Vom C-Programm zum Applet am Beispiel von KIEL

Diplomarbeit
vorgelegt von
Oliver Lorenz
am 20. April 2000

Betreuer: Prof. Dr. R. Berghammer

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Das Internet hat durch die Hypertext Markup Language (HTML) und die damit verbundene Browser-Technologie seit Anfang der 90er Jahre stark an Einfluß in Bildung, Wirtschaft und Unterhaltung gewonnen. Im Jahre 1995 stellte die Firma Sun Microsystems die Programmiersprache *Java* vor, mit der kleine Programme, sogenannte *Applets*, entwickelt werden können, die dann aus dem WorldWideWeb heruntergeladen und innerhalb eines Browsers ausgeführt werden können. Dadurch können nun nicht nur Textdokumente in Browsern angezeigt werden; das Browserfenster kann mit Hilfe von Java zu einer interaktiven Programmoberfläche gemacht werden. Java hat insbesondere in den ersten 18 Monaten seit Ausgabe der Version 1.0 eine immense Verbreitung erfahren, die immer noch anhält. Mit Hilfe der Java-Technologie lassen sich heute Programme beliebiger Größe realisieren, die dann nicht nur im Internet, sondern auch als eigenständige Applikationen ausgeführt werden können. Bislang haben die meisten im Internet veröffentlichten Applets jedoch eher Spielcharakter, oder sind relativ klein, bestehen beispielsweise nur aus einer einzigen Klasse.

Um der rasanten Entwicklung der Internet-Technologie, insbesondere auf dem Gebiet von Java, Rechnung zu tragen, ist hier nun ein ernsthaftes, größeres Programm als Java-Applet realisiert worden.

Im Rahmen meiner Diplomarbeit habe ich das an der Christian-Albrechts-Universität zu Kiel entwickelte Programm KIEL, ein Werkzeug zur Visualisierung von Termersetzungssemantik, nach Java übersetzt, um es als Applet im WorldWideWeb zur Verfügung zu stellen. So kann das KIEL-System im Internet präsentiert werden, ohne daß der Anwender die C-Applikation vom ftp-Server herunterladen und installieren muß. Außerdem sind Java-Applets plattformunabhängig, und nicht wie die zugrundeliegende C-Applikation auf die Betriebssysteme Unix und Linux beschränkt. Das Applet ist auf jedem System ausführbar, für das Java verfügbar ist (Linux ,Unix, Microsoft-Windows, MacOS, OS-2). Außerdem kann das Applet unverändert auf diesen Plattformen auch als lokale Applikation gestartet werden.

Die C-Applikation KIEL wurde dazu schrittweise in ein Java-Applet übersetzt.

In dieser Diplomarbeit werden anhand des entstandenen Applets allgemeine Konzepte dargestellt, die aus dieser Umsetzung abgeleitet wurden.

Das zweite Kapitel liefert einen Überblick über das existierende KIEL-System und beschreibt seine Funktionalität und Bedienung. Im dritten Kapitel werden, ausgehend vom bestehenden System, die Anforderungen an das Applet formuliert, im darauffolgenden Kapitel wird die gewählte Vorgehensweise erläutert. Kapitel 5 und 6 bilden den Kern dieser Arbeit und schildern das genaue Vorgehen bei der Umsetzung des C-Codes in Java-Code. Dabei werden im 5. Kapitel allgemeine Konzepte für die Transformation einer prozeduralen in eine objektorientierte Programmiersprache behandelt, Kapitel 6 befaßt sich mit konkreten Fragestellungen, die bei der Umwandlung in ein Java-Applet auftreten. Hier werden insbesondere die speziellen Eigenschaften der Programmiersprache Java und die Einschränkungen besprochen, die für Applets gelten. Den Abschluß bildet dann eine Beschreibung des Ergebnisses, KIEL-online, und ein Ausblick auf zukünftige Applet-Entwicklungen.

Kapitel 2

Das KIEL-System

KIEL ist eine Abkürzung für “Kiel Interactive Evaluation Laboratory”. Dieses an der Christian-Albrechts-Universität entwickelte System wertet funktionale Programme erster Ordnung aus. Die auszuwertenden Programme gehören dabei einer einfachen Teilmenge von Standart ML an. Das KIEL-System wurde insbesondere entwickelt, um Studenten der Informatik im Grundstudium als Unterrichtshilfe zu dienen und deren eigene Programme, die in die erwähnte Teilmenge von Standart ML fallen, zu debuggen.

In diesem Kapitel wird nun die Teilmenge von Standart-ML beschrieben, die das KIEL-System unterstützt. Es wird kurz auf die grafische Benutzerschnittstelle eingegangen und das Arbeiten mit KIEL vorgestellt. Bezüglich einer detaillierten Beschreibung von KIEL sei an dieser Stelle auf [1] verwiesen. Eine knappe Darstellung des Systems findet man in [2].

2.1 Die Programmiersprache

Das KIEL-System unterstützt eine Teilmenge der funktionalen Programmiersprache Standart ML. Im weiteren wird diese spezielle Teilmenge MlKiel genannt. Es wird hier nur eine informale Beschreibung von MlKiel gegeben, eine formale Definition durch Syntaxdiagramme läßt sich im Anhang dieser Diplomarbeit finden. Die Art der Parameterübergabe an Funktionen ist in KIEL frei wählbar (vergleiche Abschnitt 2.2), ansonsten verhalten sich die Sprachkonstrukte genau wie in Standart ML.

2.1.1 Datentypen und Operationen

Es gibt fünf Basisdatentypen in MlKiel, die im folgenden mit ihren Operationen aufgeführt sind.

unit Der einelementige Typ *unit* besitzt nur eine Konstante, nämlich ().

bool Der Typ *bool* implementiert die Wahrheitswerte, somit gibt es die Konstanten *true* und *false*. Operationen auf den Wahrheitswerten sind:

$not\ :\ bool \rightarrow bool$	Negation
$andalso\ :\ bool * bool \rightarrow bool$	sequentielle Konjunktion
$orelse\ :\ bool * bool \rightarrow bool$	sequentielle Disjunktion

int Die Konstanten des Typs für Integerzahlen liegen im Zahlenintervall $[-2^{30}, 2^{30} - 1]$ und werden wie bei Standart ML als Zeichenketten von einer oder mehreren Ziffern und im Fall einer negativen Zahl zusätzlich durch ein führendes "$\sim$" dargestellt. Auf Integerzahlen lassen sich folgende Operationen anwenden:

$+, -, *\ :\ int * int \rightarrow int$	arithmetische Operationen
$div, mod\ :\ int * int \rightarrow int$	ganzzahlige Division und Restfunktion
$min, max\ :\ int * int \rightarrow int$	Minimum und Maximum
$abs\ :\ int \rightarrow int$	absoluter Wert
$<, <=, >, >=\ :\ int * int \rightarrow bool$	Vergleichsoperationen

real Auch die Konstanten für reelle Zahlen werden wie in Standart ML dargestellt: Sie beginnen mit einem optionalen "$\sim$", es folgt eine Integer-Zahl, danach entweder ein Dezimalpunkt und eine weitere Integer-Zahl, oder der Buchstabe E, wiederum ein optionales "$\sim$" und eine Integerzahl. Der Wertebereich liegt ungefähr zwischen $+/-\ 10^{308}$ (IEEE 754-1985 Standart). Operationen für reelle Zahlen in MLKIEL sind:

$+, -, *, /\ :\ real * real \rightarrow real$	arithmetische Operationen
$abs\ :\ real \rightarrow real$	absoluter Wert
$sqrt\ :\ real \rightarrow real$	Wurzelfunktion
$sin, cos\ :\ real \rightarrow real$	Sinus und Cosinus
$exp, ln\ :\ real \rightarrow real$	Exponentialfunktion und Logarithmus
$real\ :\ int \rightarrow real$	Umwandlung in reelle Zahl
$truncate\ :\ real \rightarrow int$	runden
$<, <=, >, >=\ :\ real * real \rightarrow bool$	Vergleichsoperationen

string	Der Datentyp für Zeichenketten ist eine in Anführungszeichen stehende Folge von Buchstaben. Anders als bei Standart-ML gibt es aber keine speziellen Buchstaben wie 'newline' oder Tabulaturzeichen. Die maximale Länge einer Zeichenreihe kann in einer Konfigurationsdatei festgelegt werden. Operationen auf Zeichenketten sind:

$\wedge\ :\ string * string \to string$	Konkatenierung
$size\ :\ string \to int$	Länge
$chr\ :\ int \to string$	zum ASCII-Code gehörender Buchstabe als String
$ord\ :\ string \to int$	ASCII-Code des ersten Buchstabens
$substring\ :\ string * int * int \to string$	Teilwort
$<, <=, >, >= : string * string \to bool$	lexikografische Vergleichsoperationen

Außerdem gibt es für jeden Basisdatentyp m von MlKiel den Test auf Gleichheit $= : m * m \to bool$ und auf Ungleichheit $<> : m * m \to bool$.

In den meisten funktionalen Programmiersprachen gibt es einen Datentyp für Listen. In Standart ML existiert ein polymorpher Typ für Listen, da aber KIEL keinen Polymorphismus für Typen unterstützt, gibt es für jeden der oben erwähnten Basisdatentypen m einen speziellen Listentyp $m\,list$, der aus Elementen aus m besteht. In MlKiel werden die Konstanten für Listen wie in Standart ML dargestellt, d.h. in eckige Klammern eingeschlossen, wobei die einzelnen Elemente durch Kommata getrennt sind. Die leere Liste wird durch das Wort *nil* oder durch die leeren Klammern [] dargestellt. Es werden folgende Listenoperationen unterstützt:

$::\ :\ m * m\,list \to m\,list$	Elemente von links an die Liste anhängen
$hd\ :\ m\,list \to m$	erstes Element der Liste (head)
$tl\ :\ m\,list \to m\,list$	erstes Element wird von der Liste entfernt (tail)
$@ :\ m\,list * m\,list \to m\,list$	Konkatenation
$length\ :\ m\,list \to int$	Länge
$null\ :\ m\,list \to bool$	Test, ob die Liste leer ist
$rev\ :\ m\,list \to m\,list$	Umdrehen der Liste

Genau wie bei Zeichenketten kann die maximale Anzahl von Elementen in einer Liste in KIEL mit Hilfe einer Konfigurationsdatei festgelegt werden.

2.1.2 Rechenvorschriften

Ein Programm im Sinne von MlKiel besteht aus einer Folge von Deklarationen, wobei eine Deklaration entweder eine einzelne Funktion oder ein System von Funktionen ist, abgeschlossen mit einem Semikolon. Die Funktionen können beliebige Rekursionen enthalten. Die Funktionen werden genau wie in Standart ML deklariert, mit der Ausnahme, daß die Funktionen in MlKiel voll typisiert sein müssen. Eine einzelne Funktion F hat dann also allgemein die Form:

$$\texttt{fun}\ F\,(x_1 : m_1, ..., x_k : m_k) : m = E;$$

Dabei sind die formalen Parameter x_i jeweils vom Typ m_i, $(1 \leq i \leq k)$, m der Resultattyp der Funktion und E der Rumpf der Funktion, der aus einem Ausdruck im Sinne von MlKiel besteht, wobei in diesem Ausdruck die Funktion selbst rekursiv aufgerufen werden kann. Es kann im übrigen auch $k = 0$ sein. Ein Programm im Sinne von MlKiel ist beispielsweise:

```
fun fac (x : int) : int = if  x = 0 then 1
                                    else x * fac(x - 1);
fun even (x : int) : bool = if  x < 0 then false
                                      else if x = 0 then true
                                                    else odd(x - 1)
and odd (x : int) : bool = if  x < 0 then false
                                     else if  x = 1 then true
                                                    else even(x - 1);
```

2.1.3 Ausdrücke

Ausdrücke können in MlKiel aus Parametern, Konstanten und Operationen auf den in Kapitel 2.1.1 vorgestellten Datentypen bestehen, außerdem aus den Aufrufen von bereits deklarierten Funktionen und der Verzweigungsanweisung `if` B `then` A_1 `else` A_2. In MlKiel wird nur die klassische mathematische Notation $f(A)$ bzw. $f(A_1, ..., A_k)$ für Funktionsaufrufe unterstützt, wobei aber $k = 0$ sein kann. Ausdrücke auf dem im vorigen Abschnitt angegebenen Programm sind beispielsweise:

$$true,\ even(2),\ \texttt{if}\ fac(2) = fac(3)\ \texttt{then}\ even(fac(2))\ \texttt{else}\ odd(fac(3))$$

2.2 Struktur und Benutzung von KIEL

Das KIEL-System besteht aus mehreren Fenstern. Mit diesen lassen sich Ausdrücken einlesen und bearbeiten. Hier werden die einzelnen Fenster beschrieben, und es wird anhand einer Beispielauswertung die Funktionsweise von KIEL beschrieben.

2.2.1 Bedienoberfläche

Das KIEL-System besteht grob gesagt aus zwei verschiedenen Komponenten: Die Arbeitsfläche enthält Rechenvorschriften, die über ein Dateidialogfenster eingelesen werden können, und Ausdrücke, die ebenso aus einer Datei geladen oder über ein spezielles Fenster eingegeben werden können. Die zweite Komponente beinhaltet alle Fenster, die die Auswertung eines eingegebenen Ausdrucks ermöglichen. Dies ist in erster Linie das Baumfenster, in ihm werden Ausdrücke als Baumstruktur dargestellt und interaktiv ausgewertet. Außerdem gibt es ein Fenster, das eine Undo/Redo-Funktion zur Verfügung stellt, sowie ein Fenster, in dem sich spezielle Einstellungen zur Auswertung machen lassen (leftmoust-innermoust, leftmoust-outermoust oder full–Ersetzung). Darüber hinaus läßt sich hier wählen, wie viele Auswertungsschritte nacheinander ausgeführt werden sollen. Das gesamte System sieht im allgemeinen wie in Abbildung 2.1 dargestellt aus.

2.2.2 Bearbeitung von Ausdrücken

Nachdem eine Rechenvorschrift und ein Ausdruck eingelesen wurden, kann der Ausdruck im Baumfenster dargestellt und bearbeitet werden. Der aktuelle Ausdruck wird als Baum dargestellt, die Knoten werden von Konstanten, vordefinierten Datentypen und Funktionen gebildet. Außerdem gibt es einen Knoten für aufgetretene Fehler. Verschiedene Knotenarten werden grafisch unterschiedlich dargestellt.

Es läßt sich auswählen, ob die Auswertung der Knoten halb- oder vollautomatisch geschehen soll. Darüber hinaus kann man die Auswertungsart (leftmoust-innermoust, leftmoust-outermoust oder full–Ersetzung) und im Falle der halbautomatischen Auswertung die Anzahl der nacheinander auszuführenden Auswertungsschritte festsetzen. Mit dem Mauszeiger lassen sich Knoten im Baumfenster anwählen, danach kann mit den Maustasten der betreffende Knoten bearbeitet werden. Je nachdem, welche Maustaste betätigt wird, wird ein Knoten, der einen Funktionsaufruf enthält, durch den entsprechenden Funktionsrumpf ersetzt oder ein Knoten ausgewertet, sofern er eine Rechenvorschrift beinhaltet. Knoten, die eine `if`-Verzweigung darstellen, lassen sich per Mausklick vereinfachen. So läßt sich ein eingegebener Ausdruck schrittweise bearbeiten, bis er vollständig ausgewertet ist.

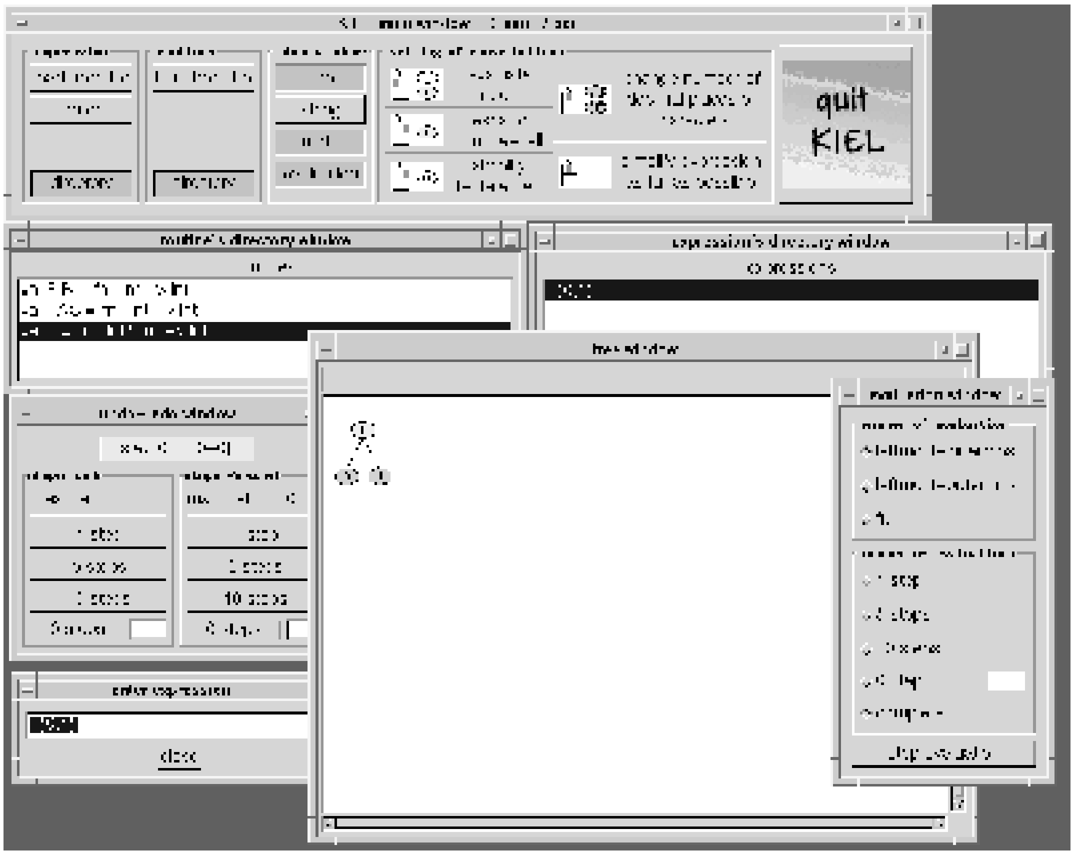


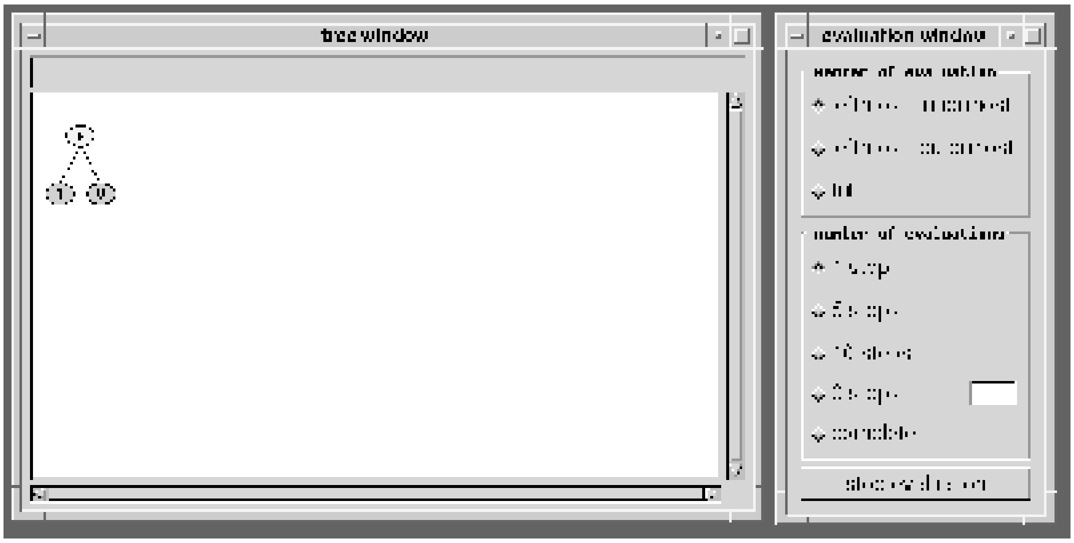
Abbildung 2.1: Das KIEL-System

2.2.3 Beispiel

In diesem Abschnitt soll nun anhand eines Beispiels die Funktion von KIEL dargestellt werden, wobei besonders auf die verschiedenen Auswertungsarten abgezielt wird. Wir betrachten die bekannte Morris-Funktion (vgl. [14]):

$$
\begin{array}{l}
\texttt{fun } F(x : int, y : int) : int = \\
\texttt{if } x <= 0 \texttt{ then } 1 \\
\qquad\qquad \texttt{else } F(x-1, F(x-y, y));
\end{array}
$$

Verschiedene Auswertungsarten führen bei dieser Funktion zu unterschiedlichen Funktionswerten: Der Aufruf von $F(0,1)$ terminiert bei leftmoust-innermoust–Ersetzung nicht, bei leftmoust-outermoust–Ersetzung kommt als Ergebnis 1 heraus. Als erstes soll der Ausdruck $F(0,1)$ mit leftmoust-innermoust–Ersetzung berechnet werden. Nachdem die Rechenvorschrift F und der Ausdruck eingegeben wurden, ergibt sich folgendes Bild:

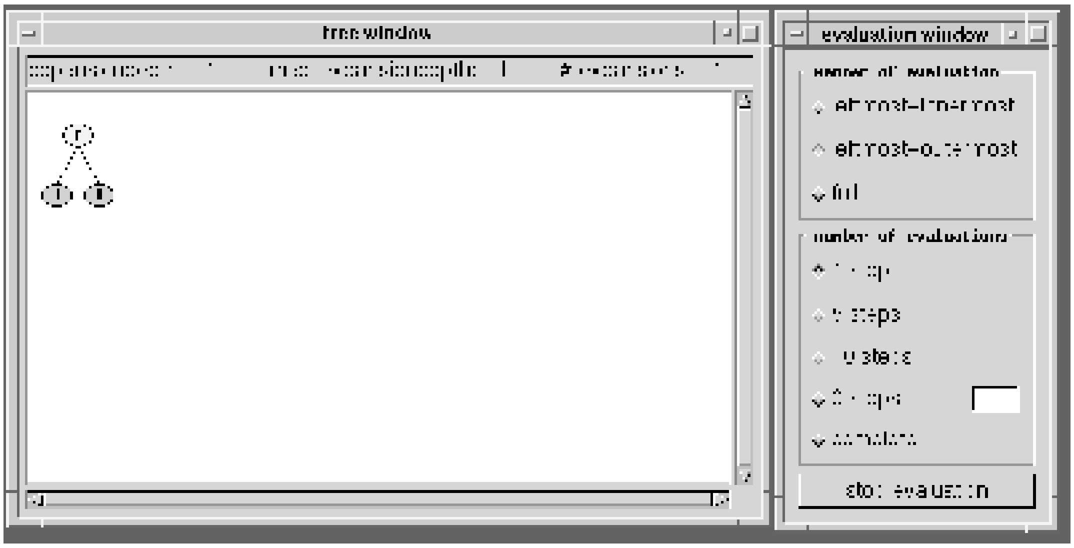


In den ersten beiden Ausführungsschritten wird der Funktionsaufruf $F(0,1)$ ersetzt und der Ausdruck anschliessend soweit wie möglich vereinfacht. Dies führt zu folgendem Ausdruck:

Leftmoust-innermoust–Ersetzung bedeutet grundsätzlich, daß der am weitesten links stehende Aufruf einer Funktion, der frei von weiteren Funktionsaufrufen ist, ersetzt wird. In diesem Fall ist das der Knoten F, der als Nachfolger nur die Knoten 1 und 0 besitzt. Dieser Funktionsaufruf wird nun wiederum durch den Rumpf der Funktion ersetzt, und der gesamte Ausdruck vollständig vereinfacht. Es ergibt sich folgender Zustand:

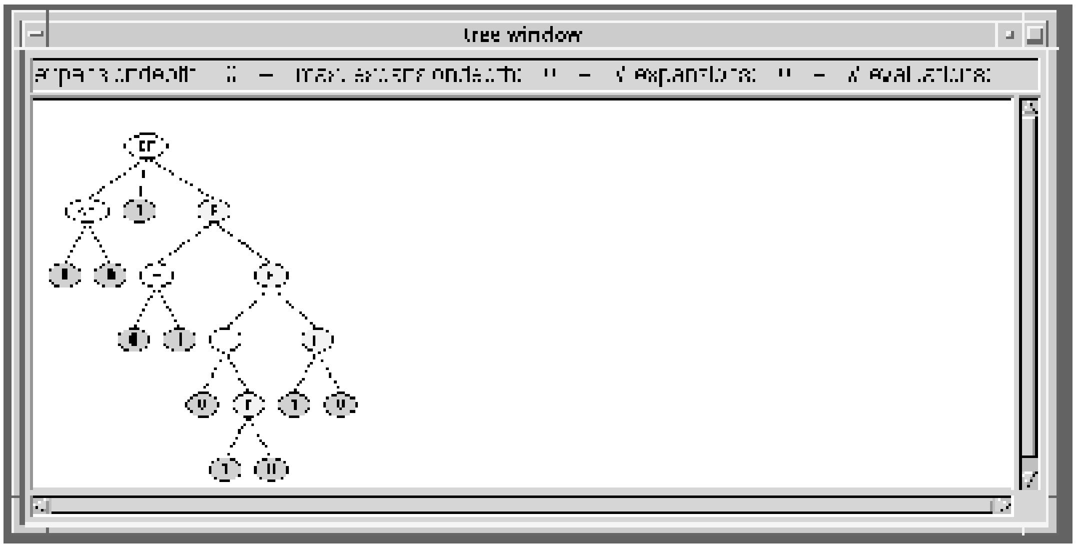
Schon nach diesen wenigen Schritten wird klar, daß die Berechnung von $F(0,1)$ nicht terminiert, das Ergebnis also undefiniert ist.

