Benutzergesteuerte Termersetzung - Erweiterung von KIEL um *datatype*-Deklarationen und die Deklaration von Rechenvorschriften mit Mustererkennung

Diplomarbeit
vorgelegt von
Helge Ernst
am 13.März.2001

Betreuer: Prof. Dr. R. Berghammer

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Im Rahmen einer Diplomarbeit [1] wurde das KIEL-System an der Christian-Albrechts-Universität zu Kiel entwickelt. Der Name KIEL steht für *Kiel Interactive Evaluation Laboratory*. Es handelt sich dabei um ein Programm, mit dem Ausdrücke benutzergesteuert ausgewertet werden können. Die Interaktion mit dem Anwender geschieht über eine graphische Bedienoberfläche. Diese graphische Oberfläche ermöglicht es weiterhin, Ausdrücke als Bäume zu visualisieren. Benutzergesteuerte Auswertung bedeutet, dass der Anwender den als Baum dargestellten Ausdruck in Einzelschritten an jedem beliebigen Baumknoten auswerten kann. Andererseits bietet KIEL auch die Visualisierung der Auswertung nach verschiedenen Auswertungsstrategien, die dann nicht mehr vollständig vom Benutzer selbst gesteuert wird, da die Auswertung dann immer an der Wurzel des als Baum dargestellten Ausdrucks gestartet werden muss.

Die Ausdrücke, die KIEL auswerten kann, müssen einer Teilsprache der funktionalen Programmiersprache Standard ML angehören. Der Ursprung von ML ist, an der Universität Edinburgh gegen Ende der 70er Jahre zu finden. Dort wurde für ein größeres Programmsystem mit Namen LCF eine Implementierungssprache entwickelt, deren Name *Meta Language for LCF* lautet. Aus ihr entstand das heutige Standard ML.

Die von KIEL akzeptierte Teilsprache besteht im Wesentlichen aus Konstanten der fünf Sorten *unit*, *bool*, *int*, *real*, *string* und Listen über diesen Sorten, zugehörigen Basisoperatoren und -funktionen, der Verzweigung und rekursiven Rechenvorschriften.

Diese Arbeit befasst sich mit der Erweiterung der von KIEL akzeptierten Sprache um zusammengesetzte Datenstrukturen, Rechenvorschriften mit Mustererkennung und geschachtelte Listen. Sie gliedert sich wie folgt:

Kapitel 2 beschreibt die akzeptierte Sprache, die Bedienung und die Limitierung des bisherigen KIEL-Systems. Die Kapitel 3, 4 und 5 befassen sich mit der Erweiterung von KIEL. Zuerst werden die Sprachmittel eingeführt, um die KIEL erweitert wird, dann werden diese Anhand von Beispielauswertungen veranschaulicht und schließlich einige Details der Implementierung angesprochen. Kapitel 6 bildet den Abschluss der Arbeit mit einer Zusammenfassung und einem Ausblick auf die zukünftige Weiterentwicklung von KIEL.

Kapitel 2

Bisheriges KIEL-System

Das folgende Kapitel gibt einen Überblick über das bisherige KIEL-System. Begonnen wird mit der Motivation, ein solches System zur Visualisierung der Auswertung von Ausdrücken unter verschiedenen Auswertungsstrategien zu entwickeln. Anschließend wird die Sprache, die KIEL akzeptiert, und die Bedienung der graphischen Oberfläche beschrieben. Schließlich wird auf die Einschränkungen des bisherigen KIEL-Systems eingegangen.

2.1 Motivation für die Entwicklung von KIEL

Das Auswerten von Ausdrücken gehört zu den grundlegenden Aufgaben in der Informatik. Dementsprechend wichtig ist es zu wissen, dass die Auswertung desselben Ausdrucks unter verschiedenen Auswertungsstrategien zu unterschiedlichen Resultaten führen kann. Um dieses Verständnis zu erlangen, ist es fast unumgänglich, ausgesuchte Beispielausdrücke Schritt für Schritt unter den verschiedenen Auswertungsstrategien zu berechnen und zu vergleichen. Eine handschriftliche Auswertung stößt hier schnell an die Grenze des Zumutbaren und ist des Weiteren auch anfällig für Flüchtigkeitsfehler.

Hier soll das KIEL-System, welches im Rahmen der Diplomarbeit von M. Thiedt [1] entwickelt wurde, Abhilfe schaffen. Es ermöglicht dem Anwender, Ausdrücke einer grundlegenden Teilmenge der funktionalen Programmiersprache Standard ML schrittweise oder vollständig unter verschiedenen gängigen Auswertungsstrategien zu evaluieren. Dazu wird der zu berechnende Ausdruck in einem Fenster als Baum dargestellt. Nach Wahl einer Auswertungsstrategie kann nun der Ausdruck Schritt für Schritt per Mausklick berechnet werden. Das Ergebnis der Einzelschritte wird umgehend im Fenster angezeigt.

Der Benutzer hat aber auch die Möglichkeit, an jedem beliebigen Knoten im Baum die Auswertung per Mausklick voranzutreiben. Auch hier wird das Ergebnis dieser Auswertung umgehend im Fenster angezeigt und der Anwender kann mit der weiteren

Auswertung an beliebiger Stelle fortfahren. Dieses Verfahren gibt dem Benutzer somit Gelegenheit, auch eigene Auswertungsstrategien zu entwickeln.

Das KIEL-System bietet weiterhin die Möglichkeit, als Debug-Tool für Programme aus der Teilmenge von Standard ML zu fungieren. Dazu muss allerdings die Auswertungsstrategie gewählt werden, die für Standard ML-Programme vorgeschrieben ist. Dann kann die Ursache für ein nicht terminierendes Standard ML-Programm einfacher mit Hilfe von KIEL aufgespürt werden als mit einem Standard ML-System selbst, da sich das Programm mit KIEL schrittweise auswerten lässt. [1] Einen Überblick über das KIEL-System gibt Abbildung 2.1.

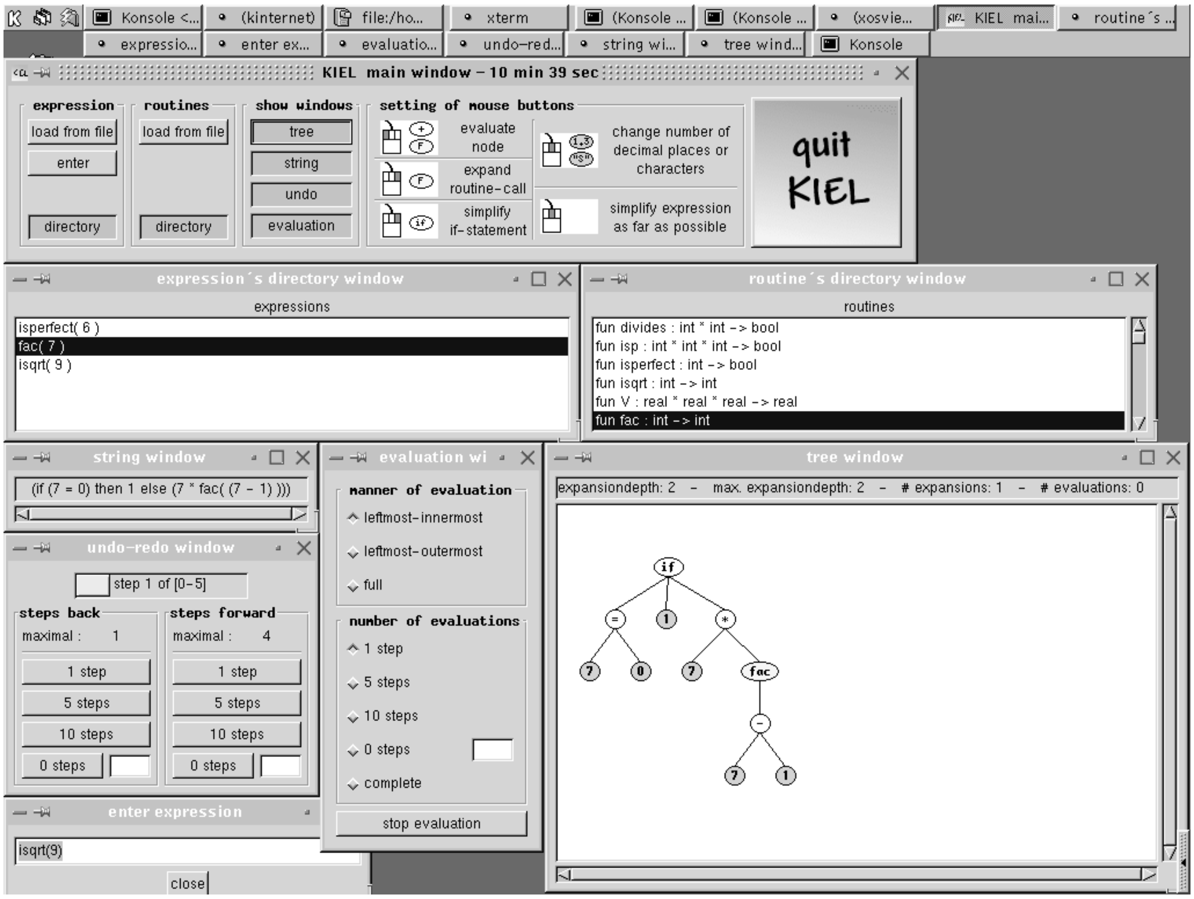


Abbildung 2.1: Das KIEL-System im Überblick

Für das Internet existiert mit KIEL-Online eine Version von KIEL, die in jedem java-fähigen Webbrowser ausgeführt werden kann. KIEL-Online entstand in der Diplomarbeit von O. Lorenz [3] und bietet fast die gleiche Funktionalität wie das in diesem Kapitel vorgestellte KIEL-System. Da KIEL-Online innerhalb eines Webbrowsers

[1] Standard ML schreibt die $leftmost - innermost$ Strategie vor.

nur als Applet ausgeführt werden kann, sind ihm keine Zugriffe auf lokale Dateien des Rechners gestattet, auf dem es gerade ausgeführt wird. Daher müssen alle Eingaben per Hand durchgeführt werden. Im Internet ist KIEL-Online zu finden unter:
`http://www.informatik.uni-kiel.de/~progsys/kiel/kiel_applet.html`

2.2 Von KIEL akzeptierte Sprache

Wie schon im vorherigen Abschnitt erwähnt, implementiert das KIEL-System nicht den gesamten Sprachumfang von Standard ML, sondern nur eine Teilmenge davon. Diese Teilmenge wird nun kurz und informal beschrieben. Eine ausführliche Beschreibung der zugrunde liegenden Grammatik befindet sich in [1] und [3] in Form von Syntaxdiagrammen. Die Semantik der implementierten Basisoperatoren und -funktionen entspricht der von Standard ML vorgeschriebenen und kann in [5] nachgeschlagen werden.

2.2.1 Elementare Datenstrukturen

Im folgenden werden die fünf Sorten *unit*, *bool*, *int*, *real*, *string*, die die Basis aller in KIEL implementierten Typen[2] bilden, und die fünf aus ihnen konstruierten Listentypen *unit list*, *bool list*, *int list*, *real list* und *string list* samt ihrer Basisoperatoren oder -funktionen vorgestellt.

Einelementiger Bereich
Die einelementige Sorte *unit* besitzt nur eine Konstante, und zwar (). Es existieren weder Basisoperatoren noch -funktionen.

Wahrheitswerte
Die Sorte *bool* implementiert die Wahrheitswerte und besitzt die beiden Konstanten $true$ und $false$. Die Basisoperatoren auf Wahrheitswerten sind:

not:	$bool \to bool$	Negation
andalso:	$bool \times bool \to bool$	sequentielle Konjunktion
orelse:	$bool \times bool \to bool$	sequentielle Disjunktion
$=, <>$:	$bool \times bool \to bool$	Vergleichsoperatoren

Ganze Zahlen
Eine Teilmenge der Ganzen Zahlen wird in Standard ML durch die Sorte *int* repräsentiert. Die Konstanten dieser Sorte liegen im Intervall $[-2^{30}, 2^{30} - 1]$, wobei negativen

[2] jede Sorte ist auch ein Typ

Zahlen nicht das übliche "$-$" sondern ein "$\sim$" vorangestellt wird. Dies dient der besseren Unterscheidung zwischen der einstelligen Negation ($\sim$) und der zweistelligen Subtraktion ($-$).
Beispiele:

- 59228
- $\sim$1879

Folgende Basisoperatoren und -funktionen sind auf der Sorte *int* realisiert:

$\sim$:	$int \to int$	Negation
$+, -, *$:	$int \times int \to int$	übliche Arithmetik
div:	$int \times int \to int$	ganzzahlige Division
mod:	$int \times int \to int$	Divisionsrest
min, max:	$int \times int \to int$	Minimum und Maximum
abs:	$int \to int$	Absolutbetrag
$=, <>, <, <=, >, >=$:	$int \times int \to bool$	Vergleichsoperatoren

Reelle Zahlen
Der Menge der darstellbaren Reellen Zahlen ist beschränkt auf Maschinenzahlen im Intervall von $[-10^{308}, 10^{308}]$. Der Name der zugehörigen Sorte lautet *real*. Wie bei den Ganzen Zahlen wird negativen Zahlen zur besseren Unterscheidung von einstelliger Negation ($\sim$) und zweistelliger Subtraktion ($-$) das "$\sim$" statt des "$-$" vorangestellt. Die folgenden drei Beispielzahlen verdeutlichen, wie Zahlen der Sorte *real* angegeben werden können:

- 1.23456
- $\sim 1e308$
- $1.23456e\sim 78$

Auf der Sorte *real* sind folgende Basisoperatoren und -funktionen definiert:

$\sim$:	$real \to real$	Negation
$+, -, *, /$:	$real \times real \to real$	übliche Arithmetik
abs:	$real \to real$	Absolutbetrag
$sqrt$:	$real \to real$	Quadratwurzel
$=, <>, <, <=, >, >=$:	$real \times real \to bool$	Vergleichsoperatoren
sin, cos:	$real \to real$	Sinus und Cosinus
exp, ln:	$real \to real$	e-Funktion und natürlicher Logarithmus
$real$:	$int \to real$	Umwandlung in reelle Zahl
$truncate$:	$real \to int$	Abhackung der Nachkommastellen

Zeichenreihen
Zeichenreihen werden in KIEL mit der Sorte *string* bezeichnet. Dabei handelt es sich um eine Kette von Zeichen, die von Anführungszeichen eingeschlossen ist.
Beispiel:

- $''azAZ09''$
- $''+-*/@\_''$

Folgende Basisoperatoren und -funktionen sind implementiert:

size:	$string \rightarrow int$	Länge der Zeichenkette
$=, <>, <, <=,$		
$>, >=$:	$string \times string \rightarrow bool$	lexikographische Vergleichsoperatoren
ˆ:	$string \times string \rightarrow string$	Konkatenation
ord:	$string \rightarrow int$	ASCII-Code des ersten Zeichens
chr:	$int \rightarrow string$	ASCII-Code in Zeichen wandeln
substring:	$string \times int \times int \rightarrow string$	Teilstring ab Position mit Länge

Listen
Das bisherige KIEL-System implementiert nur sehr einfache Listen. Es handelt sich dabei um ungeschachtelte Listen über einer der fünf schon vorgestellten Sorten. Bzw. es existieren neben den fünf Sorten die fünf folgenden Typen:
unit list, *bool list*, *int list*, *real list* und *string list*.

Die Notation einer Liste beginnt mit “[”, wird gefolgt von den einzelnen Elementen, die durch “,” von einander getrennt werden, und endet mit “]”. Die leere Liste kann dementsprechend durch [ ] dargestellt werden oder aber durch die Konstante *nil*.
Beispiel:

- *nil*
- $[\,]$
- $[1, 2, 3, 4, 5]$
- $[''a'','' b'','' c'']$
- $[true, false]$

Auf allen Listentypen existieren folgende Basisoperatoren und -funktionen, wobei m jeweils einer der fünf Sorten entspricht:

$::$:	$m \times m\ list \rightarrow m\ list$	links anfügen
hd:	$m\ list \rightarrow m$	erstes Element
tl:	$m\ list \rightarrow m\ list$	Restliste ohne erstes Element
$null$:	$m\ list \rightarrow bool$	Test ob Liste leer ist
$length$:	$m\ list \rightarrow int$	Länge der Liste
$@$:	$m\ list \times m\ list \rightarrow m\ list$	Konkatenation
rev:	$m\ list \rightarrow m\ list$	Umkehrung der Liste
$=, <>$:	$m\ list \times m\ list \rightarrow bool$	Vergleichsoperatoren

2.2.2 Ausdrücke

In KIEL können Ausdrücke aus Konstanten, Basisoperatoren, Basisfunktionen, Aufrufen bereits deklarierter Rechenvorschriften und der Verzweigung aufgebaut werden.

Der Aufruf einer Rechenvorschrift f mit k formalen Parametern hat die Form

$$f(e_1, \ldots, e_k)$$

wobei alle e_i wiederum Ausdrücke sind $(1 \leq i \leq k)$. Bei der Verzweigung

$$if\ c\ then\ e_1\ else\ e_2$$

sind sowohl die Bedingung c als auch die beiden Alternativen e_1 und e_2 wiederum Ausdrücke, wobei c von der Sorte *bool* ist und e_1 und e_2 von einem beliebigen aber beide vom gleichen Typ sind.

Im Rumpf einer Rechenvorschrift können zusätzlich die formalen Parameter dieser Rechenvorschrift und Aufrufe dieser Rechenvorschrift selbst zum Aufbau eines Ausdrucks verwendet werden. Bei Systemen von Rechenvorschriften dürfen zusätzlich auch Aufrufe alle weiteren im System deklarierten Rechenvorschriften benutzt werden. Siehe dazu auch 2.2.3

2.2.3 Rechenvorschriften

Wie in Standard ML, so besteht auch im KIEL-System ein Programm aus einer Reihe aufeinander folgender Deklarationen von Rechenvorschriften und Systemen von Rechenvorschriften. Eine einzelne Rechenvorschrift hat dabei schematisch die folgende Form:

$$fun\ f(x_1 : m_1, \ldots, x_k : m_k) : m = b;$$

Dabei darf $k = 0$ sein. Der Rumpf b vom Typ m einer so deklarierte Rechenvorschrift f mit den formalen Parametern x_i vom Typ m_i ist ein Ausdruck im Sinne von 2.2.2.

Innerhalb dieses Rumpfes b dürfen sowohl Funktionsaufrufe bereits deklarierter Rechenvorschriften als auch von f selbst auftreten. Als konkretes Beispiel ist im folgenden eine Rechenvorschrift zur Berechnung der Fakultät angegeben:

```
fun fak(n: int): int =
  if n = 0
  then 1
  else n * fak(n-1);
```

Sollen sich mehrere Rechenvorschriften wechselseitig im Rumpf aufrufen, so muss ein System von Rechenvorschriften deklariert werden. Solch ein System sieht schematisch wie folgt aus:

$$\begin{aligned} &fun\ f_1(x_{11} : m_{11}, \ldots, x_{1k} : m_{1k_1}) : m_1 = b_1 \\ &and\ f_2(x_{21} : m_{21}, \ldots, x_{2k} : m_{2k_2}) : m_2 = b_2 \\ &\qquad\vdots \\ &and\ f_j(x_{j1} : m_{j1}, \ldots, x_{jk} : m_{jk_j}) : m_j = b_j\,; \end{aligned}$$

Dabei müssen $j \geq 2$ und $\forall\, 1 \leq i \leq j : k_i \geq 0$ gelten. Im folgenden Beispielprogramm zur Berechnung, ob eine Zahl gerade bzw. ungerade ist, muss ein System von zwei Rechenvorschriften verwendet werden:

```
fun even(n: int): bool =
  if abs(n) = 0
  then true
  else odd(abs(n) - 1)

and odd(n: int): bool =
  if abs(n) = 0
  then false
  else even(abs(n) - 1);
```

2.3 Auswertung

Die Auswertung eines Ausdrucks besteht in der Regel aus mehreren Auswertungsschritten. Ein Auswertungsschritt überführt einen Ausdruck e_0 in einen Ausdruck e_1, indem in e_0 ein Teilausdruck durch einen anderen Ausdruck ersetzt wird. Durch welchen Ausdruck ein Teilausdruck ersetzt wird, hängt vom Teilausdruck ab. Beispielsweise wird der Ausdruck $1 + 2 * 3$ überführt in $1 + 6$, indem der Teilausdruck $2 * 3$ durch den Ausdruck 6 ersetzt wird.

Es gibt zwei verschiedene Arten von Auswertungsschritten, nämlich Vereinfachungs- und Expansionsschritte.

2.3.1 Vereinfachungsschritt

Vereinfachungsschritte können angewendet werden auf Basisoperatoren und -funktionen, wenn die dazugehörigen Operanden bzw. Argumente schon vollständig ausgewertet sind, und auf die Verzweigung, wenn die Bedingung vollständig ausgewertet ist. Ist das der Fall, werden Basisoperatoren und -funktionen durch ihren von den Operatoren bzw. ihren Argumenten abhängigen Wert ersetzt. Eine Verzweigung wird durch die entsprechende Alternative ersetzt. Beispiele:
$1 + 2$ wird ersetzt durch 3
$hd([1.3, 2.2])$ wird ersetzt durch 1.3
$if\ true\ then\ e_1\ else\ e_2$ wird ersetzt durch e_1

2.3.2 Expansionsschritt

In einem Expansionsschritt wird der Aufruf einer Rechenvorschrift ersetzt durch den an die Argumente angepassten Rumpf dieser Rechenvorschrift. Die Rechenvorschrift f sei durch die folgende schematische Deklaration gegeben:

$$fun\ f(x_1 : m_1, \dots , x_k : m_k) : m = b;$$

Seien $e_1, \dots , e_k$ Ausdrücke von den Typen $m_1, \dots , m_k$, dann kann der Aufruf der Rechenvorschrift

$$f(e_1, \dots , e_k)$$

mittels eines Expansionsschrittes ersetzt werden. Dazu werden zunächst die Argumente $e_1, \dots , e_k$ an die Variablen $x_1, \dots , x_k$ gebunden. Dann wird der gesamte Aufruf $f(e_1, \dots , e_k)$ ersetzt durch den Rumpf b, wobei in b jedoch alle Vorkommen der Variablen $x_1, \dots , x_k$ durch die Ausdrücke ersetzt werden, die zuvor an diese Variablen gebunden wurden. Ein Beispiel:
Deklaration:

```
fun fak(n: int): int =
  if n = 0
  then 1
  else n * fak(n - 1);
```

Der Ausdruck `fak(1+2)` wird in einem Expansionsschritt zu `if 1+2 = 0 then 1 else 1+2 * fak(1+2 - 1)`, wobei das Argument `1+2` an die Variable `n` gebunden wurde und sie anschließend im Rumpf ersetzt hat.

2.3.3 Auswertungsstrategien

Zur Auswertung eines Ausdruck e wird eine Folge von Ausdrücken nach dem folgenden Schema konstruiert:

1. Setze $i := 0$; $e_i := e$.
2. Bilde e_{i+1} aus e_i durch Anwendung aller möglichen Vereinfachungsschritte. Enthält e_{i+1} keine Aufrufe von Rechenvorschriften mehr, dann stoppe, sonst setze $i := i + 1$.
3. Bilde e_{i+1} aus e_i durch simultane Anwendung von Expansionsschritten. Setze $i := i + 1$.

Ist die konstruierte Folge endlich, so ist das letzte Folgenglied das Ergebnis der Auswertung, andernfalls ist es undefiniert.

Entscheidend für das Ergebnis der Auswertung ist jedoch die Auswertungsstrategie, die bestimmt, welche Rechenvorschriftaufrufe in Punkt 3 des Schemas expandiert werden. KIEL unterstützt die drei Auswertungsstrategien

- $leftmost - innermost$,
- $leftmost - outermost$ und
- $full$

Welche Rechenvorschriftaufrufe im Baum des Ausdrucks ersetzt werden, wird im folgenden festgelegt:

$leftmost - innermost$	der Aufruf, der bei einem Tiefensuche-Durchlauf im Baum, der erste ist, der keinen Rechenvorschriftaufruf in den Argumenten enthält
$leftmost - outermost$	der Aufruf, der bei einem Tiefensuche-Durchlauf im Baum, zuerst besucht wird
$full$	alle Aufrufe

Dazu sei angemerkt, dass eine Tiefensuche die Nachfolger eines Knotens von links nach rechts besucht.

Im folgenden Beispiel sind diejenigen Rechenvorschriftaufrufe markiert, die simultan in Punkt 3 des Schemas zu expandieren sind.

$leftmost - innermost$	$f(0, \mathbf{f}(1, 1)) + f(f(2, 2), f(3, 3))$
$leftmost - outermost$	$\mathbf{f}(0, f(1, 1)) + f(f(2, 2), f(3, 3))$
$full$	$\mathbf{f}(0, \mathbf{f}(1, 1)) + \mathbf{f}(\mathbf{f}(2, 2), \mathbf{f}(3, 3))$

Während in Standard ML die $leftmost - innermost$ Auswertungsstrategie vorgeschrieben ist, kann im KIEL-System der Benutzer zwischen allen drei vorgestellten auswählen.

2.4 Bedienung und Anwendung

Das KIEL-System wird über eine graphische Oberfläche mittels Maus bedient. Mit ihm können sowohl Programme, bestehend aus Rechenvorschriften, geladen, alsauch Ausdrücke geladen oder eingegeben und anschließend ausgewertet werden. Diese Programme und Ausdrücke müssen allerdings der in Abschnitt 2.2 beschriebenen Teilmenge von Standard ML angehören. Des Weiteren können noch einige Einstellungen bezüglich der Auswertung von Ausdrücken getätigt werden. Alle diese Aufgaben sind in KIEL auf mehrere Fenster verteilt, die im folgenden näher beschrieben werden.

2.4.1 Befehlsfenster

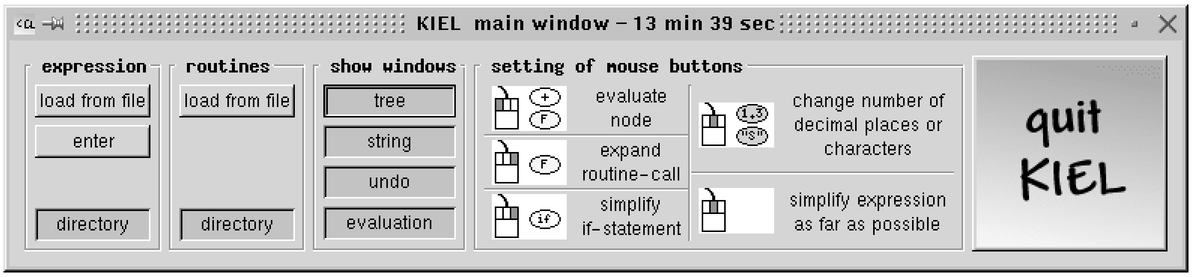


Abbildung 2.2: Das Befehlsfenster

Das Befehlsfenster (main window) ist das einzige im KIEL-System, welches nicht geschlossen werden kann, es sei denn, KIEL soll beendet werden. Von diesem zentralen Fenster aus können alle weiteren erreicht werden. Wie in Abbildung 2.2 zu erkennen ist, lässt es sich in die fünf Bereiche Rechenvorschriften, Ausdruck, Fenster anzeigen, Maustastenbelegung und Kiel beenden unterteilen.

Im **Bereich Rechenvorschriften** (routines) befinden sich der Übersichts-Toggel[3], der das Übersichtsfenster für Rechenvorschriften ein- bzw. ausblendet, und der Lade-Button. Dieser öffnet einen Dialog zum Laden von Rechenvorschriften aus einer Textdatei, deren Name mit ".prg" enden muss. Diese Dateinamenendung steht für das englische Wort *program*.

Der **Bereich Ausdruck** (expression) enthält einen Übersichts-Toggel und einen Lade-Button, die denselben Zweck erfüllen, wie die im Bereich Rechenvorschriften, nur für Ausdrücke. Der Name einer zu ladenden Textdatei muss in diesem Fall jedoch mit ".exp" enden. Diese Abkürzung steht für das englische Wort *expression*. Der zusätzliche

[3]Ein Toggel ist ein Schalter mit zwei Zuständen. Im ersten erscheint er wie ein Button und im zweiten wie ein gedrückter Button. Mittels Mausklick wird dieser Zustand gewechselt.

Eingabe-Button ermöglicht es, Ausdrücke auch direkt per Tastatur einzugeben.

Der **Bereich Fenster anzeigen** (show windows) besteht aus vier Toggels, die weitere Fenster ein- bzw. ausblenden. Dazu zählen das Baumfenster, das Zeichenreihenfenster, das Undo-Redo-Fenster und das Auswertungsfenster.

Im **Bereich Maustastenbelegung** (setting of mouse buttons) wird mittels Piktogrammen verdeutlicht, welche Aktion welche Maustaste bei einem Klick im Baumfenster auslöst.

