Animation der semantischen Analyse

Diplomarbeit

von

Andreas Kerren

angefertigt am Fachbereich Informatik der Universität des Saarlandes nach einem Thema von Prof. Dr. R. Wilhelm, Lehrstuhl für Compilerbau und Programmiersprachen

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit selbständig und nur mit Hilfe der angegebenen Literatur angefertigt habe.

Saarbrücken, im April 1997

Danksagung

Ich danke Herrn Prof. Dr. Reinhard Wilhelm für die Vergabe des in vieler Hinsicht interessanten Themas, seiner Unterstützung und seinem Enthusiasmus für dieses Projekt.

Bedanken möchte ich mich auch bei meiner Betreuerin Beatrix Braune und Dr. Stephan Diehl für die vielen und umfangreichen Diskussionen, die mir bei der Konzeptions- und Realisierungsphase dieser Arbeit sehr weitergeholfen haben.

Weiterer Dank gilt meinen FreundInnen und meinen Eltern, die mich während dieses Studiums stets motiviert und an mich geglaubt haben.

Andreas Kerren April 1997

Inhaltsverzeichnis

Einleitung

Viele Experten aus der Pädagogik, Psychologie etc. streiten sich darüber, wie man Interesse für ein bestimmtes Thema wecken kann und es so aufbereitet, daß es sich möglichst gut im Gedächtnis eines/einer Lernenden festsetzt und verstanden wird. Dabei spielt die Bedeutung der visuellen Informationsvermittlung eine immer größere Rolle: Zum Beispiel lernen wir 83% durch das, was wir sehen, und nur 11% durch das, was wir hören (vgl. [Alt93]).

Der Grund für den Vorrang des visuellen Lernens liegt in der Aufteilung des menschlichen Gehirns in zwei Hemisphären (verbunden durch das „Corpus Callosum"). Die linke Hemisphäre ist die „kognitive"; sie verarbeitet eins nach dem anderen, denkt logisch und entscheidet nach Regeln und Gesetzmäßigkeiten. Andererseits ist die rechte Hemisphäre die „affektive"; sie denkt in Bildern, nutzt Analogien, sprengt Regeln, reagiert spontan und kreativ.

Es ist wichtig, eine Information so aufzubereiten, daß *beide* Hemisphären angesprochen werden. Zeigt man zu einem abstrakten Begriff ein Bild, so nutzt man beide „Informationskanäle" und ermöglicht die Verbindung des Begriffs mit einer bildhaften Vorstellung. Dies hängt aber offensichtlich auch von dem Themengebiet ab. Spricht man beispielsweise von einem bekannten Begriff, etwa einem PC, so sendet die rechte Hemisphäre sofort ein Bild von einem eckigen Kasten, mit Monitor, Tastatur und vielleicht auch einer Maus. Spricht man andererseits zum Beispiel von dem Begriff KI, und man hört diesen Begriff zum erstenmal, dann kann man sich davon kein „Bild machen".

Um Informationen im Gedächtnis zu verankern, gibt es drei grundsätzliche Möglichkeiten des Lernens (vgl. Abb. E.1):

Wiederholungslernen: Darunter versteht man beispielsweise das Lernen von Vokabeln, Regeln und Formeln. Da Wiederholungen primär linkshirnorientiert sind, kommt es zu Schwierigkeiten, diese Informationen im Langzeitgedächtnis zu behalten.

Intensivlernen: Emotional „stark aufgeladene" Informationen garantieren höchste Behaltenssicherheit. Wer z.B. zum Löschen des gesamten Inhalts eines Verzeichnisses den Befehl FORMAT C: ausführt, der verwechselt diesen Befehl vermutlich nicht wieder mit DEL *.*. Es bedarf in diesem Fall keiner Wiederholung.

Integrationslernen: Hierbei wird der Lernvorgang durch eine Aufarbeitung der Informationen, welche die Spezialisierung und Eigenschaften beider Hemisphären ausnutzt, unterstützt. Wer es versteht Informationen zu visualisieren, kann die Merkfähigkeit bei den Lernenden steigern.

Manchmal ist es aufgrund der Umstände und der Thematik für LehrerInnen nicht möglich, das *Integrationslernen* zu unterstützen. Nehmen wir als Beispiel eine Vorlesung über Compilerbau. Soll etwa die Arbeitsweise von nicht-deterministischen bzw. deterministischen Automaten (endlicher Automat, Kellerautomat) an einer Tafel erläutert werden, so bedeutet dies, das jede Änderung der Zustände, Kellereinträge etc. angeführt werden muß. Es kommt

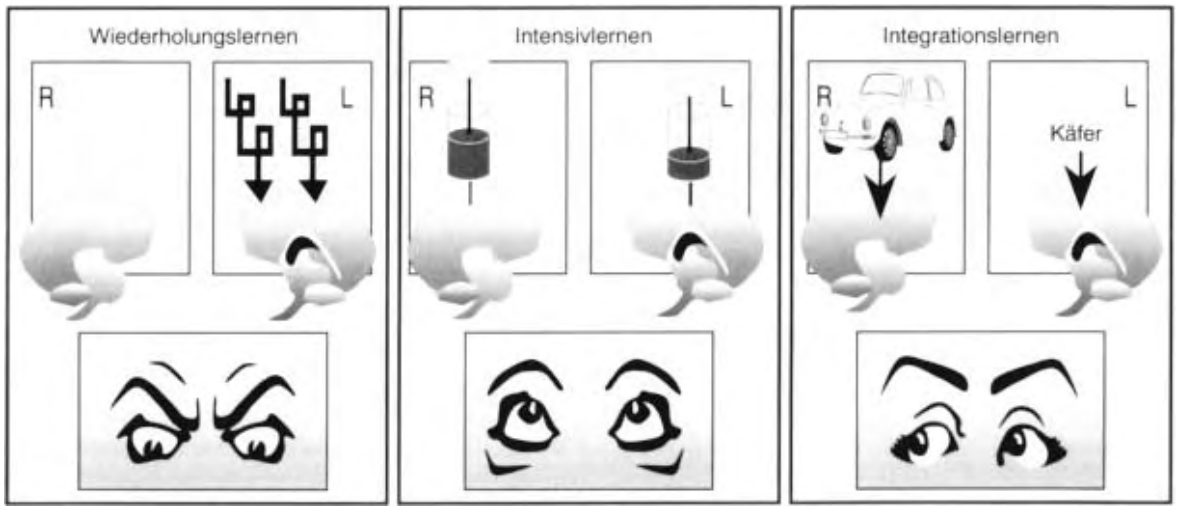


Abb. E.1: Drei Wege, Informationen ins Gedächtnis aufzunehmen; aus [Alt93]

zu „dynamischen Tafelbildern“, in denen ältere Bestandteile weggewischt und durch neue zu ersetzen sind. Am Ende der Prozedur ist dann nur noch das Endergebnis zu sehen. Die Lernenden haben nicht die Möglichkeit alle Zwischenschritte abzumalen, weil alles zu schnell geht und zu unübersichtlich ist. Damit ist die Vorführung eines solchen Automaten auch nur sehr schwer reproduzierbar. Eine Lösung dieses Problems sind Computeranimationen, die den Vortrag der LehrerInnen begleiten oder von den Lernenden selbst ausgeführt werden.

Arbeitsumgebung

Am Lehrstuhl für Compilerbau und Programmiersprachen, an dem auch diese Arbeit entstand, gab es bereits Visualisierungen abstrakter Maschinen bestimmter imperativer, logischer und funktionaler Programmiersprachen:

- Visualisierung der abstrakten P-Maschine, [Koh95]
- Visualisierung der abstrakten Maschine WIM, [Wir95]
- Visualisierung der abstrakten Maschine MAMA, [Ste92]

Diese Visualisierungen sind unter X-Windows (UNIX) implementiert worden. Sie enthalten jedoch kaum Animationen.

Weiterhin wurde an diesem Lehrstuhl ein Tool zur Visualisierung von Graphen aus dem Compilerbau, genannt VCG („Visualisation of Compiler Graphs“) entwickelt. Es existieren Versionen des VCG-Tools für mehrere Computersysteme, auch Windows 3.1. Siehe hierzu [Lem94], [LS94], [San95] und [San96].

Um ein möglichst großes Publikum zu erreichen, wurde dann ein Projekt mit dem Ziel ins Leben gerufen, daß große Teile des Buches „Übersetzerbau“ [WM92] unter **Microsoft Windows 3.1** visualisiert und vor allem animiert werden. Das zugrundegelegte Hilfsmittel hierfür war **Asymetrix Multimedia ToolBook 3.0** (MTB 3.0), ein Autorensystem, das die Erstellung von Lernsoftware unter Windows 3.1 unterstützt. Als Teil dieses Projekts ist auch meine Arbeit, die *Animation der semantischen Analyse*, entstanden. Parallel dazu entwickelte Beatrix Braune die *Animation der lexikalischen Analyse*. Weiterhin entstanden im Rahmen eines Fortgeschrittenenpraktikums *Animationen ausgewählter Parserkonzepte*.

Inhalt dieser Arbeit

Ich hatte die Aufgabe eine Software zu entwickeln, mit deren Hilfe man die wichtigsten Konzepte der semantischen Analyse animieren kann. Dazu gehören neben den Voraussetzungen der semantischen Analyse (abstrakter Syntaxbaum, etc.) und den Gültigkeits- und Sichtbarkeitsregeln, die Animation der Überprüfung der Kontextbedingungen (Deklarations-Analysator und Typkonsistenz-Analysator), der Überladung von Bezeichnern und schließlich des Polymorphismus. Der erste Programmteil, die *animierte Präsentation*, präsentiert und erläutert zunächst schrittweise die Definitionen dieser Konzepte. Sie werden anschließend anhand von animierten Beispielen verdeutlicht. Beides geschieht vollständig interaktiv, d.h. die BenutzerInnen können sich per Mausklick durch eine graphische Umgebung bewegen, Themen, die sie interessieren auswählen, vertiefen und Beispielanimationen steuern. Schließlich besteht in einem zweiten Programmteil (ASA = Animationstool zur Semantischen Analyse) die Möglichkeit, Beispiele selbst anzugeben und die vorgestellten Algorithmen dynamisch auf den abstrakten Syntaxbäumen der eingegebenen Beispiele animiert ablaufen zu lassen. Beispiele können Eingabeprogramme, Ausdrücke oder Spezifikationen sein.

Die *animierte Präsentation* wurde mit dem Autorensystem MTB 3.0 entwickelt. Das ASA-Tool habe ich dagegen direkt auf Windows-Ebene in der Programmiersprache C implementiert und mit dem Compiler *MS Visual C++* übersetzt. Auf die Gründe für diese Aufteilung wird an den entsprechenden Stellen in dieser Arbeit eingegangen.

Zuerst beschreibe ich ausführlich das Windows 3.1 System und dessen Programmschnittstelle. Da das aus technischer Sicht anspruchsvollere ASA-Tool (z.B. Layout von Bäumen und deren graphische Darstellung, etc.) unmittelbar diese Programmschnittstelle verwendet, messe ich deren Beschreibung eine höhere Bedeutung bei, als einer ausgiebigen Schilderung des Autorensystems und dessen Skriptsprache OPENSCRIPT. Im Anschluß daran gebe ich einen Umriß über das der *animierten Präsentation* zugrundeliegende Autorensystem Multimedia ToolBook 3.0. Die genaue Beschreibung von Windows im vorausgegangenen Kapitel erleichtert auch das Verständnis der Funktionsweise dieses Autorensystems. Darauf folgt die Theorie der semantischen Analyse, sowie die Darstellung einiger Richtlinien zur Erstellung von graphischen Benutzeroberflächen und Animationen. Abschließend beschreibe ich den zwei Programmteilen entsprechend die Eigenschaften der *Animation der semantischen Analyse*, sowie die Möglichkeiten zur Interaktion und gehe auf die wichtigsten Details der Implementierung ein.

Kapitel 1

Microsoft® Windows™ 3.1

Mit **Windows** stellt Microsoft eine graphisch orientierte Benutzerumgebung zur Verfügung, die momentan alle vergleichbaren Konkurrenzprodukte auf dem PC-Markt dominiert. Hunderte sehr leistungsfähige Applikationen stehen dem/der BenutzerIn unter dieser Oberfläche zur Verfügung, deren komfortable Bedienung einen großen Fortschritt in der Benutzerfreundlichkeit im Vergleich zu früheren DOS-Programmen darstellt.

Die Grundlagen für graphische Benutzeroberflächen dieser Art wurden gegen Mitte der 70er Jahre im Palo Alto Research Center (*PARC*) von Xerox für Programmiersysteme wie z.B. Smalltalk geschaffen. Anschließend hat die Firma Apple die dabei entwickelten Prinzipien der breiten Öffentlichkeit durch die Einführung der Macintosh-Serie zugänglich gemacht. Ein kurzer Rückblick auf die Geschichte des Windows-Systems gibt Auskunft über dessen Entwicklung und technischen Veränderungen:

Windows 1.01, 1985: Erste freigegebene Version, noch ohne sich überdeckende Fenster oder Dialoge, aber schon mit eigener Speicherverwaltung (640 KByte Grenze von MS-DOS fällt weg).

Windows 2.0, 1987: Erstmals mit sich überdeckenden Fenstern und Dialogen.

Windows/386, 1988: Der *Virtual 8086 Mode* des 80386 wird verwendet, um den Parallelbetrieb von DOS-Programmen mit direkten Hardwarezugriffen zu realisieren.

Windows 3.0, 1990: Diese Version unterstützt den *Protected Mode* des 80286 sowie seiner Nachfolger auch für Anwendungsprogramme und stellt somit bis zu 16 MByte durchgehenden Hauptspeicher zur Verfügung.

Windows 3.1, 1992: Anwendung der TrueType-Technologie, wodurch WYSIWYG-Systeme („What You See Is What You Get“) erst möglich wurden. Bereitstellung von fertigen Dialogen für Standardprozeduren wie das Öffnen von Dateien, OLE („Object Linking and Embedding“), Kommunikationsprotokollen und Multimediafunktionen. Windows 3.1 läuft nur noch im *Protected Mode*, d.h. es setzt einen 80286 und 1 MByte Hauptspeicher voraus.

Graphische Umgebungen versuchen Informationen bildhaft darzustellen. Durch diese visuelle Form der Bedienerführung durch Programme und Darstellung der Daten werden auch Menschen angesprochen, die Berührungsängste vor kompliziert anmutender Software haben. Weiterhin ermöglicht eine gewisse Uniformität im Erscheinungsbild der Programme eine verkürzte Einarbeitungszeit. Um diese Ziele zu erreichen, gibt es einige Voraussetzungen an das System:

Geräteunabhängige Graphik: Dies wird durch eine gemeinsame Graphikschnittstelle für alle Anwendungsprogramme erreicht.

Konsistenz in der Benutzung: Alle identischen Operationen (z.B. Datei öffnen) sollten gleich repräsentiert werden. Einige Bibliotheken, mit entsprechenden Funktionen ausgestattet, werden mit dem System mitgeliefert.

Multitasking: Mehrere Programme (repräsentiert durch Fenster) können „gleichzeitig“ laufen.

Eine nahezu vollständige Beschreibung des Windows 3.1 Systems findet man in [Pet92].

1.1 Programmierung

Anders als kommandozeilenbasierte Programme müssen sich Windows-Programme, wie das ASA-Tool, zusätzlich um ihre eigene graphische Repräsentation bemühen. Da ein Windows-Programm grundsätzlich vom System mit einem Fenster assoziiert wird, müssen auch bei Windows-Programmen, die im Hintergrund laufen, Fenster erzeugt und Fenstereigenschaften gesetzt werden. Die entsprechenden Fenster werden in diesem Fall einfach unsichtbar gemacht. Eine Folge daraus ist, daß jedes Windows-Programm einen gewissen Überbau benötigt, der es ihm ermöglicht, sowohl sein Fenster zu verwalten, als auch eine Kommunikation mit dem System vorzubereiten.

1.1.1 Schnittstelle zum System

Windows teilt sich mit DOS die Verwaltung des Computers. Während DOS alle Funktionen bezüglich des Dateisystems übernimmt, verwaltet Windows den Rest (Bildschirm, Tastatur, Maus, Speicher, ...). Die Systemschnittstelle, genannt **Windows-API** („Application Programming Interface“), besteht aus etwa 1000 Funktionen und mehreren tausend Konstanten, die alle in einer speziellen Headerdatei WINDOWS.H deklariert sind. Größtenteils besteht Windows aus drei dynamischen Bibliotheken (DLL’s)

- KRNL386 (Speicher- und Programmverwaltung),
- USER (Benutzerschnittstelle und Fenster),
- GDI (Graphik),

die alle erwähnten Funktionen enthalten. Jedes Windows-Programm wird beim Laden an die benötigten Funktionen in diesen DLL’s dynamisch gebunden. Falls sich die entsprechende DLL noch nicht im Speicher befindet, lädt das System sie vorher in den Speicher.

Um eine dynamische Bindung zu realisieren, erzeugt der Compiler bei der Übersetzung des Windows-Programms für jeden Aufruf einer Windows-Funktion einen Eintrag in einer zusätzlichen Tabelle, die der Linker an die zu erzeugende Datei anheftet. Dieser Eintrag enthält jeweils den Namen einer dynamischen Bibliothek und entweder den Namen oder eine Referenznummer der betreffenden Funktion. Beim Laden des Programms werden sämtliche Aufrufe von Windows-Funktionen auf die tatsächlichen Speicheradressen angepaßt. Aus diesem Grund sind Windows-Funktionen selbst vollständig adressiert. Sie befinden sich in einem anderen Codesegment. Zusätzlich erwarten Windows-Funktionen vollständige Adressen, da dynamische Bibliotheken ein eigenes Datensegment besitzen. Für die Beschreibung der vollständigen Adressierung ist eigens ein neuer Datentyp *far* eingeführt worden. Er teilt dem Compiler mit, daß sich das Objekt in einem anderen Code- bzw. Da-

tensegment befindet. Dementsprechend sind alle Funktionen (bis auf eine Ausnahme) und deren Parameter in WINDOWS.H als *far* deklariert.

1.1.2 Programme

DOS-Programme rufen das Betriebssystem auf, um bestimmte Aktionen, wie das Laden von Dateien, durchzuführen. Unter Windows kann das System umgekehrt auch Funktionen der Windows-Applikation aufrufen. Somit können Windows-Programme auf **Ereignisse** (z.B. Mausklicks) in geeigneter Weise reagieren.

Jedes in Windows angelegte Fenster basiert auf einer **Fensterklasse**, die ihrerseits eine **Window-Prozedur** festlegt und dem System deren Adresse mitteilt. Mehreren Fenstern kann so eine einzige Window-Prozedur zugeordnet werden. Sie kann sich im Programm selbst oder in einer dynamischen Bibliothek befinden. Windows übermittelt Botschaften (z.B. angeregt durch Ereignisse) an ein Fenster, indem es seine Window-Prozedur aufruft. Fenster werden durch eindeutige Bezeichnungen, Fenster-Handles genannt, unterschieden.

Beim Start eines Windows-Programms legt das System eine **Ereignis-Warteschlange** an, in die alle Ereignisse eingetragen und nach dem FIFO-Prinzip („First In, First Out“) an die Fenster des Programms weitergeleitet werden. Dementsprechend besteht die eigentliche Hauptfunktion des Programms **WinMain** im wesentlichen aus folgenden Elementen:

1. Definition der Fensterklasse.
2. Registrierung der Klasse beim System und damit Bekanntmachung der Window-Prozedur.
3. Erzeugung des Hauptfensters auf Basis der definierten Fensterklasse.
4. Ereignis-Warteschleife zum Lesen der Warteschlange.

Die **Ereignis-Warteschleife** testet, ob die Warteschlange leer ist (vgl. Abb. 1.1). Wenn eine Botschaft enthalten ist, so leitet sie die (eventuell übersetzte) Botschaft über einen zusätzlichen Aufruf des Systems an ihre programmeigenen Window-Prozeduren, die dann mittels

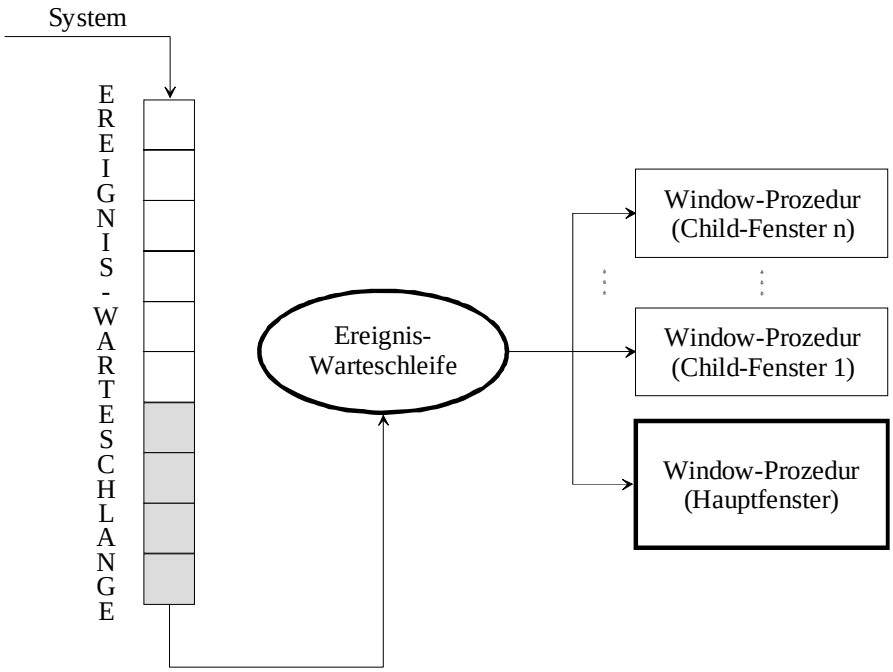


Abb. 1.1: Die Warteschlange eines Window-Programms

einer Fallunterscheidung entsprechend reagieren.

Es gibt zwei Arten von Botschaften. Sie werden unterteilt in

- **direkte** Botschaften, die mit bestimmten Funktionen an ein Fenster gesendet werden.
- **indirekte** Botschaften, die in die Warteschlange eines Programms eingereiht werden, wie bereits oben beschrieben wurde.

Jedes Windows-Programm kann aus mehreren Fenstern bestehen. Sie sind normalerweise in einer **Hierarchie** organisiert, die während der Erzeugung der Fenster von dem/der EntwicklerIn festgelegt wird. Man bezeichnet das einem Fenster direkt übergeordnete Fenster als *Parent-Fenster*, das untergeordnete Fenster hingegen als *Child-Fenster*. Child-Fenster können nicht außerhalb der Oberfläche des Parent-Fensters gezeichnet werden. Gibt man bei der Erzeugung eines Fensters kein Parent-Fenster an, so gilt diese Einschränkung nicht.

1.1.3 Prozesse

Das Windows-System beherrscht sogenanntes *non-preemptives* (nicht-verdrängendes) Multitasking. Im Gegensatz zu *preemptiven* Multitasking-Systemen, die mit Hilfe einer Zeitscheibe nach einem bestimmten Zeitintervall von einem Prozeß zu einem anderen umschalten, wechselt Windows i.d.R. erst zu einem anderen Prozeß, wenn die Warteschlange des aktuellen Prozesses leer ist bzw. dort nur Botschaften niedriger Priorität eingesetzt sind und ein Ereignis in der Warteschlange eines anderen Programms existiert. Falls sich mehrere andere Programme in diesem Zustand befinden, so wählt das System das Programm mit der höchsten Botschaftspriorität aus. Es ist aber auch möglich manuell, durch Aufrufen entsprechender Windows-Funktionen, direkt zu anderen Prozessen zu wechseln.

1.2 Ereignisse

Ereignisse werden zu Nachrichten (hier: **Botschaften**) übersetzt und als Parameter der Window-Prozedur an das Programm übertragen. Eine Botschaft und somit auch die formalen Parameter der Window-Prozedur besteht aus vier Elementen:

- Handle des Fensters.
- ID der Botschaft, die in WINDOWS.H als Konstante festgelegt ist.
- *wParam*, ein 16 Bit Parameter, der zur Botschaft gehört und in Abhängigkeit von der Botschaft unterschiedliche Bedeutung hat.
- *lParam*, ein 32 Bit Parameter, der ebenfalls zur Botschaft gehört und verschiedene Informationen enthalten kann.