Jetzt soll die Auswertung mit der leftmoust-outermoust–Ersetzungsstrategie gezeigt werden. Dazu beginnen wir wieder mit dem ursprünglichen Ausdruck:

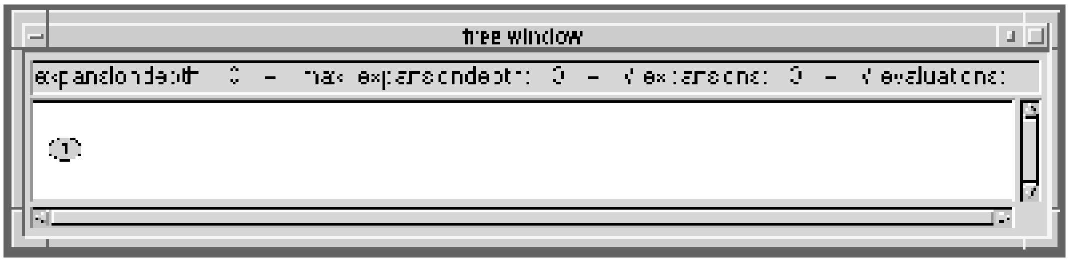


Es gibt hier nur einen Funktionsaufruf, also wird genau wie im ersten Fall der Aufruf durch den betreffenden Rumpf ersetzt und danach der Ausdruck weitestgehend vereinfacht. Wie im ersten Fall ergibt sich also:

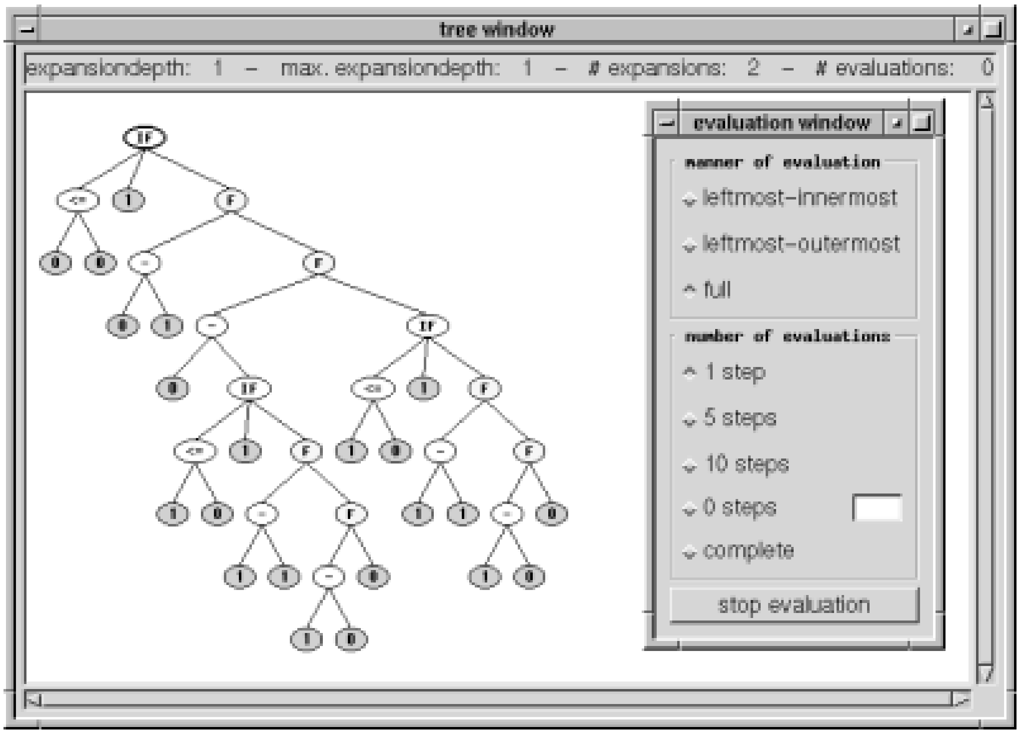
Jetzt ändert sich aber die Situation: Leftmoust-outermoust–Ersetzung bedeutet, daß immer das am weitesten links stehende Vorkommen eines Funktionsaufrufs ersetzt wird, das kein Argument eines weiteren Funktionsaufrufs ist. Es wird also in diesem Fall die Wurzel des Ausdrucksbaums durch den Funktionsrumpf ersetzt, wie hier zu sehen ist:



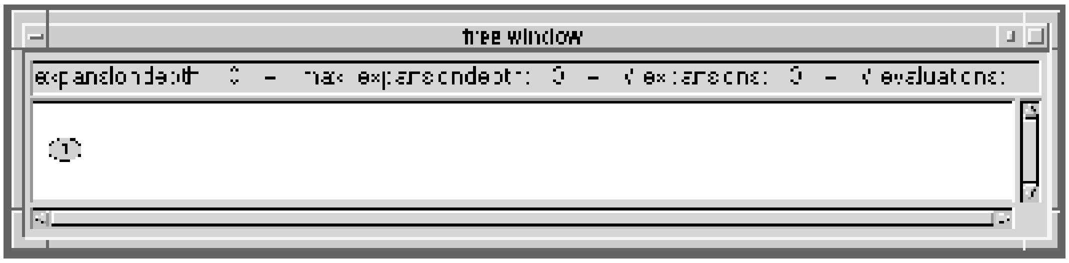
Weil die Semantik der `if`-Verzweigung nicht strikt ist (d.h. es müssen nicht alle Operanden definiert sein), ergibt eine weitere Vereinfachung des Ausdrucks den Wert 1, wie der folgende Bildschirmausschnitt zeigt.



Nun soll noch die dritte mögliche Berechnungsart, die KIEL unterstützt, vorgestellt werden. Dies ist die full–Ersetzung. Hierbei werden in jedem Ersetzungsschritt alle Funktionsaufrufe in dem Ausdruck durch die entsprechenden Rümpfe ersetzt. Wendet man die full–Ersetzung auf den Ausdruck $F(0, F(1, 0))$ an, so ergibt sich folgendes Bild:



Wertet man den hier entstandenen Ausdruck weiter aus, so wird die `if`-Anweisung vereinfacht, man erhält wiederum das Ergebnis 1:



Da immer nur der am weitesten links stehende Teil des Ausdrucks, der frei von weiteren Funktionsaufrufen ist, ausgewertet wird, können die Ausdrücke bei der full–Ersetzung sehr groß werden, so daß beispielsweise die Auswertung des Ausdrucks $F(3,0)$ auf einer modernen SUN ULTRA schon mehr als 20 Minuten Rechenzeit benötigt.

2.3 Konfiguration von KIEL

KIEL ist in weiten Teilen ein System, das vom Benutzer konfigurierbar ist. Neben der bereits erwähnten Konfigurierbarkeit einiger Eigenschaften der Datentypen lassen sich spezielle Grenzen für die Auswertung der Ausdrücke und viele Eigenschaften der Benutzerschnittstelle konfigurieren. Die Konfiguration ist optional und erfolgt über zwei verschiedene Dateien, die nach dem Start des KIEL-Systems eingelesen werden.

In der Datei kiel.rc läßt sich die maximale Anzeigegröße für die in Abschnitt 2.1.1 beschriebenen Typen festlegen. So kann die Länge der Anzeige für Listen und Zeichenketten und die maximal angezeigten Nachkommastellen für den Typ *real* angegeben werden. Diese Werte lassen sich für einzelne Knoten auch während des Programmablaufs über ein weiteres Fenster ändern, jedoch nur bis zu einem bestimmten Wert, der sich wiederum in der Datei kiel.rc setzen läßt.

Es lassen sich außerdem in derselben Datei Grenzen für die Auswertung setzen, und zwar die Expansionstiefe, bei der die Auswertung automatisch abbrechen soll, und die Anzahl der maximal anzuzeigenden Ebenen eines Ausdrucks im Baumfenster.

In der Datei kiel.rc.gtk läßt sich die Benutzerschnittstelle konfigurieren. Es können beispielsweise verschiedene Farben für die unterschiedlichen Knoten gesetzt und Fontgrößen und -arten verändert werden.

Das genaue Format dieser Konfigurationsdateien und alle konfigurierbaren Parameter lassen sich in [1] im Kapitel über die Ressource-Dateien nachlesen. Einen Einstieg bietet auch Kapitel 3 des Manuals für KIEL ([2]).

Kapitel 3

Die Aufgabe: KIEL-online

Die C-Applikation KIEL ist bisher nur auf Unix- und Linux-Plattformen verfügbar, und muß vom ftp-Server des Instituts für Informatik und Praktische Mathematik der Universität Kiel heruntergeladen und danach installiert werden. Um sich von der Plattformabhängigkeit zu lösen, sollte ein Java-Applet entwickelt werden, das aus dem Internet in das Fenster eines Web-Browsers geladen werden kann, und dieselben Funktionalitäten wie das ursprüngliche System besitzt. So müssen Rechenvorschriften eingelesen und schrittweise Ausdrücke ausgewertet werden können.

Als der Entschluß gefaßt wurde, ein Applet zu erstellen, das wie KIEL arbeitet, wurde gleichzeitig festgesetzt, daß es genau dieselben Funktionalitäten wie die Applikation erhalten soll, die gleichen Fenster verwendet werden sollen und das gesamte Erscheinungsbild unverändert bleiben sollte. Einschränkungen durch die Programmiersprache und die spätere Umgebung, nämlich das Browserfenster, waren aber natürlich auch zu berücksichtigen. Ich habe deshalb den Entschluß gefaßt, die genauen Funktionalitäten des Applets beizubehalten, sofern sie nicht durch diese Einschränkungen betroffen sind.

Im diesem Kapitel wird das Anforderungsprofil an das Java-Applet erarbeitet und gegebenenfalls Einschränkungen der ursprünglichen Funktionalitäten diskutiert. Wie im Fall KIEL-online mit diesen Einschränkungen konkret umgegangen wurde, wird dann in Kapitel 7 besprochen.

3.1 Java-Version

Im Januar 1996 wurde die Java-Version 1.0 von Sun vorgestellt. In dieser ersten Version ist die Standart-Bibliothek allerdings noch nicht sehr ausgereift, das Eventmodell (also die Reaktion auf Ereignisse, die der Anwender auslöst) der Version 1.0 ist noch sehr unkomfortabel und langsam. Anfang 1997 erschien dann die Java-Version 1.1. Die Standartbibliothek ist erweitert worden und das Eventmodell vollständig neu modelliert und implementiert. Anfang 1998 wurde dann eine auf den Namen *Swing* getaufte Menge von Dialogelementen vorgestellt. Mit diesen Dialogelementen lassen sich nun auch anspruchsvolle grafische Oberflächen entwickeln, die einzelnen Dialogelemente können das Look&Feel von unterschiedlichen Computerplattformen imitieren. Diese neuen Bedienelemente sind Teil der *Java Foundation Classes* (JFC), die dann Anfang 1999 in die Java-Version 1.2 (auch Java 2 genannt) aufgenommen wurden.

Da diese Java-Version noch relativ neu ist, kann man nicht davon ausgehen, daß sie auf allen Rechnern schon installiert ist. Für eine Applikation, die in Java geschrieben würde, wäre es denkbar, eine Java-Version 1.2 zu wählen und dann die gesamten benutzten Java-Bibliotheken dem Quellprogramm hinzuzufügen. Dies würde aber die Größe eines Applets, das ja aus dem Internet heruntergeladen wird, stark aufblähen und zu nicht akzeptablen download-Zeiten führen.

Die beiden meist genutzten Internet-Browser Netscape Navigator und Internet Explorer unterstützen seit ihren 4.0-Versionen Java 1.1, so ist es sinnvoll, KIEL-online auf der Basis dieser Java-Version zu entwickeln.

3.2 Unterstützte Programmiersprache

Applikation und Applet müssen die gleiche Sprache unterstützen, in der Ausdrücke und Rechenvorschriften formuliert werden können, nämlich die in Kapitel 2.1 beschriebene Teilmenge von Standart-ML. Dafür ist für Rechenvorschriften und Ausdrücke jeweils ein Parser zu entwickeln, der auf der Grammatik basiert, die auch den Parsern der Applikation zugrundeliegt. Für Scanner und Parser gibt es zu Lex und Yacc ähnliche Scanner- und Parsergeneratoren, die entsprechende Java-Klassen ausgeben.

3.3 Laden von Ausdrücken und Rechenvorschriften

Das Einlesen von Rechenvorschriften erfolgt in der C-Applikation über einen Dateidialog (vergleiche Kapitel 2.2.1). Ausdrücke können genauso aus Textdateien eingelesen werden, die der Anwender auf der Festplatte abgespeichert hat. Sie werden aber wohl meistens direkt eingegeben. Das Applet sollte nun zumindest Rechenvorschriften von der Festplatte des Anwenders lesen können. Java verfolgt aber ein strenges Sicherheitskonzept, deshalb sind Festplattenzugriffe nicht möglich (genaueres hierzu folgt in Abschnitt 6.2.1). Die Übergabe der Rechenvorschriften an das Applet muß also auf eine andere Weise als über das Einlesen von Textdateien von der Festplatte erfolgen. Es muß hierfür einen Textbereich im zu entwickelnden Applet geben, in den dann die Rechenvorschrift übertragen werden kann. Dazu muß eine Textdatei, die die Rechenvorschriften enthält, vom Anwender in einem Editor geöffnet und die gewünschten Rechenvorschrift mit der Maus markiert werden. Dann kann der markierte Text per Mausklick oder auf MS-Windows-Systemen durch die Tastenkombinationen CTRL-c und CTRL-v in das Textfenster übertragen werden. Die Datenübertragung von Rechenvorschriften erfolgt also durch direkte Eingabe oder durch die beschriebene copy-and-paste–Technik.

3.4 Konfiguration

Das KIEL-System ist umfangreich konfigurierbar. Dies erfolgt über Dateien auf der Festplatte, die beim Starten von KIEL eingelesen werden (vgl. Abschnitt 2.3). Wie im vorherigen Abschnitt bereits ausgeführt, ist diese Möglichkeit des Dateizugriffes einem Applet nicht erlaubt. Trotzdem sollen auch Auswertungsgrenzen und andere Parameter, wie beispielsweise die verwendete Schriftgröße, vom Benutzer einstellbar sein. Dies muß bei KIEL-online zur Laufzeit durch den Aufruf eines Dialogfensters möglich sein.

3.5 Bedienoberfläche

Eine weitere Anforderung an das Applet war, die Struktur der Bedienoberfläche, d.h. alle Fenster und das Aussehen der einzelnen Fenster beizubehalten. Grundsätzlich ist dies möglich; Applets, die im Browserfenster ausgeführt werden, können eigene Fenster öffnen, die nicht zum Web-Browser gehören. Tests haben aber gezeigt, daß speziell auf Unix-Systemen das Initialisieren und Darstellen eines neuen Fensters überdurchschnittlich lange dauert. Selbst kleine Fenster, die nur Text enthalten, benötigen mehrere Sekunden, um angezeigt zu werden. Ein Grund hierfür ist sicherlich die Systemarchitektur, in der alle grafischen Ausgabedaten von einem X-Server generiert werden. Unter MS-Windows tritt das beschriebene Problem nicht auf.

Da das Applet aber auf allen Systemen effizient benutzbar sein soll, muß also die Anzahl der Fenster sinnvoll minimiert werden, und die Bedienelemente für Funktionen, die ursprünglich in einzelnen Fenstern waren, mit im Hauptfenster, also im Browserfenster, eingebettet werden. Dazu sind auch gewisse Einschränkungen in den Bedienmöglichkeiten nötig (vergleiche hierzu Kapitel 7). Es muß also an einigen Stellen der Funktionsumfang eingeschränkt werden, ein Beispiel hierfür ist das Ausdrucksübersichtsfenster von KIEL, in dem eine Liste aller bereits eingegebener Ausdrücke angezeigt wird. Wählt man hier per Mausklick einen Ausdruck aus, so wird sein derzeitiger Auswertungsstand im Baumfenster angezeigt. Da besonders diese Ausdrucksübersicht im zur Verfügung stehenden Browserfenster viel Platz benötigt, fiel der Entschluß, sie ganz wegzulassen und nur einen Ausdruck zur Zeit zuzulassen, der jeweils neu in einem Textfeld eingegeben werden muß.

Die hier beschriebenen Veränderungen in der Bedienoberfläche spiegeln sich natürlich auch in den Datenstrukturen wider. So muß das Applet beispielsweise keine Liste von Ausdrücken mehr verwalten können.

Ein weiteres Problem ist die Auswahl der Aktion, die auf einem Knoten ausgeführt werden soll, also ob er vereinfacht, ausgewertet oder expandiert werden soll. Dies erfolgt in der Applikation über die Betätigung der linken, mittleren oder rechten Maustaste. In Java kann man aber die Maustasten nicht getrennt voneinander abfragen, sondern nur ein Drücken der Maus registrieren. Diese Einschränkung dient dazu, die Neutralität bezüglich der speziellen Plattform, auf der Java benutzt wird, zu gewährleisten. Der Anwender des Applets muß also eine andere komfortable Möglichkeit bekommen, um die Knoten bearbeiten zu können.

3.6 Baumfenster

Die Größe des Fensters, in dem die Baumstruktur des auszuwertenden Ausdrucks angezeigt wird, ist in der C-Applikation frei veränderbar. Auch für das Applet wird eine Möglichkeit gewünscht, das Baumfenster beliebig zu vergrößern. Da, wie im Abschnitt 3.5 erarbeitet, die gesamte Bedienoberfläche in einem Fenster Platz finden sollte, kann die Größe des Baumfensters höchstens den Platz einnehmen, der in dem Browserfenster neben allen Bedienelementen noch bleibt. Da dies in einigen Fällen, speziell bei geringer Bildschirmauflösung, allerdings nicht ausreichend ist, ist eine Möglichkeit zu realisieren, die auf Wunsch ein größeres Baumfenster bietet.

3.7 Applikation

Das Applet sollte außerdem auch als Applikation lokal auf einem Rechner ausgeführt werden können. Dabei sollen alle Betriebssysteme unterstützt werden, für die es eine Java 1.1 - Version gibt. Dazu muß eine Java-Klasse implementiert werden, die eine `main`-Methode enthält. Diese wird im folgenden `Main` genannt. Sind alle Applet-Klassen in einem Verzeichnis des lokalen Rechners abgelegt, so erfolgt dann ein Programmstart durch den Befehl `java Main`. Da eine lokale Applikation nicht denselben Beschränkungen wie ein Applet unterliegt, soll KIEL-online als Applikation die Möglichkeit haben, Dateien über einen Dateidialog auszuwählen und dann in das System zu laden.

Kapitel 4

Das Vorgehen bei der Entwicklung

Nachdem nun also die Anforderungen formuliert sind, soll die Vorgehensweise bei der Entwicklung des Java-Applets aus der C-Applikation beschrieben werden. Dies geschieht in drei Abschnitten:

Zuerst werden unabhänig von Quell- und Zielsprache drei verschiedene Ansätze angesprochen, die aus dem Code eines Quellprogramms das Zielprogramm entwickeln. Ein erster Ansatz generiert ein Objektmodell, aus dem dann in einem weiteren Schritt konkreter Code einer Zielsprache erzeugt werden kann. In den darauf folgenden Abschnitten werden zwei weitere unterschiedliche Ansätze beschrieben, mit denen Code direkt aus der Zielsprache generiert werden kann. Abschließend wird das Vorgehen bei KIEL-online begründet.

4.1 Generierung eines Objektmodells

Aus dem vorhandenen Code der Quellsprache läßt sich durch die genaue Analyse der Datenstrukturen und Prozeduren ein Objektmodell erstellen. Dieses beschreibt dann die statische Struktur des Systems durch Objekte und Relationen. Eine Technik hierzu ist beispielsweise die *Object Modeling Technique* (OMT). Durch diese Art der Modellierung geht man aber im Grunde einen Schritt zurück: Es kann davon ausgegangen werden, daß schon bei der Entwicklung des Quellsystems ein (möglicherweise objektorientiertes) Modell entworfen wurde, aus dem dann schrittweise der nun vorliegende Code des Quellprogramms gewonnen wurde. Es macht wohl wenig Sinn, diese Schritte wieder bis zum Modell zurückzuverfolgen. Möglicherweise ist sogar das beim Entwurf des Systems entwickelte Modell noch vorhanden, muß dann also auch nicht noch einmal erzeugt werden.

4.2 Generierung von Zielcode mit Toolunterstützung

Große Informationssysteme der Vergangenheit sind in den meisten Fällen in den Programmiersprachen Cobol oder PL/1 implementiert und bedienen ein Datenbanksystem. Seit Anfang der 90er Jahre versucht man, solche Systeme beispielsweise nach C++, Java oder objektorientiertem Cobol zu migrieren. Da die zugrundeliegenden Programme extrem groß sind, ist hier eine teilweise oder vollständige Unterstützung durch Software bei der Transformation der Programme unerläßlich.

Die Tools, die dafür seit Ende der 90er Jahre entwickelt werden, stützen sich auf verschiedene Transformationsansätze. Sneed beschreibt in [9] diese unterschiedlichen Ansätze und bemerkt, daß sie im wesentlichen jeweils von der Art des Programmes abhängen, das umgesetzt werden soll. Er schließt, daß es deshalb kein universelles Rezept gibt, und die Ansätze zur Transformation immer dem Programm angepaßt sein müssen.

Die verschiedenen Tools sind außerdem keine Wunderwerkzeuge. Das von ihnen erzeugte objektorientierte Programm wird sich vom Ideal der reinen Objektorientiertheit mehr oder weniger unterscheiden. Es muß also hier nach der maschinellen Transformation noch ein weiterer Bearbeitungsschritt erfolgen, in dem von Programmierern der erzeugte Code verbessert und umstrukturiert wird.

4.3 Generierung von Zielcode ohne Toolunterstützung

In diesem Abschnitt werden verschiedene Möglichkeiten besprochen, wie aus dem Code des Quellprogramms auf manuelle Weise sofort Zielsprachencode generiert werden kann. Eine Grundvoraussetzung für alle in diesem Abschnitt vorgestellten Verfahren ist, daß das Quellprogramm nicht zu groß ist. Dabei sind wohl ca. 20.000 Codezeilen die oberste Grenze für die hier beschriebenen Verfahren. Die spezielle Art des Vorgehens ist von der Quellsprache und der Qualität des Codes abhängig.

- **Vollständige Neuentwicklung**

  Der Code für das Zielprogramm sollte vollständig neu entwickelt werden, wenn es für Sprachkonstrukte der Quellsprache in der Zielsprache keine gut passenden Entsprechungen gibt, oder es in der Quellsprache überhaupt keine modernen Datenstrukturierungskonstrukte gibt. Dieser Fall tritt beispielsweise ein, wenn die Quellsprache eine funktionale Programmiersprache ist, oder beispielsweise Assembler oder Fortran, und die Zielsprache eine moderne objektorientierte Programmiersprache wie Java oder C++.

Ist außerdem der Code der Quellsprache fehlerhaft und läuft das Programm daher instabil, oder sind die im Quellprogramm gewählten Datenstrukturen ineffizient, sollte man sich dafür entscheiden, das Zielprogramm unabhängig von dem Sourcecode des Quellprogramms zu entwickeln, und sich nur an den zu implementierenden Funktionalitäten orientieren.

- **Teilweise Neuentwicklung**

  In den meisten Fällen ist keine vollständige Neuentwicklung nötig, sondern es müssen nur Teile der Quellapplikation neu entwickelt werden. Dies sind in erster Linie Programmteile, die auf Bibliotheken zugreifen, zu denen es in der Zielsprache keine genaue Entsprechung gibt. So müssen alle Programmteile, die eine grafische Benutzerschnittstelle implementieren, völlig neu entwickelt werden, da sich besonders Bibliotheken für grafische Bedienelemente und deren Ereignisverarbeitung sehr stark voneinander unterscheiden. Beispielsweise hat die Java-Bibliothek für grafische Komponenten, das *AWT* (Abstract Windowing Toolkit), mit anderen grafischen Bibliotheken zu wenig gemeinsam, als daß man den Code Befehl für Befehl wie in Abschnitt 4.3 beschrieben, umsetzen könnte.

- **Methoden mit gleicher Semantik**

  Für den Fall, daß es für die Sprachkonstrukte der Programmiersprache des Quellprogramms keine oder kaum Entsprechungen in der Zielsprache gibt, oder der Quellcode möglicherweise fehlerhaft ist, läßt sich auch eine andere Möglichkeit wählen: Die Datenstrukturen werden weitestgehend übernommen, und aus den Methoden bzw. Prozeduren des Quellprogramms werden jeweils Zielprogramm-Methoden bzw. Prozeduren mit gleicher Semantik abgeleitet, wobei aber der genaue Code der jeweiligen Prozedurrümpfe unbeachtet bleibt und nur die Funktionalität abgebildet wird.

- **Syntaktisches Vorgehen auf Prozedurebene**

  Findet man für die meisten Konstrukte der Quellsprache semantische Entsprechungen in der Zielsprache, ist idealerweise noch die Syntax der entsprechenden Konstrukte mit der der Zielsprache verwandt oder gleich, wie z.B. bei C++, Java und C, ist der Code weitestgehend fehlerfrei und sind die benutzten Datenstrukturen effektiv, so sollte man den Zielcode für die Methoden direkt aus den jeweiligen Methoden bzw. Prozeduren der Quellsprache ableiten. Falls das Quellprogramm in einer prozeduralen Programmiersprache geschrieben ist, sollten aus den im Quellprogramm benutzten Datenstrukturen Klassen gebildet und die Prozeduren zu Methoden der Klassen gemacht werden. Die Methoden der Zielsprache können in diesem Fall direkt durch befehlsweises Übersetzen der Quellsprache gebildet werden, hierbei wird dann für jeden Befehl der semantisch gleiche in der Zielsprache gewählt.

4.4 Vorgehen bei Kiel-online

Bei der Entwicklung von KIEL-online aus der Applikation KIEL habe ich mich dafür entschieden, vom C-Code der Applikation ausgehend den Java-Code nach der in Abschnitt 4.3 skizzierten Vorgehensweise zu entwickeln, wobei die Benutzerschnittstelle unabhänig vom Quellcode neu programmiert wurde. Die Generierung eines Objektmodells ist, wie in Abschnitt 4.1 bereits angesprochen, wenig sinnvoll. Es wurde ohne Toolunterstützung gearbeitet.

Gründe für das gewählte Vorgehen sind folgende:

- Das Applet soll bis auf wenige Einschränkungen die gleichen Funktionalitäten wie die C-Applikation zur Verfügung stellen (vgl. Abschnitt 3). Schon deshalb bietet sich eine Orientierung am Code des Quellprogramms an.
- Syntax und Semantik entsprechender Sprachkonstrukte von C und Java sind weitestgehend gleich, damit ist die bloße Codegenerierung für die Prozeduren bis auf einige Einschränkungen, die in Kapitel 6 besprochen werden, trivial.
- Alle weiteren in Abschnitt 4.3 erwähnten Voraussetzungen sind gegeben: Der Quellcode ist gut lesbar, klar strukturiert und fehlerfrei. Darüber hinaus sind die Datenstrukturen effizient und nach Java übertragbar.
- In Abschnitt 3.5 wurden Einschränkungen für das Applet formuliert, mit der Folge, daß die Bedienoberfläche von der Applikation abweichen wird. Schon aus diesem Grund müssen die Elemente für die Bedienoberfläche unabhängig vom C-Quellcode entworfen werden. Im gleichen Schritt kann dann auch eine klare Trennung zwischen GUI-Code und Code, der auf den Datenstrukturen arbeitet, erzielt werden.
- Das Quellprogramm ist noch nicht zu groß, deshalb ist ein Ansatz, bei dem eine manuelle Transformation vorgenommen wird, noch vertretbar. Außerdem sollte das entstehende Java-Applet aus sinnvoll gebildeten Klassen bestehen, eine manuelle Nachbereitung wäre also auch nötig, wenn werkzeugunterstützt gearbeitet würde.