Der letzte **Bereich Beenden-Button** (quit KIEL) enthält auch nur eben diesen Button, der alle Fenster schließt und KIEL beendet.

2.4.2 Übersichtsfenster

Abbildung 2.3: Das Ausdruckübersichtsfenster

Das **Ausdruckübersichtsfenster** (expression's directory window), welches in Abbildung 2.3 dargestellt ist, enthält eine Liste aller momentan im KIEL-System verwalteten Ausdrücke. Durch die Eingabe oder das Laden weiterer Ausdrücke im Befehlsfenster wird die Liste im Ausdruckübersichtsfenster erweitert. Wird ein Ausdruck aus der Liste per Maus selektiert, so öffnet sich das Baumfenster, in dem dieser Ausdruck nun als Baum dargestellt wird. Wurde dieser Ausdruck schon früher im Baumfenster bearbeitet, so wird dieser Bearbeitungszustand im Baumfenster angezeigt.

Um bei der Eingabe neuer Ausdrücke über die Tastatur nicht den Überblick über die verwendbaren Rechenvorschriften zu verlieren, werden diese im **RV-Übersichtsfenster**[4]

[4]RV steht für **R**echen**v**orschrift

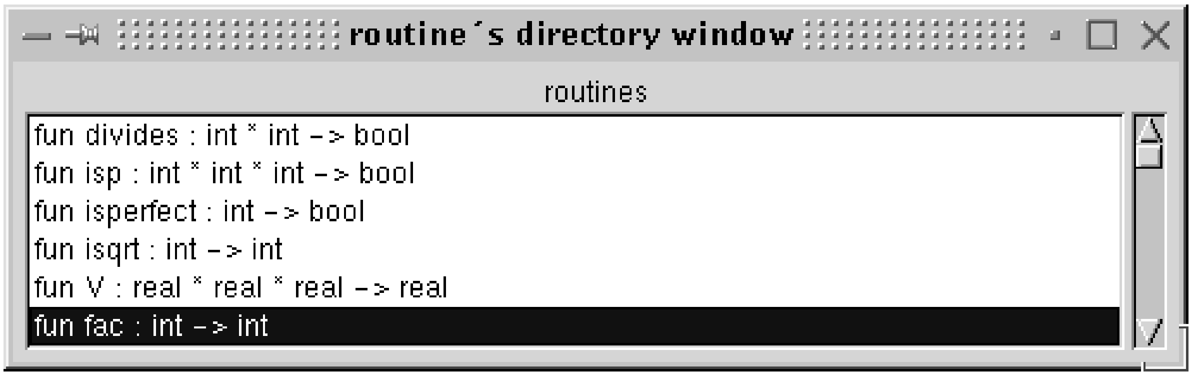


Abbildung 2.4: Das RV-Übersichtsfenster

(routine's directory window) angezeigt (siehe Abbildung 2.4). Beim Laden eines Programms mittels des Befehlsfensters werden sowohl die Liste im Ausdruckübersichtsfenster als auch im RV-Übersichtsfenster geleert und letztere dann mit den Namen und der Funktionalität aller Rechenvorschriften, die im Programm deklariert sind, aufgefüllt.

2.4.3 Auswertungs- und Undo-Redo-Fenster

Das **Auswertungsfenster** (evaluation window) besteht aus drei Bereichen, wie Abbildung 2.5 zeigt. Erstens kann die Auswertungsstrategie gewählt werden, mit der alle folgenden Evaluierungsaktionen im Baumfenster ausgeführt werden. Zur Wahl stehen die schon in Abschnitt 2.3 beschriebenen Strategien $leftmost - innermost$, $leftmost - outermost$ und $full$.

Zweitens kann die maximale Anzahl der Auswertungsschritte festgelegt werden, die bei einer Aktion im Baumfenster erfolgen darf.

Ein besonderer Hinweis gebührt an dieser Stelle dem Auswertung-anhalten-Button. Sollte eine Berechnung nicht terminieren oder aber dem Benutzer einfach zu lange dauern, so kann mittels dieses Buttons die aktuelle Auswertungsaktion, die im Baumfenster gestartet wurde, angehalten werden. Anschließend ist der Auswertungszustand im Baumfenster zu sehen, an dem die Evaluierung angehalten wurde.

Das **Undo-Redo-Fenster** (undo-redo window) ermöglicht es dem Anwender, jede einzelne Auswertungsaktion, die er im Baumfenster vorgenommen hat, zurückzunehmen. Solange zwischendurch keine neue Auswertungsaktion gestartet wurde, können zurückgenommene Auswertungsaktionen auch wiederhergestellt werden. Sollte doch eine solche Aktion ausgeführt worden sein, so ist eine sinnvolle Wiederherstellung nicht mehr möglich und alle Wiederherstellungsschritte werden verworfen. Abbildung 2.6 vermittelt einen visuellen Eindruck von diesem Fenster.

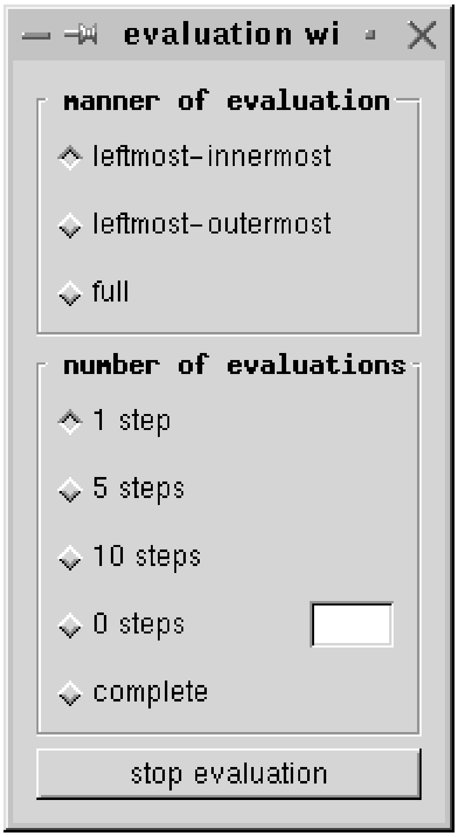


Abbildung 2.5: Das Auswertungsfenster

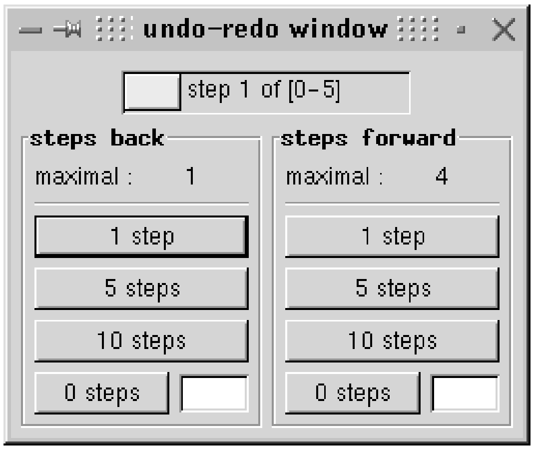


Abbildung 2.6: Das Undo-Redo-Fenster

Es ist zu beachten, dass Rücknahme- bzw. Wiederherstellungsschritte nicht mit den Auswertungsschritten gleichzusetzen sind. Wurden beispielsweise im Auswertungsfenster 5 Auswertungsschritte eingestellt und im Baumfenster eine Auswertungsaktion vorgenommen, so kann nicht ein einzelner Auswertungsschritt zurückgenommen werden, sondern nur die gesamte Aktion, d.h. alle maximal 5 Auswertungsschritte zusammen.

2.4.4 Baum- und Zeichenreihenfenster

Das **Baumfenster** (tree window) dient im Wesentlichen der Visualisierung eines Ausdrucks und der Interaktion mit dem Benutzer. Sobald der Anwender im Ausdrucksübersichtsfenster einen Ausdruck selektiert, öffnet sich das Baumfenster und stellt diesen als Baum so dar, wie es Abbildung 2.7 veranschaulicht. Sollte dieser Ausdruck allerdings schon früher bearbeitet worden sein, so wird dieser Bearbeitungszustand wiederhergestellt. Die Knoten des Baumes werden je nach Art auch graphisch unterschiedlich dargestellt. So werden Konstanten aller Sorten durch ein Oval mit hellblauem Hintergrund,

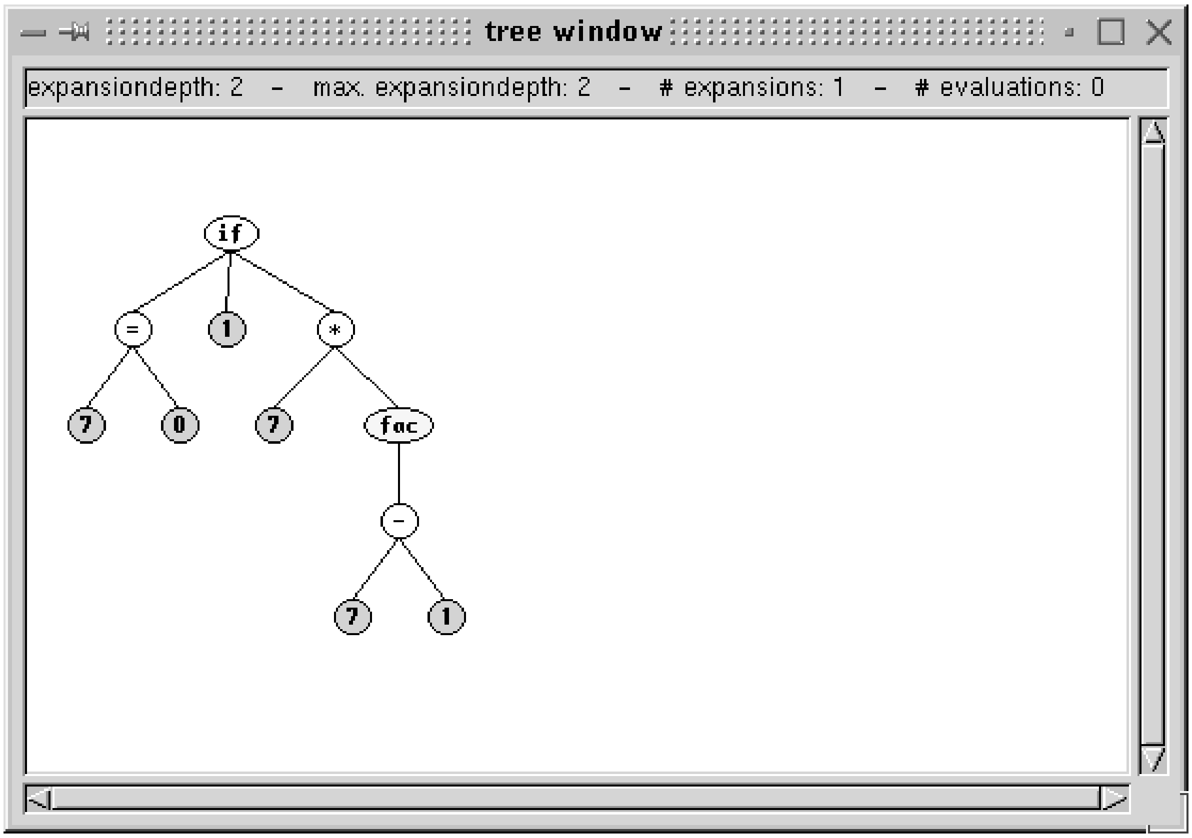


Abbildung 2.7: Das Baumfenster

die in 2.2.1 beschriebenen Basisoperatoren und -funktionen und die Verzweigung durch ein Oval mit weißem Hintergrund, Aufrufe von benutzerdefinierte Rechenvorschriften durch ein Oval mit hellgelbem Hintergrund und ein Fehlerknoten durch ein Rechteck mit rotem Hintergrund repräsentiert.

Durch verschiedene Aktionen, die durch einen einfachen Mausklick auf einen Knoten oder den Hintergrund gestartet werden, kann die Auswertung der Ausdrucks vorangetrieben werden. Welche der drei Maustasten beim Klick auf eine Art von Knoten welche Aktion auslöst, wird im Bereich Maustastenbelegung des Befehlsfensters durch Piktogramme verdeutlicht. Die verfügbaren Aktionen sind:

- *Auswertung des Knotens* (evaluate node) - Diese Aktion evaluiert den Ausdruck nach den gewählten Parametern im Auswertungsfenster so, als ob der angeklickte Knoten die Wurzel des Ausdrucks wäre.

- *Expandieren des RV-Aufrufes* (expand routine-call) - Falls der angeklickte Knoten tatsächlich einen Aufruf einer Rechenvorschrift repräsentiert, so wird dieser expandiert.

- *Vereinfachung der Verzweigung* (simplify if-statement) - Falls der angeklickte Knoten tatsächlich eine Verzweigung darstellt, wird die Bedingung vollständig berechnet und dann auch die Verzweigung ausgewertet, so dass nur noch die richtige Alternative übrig bleibt.

- *Ändern der Nachkommastelligkeit bzw. der Zeichenanzahl* (change number of decimal places or characters) - Bei Konstanten der Sorte *real* kann die Anzahl der angezeigten Nachkommastellen bestimmt werden und bei Konstanten der Sorte *string* die maximale Anzahl der angezeigten Zeichen.

- *Vereinfachung des Ausdrucks weitmöglichst* (simplify expression as far as possible) - Der Klick auf den Hintergrund des Baumfensters bewirkt diese Aktion, die den gesamten Ausdruck so weit wie möglich vereinfacht, ohne einen Aufruf einer Rechenvorschrift zu expandieren.

Der Deutlichkeit halber sei angemerkt, dass für jede vom Anwender ausgelöste Aktion im Baumfenster die Parameter im Auswertungsfenster verändert werden können. Die Möglichkeiten, Aktionen mittels des Undo-Redo-Fensters zurückzunehmen und wiederherzustellen, hängen hingegen nicht von diesen Parametern ab und bleiben auch dann erhalten, wenn zwischenzeitlich der selektierte Ausdruck im Ausdrucksübersichtsfenster, und damit im Baumfenster, gewechselt hat.

Weiterhin befindet sich am oberen Rand des Baumfensters eine Statuszeile. Sie zeigt Informationen über

- die aktuelle Expansionstiefe,

- die bisher maximale Expansionstiefe,

- die Anzahl der bisherigen Expansionen,

- die Anzahl der bisherigen Auswertungen von Basisoperatoren, -funktionen und der Verzweigung,

die sich auf die zuletzt vom Benutzer gestartete Auswertungsaktion beziehen. Indem diese Statuszeile auch während der Auswertung aktualisiert wird, vermittelt sie dem Anwender einen Eindruck über den Fortschritt der aktuellen Auswertungsaktion.

Beim **Zeichenreihenfenster** (string window) handelt es sich um ein passives Fenster, welches den aktuell im Baumfenster dargestellten Ausdruck als Zeichenreihe enthält. Es ist in Abbildung 2.8 zu sehen.

Abbildung 2.8: Das Zeichenreihenfenster

2.5 Beschränkungen und Motivation für Erweiterungen

2.5.1 Zusammengesetzte Datenstrukturen

Da die vom KIEL-System akzeptierte Sprache, wie in Abschnitt 2.2 beschrieben, nur die fünf Sorten und die fünf entsprechenden Listentypen bereitstellt, ist es dem Anwender nicht möglich, komplexere Datenstrukturen zusammen zu setzen. Es gibt beispielsweise keinen Weg, im bisherigen KIEL-System knotenmarkierte Binärbäume zu implementieren. Hier ist eine sinnvolle Erweiterung des KIEL-Systems auf Grundlage von Standard ML erforderlich, die nicht nur die bloße Deklaration solcher zusammengesetzter Datenstrukturen zulässt, sondern auch ein sinnvolles Zusammenspiel mit Rechenvorschriften ermöglicht.

2.5.2 Rechenvorschriften mit Mustererkennung

Unabhängig von der Frage, welche anderen Möglichkeiten es gibt, um auf die einzelnen Komponenten der zusammengesetzten Datenstrukturen zuzugreifen, gestalte sich dieser Zugriff sehr komfortabel, wenn im KIEL-System die Deklaration von Rechenvorschriften mit Mustererkennung eingeführt wird. Im bisherigen KIEL-System stellt das Fehlen von Rechenvorschriften mit Mustererkennung nur eine sehr geringe Einschränkung dar. Mit der Erweiterung um zusammengesetzte Datenstrukturen werden diese jedoch relevanter, da die Mustererkennung den Umgang mit den neuen Datenstrukturen stark erleichtert. Somit werden mit der Erweiterung um zusammengesetzte Datenstrukturen auch Rechenvorschriften mit Mustererkennung eingeführt.

2.5.3 Geschachtelte Listen

Eine weitere Einschränkung besteht bei den Listen. Das KIEL-System akzeptiert nur Listen über den fünf Sorten. Aber mit Einführung von zusammengesetzten Datenstrukturen sollten auch entsprechende Listen ermöglicht werden. Weiterhin würden geschachtelte Listen die Ausdruckskraft der von KIEL akzeptierten Sprache erhöhen. Dann könnte beispielsweise mittels einer Liste von Listen reeller Zahlen (*real list list*) mit Matrizen gerechnet werden.

Kapitel 3

Erweiterung von KIEL

Im Verlauf dieses Kapitels wird gezeigt, wie die aus Abschnitt 2.5 bekannten Beschränkungen des bisherigen KIEL-Systems beseitigt werden.

Dazu wird zuerst beschrieben, welche Möglichkeiten es gibt, KIEL um zusammengesetzte Datenstrukturen zu erweitern. Es wird sich zeigen, dass ein um zusammengesetzte Datenstrukturen erweitertes KIEL-System nicht ohne Rechenvorschriften mit Mustererkennung auskommt. Schließlich werden noch geschachtelte Listen in KIEL integriert.

3.1 Zusammengesetzte Datenstrukturen

Für die Erweiterung des KIEL-Systems um zusammengesetzte Datenstrukturen bietet Standard ML zwei Möglichkeiten. Zum einen wird das durch Typkonstruktoren erreicht und zum anderen durch die *datatype*-Deklaration. Beide Ansätze werden im folgenden kurz vorgestellt und es wird ihre Eignung für die Zwecke des KIEL-Systems bewertet.

3.1.1 Typkonstruktoren

In Standard ML, wie in anderen Programmiersprachen auch, bilden die implementierten Sorten die Basis für alle weiteren Typen. So lassen sich mittels der Typkonstruktoren $*, \rightarrow$ und *list* komplexere Typen zusammensetzen. D.h., jede Sorte ist ein Typ und sind $m, n, m_1, \ldots, m_k$ $(k \geq 2)$ Typen, so sind auch $m_1 * \ldots * m_k$, $m \rightarrow n$ und m *list* Typen, genannt Produkt-, Funktions- und Listentyp.

Während die Produkttypen genau die in 2.5.1 beschriebenen Beschränkungen aufheben, so übertreffen die Funktionstypen die Anforderungen und würden den Aufwand der Implementierung erheblich erhöhen. Die mittels des *list*-Typkonstruktors erzeugbaren geschachtelten Listen werden ab Abschnitt 3.3 gesondert behandelt.

3.1.2 *datatype*-Deklaration

Alternativ können in Standard ML zusammengesetzte Datenstrukturen mittels der *datatype*-Deklaration deklariert werden. Sie werden im folgenden kurz Datentypen genannt.

Sei D Menge aller Sorten und bereits deklarierter Datentypen, sei f eine Funktion, die zur gegebenen Menge B von Typen die Menge aller Typen liefert, die aus der Basismenge B mittels der Typkonstruktoren aus 3.1.1 ($*, \rightarrow, list$) erzeugbar sind[1] und sei $k \geq 1$, dann hat eine *datatype*-Deklaration die Form

$$\begin{aligned} datatype\ m = {} & con_1\ of\ n_1\ | \\ & \vdots \\ & con_k\ of\ n_k\ ; \end{aligned}$$

wobei $con_1, \ \ldots \ , con_k$ und m jeweils neue Bezeichner sind und alle $n_i \in f(D \cup \{m\})$ $(0 \leq i \leq k)$. Nach dieser Deklaration kann der neue Typ mit der Bezeichnung m im folgenden Programmtext verwendet werden. Die sogenannten Konstruktoren con_i haben die Funktionalität $n_i \rightarrow m$ $(1 \leq i \leq k)$ und bilden die einzige Möglichkeit, Daten vom Typ m zu erzeugen. Des Weiteren kann die Schreibweise $con_i\ of\ unit$ abgekürzt werden durch con_i $(1 \leq i \leq k)$, wobei con_i dann als Konstruktor ohne Parameter bezeichnet wird.

Eine Datenstruktur kann also mittels verschiedener Konstruktoren auf verschiedene Arten beschrieben werden. Beispielsweise lassen sich Geraden im $\mathbb{R} \times \mathbb{R}$ in Normalform oder in Vektorform angeben. Der Normalform liegt die Formel

$$g = \{(x, y) \in \mathbb{R} \times \mathbb{R} : ax + by + c = 0\}$$

zugrunde und der Vektorform die Formel

$$g = \{v + \lambda w : \lambda \in \mathbb{R}\}$$

mit dem Ortsvektor $v = (v_1, v_2) \in \mathbb{R} \times \mathbb{R}$ und dem Richtungsvektor $w = (w_1, w_2) \in \mathbb{R} \times \mathbb{R}$. In Standard ML können Geraden folgendermaßen deklariert werden:

```
datatype TVec  = mkvec of real * real;
datatype TLine = mknormform of real * real * real |
                 mkvecform  of TVec * TVec;
```

Auf Grundlage dieser Deklaration repräsentieren die beiden Ausdrücke

[1] Dazu zählen insbesondere Produkttypen.

```
mknormform(1.0, -1.0, 0.0)
mkvecform(mkvec(0.0, 0.0), mkvec(1.0, 1.0));
```

die Gerade $g = \{(x, y) \in \mathbb{R} \times \mathbb{R} : x = y\}$.

Rekursive Datenstrukturen bilden einen weiteren Aspekt. Die schon in 2.5.1 angesprochenen knotenmarkierten Binärbäume können mittels der *datatype*-Deklaration durch

```
datatype TBinTree = mkleaf of int |
                    mkbintree of TBinTree * int * TBinTree;
```

deklariert werden. Einen solchen Binärbaum mit einer Wurzel, einem normalen Knoten und drei Blättern bildet folgender Ausdruck:

```
mkbintree(mkbintree(mkleaf(4), 2, mkleaf(5)), 1, mkleaf(3))
```

Schließlich bleibt noch die Möglichkeit, wie bei der Deklaration von Rechenvorschriften auch, Systeme von Datentypen zu deklarieren. Sei D Menge aller Sorten und bereits deklarierter Datentypen, sei f eine Funktion, die zur gegebenen Menge B von Typen die Menge aller Typen liefert, die aus der Basismenge B mittels der Typkonstruktoren aus 3.1.1 $(*, \rightarrow, list)$ erzeugbar sind und sei $i \geq 2$, dann hat ein solches System die Form

$$
\begin{aligned}
datatype\ m_1 &= con_{11}\ of\ n_{11} \mid \\
&\vdots \\
&\quad con_{1k_1}\ of\ n_{1k_1} \mid \\
and\ m_2 &= con_{21}\ of\ n_{21} \mid \\
&\vdots \\
&\quad con_{2k_2}\ of\ n_{2k_2} \mid \\
\vdots \\
and\ m_i &= con_{i1}\ of\ n_{i1} \mid \\
&\vdots \\
&\quad con_{ik_i}\ of\ n_{ik_i}\ ;
\end{aligned}
$$

wobei $con_{11}, \ldots, con_{1k_1}, \ \ldots\ , con_{i1}, \ldots, con_{ik_i}$ und $m_1, \ldots, m_i$ jeweils neue Bezeichner sind und es gilt:

$$(\forall 1 \leq j \leq i : (k_j \geq 1) \wedge (\forall 1 \leq l \leq k_j : n_{jl} \in f(D \cup \{m_1,\ \ldots\ , m_i\})))$$

3.1.3 Eignung für KIEL

Ein Aspekt, der gegen eine Erweiterung um die Typkonstruktoren aus Standard ML spricht, sind die Funktionstypen. Sie sind nicht bei den Anforderungen für das KIEL-System aufgelistet und erfordern einen hohen Implementierungsaufwand. Wichtiger ist jedoch die Tatsache, dass solche Typen weder rekursive Datenstrukturen erlauben, noch die Möglichkeit zulassen, dieselbe Datenstruktur auf verschiedene Arten darzustellen.

Aus diesen Gründen sind Datentypen für die Erweiterung des KIEL-Systems geeigneter.

Da das KIEL-System jedoch auch nach der Erweiterung nicht über Produkttypen verfügt, muss die Deklaration eines Datentyps noch an KIEL angepasst werden.

Sei D Menge aller Sorten und aller bereits deklarierten Datentypen und sei $k \geq 1$, dann hat im erweiterten KIEL-System eine *datatype*-Deklaration die Form:

$$\begin{aligned} \textit{datatype } m = {} & \textit{con}_1 \ \textit{of} \ n_{11} * \ \ldots \ * n_{1l_1} \ | \\ & \vdots \\ & \textit{con}_k \ \textit{of} \ n_{k1} * \ \ldots \ * n_{kl_k} \ ; \end{aligned}$$

wobei $\textit{con}_1, \ \ldots \ , \textit{con}_k$ und m jeweils neue Bezeichner sind und es gilt:

$$\forall 1 \leq i \leq k : (l_i \geq 1 \wedge (\forall 1 \leq j \leq l_i : n_{ij} \in D \cup \{m\}))$$

Der so deklarierte neue Typ m kann im folgenden Programmtext verwendet werden. Der Aufruf des Konstruktors $\textit{con}_i$ mit der Funktionalität $n_{11} \times \ \ldots \ \times n_{1l_i} \rightarrow m$ hat die From $\textit{con}_i(e_1, \ \ldots \ , e_{l_i})$, wobei $e_1, \ \ldots \ , e_{l_i}$ Ausdrücke von den Typen $n_{11}, \ \ldots \ , n_{1l_i}$ sind $(1 \leq i \leq k)$.

Die angepasste Deklaration ist jedoch nicht als verschleierte Einführung von Produkttypen zu interpretieren, sondern als eine rein syntaktische Modifikation. Auch bei dieser angepassten Deklaration kann die Schreibweise $\textit{con}_i \ \textit{of unit}$ abgekürzt werden durch $\textit{con}_i$ $(1 \leq i \leq k)$, wobei $\textit{con}_i$ wiederum als Konstruktor ohne Parameter bezeichnet wird. Dabei ist zu beachten, dass der Aufruf eines solchen Konstruktors von der verwendeten Schreibweise bei der Deklaration abhängt. Wird $\textit{con}_i \ \textit{of unit}$ benutzt, so lautet der Aufruf $\textit{con}_i(())$, bei $\textit{con}_i$ wird auch der Aufruf als $\textit{con}_i$ geschrieben $(1 \leq i \leq k)$.

Auch im erweiterten KIEL kann ein System von Datentypen deklariert werden. Die Deklaration eines solchen Systems erfolgt genau wie in Standard ML und wie in 3.1.2 beschrieben nur mit der Anpassung an KIEL bei den Typangaben, wie gerade beschrieben.

3.2 Rechenvorschriften mit Mustererkennung

Die Erweiterung des KIEL-Systems ist jedoch noch nicht ausreichend für den Umgang mit zusammengesetzten Datenstrukturen. Mit Hilfe der in der *datatype*-Deklaration deklarierten Konstruktoren lassen sie sich zwar erzeugen, aber es besteht noch keine Möglichkeit wieder auf die einzelnen Komponenten der Struktur zuzugreifen. Ohne diesen Zugriff sind die zusammengesetzten Datenstrukturen allerdings recht nutzlos - vergleichbar mit einem Dateisystem, welches ausschließlich einen Schreibzugriff erlaubt.

Ein Sprachmittel für den direkten Zugriff auf die Komponenten einer zusammengesetzten Datenstruktur stellt auch Standard ML nicht zur Verfügung. Die einzige Lösung für dieses Problem stellen die schon in 2.5.2 erwähnten Rechenvorschriften mit Mustererkennung dar.

3.2.1 Muster

Syntaktisch ähneln Muster den Ausdrücken. Sie sind jedoch einerseits eingeschränkt, da sie nur aus Konstanten, Bezeichnern, und Konstruktoren[2] aufgebaut sind, andererseits sind sie erweitert um den Unterstrich "_".