Wird die Botschaft nicht von der Window-Prozedur bearbeitet, so schickt sie die Prozedur an das System weiter.

Die Windows-Prozedur des ASA-Tools im besonderen erhält neben Systembotschaften auch solche Botschaften, die durch Ereignisse der Tastatur, der Maus, mehrerer Windows-Timer und diverser Kontrollelemente erzeugt werden.

1.2.1 Tastatur

Als Reaktion auf **Tastaturereignisse**, wie das Betätigen einer oder mehrerer Tasten, reiht

der Tastaturtreiber entsprechende Codes in die Warteschlange des Systems ein. Windows gibt diese Codes in Form von Botschaften an die Warteschlange des Programms, das den **Eingabefokus** besitzt. Die Zwischenspeicherung in der Warteschlange des Systems ist notwendig, wenn mehr Zeichen eingegeben werden, als das Programm bearbeiten kann.

Tastenbotschaften werden danach unterteilt, ob es sich bei der Taste um eine Standard- oder Systemtaste handelt und ob sie gedrückt oder losgelassen wird. Auf Programmierebene sind dies die Botschaften WM_KEYDOWN und WM_KEYUP bzw. WM_SYSKEYDOWN und WM_SYSKEYUP. WM steht hier für „Window Message“.

wParam enthält einen virtuellen Tastencode, der für die betätigte Taste steht. *lParam* besteht aus sechs getrennten Elementen:

1. einen Wiederholungszähler (16 Bit), der die Anzahl gleichartiger Tastendrücke enthält,
2. den OEM-Scancode (8 Bit), der normalerweise ignoriert wird,
3. einen Flag für erweiterte Tasten (1 Bit), wenn eine Tastenkombination gedrückt wurde,
4. den Kontext-Code (1 Bit), wenn die Alt-Taste zusätzlich betätigt wurde,
5. den vorherigen Tastenzustand (1 Bit), der feststellt, ob die Taste vorher auch schon gedrückt war, und
6. die Art des Wechsels (1 Bit), die eine zusätzliche Möglichkeit bietet, eine Unterscheidung zwischen „gedrückt“ und „gelöst“ zu machen.

1.2.2 Maus

Im Gegensatz zu Tastaturereignissen gilt für **Mausereignisse** nicht, daß sie nur das jeweils aktive Fenster betreffen. Sobald der Mauszeiger ein beliebiges, auch eventuell inaktives Fenster überstreicht, so sendet das System durch Mausereignisse ausgelöste Botschaften an dieses Fenster. Um die Eindeutigkeit des Empfangsfensters zu gewährleisten, besitzt jeder Mauszeiger einen sogenannten *Hot Spot*, einen Zielpunkt der seine exakte Position festlegt. Die Art und Weise, wie Windows ein Mausereignis in eine Botschaft transformiert, hängt

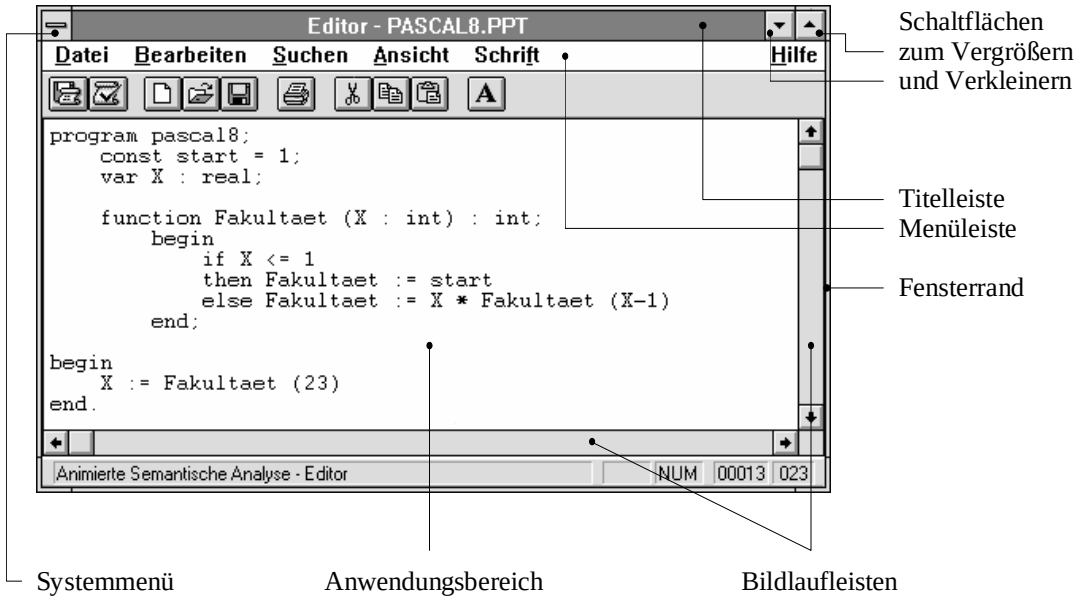


Abb. 1.2: Das Fenster eines typischen Window-Programms

von der Position des Mauszeigers im Fenster ab. Fenster können aus vielen Elementen bestehen: Titelleiste, Menüleiste, Fensterrand, Anwendungsbereich, Schaltflächen usw. (siehe Abb. 1.2). Befindet sich der *Hot Spot* im Anwendungsbereich eines Fensters, dann löst das Betätigen einer Maustaste eine zu den Tastaturereignissen analoge Botschaft aus. Jedoch werden hier auch Doppelklicks berücksichtigt. Das Botschaftsattribut *wParam* enthält einen Zustand, der über eine zusätzlich gedrückte Taste informiert. *lParam* besitzt eine Positionsangabe der Stelle, an der das Ereignis aufgetreten ist. Diese wird relativ zur linken oberen Ecke des Anwendungsbereichs gemessen.

Außerhalb des Anwendungsbereichs, also z.B. in der Menüleiste, haben die Mausereignisse fast denselben Effekt wie innerhalb des Anwendungsbereichs. Eine Ausnahme bilden die Botschaftsattribute. Hier gibt *wParam* die Kennziffer des Bereichs (z.B. Menü) an, *lParam* auch die Position, jedoch relativ zur linken oberen Ecke des entsprechenden Bereichs.

Schließlich gibt es noch Bewegungsbotschaften, wie WM_MOUSEMOVE, die bei jeder Bewegung des Mauszeigers an das darunter befindliche Fenster gesendet werden.

1.2.3 Timer

Timerbotschaften sind nützlich um die Geschwindigkeit von Animationen zu regeln. Sie werden vom System nur nach Anfrage des Anwendungsprogramms gesendet. Windows ist in der Lage, 32 Timer gleichzeitig zu verwalten. Die Anfrage erfolgt durch den Aufruf der Window-Funktion *SetTimer*, die als Parameter u.a. den Fenster-Handle des anfragenden Fensters und ein Zeitintervall (Periode) erwartet. Kann ein Timer bereitgestellt werden, so fügt das System nach Ablauf jeder Periode die Botschaft WM_TIMER in die Warteschlange des Programms ein. Für jeden Timer vergibt das System eine eindeutige Bezeichnung (ID), mit deren Hilfe sich der Timer beenden läßt.

wParam enthält die ID des Timers, der die Botschaft sendet, *lParam* ist auf 0 gesetzt.

1.2.4 Kontrollelemente

Kontrollelemente sind Child-Fenster, die Maus- und Tastaturereignisse über eine Window-Prozedur bearbeiten. Sie zeigen Reaktionen auf die Aktion optisch an und informieren ihr Parent-Fenster über Veränderungen ihres Status, wobei das Parent-Fenster bei der Erzeugung des Kontrollelements festgelegt wird. Meist sind sie in Dialogen vertreten.

Windows stellt Fensterklassen und Window-Prozeduren für Elemente dieser Art dem/der ProgrammiererIn zur Verfügung. Sie werden auf Basis der Fensterklasse erzeugt, erhalten einen Fenster-Handle und eine eindeutige ID, und werden ansonsten wie „normale“ Fenster gehandhabt:

Schaltflächen, Kästchen, Radiokontrollen, ... (Fensterklasse *button*): Es gibt insgesamt zehn verschiedene Formen von Knöpfen.

Editfenster (Fensterklasse *edit*): Sie dienen als Eingabefeld für Text und dessen Bearbeitung. Das Kontrollelement übernimmt die Verwaltung über die Bewegung des Carets (Textcursor), das Löschen und Einfügen von Zeichen, das Markieren von Bereichen über die Maus oder die Pfeiltasten der Tastatur, das Ausschneiden markierter Bereiche in die Zwischenablage und das Einfügen aus der Zwischenablage.

Listenfenster (Fensterklasse *listbox*): Sie ermöglichen die Auswahl von ein oder mehreren Elementen, die als Textstrings spaltenweise dargestellt werden. Eine Selektion eines Elements wird farbig markiert.

Kombielemente (Fensterklasse *combobox*): Diese kombinieren Listenfenster mit Text- oder Eingabefeldern.

Bildlaufleisten (Fensterklasse *scrollbar*): Sie unterscheiden sich von ihren verwandten Vorkommen als Fensterbestandteil am rechten oder linken Fensterrand dadurch, daß sie an beliebiger Stelle im Anwendungsbereich plaziert werden können. Sie können als horizontale und vertikale Bildlaufleisten definiert und in ihrer Breite bzw. Höhe variiert werden.

Textfelder (Fensterklasse *static*): Solche Kontrollelemente geben lediglich einen Text formatiert auf dem Bildschirm aus.

Parent-Fenster und Kontrollelement können sich gegenseitig Botschaften senden. Die Botschaft der Kontrollelemente an die Window-Prozedur ihres Parent-Fensters ist WM_COMMAND. Dabei enthält *wParam* die ID des Kontrollelements. Der 32 Bit Parameter *lParam* besteht aus zwei Teilen, dem 16 Bit niederwertigen *LowWord* und dem ebenfalls 16 Bit höherwertigen *HighWord*. *LowWord* informiert über den Fenster-Handle des Kontrollelements und *HighWord* über den Aktionscode, der die Art der Aktion angibt. Diese kann beispielsweise ein Klick auf einen Knopf oder das Eingeben eines Buchstabens sein, wenn der Eingabefokus auf ein Editfenster zeigt.

Andererseits können auch Botschaften an das Kontrollelement selbst gesendet werden, um z.B. eine Radiokontrolle anzukreuzen. Beschriftungen sind jedoch eine Fenstereigenschaft und entsprechen der Titelleiste eines PopUp-Fensters. Änderungen werden hier mit einer Window-Funktion vorgenommen.

Das Aussehen der Kontrollelemente gehört zu den Eigenschaften des Systems und wird, falls gewünscht, systemglobal geändert. Es ist möglich, das Aussehen lokal, d.h. nur für das Anwendungsprogramm, neu festzulegen. Dies ist aber aus Konsistenzgründen nicht anzuraten.

1.3 Ressourcen

Der Begriff **Ressourcen** ist im Zusammenhang mit Windows 3.1 überladen. Einerseits beschreibt das Wort, wie im allgemeinen Sprachgebrauch üblich, die Hilfsmittel, die Programmen zur Verfügung stehen. Darunter fällt die Speicherverwaltung des Systems. Andererseits gibt es Objekte, wie Ikonen, Menüs oder Dialoge, die als Ressourcen definiert werden. Der Begriff ist in diesem Fall fest definiert. Es handelt sich um Daten, die in der .EXE Datei eines Programms gespeichert sind, aber keinen Speicherplatz im programmeigenen Datensegment belegen. Wenn von Ressourcen gesprochen wird, so bezieht sich der Begriff immer auf die letzte Definition.

Wie alle Windows-Programme nutzt auch ASA die Ressourcen des Windows-API. Das Hauptfenster von ASA und der ASA-Editor (vgl. Kapitel 6) haben je ein eigenes Menü, die Kommunikation zwischen Mensch und Programm erfolgt über Dialogfenster, usw.

Zunächst stellen wir im folgenden Abschnitt 1.3.1 die Speicherverwaltung des Windows-Systems vor. Sie bildet u.a. die Grundlage für das richtige Verständnis der an späterer Stelle besprochenen dynamischen Bilbliotheken (DLL's), die auch im ASA-Tool zur Anwendung kommen.

1.3.1 Speicherverwaltung

Ein System, welches gestattet, mehrere Prozesse nebeneinander laufen zu lassen, benötigt eine eigene **Speicherverwaltung**. Programme brauchen für ihren Code, ihre Daten und ih-

ren Stack Speicherplatz, der organisiert werden muß. Die Speicherverwaltung ist bei Window-Systemen davon abhängig, in welcher Betriebsart sich der Prozessor, und damit auch das System, befindet. Wir betrachten hier nur den *Protected Mode*, der ab dem Intel 80286 und seinen Nachfolgern zur Verfügung steht.

1.3.1.1 Segmente

Die Segmentierung des Speichers ist die grundlegende Eigenschaft der Speicherverwaltung von Windows. Jedem Windows-Programm wird beim Start ein Code-, Daten-, und Stacksegment zugeordnet, wobei die beiden letzteren zusammengelegt werden können, aber nicht müssen. Der Prozessor stellt mindestens vier Register zur Adressierung der Segmente zur Verfügung (Prozessoren ab dem 80386 haben noch zusätzliche Segmentregister):

- CS enthält die Adresse des Codesegments,
- DS die Adresse des Datensegments,
- SS die des Stacksegments und
- ES die eines frei verwendbaren „Extrasegments“.

Zeigervariablen, die sich auf das aktuelle Segment beziehen werden als *near* (16 Bit) bezeichnet. Die schon erwähnten *far*-Zeiger enthalten eine explizite Segmentadresse (32 Bit).

Die Segmentregister nehmen im *Protected Mode* sogenannte **Selektoren** auf. Der Prozessor interpretiert die höherwertigen 16 Bits eines Selektors als Index einer **Deskriptor-Tabelle**, deren Elemente jeweils 8 Bytes umfassen. Dadurch wird eine vollständige Beschreibung des jeweiligen Speicherbereichs erreicht. Hierbei wird auch eine **Basisadresse** festgelegt, die beim 80286 24 Bit umfaßt, woraus ein adressierbarer Gesamtspeicher von 16 MByte resultiert. 80386 Prozessoren erlauben 32 Bit Basisadressen und 4 GByte adressierbaren Gesamtspeicher. Die restlichen 5 Byte beim 80286, bzw. 4 Byte beim 80386, werden verwendet, um Informationen über die Größe und Art des Code-, oder Datensegments bereitzustellen. Hinzu kommt noch die Speicherung der **Privilegstufe**, die ein Programm haben muß, um auf den Speicherbereich zuzugreifen.

Speichersegmente können vom System verschoben werden, um Raum für andere Anwendungen zu schaffen, die diesen eventuell dringender brauchen. Da jeder Bezug auf ein Segment über die Deskriptor-Tabelle läuft, beschränkt sich die Korrektur von *far*-Zeigern auf eine Veränderung des Tabelleneintrags (Basisadresse).

Der gesamte von Windows verwaltete Speicher wird als **globaler Heap** bezeichnet. Er beginnt bei der Adresse, ab der das Window-System von DOS in den Hauptspeicher geladen wurde und endet mit der letzten Adresse des physikalisch vorhandenen Speichers (RAM).

Windows-Programme können mehrere Code- bzw. Datensegmente im globalen Heap haben, mindestens jedoch jeweils eines. Im Speichermodell *Small* (s.u.) werden das Daten- und Stacksegment zu einem **automatischen Datensegment** zusammengefaßt. Dabei zeigen DS und SS auf dasselbe Segment.

Jedes Segment hat bestimmte Attribute, die vor dem Übersetzen eines Windows-Programms in einer eigenen Modul-Definitionsdatei gespeichert werden. Das Attribut FIXED steht für unbewegliche, das Attribut MOVEABLE für bewegliche Segmente. Bewegliche, mit MOVEABLE attributierte Segmente können zusätzlich als DISCARDABLE, als entfernbar, gekennzeichnet werden. Bei Bedarf kann sie das System ganz aus dem Speicher entfernen. Dieses nimmt Windows nach dem LRU-Prinzip („Least Recently Used“) vor. Das Segment, das die längste Zeit nicht benutzt wurde, wird als erstes entfernt. Man verwendet DISCARDABLE nur für Segmente, die sich von der Festplatte oder Diskette nach-

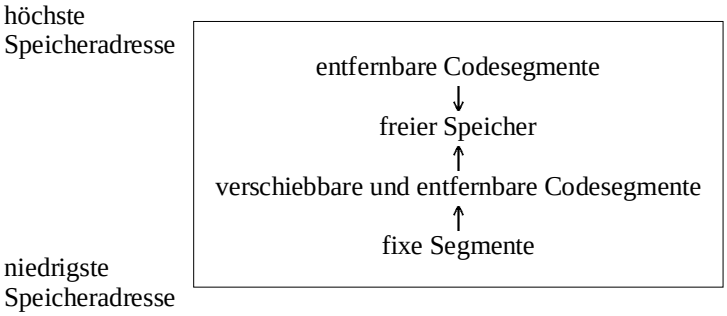


Abb. 1.3: Das Speicherlayout von Windows, aus [Pet92]

laden lassen, also die das Programm während der Laufzeit nicht verändert. Das ist bei Codesegmenten, Ressourcen, usw. der Fall, nicht aber bei Datensegmenten.

Windows ordnet die Segmente im globalen Heap wie in Abb. 1.3 gezeigt an. Beim Belegen von Speicherplatz für fixe Segmente versucht das System zuerst niedrige Speicheradressen zu vergeben. Findet sich im Speicherbereich niedriger Adressen kein Block, der ausreichend groß ist, so verschiebt Windows bewegliche Segmente in Speicherbereiche höherer Adressen. Erst wenn dann immer noch nicht genügend Speicher vorhanden ist, werden als zusätzlich entfernbar markierte Segmente aus dem Speicher gelöscht.

Die Suche nach freien Blöcken für verschiebbare Segmente beginnt oberhalb und endet unterhalb der entfernbaren Segmente. Läßt sich in diesem Fall kein Block ausreichender Größe finden, löscht das System entfernbare Segmente aus dem Speicher, schichtet die verbleibenden Segmente um und kann den dadurch freigewordenen Speicherplatz neu belegen.

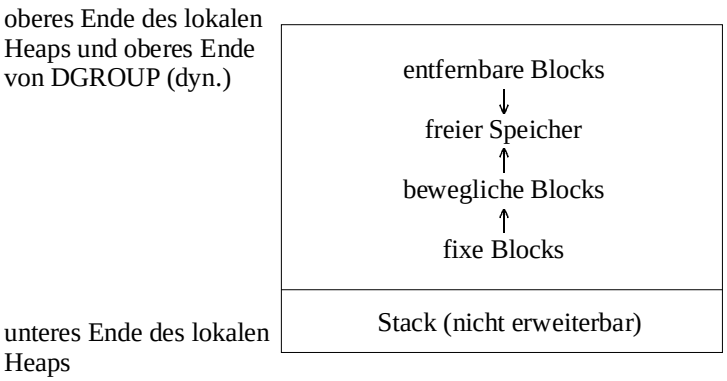


Abb. 1.4: Die Organisation des lokalen Heaps, aus [Pet92]

Der **lokale Heap** eines Windows-Programms (vgl. Abb. 1.4) besteht mindestens aus einem automatischen Datensegment, welches Daten- und Stacksegment miteinander vereint. Das

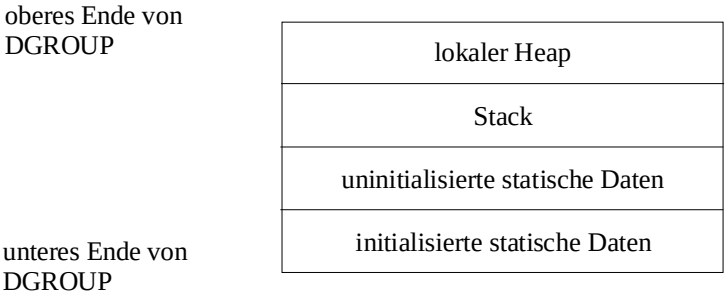


Abb. 1.5: Die vier Bereiche von DGROUP, aus [Pet92]

automatische Datensegment DGROUP im Speichermodell *Small* ist beweglich, aber nicht entfernbar und enthält vier Bereiche, wie in Abb. 1.5 veranschaulicht ist.

1.3.1.2 Speicherplatzbelegung

Für den globalen Heap des Systems und den lokalen Heap eines Programms gibt es jeweils eine Gruppe von Speicherverwaltungsfunktionen, die Windows zu Verfügung stellt, um die beiden Heaps zu organisieren bzw. Speicherplatz zu allokieren. Die Speicherbelegung läuft bei beiden Gruppen analog ab und berücksichtigt die Bewegungen der Segmente. Jeder Punkt (außer Punkt 3) entspricht dem Aufruf einer Funktion:

1. **Allokation:** Der Rückgabewert ist ein Handle zum belegten Block.
2. **Sperrung:** Der in Punkt 1 allokierte Block wird als unbeweglich gekennzeichnet und ein *far*-Zeiger beim globalen bzw. ein *near*-Zeiger beim lokalen Heap (in kleinen Speichermodellen) zurückgeliefert.
3. **Verwendung des Zeigers**.
4. **Rückname der Sperrung**.
5. **Freigeben des allokierten Speicherblocks**.

1.3.1.3 Speichermodelle

Die Anzahl der Segmente wird über das **Speichermodell** des Compilers festgelegt. Man kann in einem einzigen Windows-Programm mehrere Speichermodelle mischen. Es werden insgesamt fünf Speichermodelle angeboten:

Modell	**Codesegmente**	**Datensegmente**
Small	1 (max. 64 KByte)	1 (max. 64 KByte)
Medium	mehrere (unbegrenzt)	1 (max. 64 KByte)
Compact	1 (max. 64 KByte)	mehrere (unbegrenzt)
Large	mehrere (unbegrenzt)	mehrere (unbegrenzt)
Huge	mehrere (unbegrenzt)	mehrere (unbegrenzt)

Das Modell *Huge* unterscheidet sich von *Large* dadurch, daß es einzelne Variablen mit mehr als 64 KByte Platzbedarf unterstützt.

1.3.2 Ikonen, Cursorformen, Bitmaps und Strings

Wie bereits oben erwähnt belegen Ressourcen keinen Platz im Datensegment. Jede Ressource belegt vielmehr ein eigenes, als beweglich und meist entfernbar gekennzeichnetes Segment im Hauptspeicher. Sie werden dann geladen, wenn sie das Programm benötigt und wieder entfernt, wenn sie längere Zeit nicht benutzt werden. Es gibt insgesamt neun Ressourcenklassen: Ikonen, Cursorformen, Bitmaps, Zeichenketten, benutzerdefinierte Daten, Menüs und deren Abkürzungsbefehle, Dialoge und Zeichensätze. Die ersten fünf werden hier diskutiert, den anderen sind eigene Abschnitte gewidmet.

Ressourcen werden über eine spezielle ASCII-Datei, der **Ressourcen-Datei**, definiert und mit Namen versehen. Sie enthält einerseits direkte Definitionen, als auch Querverweise

auf andere Ressourcen-Dateien. Ein **Ressource-Compiler** übersetzt die Ressourcen-Datei, hängt den übersetzten Code an die ausführbare .EXE Datei des Programms und erzeugt eine Ressourcen-Tabelle, die im Dateikopf der ausführbaren Datei untergebracht wird.

Die erste Ressourcenklasse, der wir begegnen, ist die Klasse der **Bitmaps**. Das sind Bitarrays, bei denen Pixel durch ein oder mehrere Bits vertreten sind. Monochrome Bitmaps sind durch ein Bit pro Pixel vollständig definiert, wogegen bei farbigen Bitmaps je nach Farbtiefe mehrere Bits pro Pixel Verwendung finden. Beispielsweise sind bei 16 Farben 4 Bits pro Pi-xel nötig, um das Bitmap beschreiben zu können. Bitmaps dienen zwei Zwekken: dem Zeichnen von Bildern und der Darstellung von Füllmustern, z.B. für den Fensterhintergrund. Es gibt jedoch Unterschiede zu allen anderen Ressourcen. Sie sind GDI-Objekte (vgl. auch Abschnitt 1.4.3) und können zur Laufzeit des Programms verändert werden. Ihr Segment darf deshalb nicht als entfernbar gekennzeichnet sein.