C ist eine prozedurale Programmiersprache ohne Objektorientierung, Java verfolgt jedoch ein streng objektorientiertes Konzept. Es gilt also, aus dem C-Code sinnvoll Klassen zu bilden, und die objektorientierten Möglichkeiten wie Vererbung, Polymorphie und Kapselung zu nutzen. Im folgenden Kapitel wird beschrieben, wie dabei vorgegangen werden kann.

Kapitel 5

Allgemeine Konzepte zur Übertragung von prozeduralen in objektorientierte Sprachen

In diesem Kapitel wird nun eine Vorgehensweise für die Umsetzung von Quellcode einer prozeduralen Sprache in Zielcode einer objektorientierten Sprache vorgestellt. Nach den geschilderten Prinzipien wurde bei der Entwicklung von KIEL-online vorgegangen. Das Vorgehen läßt sich aber auf viele Programme bis zu einer gewissen Größe hin anwenden und basiert auf allgemeinen, sprachunabhängigen Konzepten. In den Beispielen werden allerdings die bei der tatsächlichen Umsetzung zugrundeliegenden Sprachen C bzw. Java benutzt.

An dieser Stelle sollen noch einige sprachliche Vereinbarungen getroffen werden. Wenn in diesem und in den nachfolgenden Kapiteln jeweils von einer prozeduralen Sprache die Rede ist, so ist damit eine prozedurale Programmiersprache gemeint, die im Gegensatz zu einer objektorientierten Sprache keine objektorientierten Konzepte enthält.

Um Quellsprache und Zielsprache zu unterscheiden, wird außerdem folgende Sprechweise benutzt: Funktionen und Prozeduren des Quellprogramms werden zusammenfassend immer Prozeduren genannt, alle Operationen in der Zielsprache werden Methoden genannt. Variablen werden entsprechend zu Attributen.

Im folgenden wird geschildert, wie aus prozeduralem Code Klassen der objektorientierten Zielsprache gebildet werden. Wo es sinnvoll erschien, wurden dem zugrundeliegenden System Beispiele entnommen. Ein auftretendes Problem ist der Umgang mit ehemals globalen Variablen des Quellprogramms. Hierzu werden verschiedene Lösungsmöglichkeiten diskutiert.

5.1 Klassenbildung

Um Klassen zu generieren, müssen die speziellen Merkmale der Klassen (Attribute und Methoden) aus dem Code des Quellprogramms herausgefiltert werden. Ein erster Schritt auf dem Weg zur Klassenbildung ist die Analyse der Typen im Quellprogramm. Es gibt zwei verschiedene Arten dieser Typen:

- Typen, die bezüglich des gesamten Systems als global betrachtet werden können, das heißt, sie sind beispielsweise in C über include-Dateien ansprechbar, und
- Typen, die jeweils nur in einer einzelnen Datei verwendet und nur von Funktionen benötigt werden, die in dieser Systemkomponente auftreten.

Ist der Typ nun ein Verbundtyp (C: `struct`..., Pascal: `record`...) oder ein Zeiger auf einen Verbundtyp, so wird für jeden dieser Verbundtypen eine Klasse definiert. Beide Arten von Typen werden dabei zuerst gleich behandelt, Unterschiede treten erst auf, wenn den entstandenen Klassen Dateien zugeordnet werden. Diese Zuordnung wird später in Abschnitt 5.1.8 besprochen.

5.1.1 Attribute

Aus den einzelnen Komponenten des Verbundtyps werden Attribute der Klasse gebildet, die Bezeichnung der Attribute sollte dabei unverändert bleiben, der Typ der Attribute ist jeweils der der Quellsprache entsprechende. In den meisten Fällen wird es eine solche Entsprechung auf der Typebene geben, wenn es sich um Basisdatentypen der Quellsprache handelt. Ist ein entsprechender Datentyp in der Zielsprache nicht vorhanden, so muß seine Implementierung in einer weiteren Klasse vorgenommen werden. Falls der Typ einer Komponente ein weiterer Verbundtyp ist, ist für diesen wiederum eine eigene Klasse zu definieren, der Typ des Attributes ist dann diese neue Klasse. Ist der Typ einer Komponente ein Zeiger auf einen Verbundtyp, so handelt es sich hierbei möglicherweise um einen Kandidaten für eine spätere Assoziation zwischen den den Verbundtypen entsprechenden Klassen (vergleiche Abschnitt 5.1.12).

In manchen Programmiersprachen (Java, Smalltalk) werden die hier erzeugten Attribute auch *Instanzvariablen* genannt, da sie von einer konkreten Instanz der Klasse abhängen und diese beschreiben. Im folgenden werden die beiden Begriffe Attribut und Instanzvariable synonym verwendet.

Beispiel (Attribute)

In der Quellsprache (C) gebe es eine globale Variable `rechenvorschrift` vom Typ `*RECHENVORSCHRIFT_TYP`. Der Verbundtyp hat folgende Form:

```
typedef struct RECHENVORSCHRIFT_STRUCT
{
  gchar                 *datei_name;    /* Dateiname      */
  gchar                 *name;          /* Name           */
  GList                 *argumente;     /* Argumente      */
  gint                   ergebnissorte; /* Ergebnissorte  */
  AUSDRUCK_TYP           rumpf;         /* Rumpf          */
  gchar                 *eingabe_string;/* Eingabe-String */
} RECHENVORSCHRIFT_STRUCT;
typedef RECHENVORSCHRIFT_STRUCT    *RECHENVORSCHRIFT_TYP;
```

Daraus wird dann eine Java-Klasse wie folgt abgeleitet:

```
public class Rechenvorschrift
{
  public String           datei_name;
  public String           name;
  public Vector           argumente;
  public int              ergebnissorte;
  public Ausdruck         rumpf;
  public String           eingabe_string;
...// weitere Variablen und Methoden können folgen
}
```

Es ist hier erkennbar, wie die entsprechenden Basisdatentypen ersetzt wurden, beispielsweise wurde für den GLIB-Typ `gchar*`, mit dem in C eine Zeichenkette dargestellt werden kann, der Java-Typ `String` benutzt, genauso wurde der Typ für Listen aus der GLIB-Bibliothek durch die Java-Klasse für Listen, `Vector`, ersetzt. Der Typ der Instanzvariablen `rumpf` ist die Klasse `Ausdruck`, die genau wie die Klasse `Rechenvorschrift` generiert wird. Weiterhin fällt auf, daß in der Java-Version keine Zeiger vorkommen. Diese gibt es in Java nicht (vergleiche hierzu Abschnitt 6.1.2). In einer C++ -Version würden die Zeiger beibehalten werden.

In objektorientierten Sprachen gibt es weiterhin Möglichkeiten zur Kapselung von Attributen, mit denen der Zugriff auf sie eingeschränkt werden kann. Dies können bestimmte vorangestellte Modifier sein oder ein *export*-Anweisungsteil, der angibt, welche Variablen nach außen sichtbar sind. Im Fall von Java und C++ wird der Modifier `public` benutzt, um den Zugriff auf Attribute für alle anderen Klassen zu erlauben, und `private`, um den Zugriff einzuschränken. Unabhängig von der speziellen Zielsprache ist es auf jeden Fall notwendig, die hier gewonnenen Attribute zu veröffentlichen, da andere Klassen auf sie zugreifen können sollen.

5.1.2 Von der Prozedur zur Methode

Bisher bestehen die erzeugten Klassen nur aus ihren Instanzvariablen. Nun werden die Prozeduren des Quellprogramms ihren entsprechenden Klassen zugeordnet. Die Umformung der Prozedurrümpfe erfolgt dann in einem zweiten Schritt.

Es werden nun alle Prozeduren, die hauptsächlich auf einem der Verbundtypen des Quellprogramms arbeiten, ihn erzeugen, verändern oder für Berechnungen benutzen, zusammengefaßt und der dem Verbundtyp entsprechenden Klasse hinzugefügt. Einen guten Anhalt für die Identifizierung zusammengehörender Prozeduren kann hierbei die Dateistruktur des Quellcodes des Quellprogramms liefern. Die entstehenden Methoden müssen auch passende Modifier erhalten. Wird im Quellprogramm eine Prozedur anderen Systemkomponenten beispielsweise über Header-Dateien zugänglich gemacht, so muß die entsprechende Methode öffentlich sein, im anderen Fall privat. Dies trägt zur Kapselung der Klassen bei.

Beispiel (Von der Prozeduren zur Methode)

In einer Datei, die zum Code des Quellprogramm gehört, ist beispielsweise folgende Funktion enthalten:

```
gchar *rv_mit_funktionalitaet(RECHENVORSCHRIFT_TYP rv)
{
  ...//Anweisungsteil der Funktion
}
```

Nach Analyse des Codes der Funktion stellt man dann fest, daß hier von der als Parameter übergebenen Rechenvorschrift die Funktionalität als String zurückgeliefert wird. Es ist also unmittelbar klar, daß diese Funktion als Methode zur Klasse `Rechenvorschrift` gehört, im folgenden soll gezeigt werden, wie sie in Java in die Klasse eingefügt wird:

```
public class Rechenvorschrift
{
  ...       //Attribute
  public String rv_mit_funktionalitaet()
  {
   ...    //Anweisungsteil der Methode
  }
}
```

Der Rückgabeparameter wurde entsprechend Abschnitt 5.1.1 angepaßt, und die Methode wird mit einem Modifier versehen, um sie zu veröffentlichen. Je nachdem, ob die betreffende Funktion im Quellprogramm anderen Systemkomponenten zugänglich ist, also beispielsweise über Header-Dateien eingebunden werden kann, wird hier der Zugriff spezifiziert.

Die Signatur der Prozedur ändert sich jedoch: Es wird nun die Rechenvorschrift, deren Funktionalität bestimmt werden soll, nicht mehr als Parameter übergeben, sondern sie ist implizites Argument der Methode. Innerhalb der Methode erfolgt der Zugriff dann durch spezielle vordefinierte Bezeichner, beispielsweise `this` in C++ und Java, `Current` in Eiffel und `self` in Smalltalk. Dieser Zeiger liefert jeweils eine Referenz auf die konkrete Instanz der Rechenvorschrift, mit der die Methode aufgerufen wurde. Ein Aufruf sieht in Java beispielsweise wie folgt aus:

```
Rechenvorschrift rv=new Rechenvorschrift(...);//Erzeugen
                                         //einer Rechenvorschrift
String erg=rv.rv_mit_funktionalitaet();      //Aufruf der Methode
```

Andere Parameter, die übergeben werden, bleiben natürlich erhalten, von ihnen wird nur der Typ angepaßt.

5.1.3 Probleme beim Aufteilen der Prozeduren

Es ist auch möglich, daß bestimmte Prozeduren auf mehreren Verbundtypen gleichzeitig arbeiten oder diese verändern, und es nicht entscheidbar ist, welcher Klasse die Prozedur zugeordnet werden kann. In diesem Fall bietet sich an, eine neue Klasse zu schaffen, und die Prozedur der neuen Klasse zuzuordnen. Hier kann es sinnvoll sein, aus dieser neuen Methode eine statische Elementfunktion (auch Klassenmethode genannt) zu machen. So kann diese Methode ausgeführt werden, ohne daß eine konkrete Instanz der neuen Klasse erzeugt werden muß. In C++ und Java erfolgt die Qualifizierung als statische Elementfunktion durch den Modifier `static`.

Beispiel (Probleme beim Aufteilen der Prozeduren)

Im C-System gebe es eine Prozedur `auswertung()` , die als Parameter die aktuelle Auswertungsart, den auszuwertenden Knoten, den gesamten Ausdruck, eine Liste von Rechenvorschriften und einen weiteren Ausdrucksbaum enthält:

```
void auswertung(gint ausw_art,AUSDRUCKBAUM_TYP *abaump,
                AUSDRUCK_TYP adk,GList *rv_liste,
                AUSDRUCKBAUM_TYP *ursprungs_knotenp)
{...}
```

Diese Prozedur wertet den Ausdruck an der Stelle, die der erste Ausdrucksbaum bezeichnet, mit Hilfe der Rechenvorschriften in der übergebenen Liste aus. An dieser Stelle ist es nicht eindeutig entscheidbar, welcher Klasse diese Prozedur zuzuordnen ist. Eine Möglichkeit wäre, sie zu der Klasse für Ausdrücke zu verschieben. Da allerdings die Hauptfunktion des gesamten Systems die Auswertung von Ausdrücken ist, sollte eben diese Auswertung nicht in einer Klasse als Methode versteckt werden. So wurde eine eigene Java-Klasse `Auswertung` implementiert, die als Attribute alle Informationen über die aktuelle Auswertung beinhaltet, und als Operationen alle Methoden, die für die Auswertung zuständig sind.

5.1.4 Polymorphismus

Es wurde bisher noch nichts über die Namensgebung für die neuen Methoden gesagt. Grundsätzlich sollte der Methodenbezeichner nicht verändert werden. Es gibt allerdings Ausnahmen, in denen spezielle Methodennamen gewählt werden sollten.

Objektorientierte Sprachen unterstützen das Prinzip des Polymorphismus, d.h. eine Operation kann sich in unterschiedlichen Klassen unterschiedlich verhalten. So ist beispielsweise das Verhalten der Operation `verschieben` in einer Klasse `Fenster` ein anderes als in einer Klasse `Schachfigur`. Zur Laufzeit wird dann abhängig von der Klasse die passende Methode gewählt (Methodenresolution).

Führen also bestimmte Methoden auf verschiedenen Klassen anschaulich ähnliche Dinge aus (z.B. löschen, einfügen), so sollten diese Methoden in jeder Klasse den gleichen Bezeichner bekommen.

Beispiel(Polymorphismus)

In dem zugrundeliegenden System gebe es die Möglichkeit, Rechenvorschriften und Ausdrücke zu laden, zu parsen und auf Richtigkeit zu überprüfen. In der C-Version gebe es dafür folgende Prozeduren:

```
lade_Ausdruck(), Ausdruck_parsen(),
ueberpruefe_ob_Ausdruck_richtig(),
lade_Rechenvorschrift(), Rechenvorschrift_parsen(),
ueberpruefe_ob_Rechenvorschrift_richtig()
```

Nach der Umwandlung der Prozeduren in Methoden und der Aufteilung auf passende Klassen sollten in diesem Fall Namensänderungen vorgenommen werden, um Polymorphismus zu unterstützen:

```
class Ausdruck
{
 public void laden(){...}
 public void parsen(){...}
 public boolean istRichtig(){...}
}
```

```
class Rechenvorschrift
{
 public void laden(){...}
 public void parsen(){...}
 public boolean istRichtig(){...}
}
```

So erhalten Operationen, die gleiche Dinge leisten, denselben Namen. Dies dient zur Klarheit und hilft insbesondere, Vererbungsmöglichkeiten später besser zu erkennen.

5.1.5 Überladen von Methoden

Es gibt weitere Gründe, Methodenbezeichner zu verändern. So lassen sich Methoden in objektorientierten Sprachen überladen, d.h. es kann Methoden mit unterschiedlicher Signatur geben, die aber den gleichen Bezeichner haben. Sind im Code des Quellprogramms beispielsweise mehrere Methoden mit unterschiedlicher Parameterliste enthalten, die aber alle auf demselben Datentypen dieselbe Operation ausführen, beispielsweise in eine Liste anfügen, so sollten alle Methoden einen Methodennamen wie `anfuegen` bekommen. Je nach den Parametern wird dann die passende Methode ausgewählt.

Beispiel (Überladen von Methoden)

In einer C-Applikation gebe es eine Prozedur, die dazu da ist, den Wert einer arithmetischen Operation aufzunehmen und zu testen, ob keine Zahlenbereichsdefinitionen überschritten werden. Der erzeugte Wert tritt als Referenzparameter in der Parameterliste auf. Es gebe nun mehrere Prozeduren für verschiedene Typen, die im folgenden aufgeführt sind:

```
void uebernehme_int(int n,RETVAL_TYP retval)
{
 if ((!teste_int_overflow(n,retval))&(!teste_int_underflow(n,retval)))
 { retval->typ=INT_TYP; retval->_int=(int)n;}
}

void uebernehme_real(real n, RETVAL_TYP retval){...}
void uebernehme_string(*char n, RETVAL_TYP retval){...}
```

Dabei sind die Prozedurrümpfe von allen Prozeduren sehr ähnlich. Diese drei Prozeduren können nun in eine Java-Klasse `RetVal` verschoben und umbenannt werden:

```
class RetVal
{
 /* ...  Attribute und weitere Methoden */
 public void uebernehme(int i)
 {...}
 public void uebernehme(float f)
 {...}
 public void uebernehme(String s)
 {...}
}
```

Ruft man `uebernehmen()` nun mit einer Ganzzahl, einer Gleitkommazahl oder einer Zeichenreihe auf, so wird beim Compilieren und Binden oder spätestens zur Laufzeit die richtige Methode ausgewählt.

5.1.6 Konstruktoren und Destruktoren

Unter den Prozeduren des Quellprogramms wird es spezielle geben, die dafür zuständig sind, Speicherplatz für Variablen eines Verbundtyps vom System anzufordern und die Variablen zu initialisieren. In manchen Programmiersprachen wie Pascal muß der Speicher nicht explizit allokiert werden, es reicht hier die einfache Deklaration der Variable oder im Fall eines Zeigers der Prozeduraufruf `new(...)`, in anderen Programmiersprachen wie C muß die benötigte Speichergröße für eine Struktur genau spezifiziert werden.

Aus diesen Prozeduren werden in objektorientierten Sprachen Konstruktoren. Dies sind spezielle Methoden, die für den Programmierer passenden Speicherplatz zur Verfügung stellen und ein neues Objekt der Klasse generieren. Für Konstruktoren gibt es meistens bestimmte Namenskonventionen in den unterschiedlichen Sprachen. So muß in C++ der Konstruktor den gleichen Namen wie die Klasse selbst haben, in Eiffel heißt jeder Konstruktor `Create(...)`, in Smalltalk ist kein ausdrücklicher Konstruktor nötig, es können aber Methoden als Konstruktoren definiert werden.

Nachdem die als Konstruktoren in Frage kommenden Prozeduren nun identifiziert wurden, müssen sie umbenannt werden, im Anweisungsteil können alle Befehle, die explizit Speicher allokieren, weggelassen werden, denn dies übernimmt in den meisten modernen objektorientierten Sprachen der Compiler. Die weiteren Initialisierungen der Variablen werden übernommen. Zur allgemeinen Umsetzung der Befehle im Rumpf vergleiche Abschnitt 5.1.7.

Genau wie es Prozeduren gibt, die Speicherplatz anfordern, gibt es in vielen Sprachen auch Prozeduren und Befehle, die den benutzten Speicher wieder freigeben. Es gibt auf objektorientierter Seite hierzu in einigen Sprachen (z.B. C++), eine direkte Entsprechung, nämlich *Destruktoren.* In anderen Sprachen gibt es einen sogenannten *Garbage Collector*, der dem Programmierer die Speicherfreigabe abnimmt.

Ein Destruktor ist eine Methode, die alle Aufräumarbeiten durchführt, die vor der Zerstörung eines Objektes notwendig sind, und danach den Speicher wieder freigibt. Ist die Zielsprache also beispielsweise C++, so kann für die eingangs beschriebenen Prozeduren ein Destruktor erzeugt werden. Ist eine Methode ein Destruktor, so muß sie in C++ als Bezeichner den Klassennamen und ein vorangestelltes Tilde, '∼' haben.

Der Grundgedanke der Garbage Collection besteht darin, daß auf ein Objekt, auf das nicht mehr verwiesen wird, auch nicht mehr zugegriffen wird. Der Garbage Collector ist ein Prozess, der nach diesen unbenutzten Objekten sucht und den betreffenden Speicher wieder freigibt, sobald ein bestimmter Schwellwert erreicht ist, oder ein expliziter Aufruf getätigt wird.

Benutzt man also eine objektorientierte Sprache mit Garbage Collection, so sind Destruktoren nicht notwendig, die betreffenden Prozeduren oder Befehle zur Speicherfreigabe können dann in der Zielsprache weggelassen werden.

Beispiel (Konstruktoren und Destruktoren)

Wir betrachten im folgenden eine C-Prozedur, die Speicher für eine Variable des Typs `ARGUMENT_TYP` bereitstellt:

```
ARGUMENT_TYP create_argument(gchar *bez,gint sort)
{ ARGUMENT_TYP t;
  t=(ARGUMENT_TYP)mtmalloc(sizeof(ARGUMENT_STRUCT));
  t->name=g_strdup(bez);
  t->sorte=sort;
  return(t);
}
```

Hier wird also aus einer Zeichenkette und einer ganzen Zahl, die die Sorte angibt, eine Variable erzeugt. In Java würde man der Klasse `Argument` einen entsprechenden Konstruktor hinzufügen:

```
class Argument
{
  public String name; // Attribut der Klasse
  public int sorte;   // Attribut der Klasse
  public Argument(String name, int sorte) //Konstruktor
  {
   this.name=name; this.sorte=sorte;
  }
}
```

Es wird hier nicht explizit neuer Speicher angefordert. Dies ist nicht nötig, da diese Methode durch ihren Namen als Konstruktor erkannt wird, der Speicher wird dann implizit zugeteilt. Es bleiben die beiden Befehle zur Initialisierung des neuen Ausdrucks. Diese werden einfach nach Java übersetzt. Wie einzelne Befehle genau umgesetzt werden, ist dem nächsten Abschnitt zu entnehmen.

5.1.7 Umsetzung des Prozedurrumpfes

Nachdem der Prozedurkopf umgesetzt wurde, ist nun noch der Rumpf der Prozedur umzuformen. Hier wird nun, wie in Abschnitt 4.3 angedeutet, befehlsweise der Quellcode in den Zielcode übersetzt. Lokale Variablen in der Prozedur bleiben erhalten, es wird nur ihr Typ angepaßt. Für jedes Sprachkonstrukt der Quellsprache wird ein semantisch entsprechendes Konstrukt in der Zielsprache gewählt. Existiert ein bestimmtes Konstrukt nicht, so muß es durch mehrere semantisch gleiche ersetzt werden. Mit den in 4.3 getroffenen Voraussetzungen kommt dieser Fall allerdings eher selten vor.

Beispiel (Umsetzung des Prozedurrumpfes)

Als Beispiel diene hier nun noch einmal die bereits vorgestellte C-Prozedur `rv_mit_funktionalitaet(...)`. Wir nehmen die nachfolgend angegebene komplette Version an:

```
gchar *rv_mit_funktionalitaet(RECHENVORSCHRIFT_TYP rv)
 {
  gchar *erg,*erg_neu;
  gint i;
  ARGUMENT_TYP argument;
  gchar *sortenname;
  erg=g_strconcat("val ",rv->name," = fn :  ",NULL);
  for(i=0;i<g_list_length(rv->argumente);i++) {
    argument=(ARGUMENT_TYP)g_list_nth_data(rv->argumente,i);
    sortenname=sorten_name(argument->sorte);
    if (i!=g_list_length(rv->argumente)-1)
      erg_neu=g_strconcat(erg,sortenname," * ",NULL);
    else
      erg_neu=g_strconcat(erg,sortenname,NULL);
    g_free(sortenname);
    g_free(erg);
    erg=erg_neu;
  }
```

```
  sortenname=sorten_name(rv->ergebnissorte);
  erg_neu=g_strconcat(erg," -> ",sortenname,NULL);
  g_free(sortenname);
  g_free(erg);
  erg=erg_neu;
  return(erg);
 }
```

Die Java-Version dieser Prozedur hat folgende Form:

```
public String rv_mit_funktionalitaet()
{
  String erg;
  String sortenname;
  Argument argument;

  erg="val "+this.name+" = fn :  ";
   for(int i=0;i<this.argumente.size();i++) {
    argument=(Argument)this.argumente.elementAt(i);
    sortenname=AusdruckBaumKnoten.getsorten_name(argument.sorte);
    if (i!= (this.argumente.size()-1))
      erg=erg+sortenname+" * ";
    else
      erg=erg+sortenname;
  }
  sortenname=AusdruckBaumKnoten.getsorten_name(rv->ergebnissorte);
  erg=erg+" -> "+sortenname;
 return (erg);
}
```

Wie hier ersichtlich ist, wurde der Code Zeile für Zeile umgesetzt. Die `for`-Schleife wurde nicht verändert, einzige Unterschiede sind die Benutzung des Zeigers `this` und die verschiedenen Methoden, mit denen auf der Instanzvariable `argumente` gearbeitet wird. Hier werden jeweils in C die Bibliotheksfunktionen der GLIB auf dem Datentyp `g_list` benutzt, in Java die Methoden, die die Klasse `Vector` bietet. Außerdem ist der Umgang mit Zeichenketten in Java komfortabler als in C, dadurch kann man einige Befehle vollständig weglassen.

5.1.8 Dateien für die erzeugten Klassen

Die bisher gemachten Betrachtungen sind sowohl für globale als auch für lokale Verbundtypen gültig. Der Hauptunterschied tritt auf, wenn nun die aus den verschiedenen Verbundtypen generierten Klassen in Dateien geschrieben werden.

Die generierten Klassen werden in diesem Zusammenhang folgendermaßen bezeichnet: Eine Klasse, die aus einem globalen Verbundtyp generiert wurde, wird im weiteren globale Klasse genannt, ihr Gegenstück ist dann die lokale Klasse.

Es sollte jede Klasse, die aus einem globalen Typ entstanden ist, in einer Datei abgespeichert werden, die den gleichen Namen wie die betreffende Klasse hat. Außerdem muß diese globale Klasse wiederum einen Modifier erhalten, der den Zugriff regelt. Da der Typ von allen Komponenten des Systems benutztbar sein soll, muß hier der Zugriff öffentlich sein.

Die Klassen, die aus lokalen Verbunddatentypen entstanden sind, müssen zu einer der eben erzeugten Dateien zugeordnet werden. In vielen Fällen läßt sich hier eindeutig diejenige globale Klasse bestimmen, deren Methoden die entsprechende lokale Klasse benutzen. Zu dieser Datei wird dann die Klassendefinition der lokalen Klasse hinzugefügt, es muß dann wiederum der Zugriff so eingeschränkt werden, daß nur Klassen in der zugehörigen Datei auf diese lokale Klasse zugreifen können. Es gibt hier auch die Möglichkeit, aus der lokalen Klasse eine innere Klasse der globalen Klasse zu machen.

Es ist aber durchaus möglich, daß sich keine eindeutige Klasse finden läßt, der die lokale Klasse zuzuordnen ist. Dies ist dadurch begründet, daß nach dem Aufteilen der Methoden in Klassen (vergleiche Abschnitt 5.1.3) Methoden, die vorher in derselben Datei standen und auf einen lokalen Verbundtyp zugegriffen haben, nun zu unterschiedlichen Klassen gehören und demzufolge nicht mehr in der gleichen Datei stehen. In diesem Fall sollte man aus der lokalen Klasse eine globale machen, sie also in eine eigene Datei schreiben, und mit einem Modifier versehen, der die Klasse öffentlich macht.

Andererseits kann dies ein Hinweis darauf sein, daß die Methoden, die auf Variablen desselben Verbundtyps zugreifen, den falschen Klassen zugeordnet wurden und so gemäß Abschnitt 5.1.2 zu einer Klasse zusammengefaßt werden sollten.