Definition: Muster sind durch folgende Regeln induktiv definiert:

1. Der Unterstrich _ ist ein Muster.
2. Jede Konstante ist ein Muster.
3. Jeder Bezeichner, der keinen Konstruktor repräsentiert, ist ein Muster genannt Variable.
4. Jeder Konstruktor ohne Parameter ist ein Muster.
5. Die Darstellung der leeren Liste [ ] ist ein Muster.
6. Ist p ein Muster, so auch (p).
7. Ist con ein Konstruktor mit Parametern und sind $p_1, \ldots, p_k$ Muster, so ist auch $con(p_1, \ldots, p_k)$ ein Muster ($k \geq 1$).
8. Sind $p_1, \ldots, p_k$ Muster, so ist auch $[p_1, \ldots, p_k]$ eines ($k \geq 1$).
9. Sind p_1 und p_2 Muster, so auch $p_1 :: p_2$.

Grundlegend für die Auswertung von Rechenvorschriften mit Mustererkennung ist der Vergleich von Mustern und Ausdrücken.

Definition: Seien K Menge aller Konstanten, B Menge aller erlaubten Bezeichner, $C \subset B$ Menge aller Bezeichner von bereits deklarierten Konstruktoren und $V = B \setminus C$ Menge aller Variablen. Ein Ausdruck e entspricht einem Muster p , in Zeichen $e \sqsubseteq p$, wenn eine der folgenden Bedingungen erfüllt ist:

[2] dazu zählen auch die Standardkonstruktoren für Listen :: und [...]

1. $p = \_$
2. $p \in K \wedge e = p$
3. $p \in V$
4. $p \in C \wedge e = p$
5. $(p = nil \vee p = [\,]) \;\wedge\; (e = nil \vee e = [\,])$
6. $\big(p = (p') \wedge e = (e') \wedge e' \sqsubseteq p'\big) \;\vee\; \big(p = (p') \wedge e \sqsubseteq p'\big) \;\vee\; \big(e = (e') \wedge e' \sqsubseteq p\big)$
7. $c \in C \wedge k \geq 1 \wedge p = c(p_1,\ \dots\ ,p_k) \wedge e = c(e_1,\ \dots\ ,e_k) \wedge \forall\, 1 \leq i \leq k : e_i \sqsubseteq p_i$
8. $k \geq 1 \wedge p = [p_1,\ \dots\ ,p_k] \wedge e = [e_1,\ \dots\ ,e_k] \wedge \forall\, 1 \leq i \leq k : e_i \sqsubseteq p_i$
9. $p = p_1 :: p_2 \wedge \big(\ (e = e_1 :: e_2 \wedge e_1 \sqsubseteq p_1 \wedge e_2 \sqsubseteq p_2) \vee$
   $(k \geq 1 \wedge e = [e_1, \dots , e_k] \wedge e_1 \sqsubseteq p_1 \wedge [e_2, \dots , e_k] \sqsubseteq p_2)\ \big)$

Definition: Eine Liste von Ausdrücken $e_1,\ \dots\ ,e_k$ entspricht einer Liste von Mustern $p_1,\ \dots\ ,p_l$ falls gilt:

$$k = l \wedge \forall\, 1 \leq i \leq k : e_i \sqsubseteq p_i$$

3.2.2 Deklaration

Bei einer Deklaration einer Rechenvorschrift mit Mustererkennung sind einige Änderungen gegenüber der alt bekannten Deklaration zu beachten. So besteht die Liste der formalen Parameter nicht mehr aus Variablen mit Typisierung, sondern aus Mustern mit Typisierung. Des Weiteren kann eine Rechenvorschrift aus mehreren Fällen bestehen. Diese setzen sich aus einer Parameterliste und einem dazugehören Rumpf zusammen.

Formal hat eine Deklaration einer solchen Rechenvorschrift mit Mustererkennung folgendes schematische Aussehen ($k \geq 1 \wedge l \geq 0$):

$$\begin{array}{l} fun\ f(p_{11} : m_1,\ \dots\ ,p_{1l} : m_l) : m = b_1 \\ \quad \mid\ f(p_{21} : m_1,\ \dots\ ,p_{2l} : m_l) : m = b_2 \\ \qquad \vdots \\ \quad \mid\ f(p_{k1} : m_1,\ \dots\ ,p_{kl} : m_l) : m = b_k\ ; \end{array}$$

Eine so deklarierte Funktion f hat den Typ $m_1 \times \dots \times m_l \rightarrow m$. Jeder Rumpf b_i ist ein Ausdruck, der zusätzlich aus Aufrufen von bereits deklarierten Rechenvorschriften, aus Aufrufen von f selbst und aus allen Variablen, die in den Mustern $p_{i1},\ \dots\ ,p_{il}$

vorkommen, aufgebaut werden kann ($1 \leq i \leq k$). Analog zur alten Deklaration können auch Systeme von Rechenvorschrift mit Mustererkennung deklariert werden (siehe 2.2.3).

Welcher Fall bei einer Auswertung eines Aufrufs $f(e_1, \ldots, e_l)$ zur Anwendung kommt, hängt von den Argumenten $e_1, \ldots, e_l$ ab.

Nach Standard ML wird die Argumentliste der Reihe nach mit den Parameterlisten der verschiedenen Fälle verglichen. Sobald die Argumentliste $e_1, \ldots, e_l$ einer Parameterliste $p_{i1}, \ldots, p_{il}$ entspricht, steht der Rumpf b_i als derjenige fest, der bei der Auswertung des Aufrufs verwendet werden muss ($1 \leq i \leq k$). Alle weiteren Parameterlisten werden nicht mehr überprüft. Sollte die Argumentliste keiner der Parameterlisten entsprechen so handelt es sich um einen Laufzeitfehler.

Beispiel 1: Fakultät

```
fun fac1(1: int): int = 1
|   fac1(n: int): int = n * fac1(n - 1);

fun fac2(n: int): int = n * fac2(n - 1)
|   fac2(1: int): int = 1;
```

Während die Rechenvorschrift $fac1$ für Zahlen größer als Null korrekt die Fakultät berechnet, wird $fac2$ niemals terminieren, obwohl nur die Reihenfolge der Fälle vertauscht wurde.

Beispiel 2: Geraden

```
datatype TVec  = mkvec of real * real;
datatype TLine = mknormform of real * real * real |
                 mkvecform  of TVec * TVec;

fun parallel_x(mknormform(a, _, _)                       : TLine): bool =
  (a  = 0.0)
|   parallel_x(mkvecform (mkvec(vx, vy), mkvec(wx, wy)): TLine): bool =
  (wy = 0.0);
```

Die Rechenvorschrift $parallel\_x$ stellt für eine Gerade, die entweder in Normalform oder in Vektorform angegeben ist, fest, ob sie parallel zur X-Achse verläuft.

Beispiel 3: Knotenmarkierte Bäume

```
datatype TBinTree = mkleaf of int |
                    mkbintree of TBinTree * int * TBinTree;

fun bintreeheight(mkleaf(_)         : TBinTree): int =
  0
|   bintreeheight(mkbintree(l, _, r): TBinTree): int =
  1 + max(bintreeheight(l), bintreeheight(r));
```

Dieses Beispiel veranschaulicht den Umgang mit rekursiven Datenstrukturen. Genau wie die *datatype*-Deklaration besteht die Funktion *bintreeheight* aus zwei Fällen. Die Parameterliste des ersten Falles behandelt den nicht rekursiven Konstruktor $mkleaf$ und somit enthält auch der entsprechende Rumpf keinen rekursiven Aufruf von $bintreeheight$. Dementsprechend enthält der Rumpf des zweiten Falles auch zwei rekursive Aufrufe, da mit der zugehörigen Parameterliste auch der zweifach rekursive Konstruktor *mkbintree* abgehandelt wird.

Anmerkung: Die ursprüngliche Deklaration von Rechenvorschriften fügt sich problemlos in die neue ein. Eine Rechenvorschrift im ursprünglichen Sinne besteht nur aus einem Fall ($k = 1$) und alle Muster der Parameterliste $p_{11}, \ldots, p_{1l}$ bestehen ausschließlich aus Variablen. Daher kommt es in solch einem Fall auch nie zu einem Laufzeitfehler aufgrund der Mustererkennung.

3.3 Geschachtelte Listen

Soll Standard ML als Grundlage dienen, so existieren für eine Erweiterung des KIEL-Systems um geschachtelte Listen keine Alternativen. Geschachtelte Listen ermöglichen es, dass die Elemente einer Liste wiederum Listen sind. Diese Erweiterung beinhaltet jedoch keine polymorphen Listen. Alle Elemente einer Liste müssen weiterhin vom selben Typ sein.

3.3.1 Typangabe

Sei m eine Sorte oder ein bereits deklarierter Datentyp und sei $k \geq 0$, dann hat die Typangabe einer k-fach geschachtelten Liste die Form:

$$m \ \underbrace{list \ \ldots \ list}_{(k+1)\ mal}$$

Für die Darstellung einer $m \times n$-Matrix kann beispielsweise eine Liste vom Typ *real list list* verwendet werden, deren m Elemente Listen vom Typ *real list* sind und jeweils aus n Elemente vom Typ *real* bestehen.

3.3.2 Notation

Bei der Notation geschachtelter Listen ändert sich prinzipiell nichts. Eingeschlossen von eckigen Klammern ("[" und "]") werden die Elemente der Liste, durch Kommata (",") voneinander getrennt, notiert. Bei den Elementen darf es sich wiederum um Listen handeln. Die 3×3-Matrix

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$$

kann beispielsweise durch folgende Liste repräsentiert werden:

```
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Anmerkung: An dieser Stelle sei darauf hingewiesen, dass alle in 2.2.1 vorgestellten Basisoperatoren und -funktionen auch auf geschachtelten Listen weiterhin unverändert definiert sind.

3.3.3 Anwendungsbeispiel

Um den Umgang mit geschachtelten Listen zu verdeutlichen, wird hier noch einmal das Beispiel der Matrizen aufgegriffen. Weiterhin zeigt es, wie Mustererkennung auch mit Listen funktioniert. Exemplarisch wird ein Algorithmus zum Transponieren einer Matrix vorgestellt.

Als Datenstruktur zur Darstellung einer Matrix A dient eine *real list list*, wie schon in 3.3.1 eingeführt. Die Elemente dieser Liste repräsentieren die Zeilen von A. Die Grundidee des Transpositionsalgorithmuses besteht darin, dass die jeweils ersten Elemente der Zeilen von A die erste Zeile von A^T bilden. Die jeweils zweiten Elemente bilden die zweite Zeile usw..

Daraus ergeben sich zwei Hilfsrechenvorschriften $headsOfRows$ und $tailsOfRows$. Die Erste liefert zu einer Matrix (*real list list*) die Liste aller ersten Elemente der Zeilen (*real list*). Die Zweite liefert zu einer $m \times n$-Matrix (*real list list*) die $m \times (n-1)$-Matrix (*real list list*), bei der das jeweils erste Element jeder Zeile (*real list*) entfernt wurde.

Der gesamte Transpositionsalgorithmus lautet dann wie folgt:

```
fun headsOfRows(nil                : real list list): real list =
  nil
|   headsOfRows((head :: _) :: rows: real list list): real list =
  head :: headsOfRows(rows);

fun tailsOfRows(nil                : real list list): real list list =
  nil
|   tailsOfRows((_ :: tail) :: rows: real list list): real list list =
  tail :: tailsOfRows(rows);

fun transpose(nil            : real list list): real list list =
  nil
|   transpose(nil :: nilrows  : real list list): real list list =
  nil
|   transpose(A              : real list list): real list list =
  headsOfRows(A) :: transpose(tailsOfRows(A));
```

Kapitel 4

Beispiele / Auswertung mit KIEL

Die in den vorherigen Kapiteln beschriebenen Aspekte der Bedienung und der Erweiterung des KIEL-Systems werden im folgenden demonstriert. Dazu wird die Auswertung ausgesuchter Beispielprogramme Schritt für Schritt gezeigt.

Zum Einstieg greift das Beispiel 1 nochmal auf die schon in Abschnitt 3.1 vorgestellte Rechenvorschrift *bintreeheight* zur Berechnung der Höhe eines Baumes zurück. Das zweite Beispiel zeigt anhand einer Rechenvorschrift zur iterativen Berechnung der Fibonacci-Zahlen, wie eine Auswertung nach der $leftmost-outermost$-Strategie durch die Verwendung von Mustererkennung fast wie nach $leftmost-innermost$ erfolgt. Wie die naive Verwendung von Mustererkennung unter Verwendung der$leftmost-outermost$ Strategie zu einer Endlosrekursion führen kann, wird anhand einer speziellen Version der *morris*-Funktion in Beispiel 3 veranschaulicht. Mit unendlichen Strukturen wird dann in Beispiel 4 gerechnet. Ein Anwendungsbeispiel für geschachtelte Listen bietet schließlich Beispiel 5. Dabei handelt es sich um den Transpositionsalgorithmus aus 3.3.3.

Um mit dem KIEL-System vertraut zu werden, bietet es sich an, diese Beispiele direkt am Rechner nachzuvollziehen. Aber auch ohne Computer, können die Auswertungen anhand der beigefügten Abbildungen nachvollzogen werden.

4.1 Beispiel 1

4.1.1 Vorbereitungen

Um den Einstieg zu erleichtern, wird an dieser Stelle nochmals die Rechenvorschrift *bintreeheight* aus 3.2.2 aufgegriffen. Damit diese Rechenvorschrift in KIEL verwendet werden kann, muss der folgende Programmtext als ASCII-Textdatei mit dem exemplarischen Namen *“beispiel1.prg”* vorliegen:

```
datatype TBinTree = mkleaf of int |
                    mkbintree of TBinTree * int * TBinTree;

fun bintreeheight(mkleaf(_)          : TBinTree): int =
  0
|   bintreeheight(mkbintree(l, _, r): TBinTree): int =
  1 + max(bintreeheight(l), bintreeheight(r));
```

Wurde KIEL bereits gestartet, so kann die Datei *"beispiel1.prg"* mittels des Lade-Buttons im Bereich Rechenvorschriften des Befehlsfensters eingelesen werden. Anschließend wird der Ausdruck

```
bintreeheight(mkbintree(mkbintree(mkleaf(4), 2, mkleaf(5)), 1, mkleaf(3)))
```

mittels des Eingabe-Buttons im Bereich Ausdruck des Befehlsfensters eingegeben, damit er später bearbeitet werden kann. Alternativ könnte der Ausdruck auch aus einer Datei mit dem exemplarischen Namen *"beispiel1.exp"* mittels des Lade-Buttons im Bereich Ausdruck des Befehlsfensters geladen werden. Der Ausdruck ist nun im automatisch erschienenen Ausdruckübersichtsfenster eingetragen.

Für die Auswertung des Ausdrucks müssen die Auswertungsstrategie auf $leftmost-innermost$ und die Anzahl der Auswertungen auf eins eingestellt werden. Dies geschieht im Auswertungsfenster, welches mittels des Auswertungsfenster-Toggels im Bereich Fenster anzeigen des Befehlsfensters eingeblendet wird.

Durch einen Linksklick mit der Maus auf den Ausdruck im Ausdruckübersichtsfenster öffnet sich das Baumfenster, welches nun diesen Ausdruck, wie in Abbildung 4.1 gezeigt, als Baum darstellt. Im erweiterten KIEL-System werden alle Konstruktoren durch ein Oval mit rotem Hintergrund dargestellt. Dazu zählen nun auch die Listenkonstruktoren :: und [...]. Alle anderen Knoten werden weiterhin wie in 2.4.4 beschrieben angezeigt.

4.1.2 Auswertung

Die gesamte Auswertung wird nach der $leftmost-innermost$ Strategie und Schritt für Schritt erfolgen.

Erster Auswertungsschritt
Da der in Abbildung 4.1 angezeigte Ausdruck weder Basisoperatoren noch Basisfunktionen enthält, kann der erste Auswertungsschritt nur in der Expansion des Aufrufs der Rechenvorschrift *bintreeheight* bestehen. Da es sich um eine Rechenvorschrift mit

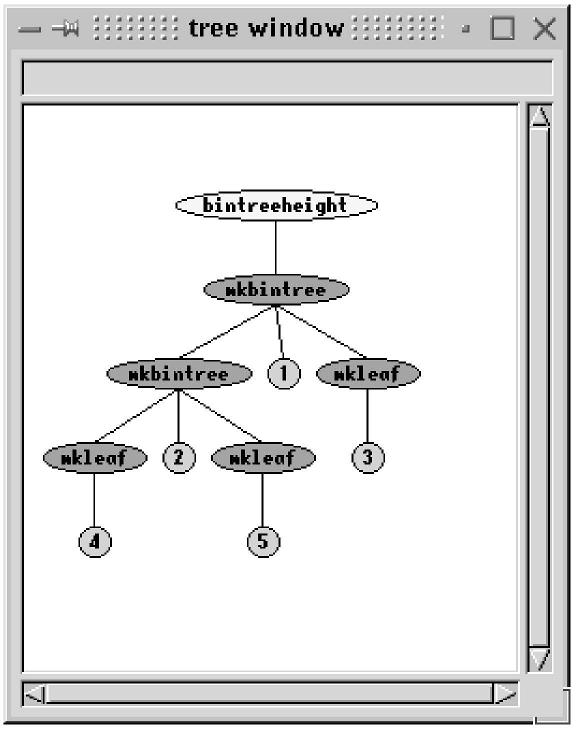


Abbildung 4.1: Der Ausdruck als Baum dargestellt

Mustererkennung handelt, muss zuerst entschieden werden, welcher der zwei Fälle, aus denen *bintreeheight* besteht, zur Auswertung herangezogen werden muss.

Das Argument der Rechenvorschrift besteht aus dem Konstruktor $mkbintree(\ldots)$ mit weiteren Argumenten und entspricht daher nicht der Parameterliste des ersten Falles. Aber es entspricht dem Muster $mkbintree(l, \_, r)$ des zweiten Falles, denn es gilt:

$$(mkbintree(mkleaf(4), 2, mkleaf(5)) \sqsubseteq l) \wedge (mkleaf(3) \sqsubseteq r)$$
$$\rightarrow$$
$$bintreeheight(mkbintree(mkbintree(mkleaf(4), 2, mkleaf(5)), 1, mkleaf(3))) \sqsubseteq mkbintree(l, \_, r)$$

Da der Aufruf der Rechenvorschrift auch die Wurzel des gesamten Ausdrucks ist, muss dieser durch den Rumpf $1 + max(bintreeheight(l), bintreeheight(r))$ des zweiten Falles ersetzt werden, wobei allerdings die beiden Variablen l und r noch ersetzt werden müssen.

Dazu werden sogenannte Bindungen aufgebaut.

Ein Ausdruck wird an eine Variable, die in der Parameterliste einer Rechenvorschrift auftritt, gebunden. Durch ihn werden alle Vorkommen dieser Variable im Rumpf der Rechenvorschrift ersetzt. Bei Rechenvorschriften ohne Mustererkennung ist die Findung der richtigen Bindung einfach, da ein Parameter genau aus einer Variable besteht und damit das gesamte Argument an die Variable gebunden werden muss. Bei Rechenvorschriften mit Mustererkennung ist ein Parameter aber nicht mehr nur eine Variable, sondern ein Muster, welches komplexer aufgebaut sein kann und insbesondere mehrere Variablen enthalten darf. Dies ist im obigen Beispiel der Fall. Das Muster $mkbintree(l, \_, r)$ enthält die beiden Variablen l und r, die auch im Rumpf verwendet werden. Sie müssen an die Teilausdrücke des Arguments gebunden werden, die ihnen bei der Mustererkennung entsprachen. In diesem Beispiel wird also der Teilausdruck $mkbintree(mkleaf(4), 2, mkleaf(5))$ an l und der Teilausdruck $mkleaf(3)$ an r gebunden. Damit ergibt sich nach einem Auswertungsschritt der Ausdruck, wie in Abbildung 4.2 dargestellt. Ein Linksklick mit der Maus auf die Wurzel des ursprünglichen Ausdrucks bestätigt dies.

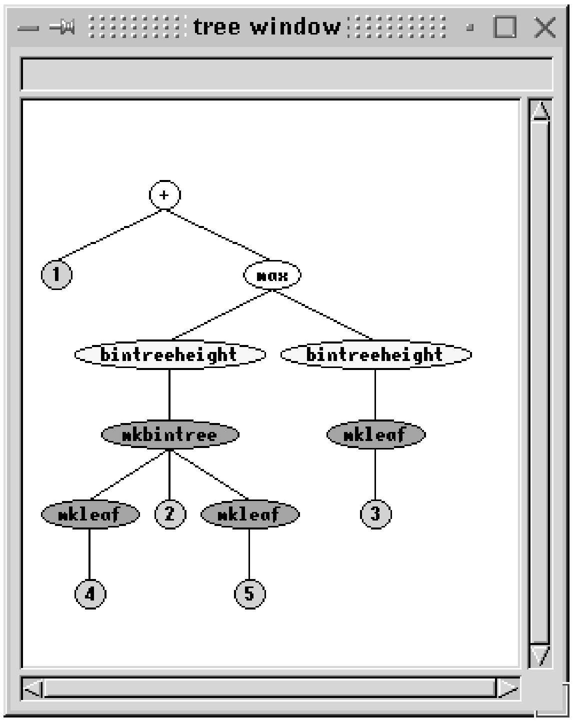


Abbildung 4.2: Der Ausdruck nach einem Auswertungsschritt

Zweiter Auswertungsschritt

Der in Abbildung 4.2 dargestellte Ausdruck enthält weder Basisoperatoren noch Basisfunktionen, die ausgewertet werden könnten. Folglich wird der linke Aufruf von *bintreeheight* expandiert. Wie im vorherigen Schritt wird wiederum der zweite Fall für die

Auswertung verwendet. Diesmal wird jedoch nur der Teilausdruck ab dem zu expandierenden Aufruf durch den entsprechenden Rumpf ersetzt. Es wird hier $mkleaf(4)$ an l und $mkleaf(5)$ an r gebunden und dementsprechend werden alle Vorkommen der Variablen l und r im Rumpf ersetzt. Nach einem weiteren Linksklick mit der Maus auf die Wurzel wird im Baumfenster der Ausdruck aus Abbildung 4.3 angezeigt.

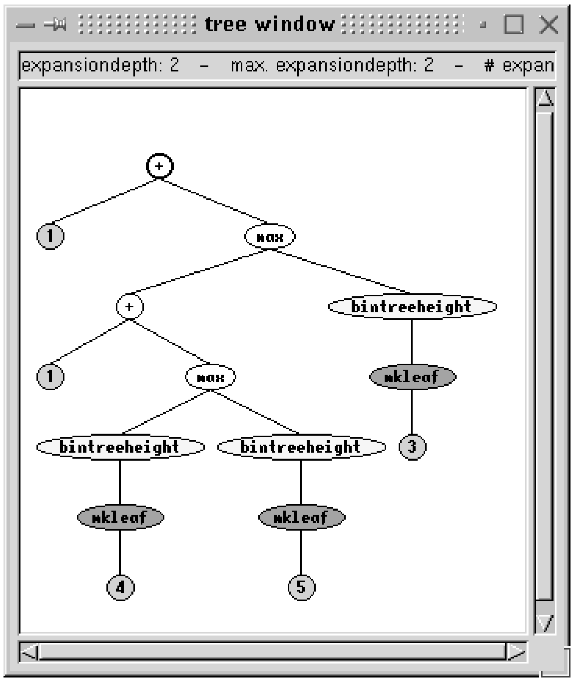


Abbildung 4.3: Der Ausdruck nach einem weiteren Auswertungsschritt

Dritter und vierter Auswertungsschritt

Aus obigem Grund wird wiederum der ganz linke Aufruf von *bintreeheight* expandiert. Das Argument $mkleaf(4)$ entspricht jedoch dem Muster des ersten Falles. Weil der Rumpf keine Variablen enthält, wird der zu expandierende Aufruf komplett durch den unveränderten Rumpf ersetzt. Das Ergebnis eines weiteren Linksklicks ist in Abbildung 4.4 zu sehen.

Der nächste Schritt erfolgt analog. Abbildung 4.5 zeigt den entsprechenden Ausdruck.

Weitere Auswertungsschritte

Nun werden zum ersten Mal eine Basisfunktion und ein Basisoperator ausgewertet. Zuerst wird $max(0, 0)$ zu 0 und nach einem weiteren Linksklick mit der Maus $1 + 0$ zu 0 evaluiert. Abbildung 4.6 zeigt das Resultat.

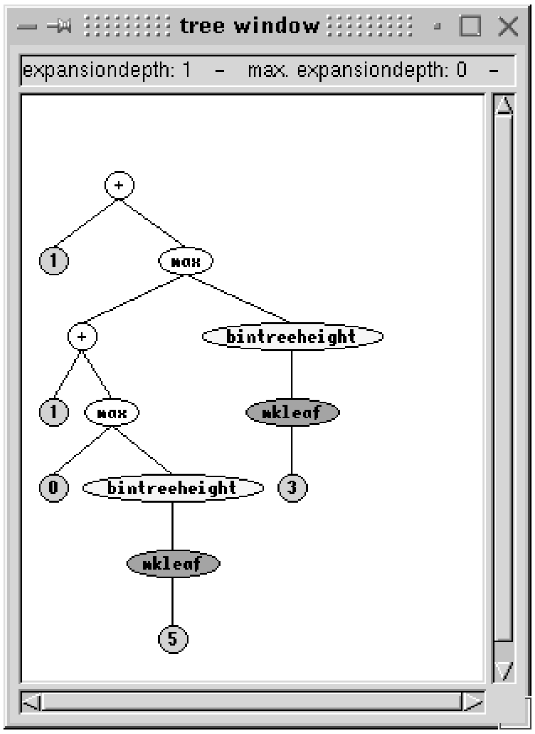


Abbildung 4.4: Der Ausdruck nach dem dritten Auswertungsschritt

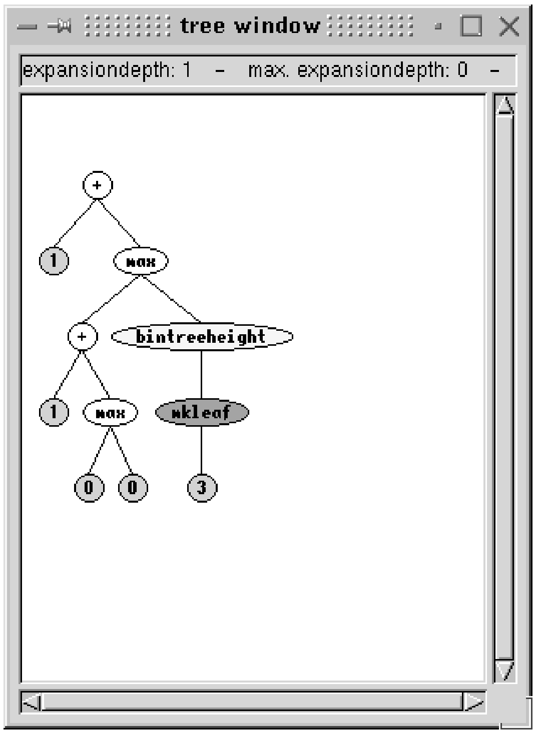


Abbildung 4.5: Der Ausdruck nach dem vierten Auswertungsschritt

Wie im dritten und vierten Schritt wird nun der letzte Aufruf von *bintreeheight* unter Verwendung des ersten Falles expandiert. Anhand von Abbildung 4.7 ist zu erkennen, dass nun nur noch $max(1,0)$ zu 1 und dann $1+1$ zum endgültigem Ergebnis 2 ausgewertet werden müssen.

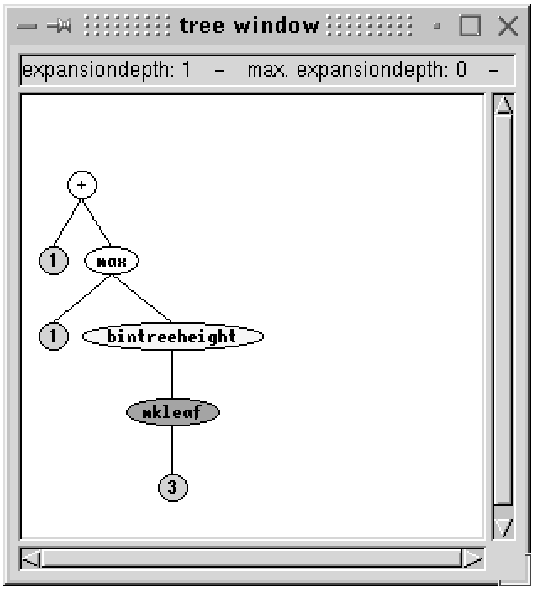


Abbildung 4.6: Der Ausdruck nach sechs Auswertungsschritten

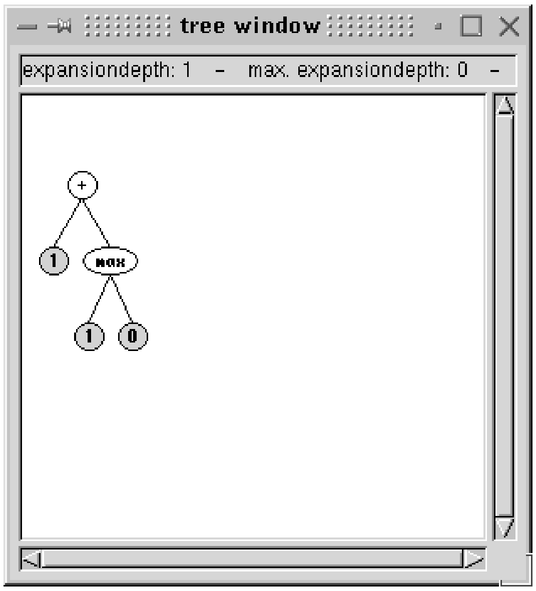


Abbildung 4.7: Der Ausdruck nach sieben Auswertungsschritten

4.2 Beispiel 2

Wie in Beispiel 1 gesehen funktioniert die Auswertung von Rechenvorschriften mit Hilfe der Strategie $leftmost-innermost$ sehr gut. Dies gilt auch für komplexere Beispiele und hängt damit zusammen, dass Aufrufe von Rechenvorschriften erst expandiert werden, wenn bereits alle Argumente ausgewertet wurden.

Bei der Auswertungsstrategie $leftmost - outermost$ kann das jedoch anders aussehen. Diese Problematik wird anhand des folgenden Beispiels erörtert. Es handelt sich hierbei um eine spezielle Berechnung der Fibonacci-Zahlen. Folgender Programmcode muss als ASCII-Textdatei vorliegen und mit KIEL geladen werden:

```
datatype TIntpair = mk of int * int;

fun getFirst  (mk(a,b): TIntpair) : int = a;

fun calcNextFibPair (mk(a,b): TIntpair) : TIntpair = mk(b,a+b);

fun fibPair (n : int) : TIntpair =
  if n = 0
  then mk(1,1)
  else calcNextFibPair(fibPair(n-1));

fun fib_it (n : int) : int = getFirst(fibPair(n));
```

Diese Implementierung basiert letztendlich darauf, dass sich aus einem Paar aufeinander folgender Fibonacci-Zahlen (fib_n, fib_{n+1}) leicht das nachfolgende Paar (fib_{n+1}, fib_{n+2}) berechnen lässt, denn es gilt $(fib_{n+1}, fib_{n+2}) = (fib_{n+1}\ ,\ fib_n + fib_{n+1})$ $(n \geq 0)$. Der Datentyp $TIntpair$ repräsentiert ein Paar ganzer Zahlen und die Hilfsrechenvorschrift $calcNextPair$ berechnet zu einem gegebenen Paar von Fibonacci-Zahlen das nachfolgende. Dies ermöglicht eine iterative Berechnung und die Rechenvorschrift $fibPair$, welche zum übergebenen Parameter n das Paar (fib_n, fib_{n+1}) berechnet, kommt mit nur einem rekursiven Aufruf aus, statt der üblichen zwei $(n \geq 0)$. Die Rechenvorschrift fib_it isoliert aus dem von $fibPair$ gelieferten Paar lediglich die erste Komponente.

Ausgewertet wird im folgenden der Ausdruck $fib\_it(2)$, der mittels des Eingabefensters eingegeben oder aus einer Datei (*"beispiel2.exp"*) geladen wird. Dieser Ausdruck wird anschließend im Ausdrucksübersichtsfenster aufgelistet und kann mittels eines Linksklicks mit der Maus im Baumfenster als solcher angezeigt werden.

Da bei der Auswertung nach der $leftmost-innermost$ Strategie keine Besonderheiten auftreten, muss für dieses Beispiel im Auswertungsfenster auf $leftmost-outermost$ umgeschaltet werden. Die Anzahl der Auswertungsschritte bleibt weiterhin auf Eins.

4.2.1 Auswertung

Wie auch beim vorherigen Beispiel, so wird auch in diesem ein Auswertungsschritt jeweils am Wurzelknoten mittels eines Linksklicks mit der Maus gestartet.

Im ersten Auswertungsschritt kann nur der Aufruf von fib_it expandiert werden. Abbildung 4.8 zeigt den resultierenden Ausdruck.

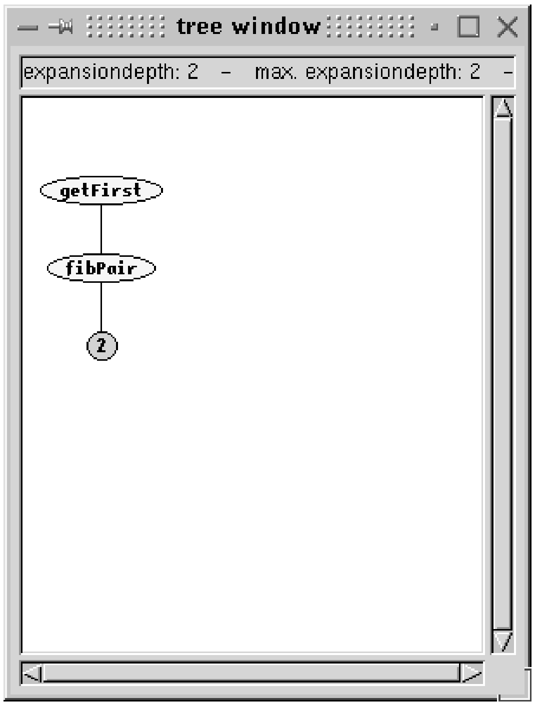


Abbildung 4.8: Nach einem Auswertungsschritt

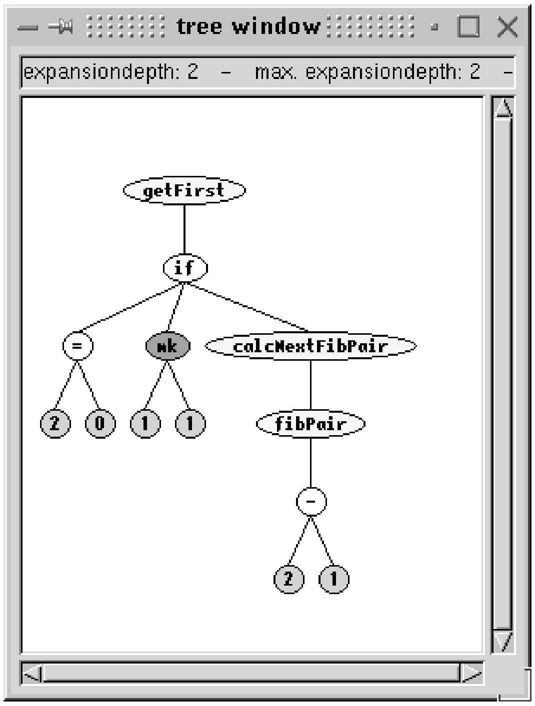


Abbildung 4.9: Nicht $getFirst$ sondern $fibPair$ wurde expandiert

Nach der $leftmost - outermost$ Auswertungsstrategie müsste nun im folgenden Schritt die Rechenvorschrift $getFirst$ expandiert werden. Bei dem an $getFirst$ übergebenen Argument handelt es sich jedoch um einen weiteren Aufruf einer Rechenvorschrift. Da diese jedoch nach 3.2.1 nicht in Mustern enthalten sein dürfen und das Muster des Parameters weder nur aus dem Unterstrich "_" noch nur aus einer Variable besteht, entspricht das Argument nicht dem Muster. Folglich würde die Mustererkennung zu einem Laufzeitfehler führen. Gleiches gilt für alle Basisoperatoren und -funktionen, die nicht in Mustern enthalten sein dürfen. Diese Einschränkung wäre allerdings so erheblich, dass eine sinnvolle Evaluierung nach der $leftmost - outermost$ Strategie nicht mehr möglich wäre.

Aus diesem Grund existiert bei der Auswertung von Rechenvorschriften mit Mustererkennung unter der $leftmost - outermost$ Strategie eine Besonderheit. Findet die

Mustererkennung beim Versuch der Expansion einer Rechenvorschrift keine der Argumentliste entsprechende Parameterliste, so führt dies nur dann zu einem Laufzeitfehler, wenn die einzelnen Argumente jeweils nicht weiter auswertbar[1] sind. Andernfalls wird zuerst die Auswertung der Argumente soweit vorangetrieben, bis sie den Mustern der Parameterliste entsprechen oder nicht weiter auswertbar sind. Nach jedem Auswertungsschritt eines Argumentes wird wiederum eine Mustererkennung durchgeführt, so dass das Argument nach möglichst wenigen Auswertungsschritten dem Muster entspricht. Sollte ein Argument jedoch nicht weiter auswertbar sein und es entspricht immer noch nicht dem zugehörigen Muster, so führt die Mustererkennung auch weiterhin zu einem Laufzeitfehler.

Demzufolge ergibt eine Auswertung des Beispiels um einen weiteren Schritt den Ausdruck, der in Abbildung 4.9 dargestellt wird. Nicht $getFirst$ sondern $fibPair$ wurde expandiert.

Da auch eine Verzweigung nicht der Parameterliste von $getFirst$ entspricht, wird anschließend in zwei Schritten erst die Bedingung $2 = 0$ zu $false$ und dann die Verzweigung selbst zur zweiten Alternative ausgewertet, wie in Abbildung 4.10 zu sehen ist.

Beim fünften Auswertungsschritt wird wieder $getFirst$ nicht expandiert, sondern das Argument. Dabei handelt es sich um den Aufruf der Rechenvorschrift $calcNextFibPair$ welcher nun expandiert werden müsste. Da aber auch deren Argument ein Rechenvorschriftaufruf ist, wird auch $calcNextFibPair$ nicht expandiert, sondern die Auswertung des Argumentes vorangetrieben. Dieser Aufruf von $fibPair$ kann nun wie im zweiten Schritt expandiert werden, da der Parameter nur aus einer Variable besteht. Der resultierende Ausdruck ist in Abbildung 4.11 zu sehen.

In den nachfolgenden drei Schritten wird wiederum die Bedingung der Verzweigung zu $false$ und somit die Verzweigung selbst zur zweiten Alternative ausgewertet (siehe Abbildung 4.12). Aus den bekannten Gründen wird wiederum $fibPair$ expandiert, wie Abbildung 4.13 zeigt.

Diesmal wird die Bedingung jedoch zu $true$ ausgewertet und die Verzweigung daher zur ersten Alternative (siehe Abbildung 4.14). Diese Alternative besteht aus dem Paar der Fibonacci-Zahlen von 0 und 1. Im nächsten Auswertungsschritt wird aus den schon genannten Gründen der untere Aufruf von $calcNextFibPair$ expandiert, denn das Argument entspricht diesmal dem Parameter. So wird aus dem Paar der Fibonacci-Zahlen von 0 und 1 das Paar der Fibonacci-Zahlen von 1 und 2 berechnet. Gleiches gilt für den folgenden Schritt, in dem wiederum $calcNextFibPair$ expandiert und somit das Paar

[1] d.h. sie bestehen nur aus Konstanten und Konstruktoren

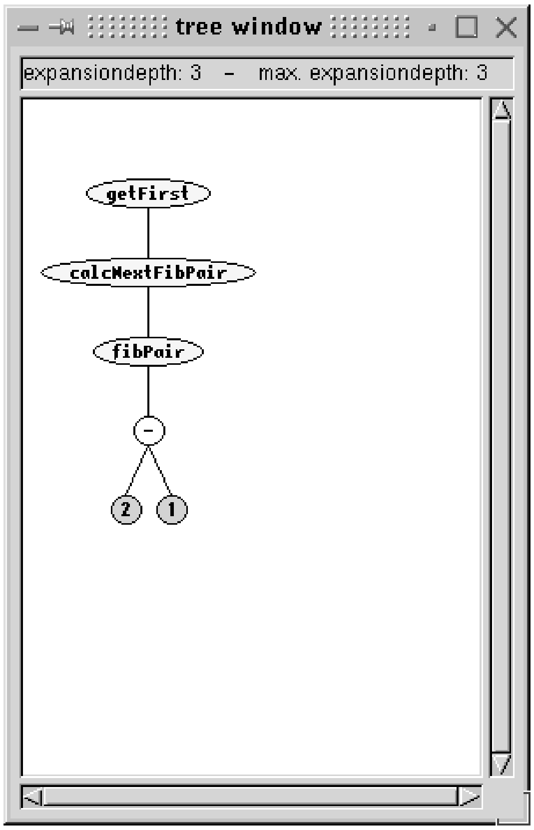


Abbildung 4.10: Nach dem vierten Auswertungsschritt

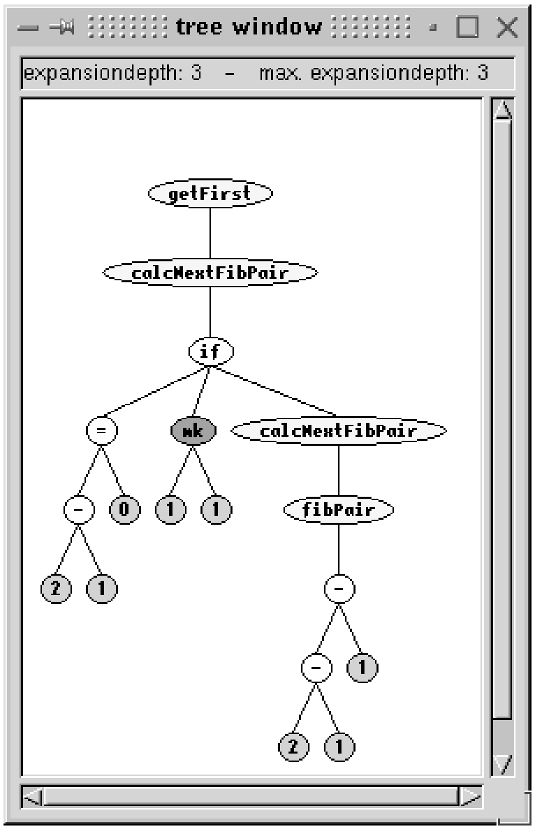


Abbildung 4.11: Die Auswertung wurde mehrfach ins Argument vorgetrieben

der Fibonacci-Zahlen von 2 und 3 berechnet wird (siehe Abbildung 4.15).

Nun wird schließlich die Rechenvorschrift $getFirst$ expandiert, welche aus dem übergebenen Paar die erste Komponente isoliert. In diesem Beispiel handelt es sich also um einen Ausdruck, der die Fibonacci-Zahlen von 2 repräsentiert. Nun muss nur noch der Plusoperator ausgewertet werden und das endgültige Ergebnis 2 steht fest.

4.2.2 Anmerkung:

Auffällig an diesem Beispiel ist, dass der Ausdruck zwar nach der $leftmost-outermost$ Strategie ausgewertet wurde, die tatsächliche Reihenfolge der Expansionen, wegen der Besonderheit für Rechenvorschriften mit Mustererkennung, aber fast der der $leftmost-innermost$ Strategie entsprach. Nur die Basisoperatoren Plus und Minus wurden nicht vor der Expansion einer Rechenvorschrift sondern verzögert ausgewertet.

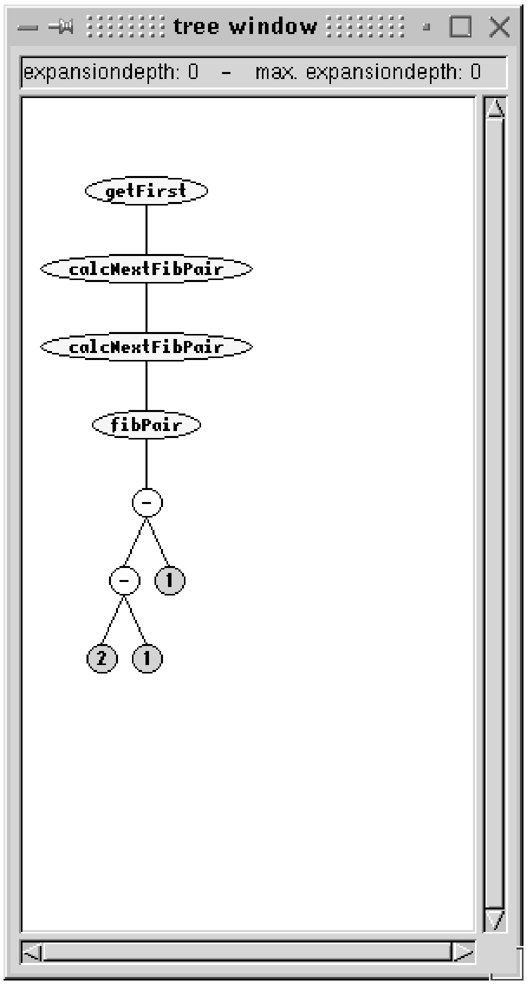


Abbildung 4.12: Nach dem achten Auswertungsschritt

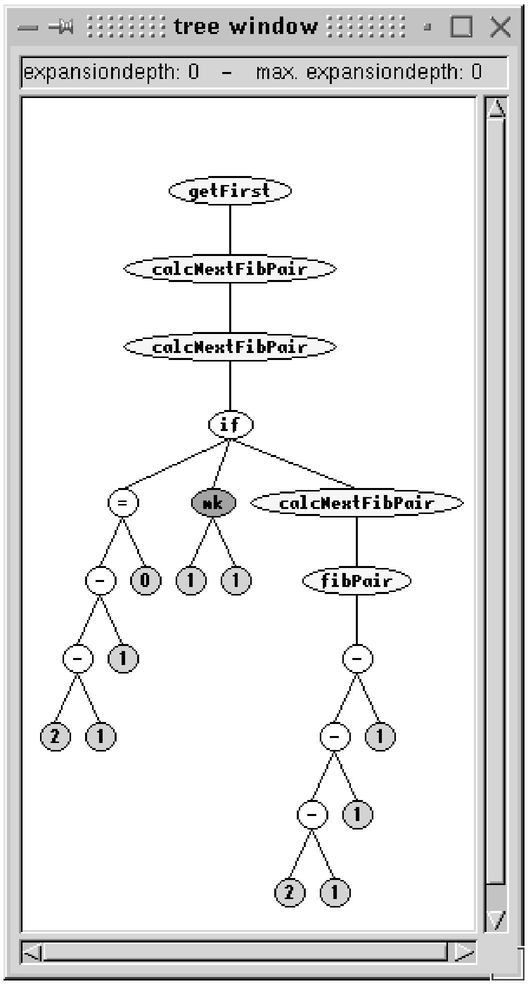


Abbildung 4.13: Die Auswertung wurde mehrfach ins Argument vorgetrieben

Ohne Kenntnis über die konkrete Deklaration einer Rechenvorschrift ist bei der $leftmost - outermost$ Strategie nicht mehr entscheidbar, welcher Aufruf innerhalb eines Ausdrucks als nächster zu expandieren ist.

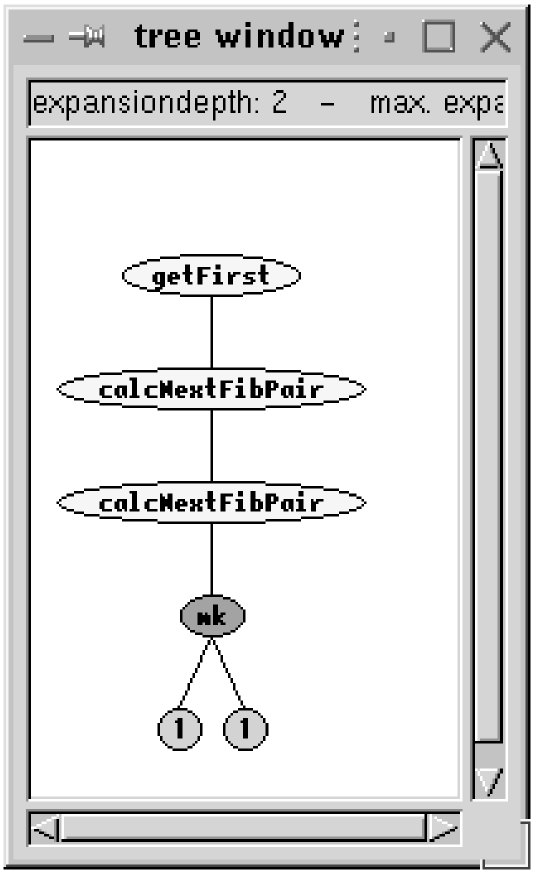


Abbildung 4.14: Die Rekursion stoppt

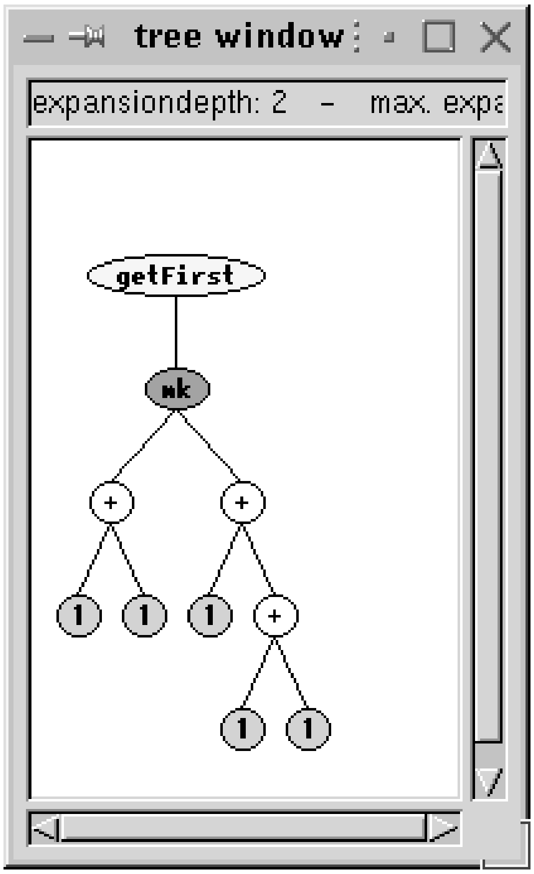


Abbildung 4.15: Nach 15 Auswertungsschritten

4.3 Beispiel 3

Wie das Beispiel 2 schon zeigte, sind bei der Verwendung von Rechenvorschriften mit Mustererkennung unter der $leftmost - outermost$ Strategie Besonderheiten zu berücksichtigen. Dieses Beispiel veranschaulicht, dass Rechenvorschriften ohne die Ausnutzung der Mustererkennung nicht ohne weiteres in solche mit Mustererkennung umgeschrieben werden können. Auch diese Problematik ist auf die $leftmost - outermost$ beschränkt.

Als Beispiel dient die Morris-Funktion in zwei Varianten. Die erste ist die Ursprüngliche und die zweite eine mit Verwendung von Mustererkennung.

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

fun morris_pm(0: int, _: int): int = 1
|   morris_pm(x: int, y: int): int = morris_pm(x-1, morris_pm(x-y,y));
```

Wie der Programmcode zeigt, ist die Verzweigung der ursprünglichen Variante in der anderen durch Mustererkennung ersetzt worden, so dass sie naiv betrachtet, gleichwertig sein sollten.

4.3.1 Auswertung

Unter der $leftmost - innermost$ Auswertungsstrategie wird der Ausdruck $morris(1,0)$ in fünf Schritten zu $morris(0, morris(1,0))$ evaluiert. Da im resultierenden Ausdruck der ursprüngliche Ausdruck $morris(1,0)$ als Teilausdruck wieder auftritt und dieser als nächstes ausgewertet wird, terminiert diese Berechnung nie. Das gleiche gilt für $morris\_pm(1,0)$. Dieser Ausdruck wird jedoch in nur drei Schritten zu $morris\_pm(0, morris\_pm(1,0))$ berechnet, da sozusagen die Mustererkennung bei der Expansion schon die Verzweigung auswertet.

Unter der $leftmost - outermost$ Strategie unterscheiden sich die Berechnungen der beiden Ausdrücke jedoch. In drei Schritten wird $morris(1,0)$ zu dem Ausdruck aus Abbildung 4.16 und $morris\_pm(1,0)$ zu dem ähnlichen aus Abbildung 4.17 evaluiert.

Anschließend wird in beiden Ausdrücken der jeweils obere Aufruf expandiert. Obwohl der Teilausdruck $1 - 1$ die Null repräsentiert, ist er noch nicht vollständig ausgewertet, so dass $morris\_pm$ mittels des zweiten und nicht mittels des ersten Falles expandiert wird (siehe Abbildung 4.18 und 4.19).

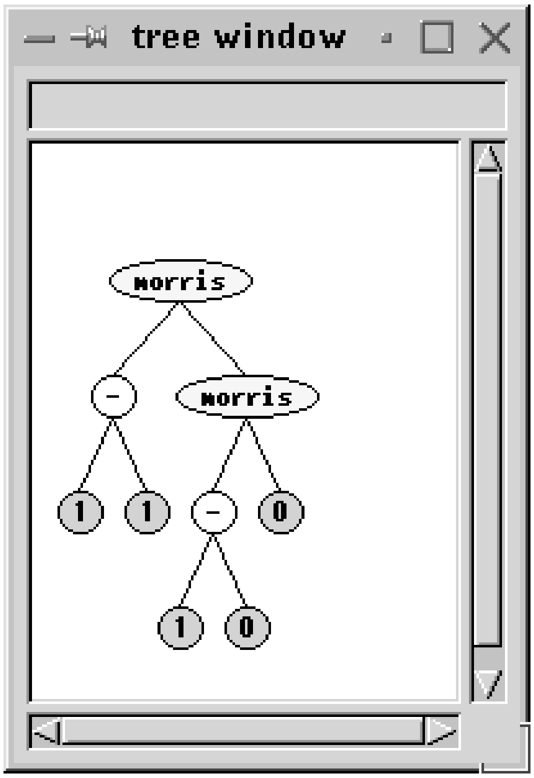


Abbildung 4.16: Nach drei Schritten

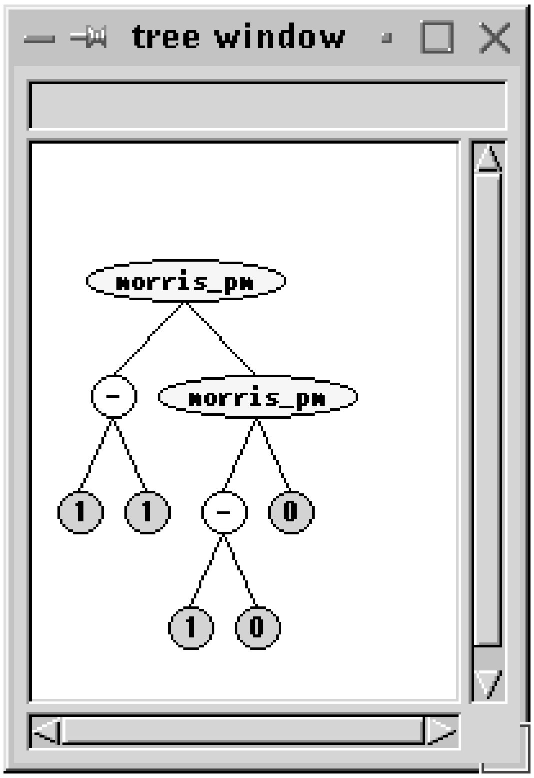


Abbildung 4.17: Nach nur einem Schritt

Im Ausdruck aus Abbildung 4.18 wird nun zuerst die Bedingung und dann die Verzweigung selbst zur ersten Alternative ausgewertet, was zum endgültigen Ergebnis 1 führt. Im anderen Ausdruck steht jedoch als Wurzel ein Aufruf von $morris\_pm$. Dieser wird mittels des zweiten Falles von $morris\_pm$ expandiert, da der Parameter des ersten Falles nur aus der Konstante 0 besteht, das Argument jedoch aus einem komplexeren Ausdruck. Nach dieser Expansion steht aber wieder ein Aufruf von $morris\_pm$ als Wurzel, so dass diese Berechnung nie terminiert.

4.3.2 Anmerkung:

Die in diesem Beispiel angesprochene Problematik zeigt, dass die Mustererkennung nicht so optimal mit der $leftmost-outermost$ Auswertungsstrategie harmoniert, wie das der $leftmost - innermost$ Strategie gelingt. Sie kann nur auftreten, wenn eine Rechenvorschrift aus mindestens zwei Fällen besteht, wobei einer, im Allgemeinen der letzte, eine Parameterliste enthält, der jede beliebige Argumentliste entspricht, d.h. die Parameterliste ausschließlich aus Variablen besteht.