Die Klassen der **Ikonen** und **Cursorformen** können gemeinsam untersucht werden. Ikonen dienen hauptsächlich der symbolhaften Darstellung von verkleinerten Fenstern, Cursorformen der Anpassung des Mauszeigers an veränderte Situationen. In beiden Fällen handelt es sich um Variationen von Bitmaps, aber sie können nicht zur Laufzeit verändert werden. Um auf jedem Bildschirm ein gleiches Aussehen zu haben, sind sie von der Bildschirmauflösung abhängig. Für jede Ikone bzw. jeden Cursor ist es notwendig, zwei Bitmaps abzuspeichern. Das erste repräsentiert das eigentliche Bild, das zweite Bitmap definiert eine Maske, die den Hintergrund durchscheinen läßt. Sie können lediglich monochrom (nicht unbedingt schwarzweiß) oder in 16 Farben definiert werden. Windows stellt eine Menge von Standardikonen und -cursorformen zur Verfügung, die durch selbstdefinierte Ressourcen ersetzt werden können.

String-Ressourcen dienen dazu, die Übersetzung von Programmen in andere Sprachen zu vereinfachen. Da Menüs und Dialoge auch als Ressourcen in der Ressourcen-Datei angegeben werden, kann man alle Texte, die ein Programm standardmäßig verwendet, in der Ressourcen-Datei zusammenfassen. Die Übersetzung erfolgt nur noch in einer Änderung einer einzigen Datei, der Ressourcen-Datei. String-Ressourcen haben bei sachgemäßer Verwendung noch einen weiteren Vorteil: Es läßt sich gegenüber der Definition von Strings im Quelltext Speicherplatz sparen. Hierbei macht man sich die Art der Speicherverwaltung zunutze. Das System faßt jeweils 16 Strings der **Stringtabelle** der Ressource-Datei zu einem separaten Segment zusammen. Dies geschieht in Abhängigkeit der Kennziffern, welche die Strings benennen. Von 0 bis 15 bezeichnete Strings kommen ins erste Segment, die mit 16 bis 31 bezeichneten ins zweite Segment usw. Wird ein String benötigt, so lädt das System das gesamte Segment, das den String enthält, in den Speicher. Versieht man die Strings mit Kennziffern, deren Gruppierung der Logik des Programms oder einem Modul entspricht und somit eventuell gleichzeitig benötigt wird, so wird diese Gruppe auf einmal in den Speicher geladen. Bei längerer Nichtverwendung kann Windows dieses Segment wieder aus dem Speicher nehmen.

Eine Ressourcenklasse läßt sich für Daten beliebiger Art definieren. Man spricht in diesem Fall von **benutzerdefinierten Ressourcen**, und sie lassen sich analog zu den eingebauten Ressourcen verwenden.

1.3.3 Menüs und Abkürzungsbefehle

Menüs gehören zu den Hauptmöglichkeiten, mit Windows-Programmen zu interagieren. Jedes Fenster kann maximal ein Menü unter seiner Titelleiste definieren. Die Menüstruktur, der sogenannte Menü-Prototyp, wird in der Ressourcen-Datei festgelegt, wobei jedem Wahlpunkt eine eindeutige Kennziffer zugesprochen wird. Die Zuordnung zum Fenster geschieht durch die Übergabe des ebenfalls eindeutigen Menünamens an die Fensterklasse.

Ein Menü besteht aus mehreren Ebenen. Die oberste Menüebene (Hauptmenü) ist die meist ständig sichtbare Menüleiste direkt unterhalb der Titelleiste. Jeder dort befindliche Wahlpunkt öffnet bei seiner Betätigung ein Untermenü („popup“), welches beliebig tief mit anderen Untermenüs geschachtelt sein kann. Die einzelnen Wahlpunkte können markiert, abgeschaltet, angeschaltet oder temporär nicht anwählbar sein.

Das System koordiniert das gesamte Verhalten eines Menüs, das Aufklappen der Untermenüs und das Selektieren der einzelnen Wahlpunkte. Wählt der Benutzer einen Menüpunkt aus, so sendet das System die Botschaft WM_COMMAND an die Window-Prozedur des dazugehörigen Fensters. Die Kennziffer des ausgewählten Menüpunkts erscheint in *wParam*, *lParam* ist höher- und niederwertig 0.

Es gibt eine weitere, wenn auch seltener verwendete Möglichkeit, Menüs zu definieren. Man baut das Menü durch das Aufrufen entsprechender Window-Funktionen zur Laufzeit schrittweise auf, ohne einen Prototyp in der Ressourcen-Datei festzulegen. Dieses Verfahren ermöglicht zur Laufzeit veränderbare Menüs.

Abkürzungsbefehle („accelerators“) sind Tastenkombinationen, die Botschaften des Typs WM_COMMAND (oder WM_SYSCOMMAND für das Systemmenü) erzeugen. Man verwendet sie meist, um die Auswahl bestimmter Wahlpunkte von der Tastatur vorzunehmen.

Definiert werden sie ebenfalls in der Ressourcen-Datei und erhalten die identische Kennziffer zu derjenigen, die der betreffende Wahlpunkt im Menü-Prototyp erhalten hat. Mit dieser Vorgehensweise ist es möglich, ohne zusätzlichen Programmieraufwand einen Menüwahlpunkt über die Tastatur anzusprechen, da identische Botschaften verschickt werden. Das Botschaftsattribut *wParam* enthält dementsprechend die Kennziffer des ausgewählten Wahlpunkts. Der niederwertige Teil von *lParam* ist ebenfalls 0, aber *HighWord* ist 1, wodurch das Programm zwischen der Auswahl eines Menüpunkts durch die Maus oder durch einen Abkürzungsbefehl unterscheiden kann.

Im Abschnitt 1.2.1 wurde über Tastaturereignisse gesprochen, die das System in die Warteschlange des Programms einreiht. Nun muß das Programm zwischen normalen Tastatureingaben und Abkürzungsbefehlen unterscheiden können. Die Lösung dieses Problems erfolgt in zwei Schritten:

1. Die Abkürzungsbefehle werden aus der Ressource-Datei in *WinMain* geladen, was vor dem ersten Aufruf der Ereignis-Warteschleife geschieht.
2. Die aus der Warteschlange entnommenen Tastaturereignisse werden mit Hilfe der Windows-Funktion *TranslateAccelator* mit den geladenen Abkürzungsbefehlen verglichen. Bleibt die Suche in der Accelator-Tabelle erfolglos, so wird die Botschaft nicht als Abkürzungsbefehl erkannt, und die Botschaftsbearbeitung geht auf normalem Weg weiter. Falls es sich aber um einen Abkürzungsbefehl handelt, dann sendet *TranslateAccelator* die Botschaft WM_COMMAND an die Window-Prozedur des Fensters, welches das Menü besitzt, unabhängig davon, welches Fenster den Eingabefokus hat.

1.3.4 Dialoge

Zur Abfrage von Informationen verwenden Programme meist **Dialoge**, die in Form eines eigenen Fensters erscheinen (vgl. Abb. 1.6). Zur Unterscheidung von normalen Fenstern haben sie ein standardisiertes Aussehen und enthalten meist mehrere Kontrollelemente. Ihr jeweiliger Prototyp in der Ressource-Datei enthält Informationen über ihre Größe und Aussehen, sowie über die verwendeten Kontrollelemente.

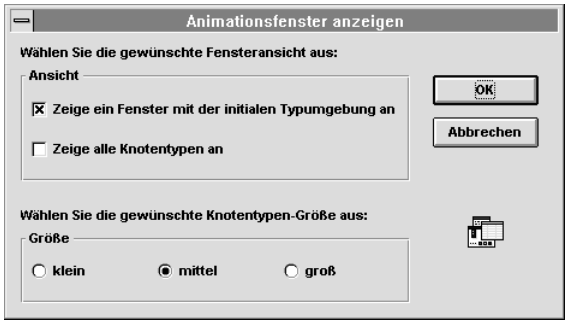


Abb. 1.6: Beispiel eines Dialogfensters. Die 3D-Effekte werden von Windows 3.1 selbst nicht unterstützt. Sie entstammen der Funktionalität der von Microsoft freigegebenen dynamischen Bibliothek CTL3DV2.DLL.

Für das Anlegen des Dialogfensters und dessen Kontrollelemente, das Erzeugen bzw. Verarbeiten von Botschaften und die Verarbeitung von Tastatur- und Mausereignissen ist der **Dialog-Manager** zuständig. Dabei handelt es sich um eine Gruppe von Funktionen, die das Einfügen von Dialogen in Programmen vereinfachen und standardisieren. Aus diesem Grund muß der/die ProgrammiererIn keine eigene Window-Prozedur für einen Dialog anlegen, sondern nur eine **Dialog-Prozedur**, die analog zu einer Window-Prozedur aufgebaut ist, aber wesentlich weniger Botschaften zugesendet bekommt. Die notwendige Window-Prozedur wird vom Dialog-Manager gestellt und erfüllt die Funktion eines Filters, der sich um die meisten Botschaften des Systems selbst kümmert und bestimmte Botschaften an die Dialog-Prozedur weitersendet. Diese ist nur noch für die Initialisierung der Kontrollelemente beim Start des Dialogs, das Bearbeiten von Botschaften, die diese Kontrollelemente ausgeben und für eventuelle Aufräumarbeiten beim Beenden des Dialogs zuständig.

Dialog-Prozeduren verarbeiten Botschaften genauso wie normale Window-Prozeduren, liefern aber für verarbeitete Botschaften den Wert *true* und für nichtverarbeitete Botschaften den Wert *false* an die darüberliegende Window-Prozedur des Dialog-Managers zurück, damit diese reagieren kann. Botschaften innerhalb eines modalen Dialogs (s.u.) verwenden eine eigene Warteschlange, nicht die des Programms; dadurch werden Verwechslungen, z.B. bei Abkürzungsbefehlen, vermieden. Obwohl modale Dialoge andere Programmaktivitäten verhindern, kann man aus der Dialog-Prozedur anderen Window-Prozeduren Nachrichten und Botschaften senden, die dann dort verarbeitet werden.

Weil eine Dialog-Prozedur nicht direkt mit einer Fensterklasse identifiziert werden kann, muß man sie vor dem Aufruf des Dialogs dem System bekanntmachen. Die dazu verwendete Window-Funktion *MakeProcInstance* erzeugt einen Vorspann für das korrekte Setzen des DS-Registers bei Aufrufen der Dialog-Prozedur und liefert die Adresse dieses Vorspanns zurück. Zu Beginn des Dialogs teilt man diese Adresse dem Dialog-Manager mit.

1.3.4.1 Modale Dialoge

Modale Dialoge erlauben bis zu ihrem Abschluß durch den/die AnwenderIn keine anderen Aktivitäten des Programms. Das System verwirft Ereignisse, die in die Warteschlange des Programms eingefügt werden. Eine Unterklasse der modalen Dialoge bilden die **systemmodalen Dialoge**. Sie blockieren das gesamte System bis zu dem Zeitpunkt, an dem sie geschlossen werden.

Windows liefert eine Bibliothek von Standard-Dialogen, z.B. der *Datei öffnen*-Dialog. Um einen solchen Dialog einzuleiten, muß der/die ProgrammiererIn lediglich eine entsprechende Window-Funktion, z.B. *GetOpenFileName*, aufrufen.

1.3.4.2 Moduslose Dialoge

Diese zweite Art von Dialogen erlaubt die Umschaltung zu anderen Fenstern und deren Menüs. Sie sind normalen Child-Fenstern sehr ähnlich und haben auch ein Systemmenü. Der Hauptunterschied zu modalen Dialogen ist das Fehlen einer eigenen Warteschlange. **Moduslose Dialoge** benutzen die Warteschlange des aufrufenden Programms. Ähnlich zu den Modifikationen der Ereignis-Warteschleife in *WinMain* bei Abkürzungsbefehlen muß dort getestet werden, ob die aktuelle Botschaft für den moduslosen Dialog bestimmt ist oder nicht. Die dazu verwendete Funktion *IsDialogMessage* prüft, ob die Botschaft für den moduslosen Dialog zuständig ist. Falls ja, dann übergibt sie die Botschaft an die Window-Prozedur des Dialogs, andernfalls geht die Botschaftsbearbeitung normal weiter.

1.4 Graphik

Das Thema dieser Arbeit ist die Visualisierung bzw. Animation der semantischen Analyse. Demzufolge spielen die Graphikfähigkeiten und -eigenschaften des Windows-Systems eine herausragende Rolle. Zu diesen Eigenschaften zählt beispielsweise, wie ein Programm den Anwendungsbereich eines Fensters zeichnet, die Textausgabe vornimmt oder Bitmapgraphiken behandelt, aus denen u.a. graphische Schaltflächen konstruiert werden können.

Unter Windows wird jedes Anwendungsprogramm mit einem Fensterobjekt assoziiert. Die **Graphikschnittstelle** von graphischen Benutzerumgebungen muß so konstruiert sein, daß das System den Anwendungsbereich eines Fensters zu jedem beliebigen Zeitpunkt neu zeichnen kann. Windows ist ein ereignisgesteuertes System und ein benutzerinitiiertes Öffnen einer Dialogbox oder Wechseln von Fensterüberlappungen erfordert ein Neuzeichnen des vormals verdeckten Teils eines Fensters.

Die Lösung dieses Problems ist eine Botschaft mit Namen WM_PAINT. Sowohl das System als auch ein Programm selbst ist in der Lage, diese Botschaft an das Fenster zu senden, dessen Anwendungsbereich ganz oder auch nur zum Teil neu gezeichnet werden soll. WM_PAINT ist nur für das Zeichnen des Anwendungsbereichs zuständig. Das Zeichnen der Titelleiste, des Menüs, des Fensterrahmens usw. übernimmt das System selbst. Aus diesem Grund wollen wir alles, was sich im Anwendungsbereich eines Fensters befindet, der Einfachheit halber, mit dem Wort *Fensterinhalt* bezeichnen. Man sagt, WM_PAINT informiert ein Fenster darüber, daß sein Inhalt teilweise oder komplett *ungültig* geworden ist.

Windows verwendet zur Bestimmung derartiger Teilbereiche ein Konstrukt, welches als *ungültiges Rechteck* bezeichnet wird. Intern ist dieses ungültige Rechteck unter anderem in einer jedem Fenster zugeordneten Struktur mit Zeicheninformationen enthalten. Als ProgrammiererIn muß man sich also darum kümmern, daß der gesamte Fensterinhalt beim Empfang der Botschaft WM_PAINT neu ausgegeben werden kann. Ist das ungültige Rechteck kleiner als der ganze Anwendungsbereich, so zeichnet das System automatisch nur den Bereich des ungültigen Rechtecks und setzt es auf Null, womit auch der Fensterinhalt als gültig deklariert ist. Es gibt immer nur genau ein ungültiges Rechteck im Anwendungsbereich eines Fensters. Tritt der Fall ein, daß die Warteschlange mehrere aufeinanderfolgende Botschaften dieses Typs enthält, so faßt sie das System zu einer einzigen zusammen und das ungültige Rechteck wird so neu berechnet, daß es alle Bereiche überdeckt.

1.4.1 GDI

Die Ausgabe von Text und Graphikelementen im Anwendungsbereich eines Fensters erfolgt über die Funktionen des **GDI** („Graphics Device Interface“). Eines der Hauptziele des GDI

ist eine möglichst weitgehende Geräteunabhängigkeit, die sich nicht auf Geräte einer einzigen Kategorie beschränkt. Es soll also nicht nur möglich sein, Unabhängigkeit von unterschiedlichen Bildschirmadaptern und ihren jeweiligen Treibern zu erreichen, sondern beispielsweise auch Unabhängigkeit von Bildschirmadaptern und Druckern. Deshalb sind Zeichenoperationen in erster Linie Routinen des GDI.

1.4.1.1 Eigenschaften

Zwischen Programm und GDI findet eine bidirektionale Kommunikation statt. Das Anwendungsprogramm kann nicht nur das GDI dazu anweisen, eine Zeichenoperation auszuführen, sondern auch Informationen über den aktuellen Treiber bzw. dessen Gerät abfragen.

Anwendungsprogramm $\leftrightarrow$ GDI $\leftrightarrow$ Treiber des Ausgabegerätes

Andererseits paßt sich das GDI an das jeweilige Ausgabegerät an. Bei einer Standard-VGA-Graphikkarte berechnet das GDI jeden zu zeichnenden Punkt von Linien, Kreisen, Rechtekken usw. selbst. Handelt es sich um eine Graphikkarte mit einem eigenen Coprozessor, so übermittelt das GDI lediglich die entsprechenden Befehlscodes und Koordinatenpaare. Die eigentlichen Zeichenoperationen berechnet der Prozessor der Karte, was als Ergebnis eine wesentlich höhere Ausgabegeschwindigkeit zur Folge hat.

Graphische Ausgabegeräte lassen sich allgemein in die zwei Klassen *raster-* und *vektororientiert* einteilen. Zur ersten Klasse gehören Bildschirme und Drucker, zur zweiten Klasse Plotter. Das GDI bietet nicht nur Unterstützung auf Vektorebene, sondern ebenfalls auf Rasterebene. Es ermöglicht Manipulationen einzelner Bits und Pixel, womit doch ein gewisser Grad an „Durchlässigkeit“ zur jeweiligen Hardware gewährleistet ist. Zusätzlich ergeben sich hier Geschwindigkeitsvorteile bei der Verwendung und Manipulation von Bitmaps.

Das GDI wurde hauptsächlich zur graphischen Unterstützung der Repräsentation der Fenster, Fensterelemente und Schriften konstruiert. In diesem Zusammenhang ist Windows ein eher statisches System und zweidimensional orientiert. Daraus ergeben sich einige Nachteile: Animationen werden nicht gut unterstützt. Das GDI bietet keinerlei Funktionalität zur räumlichen Darstellung von Objekten oder zur Rotation an. Beispielsweise müssen Halbachsen von Ellipsen immer parallel zur Horizontalen bzw. Vertikalen verlaufen. Hinzu kommt, daß Windows nur mit *signed integer* (ganze Zahlen im Bereich von -32768 bis 32767) statt mit Fließkommazahlen arbeitet. Daraus ergeben sich Darstellungsprobleme durch Rundungsfehler.

1.4.1.2 Gerätekontext

Bisher ist die Frage unbeantwortet geblieben, auf welche Art und Weise das GDI zwischen den Ausgabegeräten unterscheidet und gegebenenfalls umschaltet. Diese Aufgabe übernimmt der sogenannte **Gerätekontext**. Der Gerätekontext („Device Context“ = DC) ist eine interne Datenstruktur des GDI und ermöglicht die Verbindung zu einem Ausgabegerät, also Bildschirm, Drucker oder Plotter. Im Falle des Bildschirms gibt es noch eine zweite Verbindung, nämlich zum Anwendungsbereich eines Fensters. Um eine Vielzahl von Parametern einzusparen, wendet das Windows-System die im Gerätekontext gespeicherten Daten auf sämtliche GDI-Funktionen an. Daher muß man für eine Ausgabesequenz folgende Schritte vornehmen, die innerhalb ein und derselben Botschaft stattfinden müssen:

1. Ermittlung des Handles zum entsprechenden Gerätekontext.
2. Gewünschte Zeichenoperationen, einschließlich Textausgaben.

3. Freigabe des vormals ermittelten Handles.

Ein Gerätekontext enthält einerseits Informationen über das Gerät selbst, etwa Auflösung (Pixel), physikalische Dimensionen (Papiergröße) oder Anzahl der Farben. Andererseits sind im Gerätekontext sogenannte **Geräteattribute** enthalten.

Das System besetzt zunächst die Attribute bei der Kontextanforderung mit Standardvorgaben. Über entsprechende Aufrufe von GDI-Funktionen lassen sich diese aber abfragen oder ändern:

Attribut	**Standardvorgabe**	**Attribut**	**Standardvorgabe**
Koordinatensystem	MM_TEXT	Hintergrundmodus	OPAQUE
Fensterursprung	(0, 0)	Hintergrundfarbe	Weiß
Ursprung des Zeichenfensters	(0, 0)	Textfarbe	Schwarz
Fenstergröße	(1, 1)	Zeichenmodus	R2_COPYPEN
Größe des Zeichenfensters	(1, 1)	Objekte strecken	BLACKONWHITE
Stift	BLACK_PEN	Polygon-Füllmodus	ALTERNATE
Füllmuster	WHITE_BRUSH	Zeichenabstand	0
Schriftart	SYSTEM_FONT	Ursprung des Füllmusters	(0, 0) (Bildschirm)
Bitmap	None	Clip-Fenster	None
Zeichenposition	(0, 0)		

Erklärungsbedürftige Einträge, wie MM_TEXT, werden an geeigneter Stelle erläutert.

Zeichenstifte, Bitmaps, Regionen, Schriftarten, Paletten und Füllmuster sind **GDI-Objekte**, bei denen nicht ein einzelnes Programm, sondern das GDI selbst als Besitzer eingesetzt ist. Jedes von einem Programm definierte GDI-Objekt, das in einen Kontext eingesetzt ist, muß vor Beendigung des Programms wieder manuell freigegeben werden.

1.4.1.3 Koordinatensysteme

Windows verwendet standardmäßig ein auf Pixeln basierendes **Koordinatensystem**. Damit in diesem Fall die Unabhängigkeit von der Hardware erhalten bleibt, müssen Anwendungsprogramme die physikalische Auflösung des Ausgabegeräts über eine Anfrage beim entsprechenden Treiber ermitteln. Man kann aber das Koordinatensystem beliebig verändern und auf *virtuelle Koordinaten* wechseln.

Aufgrund dessen, daß in dieser Arbeit lediglich das Standard-Koordinatensystem von Windows MM_TEXT („Mapping Mode“ = MM) verwendet worden ist, sei an dieser Stelle nicht auf andere Möglichkeiten eingegangen und auf [Pet92] verwiesen.

Standardvorgaben im Modus MM_TEXT		
Fenster-Ursprung	(0, 0)	variabel
Zeichenfenster-Ursprung	(0, 0)	variabel

Standardvorgaben im Modus MM_TEXT		
Fenstergröße	(1, 1)	fix
Zeichenfenstergröße	(1, 1)	fix

Mit *Zeichenfenster* ist das physikalische und mit *Fenster* das logische Ausgabemedium gemeint. In der Tabelle versteht man unter dem Größeneintrag (1, 1) das Verhältnis zwischen logischen und physikalischen Werten. Unter MM_TEXT entsprechen sich Fenster- und Zeichenfenstergröße somit direkt, und das Größenverhältnis ist unveränderbar.

Da das GDI ausschließlich mit Integer-Werten arbeitet, sind Koordinaten als Parameter von GDI-Funktionen unanhängig vom aktuellen Koordinatensystem auf den Bereich von -32768 bis 32767 begrenzt. Eine weitere Einschränkung gibt es im Zusammenhang mit Rechtecken: der Start- und Endpunkt (links oben, rechts unten) ein und desselben Rechtecks dürfen nicht mehr als 32767 Einheiten voneinander entfernt sein.

1.4.2 Zeichenfunktionen

Dieser Abschnitt gibt eine Übersicht der vom GDI unterstützten **Zeichenoperationen**. Im allgemeinen verwenden diese GDI-Funktionen die im Gerätekontext festgelegten Attribute, etwa die Zeichenfarbe. Wie fast alle GDI-Funktionen, erwarten Zeichenfunktionen als Parameter die Übergabe eines Handles zu einem Gerätekontext und diverse Koordinatenangaben. Meist beschreiben diese Koordinatenangaben das zu zeichnende Objekt vollständig. Eine Ausnahme bildet *LineTo*, hier wird lediglich die Zielkoordinate angegeben, die Startkoordinate ist die aktuelle Zeichenposition im Fenster, welche mit der Funktion *MoveTo* verschoben werden kann.