5.1.9 Private Attribute

Nachdem nun zu jeder Klasse die entsprechenden öffentlichen Attribute und die Methoden hinzugefügt wurden, und die Typen für lokale Variablen in der entsprechenden Datei vorhanden sind, müssen jetzt noch die privaten Attribute generiert werden. Diese werden nur von den Methoden einer Klasse selbst benutzt.

Dazu analysiert man wiederum den Quellcode und filtert alle Variablen heraus, die bezüglich einer einzelnen Systemkomponente global sind, d.h. auf die alle Prozeduren der Komponente Zugriff haben, die aber nicht beispielsweise in einer Header-Datei zum öffentlichen Zugriff zur Verfügung gestellt werden. Diese Variablen werden dann auch zu Attributen der betreffenden Klasse (vergleiche Abschnitt 5.1.1). Sie entsprechen entweder Basisdatentypen, oder sie gehören zu einer globalen oder lokalen Klasse. Durch das in Abschnitt 5.1.8 angegebene Vorgehen enthält die Datei bereits die lokalen Klassendefinitionen.

Die generierten Instanzvariablen sollten einen Modifier erhalten, der nur der

Klasse selbst oder möglicherweise noch abgeleiteten Klassen den Zugriff ermöglicht. In C++ und Java wird hier der Modifier `private` benutzt. In anderen Sprachen sind Variablen, die nicht ausdrücklich exportiert werden, privat. Es ist wichtig, den Zugriff auf diese Variablen zu beschränken, um eine bestmögliche Kapselung der Klassen zu schaffen.

5.1.10 Probleme beim Aufteilen von lokalen Variablen

Genau wie auf Prozedurebene (vgl. Abschnitt 5.1.3) ist es auch auf der Ebene der Variablen möglich, daß ihr Verteilen auf Klassen nicht eindeutig ist, da durch die Umwandlung der Prozeduren in Methoden und deren Aufteilung auf passende Klassen manche vormals lokale Variablen von Methoden benutzt werden, die nun zu verschiedenen Klassen gehören. In diesem Fall sollte aus so einer Variable eine globale gemacht werden, sie wird dann ein Attribut des Hauptprogramms (vgl. Abschnitt 5.1.13). In manchen objektorientierten Sprachen ist es auch möglich, nur bestimmte andere Klassen auf ein Attribut zugreifen zu lassen, dies wäre in diesem Fall eine noch bessere Lösung. In C++ gibt es hier das `friend`-Konzept, in Eiffel können Attribute selektiv exportiert werden.

Es kann aber auch andere Gründe geben, eine vormals lokale Variable zu einer globalen zu machen und wie in Abschnitt 5.1.13 zu behandeln. Die beiden wichtigsten sind:

- Andere Systemkomponenten können zwar nicht direkt auf eine bezüglich des gesamten Systems lokale Variable zugreifen, der Zugriff ist aber über eine Prozedur möglich, die beispielsweise über eine Header-Datei eingebunden werden kann, und als Rückgabeparameter die Variable liefert.

- Die Variable repräsentiert konzeptuell einen wesentlichen Teil des Hauptprogramms, ist also für die gesamte Funktionalität des Systems entscheidend.

5.1.11 Generalisierung

Analysiert man nun die erhaltenen Klassen, die aus den globalen Typen und den lokalen Typen gewonnen wurden, und speziell die Attribute und Methoden der Klassen, so stellt man oftmals fest, daß sich bestimmte Klassen gleich verhalten oder sich gewisse Methoden wiederholen.

Diese Klassen sind Kandidaten, unter denen sich unter Umständen eine Vererbungshierarchie modellieren läßt, oder es läßt sich für mehrere Klassen eine (möglicherweise abstrakte, d.h. instanzlose) Vaterklasse bilden. In diesem Fall sollte man betreffende Klassen entsprechend umformen, d.h. Attribute und Methoden in Vaterklassen verschieben und dann die betreffende Klasse von der Vaterklasse erben lassen.

Es sind in den meisten Fällen allerdings noch einige Methodeneigenschaften anzupassen, um Vererbungsmöglichkeiten besser zu nutzen. Dazu sollten Methodenbezeichner umbenannt werden (vergleiche auch Abschnitte 5.1.4 und 5.1.5), außerdem sollten Attributlisten von betreffenden Methoden passend erweitert und ähnliche Attribute, die aber in unterschiedlichen Klassen unterschiedliche Namen haben, angepaßt werden.

5.1.12 Assoziationen

Mit Assoziationen werden bei der objektorientierten Modellierung Relationen zwischen Klassen hergestellt und eine physikalische oder konzeptuelle Verbindung ausgedrückt. Assoziationen werden in der Regel durch Zeiger in einer oder beider Klassen auf die jeweils andere Klasse implementiert. In Abschnitt 5.1 wurden bereits Attribute als Implementierung von Assoziationen identifiziert.

Als Beispieldiene eine Assoziation zwischen den Klassen A und B, die aber nur in einer Richtung implementiert ist, d.h. in A ist ein Verweis auf B abgespeichert. Nun hat im allgemeinen der Aufrufer der Klasse A dafür zu sorgen, daß vorher eine Instanz b von B erzeugt wurde, und dann beim Erzeugen einer Instanz a von A dem Konstuktor als Referenz übergeben wird. Zum genauen Vorgehen zur Übergabe im Konstruktor vergleiche auch das Beispiel in Abschnitt 5.2.1.

5.1.13 Attribute des Hauptprogramms

Im objektorientierten Zielprogramm wird eine Klasse zu bilden sein, die als Hauptprogramm anzusehen ist, in der also alle Objekte enthalten sind, die für die gesamte Anwendung wichtig sind, und die die Kontrolle nach dem Programmstart übernimmt. Welche Attribute diese Klasse, die im folgenden `Main` genannt wird, genau enthalten muß, wird aus dem nächsten Analyseschritt deutlich. Dazu werden alle Variablen, die bezüglich des gesamten Programmpakets global sind, d.h. die von allen Systemkomponenten benutzt werden können, herausgefiltert. Jede globale Variable wird nun zu einem Attribut der Klasse `Main`.

Es bleibt allerdings die offene Frage, wie diese Instanzvariablen des Hauptprogramms nun anderen Klassen zugänglich gemacht werden. Damit befaßt sich der folgende Abschnitt.

5.2 Behandlung der globalen Variablen

Die globalen Variablen des ursprünglichen prozeduralen Programms waren für alle Prozeduren zugänglich, unabhängig davon, in welcher Programmdatei die Prozedur sich befand. In Programmiersprachen, die ein streng objektorientiertes Konzept verfolgen, gibt es aber keine globalen Variablen. Trotzdem müssen die betroffenen Attibute im entstandenen objektorientierten Programm allen Methoden zugänglich gemacht werden, die sie für Berechnungen benötigen. Hierzu wurde schon in Abschnitt 5.1.13 ein erster Schritt getan: Alle globalen Variablen sind in der Klasse `Main` als Attribute vorhanden. Nun muß noch geklärt werden, wie andere Klassen auf diese Attribute zugreifen können, um sie dann ihren Methoden zur Verfügung zu stellen. Es werden im folgenden verschiedene Möglichkeiten angesprochen, wie diese Attribute an die Klassen weitergereicht werden können.

5.2.1 Übergabe von Attributen im Konstruktor

Die Klasse `Main` ist dafür zuständig, ihre Attribute selbst zu erzeugen. Dies geschieht bei Attributen, die einen Basisdatentyp haben, automatisch. Für die Attribute, die zu Klassen gehören, müssen deren Konstruktoren (vgl. Abschnitt 5.1.6) aufgerufen werden. Um nun einer Klasse ein Attribut aus der `Main`-Klasse zugänglich zu machen, kann dieses im Konstruktor als Parameter gleich mit übergeben werden. Die betreffende Klasse erhält ein neues, privates Attribut, in dem ein Verweise auf das Attribut gespeichert werden kann, ihr Konstruktor muß entsprechend angepasst werden. In den meisten Fällen ist dies die beste Möglichkeit, anderen Klassen Attribute aus der Hauptprogramm-Klasse zur Verfügung zu stellen.

Beispiel (Übergabe von Attributen im Konstruktor)

Als Beispiel soll hier wieder einmal die Klasse `Rechenvorschrift` dienen. Methoden in dieser Klasse sollen auf eine globale Variable des Hauptprogramms zugreifen, in diesem Beispiel die Variable `liste`. Die Klasse `Main` hat dann folgende Form:

```
public class Main{
 private Vector liste;
 private Rechenvorschrift rv;
 /*
   ...  weitere Attribute
 */
 public Main() //Konstruktor von Main
 {
   rv=new Rechenvorschrift(...,liste);
   /*
     ...  weitere Anweisungen
   */
 }
}
```

Die Klasse `Rechenvorschrift` muß einen passenden Konstruktor haben und ein zusätzliches privates Attribut, hier `listeVonMain`, in dem ein Zeiger auf `liste` abgelegt werden kann:

```
public class Rechenvorschrift
{
  // ... bisherige Attribute
  private Vector listeVonMain;

  public Rechenvorschrift(...,Vector liste) //Konstruktor
  {
    // ... bisherige Anweisungen
    listeVonMain=liste;
  }
}
```

So kann eine beliebige Methode auf die Liste aus dem Hauptprogramm durch das Attribut `listeVonMain` zugreifen, das einen Verweis auf die Liste des Hauptprogramms enthält. In C++ würde man hier einen expliziten Pointer für diese Referenz benutzen.

5.2.2 Übergabe der Hauptprogramm-Klasse im Konstruktor

Eine zweite Möglichkeit ist, die Instanz der Klasse `Main` den anderen Klassen im Konstruktor als Parameter zu übergeben. Dazu muß der betreffende Konstruktor entsprechend erweitert werden. Im Konstruktor wird in einem neu zu definierenden privaten Attribut der Verweis auf die Instanz der Klasse `Main` abgespeichert. Soll nun in einer Methode auf ein Attribut von `Main` zugegriffen werden, so kann dies über den Verweis auf `Main` erfolgen. Das entsprechende Attribut der Klasse muß in diesem Fall öffentlich zugänglich sein.

Aus Entwurfssicht ist die Art des Zugriffs über die Übergabe der Instanz des Hauptprogramms allerdings unschön. Es ist besser, wenn man die entsprechenden Attribute einzeln übergibt (vgl. Abschnitt 5.2.1). Diese Vorgehensweise sollte deshalb nur verwendet werden, wenn eine Klasse auf zahlreiche Attribute von `Main` zugreift, und eine einzelne Übergabe der Attribute die Konstruktoren zu unübersichtlich machen würde.

Beispiel (Übergabe der Hauptprogramm-Klasse im Konstruktor)

Wir greifen hier noch einmal das Beispiel aus 5.2.1 auf. Die Attribute der Klasse `Main` sind unverändert, der Aufruf des Konstruktors `Rechenvorschrift(...)`

verändert sich:

```
public class Main
{
 public Vector liste;
 private Rechenvorschrift rv;
 /*  ...  weitere Attribute  */

 public Main() //Konstruktor von Main
 {
   rv=new Rechenvorschrift(...,this);
   /*
     ...  weitere Anweisungen
   */
 }
}
```

Die Klasse `Rechenvorschrift` bekommt das private Attribut `main`, in dem der `this`-Zeiger der Hauptprogramm-Klasse abgelegt wird:

```
public class Rechenvorschrift
{
  // ...  bisherige Attribute
  private Main main;

  public Rechenvorschrift(...,Main main) //Konstruktor
  {
     // ...  bisherige Anweisungen

    this.main=main;
  }
}
```

So kann eine beliebige Methode auf die Liste aus dem Hauptprogramm zugreifen, und zwar durch den Aufruf `main.liste`. Es ist auch möglich, daß der Zugriff auf Attribute des Hauptprogramms durch Methoden erfolgt. Dann wäre beispielsweise das Attribut `liste` in `Main` privat, und es würde eine öffentliche Methode `getListe()` zur Verfügung gestellt werden, so daß dann in der Klasse `Rechenvorschrift` der Zugriff über `main.getListe()` erfolgen könnte.

5.2.3 Statische Attribute

Manche globale Variablen des ursprünglichen prozeduralen Programms sind Konstanten oder haben nahezu konstanten Charakter, d.h sie werden möglicherweise beim Programmstart aus einer Datei eingelesen und dann die gesamte Laufzeit über nicht mehr verändert.

Diese globalen Variablen lassen sich zu statischen, öffentlichen Attributen der Klasse `Main` machen. Statische Attribute sind nicht an eine bestimmte Instanz einer Klasse gebunden. Sie existieren auch, wenn es gar keine Instanz der Klasse gibt. In Java werden diese speziellen statischen Attribute auch Klassenvariablen genannt, im Unterschied zu herkömmlichen Instanzvariablen. Der Zugriff auf diese statischen Attribute erfolgt direkt über den Klassennamen. Wird eine Klassenvariable veröffentlicht, so kann *jede* andere Klasse auf sie über den Klassenbezeichner zugreifen, in Java beispielsweise durch den Befehl `Main.constante1`.

Theoretisch ist denkbar, jedes beliebige Attribut statisch zu machen. Dies würde den etwas umständlichen Zugriff wie in Abschnitt 5.2.1 unnötig machen. Wie allerdings erwähnt, sind die Klassenvariablen instanzunabhängig, das bedeutet unter anderem auch, daß alle Instanzen der Klasse auf dieselben statischen Attribute zugreifen. Startet man von einem objektorientierten Programm also mehrere Instanzen, so teilen sich alle Instanzen die gleichen statischen Variablen, Veränderungen an diesen Variablen würden dann alle Instanzen betreffen. Man sollte also nur dann alle Attribute, die keine Konstanten sind, zu statischen Variablen machen, wenn immer nur eine Instanz des Hauptprogramms ausgeführt wird.

Eine weitere Möglichkeit, wie mit Konstanten umgegangen werden kann, ist sprachabhängig und wird deshalb nicht an dieser Stelle besprochen, sondern erst im Abschnitt 6.1.6.

5.3 Umgang mit Sprachbesonderheiten

In jeder Programmiersprache gibt es Besonderheiten, für die es in anderen Sprachen keine direkte Entsprechung gibt (z.B. Pointer auf Prozeduren, Templates). Außerdem gibt es zusätzlich Besonderheiten, die nicht direkt zur Programmiersprache gehören, aber auch bei dem geschilderten Übersetzungsverfahren berücksichtigt werden müssen (z.B. Präprozessoranweisungen, in denen Konstanten und Makros definiert sind). Manche dieser Probleme sind sehr gut lösbar (vergleiche Kapitel 6), beispielsweise das Auflösen von Präprozessoranweisungen, andere wiederum erfordern mehr Programmieraufwand, Lösungen sind hier sehr von der Zielsprache abhängig. Aber auch die schwierigeren Aufgaben, beispielsweise, wie Pointer auf Prozeduren aufgelöst werden, lassen sich in den meisten Zielsprachen durch die zur Verfügung stehenden objektorientierten Sprachelemente lösen.

Lösungsansätze für das Übertragen von Sprachbesonderheiten, die in der Quellsprache auftreten, sind speziell von Quellsprache (Art der Besonderheit) und Zielsprache abhängig. Aus diesem Grund können hier nicht allgemeine Regeln für jede Besonderheit einer möglichen Quellsprache formuliert werden.

Bei der Umsetzung von KIEL mußten auch für einige Sprachbesonderheiten von C (Präprozessoranweisungen) passende Entsprechungen in Java gefunden werden. Diese werden unter anderem im folgenden Kapitel besprochen, und können als Anhaltspunkt für Problemlösungen in anderen Quell- und Zielsprachen dienen.

Beispiel(Umgang mit Sprachbesonderheiten)

In der C-Applikation gebe es in einer Datei eine Präprozessoranweisung, die folgendes Makro enthält:

```
#define KIND(abaum,n)
        ((AUSDRUCKBAUM_TYP)g_list_nth_data((abaum)->kinder,n))
#define KIND_WERT(abaum,n)
        ((AUSDRUCKBAUM_TYP)g_list_nth_data((abaum)->kinder,n))->wert
```

Die Abkürzung `KIND(abaum,n)` liefert also das `n`-te Kind des Knotens `abaum`, `KIND_WERT(abaum,n)` dessen Wert. Aus diese beiden Makros lassen sich in Java leicht Methoden für die Klasse, die einen Knoten des Ausdrucksbaumes implementiert, entwickeln:

```
class AusdruckBaumKnoten
{
 public getKind(int i)
 {
  return((AusdruckBaumKnoten)this.getChildAt(i));
 }
 public RetVal getKindWert(int i)
 {
  return (((AusdruckBaumKnoten)this.getChildAt(i)).getWert() );
 }
}
```

So lassen sich in den allermeisten Fällen Makros problemlos auflösen. Selbst Makros, die durch `#ifdef`-Anweisungen für eine bedingte Übersetzung sorgen, lassen sich durch objektorientierte oder andere Techniken ersetzen.

Kapitel 6

Spezielle Berücksichtigungen bei Applets

Neben den Konzepten, die bei der Umsetzung von prozeduralen in objektorientierte Sprachen auf eine große Anzahl verschiedener Quell- und Zielsprachen übertragbar sind, gibt es Besonderheiten in der Zielsprache (vgl. Abschnitt 5.3), aber auch immer Besonderheiten in der Quellsprache, die bei der Umsetzung zu beachten sind. Dies war auch bei der Transformierung von KIEL nach KIEL-online der Fall. Zusätzlich zu den Besonderheiten der Programmiersprache Java waren auch noch Einschränkungen durch Applets zu berücksichtigen.

Im ersten Teil dieses Kapitels werden die Probleme erläutert, die bei der Übersetzung von prozeduralem Code in Java-Code auftreten können. Im zweiten Abschnitt werden dann zusätzliche Einschränkungen besprochen, die durch die Verwendung als Applet entstehen.

6.1 Behandlung von Java-Sprachmerkmalen

Es wird nie möglich sein, alle Konstrukte einer beliebigen Zielsprache nach Java zu übersetzen, da es bestimmte Einschränkungen von Java gibt, die berücksichtigt werden müssen. Java besitzt allerdings auch einige Merkmale, die möglicherweise in der Zielsprache nicht existieren, aber in Java benutzt werden können, um beispielsweise Code besser zu strukturieren oder eine sinnvolle Fehlerbehandlung durchzuführen.

So muß also für Konstrukte der Quellsprache, die in Java nicht existieren, eine entsprechende Implementierung gefunden werden. Es werden im folgenden für einige häufig auftretende Konstrukte Implementierungsbeispiele angegeben. Andererseits sollten Vorteile von Java-Konstrukten sinnvoll genutzt werden.

Es gibt sicherlich noch einige Besonderheiten von Java, wie eingebaute Konzepte für Nebenläufigkeit und Serialisierung von Daten, die hier nicht besprochen werden. In diesem Abschnitt geht es in erster Linie um diejenigen Features von

Java, die bei dem zugrundeliegenden Projekt benutzt wurden.

6.1.1 Fehlende Aufzählungstypen

In Java gibt es keine Aufzählungstypen wie in Pascal oder C. Dies ist in den meisten Fällen kein Problem, wenn neuer, objektorientierter Code entwickelt wird. Wird allerdings aus prozeduralem Code, der viele Aufzählungstypen verwendet, ein Java-Programm abgeleitet, wie dies in Abschnitt 4.3 besprochen wurde, so sollte man diese Aufzählungstypen alle auf dieselbe Weise auflösen, um die Übersichtlichkeit des Zielcodes zu erhalten.

In prozeduralen Sprachen werden Aufzählungstypen verwendet, um Typen mit abstrakten Werten zu generieren. Außerdem sind Aufzählungstypen in der Regel stark typisiert. Dadurch sind Zuweisungen von Variablen unterschiedlichen Aufzählungstyps nicht möglich und werden schon beim Compilieren erkannt. Genauso ist es meist nicht möglich, der Instanz eines Aufzahlungstyps beispielsweise eine Integer-Zahl zuzuweisen. Eine Umsetzung von Aufzählungstypen in Java sollte nun auch diese beiden Eigenschaften haben. Das folgende Beispiel zeigt die Definition eines Aufzählungstyps in C und zwei verschiedene Umsetzungen in Java. Eine weitere Implementierungsmöglichkeit für Aufzählungstypen wird dann noch in Abschnitt 6.1.2 angegeben.

Beispiel (Fehlende Aufzählungstypen)

In C seien folgender Aufzählungstyp und die Prozedur `p` definiert:

```
enum Ampelfarben {
 ROT;
 GELB;
 GRUEN
}
void p(...,Ampelfarbe farbe,...){...}
```

Die naheliegenste Möglichkeit in Java ist nun, die einzelnen Komponenten durch konstante Integer-Werte zu simulieren:

```
public static final int ROT=0;
public static final int GELB=1;
public static final int GRUEN=2;
void p(...,int farbe,...){...}
```

Damit könnte nun aber, anders als in der C-Version, jeder mögliche Integer-Wert der Methode `p` übergeben werden. Außerdem ist diese Version nicht so kurz und prägnant wie das Original.

Die zweite Möglichkeit hat gegenüber des Definierens von Konstanten einige Vorteile. Hier wird eine abstrakte Klasse `Ampelfarbe` definiert, die dann von den anderen Klassen erweitert wird:

```
public abstract class Ampelfarbe{}
public class Rot extends Ampelfarbe{}
public class Gelb extends Ampelfarbe{}
public class Gruen extends Ampelfarbe{}
void p(...,Ampelfarbe farbe,...){...}
```

Hierdurch erreicht man nun das gleiche Verhalten wie der ursprüngliche C-Code. Es können der Methode `p` nur Instanzen einer Subklasse von `Ampelfarbe`, also Instanzen von `Rot`, `Gelb` oder `Gruen` übergeben werden. Es bleiben außerdem auch die anderen geschilderten Aspekte der Typsicherheit erhalten. In Fallunterscheidungen, die im Zusammenhang mit Aufzählungstypen ja häufig vorkommen, können die einzelnen übergebenen Instanzen dann einfach durch ihre Klassenzugehörigkeit unterschieden werden.

6.1.2 Fehlende Zeiger

In Java gibt es keine expliziten Zeiger auf Variablen. Es gibt aber Referenztypen. Diese sind, neben den primitiven Datentypen, in Java die zweite Art von Datentypen. Zu den Referenzdatentypen gehören alle Objekte, einschließlich Strings und Arrays. Im Unterschied zu den primitiven Datentypen wird jede Objektvariable einer Klasse als Referenz gespeichert. Dadurch sind dann zwar effiziente Parameterübergaben an Methoden (vergleiche Abschnitt 6.1.3) und Zuweisungen möglich, allerdings werden die Möglichkeiten, selbst Datentypen wie verkettete Listen, Bäume oder Graphen zu implementieren, stark eingeschränkt. Die Standartbibliothek bietet hier jedoch Lösungen für die meisten gebräuchlichen Datentypen an.

Es gibt in Java auch keine Pointer auf Methoden. Diese werden besonders oft in grafischen Benutzeroberflächen und in anderen ereignisbasierten Systemen verwendet, um ausgehend von einer ankommenden Nachricht individuelle Methoden für jede Nachricht aufzurufen. Ein in Java sinnvoller work-around ist, sogenannte *Callback-Objekte* zu definieren. Diese können sich bei anderen Objekten registrieren lassen, um bei bestimmten Ereignissen von ihnen benachrichtigt zu werden. Das Callback-Objekt kann dann auf erhaltene Nachrichten reagieren. Weitere Details zur Anwendung sind [13] zu entnehmen. Eine Implementierung dieser Callback-Objekte kann recht aufwendig werden, wie das folgende Beispiel zeigt:

```
public interface Operation //Interface, das von Klassen
{                          //implementiert werden kann,
 public int invoke();      //um ein callback-Objekt
}                          //zu erzeugen.

public class RotGeworden implements Operation
{
 public int invoke(){...}  //reagiere auf rot gewordene
}                          //Ampel

public class GelbGeworden implements Operation
{
 public int invoke(){...}  //reagiere auf gelb gewordene
}                          //Ampel
```

Entsprechend werden Klassen `GruenGeworden` und `GelbRotGeworden` programmiert. Jetzt folgt eine weitere Implementierungsmöglichkeit des Aufzählungstyps `Ampelfarbe`.

```
final class Ampelfarbe
{
 private String name;
 private int val;
 public static final Ampelfarbe Rot=new Ampelfarbe("Rot",0);
 public static final Ampelfarbe Gelb=new Ampelfarbe("Gelb",1);
 public static final Ampelfarbe Greun=new Ampelfarbe("Gruen",2);
 public static final Ampelfarbe GelbRot=new Ampelfarbe("GelbRot",3);

 public Ampelfarbe(String name,int val)
 { this.name=name;this.val=val;}
 public int getVal(){return val;}
}
```

In der folgenden Klasse wird nun auf die Ampelfarben reagiert und das Ereignis jeweils an die entsprechende Klasse zur Behandlung weitergeleitet:

```
public class AutoAnAmpel
{
 private Vector opTable;
 public AutoAnAmpel()
 {
  opTable=new Vector();
  opTable.addElement(new RotGeworden());
  opTable.addElement(new GelbGeworden());
  opTable.addElement(new GruenGeworden());
  opTable.addElement(new GelbRotGeworden());
 }
 public int reagiereAufAmpelfarbe(Ampelfarbe ampelfarbe)
 {
  int index=ampelfarbe.getVal();
  Operation op=(Operation)opTable.elementAt(index);
  return op.invoke();
 }
}
```

In dieser Implementierung benutzt man also statt Zeiger auf Methoden einen Vektor, in dem verschiedene Klassen enthalten sind, die alle das Interface `Operation` implementieren. Je nach dem Ereignis kann so die `invoke()`-Operation für die entsprechende Klasse aufgerufen werden. Außerdem kann zur Laufzeit die Liste `opTable` verändert werden, dies garantiert eine große Flexibilität. Die vorgestellte Methode ist leider etwas aufwendig, aber in den meisten Fällen ein vollständiger Ersatz für Felder mit Pointern auf Methoden.

6.1.3 Parameterübergabe

Alle Parameter werden in Java per *call-by-value* an Methoden übergeben. Es lassen sich keine anderen Übergabearten wie in anderen Programmiersprachen wählen. Wie bereits erwähnt, sind Objektvariablen Referenzen auf die betreffende Instanz. Damit wird einer Methode also der Wert eines Zeigers übergegeben. Diese call-by-value–Übergabe von Zeigern entspricht wie in allen anderen Programmiersprachen der Semantik von *call-by-reference*.

Die Übergabe von Objekten an Methoden ist damit effizient, gleichgültig wie groß die betreffenden Objekte sind, da nur Zeiger übergeben werden. Oft ist aber auch ein call-by-reference–Verhalten bei der Übergabe von primitiven Datentypen erforderlich. Dazu verfügt Java in seiner Standartbibliothek über sogenannte *Wrapper*-Klassen für jeden primitiven Datentyp. Diese kapseln einen primitiven Datentyp in einer Klasse, und sie werden wie alle anderen Klassen auch als Referenz an Methoden übergeben. So kann auch ein primitiver Datentyp von einer Methode nach außen sichtbar verändert werden. Leider ist der

Einsatz dieser Wrapper-Klassen der in anderen Programmiersprachen bekannten Referenzübergabe von Datentypen nicht völlig gleichwertig, da es meistens keine Methoden gibt, die die gekapselten Werte verändern können.

