /* Wenn alles so ist, wie es sein soll, dann gibt es insgesamt maximal */
         /* einen solchen.                                                      */
 4:      for(abaum_nr=0;abaum_nr<g_list_length(adk->knotenliste);abaum_nr++){
 5:        abaum=(AUSDRUCKBAUM_TYP)g_list_nth_data(adk->knotenliste,abaum_nr);

 6:        x1=linke_obere_Ecke_des_Knotens_x(abaum,adk,baumfenster);
```

```
      y1=linke_obere_Ecke_des_Knotens_y(abaum,baumfenster);
      x2=rechte_untere_Ecke_des_Knotens_x(abaum,adk,baumfenster);
      y2=rechte_untere_Ecke_des_Knotens_y(abaum,baumfenster);

      if ((x1<=x)&&(x<=x2)&&(y1<=y)&&(y<=y2)) break;
      abaum=NULL;
    }

    if (abaum) {

      MLRVsP=hole_adresse_von_MLRechenvorschriften();

      neupositionierung_noetig=FALSE;

      switch(maustaste){
        case(LINKE_MAUSTASTE):
          if (!abaum->ausgewertet) {
            /* Baum soll nicht neu gezeichnet werden, bevor er neu */
            /* positioniert wurde, also:                           */
            adk->Zeichnen_erlaubt=FALSE;

            zuletzt_bearbeitet_abaum=
              Knoten_auswerten(abaum,adk,*MLRVsP,aktuelle_Auswertungsart(),
                               aktuelle_Anzahl_Auswertungsschritte());
            zuletzt_bearbeitet_x=(gint)x;
            zuletzt_bearbeitet_y=(gint)y;

            Protokoll_Eintrag_hinzufuegen(adk);

            neupositionierung_noetig=TRUE;
          }
          break;
        case(RECHTE_MAUSTASTE):
          if ( (Knoten_ist_Rechenvorschrift(abaum)) ||
               (Knoten_ist_IfAnweisung(abaum)) ) {
            /* Baum soll nicht neu gezeichnet werden, bevor er neu */
            /* positioniert wurde, also:                           */
            adk->Zeichnen_erlaubt=FALSE;

            if (Knoten_ist_Rechenvorschrift(abaum))
              zuletzt_bearbeitet_abaum=Knoten_expandieren(abaum,adk,*MLRVsP);
            else
              zuletzt_bearbeitet_abaum=
                Knoten_vereinfachen(abaum,adk,*MLRVsP,
                                    aktuelle_Auswertungsart());

            zuletzt_bearbeitet_x=(gint)x;
            zuletzt_bearbeitet_y=(gint)y;

            Protokoll_Eintrag_hinzufuegen(adk);

            neupositionierung_noetig=TRUE;
          }
          break;
        case(MITTLERE_MAUSTASTE):
          if ((abaum->ausgewertet)&&
```

```
                   ((knoten_sorte(abaum)==REAL_TYP)        ||
                    (knoten_sorte(abaum)==STRING_TYP)      ||
                    (knoten_sorte(abaum)==EMPTY_LIST_TYP)  ||
                    (knoten_sorte(abaum)==UNIT_LIST_TYP)   ||
                    (knoten_sorte(abaum)==BOOL_LIST_TYP)   ||
                    (knoten_sorte(abaum)==INT_LIST_TYP)    ||
                    (knoten_sorte(abaum)==REAL_LIST_TYP)   ||
                    (knoten_sorte(abaum)==STRING_LIST_TYP)
                   )
                  ) {
29:              if (event->type==GDK_BUTTON_PRESS) {
30:                abaum_fuer_geoeffnetes_popup=abaum;
                   adk_fuer_geoeffnetes_popup=adk;
31:                baumfenster->pop_adjustment->value=abaum->anz_stellen;
32:                baumfenster->pop_adjustment->upper=
                     (knoten_sorte(abaum)==REAL_TYP)?MAXIMALE_NACHKOMMASTELLEN:
                     (knoten_sorte(abaum)==STRING_TYP)?
                                                MAXIMALE_ANZAHLZEICHENIMSTRING:
                                                MAXIMALE_ANZAHLELEMENTEINLISTE;
33:                gtk_adjustment_value_changed(baumfenster->pop_adjustment);

34:                gtk_menu_popup(GTK_MENU(baumfenster->popup_menu),NULL,NULL,
                                  NULL,NULL,event->button,event->time);
                 }
               }
               break;
35:          default:break;
           }

36:        if (neupositionierung_noetig)
             /* Neupositionierung der Knoten und Erstellen des Baumes */
37:          erstelle_Baum(adk,baumfenster);
         }
         else { /* Klick hat ueber keinem Knoten stattgefunden. */
38:        if (maustaste==MITTLERE_MAUSTASTE) {
39:          if (vereinfache_basisoperationen(adk,aktuelle_Auswertungsart())) {
40:            zuletzt_bearbeitet_abaum=NULL;
               zuletzt_bearbeitet_x=0;
               zuletzt_bearbeitet_y=0;

               Protokoll_Eintrag_hinzufuegen(adk);