Die einfachste Zeichenoperation ist das **Einfärben eines Punktes** (Pixels) mittels der Funktion *SetPixel*.

Windows erlaubt die Angabe insgesamt vier verschiedener Formen von **Linien**, deren Darstellung (Farbe, Strichbreite, usw.) von der aktuellen Geräteattributierung abhängt:

Funktion	Figur
LineTo	gerade Linien
Arc	elliptische Kreisbögen
PolyLine	Folge miteinander verbundenen Linien
PolyPolyLine	voneinander unabhängige Linien

Zusätzlich stellt das GDI sieben Funktionen zu Verfügung, mit denen sich **ausgefüllte Flächen mit Rändern** zeichnen lassen:

Funktion	Figur
Rectangle	Rechteck mit rechtwinkligen Ecken
Ellipse	Ellipse
RoundRect	Rechteck mit abgerundeten Ecken
Chord	Kreisbogen auf der Außenlinie einer Ellipse und Verbindung der Endpunkte durch eine Sehne

Funktion	Figur
Pie	Kreisbogen auf der Außenlinie einer Ellipse und Verbindung der Endpunkte mit dem Mittelpunkt („Kuchenstück“)
Polygon	Figur mit einer beliebigen Anzahl von Seiten
PolyPolygon	beliebige Anzahl von Figuren mit beliebiger Anzahl von Seiten

1.4.2.1 Betroffene Geräteattribute

Geräteattribute werden innerhalb von Programmen verschieden oft verändert. Während beispielsweise das Zeichenstift-Attribut recht häufig geändert wird, bleiben andere Attribute eher fest, etwa das verwendete Koordinatensystem. Ebenso werden die Geräteattribute bei der Verwendung der GDI-Zeichenfunktionen unterschiedlich in Anspruch genommen.

Die Funktion *SetPixel* erwartet als Parameter neben dem Handle zu einem Gerätekontext die Koordinaten des einzufärbenden Pixels und die explizite Angabe eines RGB-Wertes. Damit ist deren Funktionalität vollständig beschrieben.

Anders ist es bei der Darstellung der Linien und der Ränder von ausgefüllten Flächen. Hier sind fünf Attribute des Gerätekontextes zuständig:

Zeichenposition: Dieses Attribut wird nur von *LineTo* berücksichtigt.

Zeichenstift: Ein Zeichenstift (auch *Linien-Füllmuster*) steht für eine ganze Datensammlung. Darunter fallen die Zeichenfarbe, die Breite und ein Muster (durchgehend, gepunktet oder gestrichelt). Der Standardzeichenstift BLACK_PEN zeichnet eine durchgehende schwarze Linie, die unabhängig vom Koordinatensystem genau ein Pixel breit ist. Es lassen sich auch sogenannte *logische Zeichenstifte* erzeugen, die beliebig veränderbar sind.

Hintergrundfarbe: Zeichnet das System eine gestrichelte oder ähnlich unterbrochene Linie, so werden diese „Löcher“ mit der hier definierten Hintergrundfarbe ausgefüllt. Diese Operation ist jedoch auch vom Hintergrundmodus abhängig.

Hintergrundmodus: Es gibt zwei Hintergrundmodi, die Windows zur Verfügung stellt. Die Standardvorgabe ist OPAQUE (undurchsichtig). In diesem Fall werden Linienunterbrechungen mit der im Gerätekontext festgelegten Hintergrundfarbe gezeichnet. Im Modus TRANSPARENT hingegen überspringt die Zeichenoperation die „Löcher“ des Linienmusters und läßt den Bildspeicher unverändert.

Zeichenmodus: Der Zeichenmodus ist standardmäßig auf R2_COPYPEN gesetzt. Zeichenmodi geben die Art der logischen Verknüpfung zwischen Zeichenfarbe und dem Inhalt des Bildspeichers an den jeweilige Punkten an. Der Standardmodus R2_-COPYPEN steht für ein Kopieren der Zeichenfarbe, also ein Überschreiben des Zielgebiets ohne Berücksichtigung seines vorherigen Inhalts. R2 steht in diesem Zusammenhang für Rasteroperation auf zwei Bitmustern (auch ROP2). Ein Anwendungsbeispiel anderer Zeichenmodi wäre die Vermischung zweier Bitmuster zur Simulation eines Schattenwurfs.

Um Flächen auszufüllen, verwendet Windows ein sogenannte Füllmuster-Attribut. Für die Funktionen *Polygon* und *PolyPolygon* gilt ein spezieller Füllmodus:

Füllmuster: Bei einem Füllmuster („brush“) handelt es sich um ein 8×8 Pixel Bitmap, welches horinzontal und vertikal solange auf einen Füllbereich angewendet wird, bis er vollständig aufgefüllt ist. Füllmuster untergliedern sich in zwei Gruppen: durchgehen-

de Füllmuster, die jeweils einer Mischfarbe entsprechen und Schraffuren. Man kann Muster beider Gruppen beliebig definieren oder die von Windows standardmäßig bereitgestellten Muster benutzen. Standardvorgabe ist das durchgehende Füllmuster WHITE_BRUSH.

Polygon-Füllmodus: Windows stellt an dieser Stelle zwei Modi zur Verfügung, um festzusetzen, was bei der Anwendung einer der Funktionen *Polygon* oder *PolyPolygon* gefüllt wird. Der Standardmodus ALTERNATE veranlaßt das System dazu, lediglich die abgeschlossenen Flächen eines Polygons zu bearbeiten, die von außen her über das Durchqueren einer ungeraden Anzahl von Linien zu erreichen sind. Im Modus WINDING werden dagegen sämtliche Innenflächen eines Polygons gefüllt.

1.4.3 Bitmaps und Zwischendateien

Bitmaps und Zwischendateien verkörpern zwei unterschiedliche Methoden zur Speicherung von Bilddaten.

1.4.3.1 Bitmaps

Bitmaps sind vollständige digitale Repräsentationen eines Bildes, bei denen jedes Pixel durch mindestens ein Bit dargestellt wird. Die Anzahl der Bits ist von der Farbtiefe abhängig. Unter Windows verwendet man Bitmaps meist für Fotos oder sehr komplexe Bilder.

Das Kopieren vom Haupt- in der Bildspeicher geschieht bei Bitmaps mit hoher Geschwindigkeit, was der Hauptvorteil gegenüber anderen Bildformaten, einschließlich Zwischendateien ist. Andererseits liegt hier auch eine großer Nachteil, der große Speicherplatzbedarf von Bitmaps. Ein Screendump von 640×480 Pixeln benötigt als Bitmap beispielsweise mehr als 150 KByte Platz. Zudem sind Bitmaps geräteabhängig. Es kommt zu Problemen, wenn man ein Farbbitmap auf einem monochromen Bildschirm ausgeben will. Windows 3.1 kennt auch geräteunabhängige Bitmaps, sogenannte DIB's („Device Independent Bitmaps"), die aber im Gegensatz zu normalen Bitmaps keine GDI-Objekte sind.

Programme können Bitmaps als Ressourcen einbinden, aus Dateien lesen oder zur Laufzeit selbst erzeugen. Um eine gewisse Geräteunabhängigkeit zu erhalten, erlauben es fast alle GDI-Funktionen für die Bearbeitung und Darstellung von Bitmaps nicht, ein Bitmap unmittelbar in einen Gerätekontext einzusetzen. Windows verlangt dazu einen Umweg über einen *Speicherkontext*. Dieser Speicherkontext hat genau dieselben Eigenschaften wie ein Gerätekontext, außer daß er einen Speicherblock repräsentiert. Betrachten wir ein uninitialisiertes Bitmap, daß wir neu erzeugt und dessen Größe festgelegt haben. Setzt man dieses Bitmap in einen Speicherkontext ein, so erhält er vom System dieselben Ausmaße wie das eingesetzte Bitmap. Es wird somit gewissermaßen zur Zeichenfläche, die man beliebig mit Zeichenoperationen oder Textausgaben bearbeiten kann. Auf diese Weise lassen sich auch aus Dateien entnommene Bitmaps verändern.

Nun kann man mit sogenannten *Bitblock-Transfer-Funktionen* des GDI das Bitmap aus dem Speicherkontext in einen beliebigen Gerätekontext oder auch wieder in einen anderen Speicherkontext kopieren. Eine solche Funktion ist *BitBlt*. Sie erlaubt die Angabe der Position im Zielkontext und einer Rasteroperation (ROP), welche z.B. das Invertieren eines Bitmaps während des Kopiervorgangs vornimmt. Mit anderen Bitblock-Transfer-Funktionen, wie *StretchBlt* lassen sich Bitmaps dehnen, strecken, spiegeln oder skalieren. Die mit diesen Funktionen in Zusammenhang stehenden Kontextattribute sind:

Bitmap: Das in den Speicherkontext eingesetzte Bitmaps wird hier festgelegt. Windows setzt standardmäßig kein Bitmap in einen Speicherkontext ein und besetzt das Attribut mit dem Wert *None*.

Objekte strecken: Das Attribut gibt die Art und Weise an, wie *StretchBlt* bei Verkleinerungen zwei oder mehr Pixel miteinander kombiniert, wenn Pixel gelöscht werden. Die Standardvorgabe BLACKONWHITE führt ein logisches AND auf den Pixeln aus und erzeugt nur dann ein weißes Pixel, wenn alle beteiligten Pixel weiß waren (Schwarzflächen dominieren). WHITEONBLACK arbeitet mit einem logischen OR (Weißflächen dominieren). Bei Farbbitmaps verwendet man meist COLORONCOLOR. Hier geschieht keinerlei Kombination von Pixeln, falls ein Pixel eliminiert wird.

1.4.3.2 Zwischendateien

Zwischendateien („metafiles“) speichern Bilddaten als Folge von Datensätzen, die ihrerseits für Aufrufe von GDI-Funktionen stehen. Damit werden nicht die Bilddaten selbst, sondern die Erzeugung eines Bildes beschrieben. Zwischendateien verwendet man vorwiegend für technische Zeichnungen oder Texte, die manuell gezeichnet bzw. vom Programm generiert wurden, und die durch die Zwischenablage transportiert werden sollen (siehe Abschnitt 1.5.1). Wie Bitmaps können sich Zwischendateien auf der Festplatte oder im Hauptspeicher befinden. Metafiles auf der Festplatte tragen standardmäßig die Endung .WMF („Windows MetaFile“) und können auch von einem Programm als Ressource eingebunden werden. Sie verbrauchen wenig Speicherplatz und sind aufgrund ihrer Definition vom Ausgabegerät unabhängig.

Metafiles legt man über eine spezielle Kontextart, den *Metafile Device Context* an, welchen wir in dieser Arbeit kurz mit *Dateikontext* bezeichnen. Zwischen allen bisher vorgestellten Kontextarten machen die meisten GDI-Funktionen keinen Unterschied, da alle Kontextarten die gleichen Eigenschaften besitzen.

Die in einer Zwischendatei gespeicherten GDI-Aufrufe lassen sich innerhalb eines beliebigen Geräte- oder Speicherkontextes (z.B.: Bildschirm) beliebig oft sequentiell abspielen. Maßgeblich sind beim Abspielvorgang die Attribute des Zielkontextes, nicht die des Dateikontextes. Änderungen von Attributen, wie etwa der Zeichenfarbe oder des Koordinatensystems durch die Aufrufe in der Zwischendatei sind endgültig, d.h. bleiben auch nach dem Ende des Abspielvorgangs erhalten. Wenn dies unerwünscht ist, muß der originale Attributsatz vor dem Abspielen gespeichert werden. Eine Ausnahme zu dieser Regel bilden die GDI-Objekte, beispielsweise der Zeichenstift. Nach dem Abspielen wird der Originalzustand hier automatisch wiederhergestellt.

1.4.4 Texte

Das GDI stellt eine ganze Reihe von Funktionen für die **Textausgabe** zur Verfügung, um Text optimal an die gegenwärtigen Verhältnisse anzupassen. Wie fast alle GDI-Funktionen erwarten auch diese einen Handle auf einen Gerätekontext als Parameter. Wir wollen uns in dieser Arbeit mehr auf die Bildschirmausgabe konzentrieren und weniger auf die Ausgabe auf einem Drucker. Systeminterna bzgl. der Kommunikation mit Druckern und Plottern erfordern eine umfangreiche Beschreibung und können in [Pet92] genauer nachgelesen werden.

Die Basisfunktion zur Textausgabe ist *TextOut*, die einen Textstring innerhalb eines Gerätekontextes an einer bestimmten Startposition ausgibt. Eine erweiterte Funktion *Tabbed-*

TextOut unterstützt beliebige Tabulatorabstände, und *ExtTextOut* ermöglicht es, den Abstand zwischen einzelnen Zeichen („kerning“) individuell festzulegen. Mit der Funktion *DrawText* läßt sich statt einer Startposition ein Rechteck angeben, in dem der Text erscheinen soll. Sie arbeitet somit auf einer etwas abstrakteren Ebene und ermöglicht die nahezu komplette Formatierung des Textstrings innerhalb der Rechteckgrenzen, einschließlich Zeilenumbrüchen und Spaltenausgabe. Aufgrund dieser Eigenschaften eignet sie sich beispielsweise für Knotenbeschriftungen in der graphischen Repräsentation von Bäumen. Die Funktion *SetTextAlign* formatiert einen Text horizontal oder vertikal (und beeinflußt damit die Startposition), mit *GetTextExtent* lassen sich die Ausmaße eines Textstrings in Abhängigkeit von der Schriftart und -größe bestimmen.

Ebenso wie die Zeichenoperationen verwendet die Funktionen zur Textausgabe gewisse Attribute des eingesetzten Kontextes, um die Textdarstellung zu beeinflussen:

Textfarbe: Das System versieht die Textfarbe standardmäßig mit der Farbe Schwarz. Dieses Attribut des Gerätekontextes kann mit der Funktion *SetTextColor* durch Übergabe eines RGB-Wertes geändert werden.

Hintergrundfarbe: Hier läßt sich die Farbe festlegen, mit welcher der Zeichenhintergrund gezeichnet werden soll.

Hintergrundmodus: Im Standardmodus OPAQUE füllt das System die Räume innerhalb und zwischen den Zeichen mit der in den Kontext eingesetzten Hintergrundfarbe aus, während im Modus TRANSPARENT lediglich die Zeichen selbst, d.h. ohne Änderung des Hintergrundes, ausgegeben werden.

Zeichenabstand: Dieses Attribut des Kontextes betrifft den Abstand zwischen einzelnen Zeichen. Der Abstand wird in logischen Einheiten angegeben (Standard: 0), die den entsprechenden zusätzlichen Abstand zwischen zwei Zeichen angeben. Es ist nicht möglich, negative Werte anzugeben, um so etwa einen Text enger als vorgesehen zusammenzuschieben.

Die von Windows-Programmen verwendbaren **Schriften** lassen sich in zwei Kategorien einteilen, den

- **GDI-Schriften**, die üblicherweise von Videoadaptern verwendet werden. Man bezeichnet sie als *Font-Ressource*, und sie werden in Dateien mit den Endungen .FON oder .TTF (.FOT) abgespeichert. GDI-Schriften unterteilen sich in Bitmap-, Vektor-, und TrueType-Schriften.
- **Geräteschriften**, die meist von Druckern (ganz besonders Postscript-Laserdruckern) selbst definiert werden. Diese Schriftkategorie wird an dieser Stelle nicht weiter behandelt.

Bei **Bitmap-Schriften** (auch Raster-Schriften) sind die einzelnen Zeichen bitweise, in Form eines kleinen Bitmaps gespeichert. Durch diese Definition ergibt sich eine sehr schlechte Qualität beim Skalieren der Schrift, es entsteht ein „Klötzcheneffekt“. Eine Änderung der Schriftattribute (fett, kursiv, unterstrichen, durchgestrichen) erfolgt durch die Umrechnung der einzelnen Bits, es ist also kein eigener Zeichensatz hierfür erforderlich. Ein Vorteil ist die sehr hohe Ausgabegeschwindigkeit bei der Darstellung der Schrift und eine hohe Qualität, wenn man auf Skalierung verzichtet.

Vektor-Schriften definieren ein Zeichen als eine Folge von Linien, deren Anfangs- und Endpunkte abgespeichert werden. Sie lassen sich prinzipiell beliebig skalieren, jedoch wird die Breite der Linien hier nicht geändert, was bei großen Zeichen ein zu dünnes, bei kleinen

Zeichen ein unverhältnismäßig dickes Erscheinungsbild zur Folge hat. Eine Änderung der Schriftattribute erfolgt ebenfalls durch Umrechnung.

Die Zeichen von **TrueType-Schriften** werden durch *Umrisse* („outlines“) definiert, die selbst in Form mathematischer Linien- und Kurvenfragmente gespeichert sind. Beispielsweise besteht das Zeichen „o“ aus einem kleinen inneren und einem großen äußeren Kreis. Die Skalierung von TrueType-Schriften ist ohne Qualitätsverlust durchzuführen. Hier wird die Qualität durch sogenannte *hints* (Zusätze, Hinweise) weiter verbessert, die bei der Definition der Zeichen eingeführt werden. Eine Skalierung eines „o“ hat z.B. die Verbreiterung bzw. Verkleinerung der Strichbreite zur Folge. Diese Verbesserungen lassen sich aber erst beim Ausdruck deutlich erkennen. Anders als bei Bitmap- oder Vektor-Schriften existieren für jedes Schriftattribut eigene separate Definitionen. Durch die vielen Umrechnungen bedingt, liegt der Hauptnachteil der TrueType-Schriften in der etwas langsameren Ausgabegeschwindigkeit.

Zur Repräsentation der Schriftart im Programm wird eine *logische Schrift* erstellt. Dabei handelt es sich um eine Struktur, deren insgesamt 14 Elemente (z.B.: Schriftgröße, kursiv, Zeichenbreite, Name der Schrift,…) die Schrift definieren. Obwohl man diese Elemente manuell besetzen könnte, überläßt man diesen Vorgang dem System, indem man die Schrift mittels der beiden Funktionen *EnumFonts* bzw. *ChooseFont* beim System anfordert. *ChooseFont* unterscheidet sich von *EnumFonts* hauptsächlich dadurch, daß sie eine vordefinierte Dialogbox auf den Bildschirm bringt, in welcher der/die AnwenderIn eine Schrift auswählen kann. Setzt das Programm eine Schrift in den Gerätekontext ein, so wählt das Windows-System die Schriftart aus, die auf dem System existiert und der in den Gerätekontext eingesetzten Schriftart am nächsten kommt.

1.5 Datenaustausch und Kommunikation

Bisher hatten wir nur die interne Kommunikation eines Programms entweder mit seinen Child-Fenstern oder dem System selbst betrachtet. Nun stellt das Windows-System auch Verfahren zum Austausch von Daten und zur Verbindung zwischen mehreren verschiedenen Programmen zur Verfügung. Zu diesen Verfahren gehören die Zwischenablage, dynamischer Datenaustausch (DDE), dynamische Bilbliotheken (DLL’s) und Objektverknüpfung bzw. -einbettung (OLE). Eine Sonderstellung nehmen MDI-Programme ein, die mehrere Dokumente gleichzeitig verwalten können.

Die *Animation der semantischen Analyse* besteht aus zwei unabhängigen Programmen, die *animierte Präsentation* und das ASA-Tool, welche durch ein eigenes DDE-ähnliches Protokoll und eine DLL miteinander kommunizieren. Weiterhin verwendet das ASA-Tool eine frei verfügbare DLL für die 3D-Effekte innerhalb von Dialogen. ASA benutzt die Zwischenablage für Cut&Paste-Vorgänge im Editor und für das Kopieren der Animationsausgabe zur Weiterverarbeitung (z.B. Druck) in externe Graphikprogramme (*MS Paintbrush*).

1.5.1 Zwischenablage

Windows ermöglicht mit der **Zwischenablage** („clipboard“) einen einfach zu handhabenden Austausch von Daten zwischen verschiedenen Anwendungen. Sie stellt einen einzigen Speicherbereich variabler Größe dar, der allen Windows-Programmen gemeinsam zur Verfügung steht. Die Zwischenablage wird von jedem Programm verwendet, das Operationen wie Ausschneiden und Kopieren (Transfer in die Zwischenablage), oder Einfügen (Transfer aus der Zwischenablage) benutzt. Der Inhalt der Zwischenablage steht dann dem Programm selbst oder anderen Windows-Programmen für die weitere Bearbeitung zur Verfügung.

Damit sie für eine stabile Datenübertragung verwendbar ist, definiert Windows mehrere standardisierte Datenformate („clipboard format“):

CF_TEXT: Ein nullterminierter String, der Zeichen im ANSI-Code enthält und dessen einzelne Zeilen (optional) durch die Zeichenkombination CR/LF voneinander abgesetzt sind.

CF_SYLK + CF_DIF + CF_OEMTEXT: Diverse Textformate, die von verschiedenen Softwareprodukten unterstützt werden.

CF_BITMAP + CF_DIB: Geräteabhängige und -unabhängige Bitmaps.

CF_TIFF: Dieses Format steht für das *Tag Image File Format* (TIFF), welches eine Gemeinschaftsentwicklung von Microsoft, Aldus und Hewlett-Packard ist.

CF_METAFILEPICT: Bild in Form einer Zwischendatei („metafile“).

CF_PALETTE: Farbpalette, die sich meist auf ein geräteunabhängiges Bitmap (DIB) bezieht, d.h. dessen „reale“ Farben festlegt.

Eigene Formate: Windows-Programme können auch eigene Formate definieren, um so sicherzustellen, daß andere fremde Programme mit diesen Formaten nicht zurechtkommen und somit nicht auf sie zugreifen können. Ein Beispiel wäre ein Kopieren von Daten eines Dokuments in ein anderes Dokument desselben Programms.

Die Zwischenablage erwartet einen Handle auf ein frei bewegliches Segment im globalen Heap. Gegebenenfalls sind die Daten, die man in die Zwischenablage einsetzen will, in den globalen Heap zu kopieren. Hat man die Daten entsprechend vorbereitet, so sind für den Einfügevorgang folgende Schritte nötig (in Klammern sind die Funktionen benannt):

1. **Öffnen** der Zwischenablage (*OpenClipboard*). Damit wird der Zugriff auf den Inhalt der Zwischenablage durch andere Programme blockiert und der eigene Zugriff unter der Angabe des Fensterhandles angemeldet.
2. **Entleeren** (*EmptyClipboard*). Falls die Zwischenablage noch Daten enthält, sollten diese entfernt werden.
3. **Einfügen** der Daten (*SetClipboardData*). Die Funktion *SetClipboardData* übergibt den Handle auf den Speicherbereich, der die Daten enthält unter Angabe des Datenformats an das Clipboard. Der verwendete Handle darf vom Programm selbst nicht mehr benutzt werden, da er in den Besitz des Clipboards übergegangen ist. Es ist möglich diese Funktion mehrfach hintereinander auszuführen, aber immer nur mit verschiedenen Datenformaten. Die Zwischenablage verwaltet die dazugehörigen Segmente sämtlich auf einmal.
4. **Schließen** der Zwischenablage (*CloseClipboard*). Das System hebt die Blockade der Daten für andere Programme wieder auf.

Das Auslesen von Daten geschieht in ähnlicher Weise:

1. **Test**, ob in der Zwischenablage das richtige Format enthalten ist (*IsClipboardFormatAvailable*). Falls der Test negativ ausfällt, kann man den geplanten Zugriff abbrechen. Diese Funktion ist besonders nützlich, wenn die Zwischenablage mehrere Segmente unterschiedlichen Formats verwaltet.
2. **Öffnen** der Zwischenablage (*OpenClipboard*).
3. **Entnahme** der Daten (*GetClipboardData*). Die Funktion *GetClipboardData* erwartet zur Unterscheidung der benötigten Daten eine Formatangabe und liefert einen Handle

auf das Segment zurück, das die gewünschten Daten enthält. Auch hier ist mehrfacher Aufruf hintereinander möglich.

4. **Verwenden** der Daten. Dies muß vor dem Schließen der Zwischenablage erfolgen, weil dann die Daten in den Besitz der Zwischenablage übergehen. Eventuell ist es sinnvoll an dieser Stelle die Daten zu kopieren, um sie auch nach dem Schließen weiter verwenden zu können.
5. **Schließen** der Zwischenablage (*CloseClipboard*).