Da Zeiger auf Objekte nur wertemäßig übergeben werden, treten jedoch Probleme auf, die in anderen Programmiersprachen nicht vorkommen. So ist die `String`-Variable `test` auch nach dem Aufruf der Methode `aendern` immer noch unverändert:

```
void aendern(String s)
{ s="geaendert";}
void irgendeineMethode()
{
 String t="hallo";
 aendern(t);
 // t ist immer noch "hallo".
}
```

Dieses Problem trat oftmals in der Umsetzung vom KIEL-System auf, speziell in den Methoden der Klasse `Auswertung`, in der die einzelnen Ausdrücke von Methoden bearbeitet und verändert werden müssen. Es gibt hier mehrere Lösungen, die im folgenden vorgestellt werden sollen:

- **Wrapper-Klassen**

  Das gleiche Konzept, das für alle primitiven Datentypen standartmäßig implementiert ist, läßt sich natürlich für jede beliebige Klasse realisieren. Dazu schreiben wir beispielsweise für die Klasse `String` eine Wrapper-Klasse:

  ```
  public class WrapperString
  {
  private String s;
  public WrapperString(String s)
  {this.s=s;}
  public void set(String s)
  {this.s=s;}
  public String toString()
  {return s;}
  }
  ```

Damit kann dann im ersten Beispiel das gewünschte Verhalten wie folgt erzielt werden:

```
void aendern(WrapperString s)
{
s.set("geaendert");
}
void irgendeineMethode()
{
WrapperString t=new WrapperString("hallo");
aendern(t);
// t ist nun geaendert.
}
```

- **einelementige Felder**

  Eine weitere Möglichkeit ist, ein einelementiges Feld zu benutzen, und in die einzige Komponente die `String`-Variable zu schreiben. Das Feld wird dann als Parameter an die `aendern(...)`-Methode übergeben. Obwohl diese Vorgehensweise relativ unschön ist, handelt es sich hier um eine sehr knappe Möglichkeit, die fehlende Referenzübergabe von Zeigern zu umgehen. Außerdem läßt sich dieses Verfahren auch leicht auf mehrere Objekte übertragen, indem man ein mehrelementiges Feld benutzt.

- **spezielle Referenzdatentypen**

  Es besteht außerdem die Möglichkeit, Datentypen zu benutzen, für die Operationen zur Verfügung stehen, die die Objekte verändern können. So kann beispielsweise statt der Klasse `String` die Klasse `StringBuffer` benutzt werden. Dies wird hier noch kurz ausgeführt:

```
void aendern(StringBuffer s)
{
s.setLength(0);
s.append("geaendert");
}

void irgendeineMethode()
{
StringBuffer t=new StringBuffer("hallo");
aendern(t);
// t ist nun geandert.
}
```

- **Rückgabeparameter**
  Eine letzte Möglichkeit ist, das veränderte Objekt als Rückgabeparameter der Methode zurückzuliefern. Dies läßt sich für mehr als einen Parameter aber nur durch weitere Hilfsklassen, in denen dann alle gewünschten Rückgabeparameter gekapselt werden, realisieren.

In C und C++ lassen sich außerdem Prozeduren und Funktionen schreiben, die in ihrer Parameterliste eine unspezifizierte Anzahl von Argumenten aufnehmen können. Meist wird diese Parameterliste dann durch `NULL` abgeschlossen. In Java gibt es diese Möglichkeit nicht. Möchte man also eine Methode aus einer solchen Prozedur oder Funktion entwickeln, so kann man beispielsweise an die Methode ein Feld oder einen Vektor übergeben, und die Parameter vorher in dieses Feld oder den Vektor kopieren.

6.1.4 Interfaces

Erben Klassen von mehreren Vaterklassen, so nennt man dies *Mehrfachvererbung*. In Java gibt es keine Mehrfachvererbung, jede Klasse kann nur von einer einzigen Vaterklasse abgeleitet werden. Die möglichen Schwierigkeiten im Umgang mit mehrfacher Vererbung und die Einsicht, daß das Erben von nichttrivialen Methoden aus mehr als einer Klasse selten zu erreichen ist, haben die Designer von Java dazu veranlaßt, diese Möglichkeit nicht zu realisieren. Es handelt sich hierbei um keine starke Beschränkung, da Mehrfachvererbung schon in der Modellierungsphase umgangen und aufgelöst werden kann, wie Rumbaugh et al. in [5] ausführen.

Andererseits sah man es beim Entwurf von Java sehr wohl als wünschenswert an, Methodendeklarationen von mehr als einer Klasse zu erben, und hat so mit den Interfaces ein Ersatzkonstrukt geschaffen, das genau dieses Feature bietet. Ein Interface ist eine besondere Form einer Klasse, die ausschließlich abstrakte Methoden und Konstanten enthält. Implementiert eine Klasse ein Interface, verpflichtet sie sich, alle definierten Methoden zu implementieren. Jede Klasse kann beliebig viele Interfaces implementieren.

Bei der Umsetzung von KIEL nach KIEL-online wurden Schnittstellen hierfür nicht benutzt, da nach der Analyse der entstandenen Java-Klassen (Abschnitt 5.1.11) keine Kandidaten für eine Mehrfachvererbung identifiziert wurden.

6.1.5 Pakete

Eine Klasse ist kein geeignetes Strukturierungskonstrukt für große Systeme. Den meisten objektorientierten Sprachen fehlen Partitionierungsmechanismen, die die Sichtbarkeit zwischen Klassen kontrollieren und Klassen sinnvoll zu Gruppen zusammenfassen können.

In Java gibt es hierfür Pakete. Pakete sind Mengen von Klassen, durch die Systeme in voneinander getrennte Komponenten unterteilt werden können. Die Sichtbarkeit von Klassen und Attributen läßt sich auf die Paketgrenzen mit dem Modifier `protected` einschränken. Pakete können weitere Unterpakete enthalten, so ist eine beliebig tiefe Pakethierarchie erzeugbar.

Es sollten im allgemeinen Pakete durch die Dienste identifiziert werden, die sie bereitstellen. Ein Dienst ist hierbei eine Gruppe verwandter Funktionen, wie Zeichnen von Bildern oder Ausführen arithmetischer Berechnungen. So gibt es in der Java-Standartbibliothek beispielsweise die Pakete `java.util` oder `java.awt`, in denen alle nützlichen höheren Datentypen bzw. alle graphischen Standartkomponenten implementiert sind.

In jedem größeren Java-Projekt sollte eine Verwaltung der Klassen in Paketen erfolgen.

6.1.6 Konstanten

In Java gibt es keine Präprozessoranweisungen, deshalb können Konstanten nicht in der von C gewohnten Weise durch `#define`–Anweisungen definiert werden. Es lassen sich aber statische Variablen (vergleiche Abschnitt 5.2.3) wie alle anderen Attribute mit einem weiteren Modifier, `final`, versehen. Wendet man `final` auf Attribute an, so darf die Variable nicht verändert werden. Kombiniert man also `final` mit statischen Variablen, so erzeugt man eine unveränderliche Klassenvariable, die nicht von Instanzen der Klasse abhängt und so auch ohne eine konkrete Instanz existiert. Bei Klassen mit vielen Konstanten oder bei Konstanten, auf die verschiedene Klassen zugreifen, ist es sinnvoll, diese Konstanten in eine eigene Klasse zu verschieben. Dadurch erfolgt der Zugriff auf die Konstanten aber immer durch `paket.unterpaket.Klasse.Konstantenname`, falls die Klasse nicht zum gleichen Paket wie die aufrufende Klasse gehört. Eine elegante Lösung ist hier das Verschieben der Konstantendefinitionen in Interfaces statt in Klassen, denn neben Methodenköpfen können Interfaces auch Attribute mit den Modifiern `static final` enthalten. Dann läßt man eine Klasse, die auf die Konstanten eines Interfaces zugreifen muß, diese Schnittstelle implementieren. Dadurch kann der Zugriff direkt über den Bezeichner der Konstante erfolgen.

6.1.7 Ausnahmebehandlung

Mit den *Exceptions* besitzt Java ähnlich wie C++ und ML einen Mechanismus zur strukturierten Behandlung von Fehlern, die während der Laufzeit auftreten. Dadurch kann eine aufgerufene Methode ihrem Aufrufer mitteilen, daß ein Fehler aufgetreten ist, um die weitere Behandlung des Fehlers kümmert sich dann der Aufrufer und reagiert entsprechend auf die aufgetretene Ausnahme. Die Benutzung von Ausnahmen hat damit beispielsweise in Bibliotheken gegenüber den traditionellen Methoden wie etwa der Rückgabe von Fehlerwerten einige Vorteile.

Tritt ein Laufzeitfehler auf, weil beispielsweise ein Array-Zugriff außerhalb der definierten Grenzen erfolgte oder weil eine zu öffnende Datei nicht gefunden wurde, so gibt es in Java Sprachmittel, die eine systematische Behandlung solcher Ausnahmen ermöglichen. Da eine Ausnahme eine Java-Klasse ist, können auch eigene Ausnahmen von den zur Standartbibliothek gehörenden Ausnahmen abgeleitet und eigene Fehlerobjekte instanziert werden. Dadurch erhält der Programmierer ein mächtiges Werkzeug zur Fehlerbehandlung. Das Grundprinzip des Exceptions-Mechanismus in Java kann wie folgt beschrieben werden:

- Ein Laufzeitfehler oder eine vom Entwickler gewollte Bedingung löst eine Ausnahme aus.
- Diese kann nun entweder von dem Programmteil behandelt werden, in dem sie ausgelöst wurde, oder sie kann weitergegeben werden.
- Nach der Weitergabe hat der Empfänger erneut die Möglichkeit, die Ausnahme zu behandeln oder sie weiterzuleiten.
- Wird die Ausnahme von keinem Programmteil behandelt, so führt sie zum Abbruch des Programms und zur Ausgabe einer Fehlermeldung.

Ausnahmen in Java sollten dann verwendet werden, wenn in Operationen des prozeduralen Quellprogramms, das nach Java übertragen werden soll, Rückgabewerte generiert werden, die der Fehlerbehandlung dienen, aber entweder als Rückgabeparameter zurückgeliefert werden oder als Referenz in der Parameterliste enthalten sind. Dadurch wird der Java-Code besser strukturiert, und genauer zwischen Ergebnis einer Funktion und möglichen Fehlerrückgaben unterschieden.

Prinzipiell kann Ausnahmebehandlung so implementiert werden, daß kein Mehraufwand zur Laufzeit auftritt, wenn kein Fehler geworfen wird. Zusätzlich ist es möglich, das Werfen so zu realisieren, daß es nicht viel aufwendiger als ein normaler Funktionsaufruf ist.

Beispiel(Ausnahmebehandlung)

Wir betrachten eine Methode, die testet, ob eine Zahl eine Primzahl ist. Die Methode erhält als Parameter die Zahl, die überprüft werden soll. Ist diese Zahl nicht größer als 0, so soll eine passende Ausnahme generiert werden. Dazu definieren wir zuerst eine Exceptions-Klasse, die von der Java-Klasse `ArithmeticException` erbt:

```
public class NumberLessThanOrEqualsZeroException
   extends java.lang.ArithmeticException {};
```

```
public boolean isPrim(int n)
throws NumberLessThanOrEqualsZeroException
{
  if (n<=0)
     throw new NumberLessThanOrEqualsZeroException();
  for (int i=2;i<=n/2;++i)
     if (n%i==0) return false;
  return true;
}
public void methode()
{
  try
  {
   boolean erg=isPrim(-2);
  }
  catch(NumberLessThanOrEqualsZeroException e)
  {
   //hier Verarbeitung der Ausnahme
  }
}
```

Die `throw`-Anweisung hat dabei den Charakter einer Sprunganweisung. Sie unterbricht das Programm an der aktuellen Stelle und verzweigt unmittelbar zu der umgebenden `catch`-Klausel. Gibt es eine solche nicht, wird der Fehler an den Aufrufer weitergegeben, also hier an `methode()`. Der `catch`-Block in `methode()` übernimmt dann die Reaktion auf die Ausnahme.

6.2 Einschränkungen durch Applets

Ein Applet ist ein Java-Programm, das über das Internet geladen werden kann und dann im Fenster eines Web-Browsers ausgeführt wird. Schon aus dieser Beschreibung werden zwei Hauptrestriktoren für Applets deutlich: Die Tatsache, das jedes Applet mit allen seinen Dateien aus dem WorldWideWeb heruntergeladen werden muß, beeinflußt die Größe des Applets und seiner Dateien. Weitere Einschränkungen werden durch das Browserfenster verursacht. Außerdem gibt es noch Beschränkungen, die beim Entwurf von Java aus Sicherheitsgründen vorgenommen wurden. Im folgenden wird auf die Einschränkungen eingegangen, denen Applets unterliegen.

6.2.1 Fehlender Festplattenzugriff

Um unkontrolierte Systemzugriffe zu verhindern, sind Festplattenzugriffe durch ein Applet im allgemeinen nicht möglich. Applets, die nicht lokal auf einem

Rechner abgelegt sind, d.h. sich nicht im `CLASSPATH` des Rechners befinden, oder nicht signiert sind (vgl. Abschnitt 8.2.2), können nicht auf die Festplatte des Rechners zugreifen. Mehrere Java-Sicherheitsmechanismen (*Bytecode-Verifier*, *ClassLoader*, *SecurityManager*) analysieren den gesamten Code und kontrollieren den Programmablauf. Falls eine Verletzung der Sicherheitsbestimmungen erfolgt, wird das Applet sofort abgebrochen.

Das Problem der fehlenden Festplattenzugriffe sollte bereits in einer frühen Entwicklungsphase des Applets berücksichtigt werden, damit schon bei der Planung sinnvolle Alternativen gefunden werden können. Zwei Möglichkeiten, auf Festplattenzugriffe zu verzichten, sind in dieser Diplomarbeit verwendet worden:

1. Statt einer Datei, aus der benutzerspezifische Einstellungen gelesen werden, wurde ein Dialogfenster implementiert, in dem der Anwender zur Laufzeit Veränderungen an Standarteinstellungen vornehmen kann.

2. Inhalt aus Textdateien wird per cut-and-paste in einen Textbereich übernommen, und nicht durch das Lesen der Textdateien von der Festplatte.

Die konkreten Umsetzungen sind dem Kapitel 7 zu entnehmen.

6.2.2 Größenbeschränkungen

Um Download-Zeiten möglichst kurz zu halten, muß die Programmgröße eines Applets minimiert werden. Dazu sollte der Java-Quellcode möglichst kurz, aber dennoch lesbar sein. Außerdem sollte man versuchen, mit den Klassen der Standartbibliothek und hierbei insbesondere mit den grafischen Bedienelementen des AWT auszukommen, und nicht die neuen Swing-Komponenten zu verwenden. Sollen diese nämlich verwendet werden, muß ein großer Teil des Swing-Paketes den jeweiligen Applet-Klassen hinzugefügt werden. Dies würde aber die Größe der zu übertragenden Dateien zu sehr anschwellen lassen. Wird Swing nicht benutzt, schränkt dies natürlich die Gestaltung von grafischen Bedienoberflächen ein. Allerdings sind die meisten einfachen Bedienelemente im AWT verfügbar.

- **Packen von Applets**

  Manche Applets sind klein und bestehen nur aus einer Klasse. Applets können aber grundsätzlich aus beliebig vielen Klassendateien bestehen, die dann *on demand* (d.h. wenn eine Instanz einer noch nicht geladenen Klasse erzeugt werden soll) über das Internet geladen werden.

  Es besteht die Möglichkeit, die gesamten Klassen oder große Teile der vom Applet benötigten Klassen in ein Archiv zu packen, dann dieses gesamte Archiv zu übertragen und vor dem Start des Applets wieder zu entpacken. Dies geschieht automatisch, der Web-Browser benutzt dazu das

Java-Werkzeug *jar*. Jar arbeitet nach dem gleichen Prinzip wie das Kompressionstool *zip* und reduziert beim Packen die Größe aller Applet-Dateien beträchtlich. Gleichzeitig werden alle Dateien zu einer einzigen zusammengefaßt. Dadurch muß nun nur eine einzige, gepackte Datei statt vieler kleiner übertragen werden, es ist nur ein einziger HTTP-Verbindungsaufbau erforderlich.

Im allgemeinen gilt es abzuwägen, ob in einem konkreten Fall Klassen archiviert übertragen werden sollen. Sind im normalen Ablauf des Applets viele unterschiedliche Klassen beteiligt, so ist es sinnvoll, diese alle in einem Archiv zu übertragen, und die geringe zusätzliche Zeit für das Entpacken in Kauf zu nehmen. Klassen, auf die nur in besonderen Situationen zugegriffen wird, sollten nicht gepackt werden, sondern einzeln bei Bedarf geladen werden. Dies verringert die Download-Zeit des nun reduzierten Paketes.

- **Größe von Klassendateien**

  In älteren Java 1.1-Versionen läßt es der Java-Verifier nicht zu, Klassendateien zu laden, die Methoden enthalten, die länger als 64 Kilobyte sind. Dies kann auch der Fall sein, wenn in der Java-Datei, die den Quellcode enthält, keine Methode zu lang scheint. In der vorkompilierten Klassendatei jedoch können Methoden dann stark vergrößert sein. Deshalb sollte jede Klassendatei, die die Größe von 64 Kilobyte überschreitet, in zwei oder mehrere kleinere Klassen aufgeteilt werden. Dies läßt sich beispielsweise dadurch realisieren, daß einige Methoden in eine abstrakte Vaterklasse verschoben werden.

6.2.3 Einschränkungen durch das Browserfenster

Im Idealfall sollte es durch das Browserfenster nur Einschränkungen geben, die durch die verfügbare Anzeigegröße für das Applet bedingt sind. Dem Bereich innerhalb des Browsers müssen dann alle Bedienelemente des Applets entsprechend angepaßt werden.

Praktisch verhalten sich aber die beiden meist genutzten Web-Browser, der Internet Explorer von Microsoft und der Netscape Navigator, nicht immer identisch und fehlerfrei. Dies liegt weniger an instabilen Java-Versionen als an der Schnittstelle von Browser und Applet, die von den Browser-Herstellern unterschiedlich realisiert wurde. Grundsätzlich sollte das Applet deshalb schon bei der Entwicklung, aber auf jeden Fall in der Testphase, auf unterschiedlichen Plattformen und beiden Browsern getestet werden.

Ein Problem, das bei KIEL-online auftauchte, soll hier kurz geschildert werden: Es ist nicht ohne weiteres möglich, die Größe des Applets immer auf die Größe des Browserfensters abzustimmen. Im HTML-Tag für den Aufruf eines Applets wird die Größe des Applets festgesetzt. Hier sind im Regelfall absolute Werte für die Länge und Breite anzugeben, der Internet Explorer versteht aber auch relative Werte für diese Angaben in Prozentzahlen. Macht man in der HTML-Datei relative Angaben, kann man nach dem Start des Applets die Größe des Browserfensters verändern, die Größe des Applets wird dann automatisch angepaßt. Für den Netscape Navigator gibt es diese Möglichkeit allerdings nicht, damit ist eine einheitliche Lösung also nicht möglich.

Es wird sicherlich verschiedene Möglichkeiten geben, diese Probleme zu umgehen, und möglicherweise sind sie in neueren Versionen der Browser von den Herstellern gelöst. Eine Möglichkeit, Größenänderungen des Netscape Navigators an das Applet weiterzuleiten, wird dann in Abschnitt 7.1.3 beschrieben.

Kapitel 7

Das Ergebnis: Kiel-online

Ausgehend von der C-Applikation KIEL wurde mit der in den vorherigen Kapiteln beschriebenen Vorgehensweise ein Java-Applet entwickelt, das die in Kapitel 3 erarbeiteten Anforderungen erfüllt. In diesem Kapitel folgt nun eine detaillierte Beschreibung der Programm- und Bildschirmstruktur. Gleichzeitig werden die implementierten Lösungen für die aufgeworfenen Probleme wie Konfigurierbarkeit und geringe Fensterfläche dargestellt.

7.1 Implementierung

In der objektorientierten Programmierung hat der Begriff der Klasse eine fundamentale Bedeutung. Durch Klassen werden Objekte beschrieben und ihre Verhaltensweisen definiert. Die Struktur der Klassen und ihre Beziehungen zueinander stellen das gesamte entwickelte Applet dar. Es sollen in diesem Abschnitt alle entwickelten Klassen aufgeführt werden, danach wird auf die Hauptprogramm-Klasse `Kiel` gesondert eingegangen. Im letzten Abschnitt wird beschrieben, wie das entstandene Applet in die HTML-Seite eingebunden wurde.

7.1.1 Klassen

Wie bereits in Abschnitt 6.1.5 ausgeführt, bietet Java die Möglichkeit, Klassen in Paketen, also Mengen von Klassen, zu gliedern. Alle Klassen für das Applet KIEL-online sind in dem Paket `kiel` enthalten. Dieses Paket enthält die vier Unterpakete `cons`, `util`, `parse` und `gui`. Alle entwickelten Klassen, die zum Applet gehören, sind in einem dieser vier Pakete abgelegt. Zu jedem dieser Pakete erfolgt im weiteren eine Beschreibung, und die enthaltenen Klassen werden aufgeführt und erklärt.

`kiel.cons`
Dieses Paket enthält alle Interfaces, die aus Konstantendefinitionen bestehen.

Diese sind aus den gesamten `#define`-Präprozessoranweisungen der C-Applikation entstanden.

- `Konstanten`
  Hier sind alle Konstanten für die Token enthalten, die die Scanner benutzen.

- `PozrareuoKonstanten`
  Dieses Interface enthält vier Konstanten, die bei der Überprüfung von Kontextbedingungen von Rechenvorschriften und Ausdrücken benutzt werden.

- `Zahlenbereiche`
  In diesem Interface sind die Zahlenbereiche für die Typen *int* und *real* definiert.

- `Vorgaben`
  Dieses Interface beinhaltet alle Konstanten, die für die Auswertung von Ausdrücken benutzt werden. Hier ist beispielsweise festgelegt, wie lang eine Zeichenreihe oder eine Liste maximal sein darf. Es sind außerdem die Standartgrößen für die Anzeige der Werte innerhalb der Knoten definiert.

`kiel.parse`
In diesem Paket sind die Scanner und Parser für die Ausdrücke und Rechenvorschriften enthalten. Außerdem gibt es noch Hilfsklassen, die bei der Aufteilung von zu großen Klassendateien entstanden.

- `ParserVal`
  Die Klassen für semantische Werte der beiden Parser erben von dieser abstrakten Klasse.

- `AusdruckParserVal`
  Diese Klasse enthält semantische Werte, die der `AusdruckParser` benutzt. Sie erbt von `ParserVal`.

- `AusdruckParser`
  Der `AusdruckParser` ist eine vom Parsergenerator erzeugte Klasse, die Ausdrücke parsen kann.

- `AusdruckParserVariablen1`
  Diese Klasse, sowie die Klassen `AusdruckParserVariablen2` und `AusdruckParserVariablen3`, sind beim Aufteilen der zu großen Klasse `AusdruckParser` entstanden.

- `AusdruckParserVariablen2`

- `AusdruckParserVariablen3`

- `AusdruckParserKapsel`
  Diese Klasse erbt von `AusdruckParser` und kapselt sie. Sie stellt hauptsächlich eine Referenz auf den Scanner zur Verfügung, der die Token für den Parser generiert.

- `AusdruckYylex`
  Der generierte Scanner für Ausdrücke, der über die `AusdruckParserKapsel` mit dem Parser für Ausdrücke zusammenarbeitet, ist in dieser Klasse implementiert.

- `AusdruckYylexKapsel`
  Diese Klasse kapselt den Scanner für Ausdrücke.

- `ProgrammParserVal`
  Hier sind die semantischen Werte enthalten, die der `ProgrammParser` benutzt. Diese Klasse erbt von der abstrakten Klasse `ParserVal.`

- `ProgrammParser`
  Der `ProgrammParser` implementiert den Parser für Rechenvorschriften.

- `ProgrammParserVariablen1`
  Diese und die Klassen `ProgrammParser2` und `ProgrammParser3` entstanden nach dem Aufteilen des automatisch generierten Parsers für Rechenvorschriften.

- `ProgrammParserVariablen2`

- `ProgrammParserVariablen3`

- `ProgrammParserKapsel`
  Diese Klasse bildet eine Kapsel für den Rechenvorschriften-Parser.

- `ProgrammYylex`
  Der automatisch generierter Scanner für Rechenvorschriften ist in dieser Klasse enthalten.

- `ProgrammYylexKapsel`
  Ebenso wie der Scanner für Ausdrücke ist auch der Scanner für Rechenvorschriften gekapselt. Dies geschieht in dieser Klasse.

- `AdklVal`
  Diese und die folgenden Klassen erben alle von der Java-Klasse `Vector`. Sie werden von beiden Parsern benutzt, um den von den jeweiligen Scannern gelieferten Token Typen zuzuordnen. Die Klasse `AdklVal` steht für eine Liste aus Ausdrücken.

- `ArglVal`
  Diese Klasse erbt von der Java-Klassse `Vector`. In den einzelnen Komponenten stehen die Argumenten einer Rechenvorschrift. Es gibt nur die aus der Vaterklasse geerbten Methoden.

- `ElmlVal`
  In der unterstützten Sprache MlKiel gibt es zu jedem Basisdatentypen eine entsprechende Liste. Die einzelnen Elemente dieser Liste werden in dieser Klasse abgelegt.

- `ExprlVal`
  Parameterlisten von Rechenvorschriften werden als Listen von MlKiel-Ausdrücken in dieser Klasse abgelegt, die auch von der Klasse `Vector` erbt.

- `RvlVal`
  Der Parser von Rechenvorschriften legt eine Liste von Rechenvorschrifts-Systemen an. Diese werden in einem Objekt abgelegt, das zu dieser Klasse gehört.

Als Scannergenerator wurde eine Lex-Variante benutzt, die genau wie der C-Lexer arbeitet, aber eine Java-Klasse als Scanner generiert. Als Parsergenerator wurde Byacc verwendet, der genauso einen Java-Parser generiert. Leider sind diese beiden Tools nicht so gut wie Lex und Yacc für C aufeinander abgestimmt, deshalb mußten kleine Schnittstellen per Hand entwickelt werden. Diese sind dann als Kapseln um die jeweiligen Scanner und Parser implementiert.