               erstelle_Baum(adk,baumfenster);
             }
           }
         }
       }

     #ifdef KIEL_DEBUG
       kiel_debug_out(10,"untersuche_mausklick");
     #endif
     }
```

Auszug aus der Datei `kiel_gtk_bmf.c` — Ende

`ad 1:` Die Funktion `untersuche_mausklick` hat drei Argumente. Das erste Argument beinhaltet Informationen über den Mausklick. Aus ihm werden später die genauen Koordinaten im Baumfenster ausgelesen, an dem der Mausklick stattgefunden hat. Dieses Argument hat den von GDK eigens hierfür vorgesehenen Typ `GdkEventButton *`.

Über den Parameter `maustaste` stellt die Funktion fest, welche Maustaste gedrückt wurde. Eine solche Information steht ebenfalls im ersten Argument. In der aufrufenden Callback-Funktion `button_press_event` wird die entsprechende Komponente der Ereignisstruktur in einen der von KIEL definierten Werte

- `LINKE_MAUSTASTE`,
- `MITTLERE_MAUSTASTE` oder
- `RECHTE_MAUSTASTE`

umgewandelt. An eine geänderte Kodierung in dieser Struktur, welche aus der Benutzung einer Maus mit mehr als drei Tasten resultieren könnte, muß KIEL somit nur in der überschaubaren Callback-Funktion angepaßt werden.

Schließlich wird noch der Ausdruck übergeben, der aktuell in dem Baumfenster angezeigt wird. Er wird unter anderem von den Funktionen benötigt, die aus einem Mausklick resultieren können.

`ad 2:` Der Benutzer kann natürlich auch in das Baumfenster klicken, ohne daß dort ein Ausdruck angezeigt wird. In diesem Fall kann er offenbar keine Aktion auswählen. Deshalb werden alle folgenden Anweisungen nur dann ausgeführt, wenn das Argument `adk` auf irgendeine Ausdrucksstruktur zeigt. Des weiteren wird gefordert, daß in dieser Ausdrucksstruktur die Komponente, die den Baum verwaltet, nicht leer ist. Auch hier kann keine Aktion bewirkt werden, da keine Knoten angezeigt sein können. Schließlich muß die Komponente `Zeichnen_erlaubt` der Ausdrucksstruktur den Wert `TRUE` haben. Diese Komponente wird bei der Erstellung eines Baumes erst ganz am Ende auf den Wert `TRUE` gesetzt. Ist der Wert zu dem Zeitpunkt, an dem der Benutzer auf das Baumfenster klickt, gleich `FALSE`, bedeutet dies, daß KIEL momentan intern einen Ausdruck erstellt und somit die Darstellung im Baumfenster nicht dem Speicherinhalt von KIEL entspricht. Auch in diesem Fall muß der Benutzer warten, bis die Berechnung beendet wurde.

`ad 3:` In die Variablen `x` und `y` wird die Position des Mausklicks auf der Zeichenfläche geschrieben. Dabei ist zu berücksichtigen, daß mitunter nur ein Teil der gesamten Zeichenfläche des Baumes im Baumfenster angezeigt wird. Deshalb müssen die Stellungen der beiden Scrollbalken bei der Positionsbestimmung beachtet werden.

`ad 4:` In dieser Schleife wird nach einem Knoten gesucht, für den aufgrund der Koordinaten des Mausklicks eine Aktion ausgeführt werden soll. Zu diesem Zweck werden nacheinander alle Einträge der Knotenliste des aktuellen Ausdrucks untersucht.

`ad 5:` In die Variable `abaum` werden die Teilbaumdaten des Knotens geschrieben, der durch den Wert der Schleifenvariablen bestimmt ist.

`ad 6:` Die Variablen `x1`, `y1`, `x2` und `y2` nehmen die Koordinaten der linken oberen bzw. rechten unteren Ecke des Rechtecks auf, das den aktuellen Knoten kleinstmöglich mit parallel zum Bildschirmrand verlaufender unterer Seite umschließt.

`ad 7:` Liegt die Position des Mausklicks innerhalb dieses Rechtecks, so wird es als zu bearbeitend erkannt. Die Schleifenausführung wird dann beendet, so daß ab der folgenden Anweisung die Variable `abaum` auf den ausgewählten Knoten zeigt. Es kann normalerweise nicht vorkommen, daß die Rechtecke um zwei Knoten den angeklickten Punkt überdecken.[14] Dies hängt mit dem Aufbau des Baumes zusammen, der in 5.6 genau beschrieben wird.

[14]Durch unsinnige Einträge in der Ressourcedatei `kiel.rc` kann der Benutzer eine Überschneidung zweier Knoten herbeiführen. Zum einen kann er den horizontalen Mindestabstand zweier Knoten auf einen negativen Wert setzen und zum anderen die Zeilenhöhe extrem gering bemessen.

ad 8: Es kann hingegen der Fall eintreten, daß die Koordinaten des Mausklicks von keinem Rechteck eines Knotens überdeckt werden. In diesem Fall soll nach der Schleifenabarbeitung in der Variablen `abaum` der `NULL`-Pointer stehen.

ad 9: Falls der Benutzer einen Knoten angeklickt hat, zeigt die Variable `abaum` auf die entsprechende Teilbaumstruktur. Nur in einem solchen Fall sind die folgenden Anweisungen auszuführen.

ad 10: In der Variablen `MLRVsP` wird die Adresse der Liste geschrieben, in der alle momentan in KIEL eingelesenen Rechenvorschriften stehen. Sie wird zum Ausführen der Aktionen mit dem ausgewählten Knoten benutzt.

ad 11: In der booleschen Variablen `neupositionierung_noetig` wird festgehalten, ob die Knoten am Ende der gesamten Funktion neu positioniert werden müssen. Initial wird ihr der Wert `FALSE` zugewiesen. Erfordern gewisse Aktionen eine Neupositionierung, wird ihr Wert dann auf `TRUE` geändert.

ad 12: In dem folgenden etwas längeren Block werden die Anweisungen in Abhängigkeit der gedrückten Maustaste ausgeführt.

ad 13: Zunächst wird der Fall behandelt, daß die linke Maustaste gedrückt wurde. Hier soll ein unausgewerteter Knoten ausgewertet werden.

ad 14: Die Anweisungen sind nur auszuführen, wenn der Knoten noch nicht ausgewertet ist.

ad 15: Während der Auswertung, die mitunter länger dauern kann, soll der Baum im Baumfenster nicht aktualisiert werden. Er wird bearbeitet und liegt deshalb in den seltensten Fällen in konsistenter Form vor.

Die Callback-Funktion, die das Neuzeichnen bewirken soll, wird trotzdem vom System aufgerufen. In der Darstellungsfunktion wird jedoch vor dem Darstellen überprüft, ob auf den Baum der Baumzeichenalgorithmus bereits angewendet wurde, ob also die Knotenstrukturen korrekte Positionierungen beinhalten. Nach einer solchen Positionierung aller Knoten wird die Komponente `Zeichnen_erlaubt` in der Ausdrucksstruktur auf den Wert `TRUE` gesetzt. Die Darstellungsfunktion prüft unter anderem, ob dieser Wert gesetzt ist. Das Setzen dieser Komponente des aktuellen Ausdrucks auf den Wert `FALSE` bewirkt somit, daß das Neuzeichnen des Baumes unterbunden wird.

ad 16: Hier geschieht die eigentliche Auswertung des Knotens. Diese Aufgabe wird von der Funktion `Knoten_auswerten` übernommen, die einen Knoten eines Ausdrucks unter Verwendung der geladenen Rechenvorschriften und unter Einhaltung der gewählten Auswertungsart um die Anzahl an Auswertungsschritten auswertet, die der Benutzer im Auswertungsfenster (siehe 3.3.8) ausgewählt hat. Ihr Ergebnis ist der auf diese Art und Weise bearbeitete Knoten. Dieser wird in der Teilbaum-Variablen `zuletzt_bearbeitet_abaum` (siehe 5.5.2) festgehalten.

ad 17: In den globalen Variablen `zuletzt_bearbeitet_x` und `zuletzt_bearbeitet_y` werden die Koordinaten des Mausklicks als Ganzzahlen gespeichert. Mit ihrer Hilfe ist später die Einstellung der Scrollbalken im Baumfenster möglich.

ad 18: Nach der eben erfolgten Auswertung wird das Ergebnis protokolliert, um über das Undo-Redo-Fenster verwaltbar zu sein.

ad 19: Da sich der Baum geändert hat, sollen die Knoten neu positioniert werden.

ad 20: Die folgenden Anweisungen werden bearbeitet, wenn die rechte Maustaste gedrückt wurde. Mit ihr können Rechenvorschriftsaufrufe expandiert und Fallunterscheidungen vereinfacht werden.

ad 21: Die Anweisungen sind nur dann auszuführen, wenn der ausgewählte Knoten ein Rechenvorschrifts- oder ein Fallunterscheidungsknoten ist.

`ad 22:` Da auch in diesem Fall das Baumfenster erst nach kompletter Ausführung der Aktion neu aufgebaut werden soll, wird wie bei der linken Maustaste die Komponente `Zeichnen_erlaubt` von `adk` auf den Wert `FALSE` gesetzt.

`ad 23:` Falls der Benutzer auf einen Rechenvorschriftsknoten geklickt hat, soll dieser expandiert werden.

`ad 24:` Dazu wird die Funktion `Knoten_expandieren` aufgerufen, die einen Rechenvorschriftsknoten eines Ausdrucks unter Verwendung der aktuell bekannten Rechenvorschriften expandiert. Der aus der Expandierung entstehende Knoten wird in der Variablen `zuletzt_bearbeitet_abaum` gesichert.

`ad 25:` Wurde vom Benutzer ein Fallunterscheidungsknoten angeklickt, wird dieser mit Hilfe der Funktion `Knoten_vereinfachen` vereinfacht. Dazu wird in dieser Funktion zuerst der boolesche Ausdruck ausgewertet und anschließend in Abhängigkeit dessen Wertes der `then`- oder der `else`-Zweig an die Stelle des Fallunterscheidungsknotens gesetzt. Die Wurzel des jeweiligen Zweiges wird in der Variablen `zuletzt_bearbeitet_abaum` festgehalten.

`ad 26:` Die Variablen `zuletzt_bearbeitet_x` und `zuletzt_bearbeitet_y` werden genau wie bei der Behandlung der linken Maustaste gesetzt. Außerdem wird das Ergebnis der ausgeführten Aktion protokolliert und eine Neupositionierung der Baumknoten angefordert.

`ad 27:` Der folgende Abschnitt behandelt die auszuführenden Aktionen, wenn die mittlere Maustaste auf einem Knoten gedrückt wurde. Es soll die Komponente `anz_stellen` von ausgewerteten reellen Knoten, Zeichenreihen- und Sequenzknoten verändert werden können, falls die Maustaste zu diesem Zeitpunkt noch immer gedrückt ist.

`ad 28:` Hier wird getestet, ob der angeklickte Knoten

a) ausgewertet und
b) sein Typ
   - `real`-Typ,
   - `string`-Typ oder
   - ein Sequenztyp

   ist.

`ad 29:` Die Möglichkeit, die Komponente `anz_stellen` des angeklickten Knotens zu verändern, soll der Benutzer nur dann haben, wenn er die mittlere Maustaste gedrückt hält.

`ad 30:` Die globalen Variablen `abaum_fuer_geoeffnetes_popup` und `adk_fuer_geoeffnetes_popup` werden gesetzt. Ihre Bedeutung ist in 5.5.2 näher beschrieben.

`ad 31:` Im Baumfenster gibt es eine Popup-Komponente, und zwar genau die, die das Einstellen der darzustellenden Anzahl der Nachkommastellen, der Zeichen bzw. der Komponenten ermöglicht. Da sie nun angezeigt werden soll, wird ihr aktueller Wert auf den zu ändernden Wert der Komponente `anz_stellen` des ausgewählten Knotens gesetzt.

`ad 32:` Die Obergrenze der Wählscheibe wird in Abhängigkeit des Typs des angeklickten Knotens gesetzt.

`ad 33:` Die soeben vorgenommene Änderung des Wertes in der Wählscheibe wird GTK durch den Aufruf des Befehls `gtk_adjustment_value_changed` mitgeteilt.

`ad 34:` Nach dem Setzen aller erforderlichen Werte wird die Wählscheibe dargestellt. Außerdem werden die internen Callback-Funktionen festgelegt, damit die eingegebenen Wertänderungen wunschgemäß bearbeitet werden können.

`ad 35:` Wurde eine andere als die linke, mittlere oder rechte Maustaste gedrückt, soll nichts geschehen.

`ad 36:` Falls die Neupositionierung der Knoten erforderlich ist, wird die folgende Anweisung ausgeführt.

`ad 37:` Die Knoten werden in der Funktion `erstelle_Baum` neu positioniert. Außerdem wird der Baum von dieser Funktion neu gezeichnet und der im Zeichenreihenfenster angezeigte Wert aktualisiert.

`ad 38:` Wurde eine Maustaste über einem freien Teil der Zeichenfläche gedrückt, wird geprüft, ob dies die mittlere Maustaste war. In diesem Fall soll der dargestellte Ausdruck so weit wie möglich vereinfacht werden.

`ad 39:` An dieser Stelle wird die Funktion `vereinfache_basisoperationen` aufgerufen, die die Vereinfachung durchführt.

`ad 40:` Wurde eine Vereinfachung durchgeführt, wird diese protokolliert, und der Ausdrucksbaum wird neu gezeichnet. Da kein Knoten angeklickt wurde, soll auch kein Knoten fett gezeichnet werden. Dementsprechend werden die Variablen `zuletzt_bearbeitet_abaum`, `zuletzt_bearbeitet_x` und `zuletzt_bearbeitet_y` gesetzt.

5.7.6 mt-Funktionen

Die Programmiersprache C ist in Hinblick auf Speicherverwaltung sehr großzügig. Es kann daher schnell vorkommen, daß auf reservierte Speicherbereiche zugegriffen und geschrieben werden kann, was in der Regel zu Programmabstürzen führt, deren Ursache mitunter nicht sofort nachvollzogen werden kann. Es kann zum Beispiel sein, daß ein Programm nach einem unerlaubten Zuriff auf den Speicher noch eine Zeit lang weiter läuft und dann bei einer `puts`-Anweisung abstürzt, was häufig zu einiger Verwirrung führt.

Reine C-Funktionen zum Allokieren und Freigeben von Speicherplatz sind `malloc` und `free`. Ein erster Schritt zu mehr Sicherheit ist es, anstelle dieser Funktion die GLib-Funktionen `g_malloc0` und `g_free` zu verwenden. Anders als `malloc` füllt `g_malloc0` den allokierten Speicherplatz mit dem Wert Null.

Um während des Prgrammablaufes mehr Kontrolle über den belegten Speicherplatz zu erlangen, wurden die **mt-Funktionen** geschrieben. Die Funktionen `g_malloc0` und `g_free` werden durch die Funktionen `mtmalloc` und `mtfree` ersetzt. Das eigentliche Allokieren und Freigeben geschieht in diesen Funktionen immer noch mit Hilfe der beiden GLib-Funktionen. Bei jedem Allokieren und bei jeder Freigabe werden jedoch die Adresse des Speicherbereiches sowie seine Größe protokolliert. Zunächst sind einige globale Variable notwendig, die aber nur den Speicherfunktionen bekannt sind. Daher werden sie in der privaten Headerdatei `kiel_mem_priv.h` deklariert.

Auszug aus der Datei `kiel_mem_priv.h` — Anfang