Intern geschieht diese Übergabe von Daten lediglich durch eine Änderung des Besitzers eines Speicherbereichs, dessen Bezeichnung Windows bei Segmenten im globalen Heap festhält. Die Funktion *GetClipboardData* setzt das auslesende Programm als Besitzer ein, wogegen zuvor *SetClipboardData* das Modul USER (die DLL, welche die Clipboard-Routinen enthält) als Besitzer der Daten eingesetzt hat. Nach dem Aufruf von *CloseClipboard* erhält die Zwischenablage die Besitzrechte zurück, um so wieder anderen Programmen einen Datenzugriff zu ermöglichen. Die Routinen der Zwischenablage stellen also eine Art Verwaltung gemeinsam verfügbarer Segmente im globalen Heap dar.

Im Umgang mit der Zwischenablage ist bei der Programmierung darauf zu achten, daß während des Zugriffs auf die Zwischenablage nur eigene Dialoge geöffnet werden, die keinen Einfluß auf die Zwischenablage haben. Beispielsweise verwenden dort definierte Editfenster die Zwischenablage implizit, d.h. es kann zu Problemen kommen, welche die Daten der Zwischenablage verändern bzw. löschen.

1.5.2 DDE

Eine Möglichkeit zur Kommunikation zwischen Prozessen ist der **dynamische Datenaustausch** („Dynamic Data Exchange“ = **DDE**). Zwei Programme (und damit zwei unterschiedliche Repräsentationen als Fenster) führen eine DDE-Konversation aus, indem sie sich gegenseitig Botschaften senden. Es ist also durchaus möglich, ein eigenes Kommunikationsprotokoll mit Hilfe selbstdefinierter Botschaften zu erstellen. Die meisten kommerziellen Softwareprodukte benutzen jedoch das von Windows bereitgestellte Protokoll DDE, um mit möglichst vielen anderen Produkten kommunizieren zu können.

Eine Anwendung, die bei einer anderen Daten anfordert, wird als **Client** bezeichnet, der Anbieter von Daten als **Server**. Der Client leitet die Kommunikation als erster ein. Er gibt eine Botschaft vom Typ WM_DDE_INITIATE als globaler Broadcast an alle im Hauptspeicher befindlichen Programme. *wParam* enthält (bei DDE-Botschaften immer) den Fensterhandle des Senders, *lParam* die Art der gewünschten Daten. Verfügt ein sich im Speicher befindendes Programm über solche Daten, so erklärt es sich zum Server und leitet die eigentliche Konversation ein. Dazu legt der Server ein extra verstecktes Child-Fenster auf Basis einer eigenen Fensterklasse und damit einer eigenen Window-Prozedur an, die die gesamte weitere Kommunikation mit dem Client übernimmt.

Auch hier geschieht der eigentliche Datentransfer über Segmente des globalen Heaps, die durch Aufrufe der Funktion *GlobalAlloc* mit dem Flag GMEM_DDESHARE belegt werden müssen.

1.5.2.1 Atomtabelle

Die Identifizierung der Daten geschieht über drei Strings, die für jeden Server exakt zu dokumentieren sind:

- **Anwendung** („application“): Name des Servers

- **Thema** („topic“): Ein Oberbegriff für das gewünschte Datum
- **Datum** („item“): Die einzelnen Datenelemente, die zu einem Thema verfügbar sind.

Nun übergibt das System bei einer DDE-Botschaft nicht diese drei Strings für Anwendung, Thema und Datum, sondern sogenannte **Atome**, die als Botschaftsparameter (*lParam*) mitgesandt werden. Ein Atom ist ein 16 Bit *unsigned integer* Wert, über den Windows die Bearbeitung von Zeichenketten erlaubt. Die Zeichenketten sind in der globalen **Atomtabelle** des Systems gespeichert und mit den entsprechenden Atomen verknüpft (Es existieren auch Definitionen für lokale Atomtabellen, die im programmeigenen Datensegment angelegt werden können, was in diesem Zusammenhang von nur untergeordnetem Interesse ist, da wir Kommunikation zwischen verschiedenen Prozessen untersuchen wollen). Ein Windows-Programm kann diese globale Stringverwaltung des Systems mit folgenden vier Funktionen ansprechen:

Name	**Beschreibung**
GlobalAddAtom	fügt der Atomtabelle des Systems einen String hinzu (erhöht bzw. den zugeordneten Zähler) und liefert ein Atom zurück.
GlobalDeleteAtom	erniedrigt den einem Atom zugeordneten Zähler in der Atomtabelle des Systems und löscht den String zusammen mit dem Atom, wenn der Wert 0 erreicht ist.
GlobalFindAtom	sucht die Atomtabelle des Systems nach einem String ab und liefert das dazugehörige Atom bzw. den Wert 0 zurück.
GlobalGetAtomName	sucht die Atomtabelle des Systems nach einem Atom ab und kopiert den dazugehörigen String.

Die globale Atomtabelle ist in einem Datensegment einer DLL des Systems angelegt, welches von allen Programmen gemeinsam benutzt werden kann. Jedes Programm ist für die gemeinsame Atomtabelle verantwortlich, d.h. dafür zuständig, daß es keine Atome löscht, die ein anderes Programm noch benötigt, oder umgekehrt, eigene Atome nicht mehr freigibt. Hält sich eine Anwendung nicht an diese Vorgabe, so reagiert das Window-System mit Instabilitäten.

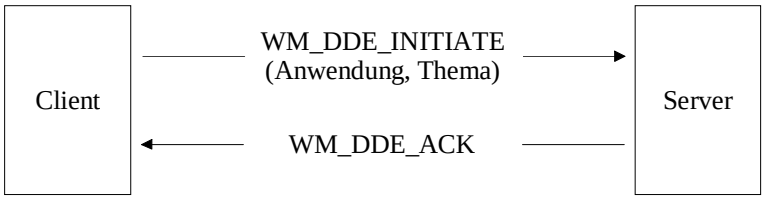
1.5.2.2 Arten der Konversation

Die Wahl, welche Art der Verbindung zwei Programme miteinander eingehen, ist von den Eigenschaften der Daten abhängig:

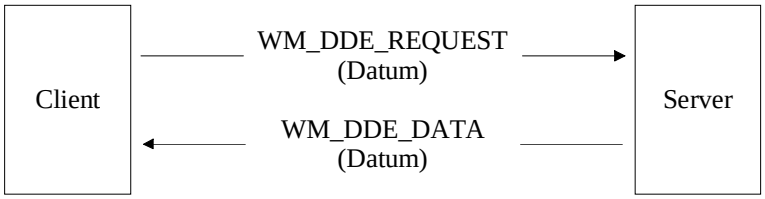
- Fixe, statische Daten fragt der Client über eine sogenannte **„kalte“ Verbindung** ab.
- Daten, die sich permanent dynamisch verändern, erfordern eine **„heiße“ Verbindung**. Sie werden dynamisch bei einer Änderung aktualisiert.
- Eine Mischung zwischen beiden Konversationsarten ist die **„warme“ Verbindung**. Der Client wird lediglich bei einer Veränderung der Daten darüber benachrichtigt. Erst wenn dieser zu einer Aktualisierung seine Zustimmung gibt, gibt der Server die Daten aus.

Ein Client baut eine **„kalte“ Verbindung** durch die Ausgabe der Botschaft WM_DDE_-INITIATE unter der Angabe von Anwendung und Thema auf. Eine sich als Server erklä-

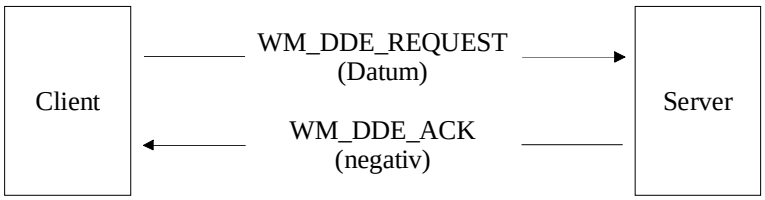
rende Anwendung reagiert mit dem Senden einer Bestätigungsbotschaft WM_DDE_ACK an den Client:



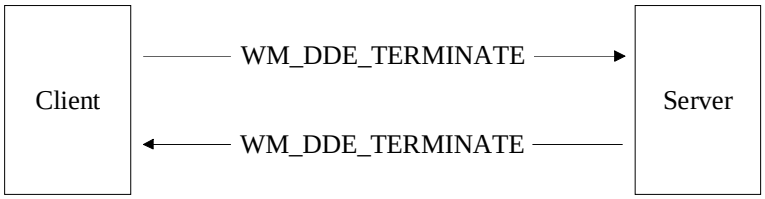
Nun gibt der Client eine Anfrage WM_DDE_REQUEST über ein spezifisches Datum des gewählten Themas aus, worauf der Server das gewünschte Datum mit der Botschaft WM_DDE_DATA zurückliefert, wenn er darüber verfügt:



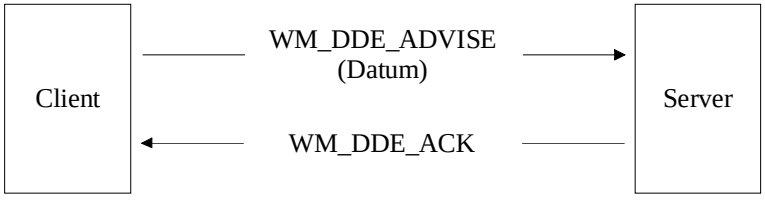
Falls der Server nicht über das Datum verfügt, so bestätigt er die Anfrage des Client mit einer Negation:



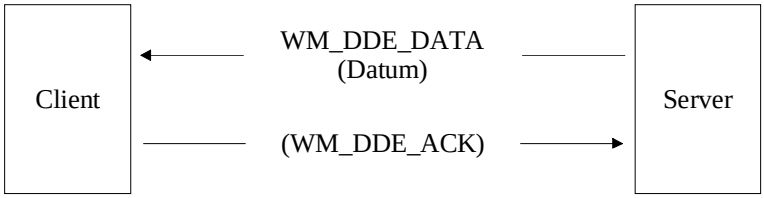
Dieses Wechselspiel zwischen Anfragen des Client und Reaktionen des Servers kann beliebig oft wiederholt werden, bis eines der beteiligten Programme die Verbindung durch das Senden der Botschaft WM_DDE_TERMINATE beendet. Als Bestätigung erfolgt ein Rücksenden dieser Botschaft durch das andere Programm:



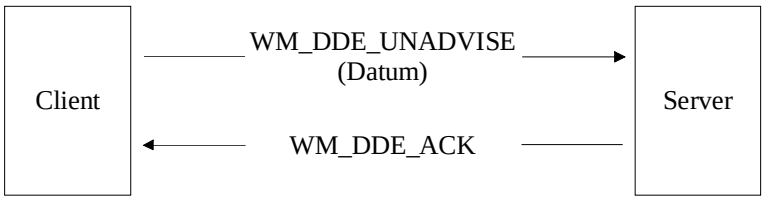
Der Aufbau einer **„heißen“ Verbindung** ist identisch zu dem einer „kalten“ Verbindung. Nachdem ein Server bestätigt hat, bemüht sich der Client um ein Datum, jedoch mit einer Botschaft WM_DDE_ADVISE. In Abhängigkeit davon, ob der Server über das Datum verfügt, akzeptiert er positiv oder negativ mit der Botschaft WM_DDE_ACK:



Bei positiver Bestätigung verpflichtet sich der Server, den Client bei jeder Veränderung dieses Datums mit einer unaufgeforderten Botschaft WM_DDE_DATA zu informieren. Eine Bestätigung des Client ist optional:

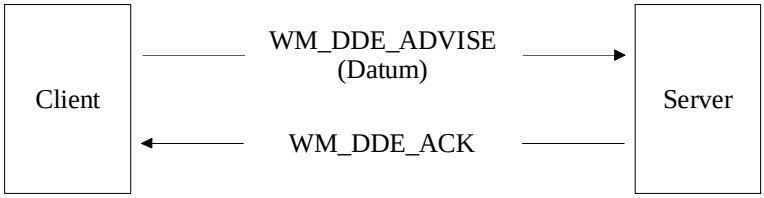


Möchte der Client die permanente Aktualisierung eines bestimmten Datums durch den Server aufheben, so benachrichtigt er ihn mit der Botschaft WM_DDE_UNADVISE, was der Server wieder mit WM_DDE_ACK bestätigen muß:

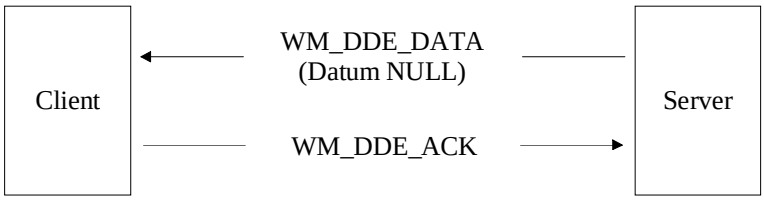


Die Terminierung der Verbindung erfolgt, wie bereits beschrieben, mit WM_DDE_-TERMINATE.

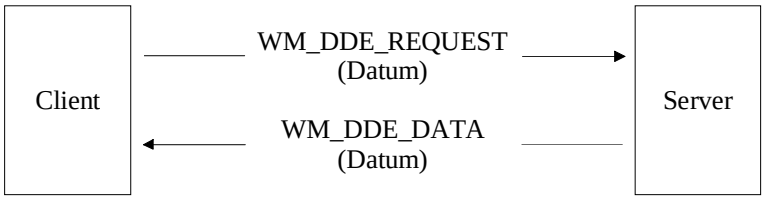
Die Konversation zweier Programme über eine **„warme" Verbindung** beginnt über WM_DDE_INITIATE bis zur Sendung der Botschaft WM_DDE_ADVISE in identischer Weise, wie die „heiße" Verbindung:



Dieser Botschaft ist hier allerdings mit einem Flag verbunden, das den Server anweist, anstelle der veränderten Daten selbst nur ein Signal zu senden. Es besteht aus der Botschaft WM_DDE_DATA mit dem Wert NULL:



Nachdem der Client auf eine Veränderung der Daten aufmerksam geworden ist, kann er sie zu einem beliebigen späteren Zeitpunkt über den für statische Daten bekannten Mechanismus abfragen:



Das Aufheben der Signalsendungen und die Beendigung der Konversation geschieht wie bei der „heißen" Verbindung mittels WM_DDE_UNADVISE und WM_DDE_TERMINATE.

Zusammen mit Windows 3.1 hat Microsoft eine dynamische Bibliothek mit Namen DDEML.DLL ausgeliefert, die das erläuterte „klassische" DDE durch ein Makropaket vereinfacht. DDEML beinhaltet insgesamt 27 Funktionen und 16 Transaktionstypen, deren

Erklärung den Rahmen dieser Arbeit sprengen würde. An der internen Funktionsweise ändert sich durch die Verwendung dieser Bibliothek nichts (siehe [Pet92]).

1.5.3 DLL

Im Rahmen der konventionellen Programmierung sucht der Linker im Quelltext nach Bibliotheksfunktionen und bindet die entsprechenden Routinen aus einer statischen Bibliothek (*.LIB) hinzu. Ein anderes Konzept verfolgt Windows bei **dynamischen Bibliotheken** (**DLL** = „Dynamic Link Library“). Erst zur Laufzeit werden die dynamischen Bibliotheken bei Bedarf in den Hauptspeicher geladen und die entsprechenden Bibliotheksfunktionen ausgeführt. Um eine DLL zu laden, sucht Windows zuerst im aktuellen Verzeichnis, dann im SYSTEM-Verzeichnis von Windows, im Windows-eigenen Verzeichnis und schließlich in allen Verzeichnissen, die über die PATH-Umgebungsvariable von DOS festgelegt wurden. DLL-Dateien können einen beliebigen Suffix im Dateinamen haben, er lautet aber meist doch .DLL. Das Konzept der dynamischen Bibliotheken hat folgende Vorteile:

- **Weniger Hauptspeicherbelastung**, da eine DLL höchstens genau einmal in den Speicher geladen wird. Mehrere Programme können auf ein und dieselbe im Speicher befindliche Bibliothek zugreifen.
- **Verbesserte Modularisierung**. Bestimmte Funktionen, die mehrere Programme gemeinsam benötigen, können in eine DLL gepackt werden.
- **Leichteres Updaten**. Bereits existierende Programme lassen sich durch einfachen Austausch von DLL’s aktualisieren, ohne das Hauptprogramm neu zu übersetzen. Rechtlich gesehen kann man DLL’s separat lizensieren.

1.5.3.1 Eigenschaften

In einer ausführbaren Datei (*.EXE. *.DLL) findet das System die Eigenschaften des entsprechenden Programms. Diese Informationen unterteilt man in fünf Abschnitte, wobei je nach Art der Datei der eine oder andere Abschnitt vollständig fehlen kann:

1. den .EXE-Dateikopf des alten DOS-Formats
2. den „neuen“ .EXE-Dateikopf von Windows
3. eine Liste der Code- und Datensegmente, die die Datei definiert
4. die von der Datei exportierten Funktionen
5. die Relozierungsdaten für die einzelnen Segmente

Wir wollen uns nur denjenigen Abschnitten zuwenden, die im Zusammenhang mit DLL’s von Belang sind. Im Dateikopf (2) findet sich u.a. ein Hinweis darauf, daß es sich bei der Datei um eine DLL handelt. In Windows-Programmen ist dort der Bezeichner MODULE eingetragen, in DLL’s LIBRARY.

Abschnitt (4) definiert die *exportierten* Funktionen, also die Funktionen, die anderen Programmen zur Verfügung gestellt werden. Jede dieser Funktionen ist mit einer Ordinalzahl als Kennziffer (Referenznummer) assoziiert. Der Aufruf einer solchen Funktion (z.B. die Zeichenoperation *Rectangle* aus GDI.EXE) durch ein beliebiges Programm oder auch einer DLL *importiert* die entsprechende Funktion. Alle zu importierenden Funktionen werden in Abschnitt (5) mit dem Namen des Bibliotheksmoduls aufgelistet. Lädt das System ein Programm, so überprüft es diesen Abschnitt (5) und bringt auch die angeforderten DLL’s in den Speicher, falls sie sich dort noch nicht befinden. Windows paßt die Aufrufe der DLL-

Funktionen im importierenden Programm an die tatsächlichen Speicheradressen an, indem ein Adressierungsvorspann für jede importierte Funktion erzeugt wird, der auf das Codesegment der DLL zeigt (vollständige Adressierung mit *far*-Zeigern).

In Abschnitt 1.1.3 haben wir gesehen, daß jedes Windows-Programm mindestens ein Datensegment (das automatische Datensegment einschließlich Stack) zugewiesen bekommt. Für jede Programmkopie (Instanz), die sich im Speicher befindet, hat dann das System ein eigenes separates Datensegment angelegt. Dynamische Bibliotheken hingegen können sich nur einmal im Speicher befinden. Ihr Datensegment wird von allen Programmen gemeinsam verwendet. Sie definieren keinen eigenen Stack, ihre Routinen benutzen grundsätzlich den Stack des jeweiligen aufrufenden Programms. Für die bereits erwähnten Segmentregister gilt nun: DS ≠ SS.

Daraus ergibt sich, daß unter Windows Stack-Umschaltungen nur bei einem Wechsel zwischen Prozessen stattfinden, d.h. die Aufrufe von Funktionen in dynamischen Bibliotheken werden vom System nicht als Prozeßwechsel angesehen.

1.5.3.2 Programmierung

Die Programmierung von DLL's erfordert einige Änderungen im Vergleich zur Programmierung von Windows-Programmen: Sie besitzen keine Hauptfunktion im herkömmlichen Sinn wie *WinMain*. Deren Analogon bei DLL's heißt *LibMain* und ist lediglich zu Initialisierungszwecken notwendig. Sie enthält keine Ereignis-Warteschleife, da dynamische Bibliotheken keine Botschaften zugesendet bekommen. Nach der Initialisierung gibt sie die Kontrolle an das System zurück und verbleibt im Speicher, bis das System die DLL wieder aus dem Speicher entfernt.

Deshalb benötigen dynamische Bibliotheken einen eigenen Exitcode, genannt *Window Exit Procedure* (WEP). Diese Funktion wird von Windows aufgerufen, entweder wenn das System heruntergefahren wird, oder wenn das letzte Programm beendet wurde, das diese dynamische Bibliothek benutzt. Sie übernimmt die Freigabe aller noch belegten Speicherbereiche.

Im Umgang mit Funktionen muß man darauf achten, daß mit *far*-Zeigern gearbeitet wird, um aus dem Datensegment des importierenden Programms kopieren zu können. Auf der Seite der DLL werden Funktionen durch den Zusatz *_export* anderen Programmen zur Verfügung gestellt, d.h. exportiert, z.B.: *int far _pascal _export XYZ (…)* (*_pascal* ist ein bestimmtes Übergabeformat für Funktionsparameter. Eine solche Funktion erwartet die Parameter in umgekehrter Reihenfolge auf dem Stack, als es die C-Konventionen verlangen). Mit diesen Zusätzen müssen auch Window-Prozeduren ausgestattet werden, damit das System sie zurückrufen kann.

Große Nachteile bei der Programmierung von DLL's enstehen durch die segmentierte Architektur der 80×86 Prozessoren und der Tatsache, daß DS ≠ SS gilt. In den Speichermodellen *Small* und *Compact* haben wir *near*-Zeiger (16 Bit, Offset) und *far*-Zeiger (32 Bit, Segmentadresse + Offset). Die Zuordnung zu Stack- und Datensegment geschieht dann wie folgt:

Variablen eines C-Programms, die außerhalb von Funktionen als *extern* und innerhalb von Funktionen als *static* deklariert wurden, kommen ins Datensegment. Es werden *near*-Zeiger verwendet und das DS-Register zur Adressierung vorausgesetzt.

Funktionsparameter und Variablen, die innerhalb von Funktionen deklariert wurden (nicht *static*), werden auf dem Stack abgelegt. Der Compiler benutzt hier ebenfalls *near*-Zeiger und setzt das SS-Register voraus.

Hat man nun einen *near*-Zeiger, so kann das System nicht entscheiden, ob er auf das Datensegment oder auf den Stack zeigt. Die Modelle *Small* und *Compact* können mit *near*-

Zeigern arbeiten und für ein in diesen Modellen übersetztes Programm gilt DS = SS, d.h es kommt hier zu keinerlei Adressierungsproblemen, weil Stack- und Datensegment zusammenfallen. Es gibt ebenso keine Probleme bei Programmen mit höheren Speichermodellen, da dort immer *far*-Zeiger zur Adressierung verwendet werden.

Hat man nun eine DLL, so gilt aber immer DS ≠ SS und damit haben wir ein grundsätzliches Adressierungproblem. Es kann nur durch explizites Setzen von *static* bzw. *far* bei der Deklaration von Variablen umgangen werden.

1.5.4 OLE 2.0

Mit der Version 3.1 stellt Windows die objektorientierte Benutzerschnittstelle **OLE** („Object Linking and Embedding“) in der nun zweiten Version zur Verfügung. Um die an OLE beteiligten Anwendungsprogramme unterscheiden zu können, führen wir folgende Begriffe ein:

- Ein **OLE-Server (Serveranwendung)** ist das Programm, dessen OLE-Objekt in das Dokument (z.B. ein Text) eines anderen Programms eingebettet werden kann.
- **OLE-Container (Clientanwendungen)** sind Programme, die in der Lage sind, OLE-Objekte anderer Anwendungen aufzunehmen.

OLE-Objekte können die verschiedensten Formen annehmen, etwa Kalkulationstabellen und Diagramme (*MS Excel*), Textdokumente (*MS Word*), Bitmaps (*MS Paintbrush*), Audiodateien, Animationen, Videos, etc. In Klammern sind mögliche populäre Serveranwendungen angegeben.