`kiel.util`

Das Paket `kiel.util` bildet mit dem eben aufgeführten Paket `kiel.parse`, das die Scanner und Parser enthält, den Kern von KIEL-online. Hier sind alle Klassen für die benutzten Datenstrukturen enthalten, die für die Auswertung von Ausdrücken relevant sind. Es wird im folgenden nicht speziell auf die Attribute der einzelnen Klasse eingegangen. Da die Klassen nach der in Abschnitt 5.1 beschriebenen Vorgehensweise gebildet wurden, ist grundsätzlich aus jeder Variable des betreffenden Verbundtyps der C-Applikation KIEL ein Attribut der entsprechenden Klasse entstanden. Die Klassenbezeichner sind von den Bezeichnern der Verbundtypen abgeleitet, so lassen sich die Merkmale jeder Klasse aus dem Aufbau der jeweiligen Strukturen ableiten, die in [1] eingehend beschrieben werden.

Die Zuordnung der meisten Methoden zu den entsprechenden Klassen war relativ trivial, deshalb wird im allgemeinen nur zusammenfassend beschrieben, welches Verhalten die Klassen nach außen haben.

- `Argument`
  Diese Klasse implementiert Argumente von Rechenvorschriften und eine statische Methode zum Zugriff auf einzelne Elemente in Argumentlisten.

- `Ausdruck`
  Die Klasse `Ausdruck` stellt die gesamte Funktionalität zur Generierung eines Ausdrucks durch den `AusdruckParser` zur Verfügung.

- `AusdruckBaumKnoten`
  In den Java-Swing Bibliotheken gibt es eine Klasse für veränderbare Knoten in einem Baum, die Klasse `com.sun.java.swing.tree.DefaultMutableTreeNode`. Von dieser Klasse erbt die Klasse `AusdruckBaumKnoten`. Jeder Knoten kennt seine Vorgänger und Nachfolger, deshalb wurden alle Prozeduren, die im C-Programm auf die Baumstruktur eines Ausdrucks zurückgreifen, in dieser Klasse als Methoden implementiert. Außerdem sind hier Methoden zum Positionieren und Zeichnen des gesamten Baumes enthalten. Als Datenelement eines jeden Baumknotens wird die folgende Klasse `AusdruckBaumKnotenObjekt` benutzt.

- `AusdruckBaumKnotenObjekt`
  Diese Klasse bildet den Datenteil jedes Knotens im Ausdruckbaum. Hier sind alle Operationen implementiert, die auf die Daten eines einzelnen Knotens zurückgreifen, ohne daß Informationen über Vorgänger oder Nachfolger nötig sind.

- `Auswertung`
  Die Klasse `Auswertung` enthält die Hauptfunktionalitäten des Applets, also die Methoden zur Auswertung eines Ausdrucks und Attribute für Auswertungsdaten wie Expansionstiefe und Auswertungsanzahl.

- `Baumzeichner`
  Hier sind alle Methoden und Konstanten enthalten, die einen einzelnen Knoten zeichnen können. Alle Methoden in dieser Klasse sind statisch, also Klassenmethoden.

- `Kiel`
  Diese Klasse erbt von der abstrakten Klasse `kiel.gui.Applet` und steht für das Hauptprogramm. Es wird im Abschnitt 7.1.2 ausführlicher auf die genaue Struktur eingegangen.

- `Pozrareou_Rueckgabe`
  Aus dem ebenso lautenden Strukturtyp ist dieser Klasse hervorgegangen.

- `Protokoll`
  Die Klasse `Protokoll` erbt von `java.lang.Vector` und verwaltet eine Liste von Ausdrücken, in der sich beliebig hin- und zurückbewegt werden kann.

- `Rechenvorschrift`
  Hier wurde die Implementierung einer einzelnen Rechenvorschrift vorgenommen.

- `Rechenvorschriftliste`
  Diese Klasse erbt wiederum von der Java-Klasse für Listen und stellt die Funktionalität zum Einlesen von Rechenvorschriften (über den Parser) und den Zugriff auf einzelne Rechenvorschriften zur Verfügung.

- `RechenvorschriftNameSorte`
  Diese Klasse enthält den Namen und die Sorte einer `Rechenvorschrift`. Sie wird insbesondere beim Zusammenspiel von Ausdruckparser und der Rechenvorschriftliste benötigt.

- `Rechenvorschriftliste_liste`
  Auf diese Hilfsklasse wird beim Parsen einer Rechenvorschrift zugegriffen.

- `RetVal`
  In dieser Klasse wird der Wert eines Ausdrucks aufgenommen. Es werden hier außerdem die Tests auf mögliche Über- oder Unterschreitung der Bereichsgrenzen der Datentypen vorgenommen.

- `Unit`
  Diese Klasse enthält die Implementierung für einen Typen, dessen einziges Element `()` ist.

- `Warnung`
  Auf die Klasse `Warnung` greifen alle Methoden zurück, die Warnungen generieren. Es ist ein String-Feld enthalten, in dem Informationen über die Warnung abgelegt werden können. Die Warnungen werden immer dann verwendet, wenn eine Fehlerausgabe auf einem Ausgabemedium erfolgen muß. Im Gegensatz hierzu stehen die Ausnahmen, die auf niedrigeren Ebenen erzeugt werden und dann vom Programm aufgefangen werden. Ausnahmen, die nicht aufgefangen werden, werden auf der höchsten Ebene des Programms in Warnungen überführt und als solche dann ausgegeben.

- `WetSha`
  Diese Klasse enthält Werte, die bei der Positionierung der Knoten des Ausdruckbaumes durch den Algorithmus von Wetherell und Shannon (vgl. [1]) benötigt werden.

`kiel.gui`
Im letzten Paket sind alle grafischen Bedienelemente enthalten, d.h. alle hier aufgeführten Klassen erben von einer Java-AWT-Klasse, die für eine grafische Komponente steht. An dieser Stelle wird nur kurz ein Überblick über die verschiedenen Klassen gegeben. Eine weitere Beschreibung ihrer Funktionalitäten und ihres Aussehens folgt dann in Abschnitt 7.2.

- `Applet`
  Die Klasse `Applet` erbt von `java.applet.Applet` und stellt die gesamte grafische Oberfläche des Applets dar. Hier werden alle Ereignisse, die der Anwender auslöst, verarbeitet. Diese abstrakte Klasse hat eine relativ große Schnittstelle: Die abstrakten Methoden, über die auf Funktionalitäten der Datenebene zugegriffen werden, müssen von einer Kindklasse, in diesem Fall die Klasse `kiel.util.Kiel`, implementiert werden (vergleiche Abschnitt 7.1.2).

- `BaumFenster`
  Das `BaumFenster` ist eine Zeichenfläche innerhalb des Applets und erbt als solche von der Klasse `java.awt.Canvas`. Die Bäume, die für die Ausdrücke stehen, können hier gezeichnet werden. Ferner wird ein PopUp-Menü angeboten, über das die Aktionen auf den Knoten ausgeführt werden können.

- `ErrorMessage`
  Diese Klasse erbt von `java.awt.Dialog` und stellt ein Fenster zur Verfügung, in dem Fehlermeldungen angezeigt werden können.

- `KielFrame`
  Damit das Applet auch als lokale Applikation ausgeführt werden kann, muß für das Applet ein Container zur Verfügung gestellt werden, in dem es angezeigt werden kann. Diesen bietet diese Klasse, die von `java.awt.Frame` erbt. Außerdem ist hier eine Menüleiste integriert, über die ein Dateidialog zum Laden von Rechenvorschriften aufgerufen werden kann.

- `PreferenceDialog`
  Diese Klasse implementiert einen Java-Dialog, in dem der Anwender spezielle Einstellungen zur Oberfläche und Auswertung vornehmen kann

- `ViewChanger`
  Mit Hilfe dieses Dialoges können die Anzeigebereiche einzelner Knoten verändert werden.

7.1.2 Die Hauptprogramm-Klasse Kiel

Bei der Implementierung von KIEL-online wurde Wert darauf gelegt, den Anwendungscode vom Code der Bedienoberfläche zu trennen. Dadurch wird eine bessere Modularisierung erreicht und der Austausch oder die Erweiterung von Teilen des Programms wird erleichtert. Dies spiegelt sich unter anderem in der Realisierung der Klasse `Kiel` wider, die das Hauptprogramm bildet:

`Kiel` erbt von der abstrakten Klasse `kiel.gui.Applet`. In dieser Klasse ist die gesamte grafische Benutzeroberfläche realisiert, die einzelnen Ereignisse, die der Anwender erzeugt, werden hier soweit wie möglich bearbeitet. Außerdem enthält diese Klasse eine `main`-Methode, durch die das Applet auch als lokale Applikation ausgeführt werden kann.

Die Kommunikation des Applets mit der Hauptprogramm-Klasse erfolgt über abstrakte Methoden, die die Klasse `Kiel` implementiert. So kann dann das Applet beispielsweise einen eingegebenen Ausdruck parsen lassen und auf das Ergebnis des Parsens entsprechend reagieren. Durch diese Architektur ist das Hauptprogramm von der GUI-Oberfläche getrennt, dies ist im Hinblick auf die Weiterentwicklung von Java und dort speziell der grafischen Bibliotheken notwendig. Es muß für eine spätere Anpassung nur die Klasse `kiel.gui.Applet` verändert werden.

Die Klasse `Kiel` bildet, wie bereits erwähnt, das Hauptprogramm. Es enthält alle Variablen, die gemäß Abschnitt 5.1.13 in die Hauptprogramm-Klasse verschoben wurden. Die Attribute der Hauptprogramm-Klasse sind die folgenden:

```
(...)//Import-Anweisungen
public class Kiel extends kiel.gui.Applet
{
 public Vector   letzte_argumentliste;
 public String   programm_aktueller_rv_name;
 public int      programm_letzte_sorte;
 public RvlVal   bisherige_RVs;
 public boolean  beim_parsen_einer_argumentliste;
 public Rechenvorschriftliste_liste rv_name_sorte_liste_liste;
 public int      aktuelle_Anzahl_Auswertungsschritte=-1;
 public int      aktuelle_Auswertungsart=
                 AUSWERTUNG_LEFTMOST_INNERMOST;
 public Ausdruck              myAusdruck;
 public Auswertung            myAuswertung;
 public Rechenvorschriftliste myRechenvorschriftliste;

 public static java.awt.Color ausgewertetBaumFarbe;
 public static java.awt.Color RVBaumFarbe;
 public static int AktVORGABE_NACHKOMMASTELLEN=
                       VORGABE_NACHKOMMASTELLEN;
 public static int AktVORGABE_ANZAHLZEICHENIMSTRING=
                       VORGABE_ANZAHLZEICHENIMSTRING;
 public static int AktVORGABE_ANZAHLELEMENTEINLISTE=
                       VORGABE_ANZAHLELEMENTEINLISTE;
 public static int AktABBRUCH_EXPANSIONS_TIEFE=
                       ABBRUCH_EXPANSIONS_TIEFE;
 public static int AktMAX_ANZUZEIGENDE_EBENEN_ANZ=
                       MAX_ANZUZEIGENDE_EBENEN_ANZ;
 (...)//Methoden
}
```

Es werden in der Klasse `Kiel` zwei verschiedene Arten von Attributen benutzt. Die Attribute im ersten Abschnitt sind ehemalige globale Variable des C-Programms, über die jeweils Scanner und Parser miteinander kommunizieren und die die Auswertung charakterisieren. Außerdem gibt es die Attribute, die den Kern des Programms bilden: eine Liste von Rechenvorschriften, einen Ausdruck und die Klasse, die für die Auswertung des Ausdrucks zuständig ist. Alle diese Attribute sind öffentlich, damit alle Klassen, denen eine Instanz von `Kiel` übergeben wird, sie setzen und lesen können. Der zweite Block besteht aus den statischen (Klassen-) Variablen, die für alle Instanzen der Klasse `Kiel` gleich sind. Jede Klasse kann über den Klassennamen `kiel.util.Kiel` auf diese zugreifen. Bei diesen statischen Variablen handelt es sich insbesondere um Konfigurationseinstellungen, die der Anwender in einem Dialog machen kann. Die Einstellungen wirken sich dann auf alle auf dem Rechner laufenden KIEL-online–Applets und Applikationen aus.

Als Methoden enthält die Klasse `Kiel` einige Initialisierungen, in denen jeweils eine Instanz der Klassen `Ausdruck`, `Rechenvorschriftliste` und `Auswertung` erzeugt wird. Außerdem werden noch einige statische Variablen wie Knotenfarben gesetzt. Die weiteren (öffentlichen) Methoden gehören zur Schnittstelle, die die Klasse `Kiel` implementieren muß, da sie von `kiel.gui.Applet` erbt. Die Schnittstelle in der abstrakten Applet-Vaterklasse umfaßt folgende Methoden:

```
public abstract Rechenvorschriftliste getRechenvorschriftliste();
public abstract Warnung addRechenvorschriften(String text);
public abstract java.util.Vector printRechenvorschriften();

public abstract Ausdruck getAusdruck();
public abstract Warnung addAusdruck(String text);

public abstract Warnung forwardHistory (BaumFenster tafel);
public abstract Warnung backwardHistory(BaumFenster tafel);

public abstract Auswertung getAuswertung();
public abstract Warnung setAuswertungsart(int i);
public abstract void setAuswertungsAbbruch(boolean val);
public abstract int getAktuelle_Anzahl_Auswertungsschritte();
public abstract void setAktuelle_Anzahl_Auswertungsschritte(int i);
```

Dabei wird durch die Methoden im ersten Abschnitt ein Zugriff auf die Rechenvorschriftliste realisiert, im zweiten Abschnitt sind Methoden zum Zugriff auf den Ausdruck enthalten. Die Methoden `forwardHistory(...)` und `backwardHistory(...)` werden aufgerufen, wenn ein Undo oder Redo vorgenommen werden soll. Änderungen des Anwenders an der Auswertung werden durch die Methoden im letzten Abschnitt zur Datenebene hin weitergeleitet. Die Klasse `Kiel` implementiert nun diese Methoden, stellt also jeweils einen entsprechenden Methodenrumpf zur Verfügung und veröffentlicht sie.

7.1.3 Die HTML-Seite

In Abschnitt 6.2.3 wurde bereits erwähnt, daß nach einer Größenänderung des Web-Browsers nicht bei jedem Browser auch die Größe des Applets angepaßt wird. Um dieses Problem zu lösen, wird mit Hilfe von *JavaScript*-Befehlen auf der HTML-Seite, die vor der Applet-Seite besucht wird, nach Browsertypen unterschieden. Dann werden für den Internet Explorer und den Netscape Navigator unterschiedliche Applet-Seiten aufgerufen.

- Seite für den Internet Explorer

  ```
  <html><head><title>Kiel Interactive Evaluation
  Laboratory - Online </title>
  <script language="JavaScript">
  <!--
  alert("The Java Applet is loading and
  initializing now.  Please wait.");
  //-->
  </script></head>
  <applet code="kiel.util.Kiel.class" archive="kiel.jar"
  name="kiel" width=100% height=100%>
  </applet></html>
  ```

- Seite für den Netscape Navigator
  Diese HTML-Seite unterscheidet sich nur durch die Einbindung des Applet-Tags, der gesamte Rest blieb unverändert. Deshalb wird hier nur die Veränderung dokumentiert:

  ```
  <body onResize=
  "document.layers['kiel'].document.applets['kiel'].resized(
  document.layers['kiel'].clip.width,
  document.layers['kiel'].clip.height)">
  <span id="kiel" style="position:absolute;top:0;left:0">
  <applet code="kiel.util.Kiel.class" archive="kiel.jar"
  name="kiel" width=100% height=100%>
  </applet></span></body>
  ```

  In der Version für den Netscape Navigator muß explizit eine Größenänderung des Browsers an das Applet weitergeleitet werden. Dies erfolgt durch den Aufruf der Methode `resized(int,int)` der Klasse Kiel. Diese bewirkt, daß das Applet auf die übergebene Breite und Länge angepaßt wird.

Es gibt bei dieser Lösung allerdings noch ein Problem: Beide Browser zeigen nach der Größenänderung nur den Bereich des Applets vergrößert an, der beim ersten Initialisieren schon angezeigt wurde. Deshalb sollte man vor jedem Start des Applets das Browserfenster auf die volle Bildschirmgröße bringen.

7.2 Bedienoberfläche

In diesem Abschnitt werden die grafische Oberfläche von KIEL-online vorgestellt und insbesondere die Unterschiede zur C-Applikation, also dem Originalsystem, besprochen. Der größte Unterschied ist die Realisierung des Hauptteils der Bedienoberfläche im Browserfester. Anders als bei der C-Applikation sind hier alle Bedienelemente, die bei einer Standartnutzung nötig sind, in diesem einen Fenster untergebracht. Nur für besondere Zwecke wie Fehlerausgabe oder Konfiguration werden separate Dialogfenster benutzt. Diese Einschränkung im Vergleich zur Applikation KIEL wurde aus Performance-Gründen vorgenommen (vergleiche auch Abschnitt 2.2.1).

7.2.1 Browserfenster

Das Browserfenster ist der Kern des Applets. Die Eingabe von Rechenvorschriften und Ausdrücken, das Parsen, die Auswertung und alle Standarteinstellungen sind hier realisiert. Abbildung 7.1 zeigt die Oberfläche des Applets KIEL-online im Netscape Navigator. Die Oberfläche besteht grob gesagt aus drei Bereichen: Im oberen Drittel befinden sich die Eingabefelder für Rechenvorschriften und Ausdrücke, der untere Bereich des Applets teilt sich in das Baumfenster, in dem der auszuwertende Ausdruck angezeigt wird, und in die Bedienelemente auf der rechten Seite, die zur Kontrolle der Auswertung dienen.

- **Eingabebereich**

  Im Eingabebereich läßt sich eine Liste von Rechenvorschriften und ein Ausdruck eingeben. Dazu werden die Rechenvorschriften auf der linken Seite in das Textfeld eingegeben oder per copy-and-paste (vergleiche Abschnitt 3.3) aus einem Texteditor übernommen. Alle Rechenvorschriften, die sich in dem Textfeld befinden, werden durch die Betätigung des Buttons “>>” geparst. Tritt kein Fehler auf, so werden danach die Bezeichner und Funktionalitäten aller gerade geparsten Rechenvorschriften in dem Übersichtsfenster rechts neben dem Textfenster angezeigt. Wählt man im Übersichtsfenster eine Rechenvorschrift aus, so springt der Cursor im linken Textfenster automatisch auf die selektierte Rechenvorschrift.

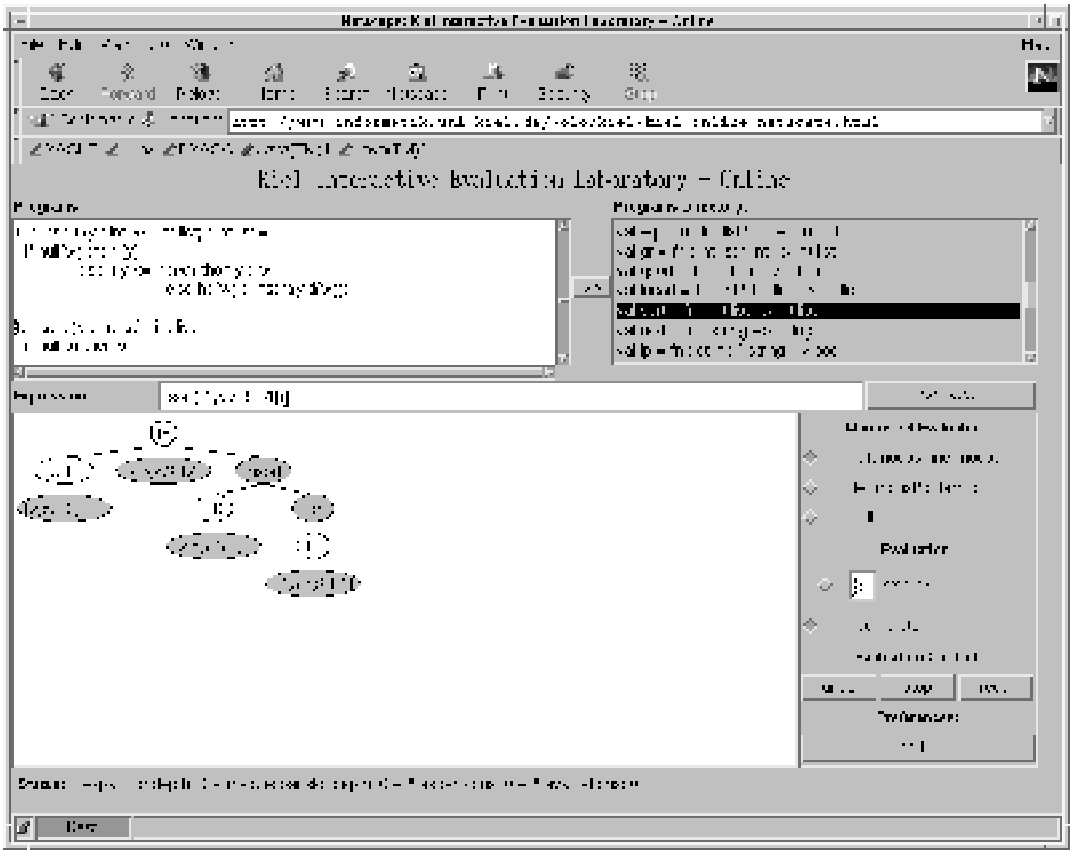


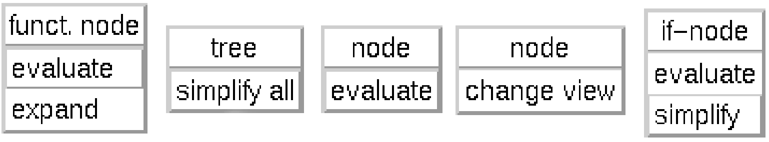
Abbildung 7.1: Das Browserfenster

Vor dem Parsen der Rechenvorschriften werden alle bereits geparsten Rechenvorschriften gelöscht, nach dem Parsen werden alle übernommen. Tritt ein Fehler auf, so erfolgt eine Ausgabe des Fehlers durch einen Fehlerdialog (vgl. Abschnitt 7.2.3). Es wird keine Rechenvorschrift, auch nicht eine möglicherweise fehlerfreie, übernommen. Diese Vorgehensweise wurde gewählt, um das Verhalten des Applets der C-Applikation anzupassen, in der Dateien, die Rechenvorschriften enthalten, als Ganzes geparst werden. Es gibt auch hier keine Möglichkeit eines Parsens, in dem neue Rechenvorschriften zu bereits geparsten hinzugefügt werden.

Anders als die C-Applikation kann das Applet nur immer mit einem Ausdruck arbeiten. Dadurch wurde ein weiterer Anzeigebereich, in dem eine Übersicht aller eingegebenen Ausdrücke angezeit werden müßte, eingespart. Diese Vereinfachung schien angebracht, da dadurch die Bedienoberfläche klein gehalten und der Funktionsumfang nicht zu sehr beschnitten wird. Der Ausdruck läßt sich unterhalb der Rechenvorschriften eingeben. Durch Betätigung des Buttons rechts neben dem Textbereich oder durch Betätigen der <Return>-Taste wird der Ausdruck geparst. Falls dies fehlerfrei geschieht, wird er dann im Baumfenster angezeigt, im anderen Fall erfolgt eine Fehlermeldung.

- **Baumfenster**

  Nachdem der Ausdruck geparst wurde, wird er im Baufenster angezeigt. Hier kann er nun schrittweise ausgewertet werden. Anders als bei der C-Applikation wird die Operation, die auf den Knoten angewendet werden soll (expandieren, auswerten, vereinfachen), nicht durch verschiedene Maustasten ausgewählt. Dies ist in Java nicht möglich, da aus Gründen der Plattformunabhängigkeit nur eine einzige Maustaste abfragbar ist. Stattdessen erscheint nach Betätigen einer Maustaste auf dem Knoten ein PopUp-Menü, über das dann die gewünschte Operation ausgeführt wird. Die verschiedenen Einträge des PopUp-Fensters sind im folgenden hintereinander aufgeführt:



  Wie hier zu sehen ist, wird der Inhalt des PopUp-Menüs immer dem Kontext des ausgewählten Knotens angepaßt. So sind dann Knoten, die einen Funktionsaufruf enthalten, expandierbar und auswertbar (siehe erstes PopUp "func. node"). Klickt man im Baumfenster auf einen Bereich, in dem sich kein Knoten befindet, so wird das PopUp-Menü "tree" angezeigt. Der gesamte Baum läßt sich damit soweit wie möglich vereinfachen. Knoten, die eine Operation auf den Basisdatentypen darstellen, werden durch das Menü "node" ausgewertet. Bei Knoten, die schon ausgewertet sind, kann der Darstellungsbereich verändert werden. Knoten, die für eine `if`-Verzweigung stehen, können durch das PopUp-Menü "if-node" vereinfacht oder ausgewertet werden.

Wie in Abschnitt 3.6 bereits erwähnt, sollte es möglich sein, das Baumfenster beliebig vergrößern zu können. Dazu kann es durch eine Option im Preference-Dialogfenster von der Oberfläche des Browserfensters abgelöst und dann als eigenständiges Fenster anzeigen werden (vergleiche Abschnitt 7.2.2). Alle Funktionen bleiben dabei voll erhalten.

- **Auswertungskontrolle**

  Rechts neben der Zeichenfläche, in der das Baumfenster angezeigt wird, befinden sich die Kontrollen für die Auswertung. Diese unterteilen sich in verschiedene Bereiche: Der erste Bereich besteht aus einer Gruppe von 3 Kontrollkästchen, mit denen sich die Auswertungsstrategie zwischen leftmoust-innermoust, leftmoust-outermoust und full–Ersetzung festsetzen läßt.

  Im zweiten Bereich kann die Schrittweite der Auswertung eingestellt werden. Hier wird über Kontrollkästchen komplette Auswertung eines Knotens ausgewählt, oder die gewünschte Anzahl der hintereinander ausgeführten Auswertungsschritte spezifiziert.

  Der letzte Bereich stellt eine Undo/Redo-Funktion und eine Taste zum Auswertungsabbruch zur Verfügung. Dadurch kann der Benutzer Bearbeitungen von Knoten rückgängig machen und ebenso die zurückgenommenen Bearbeitungen wiederherstellen. Es kann sich also über alle Stufen der Auswertung frei bewegt werden. Im Unterschied zur C-Applikation wurde aber hier aus Platzmangel darauf verzichtet, unterschiedliche Schrittweiten für das Undo/Redo anzubieten.

  Außerdem enthält der beschriebene Bereich noch einen Button, über den ein Dialog für benutzerspezifische Einstellungen aufgerufen werden kann. Dieser Dialog wird im nächsten Abschnitt beschrieben.

7.2.2 Preference-Dialogfenster

Wird im Browserfenster der Button "edit..." unter der Überschrift "Preferences" gedrückt, so wird ein Dialogfenster angezeigt, in dem der Benutzer eigene Konfigurationseinstellungen vornehmen kann. Abbildung 7.2 stellt dieses Fenster dar.

In diesem Fenster lassen sich Einstellungen zur Oberfläche des Applets machen, es können Grundeinstellungen zur Knotenanzeige gemacht werden und Abbruchwerte für die Auswertung eines Ausdrucks gesetzt werden.