```
1:  gint belegter_Speicher_durch_mtmalloc=0;
2:  #define max_anz_speicherplaetze 100000
3:  gboolean mtmem_belegt[max_anz_speicherplaetze];
    typedef void *voidp;
4:  voidp mtmem_adresse[max_anz_speicherplaetze];
5:  gint mtmem_groesse[max_anz_speicherplaetze];
```

Auszug aus der Datei `kiel_mem_priv.h` — Ende

`ad 1:` Die Variable `belegter_Speicher_durch_mtmalloc` soll jederzeit die Anzahl der Bytes enthalten, die durch Aufrufe von `mtmalloc` allokiert sind. Zu Beginn ist diese Anzahl Null.

`ad 2:` Insgesamt werden `max_anz_speicherplaetze` Speicherplätze zur Verfügung gestellt.

`ad 3:` Das boolesche Variablenfeld `mtmem_belegt` enthält zu jedem Speicherplatz die Information, ob dieser belegt ist.

`ad 4:` Wird ein Speicherplatz belegt, hat dieser eine Adresse. Sie wird in dem Feld `mtmem_adresse` aus `void`-Pointern festgehalten.

`ad 5:` Die Größe, die ein belegter Speicherplatz einnimmt, steht in dem Feld `mtmem_groesse`.

Zu Beginn des Hauptprogramms von KIEL wird die Funktion `mtmemory_init` aufgerufen. Sie dient zum Initialisieren des Variablenfeldes `mtmem_belegt`.

Auszug aus der Datei `kiel_mem.c` — Anfang

```
    void mtmemory_init()
    {
      gint i;

    #ifdef KIEL_DEBUG
      kiel_debug_in(1,"mtmemory_init");
    #endif

1:    for(i=0;i<max_anz_speicherplaetze;i++)
        mtmem_belegt[i]=FALSE;

    #ifdef KIEL_DEBUG
      kiel_debug_out(1,"mtmemory_init");
    #endif
    }
```

Auszug aus der Datei `kiel_mem.c` — Ende

`ad 1:` In der Schleife werden alle Komponenten des Feldes `mtmem_belegt` durchlaufen und auf den Wert `FALSE` gesetzt. Zu Beginn der Programmausführung ist noch kein Speicherplatz der mt-Funktionen belegt, da die Funktion `mtmalloc` noch nicht benutzt wurde.

Nach der Initialisierung kann ein Speicherplatz mit Hilfe der Funktion `mtmalloc` allokiert werden.

Auszug aus der Datei `kiel_mem.c` — Anfang

```
 1: void *mtmalloc(size_t size)
    {
 2:   gint i;
 3:   void *erg;

    #ifdef KIEL_DEBUG
      kiel_debug_in(1,"mtmalloc");
    #endif

      i=0;
 4:   while((mtmem_belegt[i])&&(i<max_anz_speicherplaetze)) i++;
 5:   if (i==max_anz_speicherplaetze) {
 6:     puts("Zuviele mtmallocs");
        gtk_exit(1);
      }

 7:   erg=g_malloc0(size);
 8:   belegter_Speicher_durch_mtmalloc+=size;

 9:   mtmem_adresse[i]=erg;
10:   mtmem_groesse[i]=size;
11:   mtmem_belegt[i]=TRUE;
```

```
     #ifdef KIEL_DEBUG
       kiel_debug_out(1,"mtmalloc");
     #endif
12:    return(erg);
     }
```

Auszug aus der Datei kiel_mem.c — Ende

ad 1: Die Funktion `mtmalloc` liefert einen Zeiger auf einen Speicherbereich, dessen Größe dem Wert des einzigen Arguments `size` entspricht und der in dieser Funktion allokiert werden soll. Da diese Funktion für verschiedenste Strukturen zum Allokieren eines Speicherbereiches genutzt wird, ist das Ergebnis ein Zeiger auf eine Variable des Typs `void`. In der aufrufenden Funktion wird es dann entsprechend gecastet.

ad 2: Die Variable `i` wird als Zählvariable einer Schleife verwendet.

ad 3: Die Ergebnisvariable `erg` dient zum Aufnehmen des Ergebnisses der Funktion.

ad 4: Die Speicherplätze der mt-Funktionen werden von vorne nach dem ersten unbelegten Speicherplatz durchsucht.

ad 5: Ist der Wert der Zählvariablen `i` gleich dem Wert der Konstanten `max_anz_speicherplaetze`, sind `max_anz_speicherplaetze` Speicherplätze bereits belegt.[15]

ad 6: In diesem Fall wird ein Fehlertext auf dem Bildschirm ausgegeben, und KIEL wird sofort beendet. Diese Maßnahme ist trotz ihrer Härte gerechtferigt, da es ohne den angeforderten Speicher dem Programm nicht möglich ist weiterzulaufen.

ad 7: An diese Stelle gelangt das Programm nur, wenn ein unbelegter Speicherplatz gefunden wurde. Nun wird mit Hilfe der GLib-Funktion `g_malloc0` ein Speicherbereich allokiert. Seine Adresse wird der Ergebnisvariablen übergeben.

ad 8: Der insgesamt durch mt-Funktionen belegte Speicher ist um `size` Bytes angestiegen.

ad 9: Die Adresse des soeben allokierten Speicherbereiches wird in das Feld `mtmem_adresse` an die Position geschrieben, die dem zu füllenden Speicherplatz entspricht.

ad 10: Analog wird die Größe der Speicherbereiches in dem Feld `mtmem_groesse` festgehalten.

ad 11: Da der ausgewählte Speicherplatz nun belegt ist, wird das Feld `mtmem_belegt` entsprechend aktualisiert.

ad 12: Am Ende von `mtmalloc` wird der aufrufenden Funktion die Ergebnisvariable `erg` übergeben.

Mit `mtmalloc` allokierte Speicherbereiche können mit der Funktion `mtfree` wieder freigegeben werden. Sie stützt sich auf die GLib-Funktion `g_free`. Im Gegensatz zu dieser und auch zur C-Funktion `free` beendet `mtfree` das Programm, falls der Speicherbereich nicht vorher mit `mtmalloc` allokiert wurde.

Auszug aus der Datei kiel_mem.c — Anfang

```
 1:  void mtfree(void *ptr)
     {
 2:    gint i;
```

[15] Wie in C üblich hat die erste Komponente aller mt-Felder die Position Null und die letzte Komponente die Position `max_anz_speicherplaetze` − 1.

```
    #ifdef KIEL_DEBUG
      kiel_debug_in(1,"mtfree");
    #endif

      i=0;
 3:   while(((!mtmem_belegt[i])||(mtmem_adresse[i]!=ptr))
            &&(i<max_anz_speicherplaetze)) i++;
 4:   if (i==max_anz_speicherplaetze) {
 5:     puts("Pointer wurde nicht gemtmalloct.");
        gtk_exit(1);
      }

 6:   belegter_Speicher_durch_mtmalloc-=mtmem_groesse[i];

 7:   mtmem_adresse[i]=0;
 8:   mtmem_groesse[i]=0;
 9:   mtmem_belegt[i]=FALSE;

10:   g_free(ptr);