Bei Datentypen, für die die Mustererkennung ja eingeführt wurde, tritt dieser Fall in

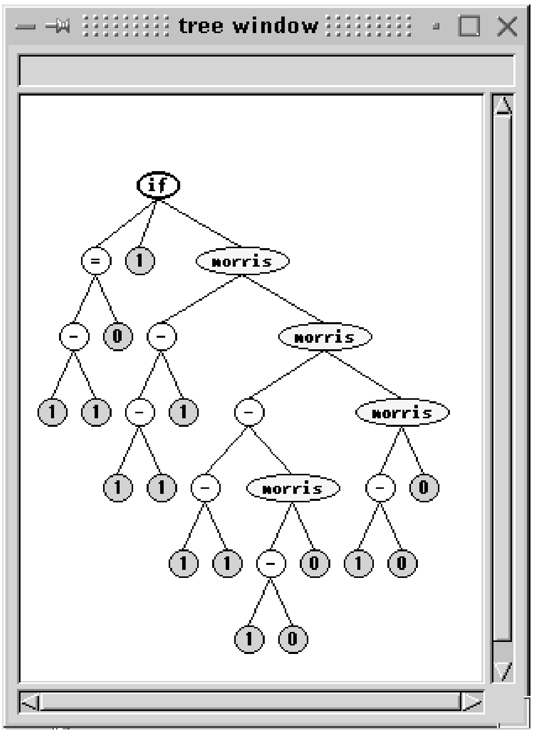


Abbildung 4.18: Nach der Expansion von *morris*

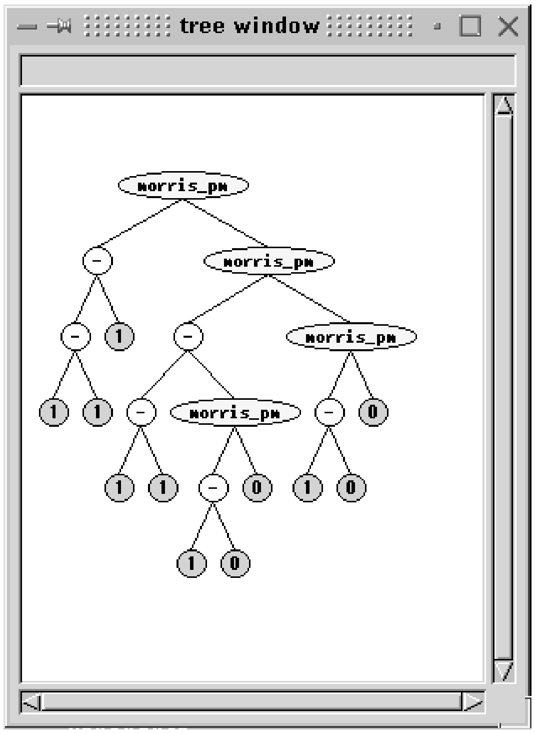


Abbildung 4.19: Nach der Expansion von *morris_pm*

der Praxis kaum auf. Bei Datentypen existiert meist für jeden deklarierten Konstruktor ein entsprechender Fall in der Rechenvorschrift.

4.4 Beispiel 4

Während die Beispiele 2 und 3 problematische Aspekte der Mustererkennung unter der $leftmost - outermost$ Auswertungsstrategie aufzeigen, ermöglicht diese Strategie andererseits den Umgang mit unendlichen Strukturen.

Dieses Beispiel besteht aus einem Datentyp, der eine einfache Liste ganzer Zahlen realisiert, einer rekursiven und für sich allein betrachtet nicht terminierenden Rechenvorschrift, die von einem Startwert an die unendliche Liste dieses Startwertes und all seiner Nachfolger generiert und einer Rechenvorschrift, die aus einer Liste die ersten n geraden Zahlen herausfiltert.

Folgender Programmtext muss als ASCII-Datei vorliegen und dann mittels KIEL geladen werden:

```
datatype TList = empty | mkl of int * TList;


fun genInfList(start: int): TList =
  mkl(start, genInfList(start + 1));


fun filterEven(empty     : TList, _: int) : TList =
  empty
|   filterEven(mkl(x, xs): TList, n: int) : TList =
  if n = 0
  then empty
  else if (x mod 2) = 0
       then mkl(x, filterEven(xs, n - 1))
       else filterEven(xs, n);
```

Beispielhaft wird der Ausdruck

```
filterEven(genInfList(0), 2)
```

berechnet.

4.4.1 Auswertung

Bei der Auswertung nach der $leftmost - innermost$ Strategie wird schnell klar, dass sie nicht terminiert. Ausgehend vom eingegebenen Ausdruck (siehe Abbildung 4.20) wird zuerst der Aufruf von $genInfList$ expandiert (siehe Abbildung 4.21). Dabei wird jedes Mal ein weiterer Aufruf von $genInfList$ erzeugt. In den folgenden zwei Schritten wird zuerst das Argument von $genInfList$ berechnet und dann wiederum $genInfList$ expandiert, so dass die Berechnung nicht terminiert.

Anders ist das jedoch bei der $leftmost-outermost$ Strategie. Da die Argumentliste von $filterEven$ keiner Parameterliste entspricht, wird die Auswertung bei den Argumenten fortgesetzt. Beim zweiten handelt es sich jedoch um eine Konstante. Daher wird wie bei der $leftmost-innermost$ Strategie der Aufruf von $genInfList$ expandiert (siehe Abbildung 4.21).

Aber schon jetzt kann an der Wurzel des Ausdrucks der Aufruf von $filterEven$ expandiert werden. Abbildung 4.22 zeigt den resultierenden Ausdruck.

Nun wird die Bedingung $2 = 0$, welche prüft, ob noch weitere Elemente gefiltert werden dürfen, zu $false$ und damit die Verzweigung zur zweiten Alternative ausgewertet.

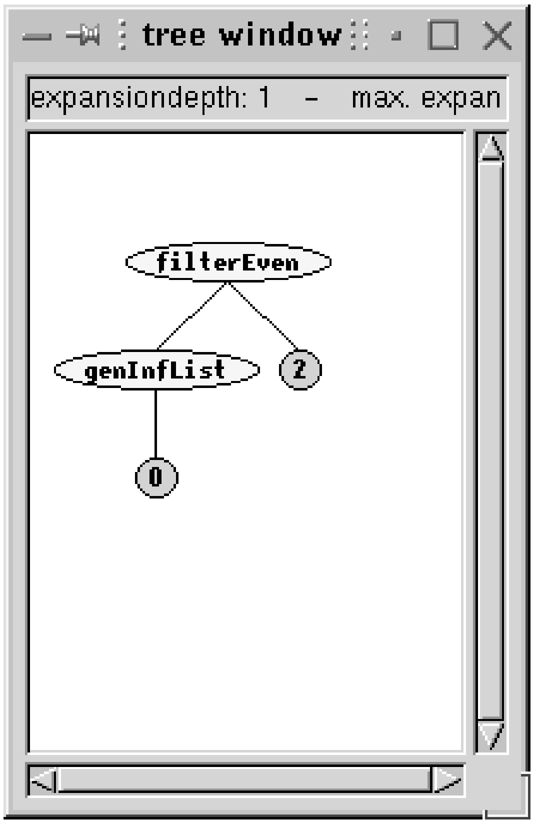


Abbildung 4.20: Der eingegebene Ausdruck

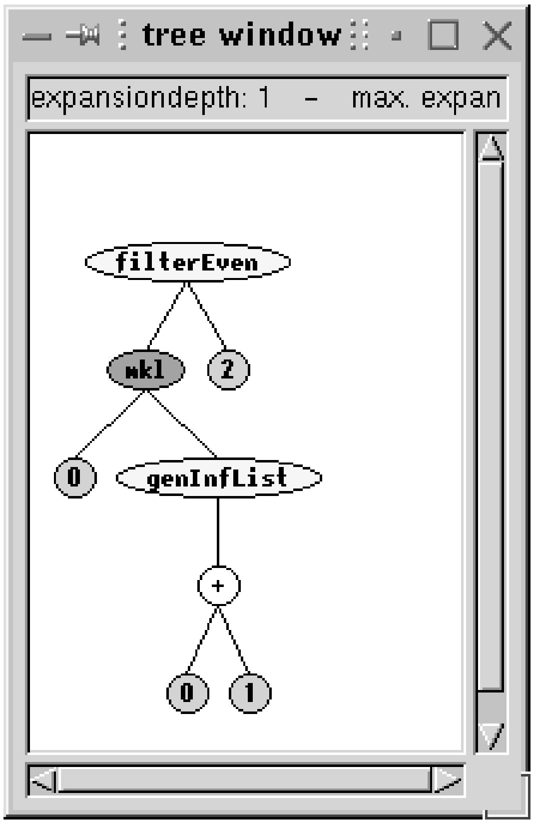


Abbildung 4.21: Nach einem Auswertungsschritt unter beiden Strategien

Die übrig bleibende Bedingung $(0\ mod\ 2) = 0$, die eigentliche Filterung, wird zu *true* und damit die Verzweigung zur ersten Alternative ausgewertet (siehe Abbildung 4.23). Damit ist das erste Element der Resultatliste bestimmt und der Rest der Liste wird durch den rekursiven Aufruf von $filterEven$ erzeugt. Dieser Aufruf filtert nun die erste gerade Zahl aus der Liste der Ganzen Zahlen beginnend mit der Eins.

Nach der Auswertung von $2 - 1$ zu 1 wird wiederum $genInfList$ expandiert, da die Mustererkennung bei $filterEven$ noch fehlschlägt. Nach dieser Expansion kann aber $filterEven$ zu dem Ausdruck aus Abbildung 4.24 expandiert werden.

Die Bedingung der ersten Verzweigung wird wiederum zu $false$ evaluiert, aber die zweite diesmal auch zu $false$. Damit ergibt sich der Ausdruck aus Abbildung 4.25 und es wurde kein Element der Resultatliste hinzugefügt.

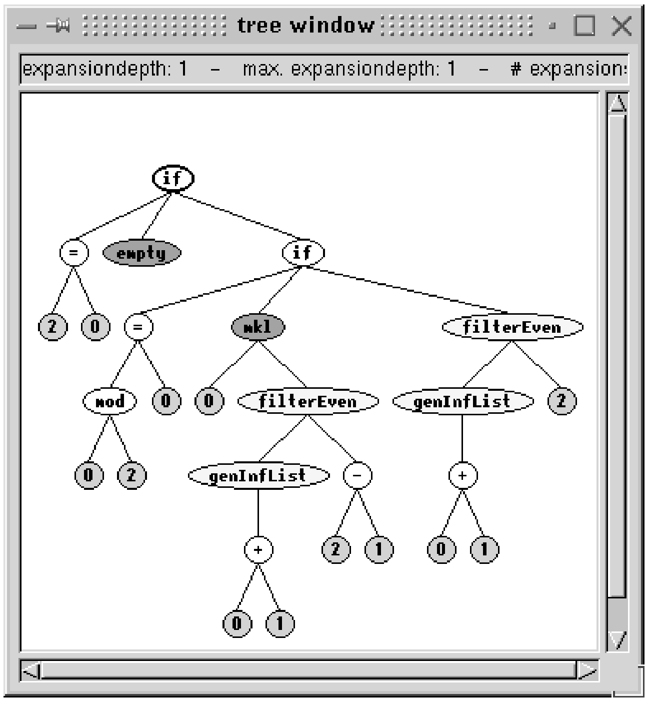


Abbildung 4.22: Nach der ersten Expansion von *filterEven*

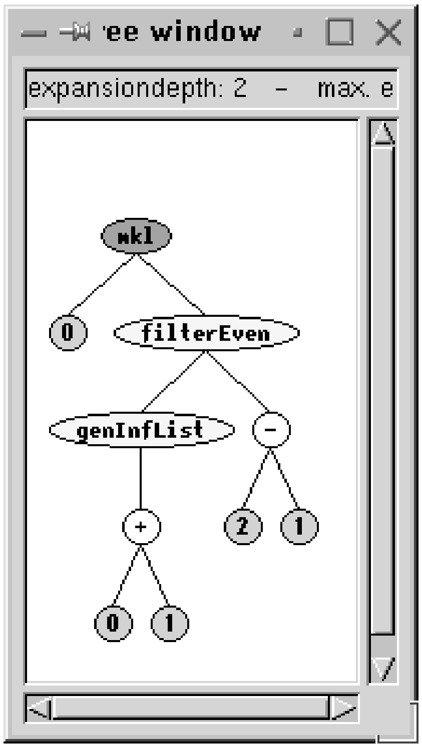


Abbildung 4.23: Das erste Element der Resultatliste steht fest

Aus bekannten Gründen wird erneut erst *genInfList* expandiert(siehe Abbildung 4.26).

Anschließend kann sofort der Aufruf von *filterEven* expandiert werden. Wie bei der ersten Expansion von *filterEven* wird die erste Bedingung zu *false* und die zweite zu *true* ausgewertet, so dass die zweite Verzweigung zur ersten Alternative reduziert wird. Dadurch wird nun das zweite, jedoch noch nicht vollständig ausgewertete, Element der Resultatliste hinzufügt. Die Berechnung dieses Elementes erfolgt direkt im Anschluss und das Ergebnis ist in Abbildung 4.27 zu sehen. Das zweite Argument von *filterEven* wird nun zu Null ausgewertet und wird somit im weiteren Verlauf der Evaluierung zur Terminierung führen.

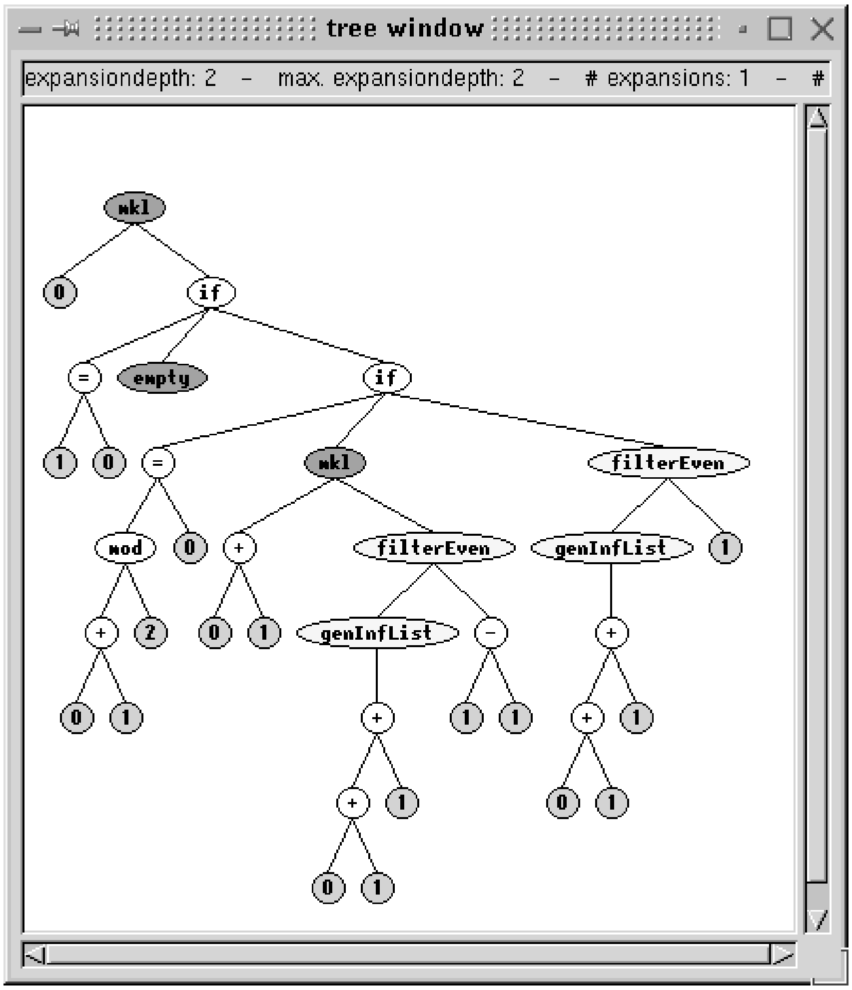


Abbildung 4.24: Nach der zweiten Expansion von $filterEven$

Doch vorher werden noch zuerst $genInfList$ und dann $filterEven$ expandiert. Dieses Mal wird die erste Bedingung $0 = 0$ aber zu $true$ ausgewertet, so dass die Verzweigung selbst zur ersten Alternative $empty$ reduziert wird. Damit steht das endgültige Resultat der Auswertung, wie in Abbildung 4.28 zu sehen, fest. Trotz der für sich betrachtet nicht terminierenden Rechenvorschrift $genInfList$ terminiert die Berechnung von $filterEven(genInfList(0), 2)$.

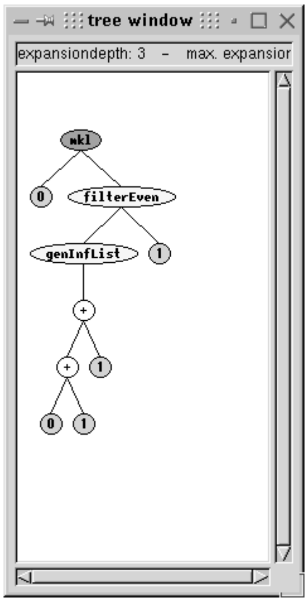


Abbildung 4.25: Die Auswertung der Verzweigungen ergibt diesen Ausdruck

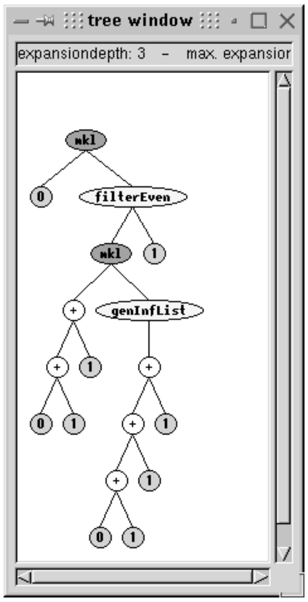


Abbildung 4.26: Wieder wird zuerst *genInfList* expandiert

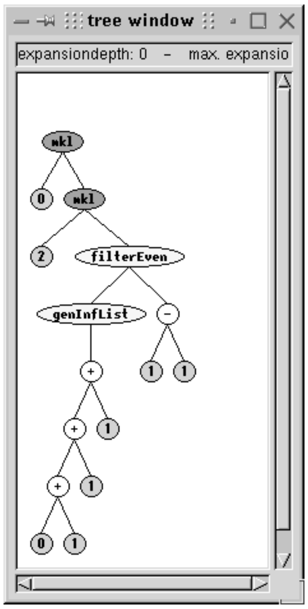


Abbildung 4.27: Durch die Null im zweiten Argument steht die Terminierung schon fest

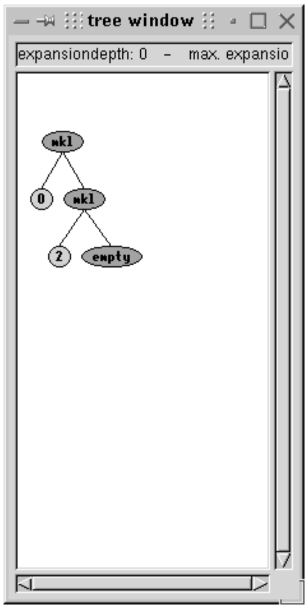


Abbildung 4.28: Das Endergebnis

4.5 Beispiel 5

Als Letztes wird ein Beispiel durchgerechnet, welches geschachtelte Listen verwendet. Dazu wird auf das schon in Abschnitt 3.3 angeführte Transponieren von Matrizen zurückgegriffen. Um die Abbildungen übersichtlicher zu gestalten, werden für die Elemente der Matrix nur noch Ganze Zahlen zugelassen, so dass der verwendete Typ für eine Matrix nun *int list list* ist. Eine $m \times n$-Matrix besteht also aus einer Liste von m Elementen, die die Zeilen der Matrix repräsentieren. Diese Zeilenelemente sind wiederum Listen mit n Elementen, die nun die einzelnen Einträge einer Matrix darstellen.

Folgender im Gegensatz zu Abschnitt 3.3 leicht veränderter[2] Programmtext muss als ASCII-Datei vorliegen und dann mittels KIEL geladen werden:

```
fun headsOfRows(nil                : int list list): int list =
  nil
|   headsOfRows((head :: _) :: rows: int list list): int list =
  head :: headsOfRows(rows);

fun tailsOfRows(nil                : int list list): int list list =
  nil
|   tailsOfRows((_ :: tail) :: rows: int list list): int list list =
  tail :: tailsOfRows(rows);

fun transpose(nil             : int list list): int list list =
  nil
|   transpose(nil :: nilrows  : int list list): int list list =
  nil
|   transpose(A               : int list list): int list list =
  headsOfRows(A) :: transpose(tailsOfRows(A));
```

Da in den beiden Hilfsfunktionen $headsOfRows$ und $tailsOfRows$ komplexere Muster verwendet werden, als in den vorherigen Beispielen, wird an ihnen die Bindung der Teilausdrücke eines Arguments an die Variablen im Muster demonstriert.

Beispielhaft wird die 3×2-Matrix

$$\begin{pmatrix} 11 & 12 \\ 21 & 22 \\ 31 & 32 \end{pmatrix}$$

transponiert. Dazu wird folgender Ausdruck in KIEL eingegeben:

```
transpose( [ [11,12], [21,22], [31,32] ] )
```

[2] *real* wurde durch *int* ersetzt

4.5.1 Auswertung

$leftmost - innermost$
Die Auswertung erfolgt zuerst nach der $leftmost - innermost$ Strategie. Abbildung 4.29 zeigt wie der eingegebenen Ausdruck im Baumfenster dargestellt wird.

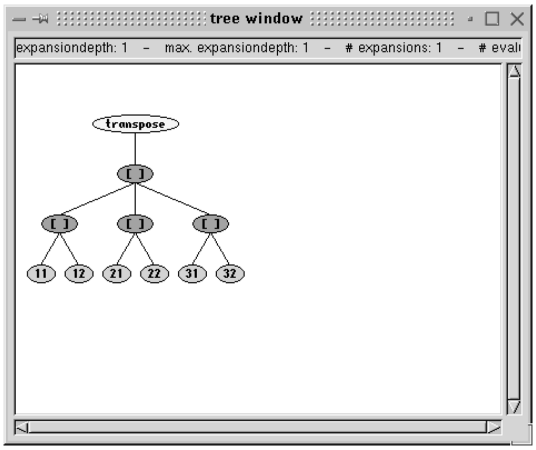


Abbildung 4.29: Der eingegebene Ausdruck

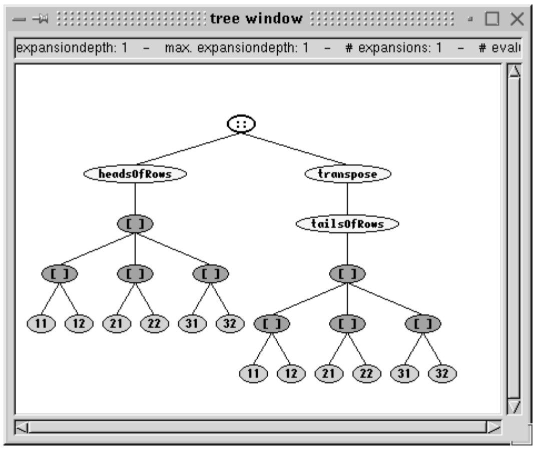


Abbildung 4.30: Nach dem ersten Auswertungsschritt

Wie bei allen vorherigen Beispielen muss auch bei diesem Ausdruck zuerst eine Expansion erfolgen. Um den Aufruf der Rechenvorschrift *transpose* zu expandieren, muss zuerst ermittelt werden, mittels welchem der drei Fälle, aus denen *transpose* besteht, dies zu erfolgen hat. Das an *transpose* übergebene Argument ist eine Liste mit drei Elementen. Da das Muster des ersten Falles ausschließlich aus der Konstanten *nil* besteht, welche die leere Liste repräsentiert, entspricht das Argument nicht dem Muster. Das Muster des zweiten Falles ist $nil :: nilrows$. Diesem Muster entspricht jede Liste, deren erstes Element eine leere Liste ist, was auf das übergebene Argument nicht zutrifft. Im dritten Fall besteht das Muster nur aus der Variable A, so dass jedes Argument passt. Den mittels des dritten Falles expandierten Ausdruck zeigt Abbildung 4.30.

Als Nächstes wird der Aufruf der Hilfsrechenvorschrift $headOfRows$ expandiert. Die Auswertung dieses Teilausdrucks liefert die erste Zeile der transponierten Matrix. Da das Muster des ersten Falles wiederum aus der Konstante *nil* besteht und das Argument von *transpose* unverändert an $headsOfRows$ übergeben wird, passen sie nicht zusammen. Das Muster des zweiten Falles ist komplexer aufgebaut:
$(head :: \_) :: rows$. Ein Ausdruck entspricht diesem Muster wenn es sich um eine Liste mit mindestens einem Element handelt ($X :: rows$), wobei dieses Element wiederum eine Liste mit einem Element ist ($head :: \_$). Demzufolge entspricht das übergebene Argument dem Muster, wobei der Ausdruck 11 an die Variable *head* und $[[21, 22], [31, 32]]$

an *rows* gebunden wird. Das Resultat dieses Auswertungsschritts ist in Abbildung 4.31 zu sehen.

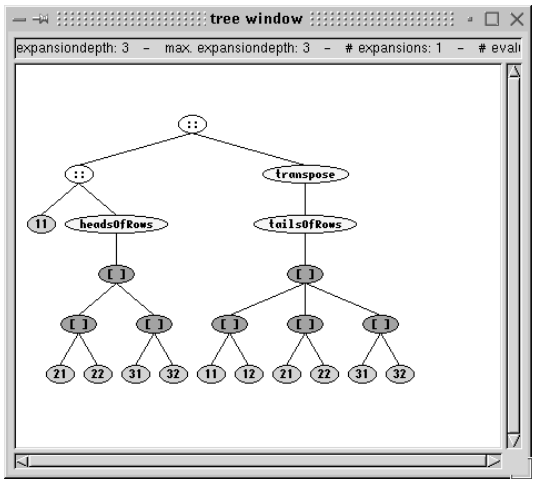


Abbildung 4.31: Nach der Expansion von *headsOfRows*

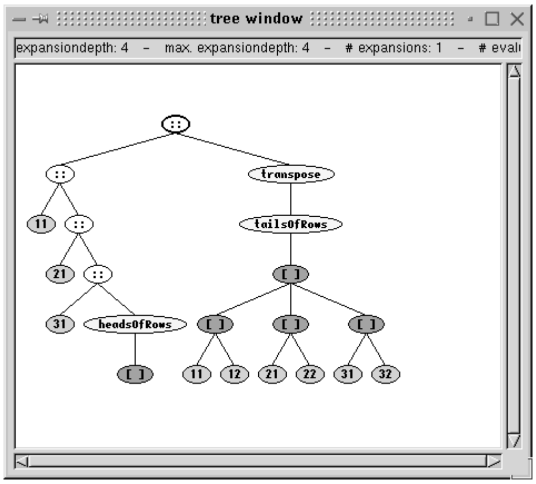


Abbildung 4.32: Nach zwei weiteren Expansionen von *headsOfRows*

Genau wie im vorherigen Schritt wird nun wiederum der Aufruf von *headsOfRows* mittels des zweiten Falles expandiert, wobei 21 an *head* und $[[31, 32]]$ an *rows* gebunden wird. Dies geschieht auch ein drittes Mal mit der Bindung von 31 an *head* und von $[\,]$ an *rows*, so dass der Ausdruck aus Abbildung 4.32 entsteht.

Im folgenden Auswertungsschritt wird wiederum *headsOfRows* expandiert. Da das übergebene Argument aber diesmal die leere Liste ist, wird der Aufruf mittels des ersten Falles expandiert, so dass der Aufruf durch die leere Liste ersetzt wird. Anschließend werden die drei unteren :: Operatoren zur Liste $[11, 21, 31]$ ausgewertet, was der ersten Zeile der transponierten Matrix entspricht. Abbildung 4.33 stellt diesen Ausdruck dar.

Da die $leftmost - innermost$ Auswertungsstrategie es vorschreibt vor der Expansion eines Rechenvorschriftaufrufs seine Argumente auszuwerten, wird vor *transpose* erst einmal *tailsOfRows* expandiert. Diese Hilfsrechenvorschrift macht aus der 3×2-Matrix eine 3×1-Matrix, indem die ersten Elemente der Zeilenlisten entfernt werden. Das sind die, die *headsOfRows* vorher zur ersten Zeile der Transponierten zusammengefasst hat. Ganz ähnlich wie bei *headsOfRows* funktioniert auch bei *tailsOfRows* die Mustererkennung. Das Muster des ersten Falles ist *nil* und passt nicht zum Argument. Aber das Argument entspricht dem Muster $(\_ :: tail) :: rows$ mit der selben Argumentation wie bei *headsOfRows*. Wie Abbildung 4.34 zeigt, wird $[12]$ an *tail* und $[[21, 22], [31, 32]]$ an *rows* gebunden.