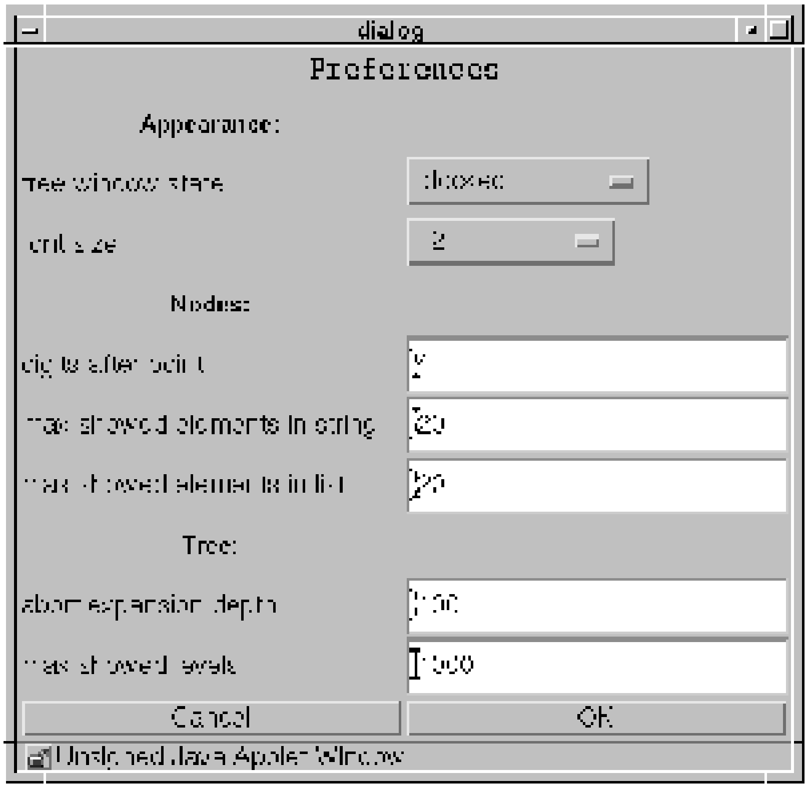


Abbildung 7.2: Der Preference-Dialog

Unter der Rubrik “Appearance” kann das Baumfenster von der Oberfläche des Applets abgekoppelt und dann als eigenes Fenster angezeigt werden. Dadurch können auch umfangreiche Ausdrücke ausgewertet werden, da das Baumfenster als eingenes Fenster beliebig vergrößerbar ist. Das losgelöste Baumfenster zeigt die Abbildung 7.3. Weiter kann die Schriftgröße für das gesamte Applet eingestellt werden, um das Arbeiten mit KIEL-online auch auf verschiedenen Bildschirmauflösungen angenehm zu machen.

Ausgewertete Knoten können unter anderem reelle Zahlen, Strings oder Listen enthalten. Unter der Rubrik “Nodes” können für diese Knoten die gewünschten Anzeigebereiche eingestellt werden. Bei reellen Zahlen kann die Zahl der Nachkommastellen angegeben werden, für den Typ `String` kann die maximal angezeigt Länge einer Zeichenkette ausgewählt werden, bei Listen die maximale Anzahl von Elementen, die in einer Liste angezeigt werden. Anders als bei dem Fenster für Knotenanzeigen wirken sich Änderungen dieser Werte auf alle angezeigten Knoten aus.

In der letzten Rubrik ist die Expansionstiefe einstellbar, bei der die Auswertung im Baumfenster automatisch abbrechen soll (wichtig für nicht-terminierende Rechenvorschriften) und die Anzahl der maximal anzeigbaren Ebenen im Baumfenster. Gemachte Änderungen in diesem Dialogfenster werden erst wirksam, wenn der “OK”-Button gedrückt wird.

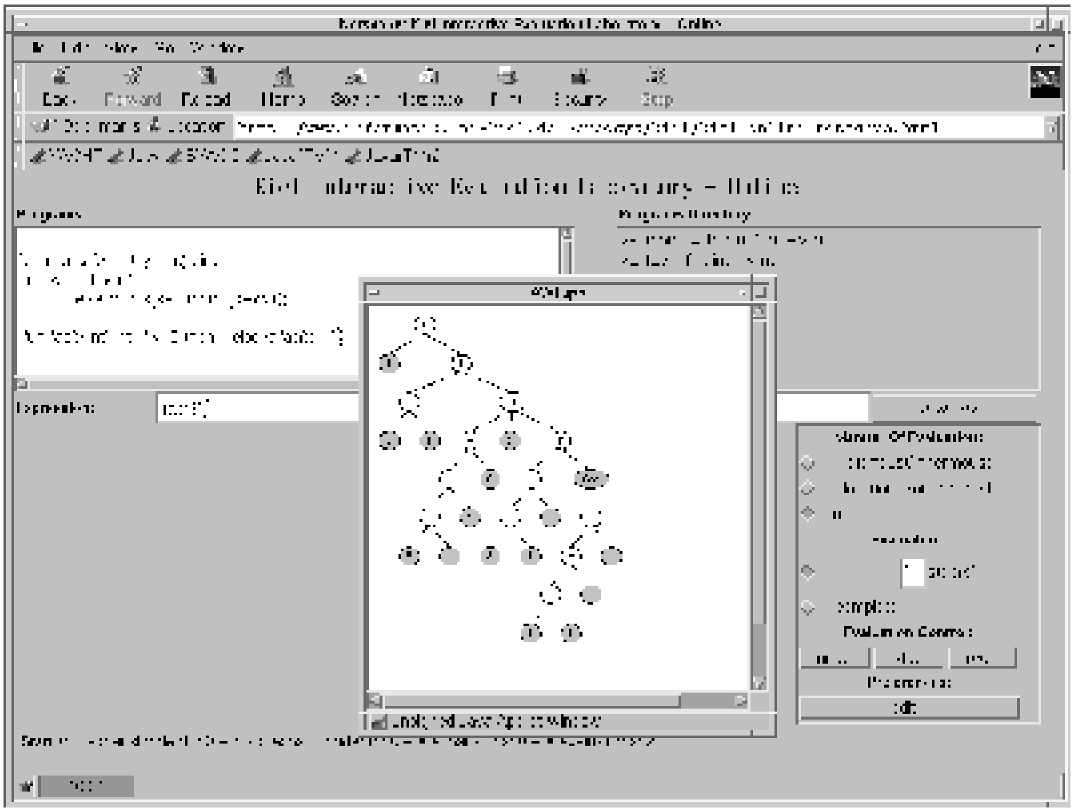


Abbildung 7.3: Losgelöstes Fenster für die Auswertung eines Ausdrucks

KIEL ist durch Ressource-Dateien in weiten Teilen vom Benutzer frei konfigurierbar (vergleiche Abschnitt 2.3 und Diplomarbeit [1]). Der beschriebene Konfigurations-Dialog, der dem Benutzer des Applets zur Verfügung steht, ist natürlich nicht so mächtig wie die Konfiguration durch Dateien bei der C-Applikation. Es wurden für den Preference-Dialog alle Eigenschaften des Applets ausgewählt, bei denen eine Veränderung durch den Nutzer wirklich notwendig sein kann. Andere Einstellungen, die ein Nutzer der Applikation zusätzlich hat, wurden hier weggelassen. Neben den Nachteilen durch diese eingeschränkte Konfigurierbarkeit sollte aber erwähnt werden, daß es in einigen Fällen auch sinnvoll sein kann, *während* der Laufzeit gewisse Eigenschaften, wie beispielsweise Auswertungsgrenzwerte, festzulegen.

7.2.3 Fenster für Fehlermeldungen

Abbildung 7.4 zeigt das Fenster, in dem Fehler angezeigt werden, die beim Parsen oder bei der Auswertung eines Ausdrucks entstehen.

Im Text wird der aufgetauchte Fehler erklärt, es gibt jeweils nur einen Button “OK” zum quittieren der Fehlermeldung.

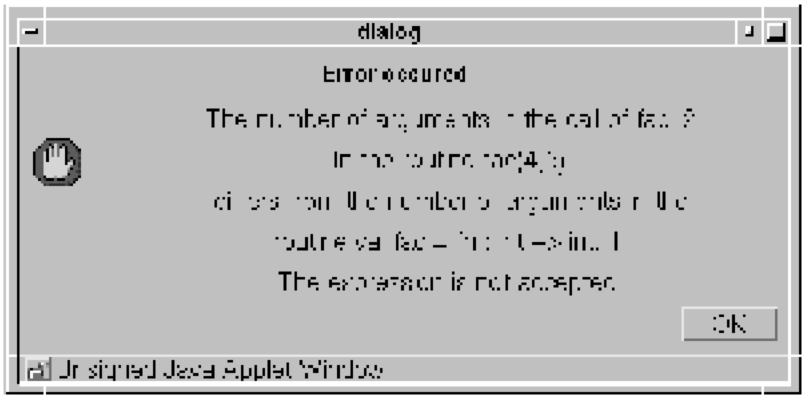


Abbildung 7.4: Fenster für Fehlermeldungen

7.2.4 Fenster für Knotenanzeigen

Wird auf einem bereits ausgewählten Knoten eine Maustaste betätigt, und dann "change view" im erscheinenden PopUp-Menü ausgewählt, so wird der in Abbildung 7.5 dargestellte Dialog angezeigt.

Es läßt sich bei Knoten, die eine reelle Zahl enthalten, die Anzahl der Nachkommastellen einstellen, für Listen und Strings die gewünschte maximale Anzeigelänge. Die Änderungen, die durch diesen Dialog vorgenommen werden, wirken sich aber nur auf den Knoten aus, der mit der Maus identifiziert wurde.

Abbildung 7.5: Der Dialog zum Ändern der Anzeige in einem Knoten

Kapitel 8

Ausblick

Im Verlauf der nächsten Jahre wird die C-Applikation sicherlich erweitert und eine größere Teilmenge von ML unterstützen. Es ist denkbar, daß die Weiterentwicklung des Applets parallel dazu fortgeführt wird. In diesem Kapitel werden unter anderem diese Erweiterungen und speziell deren Folgen für das Applet und dessen Struktur besprochen. Außerdem muß das Voranschreiten der Internet- und Hardwaretechnologie berücksichtigt werden. In diese beiden Richtungen sollen im folgenden einige Ausblicke getätigt werden.

8.1 Vergrößerung der akzeptierten Sprache

Während des Verfassens dieser Arbeit wurde schon begonnen, die vom KIEL-System unterstützte Sprache MlKiel (vergleiche 2.1) um zusammengesetzte Datentypen und Rechenvorschriften, die mittels Musteranpassung definiert sind, zu erweitern. Eine entsprechende Diplomarbeit ist zum Zeitpunkt der Erstellung dieser Arbeit bereits ausgegeben worden.

Es ist möglich, auch diesen und weitere Erweiterungsschritte auf das Applet in der bereits geschilderten Weise (vergleiche Kapitel 4 und 5.3) zu übertragen. Dazu werden wohl hauptsächlich die Scanner und Parser für Rechenvorschriften und Ausdrücke erweitert werden.

Die jetzigen Versionen von Scanner und Parser machen bereits mehr als ein Drittel der Größe des Applets aus, dies sind ca. 200 KiloByte. Wird die Sprache weiter ausgebaut, werden sich besonders die zwei Parser stark vergrößern. Dies ist aber nicht wünschenswert, da hierdurch noch mehr Applet-Code bei jedem Start des Applets aus dem Netz geladen werden müßte. Einen Ausweg bietet die Möglichkeit, die Scanner und Parser für Rechenvorschriften und Ausdrücke auf einen Server auszulagern. Die folgenden beiden Abschnitte befassen sich mit dieser Fragestellung.

8.1.1 Remote Methode Invocation

Remote Method Invocation (RMI) ist eines der Konzepte, die in Java zur Verfügung stehen, um verteilte Systeme zu implementieren (andere sind beispielsweise Sockets und Schnittstellen zu CORBA). RMI verfolgt das gleiche Konzept wie die Remore Procedure Calls in prozeduralen Sprachen. Durch RMI lassen sich Methoden von Objekten aufrufen, die sich in einem anderen Prozess (möglicherweise auf einer anderen Maschine) als der Aufrufer befinden. Um auf Objekte mit RMI zuzugreifen, sind folgende Schritte nötig:

- Es muß eine Schnittstelle definiert werden, die angibt, auf welche Methoden andere Prozesse zugreifen dürfen
- Ein RMI-Server muß implementiert werden, dies geschieht, indem man von einem geeigneten RMI-Server des Paketes `java.rmi.server` erbt.
- Auf der Klassen-Datei dieses Servers wird der `rmic`-Compiler, der in jeder neueren Java-Version enthalten ist, ausgeführt. Der `rmic`-Compiler generiert zwei weitere Klassen: Die eine Klasse sorgt auf der Serverseite für eine RMI-Verbindung, die andere fungiert clientseitig als Proxy-Server.
- Als letztes wird der neue Server in die RMI registry aufgenommen und der Port definiert, über den er referenziert werden kann.

Applikationen können auf beliebige Server im Internet zugreifen, für Applets gilt jedoch die Restriktion, daß nur auf RMI-Server zugegriffen werden darf, die auf dem gleichen Rechner wie das Applet abgelegt sind.

8.1.2 RMI und KIEL-online

Um den Code des Applets insbesondere bei Erweiterungen der akzeptierten Sprache nicht weiter zu vergrößern, ist es denkbar, die Parser und Scanner von dem Rest des Applets zu trennen und auf einen Server auszulagern. Es kann dazu ein RMI-Server implementiert werden, der die eingegebenen Zeichenreihen für Rechenvorschriften und Ausdrücke vom Applet zugeschickt bekommt, sie parst, die entspechenden Klassen generiert und zum Applet zurückschickt.

Es könnten dann alle Klassen des Paketes `kiel.parse` und die Klasse `Rechenvorschriftliste_liste` des Paketes `kiel.util` ausgelagert werden. Die Klassen `Rechenvorschriftliste` und `Ausdruck` müßten sowohl auf Client- als auch auf Serverseite zur Verfügung stellen. Dadurch ließe sich heute schon die Größe der gepackten Applet-Datei um fast 70% verkleinern (von ca. 480 auf 150 Kilobyte).

Es gibt aber auch Nachteile einer verteilten Realisierung: Jedes Parsen eines Ausdrucks oder einer Rechenvorschrift würde eine neue Verbindung mit dem RMI-Server notwendig machen, außerdem können die Instanzen der übertragenen Klassen `Rechenvorschriftliste` und `Ausdruck` sehr groß werden, dadurch würde es zu einer weiteren Verzögerung kommen. Auch muß die Auslastung des RMI-Servers und das System, auf dem er läuft, berücksichtigt werden.

Zusammenfassend läßt sich sagen, daß man nicht ohne umfassende Tests klären kann, ob sich die Performance des Systems durch RMI bei durchschnittlichem Anwendungsszenario erhöhen würde.

8.2 Technologische Auswirkungen

Im vorigen Abschnitt wurden Änderungen der Architektur und Implementierung besprochen, die durch die Erweiterung des Systems verursacht werden. Durch absehbaren technologischen Fortschritt ändern sich aber auch die Rahmenbedingungen für das Applet. Diese neuen Aspekte sollen im folgenden kurz betrachtet werden.

8.2.1 Internet und Browser

Es ist damit zu rechnen, daß viele Anwender in Zukunft über schnelle Internetverbindungen durch ISDN oder andere, noch effizientere Techniken, verfügen. Darauf können dann entsprechend auch die Inhalte im Internet angepaßt werden. So ließen sich wesentlich umfangreichere Applets implementieren, die dann den gleichen Funktionsumfang wie lokale Applikationen hätten. In dieser Richtung ließe sich dann auch KIEL-online um weitere Funktionalitäten erweitern.

Mit dem Voranschreiten der Java-Technologie werden neue Bibliotheken wie die JFC (Java Foundation Classes) mit allen Swing-Komponenten eine große Verbreitung finden und möglichweise bald bei den neuen Web-Browsern standartmäßig enthalten sein, wie dies heute schon eine Java 1.1-Version ist. Ist dies der Fall, ließen sich grafisch sehr hochwertige Benutzeroberflächen auch für Applets realisieren. Da die GUI-Ebene von KIEL-online von der Anwendungsebene gekapselt ist, ist eine Neuentwicklung der Oberfläche und ihre Integrierung in die bestehende Architektur unproblematisch.

8.2.2 Signieren

Applets lassen sich mit Hilfe eines privaten Schlüssels signieren, diese Signatur wird dann vor dem Starten des Applets vom jeweiligen Browser überprüft. Wird ein signiertes Applet akzeptiert, so kann es dann unter anderem auch auf die Festplatte des lokalen Rechners zugreifen. Es soll hier kurz beschrieben werden, wie Applets signiert werden.

Der Autor eines Applets muß sich bei einer Zertifizierungsbehörde wie *VeriSign* digitale IDs kaufen. Diese bestehen aus einem öffentlichen und einem privaten Schlüssel. Es muß für beide Browsertypen, also für den Internet Explorer und den Netcape Navigator, jeweils eine eigene digitale ID gekauft werden, die zur Zeit der Erstellung dieser Diplomarbeit jährlich 400$ kostet. Mit Hilfe der erworbenen Schlüssel und einem Tool zum Signieren, *signtool*, kann dann ein ganzes Archiv signiert werden. Ein signiertes Applet wird auf dem Rechner, auf dem es ausgeführt werden soll, mit Hilfe des öffentlichen Schlüssels des Autors verifiziert. Ist die Überprüfung erfolgreich, hat das Applet alle Zugriffsmöglichkeiten auf Systemressourcen, die auch ein lokales Applet hat. Mit Hilfe der Zertifizierungsinstanz läßt sich bei "bösen" Applets der Autor des Applets, ausgehend vom öffentlichen Schlüssel, eindeutig bestimmen.

In der Wirtschaft wird das sichere Übertragen von Daten über das Internet und damit das Signieren von Daten immer wichtiger. Es ist deshalb wohl zu erwarten, daß sich die Zahl der Zertifizierungsstellen (derzeit 2) erhöhen wird und es in Zukunft auch preiswertere Möglichkeiten als die erwähnten digitalen ID's der heutigen Zertifizierungsstellen gibt. Dann wäre das Signieren von KIEL-online eine sinnvolle Möglichkeit, da dadurch auch Systemzugriffe möglich sind.

8.3 Abschluß

Die Entwicklung der Java-Sprache wird weiter voranschreiten. Damit einhergehend wird auch die Verbreitung von neuen Java-Technologien wie Swing und *Java Beans* (die Java-Komponententechnologie) weiter zunehmen. Es ist denkbar, daß größere Applets für wissenschaftliche Anwendungen oder als Entwicklungstools sich in den nächsten Jahren vermehrt durchsetzen, und daß aus dem heutigen Demo-Charakter vieler Applets ein vollständiger Ersatz für stand-alone–Anwendungen wird.

Applets haben Vorteile gegenüber herkömmlichen Applikationen. Neben einigen bereits erwähnten Eigenschaften wie Plattformunabhängigkeit und eingebautem Sicherheitskonzept sollen hier noch die Vorteile für den Anbieter der Software und speziell für seine Support-Abteilung angesprochen werden, die daraus resultieren, daß das Applet von Anwendern aus dem WWW heruntergeladen wird:

- Es müssen clientseitig keine besonderen Software-oder Hardwarevoraussetzungen erfüllt sein, sondern nur ein neuerer Internet-Browser vorhanden sein.
- Es gibt keine Probleme bei der Inbetriebnahme, da jeglicher Installationsaufwand wegfällt.
- Die gesamte Wartung eines Applets und die Einführung neuer Versionen läßt sich vollständig serverseitig durchführen.

Diese Vorteile ergeben sich selbsverständlich nicht nur für Internet-Anwendungen. Insbesondere sind Extranets und schnelle Intranets betroffen, die verschiedene

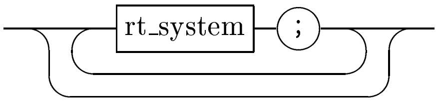
Rechnerplattformen verbinden. Hier werden Java-Applets möglicherweise innerhalb der nächsten zwei bis drei Jahre kleinere lokale Applikationen verdrängen.