Dokumente, die Daten anderer Programme enthalten, bezeichnet man als **Verbunddokumente**. Die Clientanwendung, also das Programm, welches ein OLE-Objekt aufgenommen hat, muß sich nicht um dessen Aufbereitung (Anzeige, Druck, usw.) kümmern; dies übernimmt der OLE-Server. Man bettet also nicht nur das Objekt selbst, sondern auch die Bearbeitungsfunktionen des Servers in die Clientanwendung mit ein. Klickt der/die BenutzerIn beispielsweise innerhalb des Clients auf das verbundene OLE-Objekt, so kann die Serveranwendung mit dem ausgewählten Objekt als Dokument des Servers gestartet werden, etwa um Veränderungen am Objekt vorzunehmen.

Windows bietet unterschiedliche Konzepte an, welche die Interaktion mittels OLE zwischen verschiedenen Programmen regeln:

Verbinden und Einbetten („linking and embedding“)**:** Um ein Objekt und ein Dokument zu vereinigen, gibt es diese zwei Möglichkeiten. Bei verbundenen Objekten werden dessen Daten in einer separaten Datei abgelegt. Der Client erhält nur noch eine Referenz auf diese Datei und auf den Server. Dagegen werden eingebettete Objekte im Dokument des Clients selbst als Kopie des Objekts gespeichert.

In-Place-Aktivierung: Wählt der/die BenutzerIn das OLE-Objekt im Verbunddokument aus, so erscheint in der Clientapplikation die Menüleiste und die Benutzerschnittstelle des OLE-Servers.

Automation: Dieses Verfahren ermöglicht es, daß eine Applikation (der Automations-Client) eine andere Applikation (den Automations-Server) fernsteuern kann.

Verbunddateien oder -dokumente: In diesen Dateien lassen sich die Daten verschiedenster Programme ablegen.

Drag-and-Drop: Dieser Mechanismus erlaubt den Transfer von Daten von einer Applikation zur anderen (oder auch innerhalb ein und derselben Applikation) unter Zuhilfe-

nahme der Maus. Man „greift“ mit der Maus Daten, die meist mit Ikonen symbolisiert sind, bewegt sie mit gedrückter linker Maustaste zu der Zielposition und läßt sie dort sozusagen „fallen“.

[Pet92] enthält leider nichts über OLE 2.0. Einen kleinen Überblick geben [Asy94a] und [FA94].

1.5.5 MDI

Bisher hatten wir unausgesprochen immer nur Programme betrachtet, die lediglich mit einem einziges Dokument (Text, Bild, etc.) arbeiten. Dies ist durch die Standardschnittstelle *Single Document Interface* (SDI) von Windows vorgegeben. Das *Multiple Document Interface* (**MDI**) beschreibt eine Fensterstruktur und eine Benutzeroberfläche, die das Verwalten und Bearbeiten mehrerer Dokumente innerhalb einer einzelnen Programmkopie erlaubt. Es stellt somit zum einen

- eine Schnittstelle, zum anderen
- eine Spezifikation für Windows-Programme

dar. Ein MDI-Programm kann mehrere Dokumente über separate Fenster in ein und demselben Anwendungsbereich verwalten. Eine Beispielanwendung ist *Microsoft Word*, welches mehrere Textdokumente in seinem Anwendungsbereich halten kann.

Das Hauptfenster einer MDI-Anwendung, in diesem Zusammenhang auch als *Rahmenfenster* („frame window“) bezeichnet, unterscheidet sich bis auf den Anwendungsbereich nicht von Hauptfenstern üblicher SDI-Anwendungen. Der Anwendungsbereich wird nicht direkt für Programmausgaben benutzt, sondern nur zur Darstellung einer beliebigen Anzahl von sogenannten *Dokumentenfenstern*. Deshalb benennen wir den Anwendungsbereich auch mit *Arbeitsbereich* („working space“). Die Dokumentenfenster besitzen keine eigene Menüleiste, alle Funktionen wickelt das System über die Menüleiste des Hauptfensters ab. Sie können nur innerhalb des Arbeitsbereichs existieren und lassen sich dort zu Ikonen verkleinern. Ferner kann zu jedem Zeitpunkt immer nur ein Dokumentenfenster aktiv sein.

Für das Rahmenfenster muß eine eigene Fensterklasse (und damit eine eigene Window-Prozedur) definiert werden. Den Arbeitsbereich deckt das MDI mit einem sogenannten *Client-Fenster* ab, für das Windows eine eigene Fensterklasse MDICLIENT definiert und intern eine entsprechende Window-Prozedur bereitstellt, die alle mit MDI anfallenden Verwaltungsaufgaben übernimmt. Ob man pro Dokumentenfenster eine eigene Window-Prozedur benötigt oder für alle Dokumentenfenster eine einzige, ist von der Art der An-

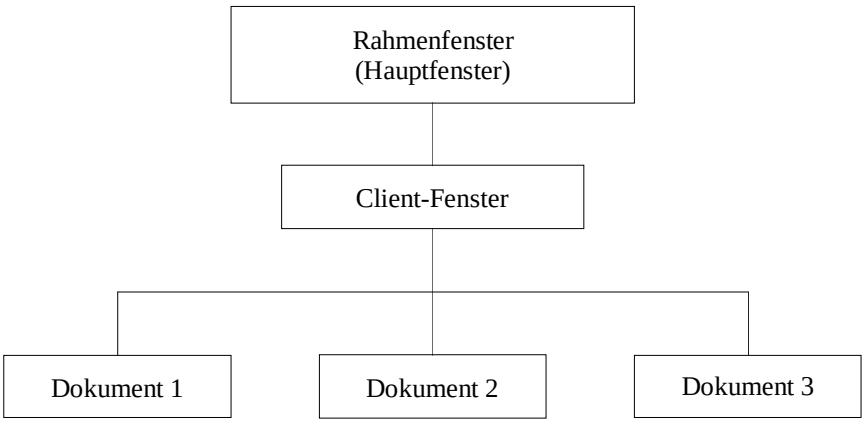


Abb. 1.7: Fensterhierarchie in MDI-Programmen mit drei Dokumentenfenstern, aus [Pet92]

wendung abhängig und dem/der ProgrammiererIn überlassen. Abbildung 1.7 zeigt die Fensterhierarchie in MDI-Programmen mit mehreren Dokumentenfenstern.

Wir wollen nun annehmen, daß für alle Dokumentenfenster eine einzige Window-Prozedur zuständig ist, die die Verwaltung und Darstellung der Dokumentendaten übernimmt. Eine beliebte Möglichkeit zur Speicherung dieser Daten ist die Reservierung von Speicherplatz in der Fensterstruktur, die bei der Definition der Fensterklasse vorgenommen wird. Jedes auf dieser Klasse basierende Dokumentenfenster erhält einen Handle zu einem Speicherbereich auf dem lokalen oder globalen Heap, in welchem es seine Daten ablegen kann.

Die Window-Prozedur des Rahmenfensters ist dagegen hauptsächlich für die Menüverwaltung und damit für Botschaften des Typs WM_COMMAND zuständig. Außerdem muß sie, falls nötig, dynamisch die einzelnen Dokumentenfenster erzeugen, was durch das Senden einer Botschaft WM_MDICREATE an das Client-Fenster geschieht. Der Botschaftsparameter *lParam* ist ein Zeiger auf eine spezielle Struktur MDICREATESTRUCT, welche die Eigenschaften des zu erzeugenden Dokumentenfensters beschreibt. Der 16 Bit Parameter *wParam* wird auf 0 gesetzt. Analog ist die Botschaft WM_MDIDESTROY für den Abbau eines Dokumentenfensters zuständig. Hier enthält *wParam* den Fensterhandle des zu löschenden Dokumentenfensters und *lParam* den Wert 0.

Ein MDI-Programm definiert standardmäßig eine Vielzahl von Abkürzungsbefehlen, um beispielsweise zwischen Dokumenten hin- und herzuschalten, sie zu verkleinern, etc. Daher muß die Ereignis-Warteschleife des Programms in geeigneter Weise modifiziert werden. Diese vom System vordefinierten Abkürzungsbefehle müssen von den programmeigenen Abkürzungsbefehlen unterschieden werden. Eine derartige Unterscheidung nimmt die MDI-Funktion *TranslateMDISysAccel* vor, die die erkannten Befehle an das Client-Fenster der MDI-Anwendung und damit an die interne Clientwindow-Prozedur sendet.

Kapitel 2

Asymetrix™ Multimedia ToolBook™ 3.0

2.1 Autorensysteme

Unter **Autorensystemen** versteht man Programmierhilfsmittel, mit denen ein Autor aus inhaltlichen und didaktischen Strukturen ein Programm erstellen und überarbeiten kann. Sie sind das zentrale Entwicklungswerkzeug für Lernprogramme. Moderne, oberflächenbasierte Autorensysteme (z.B.: MTB 3.0) eignen sich darüber hinaus auch für andere Anwendungen, wie Präsentationen, Produktdemos, Multimediakataloge, Online-Lexika etc.

2.1.1 Arten von Autorensystemen

Autorensysteme lassen sich nach drei Gesichtspunkten klassifizieren:

- **Art der Programmierung**
  - *Autorensysteme mit Autorenführung*: Systeme dieser Gruppe führen den Autor mittels Anweisungen durch eine Funktionsauswahl. Ein Lernprogramm wird nur durch diese Menü- bzw. Formularführung erzeugt.
  - *Autorensysteme mit Autorensprache*: Diese Systeme stellen eine eigene Programmiersprache zur Verfügung, deren Funktionalität auf die Erfordernisse der Entwicklung von Lernprogrammen ausgerichtet ist.
  - *Kombinierte Autorensysteme*: Hier werden die Eigenschaften der vorgenannten Gruppen gemischt. Das System unterstützt den Autor durch bestimmte Auswahlanweisungen, bietet jedoch zusätzlich die Möglichkeit an, Quellcode zu erzeugen bzw. zu manipulieren.
- **Technologisches Umfeld**
  - *Klassisch*: Damit sind vorallem ältere Systeme gemeint, die keine standardisierten graphischen Benutzeroberflächen aufweisen. Die Interaktion mit dem Autor/Lerner verläuft hauptsächlich auf Textbasis.
  - *Objektorientierte Oberflächen*: Neue graphische Oberflächen, wie Windows, erlauben eine standardisierte und damit leicht beherrschbare Nutzung von Programmen. Die einzelnen Bestandteile eines Lernprogramms werden als Objekte verstanden und als solche angesprochen.

- **Benutzeroberfläche**
  - *Getrennte Autoren- und Lerneroberfläche*: Die Oberfläche, auf der ein Autor ein Lernprogramm entwickelt, unterscheidet sich von der Lerneroberfläche, auf der ein Lerner mit dem Lernprogramm arbeitet.
  - *Vereinte Autoren- und Lerneroberfläche*: Hier fallen Autoren- und Lerneroberfläche zusammen. Durch entsprechende Modi läßt sich zwischen ihnen unterscheiden.

2.1.2 MTB 3.0 als ein Vertreter von Autorensystemen

Multimedia ToolBook 3.0 (in dieser Arbeit kurz MTB oder ToolBook) ist eine objektorientierte Oberfläche, deren Autoren- und Lerneroberfläche zusammenfällt. Es gehört zur Gruppe der kombinierten Autorensysteme. Sowohl zum Erstellen als auch zum Ausführen von Anwendungen bietet ToolBook eine interaktive Umgebung.

ToolBook stellt Zeichenmittel zur Verfügung, mit deren Hilfe die visuelle Benutzeroberfläche einer Anwendung erstellt werden kann. Durch die mitgelieferte Programmiersprache OPENSCRIPT legt der Autor die Eigenschaften und das Verhalten der einzelnen Objekte fest.

ToolBook (Runtime-Version) wird auch zum Ausführen einer Anwendung verwendet. Dabei steuert ToolBook die Interaktion zwischen der Anwendung und dem Windows-System, etwa das Verarbeiten von Mausklicks (vgl. Abb. 2.1).

ToolBook bietet durch **Multimedia** eine zusätzliche Möglichkeit, Informationen effizient zu übermitteln. Unter Multimedia versteht man allgemein Informationen in unterschiedlichen Formaten (Text, Graphik, Ton, Video und Animationen). Um Multimedia in ToolBook-Anwendungen zu integrieren, sind folgende Elemente bereitzustellen:

- **Multimedia-Quellen:** Darunter versteht man *Media-Dateien* (z.B.: Klang- oder Videodateien) und *Media-Geräte* (Videokassetten, Bildplatten, Audio-CD's, etc.).
- **Hardware:** Einige Arten von Multimedia benötigen zusätzliche Hardware, um abgespielt werden zu können (Soundkarte, Lautsprecher, CD-ROM-Laufwerk, usw.).
- **Gerätetreiber:** Für alle Arten von Multimedia sind die geeigneten Gerätetreiber bzw. Software-Schnittstellen zu installieren. Dieser Treiber genügen in der Regel dem MCI-Standard („Media Control Interface").

Da Text und Graphik schon durch das Windows-System zur Verfügung gestellt werden,

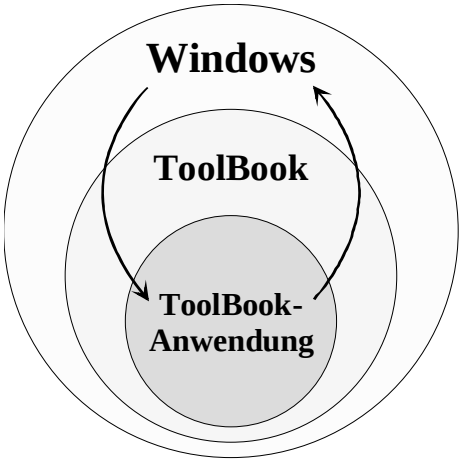


Abb. 2.1: ToolBook als Schnittstelle zwischen Anwendung und System, aus [Asy94a]

wollen wir nur kurz auf die restlichen Multimedia-Formate eingehen:

Ton: ToolBook bietet die Nutzung von Ton in drei unterschiedlichen Formaten an, *Klangdateien* (*.WAV, „Wave Audio“), *MIDI-Dateien* (*.MID, „Musical Instrument Digital interface“) und *CD-Audio*.

Video: Es können ebenfalls drei verschiedene Quellen für Videosequenzen verwendet werden. Darunter gehören *digitale Videodateien* (z.B.: AVI-Dateien, MPEG-Dateien etc.), *Bildplatten* und *Videobänder*.

Animationen: Bei *Animationen* wird eine Reihe von Standbildern, die in der Regel auf Zeichnungen basieren, in schneller Folge abgespielt. Hierzu ist keine besondere Hardware erforderlich, nur der entsprechende Gerätetreiber.

Informationen zu MTB 3.0 gibt das zugehörige Benutzerhandbuch [Asy94a] von Asymetrix. Dessen Multimediakomponenten und -funktionalität beschreibt das entsprechende Handbuch [Asy94b], welches zudem auch eine OPENSCRIPT-Referenz enthält.

2.2 Aufbau von MTB-Programmen

ToolBook ist auf der Metapher eines Buches aufgebaut, die als Basis für ToolBook-Anwendungen dient.

2.2.1 Bücher, Seiten, Objekte

MTB-Programme bestehen aus ein oder mehreren **Büchern** (TBK-Dateien). Jedes Buch unterteilt sich wie ein gedrucktes Buch in **Seiten**, wobei jede Seite einer Fensterseite (Fensterinhalt) direkt entspricht. Die Fenster, in denen die Seiten erscheinen, werden in ToolBook als **Ansichtsobjekte** bezeichnet. Es können zum gleichen Zeitpunkt mehrere Ansichtsobjekte geöffnet sein.

Eine Seite kann Felder, Schaltflächen und Grafiken enthalten. Jedes Objekt auf einer Seite sowie die Seite selbst ist ein **Objekt**. Mehrere Objekte lassen sich zu einer **Gruppe** zusammenschließen, die selbst als ein einzelnes Objekt angesehen wird. Um ein konsistentes Erscheinungsbild der Seiten zu erhalten, kann man einen Teil der Objekte, nämlich die Objekte die auf mehreren Seiten gleich angezeigt werden sollen, in dem **Hintergrund** anlegen, der von beliebig vielen Seiten verwendet wird. Diese Objekte erscheinen dann auf allen Seiten, die diesen Hintergrund verwenden, was auch mit geringerem Platzverbrauch verbunden ist, da nur ein Objekt für mehrere Seiten erzeugt wird. Eine sinnvolle Anwendung sind beispielsweise Navigationstasten, mit deren Hilfe man sich durch das Programm bewegt.

Die Seiten eines Buches müssen nicht alle dieselbe Größe haben, sie lassen sich an die verschiedenen Ansichtsobjekte in der Größe anpassen. Außerdem können die Seiten in beliebiger Reihenfolge angeordnet werden. Die Struktur des Buches bestimmt, wie der/die BenutzerIn durch die einzelnen Seiten navigiert. Die eigentliche Seitenwechselsteuerung kann über verschiedenen Arten verlaufen:

- Menübefehle
- Richtungstasten
- Schalter in der Statuszeile
- OPENSCRIPT-Befehle

Die maximale Seitengröße beträgt 35,6 × 35,6 cm (14 Zoll im Quadrat), weiterhin ist die Zahl der auf einer Seite befindlichen Objekte begrenzt.

Ein Buch besteht aus einem (Hauptfenster) oder mehreren Ansichtsobjekten, die alle bei Bedarf mit Menüleisten und anderen Fensterelementen (Systemmenü, Verkleinerungsknöpfe, etc.) versehen werden können, ganz nach dem Zweck, dem sie dienen:

- Dialogfelder
- abgetrennte Paletten
- Schalterleisten
- Statuszeilen
- Popup-Graphiken oder Textfelder
- Logo- und Startbildschirm
- Fenster, in denen andere Buchseiten angezeigt werden

Ansichtsobjekte sind also Fenster, wie sie in Kapitel 1 definiert wurden. Sie bestehen aus einem Rahmen, Anwendungsbereich und einem sogenannten Anwendungsfenster, das den Teil des Anwendungsbereichs umfaßt, in dem Seiten und Hintergründe angezeigt werden.

2.2.2 Objekte und Eigenschaften

ToolBook ist eine *objektorientierte Umgebung*. Sämtliche visuellen Elemente einer Anwendung (Schaltflächen, Felder, Ansichtsobjekte, Seiten, Hintergründe, Gruppen, etc.) sind **Objekte**, die jeweils eine Menge von **Eigenschaften** haben, die das Erscheinungsbild und das Verhalten der Objekte festlegen. Solche Eigenschaften sind etwa die Position, die Größe, die Farbe oder die Art, wie Daten in Felder eingegeben und angezeigt werden.

Das ToolBook-System verfügt über eine Menge von bereits vordefinierten Objekten, die von dem/der AutorIn ausgewählt und in die Anwendung eingebaut werden können.

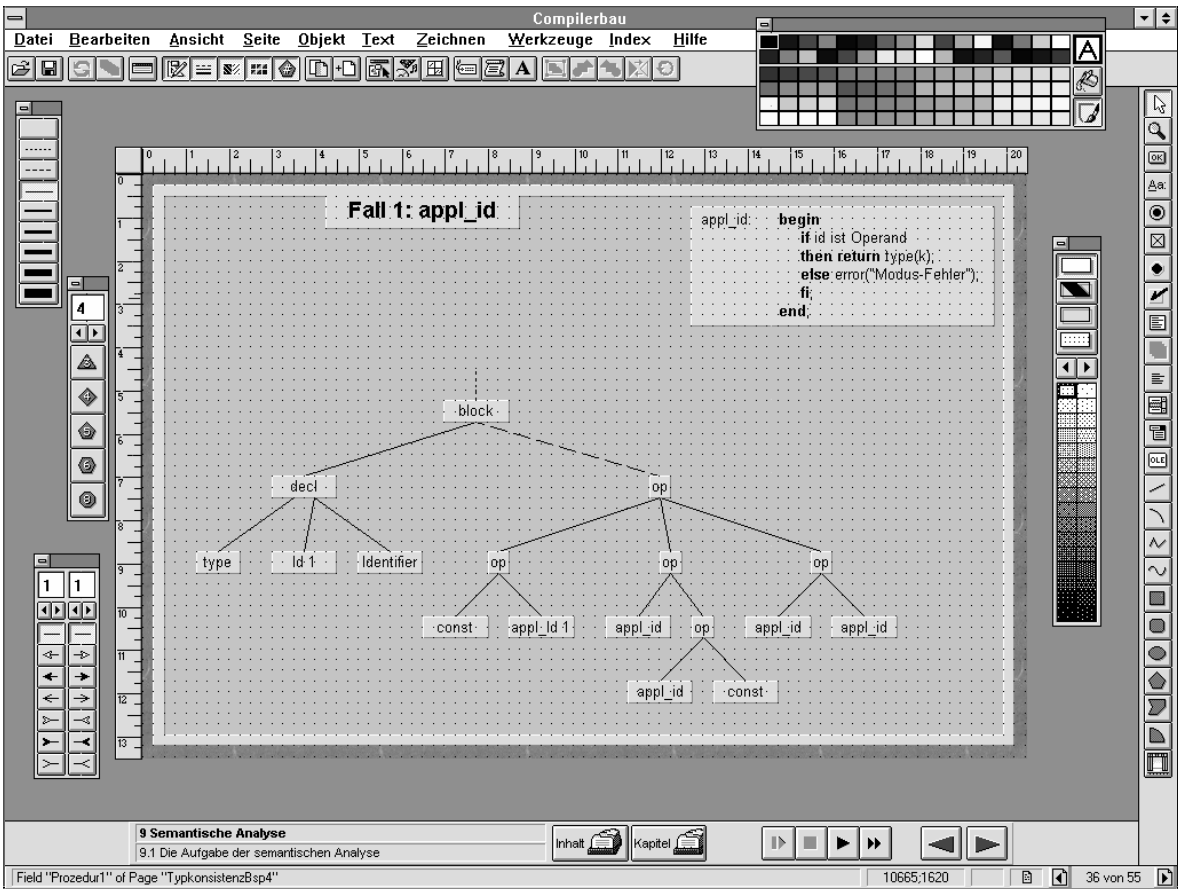
Außer den vordefinierten Eigenschaften können von dem/der AutorIn auch selbstdefinierte *User-Properties* eingeführt werden.

2.2.3 Autoren- und Leserebene

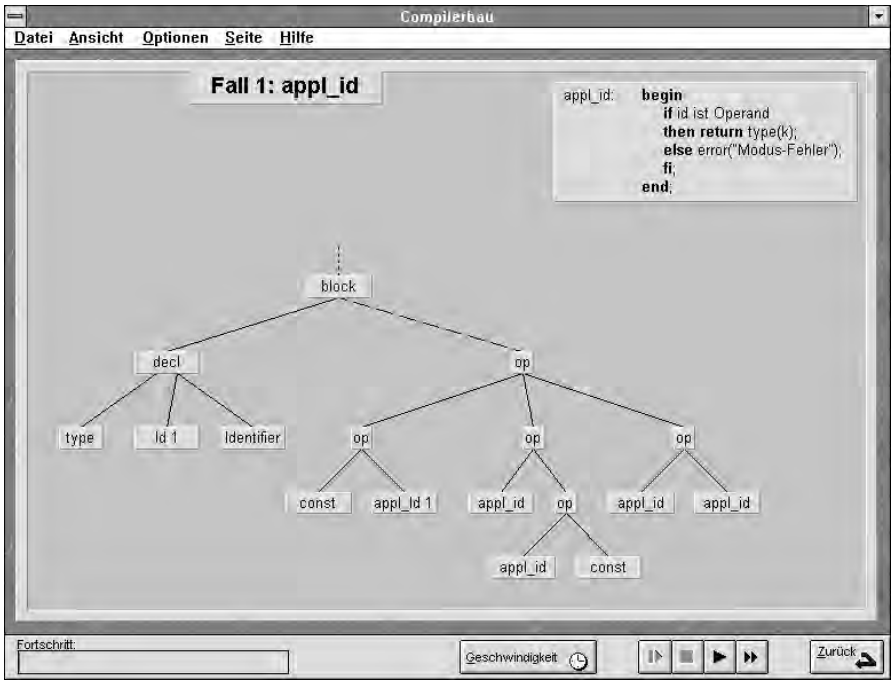
Wir haben erwähnt, daß ToolBook Autoren- und Lerneroberfläche vereint (vgl. Abb. 2.2). Anwendungen werden auf der sogenannten **Autorenebene** entwickelt, auf der die verschiedenen Zeichenhilfsmittel und Programmierwerkzeuge zur Verfügung stehen. Hier lassen sich die gewünschten Objekte erstellen und Programme in OPENSCRIPT schreiben.