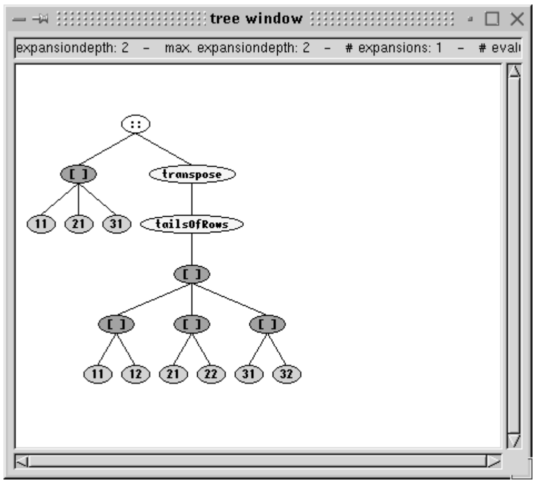


Abbildung 4.33: Die erste Zeile der Transponierten ist berechnet

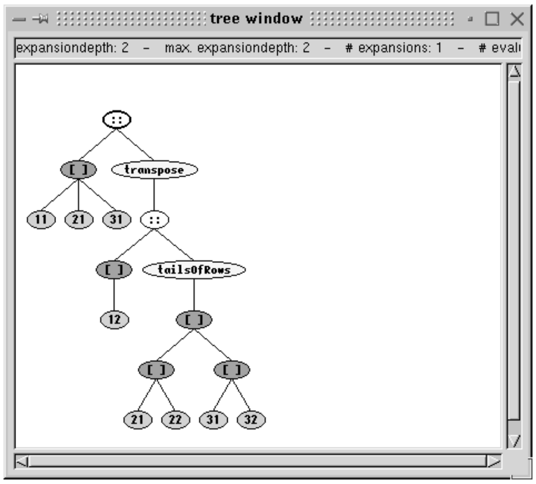


Abbildung 4.34: Nach der Expansion von $tailsOfRows$

Die folgenden zwei Schritte erfolgen analog zum vorherigen. Beim ersten Schritt wird [22] an *tail* und [[31, 32]] an $rows$ gebunden und beim zweiten [32] an *tail* und [ ] an $rows$. Wie Abbildung 4.35 zeigt erhält $tailsOfRows$ nun die leere Liste als Argument, so dass dieser Aufruf mittels des ersten Falles expandiert wird. Dementsprechend wird der Aufruf durch die leere Liste ersetzt. Vor der Expansion von *transpose* werden die drei unteren :: Operatoren evaluiert, so dass der Ausdruck aus Abbildung 4.36 entsteht.

Wie bei der ersten Expansion von *transpose* wird die jetzige wiederum mittels des dritten Falles durchgeführt (siehe Abbildung 4.37). Nach bekannter Weise werden dann viermal $headsOfRows$-Aufrufe expandiert, dann dreimal drei :: Operatoren ausgewertet, dann vier Aufrufe von $tailsOfRows$ expandiert und wieder drei :: Operatoren berechnet. Der resultierende Ausdruck ist in Abbildung 4.38 zu sehen.

Nun muss wieder *transpose* expandiert werden. Das übergebene Argument ist aber diesmal eine Liste von drei leeren Listen. D.h., dass das Argument dem Muster *nil* des ersten Falles zwar nicht entspricht, aber es entspricht dem Muster *nil* :: $nilrows$, so dass der *transpose*-Aufruf diesmal mittels des zweiten Falles zur leeren Liste expandiert wird (siehe Abbildung 4.39). Nach zwei weiteren Auswertungen des :: Operators wird das endgültige Resultat in Abbildung 4.40 gezeigt. Dieser Ausdruck stellt die Matrix

$$\begin{pmatrix} 11 & 21 & 31 \\ 12 & 22 & 32 \end{pmatrix}$$

dar, die auch tatsächlich die Transponierte der Ursprungsmatrix ist.

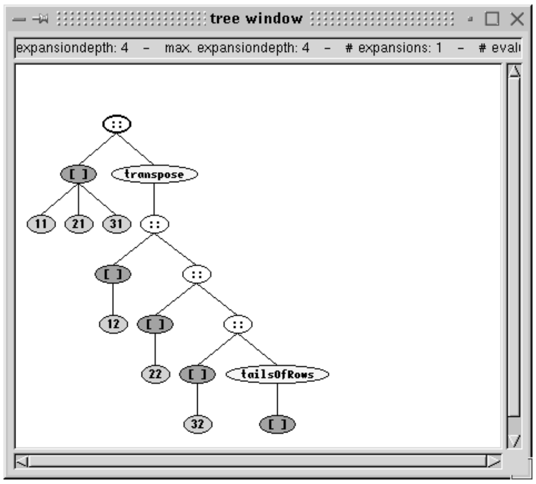


Abbildung 4.35: Nach zwei weiteren Expansionen von $tailsOfRows$

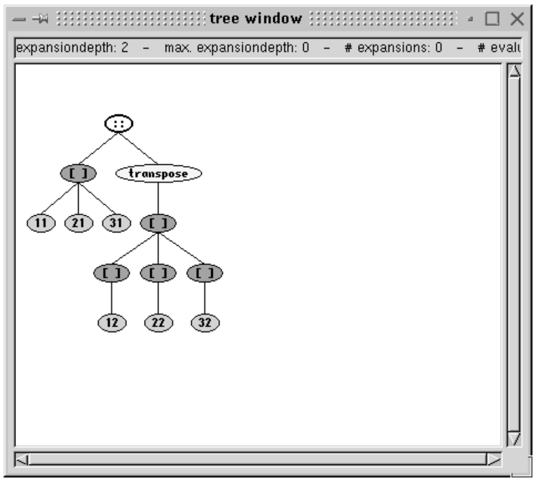


Abbildung 4.36: $tailsOfRows$ liefert insgesamt eine 3×1 Matrix

$leftmost - outermost$

Die Auswertung nach der $leftmost-outermost$ Strategie führt nicht zum selben Ergebnis wie vorherige nach $leftmost-innermost$. Das liegt an dem aus Beispiel 3 bekannten Problem.

Zuerst wird der Aufruf von *transpose* expandiert und anschließend der gesamte Teilausdruck ab dem Aufruf von $headsOfRows$ evaluiert. Bis zum Ausdruck aus Abbildung 4.33 erfolgt die Auswertung also so, wie nach der $leftmost - innermost$ Strategie.

Nun wird eine Mustererkennung für den Rechenvorschriftaufruf von *transpose* durchgeführt. Da das übergebene Argument ein weiterer Aufruf einer Rechenvorschrift ist, entspricht es keinem der ersten beiden Fälle. Beim Muster des dritten Falles handelt es sich jedoch um eine Variable, so dass eine Expansion nach dem dritten Fall erfolgt, wie in Abbildung 4.41 zu sehen ist.

Anschließend werden abwechselnd der linke Aufruf von $tailsOfRows$ und der Aufruf von $headsOfRows$ expandiert, bis der gesamte Teilausdruck ab dem $headsOfRows$-Aufruf berechnet ist. Da die Rechenvorschrift $headsOfRows$ keinen Fall enthält, dem jedes Argument entspricht, wird immer zuerst $tailsOfRows$ expandiert. Nach der ersten Expansion entsteht der Ausdruck aus Abbildung 4.42. Die Abbildung verdeutlicht weiterhin, dass nach der Expansion von $tailsOfRows$ immer der Aufruf von $headsOfRows$ mittels des zweiten Falles expandiert werden kann.

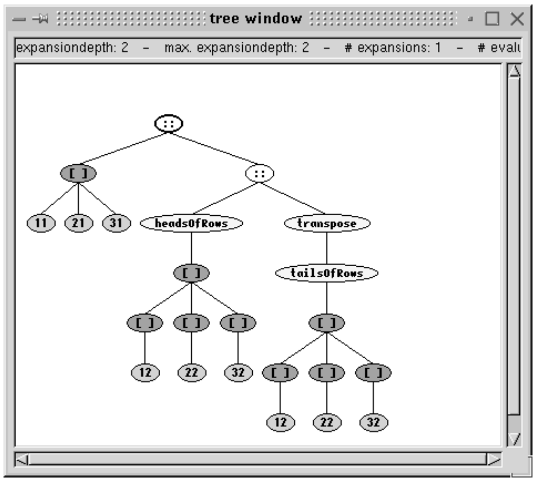


Abbildung 4.37: Nach der zweiten Expansion von *transpose*

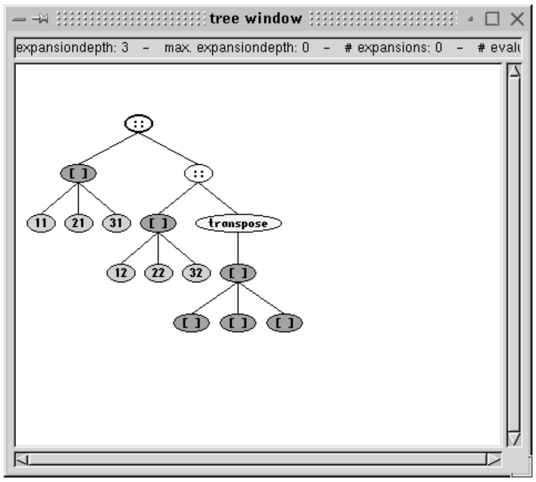


Abbildung 4.38: Nach der Berechnung der zweiten Zeile der Transponierten

Wenn nun wieder die Expansion des Aufrufs von *transpose* als nächster Schritt auszuführen ist, zeigt sich, dass wieder das Problem aus Beispiel 3 aufgetreten ist (siehe Abbildung 4.43). Nach zwei weiteren Auswertungsschritten, die in der Expansion der linken *tailsOfRows*-Aufrufen bestanden, steht schon fest, dass es zu einem Laufzeitfehler kommen wird. Abbildung 4.44 zeigt deutlich, dass das an *headsOfRows* übergebene Argument eine Liste sein wird, deren erstes Element die leere Liste ist. Ein solches Argument entspricht aber keinem der zwei Fälle von *headsOfRows*. Wenn nach 9 weiteren Auswertungsschritten das Argument von *headsOfRows* komplett ausgewertet ist, kommt es im folgenden Schritt zu einem Laufzeitfehler, da es weder einem Muster eines Falles entspricht, noch weiter auswertbar ist.

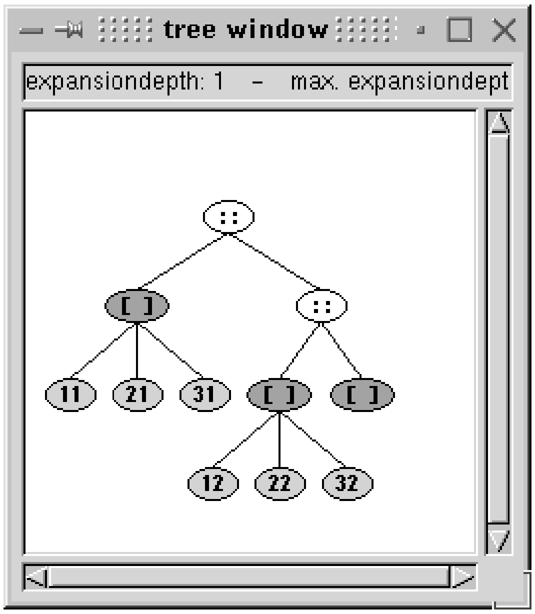


Abbildung 4.39: Nach der dritten Expansion von *transpose*

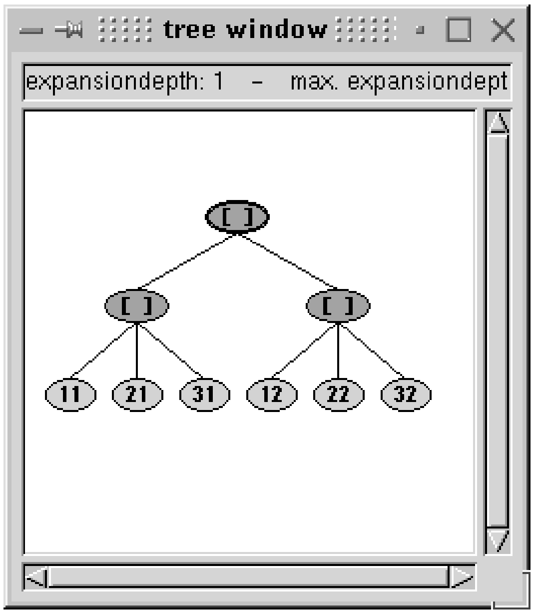


Abbildung 4.40: Die Transponierte

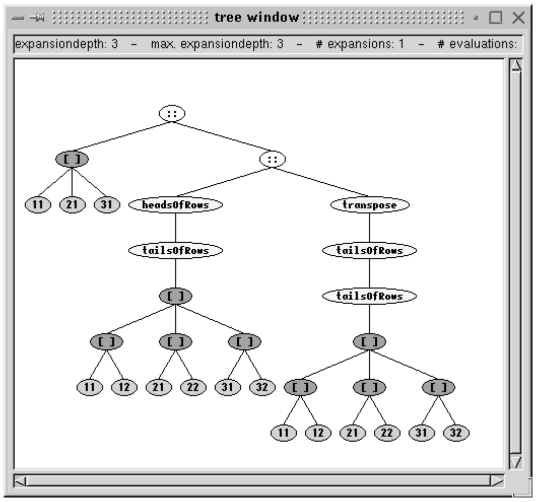


Abbildung 4.41: Erster abweichender Schritt nach $leftmost - outermost$

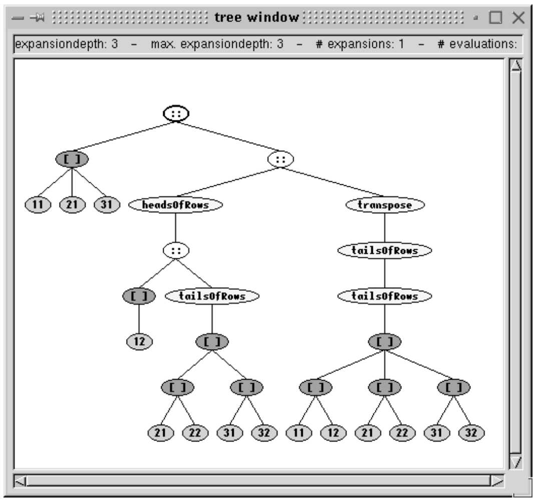


Abbildung 4.42: Nach einem weiteren Auswertungsschritt

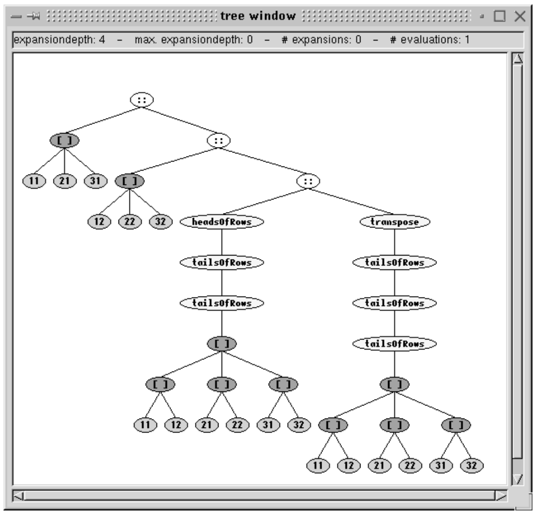


Abbildung 4.43: Nach der erneuten Expansion von *transpose*

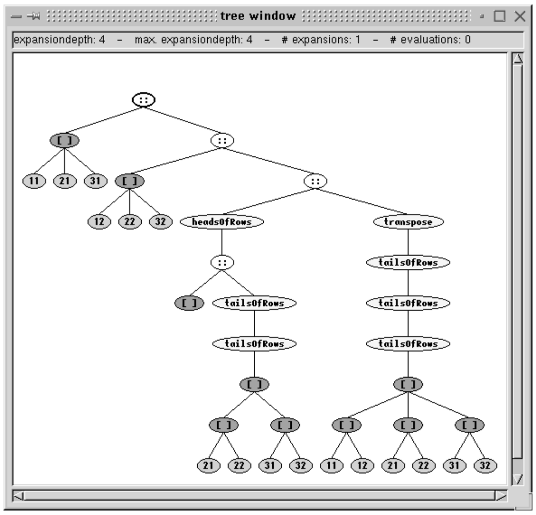


Abbildung 4.44: Es wird zu einem Laufzeitfehler kommen

Kapitel 5

Einige Implementierungsdetails

Dieses Kapitel bietet einen Überblick über die Implementierung des KIEL-Systems. Mit besonderem Augenmerk auf die zukünftige Weiterentwicklung werden die einzelnen Komponenten des Systems betrachtet und ihre Besonderheiten erläutert. Anschließend folgen Anmerkungen, die die Programmierung des gesammten Projekts betreffen. Dazu zählen Erläuterungen zu den bei der Übersetzung in ein Lauffähiges Programm verwendeten Tools und einige Aspekte der Fehlersuche im C-Programmtext. Insgesamt kann es als Einstieg in die Programmierung des KIEL-Systems genutzt werden.

5.1 Komponenten

Bei einem Programmierprojekt in der Größenordnung des KIEL-Systems muss auch der Programmtext stärker strukturiert werden. Selbst der Programmtext für Teilaufgaben muss oft schon auf mehrere Quelldateien aufgeteilt werden, um die Übersicht nicht zu verlieren. Aus diesem Grund sind zusammengehörige Teilaufgaben des KIEL-Systems zu Komponenten zusammengefasst und deren Quelldateien in einem gemeinsamen Unterverzeichnis untergebracht.

Das KIEL-System ist aufgeteilt in zwei große Bereiche. Der Interpreter übernimmt das Einlesen und Überprüfen von Programmen und das Einlesen, Überprüfen und Auswerten von Ausdrücken. Dieser Bereich ist jedoch noch feiner unterteilt in die Komponenten Syntaxbaum, Typüberprüfung und Auswertung. Den zweiten Bereich stellt die graphische Benutzeroberfläche dar. Da diese von der Erweiterung des KIEL-Systems nur wenig betroffen ist, wird auf sie nicht so ausführlich eingegangen, wie auf den Interpreter. Des Weiteren ist sie in dieser Arbeit nicht weiter unterteilt, sondern besteht nur aus einer Komponente.

In den folgenden Unterabschnitten werden die einzelnen Komponenten von KIEL erläutert.

5.1.1 Hauptprogramm

Diese Komponente besteht nur aus einer einzigen Quelldatei, die das Hauptprogramm darstellt und aus zwei Konfigurationsdateien. Hier werden nötige Initialisierungen vorgenommen und die graphische Oberfläche gestartet.

Die einzigen Dateien heißen:

- `kiel_main.c`
- `kiel.rc`
- `kiel.rc.gtk`

5.1.2 Syntaxbaum

Der Syntaxbaum ist die zentrale Struktur, in der die Programme und Ausdrücke, die mit KIEL verarbeitet werden sollen, nach dem Einlesen abgespeichert werden. Diese Umwandlung von einem ASCII-Programmtext in einen Syntaxbaum ist notwendig, da die lexikalische und die syntaktische Analyse recht aufwendig sind und deshalb nur einmal durchgeführt werden sollen. Diese Analysen extrahieren die syntaktischen Einheiten einer Programmiersprache aus dem ASCII-Text heraus und erkennen dadurch auch, ob das Programm oder der Ausdruck der in Anhang A beschriebenen Grammatik genügt. Während der syntaktischen Analyse wird der Syntaxbaum mit aufgebaut und speichert die erkannten syntaktischen Einheiten so, dass auf sie bei der späteren Weiterverarbeitung direkt zugegriffen werden kann. Zusätzlich werden an jedem Knoten des Baumes noch weitere Daten gespeichert, die für die weitere Bearbeitung benötigt werden.

Komponentenbeschreibung

Diese Komponente beinhaltet alle Funktionen zum Aufbau, zur Freigabe, zum Kopieren von Syntaxbäumen und zum Ersetzen von Teilbäumen innerhalb eines Baumes durch einen anderen. Dazu kommen noch einige Hilfsfunktionen und solche um einen Baum wieder als Zeichenkette auszugeben[1].

Des Weiteren enthält die Komponente die Quelldateien für die lexikalische und die syntaktische Analyse. Sie werden mit Hilfe der bekannten Tools *lex & yacc* [9] erzeugt. Obwohl eine lexikalische und eine syntaktische Analyse sowohl für Programme als auch für Ausdrücke erzeugt werden muss, werden sie doch aus einer Quelldatei erzeugt, da die Unterschiede nur minimal sind. Der nicht interaktive Editor *sed* [10] ermöglicht es, aus den beiden Quelldateien für Programme die für Ausdrücke automatisiert zu erzeugen.

Auch in die syntaktische Analyse integriert ist die Überprüfung, ob verwendete Bezeichner vor ihrer Verwendung deklariert wurden oder ob sie eventuell mehrfach deklariert wurden. Dazu wird eine Symboltabelle aufgebaut, die alle auftretenden Bezeichner

[1] dann allerdings vollständig geklammert

verwaltet. Die in dieser Tabelle gespeicherten Daten werden auch für die anschließende Typüberprüfung benötigt. Auch die dafür nötigen Funktionen gehören zu dieser Komponente, da die Symbolüberprüfung, wie schon erwähnt, in die syntaktische Analyse integriert ist.

Schließlich existieren noch zwei Funktionen, die die gesamte Umwandlung einer Programmdatei bzw. einer Ausdruckszeichenkette[2] mit allen Analysen und Überprüfungen in einen Syntaxbaum zusammenfassen.

Die entsprechenden Quelldateien heißen:

- `syntaxtree.[ch]`
- `prg_scanner.l`
- `prg_parser.y`
- `prg.h`
- `expr.h`
- `symboltable.[ch]`
- `st_io.[ch]`
- `yylval_types.h`

Veränderungen

Die wesentlichste Veränderung am Syntaxbaum betrifft die Speicherung der Nachfolgerknoten eines Knotens. Im ursprünglichen KIEL-System wurde eine Liste von Nachfolgern gespeichert. Das bedeutet zwar, dass beliebig viele Nachfolger gespeichert werden können, aber beim Zugriff auf diese muss erst mühsam eine Liste durchlaufen werden. Da fast alle Knoten im Syntaxbaum aber mit drei Nachfolgern auskommen, wurde die Nachfolgerliste durch ein Array für drei Nachfolger ersetzt. Dadurch wird beim Zugriff auf den ersten Nachfolger eine, auf den zweiten zwei und auf den dritten drei Indirektionsstufen eingespart. Für Parameterlisten, Argumentlisten und für Werte von einem beliebigen Listentyp werden aus den Syntaxbaumknoten selbst Listen aufgebaut. In diesen Fällen bleiben die Indirektionsstufen wie vorher und es muss beim Traversieren eines Syntaxbaumes keine zusätzliche Struktur für eine Nachfolgerliste berücksichtigt werden. Der Erfolg dieser Maßnahme kann in der Zusammenfassung (6.1.2) nachgelesen werden.

Eine weitere Änderung betrifft die Grammatik der von KIEL akzeptierten Sprache. Im ursprünglichen KIEL-System sind sowohl die Typüberprüfung für Basisoperatoren

[2] Innerhalb dieser Komponente können Ausdrücke nicht aus einer Datei geladen werden. Das muss die Bedienoberfläche erledigen, indem die Datei vorher in einer Zeichenkette gespeichert wird.

und -funktionen, als auch die Priorisierung und die Assoziativität von Basisoperatoren direkt in der Grammatik kodiert.

Dadurch wird die Grammatik einerseits unübersichtlich und andererseits wird ein Typfehler bei einem Basisoperator oder einer Basisfunktion dann schon von der syntaktischen Analyse als Syntaxfehler bemängelt. Dem Benutzer hilft es aber mehr, wenn ein solcher Fehler auch als Typfehler, mit der Angabe welcher Typ vorhanden ist und welcher erwartet wird, gemeldet wird. Des Weiteren muss immer noch eine Typüberprüfung für die Aufrufe von benutzerdefinierten Rechenvorschriften erfolgen.

Die Priorisierung und die Assoziativität von Basisoperatoren wird auch im erweiterten KIEL von der syntaktischen Analyse durchgeführt. Die Tools *lex & yacc* ermöglichen jedoch eine explizite Angabe der Priorisierung und der Assoziativität von Operatoren in den zugehörigen Quelldateien, so dass eine implizite Kodierung in der Grammatik entfallen kann. Dadurch wird die Grammatik wesentlich überschaubarer und damit verständlicher.

Im erweiterten KIEL-System wird dementsprechend eine Grammatik verwendet, die auf eine teilweise Typüberprüfung verzichtet und die Priorisierung und die Assoziativität nicht direkt kodiert (siehe Anhang A).

5.1.3 Typüberprüfung

Komponentenbeschreibung

Anschließend an die lexikalische und die syntaktische Analyse mit der integrierten Symbolüberprüfung erfolgt die Typüberprüfung. Dabei wird verifiziert, dass die Operanden der Basisoperatoren und die Argumente der Basisfunktionen, Konstruktoren und Rechenvorschriften vom erwarteten Typ sind.

Auch diese Funktionen sind in einer Komponente zusammengefasst und sind durch folgende Quelldateien implementiert:

- `typechecker.[ch]`
- `tc_body.[ch]`
- `tc_util.[ch]`

Veränderungen

Aufgrund der Erweiterung des KIEL-Systems wird auch die Typüberprüfung grundlegend verändert. Es existieren nicht mehr genau 10 feste Typen, sondern durch die Deklaration von Datentypen und die Verwendung geschachtelter Listen beliebig viele.

Des Weiteren bestehen Parameter von Rechenvorschriften nicht mehr nur ausschließlich aus Variablen, deren Typ durch die explizite Typisierung bekannt ist, sondern sie können aus komplex aufgebauten Mustern bestehen. Diese Muster dürfen außerdem mehrere Variablen enthalten und besteht das Muster nicht ausschließlich aus einer Variable, so ist ihr Typ noch nicht bekannt und muss erst gefolgert werden.

Die Erweiterung von KIEL um geschachtelte Listen hat die Überprüfung aller Listen erschwert. Besonders die Konstante *nil* ist dabei von Bedeutung. Im Gegensatz zum ursprünglichen KIEL-System kann sie im erweiterten KIEL auch innerhalb von Listen und insbesondere auch innerhalb geschachtelter Listen auftreten. Ohne den umgebenden Kontext ist der genaue Typ von *nil* unbekannt. Es steht nur fest, dass es sich um einen Listentyp handelt. Aber ob es sich um eine *int list* oder eine *real list* oder gar eine geschachtelte Liste handelt, kann nicht ohne den Kontext ermittelt werden.

Um zumindest diese Teilinformation verarbeiten zu können, werden für die Typüberprüfung intern weitere Typen eingeführt, die in dieser Arbeit als neutrale Listentypen bezeichnet werden. Die Bezeichnungen dieser neutralen Listentypen lauten *list*, *list list*, *list list list*, usw. Wie bei normalen Listentypen die Schachtelungstiefe, wird durch die Häufigkeit des Auftretens des Wortes *list* in der Bezeichnung des neutralen Listentyps die minimale Schachtelungstiefe dieses Typs festgelegt. Der neutrale Listentyp *list list* steht also für einen mindestens einfach geschachtelten Listentyp , wie z.B *int list list* oder auch *real list list list*. Er steht aber nicht für den 0-fach geschachtelten Listentyp *int list*. Wie die Typüberprüfung von Listen konkret erfolgt, wird für Parameterlisten und Rümpfe getrennt beschrieben.

Die Typüberprüfung für Rechenvorschriften kann zweigeteilt werden. Zuerst muss die Parameterliste überprüft werden, um deren Typkorrektheit zu prüfen und die Typen der auftretenden Variablen zu bestimmen. Anschließend kann der Rumpf auf Typkorrektheit überprüft werden.

Die Überprüfung der Parameterliste erfolgt Muster für Muster. Da alle Variablen, die in einem Muster verwendet werden, auch Blätter des Baumes sind und deren Typ noch unbekannt ist, kann die Überprüfung nicht von den Blättern zur Wurzel hin erfolgen. An der Wurzel hingegen ist durch die explizite Typisierung der Typ gegeben, so dass die Überprüfung von der Wurzel beginnend zu den Blättern hin erfolgt. Von einem Knoten ausgehend wird dann für die Nachfolgerknoten festgelegt, welchen Typ sie haben müssen, damit die Typüberprüfung nicht fehlschlägt. Ein Typ m passt zu einem vorgegebenen Typ n, der kein neutraler Listentyp sein darf, wenn

- beide Typen gleich sind oder
- n ein k_n-fach gschachtelter Listentyp ist, m ein neutraler Listentyp mit minimaler Schachtelung von k_m ist und $k_m \leq k_n$ gilt.

Ein Beispiel:
Sei *int list list list* der erwartete Typ der Liste $[nil, [nil], [[1]]]$, dann müssen alle Elemente den Typ *int list list* haben. Das erste Element der Liste ist *nil* vom Typ *list* und

passt nach den obigen Regeln zu Typ *int list list*. Das zweite Element ist wiederum eine Liste, deren Elemente dann vom Typ *int list* sein müssen. Deren einziges Element ist *nil* vom Typ *list* und passt zum Typ *int list*. Das dritte Element ist wieder eine Liste, deren Elemente dann vom Typ *int list* sein müssen. Deren einziges Element ist wiederum eine Liste, deren Elemente vom Typ *int* sein müssen. Dieses Element ist die Konstanten 1 vom Typ *int*, so dass alles zusammen passt.