Anhang A

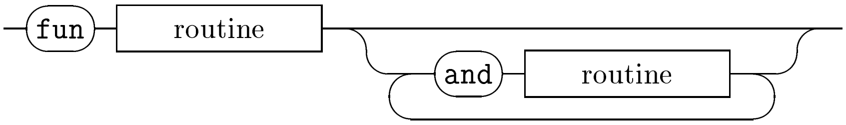
Syntaxdiagramme

A.1 Aufbau von Programmen

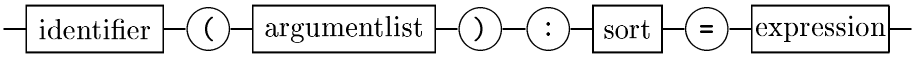
rtprogramm



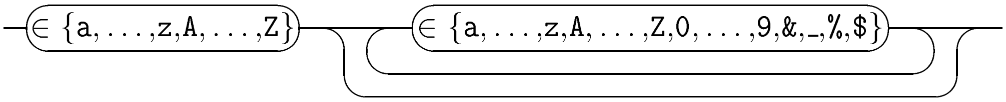
rt_system



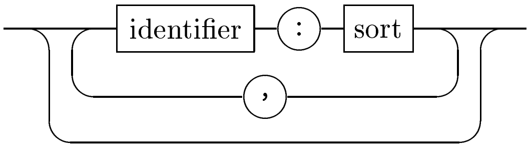
routine



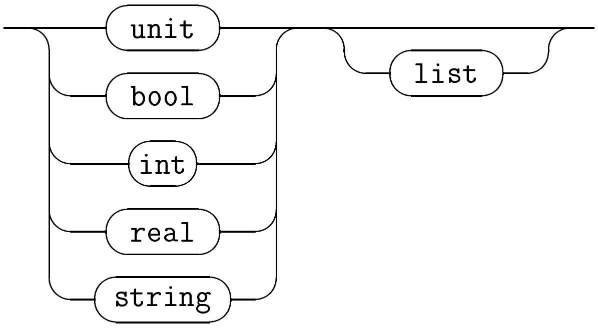
identifier



argumentlist



sort



*unit_level*1

*bool_level*3

*int_level*3

*real_level*3

*string_level*2

*unit_list_level*3

bool_list_level3

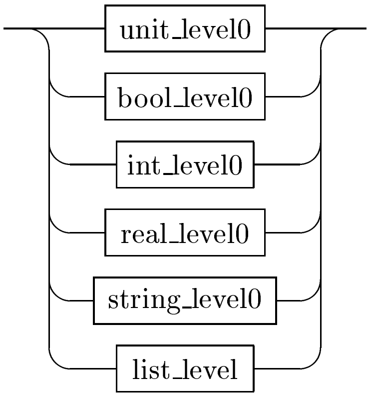
int_list_level3

real_list_level3

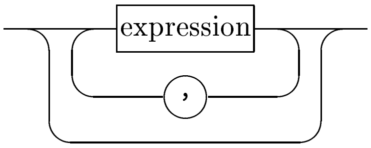
string_list_level3

A.2 Aufbau von Ausdrücken

expression

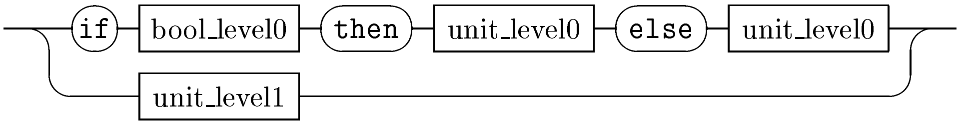


expressionlist

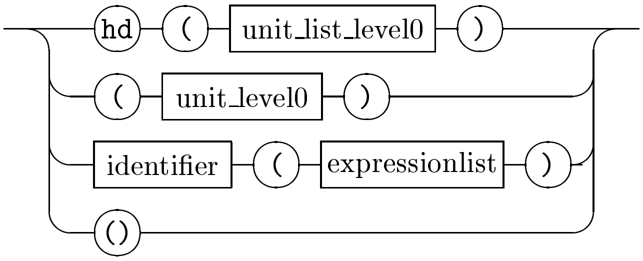


A.2.1 Aufbau von `unit`-Ausdrücken

unit_level0

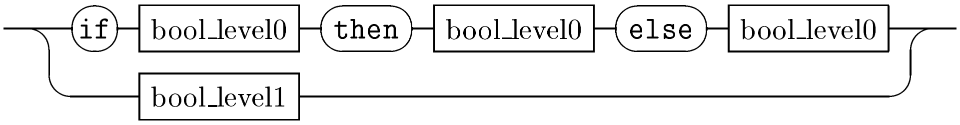


unit_level1



A.2.2 Aufbau von `bool`-Ausdrücken

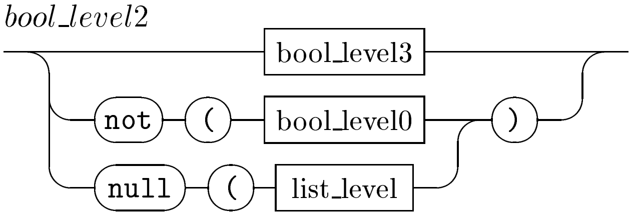
bool_level0



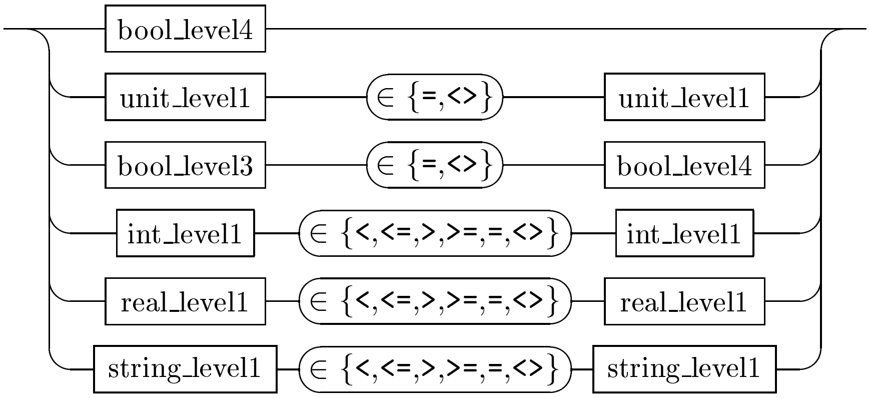
bool_level1

bool_level1_5

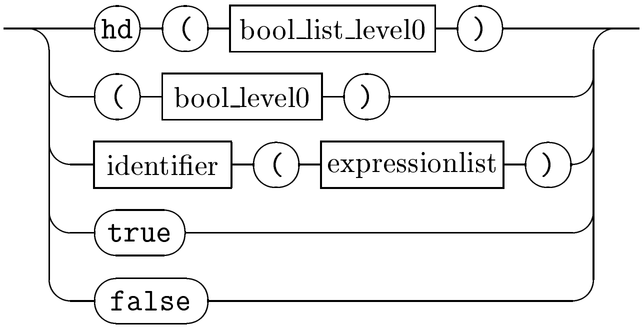
bool_level2



bool_level3

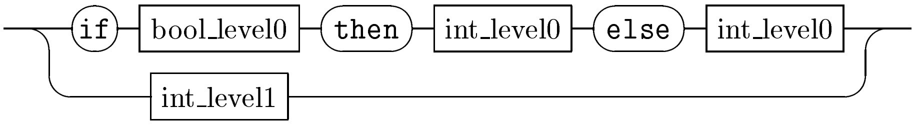


bool_level4

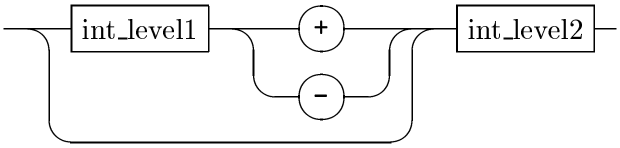


A.2.3 Aufbau von `int`-Ausdrücken

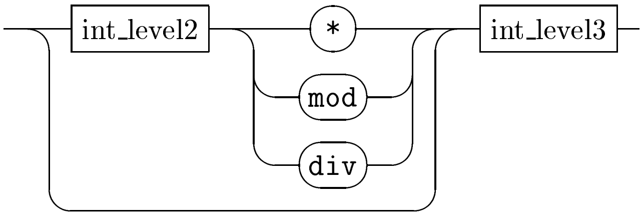
int_level0



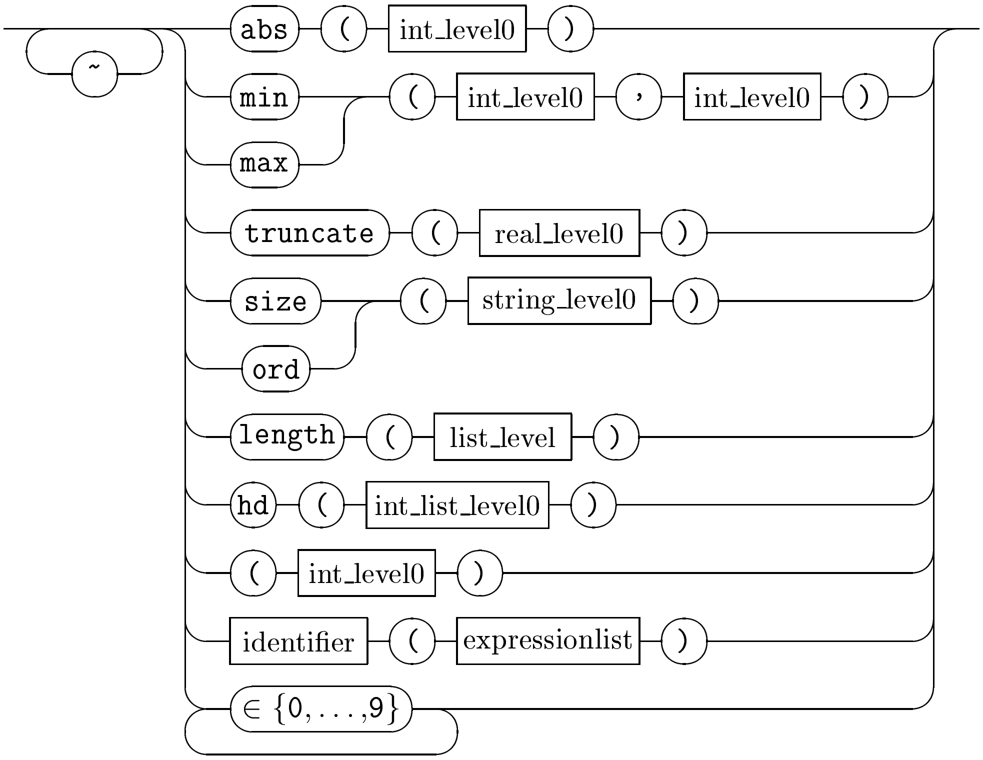
int_level1



int_level2

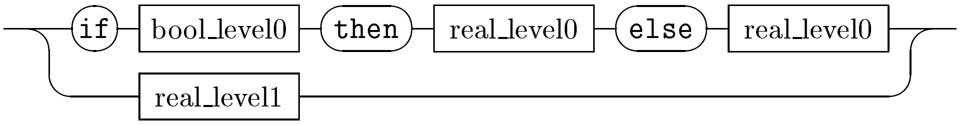


int_level3

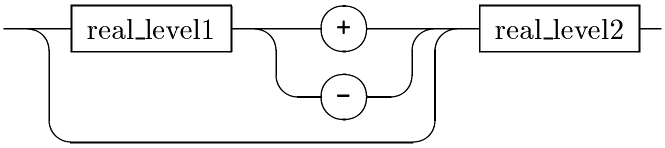


A.2.4 Aufbau von `real`-Ausdrücken

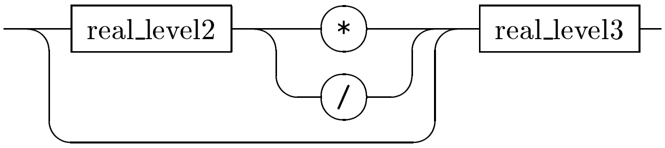
real_level0



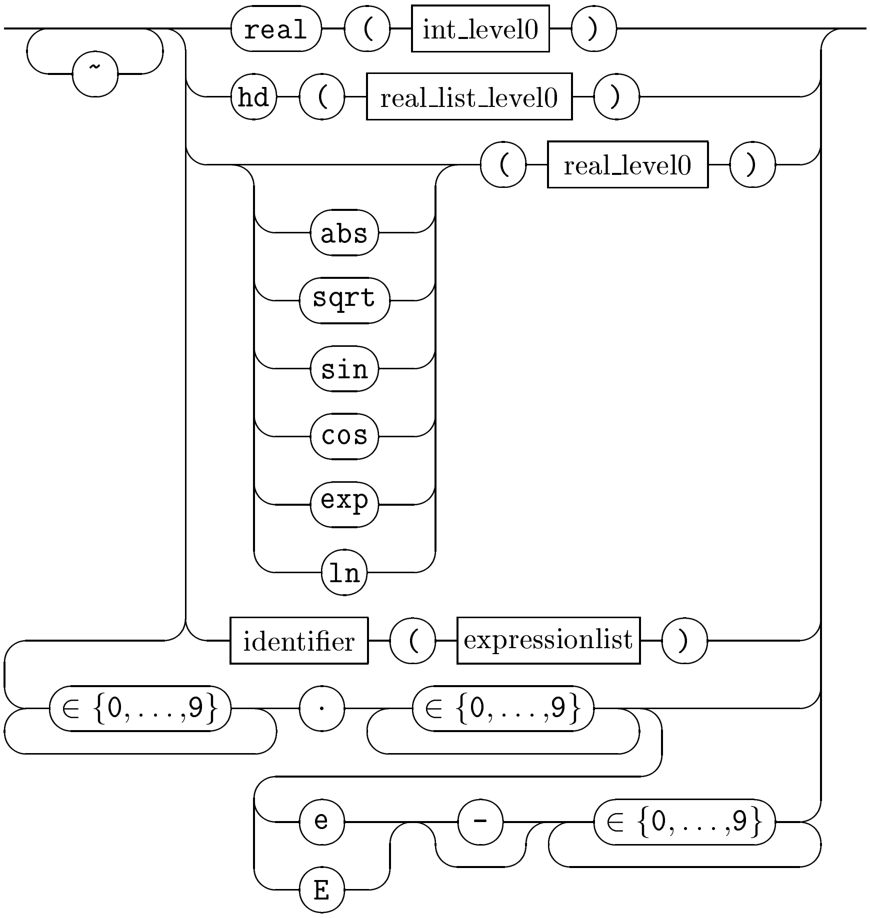
real_level1



real_level2

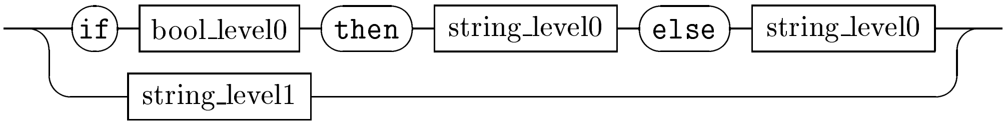


real_level3

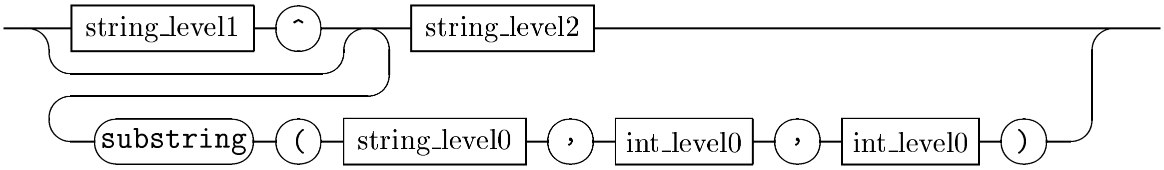


A.2.5 Aufbau von `string`-Ausdrücken

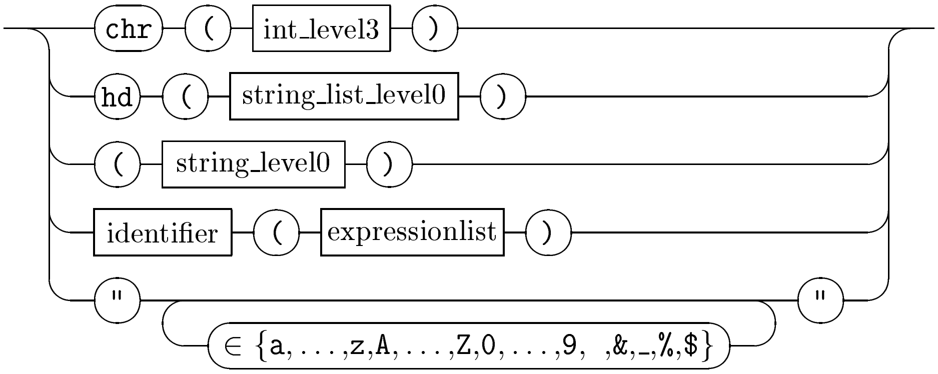
string_level0



string_level1

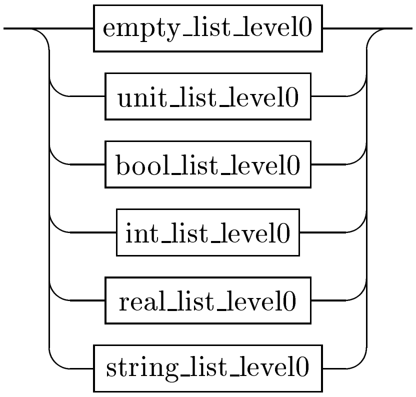


string_level2



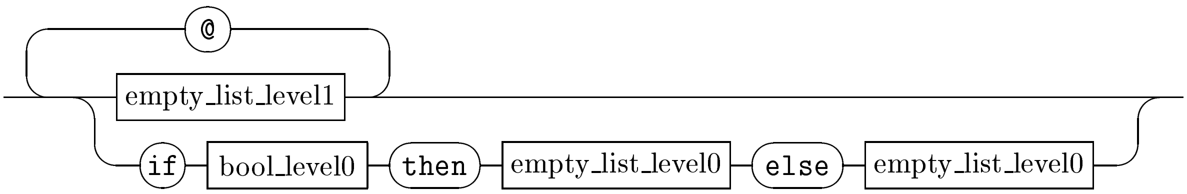
A.2.6 Aufbau von Listen-Ausdrücken

list_level

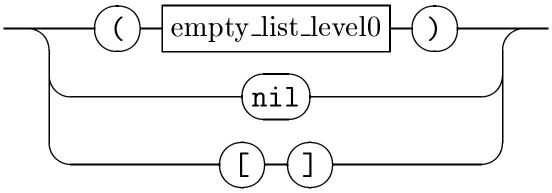


A.2.6.1 Aufbau der leeren Liste

empty_list_level0

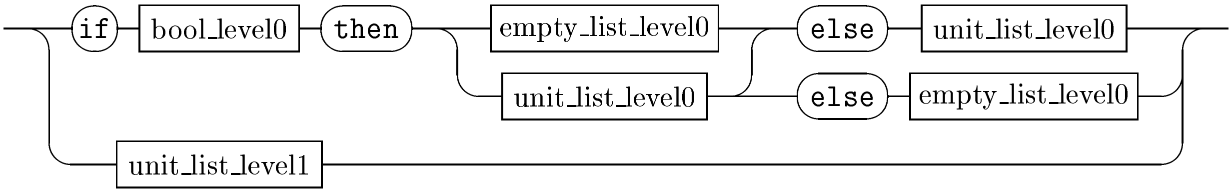


empty_list_level1

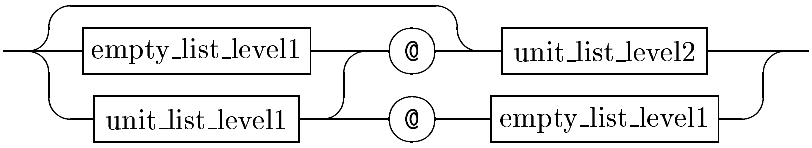


A.2.6.2 Aufbau von `unit`-Listen-Ausdrücken

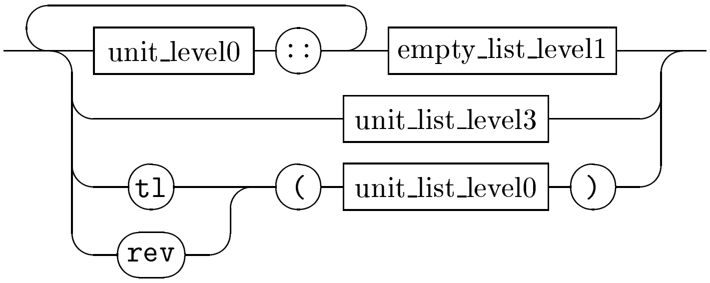
*unit_list_level*0



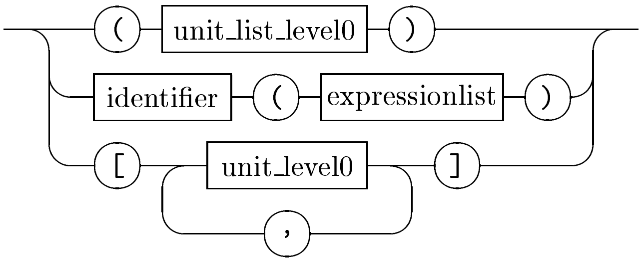
*unit_list_level*1



*unit_list_level*2

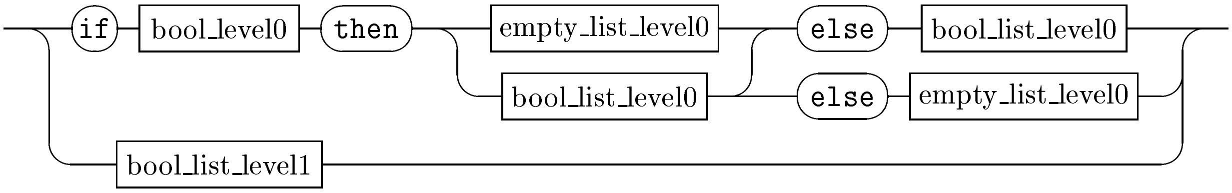


*unit_list_level*3

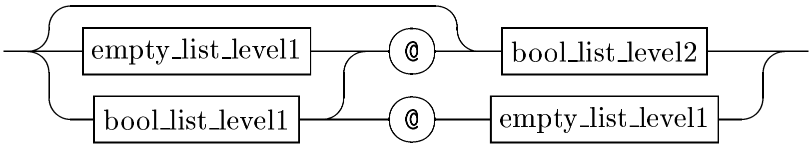


A.2.6.3 Aufbau von `bool`-Listen-Ausdrücken

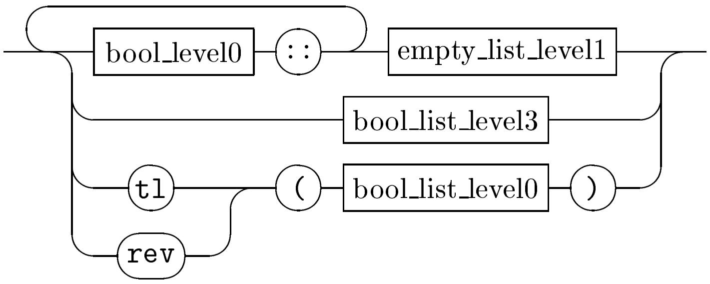
bool_list_level0



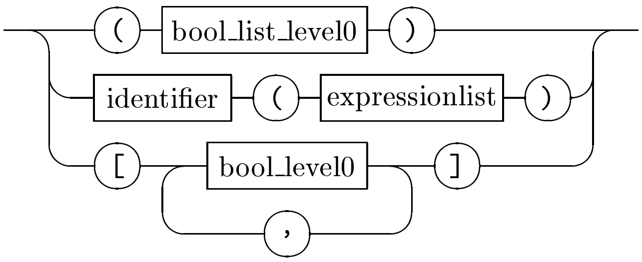
bool_list_level1



bool_list_level2

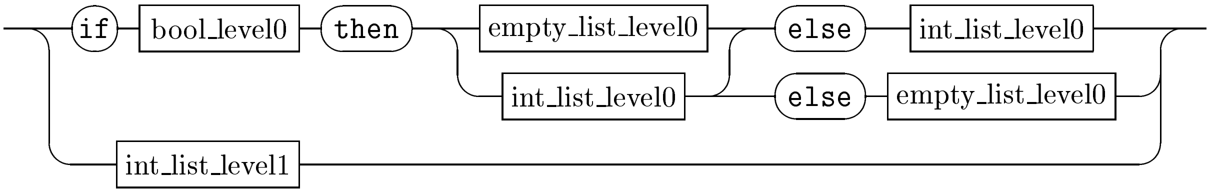


bool_list_level3

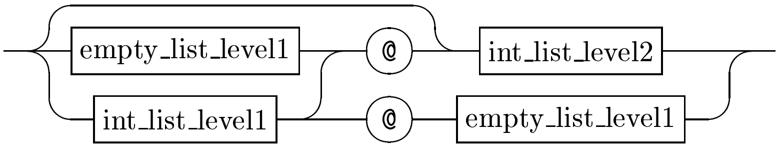


A.2.6.4 Aufbau von `int`-Listen-Ausdrücken

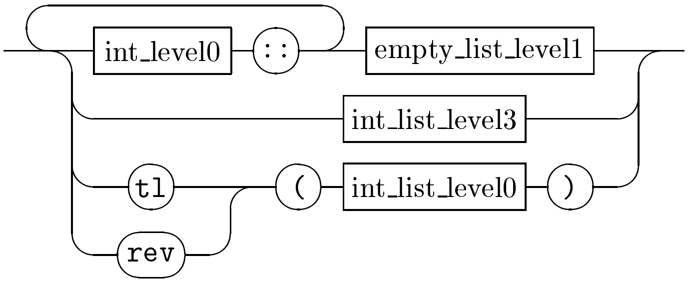
*int_list_level*0



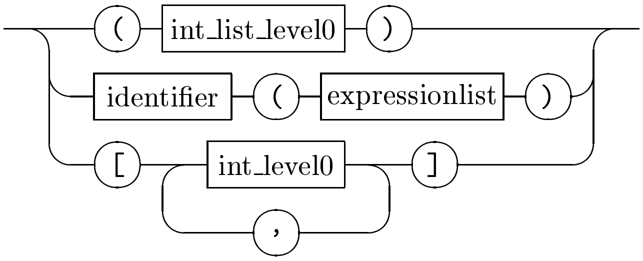
*int_list_level*1



*int_list_level*2

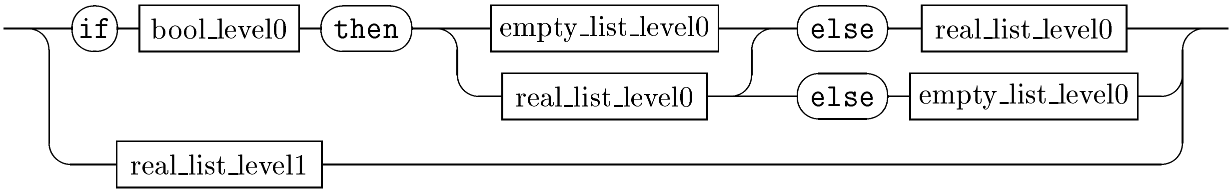


*int_list_level*3

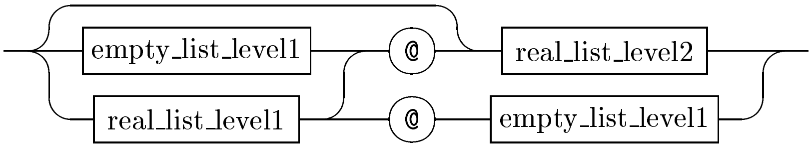


A.2.6.5 Aufbau von `real`-Listen-Ausdrücken

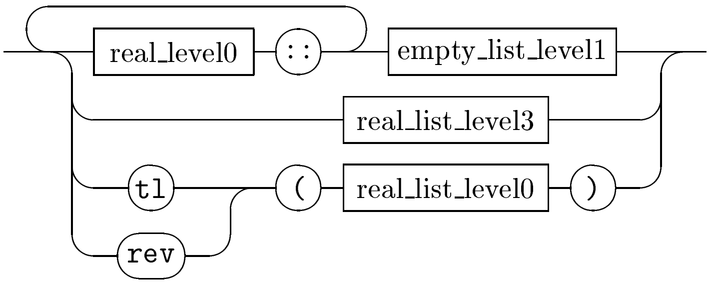
*real_list_level*0



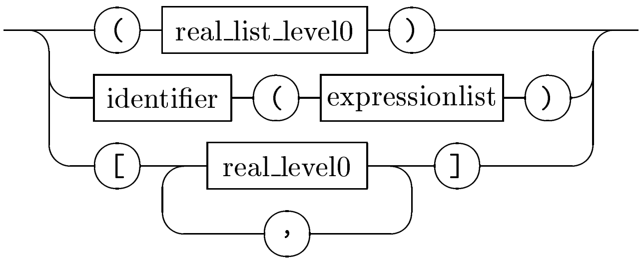
*real_list_level*1



*real_list_level*2

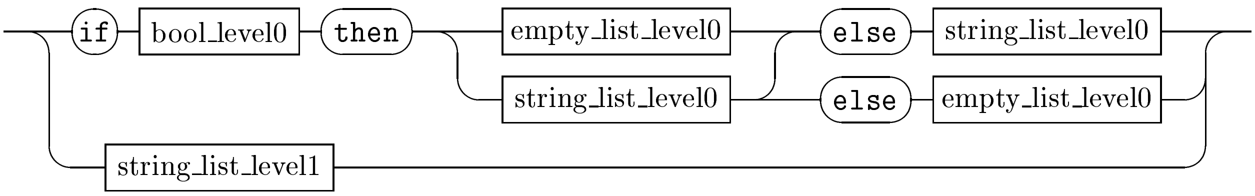


*real_list_level*3

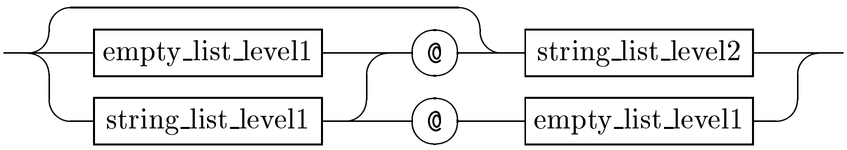


A.2.6.6 Aufbau von `string`-Listen-Ausdrücken

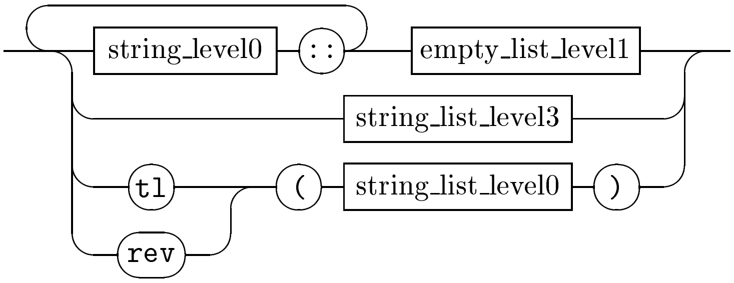
string_list_level0



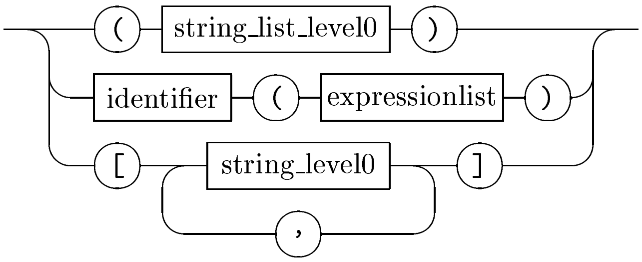
string_list_level1



string_list_level2



string_list_level3



Literaturverzeichnis

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

[2] Berghammer, Rudolf und Tied, Markus : *Kiel Interactive Evaluation Laboratory*, Institut für Informatik und Praktische Mathematik, Universität zu Kiel, Bericht 1909, 1999

[3] Krüger, Guido: *Java 1.1 Grundlagen Programmierung*, HERDT-Verlag, 1998

[4] Krüger, Guido: *Java 1.1 Fortgeschrittene Programmierung,* HERDT-Verlag, 1998

[5] Rumbaugh, J.,Blaha, M.,Premerlani, W., Eddy, F., Lorensen, W.: *Objektorientiertes Modellieren und Entwerfen,* Hanser, 1993

[6] Oesterreich, Bernd: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modelling Language,* Oldenbourg, 1998

[7] Hunt, John: *Essential Javabeans fast,* Springer, 1998

[8] Stroustrup, Bjarne: *Die C++-Programmiersprache,* Addison-Wesley, 1998

[9] Sneed, Harry M.: *Objektorientierte Softwaremigration*, Addison-Wesley, 1998

[10] Griscom, Daniel: *Code Signing for Java Applets,* http://www.suitable.com/Doc_CodeSigning.shtml

[11] Gong,L., Mueller, M., Prafullchandra, H., Schemers, R.: *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java*^{TM} *Development Kit 1.2* in Proc. of USENIX Symposium on Internet Techologies and Systems, Monterey, California, Dec. 1997, http://java.sun.com/people/gong/papers/jdk12arch.ps..gz.

[12] Jain, Prashant und Schmidt, Douglas C.: *Experiences Converting a C++ Software Framework to Java*, www.cs.wu stl.edu/~pjain/java_notes.html

[13] Lean, Doug: *Concurrent Java: Design Principles and Patterns*, Addison-Wesley, 1996

[14] Morris, J.H.: *Lambda-calculus models of programming*, Ph.D. thesis, Mass. Inst. of Technology, 1968

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbständig verfaßt habe und ausschließlich die angegebenen Hilfsmittel und Quellen verwendet habe.

Kiel, den 19. April 2000 (Oliver Lorenz)