Die BenutzerInnen einer ToolBook-Anwendung führen diese auf der **Leserebene** aus. In diesem Modus stehen alle für die Ausführung der Anwendung erforderlichen Funktionen zur Verfügung, aber nicht die Programmierwerkzeuge.

Auch während der Programmentwicklung kann jederzeit zwischen Autoren- und Leserebene umgeschaltet werden, um die Anwendung zu testen. Zum Vertrieb der fertigen Anwendung stellt Asymetrix eine frei verfügbare Runtime-Version von ToolBook zur Verfügung, die nur das Arbeiten auf der Leserebene zuläßt.



a) Autorenebene mit verschiedenen Hilfsmittelpaletten



b) Leserebene

Abb. 2.2: Autoren- und Leserebene von MTB 3.0

2.3 Programmierung in ToolBook

Einige Objekte weisen ein von ToolBook vordefiniertes Verhalten auf. Befindet sich beispielsweise der Cursor über einem Textfeld, dann zeigt ToolBook bei jedem Tastendruck das entsprechende Zeichen im Feld an.

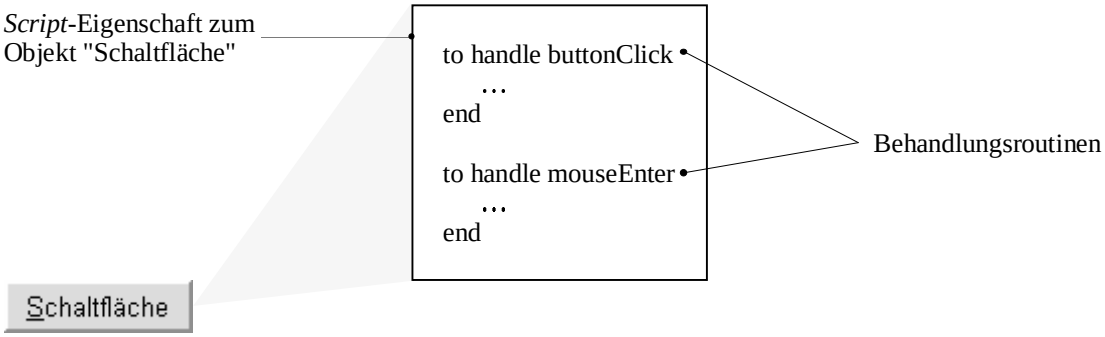


Abb. 2.3: Das Skript eines ToolBook-Objekts

Die meisten ToolBook-Objekte haben jedoch kein eingebautes Verhalten. Um das Objektverhalten zu definieren oder zu ändern, schreibt man für das jeweilige Objekt ein OPENSCRIPT-Programm, ein sogenanntes **Skript**. Jedes Skript ist direkt und untrennbar mit einem Objekt verbunden (vgl. Abb. 2.3).

2.3.1 Ereignisse und Botschaften

ToolBook ist ebenso wie das Windows-System *ereignisgesteuert*, d.h. die Benutzeraktionen steuern das Programm. Jede Handlung, die ein/eine BenutzerIn durchführt, etwa das Klikken auf eine Schaltfläche, ist ein **Ereignis**. ToolBook übersetzt solche Ereignisse in **Botschaften**, eine Kommunikation, die ToolBook an ein Objekt (*Zielobjekt*, oder kurz *Ziel*) sendet, um diesem mitzuteilen, daß ein Ereignis stattgefunden hat. Es gibt zwei Arten von Botschaften:

- **integrierte Botschaften:** Alle im ToolBook-System eingebauten Botschaften, die ToolBook als Reaktion auf eine Benutzeraktion sendet.
- **benutzerdefinierte Botschaften:** Alle nicht in ToolBook integrierten Botschaften, für die der/die EntwicklerIn eine entsprechende Behandlungsroutine zur Verfügung stellen muß.

Ein OPENSCRIPT-Programm, welches das Verhalten eines Objekts für ein bestimmtes Ereignis festlegt, heißt **Behandlungsroutine**. Dabei handelt es sich um eine Reihe von OPENSCRIPT-Anweisungen in einem Skript, die auf eine bestimmte Botschaft reagieren. Es ist damit auch möglich, daß eine Behandlungsroutine durch eine andere aufgerufen wird, die die Botschaft direkt versendet. Ein Skript kann mehrere Behandlungsroutinen enthalten und jede Behandlungsroutine im Skript kann auf ein anderes Ereignis reagieren.

2.3.2 Objekthierarchie

Das ToolBook-System sendet Botschaften in einer bestimmten Reihenfolge von Objekt zu Objekt, die als **Objekthierarchie** bezeichnet wird. Löst ein Ereignis eine Botschaft aus, so sendet das ToolBook-System diese Botschaft an das Zielobjekt, z.B. ein Graphikobjekt. Hat

dieses Objekt keine Behandlungsroutine für diese Botschaft definiert, dann wird die Botschaft an ein anderes Objekt gesendet, das sich innerhalb der Objekthierarchie über dem ursprünglichen Zielobjekt befindet (*übergeordnetes Objekt* genannt). Ist das Graphikobjekt Teil einer Objektgruppe, so wird das Skript dieser Gruppe auf eine passende Behandlungsroutine hin untersucht. Falls die Botschaft dort ebenfalls nicht von einer Behandlungsroutine aufgefangen wird, so leitet sie das ToolBook-System der Reihe nach an die Seite, den Hintergrund, das Buch, an eventuelle Systembücher und schließlich an das ToolBook-System weiter (siehe Abb. 2.4). Dort ankommende Botschaften werden entweder ignoriert oder, falls es sich dabei um eine integrierte Botschaft handelt, durch vordefinierte Aktionen behandelt.

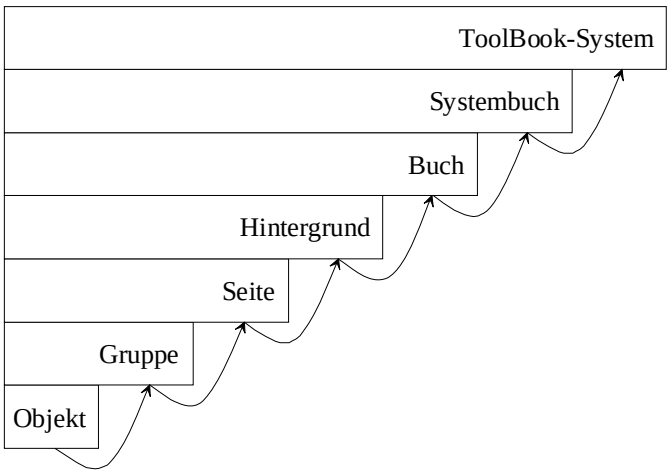


Abb. 2.4: Die Objekthierarchie

Aus Behandlungsroutinen können Botschaften an jedes beliebige Objekt verschickt werden, auch an untergeordnete Objekte oder an Objekte in anderen Büchern. Der/die ProgrammiererIn muß in diesen Fällen darauf achten, daß durch diese Verfahren keine Endlosschleife zur Laufzeit entsteht. Weiterhin läßt sich durch Ausnutzung der Objekthierarchie der Programmieraufwand stark vermindern, indem man Behandlungsroutinen weiter oben in der Objekthierarchie festlegt.

Soll ein bestimmtes Objektverhalten nicht nur lokal in einem einzelnen Buch, sondern übergreifend mehreren Anwendungen (Büchern) zur Verfügung stehen, dann können Behandlungsroutinen in einem *Systembuch* plaziert werden. Es ist damit also möglich Bibliotheken von Behandlungsroutinen aufzubauen.

2.3.3 Erstellung von Behandlungroutinen

ToolBook bietet für die Erstellung von Behandlungsroutinen eine spezielle Programmierumgebung, das *Skript-Editor-Fenster*, an (siehe Abb. 2.5). Es enthält alle Bearbeitungsfunktionen von Texteditoren, einschließlich Syntaxanalyse des Skripts und folgender Werkzeuge:

- *Skript-Rekorder*: Eine ToolBook-Programmierfunktion, die das Ergebnis von Tastenanschlägen und Mausaktionen in Form von OPENSCRIPT-Anweisungen aufzeichnet, um diese Anweisungsfolge in ein Skript einfügen zu können (nützlich beim Erstellen kurzer Animationen).
- *Auto-Skript-Bibliothek*: Eine Sammlung von vorprogrammierten OPENSCRIPT-Behandlungsroutinen und Quelltextblöcken für die am meisten verwendeten Pro-

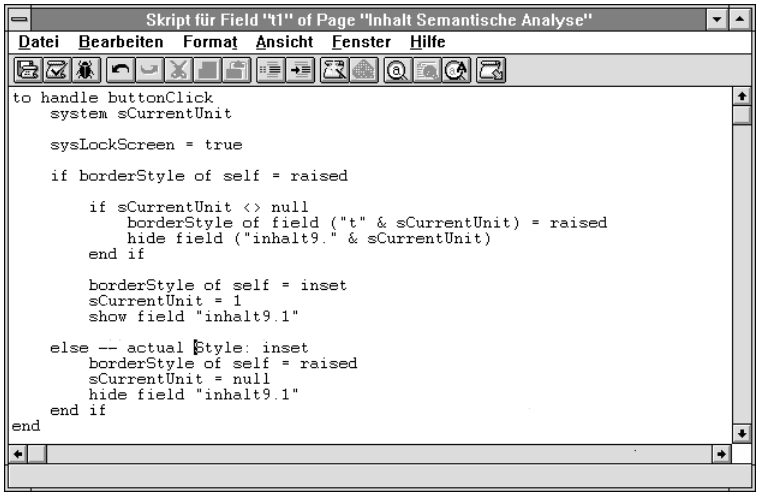


Abb. 2.5: Das *Skript-Editor-Fenster* von MTB 3.0

grammieraufgaben. Die Auto-Skript-Bibliothek kann auch nach Bedarf durch benutzerdefinierte Auto-Skripts erweitert werden.

2.4 Kommunikation mit anderen Programmen

Die Funktionalität von ToolBook läßt sich erweitern, indem man Daten aus anderen Windows-Applikationen integriert. So können beispielsweise Kalkulationstabellen oder Bitmaps in die ToolBook-Anwendung eingebunden, Daten mit anderen Anwendungen ausgetauscht oder Windows-Funktionen aus Bibliotheken aufgerufen werden.

2.4.1 DDE

Dynamischer Datenaustausch (**DDE**) ist ein Windows-Kommunikationsprotokoll, das in Abschnitt 1.5.2 bereits ausführlich erläutert wurde. Zwei Programme, die DDE unterstützen, können Daten miteinander austauschen und aneinander Befehle senden. Um DDE einsetzen zu können, benötigt man ein Exemplar des Programms, mit dem Daten ausgetauscht werden sollen, und Informationen über die DDE-Syntax, die dieses Programm erwartet.

In ToolBook ist DDE-Funktionalität über eine Gruppe von OPENSCRIPT-Botschaften realisiert, auf die man mit Behandlungsroutinen reagieren kann. Innerhalb ein und derselben DDE-Kommunikation kann ToolBook (und ebenso das andere beteiligte Programm) sowohl Client als auch Server sein.

Einige Anwendungen können permanente Datenverknüpfungen („warme“ bzw. „heiße“ Verbindungen) mit den Botschaften WM_DDE_ADVISE und WM_DDE_UNADVISE herstellen. Diese DDE-Botschaften bleiben in ToolBook ohne Wirkung, da ToolBook keine permanenten Datenverknüpfungen einrichten kann. Es ist also lediglich möglich, eine sogenannte *statische* (oder „kalte“) *Verbindung* mit ToolBook herzustellen.

2.4.2 DLL

Durch Aufrufen von **DLL**’s („Dynamic Link Libraries“, siehe Abschnitt 1.5.3) kann die Funktionalität von OPENSCRIPT erweitert werden. Das Windows-System enthält DLL’s, welche die direkte Interaktion mit dem System ermöglichen, etwa USER, KERNEL oder GDI. Zu ToolBook gehören dBase III- und Paradox-DLL’s, mit denen Datenbank-

funktionalität in ToolBook integriert werden kann. ToolBook enthält außerdem DLL's zum Ändern von DOS-Dateien und zur Bereitstellung von Windows-Systemressourcen. Das Einbinden einer DLL in ToolBook läuft in zwei Schritten ab:

1. Verknüpfen mit der DLL, damit sie für die ToolBook-Anwendung zur Verfügung steht.
2. Aufrufen von Funktionen der DLL aus einem Skript heraus.

Die Aufhebung der Verknüpfung erfolgt entweder manuell durch den Aufruf entsprechender OPENSCRIPT-Befehle, oder automatisch beim Beenden der ToolBook-Anwendung durch das ToolBook-System. Es können beliebig viele DLL's eingebunden werden.

Aus einer DLL können auch Botschaften direkt an ToolBook-Objekte in einer Anwendung verschickt werden. Ebenso ist der Aufruf ToolBook-interner Funktionen oder die Abfrage von ToolBook-Variablen aus einer DLL möglich.

2.4.3 OLE

Unter **OLE** („Object Linking and Embedding") versteht man ein Windows-Protokoll zur gemeinsamen Nutzung von Objekten zwischen Anwendungen (siehe Abschnitt 1.5.4). Damit ist es möglich Objekte, die in anderen Programmen (Server-Applikationen) erstellt wurden, in eine ToolBook-Anwendung einzubinden, z.B. eine mit *MS Paintbrush* erstellte Graphik. Die Verknüpfung mit dem Server bleibt dabei aufrechterhalten. Klickt der/die BenutzerIn innerhalb der ToolBook-Anwendung auf ein OLE-Objekt, dann startet ToolBook den Server, damit das OLE-Objekt bearbeitet (auf der Autorenebene) oder angezeigt bzw. abgespielt (auf der Leserebene) werden kann.

ToolBook unterstützt nur OLE, nicht OLE 2.0, und ist in Zusammenhang mit OLE nur als Client-Anwendung zu betreiben.

2.4.4 Importierung von Window-Ressourcen

Um die optischen Gestaltungsmöglichkeiten von ToolBook zu vergrößern, lassen sich Window-Ressourcen wie Cursors, Bitmaps, Symbole und Farbpaletten in ToolBook erstellen oder aus anderen Windows-Programmen importieren. ToolBook bietet selbst auch Programmierwerkzeuge an, mit deren Hilfe sich Menüleistenressourcen erstellen lassen, die in mehreren Büchern gemeinsam nutzbar sind.

2.4.5 Übersetzen von Windows-Botschaften

ToolBook-Anwendungen sind durch das ToolBook-System hinsichtlich der Botschaftsbearbeitung nicht vollständig und endgültig vom Windows-System getrennt. Es ist möglich, Windows-Botschaften (z.B.: WM_PAINT für das Neuzeichnen eines Fensters) abzufangen und geeignet zu verarbeiten. Dieses Verfahren wird als die **Übersetzung von Windows-Botschaften** bezeichnet. Sie kann in solchen Fällen erforderlich sein, wenn für eine Windows-Botschaft kein entsprechendes ToolBook-Gegenstück existiert oder ein anderes, von ToolBook unabhängiges Programm eine benutzerdefinierte Botschaft an die ToolBook-Anwendung senden möchte.

Im Normalfall erfolgt das Zusammenspiel der Ereignisverarbeitung zwischen Windows und ToolBook folgendermaßen (hier am Beispiel eines Mausklicks der linken Maustaste auf eine Schaltfläche einer ToolBook-Seite): Windows sendet die Botschaft WM_LBUTTONDOWN an das betroffene Fenster. Intern übersetzt ToolBook diese Windows-Botschaft in

eine ToolBook-Botschaft, hier die Botschaft *buttonDown*, und sendet diese an die betroffene Schaltfläche. Wenn hier keine Behandlungsroutine für diese Botschaft definiert wurde, steigt die Botschaft der Objekthierarchie folgend nach oben bis eine Behandlungsroutine gefunden oder das Toolbook-System als das übergeordnetste Objekt erreicht wird.

Bei der hier betrachteten Übersetzung von Windows-Botschaften wird diese Botschaftskette ganz zu Anfang aufgetrennt. Dazu muß der/die ProgrammiererIn eine spezielle ToolBook-Funktion (*translateWindowMessage*) aufrufen, die diese Trennung vornimmt. Eine empfangene Windows-Botschaft wird dabei in eine frei wählbare Botschaft übersetzt und an ein beliebiges Objekt versendet, dessen Skript eine für diesen Zweck vorgesehene Behandlungsroutine enthält. Dadurch entfällt die Standardreaktion von ToolBook auf das Ereignis, außer man leitet die Botschaft entweder direkt oder durch Ausnutzung der Objekthierarchie an das ToolBook-System weiter.

Eine eingeleitete Übersetzung von Windows-Botschaften bleibt solange aktiv, bis sie manuell durch Aufruf der Funktion *untranslateWindowMessage* terminiert wird. Die Deaktivierung der Übersetzung erfolgt automatisch durch das Schließen des betroffenen Fensters, für das die Botschaft übersetzt wird, spätestens jedoch durch das Beenden von ToolBook.

2.6 Vorteile und Schwächen des Systems

Den NutzerInnen des Autorensystems Asymetrix Multimedia ToolBook 3.0 bieten sich u.a. folgende **Vorteile**:

1. Einfache Bedienung und Erstellung von ToolBook-Anwendungen, auch für AutorInnen, die weniger Programmierkenntnisse und -erfahrung besitzen.
2. Es lassen sich sehr schnell aufwendige und anspruchsvolle Benutzeroberflächen kreieren.
3. Die Eingabe- und Interaktionmöglichkeiten durch den/die AnwenderIn der entwickelten ToolBook-Anwendung sind vielseitig und mit wenig Aufwand bereitzustellen.
4. Leichte Einbindung von Multimedia-Elementen.
5. Vielfältige Kommunikationsmöglichkeiten mit anderen Windows-Applikationen.

Andererseits hat das System einige **Nachteile** und konzeptuelle Schwächen, die jedoch zum Teil auf das Windows-System als Grundlage zurückzuführen sind:

1. Die Geschwindigkeit in graphikintensiven Anwendungen ist nicht optimal. Komplexe Bewegungsabläufe in Animationen sind deshalb kaum zu realisieren.
2. Es stehen kaum Werkzeuge zur komfortablen Objektanimation zur Verfügung. Um eine kleine Animation zu schreiben, ist sehr viel Aufwand nötig.
3. Die Seitengröße, sowie der Seitenspeicher ist begrenzt.
4. Da die Skripte bei den Objekten stehen, zu denen sie gehören, ist es schwierig, den Überblick über alle Skripte zu behalten.

Multimedia ToolBook 3.0 eignet sich Aufgrund der dargestellten Vor- und Nachteile besonders für „statische“ Anwendungsbereiche, in denen es nicht erforderlich ist, komplexe Berechnungsvorgänge vorzunehmen und dynamisch Objekte zu erzeugen. Die sehr vielfältigen Möglichkeiten, Verbindungen zu anderen Programmen herzustellen, Datenbanken anzusprechen und Multimedia-Elemente einzufügen, lassen darauf schließen, daß die EntwicklerInnen von Asymetrix diesem Anspruch eine geringere Gewichtung beigemessen haben.

Das Hauptaugenmerk lag hier in der leichten Bedienbarkeit und schnellen Erstellung allgemeinerer, weniger spezieller Software, was auch den gängigen Anforderungen an Autorensysteme entspricht.

Kapitel 3

Semantische Analyse

Übersetzer von Programmiersprachen sind in verschiedene Module zerlegt, die jeweils eine bestimmte Teilaufgabe übernehmen. Alle Teilaufgaben lassen sich in eine **Analysephase** und in eine **Synthesephase** gruppieren. In der Analysephase, die im Idealfall von der Zielsprache und der Zielmaschine unabhängig ist, wird die syntaktische Struktur und die statische Semantik einer Sprache berechnet. Die **statische Semantik** ist der Teil der semantischen Eigenschaften eines Programms, die zur Übersetzungszeit berechnet werden können, also ohne die Eingabedaten zu betrachten. Die Synthesephase erhält die Ausgabe der Analysephase als Eingabe und erzeugt daraus das Zielprogramm (siehe Abb. 3.1).

Teilweise sind die Aufgaben eines Übersetzers Erkennungsprobleme für bestimmte Typen von Grammatiken. Diese Erkennungsprobleme lassen sich durch Automaten entsprechenden Typs lösen, die automatisch erzeugbar sind. Basierend auf der Theorie der formalen Sprachen und der Automaten lassen sich auf diese Weise Übersetzerteilaufgaben formal beschreiben. Die nachfolgende Tabelle gibt eine kurze Übersicht der Teilaufgaben, deren (teilweise) Beschreibungsmöglichkeit durch formale Spezifikationen und der generierten Automaten:

Übersetzerteilaufgabe	**Spezifikationsmechanismus**	**Automatentyp**
lexikalische Analyse	reguläre Ausdrücke	deterministischer endlicher Automat
syntaktische Analyse	kontextfreie Grammatiken	deterministischer Kellerautomat
semantische Analyse	Attributgrammatiken	
effizienzsteigernde Transformationen	Baum → Baum-Transformationen	endliche Baumtransduktoren
Codeselektion in der Codeerzeugung	reguläre Baumgrammatiken	endliche Baumautomaten

Die **semantischen Analyse** berechnet Eigenschaften von Programmen, die über die reinen syntaktischen Eigenschaften hinausgehen, aber durch Prädikate auf Kontextinformationen (Kontextbedingungen) beschreibbar sind. Diese Eigenschaften sollen jedoch bereits zur Übersetzungszeit berechenbar sein, und man bezeichnet sie daher als **statische semantische Eigenschaften**. Hierunter fallen z.B. Typkorrektheit, Deklariertheit, Ausschluß von Dop-

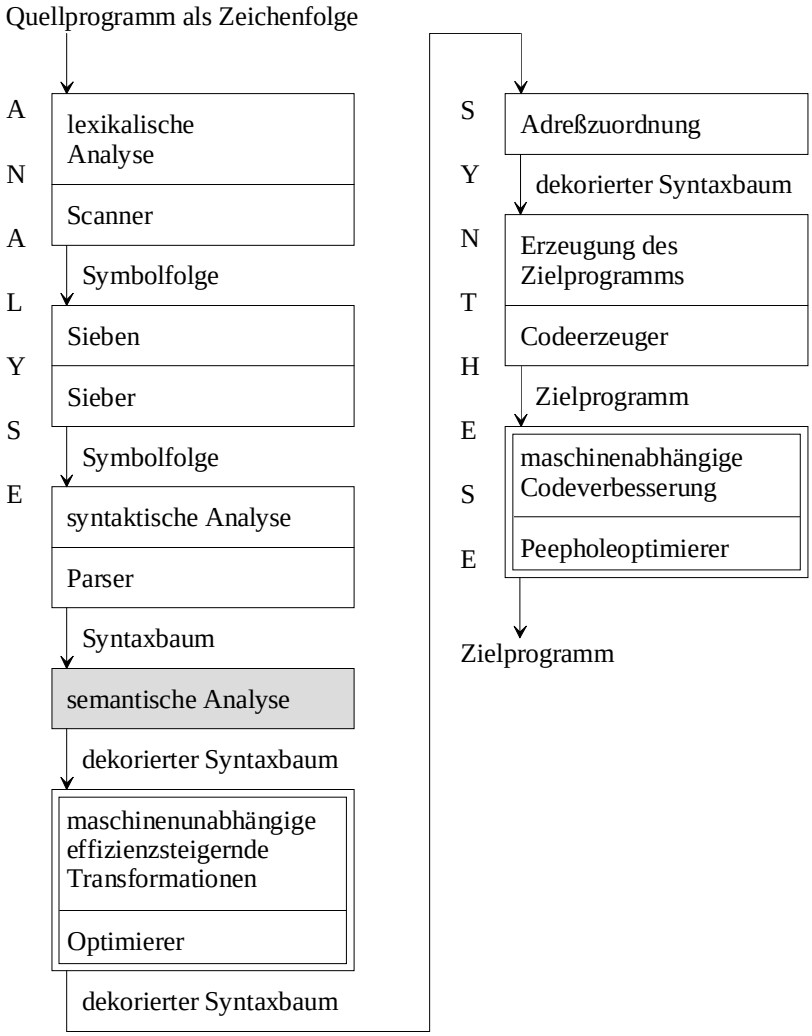


Abb. 3.1: Konzeptionelle Übersetzerstruktur mit Angabe der Programmzwischendarstellungen. Die Moduln in den doppelt umrandeten Kästchen sind optional. Programmtechnisch werden Scanner und Sieber zu einem einzigen Modul zusammengefaßt (aus [WM92]).

peldeklarationen oder konsistente Typzuordnung zu allen Funktionen eines Programms mit Polymorphismus.