Die Überprüfung der Rümpfe schließt sich danach an. Da sich Muster und Rümpfe ähneln, liegt es nah, auch die Prüfung der Rümpfe von der Wurzel zu den Blättern durchzuführen, um möglichst viel Programmtext von der Parameterprüfung übernehmen zu können. Die Wurzel eines Rumpfes ist explizit typisiert, so dass diese Vorgehensweise möglich scheint. Sie scheitert jedoch an Basisoperatoren, die auf Operanden verschiedenen Typs arbeiten können , wie die Addition (+) oder die Gleichheit (=). Die Addition kann vom Typ $int \times int \rightarrow int$ oder $real \times real \rightarrow real$ sein. D.h., wenn der aktuelle Knoten eine Addition ist und von Typ *real* sein soll, dann müssen auch die Nachfolgerknoten von diesem Typ sein. Bei der Gleichheit ist das Problem jedoch nicht so einfach zu lösen, da das Ergebnis der Operation immer vom Typ *bool* ist, auch dann, wenn die Operanden von einem anderen Typ sind. Somit kann auf die bisher beschriebene Weise nicht bestimmt werden, von welchem Typ die Nachfolgerknoten sein müssen.

Der Gleichheitsoperator legt also eine Vorgehensweise von den Blättern zur Wurzel nahe, da das beschriebene Problem dann nicht auftritt. D.h., ausgehend von einem Knoten werden erst die Typen der Nachfolgerknoten ermittelt und dann geprüft, ob sie von einem zulässigen Typ sind und zueinander passen. Bei Listen gilt die Besonderheit, dass die neutralen Listentypen nicht die vollständige Typinformation tragen. Daher muss zusätzlich ermittelt werden, welcher von zwei Typen die konkreteren Informationen trägt. Dann wird an den Vorgängerknoten der eigene Typ zurückgegeben.

Es bleibt festzulegen, wann zwei Typen zueinander passen und wenn sie zueinander passen, welcher von beiden der konkretere Typ ist. Zu zwei Typen m und n ist der konkretere Typ

- m, falls beide Typen gleich sind,
- m falls m ein k_m-fach gschachtelter Listentyp ist, n ein neutraler Listentyp mit minimaler Schachtelung von k_n ist und $k_n \leq k_m$ gilt,
- n falls n ein k_n-fach gschachtelter Listentyp ist, m ein neutraler Listentyp mit minimaler Schachtelung von k_m ist und $k_m \leq k_n$ gilt,
- m falls beides neutrale Listentypen mit minimaler Schachtelung von k_m bzw. k_n sind, aber $max(k_m, k_n) = k_m$,

- n falls beides neutrale Listentypen mit minimaler Schachtelung von k_m bzw. k_n sind, aber $max(k_m, k_n) = k_n$,
- undefiniert sonst.

Ein Beispiel:
Betrachtet wird die Liste $[nil, [nil], [[1]]]$. Zuerst werden die Typen der ersten beiden Elemente berechnet. Das erste Element besteht aus der Konstante *nil* vom Typ *list*. Das zweite Element besteht aus einer Liste mit dem einzigen Element *nil* vom Typ *list*. Folglich hat die Liste $[nil]$ den Typ *list list*. Die Typen des Ersten (*list*) und des zweiten Elementes (*list list*) passen also zusammen und der konkretere neutrale Listentyp ist *list list*, da er eine höhere minimale Schachtelungstiefe aufweist. Der Typ des dritten Elementes ist offensichtlich *int list list*. Er passt mit dem eben berechneten Typ *list list* zusammen, wobei *int list list* jedoch der konkretere Listentyp ist. Folglich ist die gesamte Liste vom Typ *int list list list*.

Bei der Typüberprüfung von Parameterlisten scheitert die Vorgehensweise von den Blättern zur Wurzel daran, dass von den Variablen die Typen unbekannt sind. Da diese Typen aber bereits in der Parameterlistenprüfung ermittelt wurden, kann bei der Typprüfung der Rümpfe darauf zurückgegriffen werden. Da nun die Typen aller Blätter bekannt sind, kann die Prüfung auch von den Blättern zur Wurzel hin erfolgen.

Ein Ausdruck, der in KIEL eingegeben wird, enthält keine Variablen. Daher kann für die Typüberprüfung solcher Ausdrücke auf die Funktionen für den Test von Rümpfen zurückgegriffen werden.

5.1.4 Auswertung

Komponentenbeschreibung
Alle Funktionen, die sich auf die Auswertung von Ausdrücken beziehen, sind in einer weiteren Komponente zusammengefasst. Sie enthält Funktionen zur Berechnung von Basisoperatoren und -funktionen, zur Mustererkennung, zum Expandieren von Rechenvorschriftaufrufen und einige Hilfsfunktionen. Die Auswertung eines komplexen Ausdrucks nach einer der drei Auswertungsstrategien basiert auf den eben erwähnten Funktionen und ist auch in dieser Komponente angesiedelt.

Die Namen der entsprechenden Quelldateien lauten:

- `eval_simplify.[ch]`
- `pattern_matcher.[ch]`
- `eval_global.[ch]`

- `eval_expand.[ch]`
- `evaluator.[ch]`
- `ml_limits.[ch]`

Veränderungen
Da im bisherigen KIEL-System keine Rechenvorschriften mit Mustererkennung zugelassen waren, musste die Mustererkennung erst entwickelt werden. Die Implementierung der Mustererkennung ist sehr eng an die Definition aus 3.2.1 angelehnt. Sie ist allerdings nicht identisch, da sie nicht auf Zeichenketten operiert sondern auf einem Syntaxbaum. Im Syntaxbaum ist implizit durch die Baumstruktur eine vollständige Klammerung enthalten, so dass es keine Regeln für die Klammerung gibt.

5.1.5 Graphische Oberfläche

Komponentenbeschreibung
Zu einem großen Teil besteht diese Komponente aus Quelldateien für die Fenster des KIEL-Systems. Zu jedem Fenster gehört eine Quelldatei, die für die Erzeugung, die Freigabe und die Interaktion mit dem Benutzer sorgt.

Da der Interpreter nicht speziell für diese Oberfläche geschrieben ist, sondern sehr allgemein gehalten ist, müssen die Knoten des schon in 5.1.2 erwähnten Syntaxbaumes noch einige zusätzliche Daten speichern können, wie z.B. eine X- und eine Y-Koordinate zur Darstellung auf dem Bildschirm. Hinzu kommen noch einige Daten, die nicht an jedem einzelnen Knoten gespeichert werden müssen, sondern nur einmal pro Ausdruck. Zu den nur einmal pro Ausdruck zu speichernden Daten zählen beispielsweise die Informationen zum Zurücknehmen von im Baumfenster gestarteten Auswertungsaktionen und deren Wiederherstellung. Dazu werden alle Knoten derart erweitert, dass sie die zusätzlichen Daten speichern können und es wird der Syntaxbaum an der Wurzel erweitert, um auch die Daten, die nur einmal pro Ausdruck benötigt werden, aufnehmen zu können. Alle für diese erwähnten Erweiterungen nötigen Funktionen gehören ebenfalls zu dieser Komponente.

Um die Rücknahme und Wiederherstellung von Auswertungsaktionen realisieren zu können, wird vor jeder Auswertungsaktion der aktuelle Ausdruck in den Rücknahmeinformationen abgespeichert. Damit die Rücknahme- und Wiederherstellungsinformationen korrekt verwaltet werden, wird aus dem Baumfenster heraus nicht direkt die entsprechende Auswertungsfunktion aus der Komponente Auswertung aufgerufen. Zwischengeschaltet sind einige Funktionen die diese Verwaltung übernehmen, indem der aktuelle Ausdruck erst kopiert und dann in die Rücknahmeinformationen verschoben wird und die Kopie soweit präpariert wird, dass die Auswertung an ihr erfolgen kann. Bei der

Rücknahme einer Aktion wird dann der aktuelle Ausdruck in die Wiederherstellungsinformation verschoben und der zuletzt in die Rücknahmeinformationen eingetragene Ausdruck dort entfernt und als aktueller Ausdruck gespeichert. Die Wiederherstellung erfolgt umgekehrt. Der aktuelle Ausdruck wird in die Rücknahmeinformation verschoben und der zuletzt in die Wiederherstellungsinformation eingetragene Ausdruck wird dort entfernt und als aktueller Ausdruck gespeichert.

Des Weiteren existieren noch einige Hilfsfunktionen. Die Komponente besteht damit aus folgenden Quelldateien:

- `kiel_gtk_aef.[ch]`
- `kiel_gtk_awf.[ch]`
- `kiel_gtk_bf.[ch]`
- `kiel_gtk_bmf.[ch]`
- `kiel_gtk_df.[ch]`
- `kiel_gtk_uef.[ch]`
- `kiel_gtk_urf.[ch]`
- `kiel_gtk_wf.[ch]`
- `kiel_gtk_zrf.[ch]`
- `drawable_expr.[ch]`
- `expr_node_misc.[ch]`
- `undoredo.[ch]`
- `gtk_eval.[ch]`
- `kiel_fkt.[ch]`
- `kiel_gtk.[ch]`

Veränderungen

Die Veränderungen dieser Komponente beziehen sich hauptsächlich auf die Integration des neuen Interpreters. Die Fenster und die Bedienung blieben unverändert. Nur die Darstellung von Ausdrücken im Baumfenster wurde an die erweiterte von KIEL akzeptierte Sprache angepasst.

Die Darstellung von Konstruktoren als Oval mit rotem Hintergrund ist hinzugefügt worden, wobei auch die beide Konstruktoren :: und [...] zum Aufbau von Listen so dargestellt werden. Außerdem ist die Darstellung von Listen verändert worden. Im bisherigen KIEL-System werden Listen anfangs folgendermaßen dargestellt: Jedes Element der Liste wird durch einen eigenen Knoten repräsentiert. Alle diese Knoten haben den Konstruktor [...], der hier aber noch nicht als Konstruktor gekennzeichnet ist, als Elternknoten. Abbildung 5.1 zeigt eine einfache Liste in dieser Darstellungsform. Handelt es sich bei allen Kindknoten um Konstante, so wird in einem Auswertungsschritt die Darstellung geändert in einen einzigen Knoten, der als Konstante markiert ist und die Liste wie bei der Eingabe einer Liste darstellt. Diese Darstellungsform wird in Abbildung 5.2 veranschaulicht.

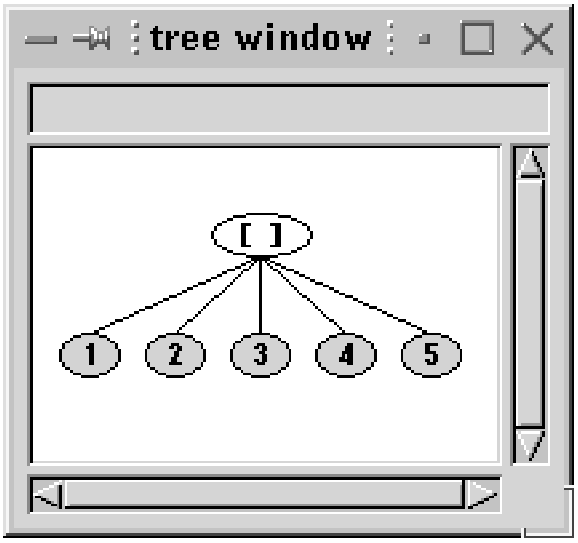


Abbildung 5.1: Die anschaulichere Darstellung von Listen

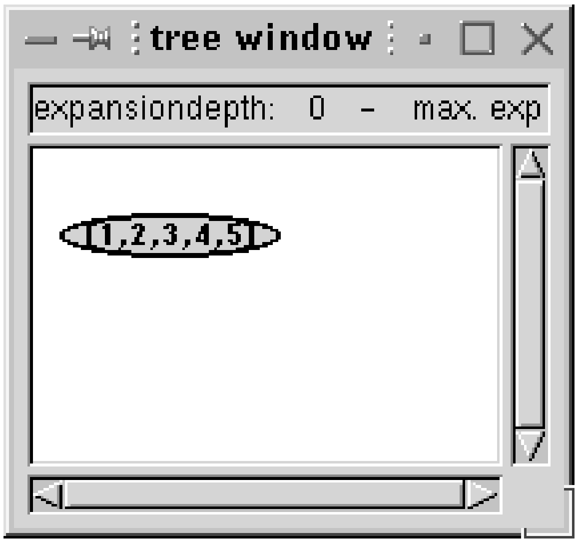


Abbildung 5.2: Die kompakte Darstellung von Listen

Das dies im erweiterten KIEL-System nicht mehr so ist hat drei Gründe:

1. Die Umstellung der Darstellung ist kein echter Auswertungsschritt.

2. Unter Berücksichtigung, dass deklarierte Konstruktoren auch nicht weiter ausgewertet werden können, ist die erste Darstellungsform konsequenter.

3. Die erste Darstellungsform verbessert die Visualisierung der Struktur von geschachtelten Listen.

Daher wird im erweiterten KIEL-System nur noch die erste Darstellungsform verwendet, wobei der Konstruktor [...] jedoch nun als solcher markiert ist.

5.1.6 Sonstiges

Neben einigen Hilfsfunktionen gehört auch eine Schnittstelle zur Weiterleitung von Fehlermeldungen des Interpreters an die Oberfläche zur Komponente. Außerdem enthält sie eine sogenannte garbage-collection, die es ermöglicht Speicherbereiche, von denen nicht sicher ist, ob sie noch benötigt werden, aufzusammeln und später alle gesammelten Bereiche freizugeben, wenn sie garantiert nicht mehr verwendet werden. Schließlich existieren noch Funktionen, um Programmierfehler zu lokalisieren und zur Speicherverwaltung. Auf die beiden letzten Punkte wird in 5.2.2 noch näher eingegangen. Letztlich gehören noch einige Hilfsfunktionen dazu.

Die Komponente besteht aus folgenden Quelldateien:

- `error_interface.[ch]`
- `garbage_collection.[ch]`
- `my_dbug.[ch]`
- `mem_management.[ch]`
- `utilities.[ch]`
- `my_main.h`

5.2 Übersetzung und Fehlersuche

5.2.1 Übersetzung

Um aus den Quelltexten des KIEL-Systems ein lauffähiges Programm zu generieren, müssen alle C-Dateien übersetzt und die erzeugten Objektdateien anschließend gebunden werden. Die Quelldateien für die lexikalische und die syntaktische Analyse müssen vorher mittels der Tools *lex & yacc* [9] in C-Dateien übersetzt werden. Alle diese Quelldateien müssen aber nicht per Hand einzeln übersetzt werden. Das übernimmt das *make*-Tool [10]. Es kann mittels einer Datei, die standardmäßig *makefile* heißt, konfiguriert werden. In diesem *makefile* können einerseits generelle Regeln angegeben werden, die festlegen wie aus einer C-Datei eine Objektdatei erzeugt werden muss. Da viele

Dateien von anderen abhängig sind, können im *makefile* auch diese Abhängigkeiten eingetragen werden, so dass bei der Änderung einer Quelldatei nur sie selbst und die von ihr abhängigen Dateien neu übersetzt werden.

5.2.2 Fehlersuche

Bei der Programmierung eines Projektes in der Größenordnung des KIEL-Systems kommt es unweigerlich zu Programmierfehlern. Dies gilt gerade für die Programmiersprache C, da sie einen sehr freizügigen Umgang mit Zeigern erlaubt und der Programmierer selbst für eine korrekte Speicherverwaltung zu sorgen hat. Um solche Programmierfehler zu beseitigen, muss zuerst einmal der Ursprung des Fehlers ermittelt werden. Dabei hilft ein frei verfügbares Paket von Hilfsfunktionen namens 'dbug' welches von Fred Fisch im Jahre 1987 entwickelt wurde. Dieses Paket wurde in leicht modifizierter Form auch bei der Entwicklung des KIEL-Systems verwendet.

Das 'dbug'-Paket

Ein wesentlicher Vorteil dieses Paketes besteht in der Protokollierung des Aufrufstacks der Funktionen. Wird dieses Hilfsmittel bei allen selbst programmierten Funktionen konsequent eingesetzt, so kann die Ursache eines Programmabsturzes durch die Protokollierung des Aufrufstacks recht genau lokalisiert werden. Der Absturz trat im Rumpf der Funktion auf, die zuletzt auf den Aufrufstack abgelegt wurde. Dadurch hat der Programmierer einen guten Ausgangspunkt für die Fehlersuche erhalten, denn der eigentliche Fehler, durch den es zu diesem Absturz kam, kann immer noch an ganz anderer Stelle zu finden sein. Um dieses Hilfsmittel sinnvoll nutzen zu können, muss in jeder selbst programmierten C-Funktion als erste Anweisung das Makro `DBUG_ENTER(“funktions_bezeichner”)` verwendet werden. Weiterhin muss jede `return`-Anweisung durch das Makro `DBUG_RETURN` ersetzt werden und in jeder Funktion ohne `return`-Anweisung als letzte Anweisung `DBUG_VOID_RETURN` eingefügt werden. Folgende C-Funktion zur Berechnung der Fakultät dient als Beispiel:

```
int fak(int n)
{
  int res;
  DBUG_ENTER("fak");
  if (n = 0) {
    res = 1;
  } else {
    res = n * fak(n - 1);
  }
  DBUG_RETURN(res);
} //fak
```

Eine weitere Hilfe bieten Zusicherungen. Sie können als eine Zuweisung in den Programmtext integriert werden und bestehen im Wesentlichen aus einer boolschen Bedingung. Wird diese Bedingung zur Laufzeit des Programms wahr, so wird die Programmausführung fortgesetzt. Wird sie jedoch falsch, so stoppt die Ausführung mit einer Fehlermeldung, die angibt, in welcher Quelldatei und in welcher Zeile dieser Datei die Zusicherung zu finden ist, die fehlschlug. Diese Zusicherungen sind beispielsweise gut geeignet, um für Zeiger, die als Parameter an eine Funktion übergeben werden, sicher zu stellen, dass sie ungleich dem Nullzeiger sind. Zur Veranschaulichung wird eine C-Funktion zur Berechnung der Fibonacci-Zahlen angegeben, wobei zugesichert wird, dass der Parameter größer oder gleich Null ist:

```
int fib(int n)
{
  int res;
  DBUG_ASSERT(n >= 0);
  if (n <= 1) {
    res = 1;
  } else {
    res = fib(n - 1) + fib(n - 2);
  }
  return res;
} //fib
```

Allen diesen Hilfsmitteln des 'dbug'-Paketes ist gemein, dass sie einen zusätzlichen Aufwand bei der Programmausführung darstellen. Dieser ist während der Entwicklungs- und Testphase auch durchaus akzeptabel, aber in einer endgültigen Version unerwünscht. Daher ist es sehr vorteilhaft, dass alle diese Hilfsmittel bei der Übersetzung des Programmierprojektes per Kommandozeilenparameter aus dem Programmtext entfernt werden können und sich somit nicht mehr auf die Laufzeit des Programms auswirken.

Die Speicherverwaltung

Ein allzu häufig vernachlässigter Fehler ist das Nicht-Freigeben von nicht mehr verwendeten Speicherbereichen. Häufig werden solche sogenannten Speicherlecks nicht einmal bemerkt, da viele Programme nach sehr kurzer Laufzeit wieder terminieren und dabei aller verwendeter Speicher wieder freigegeben wird. Bei einem Programm wie dem KIEL-System ist dies jedoch anders. KIEL könnte lange Zeit, ohne neu gestartet zu werden, vom Anwender benutzt werden, so dass auch nur wenige Speicherlecks unter Umständen sehr viel Speicher nutzlos belegen und damit Reaktionszeiten des Systems auf Benutzereingaben und ähnliches verlangsamen.

Aus diesem Grund wird im KIEL-System ein besonderes Speichermanagement verwendet. Alle Speicheranforderungen werden protokolliert. Dazu muss jede Speicheranforderung anstatt mittels der standardmäßig benutzten Funktion *malloc* aus der C-Standardbibliothek und ihrer Varianten über eine Funktion aus der Komponente Sonstiges namens *mtmalloc* und ihrer Varianten umgeleitet werden. Dabei wird im Rumpf von *mtmalloc* mittels *malloc* der Speicher angefordert und die Anforderung protokolliert. Protokolliert werden Adresse und Größe des angeforderten Speichers. Im erweiterten KIEL werden jedoch zusätzlich Quelldatei und Zeile des Aufrufs von *mtmalloc* und seiner Varianten protokolliert. Soll Speicher freigegeben werden, so darf dies nicht über das im Normalfall verwendete $free$ erfolgen, sondern muss über $mtfree$ erfolgen. Diese Funktion streicht den entsprechenden Speicherbereich aus dem Protokoll und gibt ihn frei.

Wird ein Programm beendet, so können, nachdem alle Speicherbereiche freigegeben sein sollten, alle noch im Protokoll stehenden Speicherbereiche aufgelistet werden. Werden Speicherbereiche aufgelistet, so sind sie entweder nicht mittels $mtfree$ freigegeben worden [3] oder es wurde ein Speicherleck entdeckt. Bei der Ermittlung der Ursache helfen die zusätzlich protokollierten Angaben über die Quelldatei und Zeile der Speicheranforderung. Aus ihnen kann meist schnell geschlossen werden, welche Art von Daten nicht freigegeben wurden, so dass die Suche nach der Ursache stark vereinfacht wird. Allerdings erfordert auch dieses Hilfsmittel eine gewisse Disziplin vom Programmierer, damit auch alle Speicheranforderungen mittels *mtmalloc* oder seiner Varianten erfolgt. Im KIEL-System werden durch diese Hilfsmittel keine Speicherlecks mehr aufgezeigt.

Da diese Protokollierung der Speicheranforderungen aufwendig ist und es dem Endanwender auch nichts nutzt, über Speicherlecks informiert zu werden, wenn er sie nicht beheben kann, soll diese Protokollierung in einer endgültigen Programmversion nicht mehr durchgeführt werden. Aus diesem Grund kann sie mittels des selben Kommandozeilenparameters wie die Hilfsmittel des 'dbug'-Paketes deaktiviert werden.

[3] sondern z.B. mit $free$

Kapitel 6

Zusammenfassung und Ausblick

Zum Abschluss dieser Arbeit wird eine Zusammenfassung klären, ob die in Kapitel 3 gesetzten Ziele vollständig umgesetzt wurden. Des Weiteren wird in einem Ausblick über mögliche Verbesserungen an der graphischen Oberfläche und über eine weitere Vergrößerung der von KIEL akzeptierten Sprache nachgedacht.

6.1 Zusammenfassung

6.1.1 Grundlegende Ziele der Erweiterung

Das grundlegende Ziel dieser Diplomarbeit besteht in der Erweiterung der vom KIEL-System akzeptierten Sprache auf der Grundlage von Standard ML. Vor allem soll es dem Benutzer ermöglicht werden, mit zusammengesetzten Datenstrukturen arbeiten zu können. Um den Umgang mit diesen komfortabel zu gestalten, werden zur Sprache Rechenvorschriften mit Mustererkennung hinzugenommen.

Ein weiterer Punkt bei der Erweiterung um zusammengesetzte Datenstrukturen betrifft Listen. Zumindest müssen die im bisherigen KIEL-System implementierten linearen Listen dahingehend erweitert werden, dass auch solche über zusammengesetzten Datenstrukturen ermöglicht werden. Die Erweiterung geht aber darüber hinaus. Die Einschränkung auf linearen Listen wird aufgehoben, indem eine beliebige Schachtelung von Listen auf der Grundlage von Standard ML erlaubt wird.

Bei all den Erweiterungen der akzeptierten Sprache hat die Bedienoberfläche möglichst unverändert zu bleiben, damit sich Benutzer des bisherigen KIEL-Systems nicht umgewöhnen müssen.

Wie in 3.1.3 erläutert, wurde von den zwei zur Wahl stehenden Varianten zu Erweiterung von KIEL der *datatype*-Deklaration der Vorzug gegeben, da es vor allem rekursive Datenstrukturen erlaubt. Die Implementierung der Datentypen wurde ohne Einschränkung realisiert.

Da sich die Implementierung von Rechenvorschriften mit Mustererkennung für den Zugriff auf die Komponenten der Datentypen als notwendig erwies, wie Abschnitt 3.2 zeigt, kam ein Verzicht auf sie nicht in Frage. Auch deren Implementierung wurde den Vorgaben entsprechend abgeschlossen. Dass auch das Zusammenspiel von Datentypen und Rechenvorschriften mit Mustererkennung wie erwartet funktioniert, zeigen die Beispiele aus Kapitel 4. Nur bei der Auswertung nach der $leftmost - outermost$ Strategie muss beachtet werden, dass nicht die in 4.3 erläuterte Problematik auftritt.

Auch die Erweiterung der Listen auf Listen über Datentypen und geschachtelte Listen wurde erfolgreich durchgeführt. Dabei ist weniger die Auswertung dieser Listen, als vielmehr die Typüberprüfung problematisch, da von der Konstante *nil* ohne den Kontext, in dem sie auftritt, der Typ nicht bestimmt werden kann. Aber auch diese Schwierigkeit konnte, wie in 5.1.3 gezeigt, gelöst werden, so dass die Typüberprüfung einwandfrei arbeitet.

Im Gegensatz zum Interpreter blieb die Oberfläche des KIEL-Systems weitgehend unverändert. Während sich die Bedienung unverändert dem Benutzer präsentiert, hat sich nur die Darstellung von Listen im Baumfenster gewandelt, wie in 5.1.5 beschrieben, und die für Konstruktoren ist hinzugekommen.

6.1.2 Laufzeiten

Nach der Erweiterung des KIEL-Systems stellt sich die Frage, ob sich dadurch die Laufzeiten bei der Ausführung von KIEL insgesamt oder in Teilen verschlechtert haben.

Erste Tests deuteten darauf hin, dass dem so war. Bei längeren Einzelschrittauswertungen kam es dazu, dass der Benutzer merklich auf eine Reaktion des Systems warten musste. Bei diesen frühen Tests wurde aber eine Version von KIEL verwendet, die alle in 5.2.2 beschriebenen Zusatzinformationen zur Vereinfachung der Fehlersuche enthielt. Somit war ein Rückschluss auf eine endgültige Version schwierig. Insbesondere die Protokollierung der Speicheranforderungen machte sich bei speicherintensiven Berechnungen bemerkbar.

Nach der Übersetzung einer endgültigen Version, die die Zusatzinformationen nicht mehr enthält, ändert sich das Bild. Bei der Einzelschrittauswertung bemerkt der Benutzer nun keine Verzögerungen mehr, wie auch beim KIEL-System vor der Erweiterung. So stellt sich die Frage, ob sich die Laufzeit einer kompletten Auswertung eines aufwendig zu berechnenden Ausdrucks durch die Erweiterung verändert hat.

Als Beispiel diente der folgende Programmtext zur Berechnung der Fibonacci-Zahlen:

```
fun fib(n: int):int =
  if n<=1
  then 1
  else fib(n-2) + fib(n-1);
```

Auszuwerten war der Ausdruck $fib(20)$ unter der $leftmost - innermost$ Auswertungsstrategie. Während das alte KIEL handgestoppt 67 Sekunden für die komplette Auswertung benötigte, brauchte das erweiterte KIEL nur noch 19 Sekunden.

Es stellte sich aber heraus, dass nicht nur die Überarbeitung des Interpreters dafür verantwortlich ist. Die im Baumfenster enthaltene Statuszeile wird im alten KIEL bei jeder Auswertung eines Basisoperators und einer Basisfunktion und bei jeder Expansion aktualisiert. Im erweiterten KIEL geschieht dies jedoch weitaus seltener, so dass die deutliche Laufzeitverbesserung erklärt werden kann.

Nach einer vorübergehenden Anpassung des erweiterten KIEL an dieses Verhalten lag die Ausführungszeit für die komplette Auswertung des Ausdrucks $fib(20)$ mit 57 Sekunden zwar deutlich über dem vorherigen Ergebnis, aber immer noch 10 Sekunden unterhalb der des alten KIEL.

Diese Laufzeitverbesserung sind auf die Überarbeitung des Interpreters zurück zu führen und insbesondere auf die in 5.1.2 beschriebene Änderung bei der Speicherung der Nachfolger im Syntaxbaum.

6.2 Ausblick

6.2.1 Erweiterung der akzeptierten Sprache

Da die vom KIEL-System akzeptierte Sprache nur eine Teilmenge von Standard ML ist, existieren umfangreiche Möglichkeiten sie zu erweitern. Eine, die sich auch gut in die bisher akzeptierte Sprache einfügen würde, ist die Hinzunahme lokaler Deklarationen. Eine solche lokale Deklaration hat in Standard ML die Form

$$let\ d\ in\ e\ end;$$

Damit kann das durch d deklarierte Objekt ausschließlich im Ausdruck e verwendet werden. Die gesamte lokale Deklaration ist wiederum ein Ausdruck vom selben Typ wie e und mit dem selben Wert wie e. Daraus geht hervor, dass lokale Deklarationen beliebig geschachtelt werden dürfen, da sie wiederum einen Ausdruck bilden.