    #ifdef KIEL_DEBUG
      kiel_debug_out(1,"mtfree");
    #endif
    }
```

Auszug aus der Datei `kiel_mem.c` — Ende

`ad 1:` Der Funktion `mtfree` wird als Parameter die Adresse des Speicherbereiches übergeben, der freigegeben werden soll.

`ad 2:` Die Variable `i` ist eine Zählvariable für eine Schleife.

`ad 3:` In dieser Schleife wird die Komponente der mt-Felder gesucht, die den freizugebenden Speicherbereich verwaltet. Zusätzlich wird gefordert, daß dieser Speicherplatz belegt ist.

`ad 4:` Ist der Wert der Zählvariablen `i` gleich dem Wert der Konstanten `max_anz_speicherplaetze`, wurden alle Komponenten erfolglos nach einem belegten Speicherbereich durchsucht, dessen Inhalt der übergebenen Adresse entspricht.

`ad 5:` Analog zu der Fehlerbehandlung in der Funktion `mtmalloc` wird auch hier ein Fehlertext ausgegeben und die gesamte Anwendung beendet. Dieser Fehler tritt allerdings nur dann auf, wenn ein Speicherbereich freigegeben werden soll, der entweder vorher nicht mit `mtmalloc` allokiert wurde oder der in der Zwischenzeit bereits wieder freigegeben ist. Beide Fälle dürfen während der Ausführung von KIEL nicht vorkommen.

`ad 6:` Wurde ein Speicherplatz gefunden, der zu dem freizugebenden Speicherbereich gehört, wird der Wert der Variablen `belegter_Speicher_durch_mtmalloc` um die Größe dieses Speicherbereiches vermindert.

`ad 7:` Der Speicherplatz wird zurückgesetzt, indem der Adresse der Wert Null zugewiesen wird.

`ad 8:` Die Größe wird ebenfalls auf Null gesetzt.

`ad 9:` Da der Speicherbereich im folgenden Schritt freigegeben wird, wird dieser Sachverhalt im Feld `mtmem_belegt` festgehalten.

`ad 10:` Mit Hilfe der GLib-Funktion `g_free` wird der Speicherbereich freigegeben.

5.7.7 Ausführen eines Protokolls

Bei jedem Auswertungsschritt in KIEL, den der Benutzer durch einen Mausklick im Baumfenster (siehe 3.3.6) ausführt, wird das interne Protokoll um einen Eintrag erweitert. Innerhalb des Protokolls kann mit Hilfe des Undo-Redo-Fensters (siehe 3.3.9) die Auswertung rückgängig gemacht werden. Dazu ist es notwendig, die Position im Auswertungsprotokoll einfach ändern zu können.

Die folgende Funktion `Protokoll_ausfuehren` steht in der Datei `kiel_pkl.c`. Sie dient zum Ändern der Position in einem Auswertungsprotokoll. Ihre Argumente sind

- `adk` — der Ausdruck, in dem die Protokollposition geändert werden soll —,
- `anz` — die Anzahl der Einträge, um die geändert werden soll —,
- `richtung` — die Richtung, in die geändert werden soll, — und
- `bmf` — das Baumfenster, in dem geändert werden soll.

Auszug aus der Datei `kiel_pkl.c` — Anfang

```
void Protokoll_ausfuehren(AUSDRUCK_TYP adk,gint anz,gint richtung,
                          BAUMFENSTER_TYP bmf)
{
  gint echte_anzahl;
  AUSDRUCKBAUM_TYP abaum;

#ifdef KIEL_DEBUG
  kiel_debug_in(10,"Protokoll_ausfuehren");
#endif

  if (adk) {
    switch(richtung){
      case(PROTOKOLL_RICHTUNG_ZURUECK):
        echte_anzahl=MIN(anz,adk->protokoll->position);
        if (echte_anzahl>0) {
          free_ausdruckbaum(adk->baum);
          abaum=Protokoll_Eintrag_holen(adk->protokoll,
                                        adk->protokoll->position-echte_anzahl);
          adk->baum=ausdruckbaum_dup(abaum);
          adk->knotenliste=erstelle_Knotenliste(adk->baum);

          adk->protokoll->position-=echte_anzahl;
          UndoRedo_ProgressBar_und_MaxLabel_aktualisieren(adk);
        }
        break;
      case(PROTOKOLL_RICHTUNG_VOR):
        echte_anzahl=MIN(anz,Protokoll_Anzahl_Eintraege(adk->protokoll)-1-
                             adk->protokoll->position);
        if (echte_anzahl>0) {
          free_ausdruckbaum(adk->baum);
          abaum=Protokoll_Eintrag_holen(adk->protokoll,
                                        adk->protokoll->position+echte_anzahl);
          adk->baum=ausdruckbaum_dup(abaum);
          adk->knotenliste=erstelle_Knotenliste(adk->baum);

          adk->protokoll->position+=echte_anzahl;
          UndoRedo_ProgressBar_und_MaxLabel_aktualisieren(adk);
```

```
            }
            break;
17:       default:puts("exit in switch ::Protokoll_ausfuehren");
                  gtk_exit(1);
        }

18:     if (echte_anzahl>0) {
19:       setze_zuletzt_bearbeitet_abaum_auf_NULL();

          /* Neupositionierung der Knoten und Erstellen des Baumes */
20:       erstelle_Baum(adk,bmf);

          /* Zeichenreihenlabel-Text erneuern */
21:       Zeichenreihenlabel_aktualisieren();
        }
      }

     #ifdef KIEL_DEBUG
       kiel_debug_out(10,"Protokoll_ausfuehren");
     #endif
     }
```

Auszug aus der Datei `kiel_pkl.c` — Ende

`ad 1:` Die Funktion `Protokoll_ausfuehren` hat keinen Rückgabewert. Das erste und das vierte Argument sind Strukturen, die in 5.4.1.6 und 5.4.1.10 beschrieben sind. Für die Anzahl der Positionen und für die Richtung wird jeweils eine `gint`-Variable verwendet. Die beiden verwendeten Werte für Richtungsangaben werden wie folgt in der Datei `kiel_pkl.h` definiert:

```
#define PROTOKOLL_RICHTUNG_ZURUECK 0x01
#define PROTOKOLL_RICHTUNG_VOR     0x02
```

`ad 2:` Die als Argument `anz` übergebene Anzahl an Einträgen, um die das Protokoll geändert werden soll, ist nicht immer gleich der Anzahl, um die diese Funktion das Protokoll ändert. Soll die Protokollposition um mehr Einträge verändert werden als vorhanden sind, so wird sie nur um die maximal mögliche Anzahl an Einträgen geändert; das heißt: das Protokoll wird an seinen Anfang bzw. sein Ende gesetzt (siehe 3.3.9). Die `gint`-Variable `echte_anzahl` wird verwendet, um die tatsächliche Anzahl der Einträge festzuhalten, um die die Protokollposition geändert wird.

`ad 3:` In der Variablen `abaum` vom Typ `AUSDRUCKBAUM_TYP` wird der Ausdrucksbaum festgehalten, der dem angeforderten Protokolleintrag entspricht.

`ad 4:` Nur wenn der Ausdruck `adk` existiert, hat dieser auch ein Protokoll, in dem die Position geändert werden kann.

`ad 5:` Die `switch`-Anweisung verzweigt die Programmausführung für Rückwärts- bzw. Vorwärtsgehen im Protokoll.

`ad 6:` Beim Rückwärtsgehen ermittelt sich die tatsächliche Anzahl an Einträgen, um die die Protokollposition geändert wird, als Minimum der angeforderten Anzahl und der aktuellen Position im Protokoll.

`ad 7:` Eine Änderung findet nur dann statt, wenn die tatsächliche Anzahl an Einträgen größer als Null ist.

`ad 8:` In diesem Fall wird zunächst der bisherige Auswertungsbaum des Ausdrucks `adk` freigegeben.

`ad 9:` Mit Hilfe der Funktion `Protokoll_Eintrag_holen` wird der Variablen `abaum` der Ausdrucksbaum zugewiesen, der im Protokoll `echte_anzahl` Positionen weiter vorne steht.

`ad 10:` Die Komponente `baum` des Ausdrucks `adk` bekommt als Wert ein Duplikat des eben ermittelten Baumes zugewiesen (siehe 5.7.2).

`ad 11:` Nach der Änderung des Baumes muß die Knotenliste in `adk` erneuert werden.

`ad 12:` Das Ändern der Position im Protokoll wird in der Komponente `position` des Ausdruckprotokolls festgehalten.

`ad 13:` Nach der Änderung der Protokollposition muß unter anderem der Fortschrittsbalken des Undo-Redo-Fenster aktualisiert werden (siehe 3.3.9).

`ad 14:` Beim Vorwärtsgehen ermittelt sich die tatsächliche Anzahl an Einträgen, um die die Protokollposition geändert wird, als Minimum der angeforderten Anzahl und der Anzahl an Einträgen, die auf die aktuelle Position im Protokoll folgen.

`ad 15:` Analog zur entsprechenden Zuweisung beim Rückwärtsgehen wird der Variablen `abaum` der Ausdrucksbaum zugewiesen, der im Protokoll `echte_anzahl` Positionen weiter hinten steht.

`ad 16:` Zur Aktualisierung der Protokollposition wird die Komponente `position` des Ausdruckprotokolls um `echte_anzahl` inkrementiert.

`ad 17:` Da immer einer der beiden obigen Fälle eintreten soll, erfolgt im `default`-Teil der `switch`-Anweisung ein Programmabbruch.

`ad 18:` Wenn sich die Protokollposition geändert hat, müssen noch einige Aktualisierungen vorgenommen werden.

`ad 19:` Damit kein Knoten fett gezeichnet wird, wird KIEL mitgeteilt, daß zuletzt kein Knoten bearbeitet wurde.

`ad 20:` Im Baumfenster `bmf` muß der Auswertungsbaum neu erstellt werden.

`ad 21:` Im Zeichenreihenfenster (siehe 3.3.7) muß der Ausdruck entsprechend dem neuen Auswertungsfortschritt angezeigt werden.

5.7.8 Übernehmen von `int`-Variablen

Abschließend folgt eine Beschreibung des Mechanismus zur Übernahme von `int`-Variablen.

Nach jeder Auswertung, die als Ergebnis einen Knoten vom Typ `int` hat, wird die folgende Funktion `uebernehme_int` genauso aufgerufen, wie nach dem Parsen eines `int`-Typs. Ihre Argumente sind

- `n` — der Wert, der übernommen werden soll, — und
- `retval` — die Struktur, in die der Wert geschrieben werden soll.

Jede Berechnung, die als Ergebnis einen `int`-Wert hat, wird in Gleitkommazahlen ausgeführt, um Über- und Unterläufe auffangen zu können. Die Funktion `uebernehme_int` realisiert genau diesen Sachverhalt.

Auszug aus der Datei `kiel_ausw.c` — Anfang

```
1:  void uebernehme_int(gdouble n,RETVAL_TYP retval)
    {
    #ifdef KIEL_DEBUG
      kiel_debug_in(10,"uebernehme_int");
    #endif

2:    if ((!teste_int_overflow(n,retval))&&
          (!teste_int_underflow(n,retval))) {
3:      retval->typ=INT_TYP;
4:      retval->_int=(gint)n;
      }