Im aktuellen KIEL-System könnten auf diese Weise also Rechenvorschriften und Datentypen lokal deklariert werden.

Hilfsrechenvorschriften verstecken

Der Sinn einer solchen lokalen Deklaration kann beispielsweise darin bestehen, eine Hilfsrechenvorschrift, die nur für eine spezielle Rechenvorschrift geschrieben ist, vor dem Anwender zu verstecken. Dazu soll das folgende Beispielprogramm zum Test, ob eine Zahl perfekt ist, betrachtet werden:

```
fun divides (x: int, y: int): bool =
  if y = 0
  then x = 0
  else x mod y = 0;

fun isp_help(n: int, x: int, y: int): bool =
  if x <= 1
  then y = 0
  else if divides(n, x-1)
       then isp_help(n, x-1, y-x+1)
       else isp_help(n, x-1, y);

fun isperfect(n: int): bool =
  isp_help(n, n, n);
```

Vollkommen unabhängig davon, wie das Perfektsein getestet wird, fällt auf, dass die Rechenvorschrift *isp_help* eine Hilfsrechenvorschrift ist, die speziell für die Rechenvorschrift *isperfect* geschrieben ist. Es besteht der Rumpf von *isperfect* sogar ausschließlich aus einem Aufruf der Hilfsfunktion. Da es dem Benutzer ohne zusätzliches Wissen über *isp_help* kaum möglich ist, sie sinnvoll zu verwenden, ist es aus Sicht des Programmierers ratsam, sie vor ihm zu verstecken. Genau das erlaubt eine lokale Deklaration. In Standard ML kann *isperfekt* folgendermaßen implementiert werden:

```
fun divides (x: int, y: int): bool =
  if y = 0
  then x = 0
  else x mod y = 0;

fun isperfect(n: int): bool =
  let fun isp_help(n: int, x: int, y: int): bool =
        if x <= 1
        then y = 0
        else if divides(n, x-1)
             then isp_help(n, x-1, y-x+1)
             else isp_help(n, x-1, y)
  in isp_help(n, n, n)
  end;
```

Die lokal deklarierte Rechenvorschrift *isp_help* ist für den Benutzer nicht sichtbar und kann auch nicht “blind” aufgerufen werden. Es sei angemerkt, dass auch Systeme von Rechenvorschriften bzw. von Datentypen lokal deklariert werden dürfen.

Lokale Deklarationen in Verbindung mit Objektdeklarationen
Ein Vorteil vollkommen anderer Art kann sich aus der Verbindung von lokalen Deklarationen mit Objektdeklarationen ergeben. Eine Objektdeklaration hat in Standard ML die Form

$$val\ x : m = e;$$

wobei x ein Bezeichner, m ein Typ und e ein Ausdruck vom Typ m ist. Durch diese Deklaration wird der Variable x der Wert von t zugewiesen, so dass im folgenden Programmtext jedes Auftreten von t durch x ersetzt werden kann und umgekehrt.

Der Nutzen einer lokalen Deklaration in Verbindung mit einer solchen Objektdeklaration wird im folgenden Beispielprogramm zur Berechnung des ganzzahligen Anteils der Quadratwurzel erläutert.

```
fun isqrt(n: int): int =
  if n = 0
  then 0
  else if (isqrt(n-1)+1) * (isqrt(n-1)+1) <= n
       then isqrt(n-1)+1
       else isqrt(n-1);
```

Wiederum ohne die Funktionsweise zu berücksichtigen, fällt auf, dass pro expandiertem Rumpf von *isqrt* drei rekursive Aufrufe erzeugt werden, die alle den gleichen Wert berechnen. Bei gegebenem n beträgt die Anzahl der Aufrufe demnach $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$, wodurch diese Art der Berechnung höchst ineffizient wird.

In diesem Fall hilft die lokale Deklaration einer Objektdeklaration, wie sie der folgende Programmtext zeigt:

```
fun isqrt(n: int): int =
  if n = 0
  then 0
  else let val x: int = isqrt(n-1)
       in if (x+1) * (x+1) <= n
          then x+1
          else x
       end;
```

Es fällt sofort auf, dass es nur noch einen rekursiven Aufruf im gesamten Rumpf gibt. Dieser befindet sich in der Objektdeklaration der Variable x. Alle rekursiven Aufrufe aus der vorherigen Fassung sind durch die Variable x ersetzt worden. Das führt zu einer dramatischen Verringerung der Aufrufe. Bei gegebenem n beträgt die Anzahl der Aufrufe nur noch n. Allgemein kann der Aufwand für die Berechnung eines Ausdrucks verringert werden, wenn er aufwendig zu berechnende Teilausdrücke mehrfach enthält.[1]

Aber auch wenn solche Teilausdrücke nicht mehrfach vorkommen, kann eine lokale Deklaration einer Objektdeklaration die Lesbarkeit eines Programmtextes verbessern. Tritt ein Teilausdruck, dem eine gewisse Bedeutung zugeordnet werden kann, innerhalb eines komplexeren Ausdruckes auf, so kann er leicht untergehen, so dass er nicht mehr als der Ausdruck mit der besagten Bedeutung wahrgenommen wird. Wird dieser bedeutungsvolle Teilausdruck jedoch einer Variable, deren Name diese Bedeutung nahe legt, zugewiesen, so unterstreicht die Verwendung der Variable innerhalb des komplexeren Ausdrucks seine Bedeutung.

6.2.2 Benutzeroberfläche

Bei der intensiven Entwicklungsarbeit am KIEL-System sind zwei Punkte aufgefallen, die an der Benutzeroberfläche ergänzt werden könnten.

Freigabe von Speicher

Es existiert keine direkte Möglichkeit, die Auswertung eines Ausdrucks komplett in den Ursprungszustand zurück zu versetzen. Um das Zurücknehmen und das Wiederherstellen zurückgenommener Auswertungsaktionen zu erlauben, werden alle so entstandenen Ausdrücke gespeichert. Bei stark anwachsenden Ausdrücken und langen Einzelschrittauswertungen kann dabei ein großer Anteil des Speichers belegt werden. Doch auch wenn die Bearbeitung eines Ausdrucks vollständig abgeschlossen ist, der Ausdruck also nicht mehr benötigt wird, kann der belegte Speicher nicht direkt freigegeben werden. Dadurch könnte die Bearbeitung eines anderen Ausdrucks, die ebenfalls viel Speicher erfordert, erschwert oder gar verhindert werden. Es gibt zwei Methoden, mit denen der gesamte Speicher bzw. der größte Teil des Speichers freigegeben werden kann. Einerseits können alle Auswertungsaktionen zurückgenommen werden. Da aber weiterhin die Möglichkeit der Wiederherstellung besteht, wird noch kein Speicher freigegeben. Erst wenn nun eine Einzelschrittauswertung vorgenommen wird, muss auch die Wiederherstellung verworfen werden und der Speicher wird größtenteils freigegeben. Andererseits kann das aktuell geladene Programm erneut geladen werden. Dadurch werden auch alle Ausdrücke gelöscht und der gesamte Speicher freigegeben. Daher sollte überlegt werden, ob eine Möglichkeit zum Zurücksetzen eines Ausdrucks in seinen Ursprungszustand nicht eine sinnvolle Ergänzung für das KIEL-System darstellen würde.

Programme erneut laden

Bei der Entwicklung von ML Programmen mit Hilfe des KIEL-Systems kommt es häufig dazu, dass ein Programm von Rechenvorschriften erneut geladen werden muss. Dies

[1] abgesehen davon, dass ein Ausdruck in beiden Alternativen einer Verzweigung vorkommt

kann entweder der Fall sein, weil das Programm beim vorherigen Ladeversuch auf Grund von Syntax- oder Typfehlern nicht übernommen wurde, oder weil das Programm schlicht verändert wurde. Um ein Programm erneut zu laden, muss jedoch jedes Mal der umständliche Weg über den Lade-Button im Bereich Rechenvorschriften des Befehlsfensters eingeschlagen werden. Im erscheinenden Dateidialog muss dann die Datei gesucht und ausgewählt werden. Dieses Verfahren erscheint besonders störend, wenn die Datei nicht in dem Verzeichnis liegt, mit dem der Dateidialog geöffnet wird oder wenn besonders viele Programmdateien im selben Verzeichnis liegen. Ein einfacher 'erneut laden'-Button im Bereich Rechenvorschriften des Befehlsfensters könnte hier Abhilfe schaffen. Er lädt die Datei erneut, die zuletzt mittels des Lade-Buttons geladen wurde bzw. die zuletzt versucht wurde einzuladen.

6.2.3 Visualisierung der Auswertung

Bedingt durch den Algorithmus zum Zeichnen eines Ausdrucks als Baum verändern sich die Bildschirmpositionen der Knoten im Baum, sobald eine Auswertungsaktion vom Benutzer durchgeführt wurde. Für jeden Ausdruck wird die Position seiner Knoten unabhängig vom vorherigen Ausdruck berechnet. Dadurch wird die benötigte Zeichenfläche für den Ausdruck zwar möglichst klein gehalten, aber die Position eines Knotens kann sich gegenüber der des entsprechenden Knotens im vorherigen Ausdruck verändern. Je größer die Änderungen im Ausdruck sind, desto stärker können auch die neuen Positionen von den alten abweichen.

Bei der Einzelschrittauswertung von Basisoperatoren und -funktionen sind diese Positionsänderungen so gering, dass es dem Benutzer nicht schwer fällt, zu einem Knoten den entsprechenden im neuen Ausdruck zu finden. Aber bei der Auswertung einer Verzweigung zu einer der beiden Alternativen oder bei der Expansion eines Rechenvorschriftaufrufs ist das schon schwieriger. Um dem Benutzer die Suche nach einem korrespondierenden Knoten im neuen Ausdruck zu erleichtern, würde es helfen, den Knoten, der den Verzweigungsknoten[2] bzw. den Aufrufknoten[3] ersetzt hat, besonders zu markieren. Dies ließe sich auch auf die Auswertung von Basisoperatoren und -funktionen ausweiten.

Speziell bei der Expansion von Rechenvorschriften wäre es weiterhin interessant, die Knoten nochmals anders zu markieren, die im Rumpf der Rechenvorschrift eine Variable ersetzt haben. Denn bei Rechenvorschriften mit Mustererkennung brauchen die übergebenen Argumente einer Rechenvorschrift nicht mehr im Ganzen wieder im Rumpf aufzutauchen. Wird an die in Beispiel 1 aus Kapitel 4 erläuterte Rechenvorschrift *bintreeheight* als Argument $mkbintree(l, m, r)$ übergeben, wobei l, m, r entsprechende Ausdrücke seien, so tritt dieses Argument so im Rumpf nicht mehr auf, sondern nur l

[2] die Wurzel der entsprechenden Alternative
[3] die Wurzel des Rumpfes (in dem alle Variablen durch entsprechende Argumente ersetzt wurden)

und r einzeln.

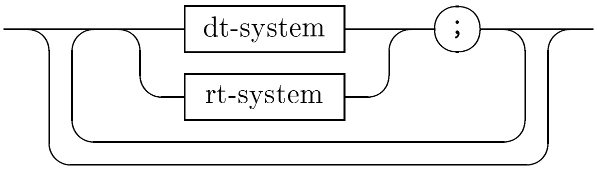
Des Weiteren ist zu Überlegen, ob es nicht auch sinnvoll wäre, den Rumpf einer Rechenvorschrift im KIEL-System anzeigen lassen zu können. Bisher kann der Anwender nur Informationen über den Rumpf einer Rechenvorschrift erhalten, indem er den Aufruf dieser Rechenvorschrift innerhalb eines Ausdruckes expandiert. Doch gerade die Verwendung von Rechenvorschriften mit Mustererkennung erschwert dies. Erstens kann eine solche Rechenvorschrift aus mehreren Fällen mit entsprechend vielen Rümpfen bestehen. Zweitens sind die Parameterlisten einer Rechenvorschrift nicht sichtbar und die an sie übergebenen Argumente brauchen nicht, wie schon erwähnt, in der Form im Rumpf verwendet werden, in der sie übergeben werden.

Anhang A

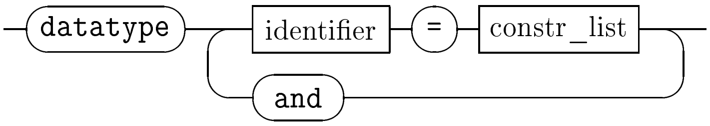
Syntaxdiagramme

A.1 Aufbau einer Programmdatei

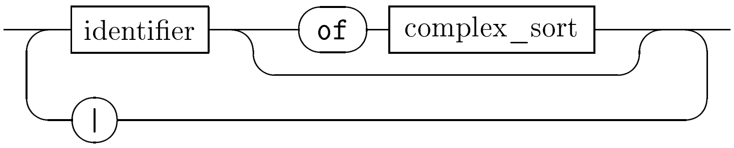
prg-file



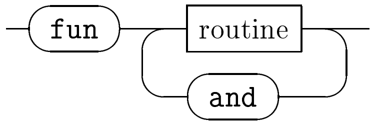
dt-system



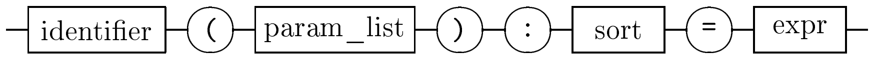
constr_list



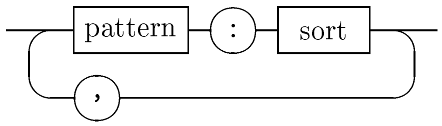
rt_system



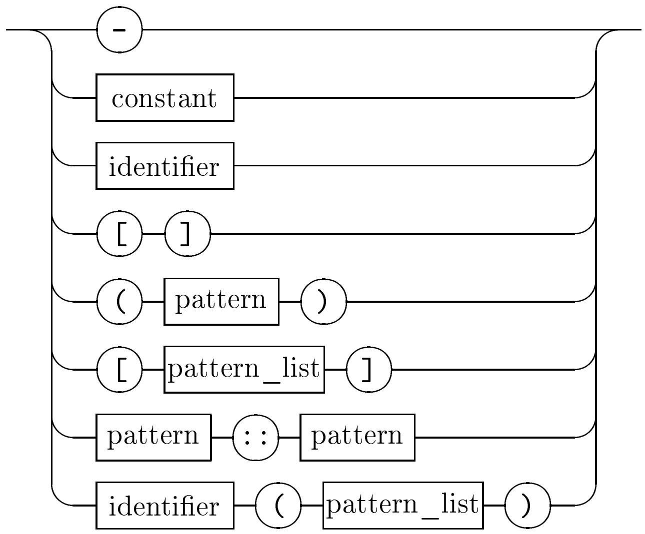
routine



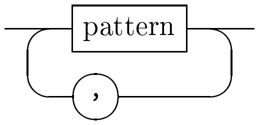
param_list



pattern

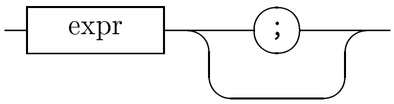


pattern_list

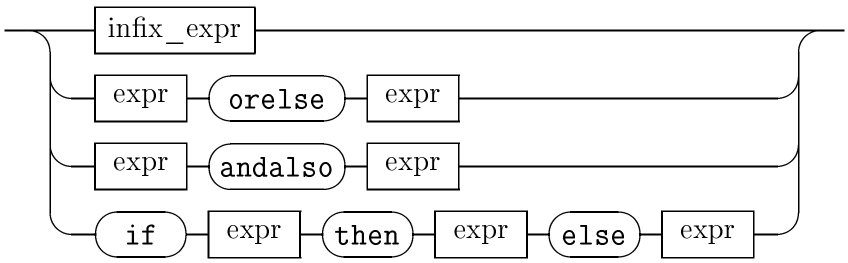


A.2 Aufbau einer Ausdruckdatei

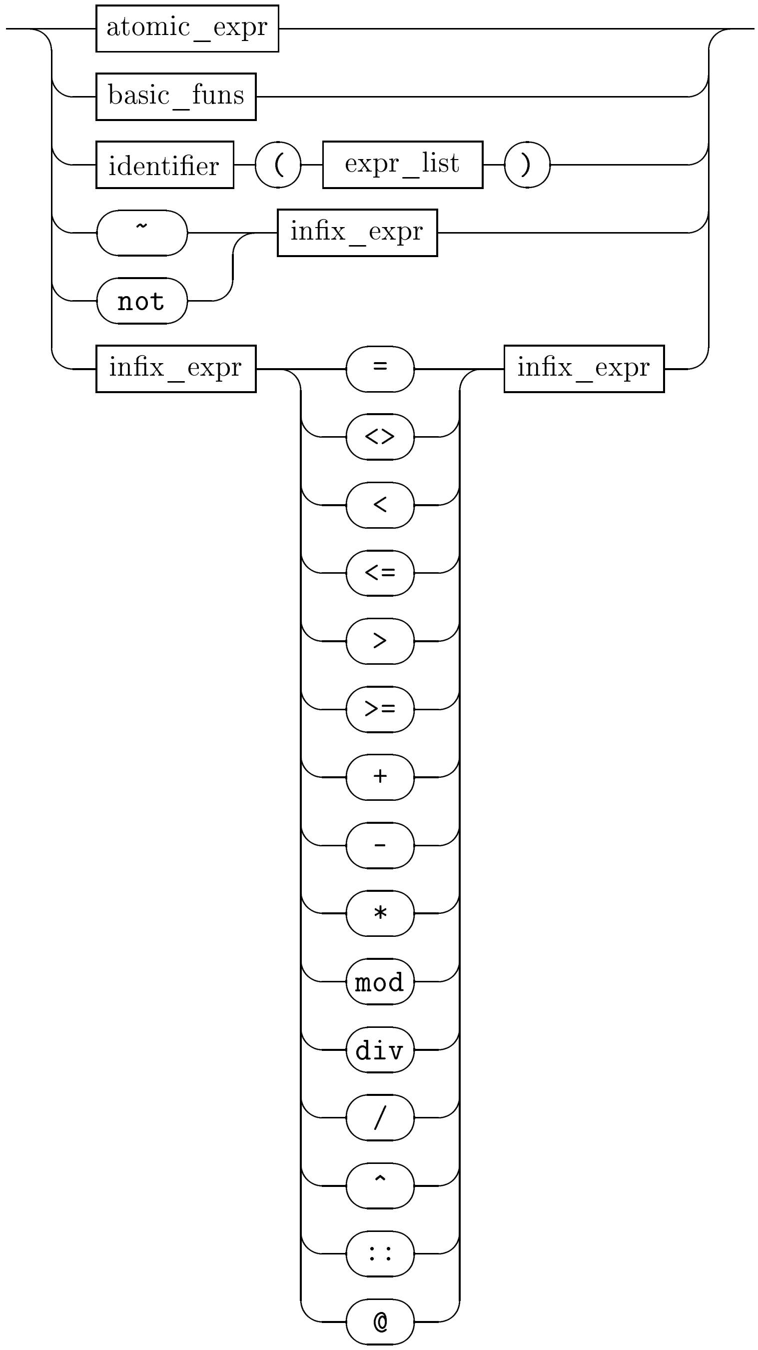
expr-file



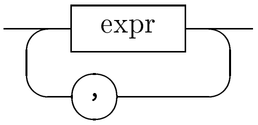
expr



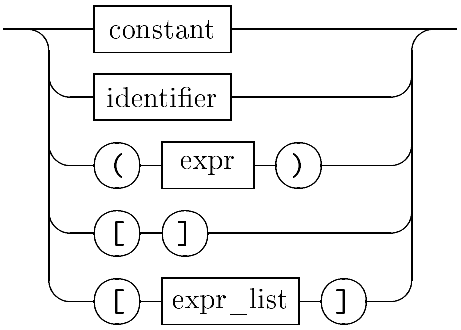
infix_expr


atomic_expr
basic_funs
identifier
(
expr_list
)
~
not
infix_expr
infix_expr
=
<>
<
<=
>
>=
+
-
*
mod
div
/
^
::
@
infix_expr

expr_list

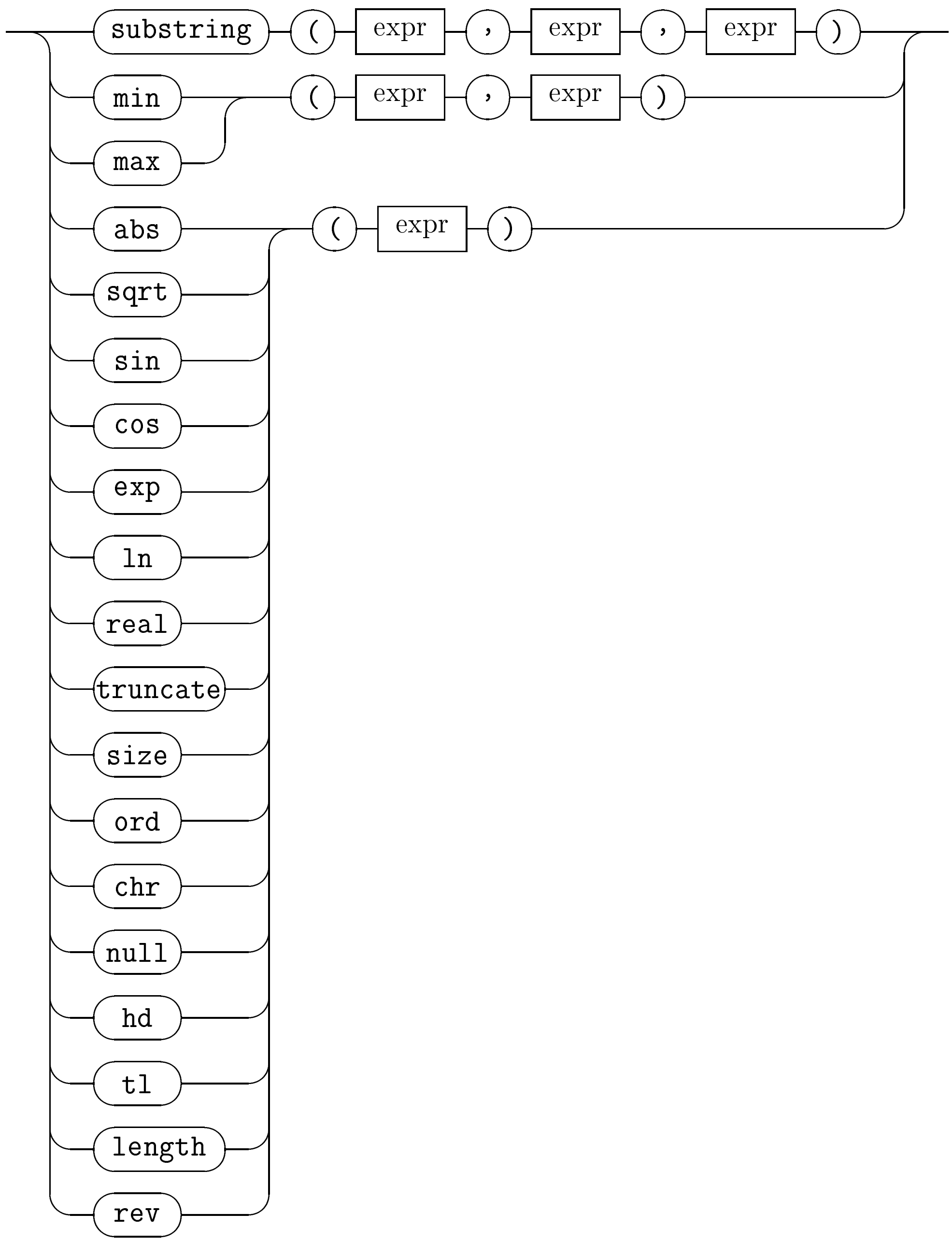


atomic_expr

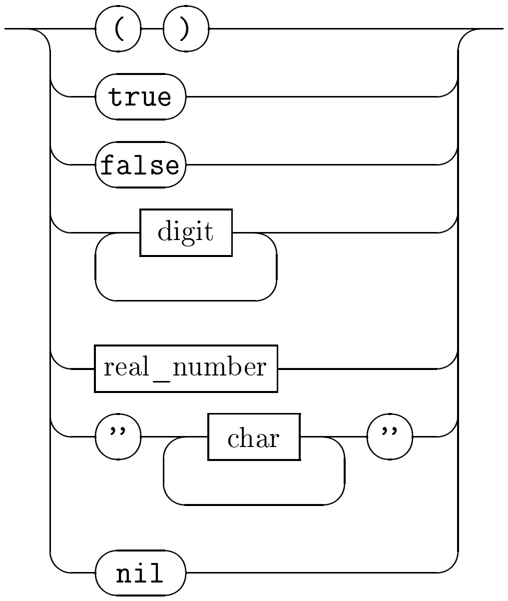


A.3 Allgemeine Syntaxdiagramme

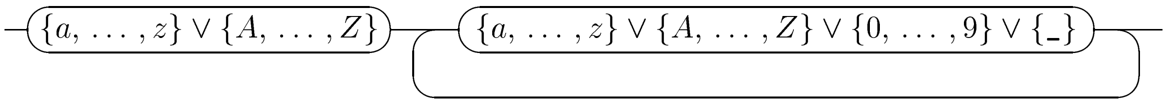
basic_funs



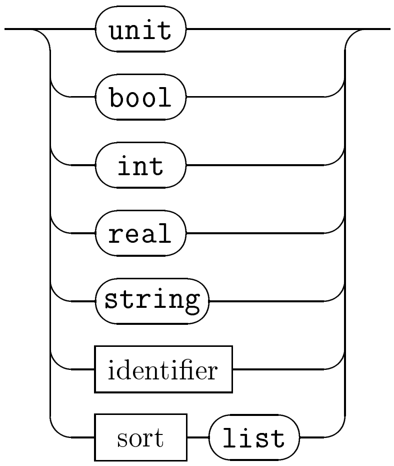
constants



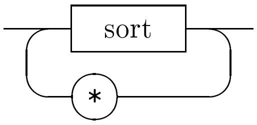
identifier



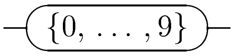
sort



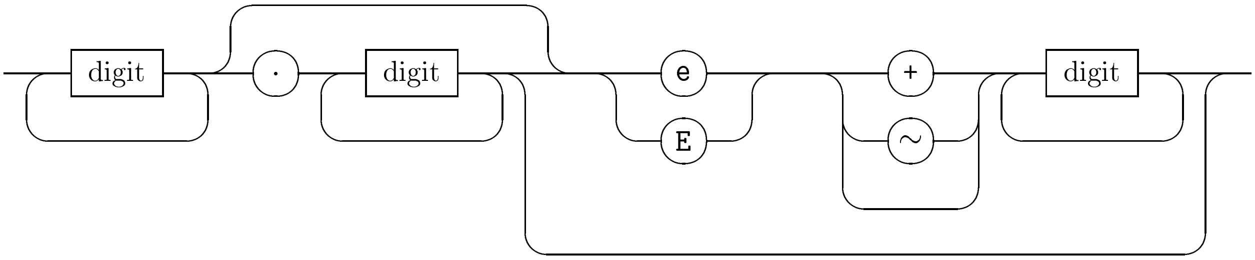
complex_sort



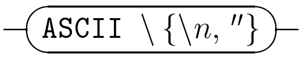
digit



real_number



char



Literaturverzeichnis

[1] M. Thiedt: *KIEL: Ein Werkzeug zur Visualisierung von Termersetzungssemantik*, Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, 1999

[2] R. Berghammer und M. Thiedt: *Kiel Interactive Evaluation Laboratory*, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, Bericht 1909, 1999

[3] O. Lorenz: *Vom C-Programm zum Applet am Beispiel von KIEL*, Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, 2000

[4] R. Berghammer: Vorlesungsskript Informatik I an der Christian-Albrechts-Universität zu Kiel, 1995

[5] J. D. Ullmann: *Elements of ML Programming*, Prentice Hall, 1994

[6] G. Michaelson: *Elementary Standard ML*, UCL Press, 1995

[7] L. C. Paulson: *ML for the working programmer*, 2. Aufl., Cambridge University Press, 1996

[8] Kernighan/Ritchie: *Programmieren in C*, 2. Ausg. ANSI C, Hanser, 1990

[9] J. R. Levine, T. Mason, D. Brown: *lex & yacc*, 2. ed. O'Reilly, 1995

[10] D. Gilly: *UNIX in a Nutshell* Deutsche Ausgabe für System V bis Release 4 und Solaris 2.0, O'Reilly, 1995

[11] Z. Manna: *Mathematical Theory of Computation*, McGraw-Hill, 1974

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst habe und ausschließlich die angegebenen Hilfsmittel und Quellen verwendet habe.

Kiel, den 13.März.2001 (Helge Ernst)