    #ifdef KIEL_DEBUG
      kiel_debug_out(10,"uebernehme_int");
    #endif
    }
```

Auszug aus der Datei `kiel_ausw.c` — Ende

`ad 1:` Die Funktion `uebernehme_int` hat keinen Rückgabewert. Ihr erstes Argument ist der zu übernehmende Wert als `gdouble`-Variable und das zweite Argument ist eine `RETVAL_TYP`-Variable.

`ad 2:` Durch die Funktionen `teste_int_overflow` und `teste_int_underflow` wird überprüft, ob das erste Argument in die Struktur übernommen werden kann. Findet ein Über- oder ein Unterlauf statt, so schreibt die entsprechende Funktion in die `RETVAL_TYP`-Variable einen Fehlerwert. Nur wenn beide Tests negative Ergebnisse liefern, werden die folgenden Befehle ausgeführt.

`ad 3:` Der Typ in der `RETVAL_TYP`-Variablen wird auf `int` gesetzt.

`ad 4:` Der Wert für die `RETVAL_TYP`-Variable ist ein gültiger `int`-Wert. Er wird hier übernommen.

Wie eben schon erwähnt, dient die Funktion `teste_int_overflow` zur Überprüfung, ob ein Wert zu groß ist, um in eine `int`-Variable umgewandelt werden zu können. Ist dies der Fall, so wird in der ebenfalls übergebenen Variablen vom Typ `RETVAL_TYP` ein Fehlerwert gesetzt. Die Argumente sind

- `n` — der Wert, der übernommen werden soll, — und
- `retval` — die Struktur, in die der Wert geschrieben werden soll.

Auszug aus der Datei `kiel_ausw.c` — Anfang

```
 1:  gboolean teste_int_overflow(gdouble n,RETVAL_TYP retval)
     {
 2:    gboolean erg;

     #ifdef KIEL_DEBUG
       kiel_debug_in(10,"teste_int_overflow");
     #endif

 3:    if (n>MAX_ML_INT) {
 4:      gchar *z[2];
 5:      gint i;

 6:      retval->typ=FEHLER_TYP;
         /* Warnung anzeigen */
```

```
 7:       z[0]=g_strdup(TEXT_INT_OVERFLOW0);
          z[1]=g_strdup_printf(TEXT_INT_OVERFLOW1,n,MAX_ML_INT);

 8:       Warnungs_Fenster_oeffnen(1,BUTTON_WARNUNG_OK_TEXT,z[0],z[1],NULL);

 9:       for(i=0;i<2;i++) g_free(z[i]);

10:       erg=TRUE;
        }
11:     else erg=FALSE;

      #ifdef KIEL_DEBUG
        kiel_debug_out(10,"teste_int_overflow");
      #endif
12:     return(erg);
      }
```

Auszug aus der Datei `kiel_ausw.c` — Ende

ad 1: Die Funktion `teste_int_overflow` hat einen Rückgabewert vom Typ `gboolean`. Es soll `TRUE` genau dann zurückgegeben werden, wenn das erste Argument einen `int`-Überlauf erzeugen würde.

ad 2: Die Variable `erg` dient zur Aufnahme des booleschen Wertes, der am Ende der Funktion der aufrufenden Funktion übergeben werden soll.

ad 3: `MAX_ML_INT` wird in der Datei `kiel_ausw_priv.h` wie folgt ein Wert zugewiesen:

```
#define MAX_ML_INT 1073741823 /* =2^30-1 */
```

Der Wert $1073741823 = 2^{30} - 1$ stellt den größten Wert dar, den eine `int`-Variable aufnehmen kann. Hier wird getestet, ob das erste Argument diesen Wert übersteigt.

ad 4: Ist das erste Argument größer als der maximale Wert einer `int`-Variablen, so wird ein Warnungsfenster geöffnet, und dem zweiten Argument wird ein Fehlerwert zugewiesen. Zur Aufnahme des Warnungstextes wird das Variablenfeld `z` deklariert.

ad 5: Die `int`-Variable `i` wird beim Freigeben des Variablenfeldes `z` benutzt.

ad 6: Der Typ des zweiten Arguments wird auf `FEHLER_TYP` gesetzt.

ad 7: Das Variablenfeld `z` wird mit Warnungstext aus der Datei `kiel_ressource.h` gefüllt.

ad 8: Es wird ein Warnungsfenster mit Hilfe der Operation `Warnungs_Fenster_oeffnen` geöffnet.

ad 9: Die Zeichenreihen des Feldes `z` werden freigegeben.

ad 10: Da das erste Argument zu einem Überlauf geführt hätte, wird die Ergebnisvariable `erg` auf den Wert `TRUE` gesetzt.

ad 11: Falls das erste Argument nicht größer als der maximal zulässige `int`-Wert ist, wird die Ergebnisvariable `erg` auf den Wert `FALSE` gesetzt, da nun eine Übernahme stattfinden kann.

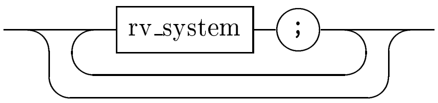
ad 12: Der Inhalt der Ergebnisvariablen wird als Ergebnis von `teste_int_overflow` der aufrufenden Operation übergeben.

Anhang A

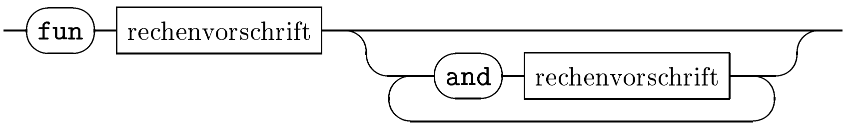
Syntaxdiagramme

A.1 Aufbau einer Rechenvorschriftendatei

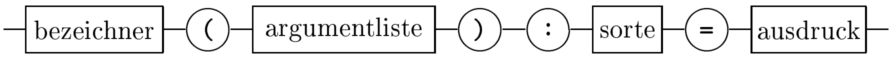
rvprogramm



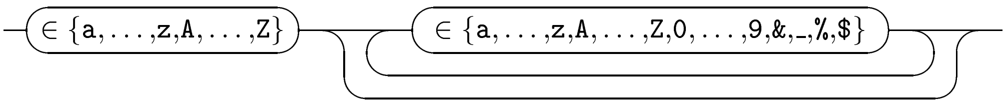
rv_system



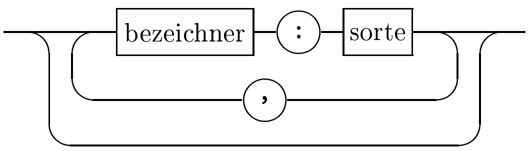
rechenvorschrift



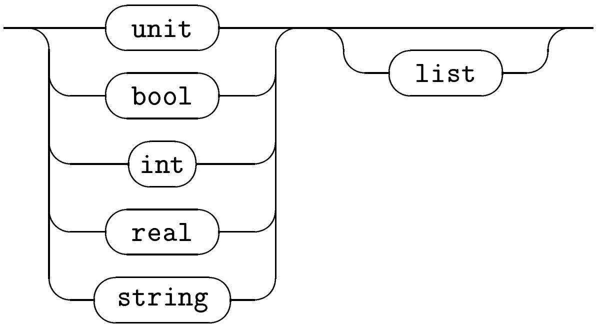
bezeichner



argumentliste



sorte



unit_ebene1

bool_ebene3

int_ebene3

real_ebene3

string_ebene2

unit_list_ebene3

bool_list_ebene3

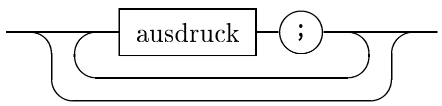
int_list_ebene3

real_list_ebene3

string_list_ebene3

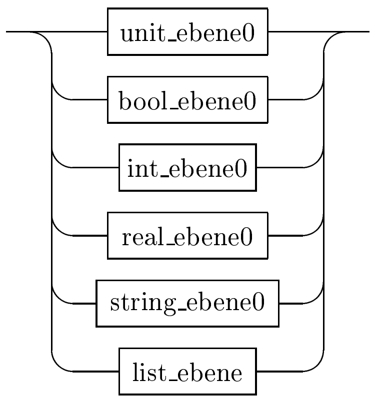
A.2 Aufbau einer Ausdruckdatei

adkprogramm

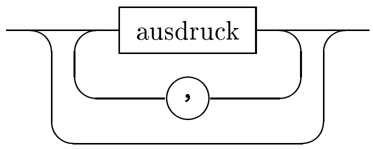


A.3 Aufbau eines Ausdrucks

ausdruck

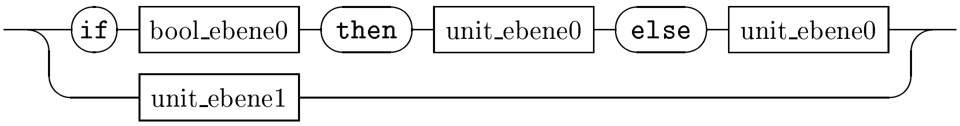


ausdruckliste

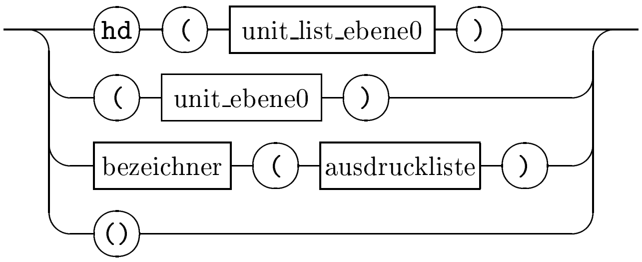


A.3.1 Aufbau von `unit`-Ausdrücken

*unit_ebene*0

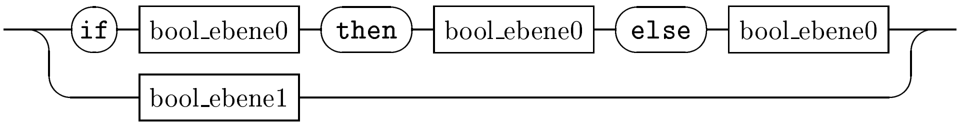


*unit_ebene*1

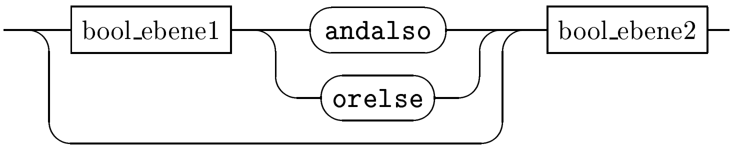


A.3.2 Aufbau von `bool`-Ausdrücken

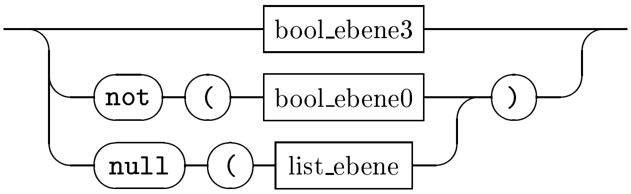
*bool_ebene*0



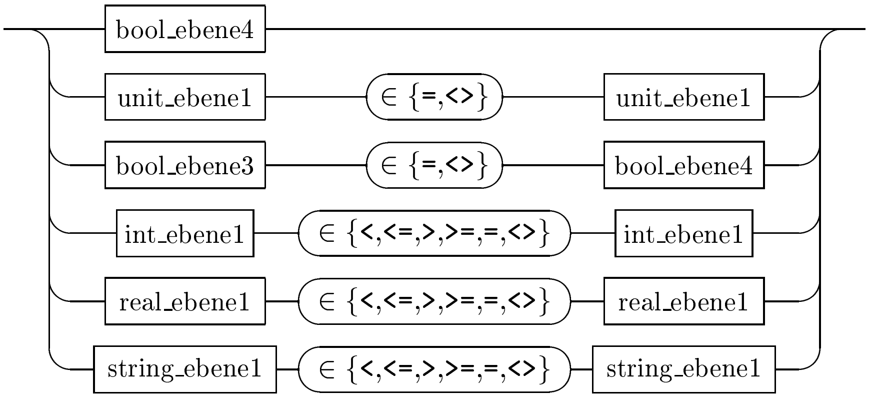
*bool_ebene*1



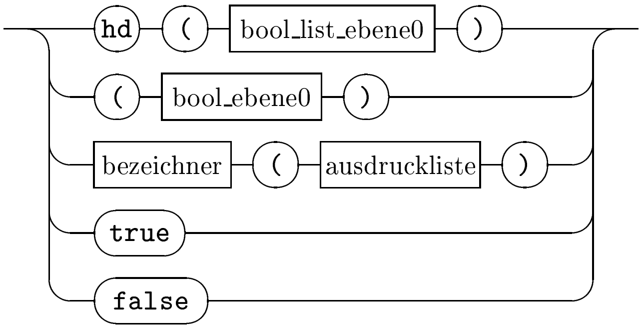
*bool_ebene*2



bool_ebene3

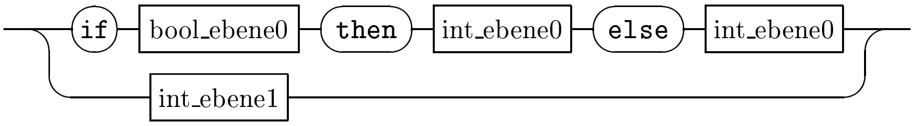


bool_ebene4

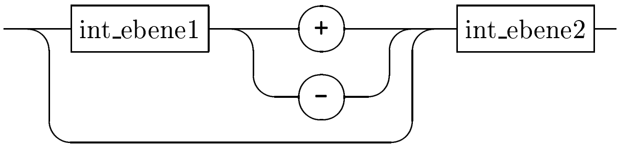


A.3.3 Aufbau von `int`-Ausdrücken

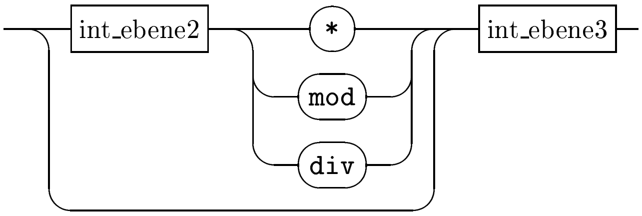
int_ebene0



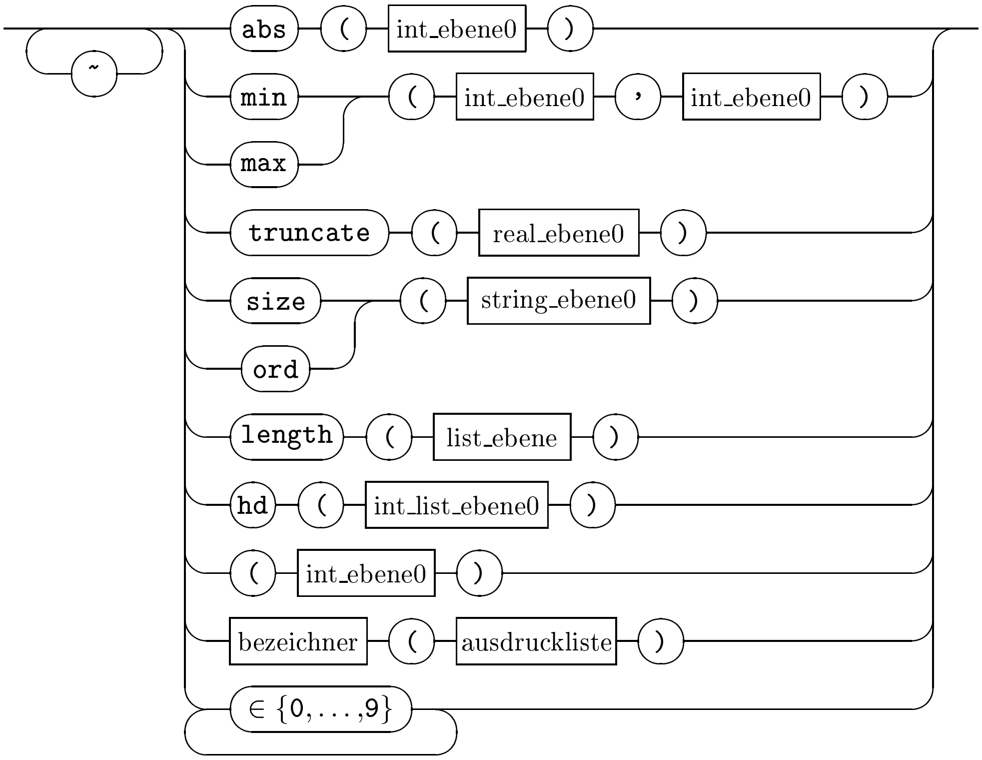
int_ebene1



int_ebene2

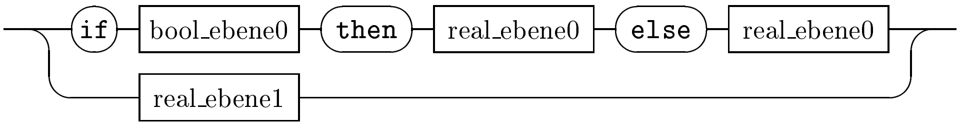


int_ebene3

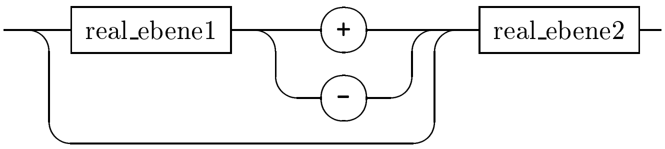


A.3.4 Aufbau von `real`-Ausdrücken

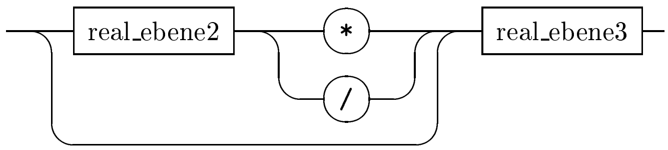
real_ebene0



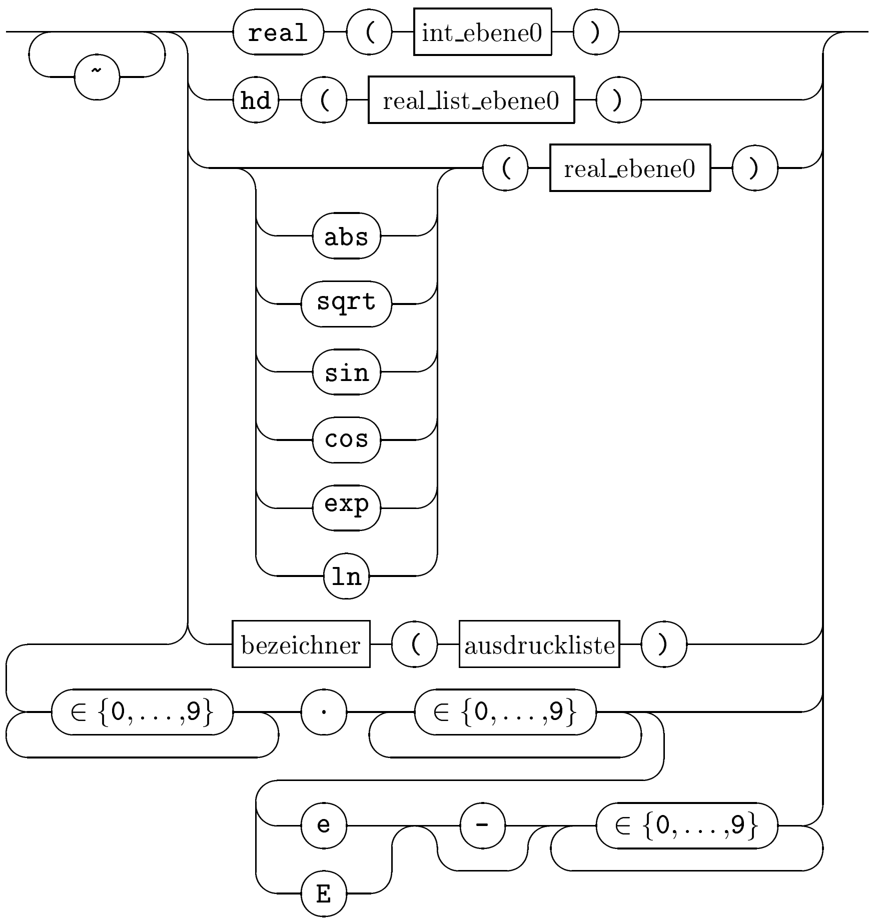
real_ebene1



real_ebene2

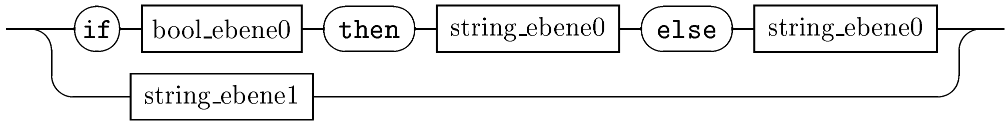


real_ebene3

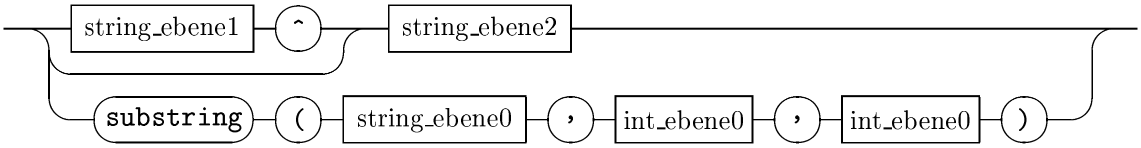


A.3.5 Aufbau von `string`-Ausdrücken

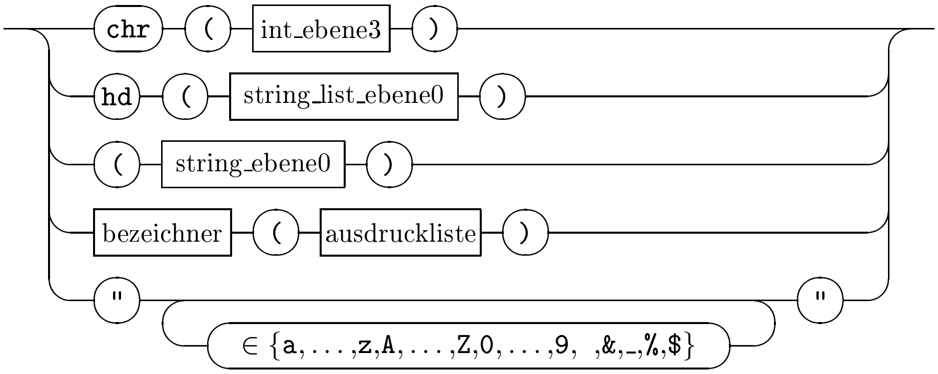
string_ebene0



string_ebene1

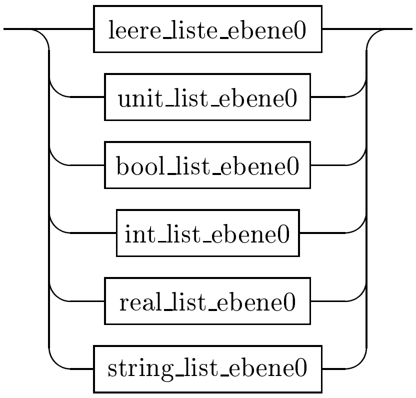


string_ebene2



A.3.6 Aufbau von Listen-Ausdrücken

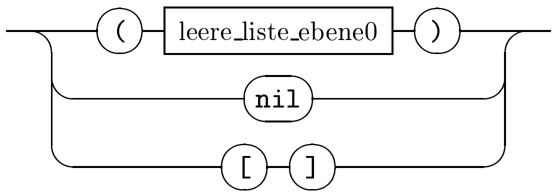
list_ebene



A.3.6.1 Aufbau der leeren Liste

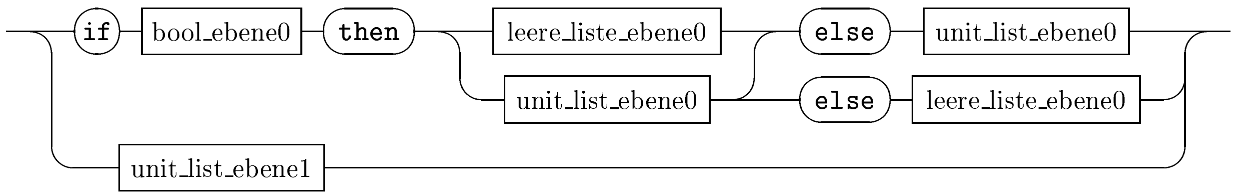
leere_liste_ebene0

*leere_liste_ebene*1

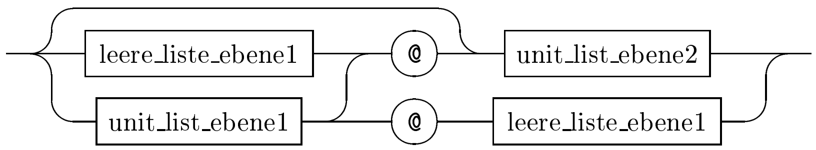


A.3.6.2 Aufbau von `unit`-Listen-Ausdrücken

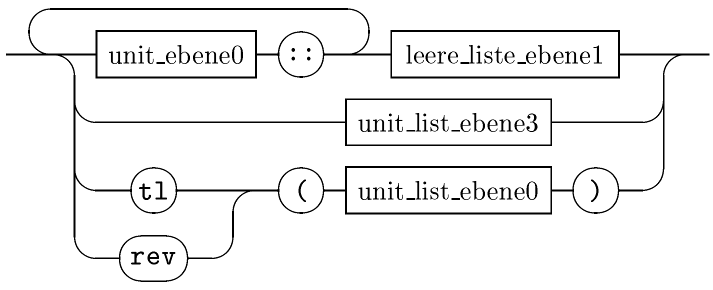
*unit_list_ebene*0



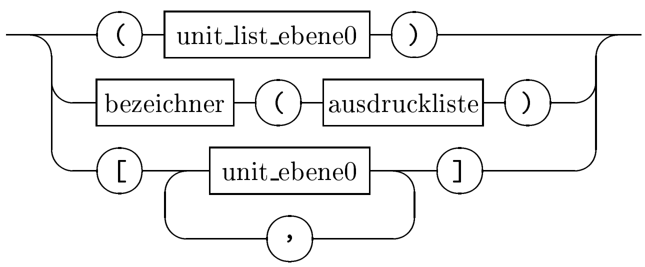
*unit_list_ebene*1



*unit_list_ebene*2

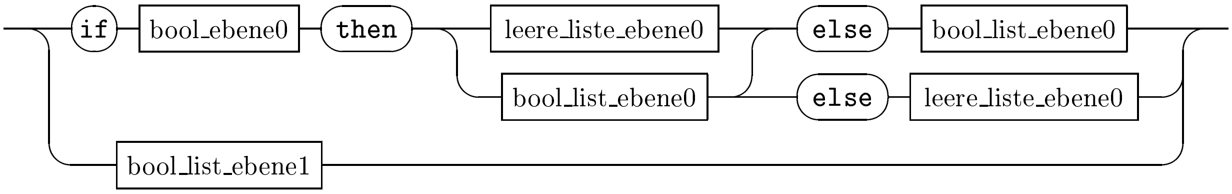


*unit_list_ebene*3

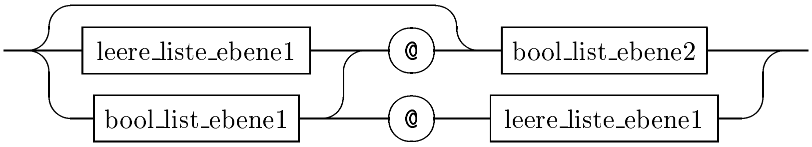


A.3.6.3 Aufbau von `bool`-Listen-Ausdrücken

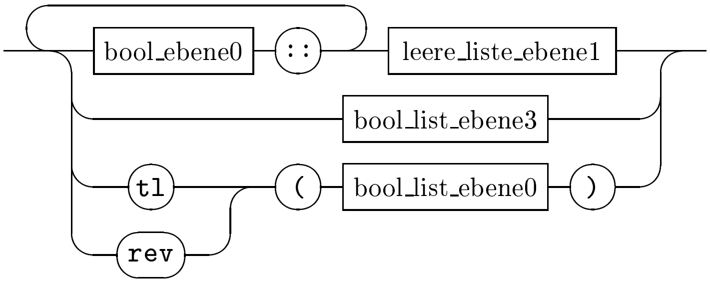
bool_list_ebene0



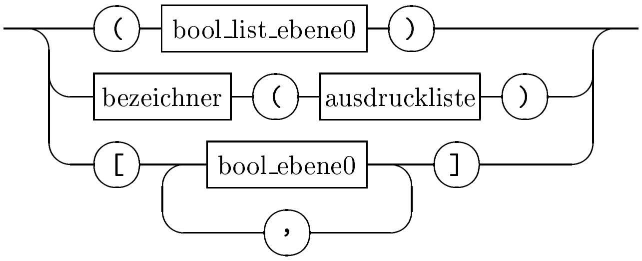
bool_list_ebene1



bool_list_ebene2

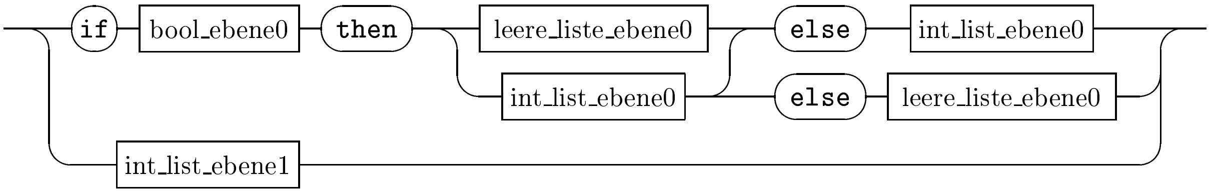


bool_list_ebene3

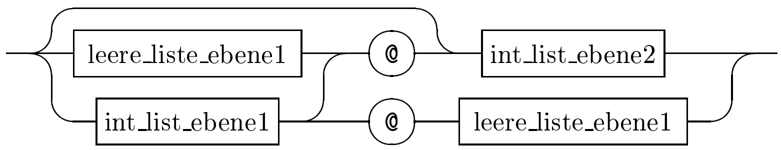


A.3.6.4 Aufbau von `int`-Listen-Ausdrücken

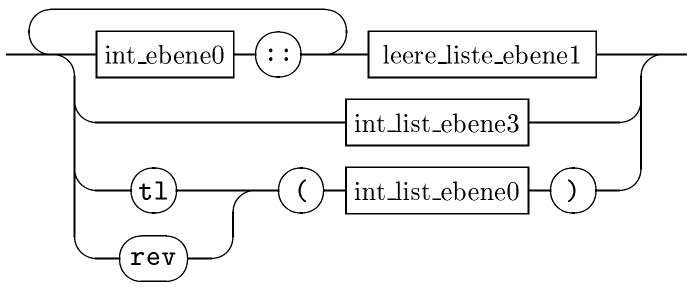
int_list_ebene0



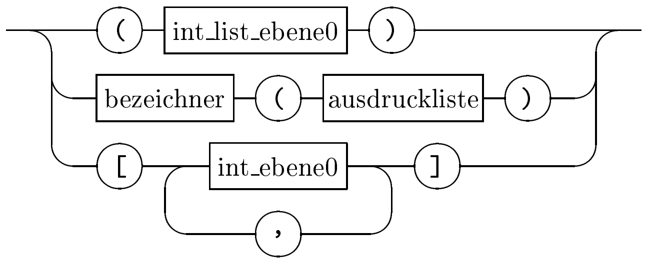
int_list_ebene1



int_list_ebene2

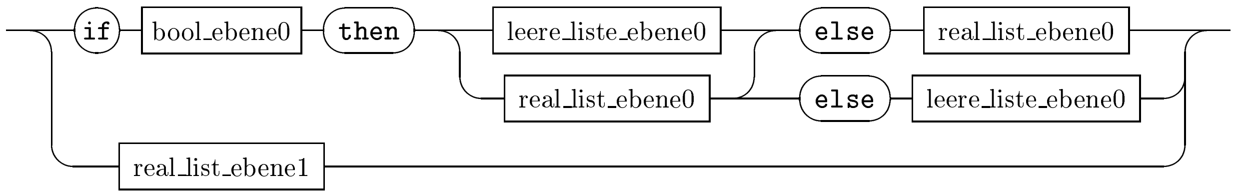


int_list_ebene3

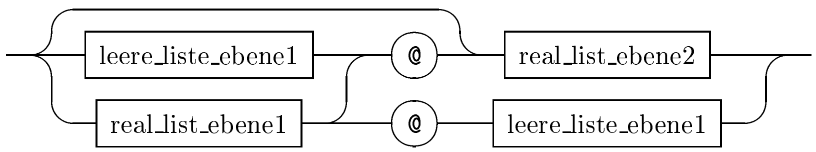


A.3.6.5 Aufbau von `real`-Listen-Ausdrücken

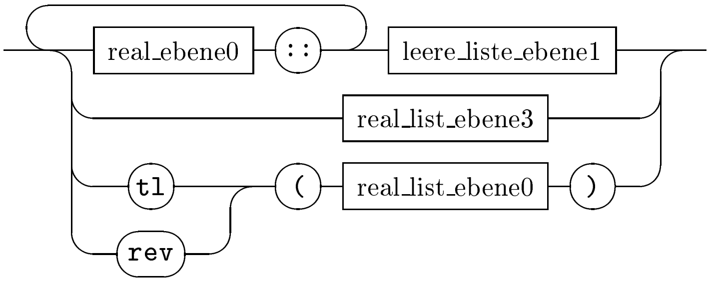
real_list_ebene0



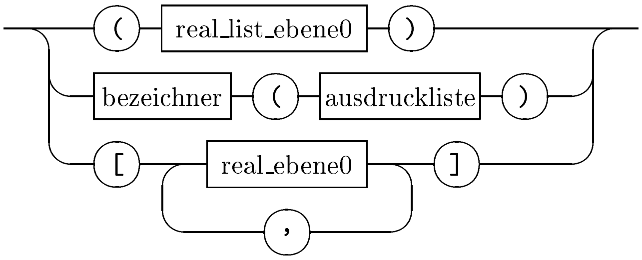
real_list_ebene1



real_list_ebene2

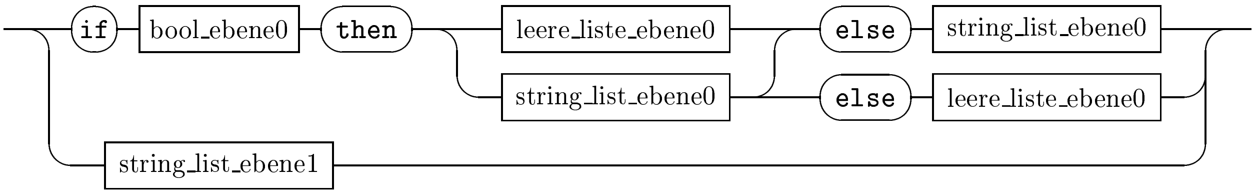


real_list_ebene3

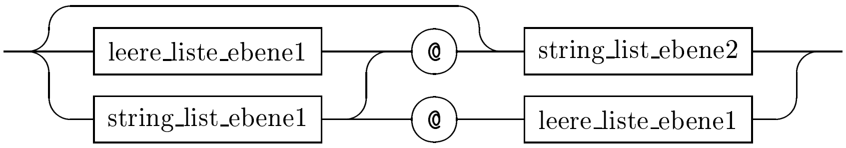


A.3.6.6 Aufbau von `string`-Listen-Ausdrücken

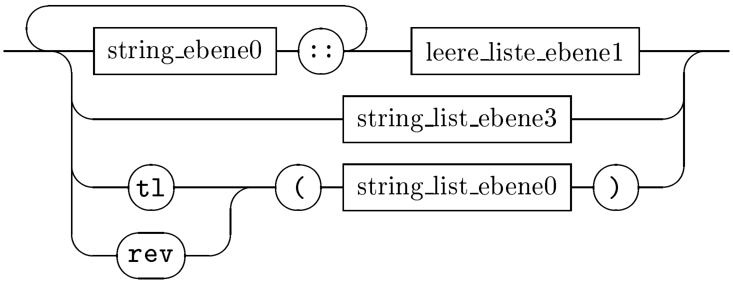
string_list_ebene0



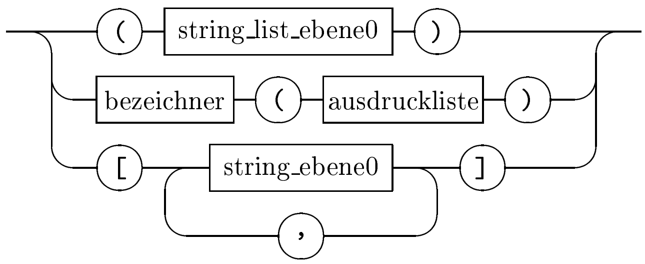
string_list_ebene1



string_list_ebene2



string_list_ebene3



Literaturverzeichnis

[Bergh95] R. Berghammer: Vorlesungsskript Informatik I an der Christian-Albrechts-Universität zu Kiel, 1995.

[GLib99] GLib Reference Manual, 1999, via `www.gtk.org/rdp/glib/book1.html`.

[GDK99] GDK Reference Manual, 1999, via `www.gtk.org/rdp/gdk/book1.html`.

[GTK99] T. Gale und I. Main: GTK v1.2 Tutorial, 10. April 1999, via `www.gtk.org/tutorial`.

[LexYacc97] 08208 - Introduction to Lex and Yacc, 1997, via `www.engmath.hull.ac.uk/people/pjp/Teaching/08208-9697/08208IntroductoryLab.html`.

[LoeSie84] J. Loechx und K. Sieber: The Foundations of Program Verification. Teubner, 1984.

[Manna74] Z. Manna: Mathematical Theory of Computation. McGraw-Hill Inc., 1974.

[Meier96] S. Meier: Algorithmen zum schönen Zeichnen von Graphen und deren Integration in das RELVIEW-System. Diplomarbeit an der Christian-Albrechts-Universität zu Kiel, 1996.

[SML93] Standard ML of New Jersey - Base Environment, 15. Februar 1993.

[Ull94] J. D. Ullman: Elements of ML Programming. Prentice Hall, 1994.

[WetSha79] C. Wetherell und A. Shannon: Tidy Drawings of Trees. IEEE Trans. Software Eng., Band SE-5, Nr. 5, S. 514-520, 1979.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, im Juli 1999