Die Übersetzung von Programmiersprachen, einschließlich der semantischen Analyse, ist in [WM92] beschrieben. Darin ist auch die formale Spezifikation der semantischen Analyse durch Attributgrammatiken beschrieben, auf die in dieser Arbeit nicht näher eingegangen wird.

3.1 Voraussetzungen

Um die semantischen Analyse richtig zu verstehen, ist es wichtig, daß die Voraussetzungen und die Begriffe klar definiert werden:

3.1.1 Statische semantische Eigenschaften

Man bezeichnet eine (nicht kontextfreie) Eigenschaft eines Konstrukts einer Programmiersprache als eine **statische semantische Eigenschaft**, wenn

1. für jedes Vorkommen dieses Konstrukts in einem Programm der „Wert“ dieser Eigenschaft für alle (dynamischen) Ausführungen des Konstrukts konstant ist und wenn
2. für jedes Vorkommen des Konstrukts in einem korrekten Programm diese Eigenschaft berechnet werden kann.

Statische semantische Eigenschaften beschreiben allen dynamischen Ausführungen gemeinsame Eigenschaften, die zur Übersetzungszeit mithilfe des Programmtextes berechnet werden können. Im Gegensatz dazu sind **dynamische semantische Eigenschaften** erst zur Laufzeit des übersetzten Programms feststellbar. Die zweite Bedingung dient zur Trennung der statisch semantischen Eigenschaften von solchen Eigenschaften, die durch abstrakte Interpretation (Datenflußanalyse, siehe [WM92]) berechnet werden können. Abstrakte Interpretation versucht lediglich, möglichst gute statische Annäherungen an dynamische Eigenschaften von Programmkonstrukten zu berechnen. Hier ist die „leere“ Information i.a. eine mögliche Information.

3.1.2 Terminologie

Wir verwenden zur Beschreibung der semantischen Analyse folgende Begriffe:

- **Bezeichner** („identifier“) werden zur Benennung von Objekten einer Programmiersprache (z.B. Variablen) verwendet. Es handelt sich um Symbole im Sinne der lexikalischen Analyse, also eine Folge von lexikalischen Einheiten, die vom Scanner-Modul gelesen werden.
- Die **Deklaration** eines Bezeichners ist ein Konstrukt, das den Bezeichner als Name für ein Objekt in das Programm einführt.
- Ein **definierendes Vorkommen** eines Bezeichners ist sein Vorkommen in seiner Deklaration. Es gibt jedoch Ausnahmen, in denen nicht jedes Vorkommen in einer Deklaration auch ein definierendes Vorkommen ist, beispielsweise ein Vorkommen innerhalb einer rekursiven Deklaration.
- Das Komplement der definierenden Vorkommen bezeichnet man als die **angewandten Vorkommen** eines Bezeichners.
- **Blöcke** sind Vorkommen von Scope-Konstrukten, die die Gültigkeit eines Bezeichners im Programmtext begrenzen. Unter Scope-Konstrukten versteht man das Vorkommen bestimmter Elemente einer Programmiersprache, etwa Prozedurdeklarationen, Blockkonstrukte oder Moduln.
- Der **Typ** eines Objekts bestimmt seine Verwendungsmöglichkeit zur Laufzeit des Programms.

3.1.3 Konkrete vs. abstrakte Syntax

Die Eingabe der semantischen Analyse ist ein durch die Syntaxanalyse erstellter Baum, der die konkrete oder die abstrakte Syntax repräsentieren kann. Die **konkrete Syntax** wird durch die der Sprache zugrundeliegende kontextfreie Grammatik dargestellt. Sie bewirkt, daß der entsprechende Syntaxbaum zum Beispiel alle Schlüsselwörter und viele Nichtterminale enthält, die für die weitere Analyse überflüssig sind.

Aus diesem Grund benutzen Übersetzer zu Darstellung der syntaktischen Struktur des Programmtextes die **abstrakte Syntax**. In ihr werden alle unnötigen Teile eliminiert, die

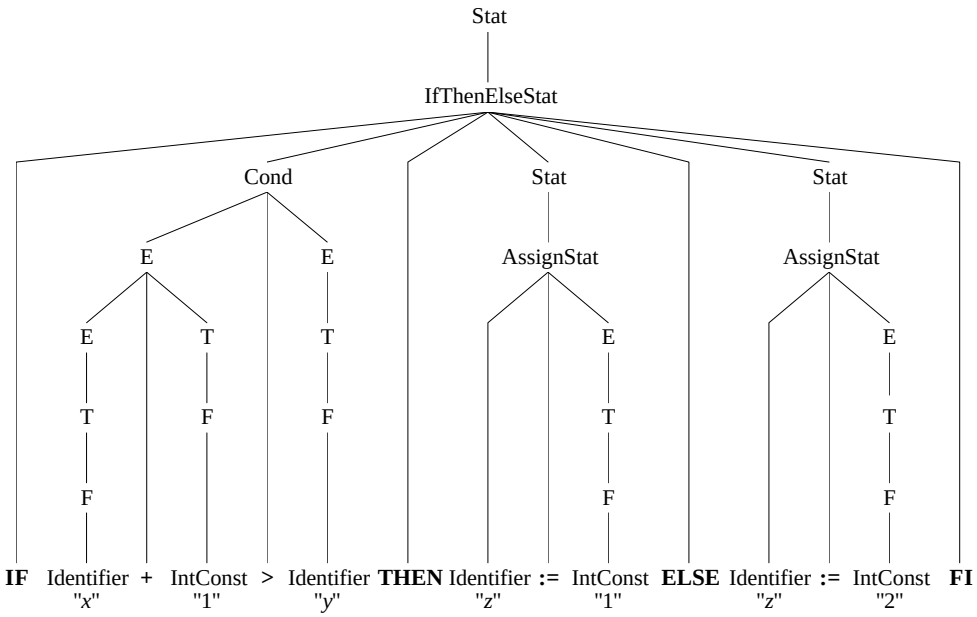


Abb. 3.2: Baum zur konkreten Syntax

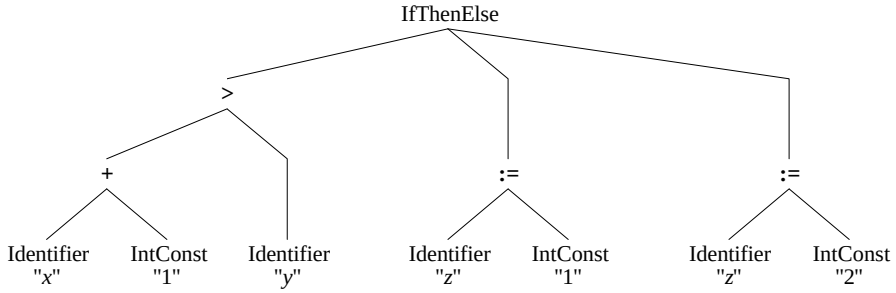


Abb. 3.3: Baum zur abstrakten Syntax

zwar zum Erkennen der Syntax wichtig waren, von nun an aber vom Übersetzer nicht mehr benutzt werden. Betrachten wir als Beispiel das folgende Programmfragment:

if $x + 1 > y$
then $z := 1$
else $z := 2$
fi

Die Abbildungen 3.2 und 3.3 stellen die zu diesem Programmstück gehörigen Bäume zur konkreten wie abstrakten Syntax dar (bei entsprechender kontextfreier Grammatik).

3.2 Gültigkeits- und Sichtbarkeitsregeln

Die Gültigkeits- und Sichtbarkeitsregeln ermöglichen es festzustellen, auf welches definierende Vorkommen sich ein angewandtes Vorkommen eines Bezeichners bezieht. Diese Aufgabe nennt man die Identifizierung von Bezeichnern („identification of identifiers“). In Programmiersprachen, die das Überladen von Bezeichnern erlauben, kann es zu einem angewandten Vorkommen mehrere definierende Vorkommen geben.

3.2.1 Gültigkeit

Der **Gültigkeitsbereich** („scope“, „range of validity“) eines definierenden Vorkommens eines Bezeichners *x* ist der Teil des Programms, in dem sich ein angewandtes Vorkommen von *x* auf dieses definierende Vorkommen beziehen kann. Wichtig dabei ist die Schachtelung von Scope-Konstrukten zu erklären. COBOL gestattet keine Schachtelung, alle Bezeichner sind im ganzen Programm gültig und sichtbar. FORTAN erlaubt nur Schachtelungstiefe 1, und Blöcke in Programmiersprachen wie ADA, ALGOL60, ALGOL68, PASCAL und funktionale Sprachen dürfen rekursiv, unendlich tief geschachtelt werden.

Da bei einigen Programmiersprachen (ALGOL60, ALGOL68) jeder im Block deklarierte Bezeichner im ganzen Block gültig ist, benötigt der Übersetzer zur Deklarationsübersetzung eventuell Informationen über einen Bezeichner, dessen Deklaration noch gar nicht abgearbeitet wurde. Folglich ist die 1-Pass-Übersetzbarkeit eines Programms so nicht möglich. Abhilfe schaffen **Regeln**, die dieses Problem vermeiden, hier zwei Beispiele:

- In ADA beginnt der Gültigkeitsbereich eines Bezeichners mit dem Ende der Deklaration und hört mit dem Ende des Blocks auf.
- In PASCAL ist der Gültigkeitsbereich der ganze Block. Es darf jedoch kein angewandtes Vorkommen vor dem Ende der Deklaration stehen.

3.2.2 Sichtbarkeit

Der **Sichtbarkeitsbereich** ist der Teil des Gültigkeitsbereichs, in dem ein definierendes Vorkommen eines Bezeichners *x* nicht durch ein anderes überdeckt ist.

Ein definierendes Vorkommen eines Bezeichners *x* kann vom Standpunkt eines angewandten Vorkommen entweder direkt sichtbar oder eventuell durch eine andere Deklaration desselben Bezeichners *x* verdeckt sein. Ist beispielsweise das definierende Vorkommen nicht im Deklarationsteil des aktuellen Blocks, so kann eine lokale Deklaration diese globale Bezeichnerdeklaration verdecken (siehe Abb. 3.4).

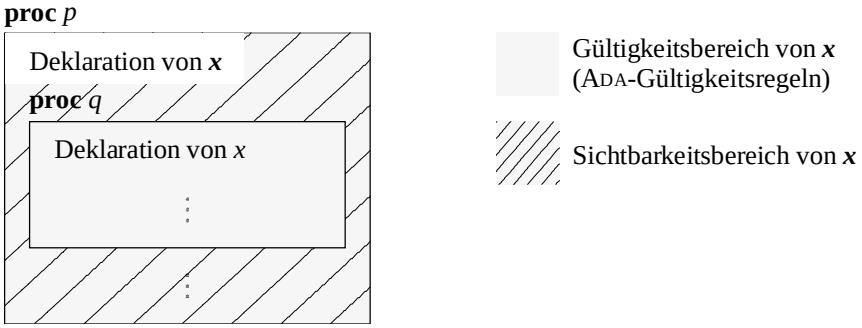


Abb. 3.4: Beispiel zum Gültigkeits- und Sichtbarkeitsbereich

Analog zum Problem des Gültigkeitsbereichs eines definierenden Vorkommens wird auch dieses Problem durch Regeln gelöst. Mit Hilfe dieser **Sichtbarkeitsregeln** lassen sich auch verdeckte definierende Vorkommen eines Bezeichners sichtbar machen, beispielsweise:

- Die **Erweiterung eines Bezeichners** um den Bezeichner eines die Deklaration enthaltenen Konstrukts. In PASCAL werden sogenannte qualifizierte Bezeichner dadurch erzeugt, indem man den Bezeichner um den Namen des Konstrukts erweitert, welche seine Deklaration enthält. Somit wird ein Bezug auf ein bisher verdecktes Vorkommen ermöglicht.

- Sichtbarmachen einer bestimmten **Region** des Gültigkeitsbereichs, ohne die Erweiterung eines Bezeichners. Die *use*-Anweisung in ADA listet Bezeichner von umfassenden Programmeinheiten auf, die dadurch innerhalb der Region sichtbar werden. Die Region erstreckt sich vom Ende der *use*-Anweisung bis zum Ende der umgebenden Programmeinheit.
- **Direktiven**, welche die Definitionen in getrennt übersetzten Einheiten sichtbar machen. In ADA werden durch die *with*-Klausel Bezeichner, die innerhalb eines getrennt übersetzten Pakets (einem Modul) deklariert wurden, sichtbar. Deren Gültigkeitsbereich umfaßt alle Programme, die nach der getrennten Übersetzung zusammengebunden werden.

3.3 Überprüfung der Kontextbedingungen

Wir betrachten die Überprüfung der Kontextbedingungen (Deklariertheitseigenschaften, Typkonsistenz) an einem vereinfachten Fall, einer Programmiersprache mit geschachtelten Scopes, ohne Moduln und ohne Überladung. Eingebaute (arithmetische) Operatoren dürfen jedoch überladen sein; man spricht hier von *trivialer Überladung*. Die Überprüfung der Deklariertheitseigenschaften und der Typkonsistenz erfolgt getrennt, und wird somit auch von zwei korrespondierenden Moduln, dem *Deklarations-Analysator* („scope analyzer“) und dem *Typ-Analysator* („type analyzer“), vorgenommen. Die folgende Darstellung der Kontextbedingungen ist in [Wat84] und [WM92] vertieft.

3.3.1 Identifizierung von Bezeichnern

Unter der **Identifizierung von Bezeichnern** (siehe Anhang B.1) versteht man die Aufgabe, jedem angewandten Vorkommen eines Bezeichners das gemäß der Gültigkeits- und Sichtbarkeitsregeln zugehörige definierende Vorkommen zuzuordnen. In unserem vereinfachten Fall gilt: Zu jedem angewandten Vorkommen eines Bezeichners gehört in einem korrekten Programm genau ein definierendes Vorkommen, da keine Überladung zugelassen ist! Das Resultat dieser Identifizierung wird von der Typüberprüfung und der Codeerzeugung verwendet. Aus diesem Grund muß das Ergebnis diese Phase überleben.

Um die Identifizierung von Bezeichnern durchzuführen, erstellt der Übersetzer eine sogenannte **Symboltabelle**, in der für jedes definierende Vorkommen eines Bezeichners die zugehörige deklarative Information abgespeichert ist. Die Symboltabelle ist analog zur Blockstruktur des Programms organisiert und sollte idealerweise nach Ausführung des Deklarations-Analysators gelöscht werden können. Sie dient lediglich dazu, die Identifizierung vorzunehmen. Es existieren mehrere Möglichkeiten, auf welche Art man das Ergebnis der Identifizierung darstellt. Man speichert für jeden Knoten, der ein angewandtes Vorkommen eines Bezeichners symbolisiert, entweder

1. einen Verweis auf den Knoten der entsprechenden Deklaration oder
2. die Adresse des Eintrags für das definierende Vorkommen in der Symboltabelle oder
3. die deklarative Information zu dem definierenden Vorkommen

ab. Alternative (2) widerspricht der Forderung, die Symboltabelle nach der Identifizierungsphase zu löschen. (1) hat vor (3) den Vorteil, daß es sich um einen Verweis (Zeiger) auf das definierende Vorkommen handelt und damit alle angewandten Vorkommen zu einem definierenden Vorkommen dieses gemeinsam verwenden. Aus diesem Grund wählen wir Alternative (1). Der abstrakte Syntaxbaum bleibt (erweitert um die nicht kontextfreie Informati-

on, nämlich die Zeiger auf die definierenden Vorkommen) die einzige Datenstruktur. Man spricht in diesem Fall von einem **dekorierten** abstrakten Syntaxbaum.

Die Implementierung der Symboltabelle (Deklarations-Analysator) muß verschiedene Operationen anbieten, die die Symboltabelle verwalten. Trifft der Analysator auf eine Deklaration, so muß er den deklarierten Bezeichner und einen Verweis auf den entsprechenden Deklarationsknoten im abstrakten Syntaxbaum in die Tabelle eintragen. Ebenso muß das Öffnen bzw. das Schließen von Blöcken vermerkt werden. Trifft der Analysator auf ein angewandtes Vorkommen eines Bezeichners, dann sucht er in der Symboltabelle nach dem entsprechenden Eintrag des definierenden Vorkommens. Findet er ihn nicht, so gibt er eine Fehlermeldung aus, daß der betreffende Bezeichner nicht deklariert wurde. Findet er ihn, so trägt er die Deklarationsstelle im Knoten des abstrakten Syntaxbaums für dieses angewandte Vorkommen ein.

3.3.2 Überprüfung der Typkonsistenz

Hierbei wird getestet, ob die Operandentypen für jeden Operator innerhalb eines Ausdrucksbaumes zu ihm passen. Dieser Test kann in einem *bottom up*-Pass über den Ausdrucksbaum erfolgen, und der Modul, der dies leistet, wird als **Typ-Analysator** bezeichnet.

Um die **Überprüfung der Typkonsistenz** (vgl. Anhang B.2) vorzunehmen, berechnet der Übersetzer die Typen für alle getypten Objekte.

- Für terminale Operanden, die Konstanten sind, steht der Typ schon fest.
- Für Bezeichner nutzt der Übersetzer den Verweis auf seine Deklarationsstelle im dekorierten abstrakten Syntaxbaum, den wir bei der Identifizierung der Bezeichner berechnet haben.
- Für Operatoren werden zunächst einmal die einzelnen Operandentypen festgestellt.

Wir bezeichnen die Überladung in die Sprache eingebauter Operatoren (z.B. arithmetischer Operatoren) als *triviale Überladung*. Ist der Operator nicht (trivial) überladen, dann werden die einzelnen berechneten Operandentypen mit den geforderten Parametertypen des Operators, die in einer Tabelle festgelegt sind, verglichen. Sind sie identisch, so wird der Ergebnistyp zurückgegeben; wenn nicht, eine Fehlermeldung. Ist der Operator (trivial) überladen, so wird zusätzlich noch die richtige Operation ausgewählt und in den abstrakten Syntaxbaum eingetragen.

Erlaubt die Programmiersprache Typanpassungen, so wird für jeden Operator und jede Kombination aus Operandentypen, die nicht zu ihm passen, geprüft, ob die Operandentypen durch Typanpassung zu einer für den Operator gültigen Operandentypenkombination gemacht werden können.

3.4 Überladung von Bezeichnern

Ein Symbol heißt **überladen**, wenn es an einer Stelle im Programm mehrere Bedeutungen haben kann. In diesem Abschnitt wird im Gegensatz zur trivialen Überladung nur die Überladung von benutzerdefinierten Bezeichnern (z.B. Prozedur- oder Funktionsnamen) betrachtet. Erlaubt eine Programmiersprache die Überladung von benutzerdefinierten Symbolen (ADA gestattet beispielsweise die Überladung von Funktionsbezeichnern), dann ist es auch in einem korrekten Programm möglich, daß es zu einem angewandten Vorkommen *x* mehrere sichtbare definierende Vorkommen von *x* gibt. Man bezeichnet die Überladung von Symbolen im obigen Sinne auch als **ad-hoc-Polymorphismus**.

Ein Programm ist nur dann korrekt, wenn aufgrund der *Typumgebung* des angewandten Vorkommens genau eines der definierenden Vorkommen ausgewählt werden kann. Bei Funktions- oder Prozeduraufrufen ist die Typumgebung die Kombination der aktuellen Parameter. Das Auswählen eines eindeutigen definierenden Vorkommens wird als **Auflösung der Überladung** („overload resolution“) bezeichnet und findet nach der Identifizierungsphase statt.

3.4.1 Verfahren zur Auflösung der Überladung

Der **Auflösungsalgorithmus** (Anhang B.3) betrachtet nur bestimmte Sprachkonstrukte, wie Ausdrücke, Bezeichner, usw. Er benutzt dabei zwei Läufe über je einen Ausdrucksbaum, erst *bottom up* und dann t*op down*. Mit jedem Knoten eines Ausdrucksbaumes wird zunächst eine Menge von allen möglichen (sichtbaren) Definitionen des betreffenden Operators assoziiert, die wir nach der Identifizierung der Bezeichner zur Verfügung stehen haben. Der *bottom up*-Pass analysiert die Typen aller Parameter eines Operators. Im Ausdrucksbaum handelt es sich dabei gerade um seine Kinder. Sind alle Parametertypen des Operators berechnet, dann werden alle die Operatoren aus der o.g. Menge entfernt, deren *i*-ter Parametertyp zu keinem der, soeben ermittelten, möglichen Resultatstypen des *i*-ten Operanden paßt. Dieses Streichen aus der Menge wird *bottom up*-Elimination genannt. Ein Lauf über den Ausdrucksbaum reicht aber noch nicht aus, um die Überladung aufzulösen.

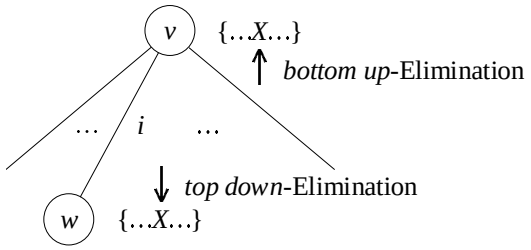


Abb. 3.5: Eliminationen, aus [WM92]

Betrachten wir einen Operatorknoten *v*, dessen *i*-ter Parameter der Operatorknoten *w* sei (vgl. Abb. 3.5). Knoten *w* hat im ersten Lauf seine Resultatstypen an seinen Elterknoten *v* weitergegeben. Aus der Menge aller für *v* sichtbaren Operatordefinitionen (Abk.: *ops*(*v*)) wurden durch die *bottom up*-Elimination all die Operatoren eliminiert, deren *i*-ter Parametertyp zu keinem Resultatstyp von *w* passen. Die Operatormenge des Knotens *w* ist durch diese Operation nicht verändert worden. Nun ist es möglich, daß in dieser Menge Operatoren existieren, deren Resultatstyp nicht mit irgendeinem Typ der entsprechenden *i*-ten Parametertypen der Operatoren in der Menge des Knotens *v* korrespondiert. Diese Operatoren innerhalb der Menge von *w* können gelöscht werden. Es ist also ein Informationsfluß von „oben nach unten“ nötig, der erst alle i-ten Parametertypen aus der Menge des Knotens *v* aufsammelt und an den *i*-ten Kinderknoten *w* übergibt. In dessen Operatormenge können nun durch Vergleiche die inkonsistenten Operatoren eliminiert werden. Dieses Verfahren übernimmt der zweite Pass, die *top down*-Elimination. Sind jetzt alle Operatormengen einelementig, dann wurde die Überladung erfolgreich aufgelöst. Enthält eine Menge kein Element bzw. mehrere Elemente, dann gibt der Algorithmus eine entsprechende Fehlermeldung aus.

Als Beispiel wollen wir mit diesem Algorithmus die Überladung in der Zuweisung *A* := −(1/2) an die **real**-Variable *A* auflösen. Für die betroffenen Operatoren seien folgende Operatordefinitionen sichtbar:

(1) − : **int** → **int**
(2) − : **real** → **real**
(3) − : **real** × **real** → **real**
(4) / : **int** × **int** → **int**
(5) / : **int** × **int** → **real**
(6) / : **real** × **real** → **real**

Der Typ der Variable *A* ist **real**. Deshalb ist es notwendig, daß der zugewiesene Ausdruck ebenfalls von diesem Typ ist. Wir können somit die sichtbare Definition (1) von der weiteren Betrachtung ausschließen, da der Resultatstyp dieser Operation **int** ist. Wir sehen weiterhin, daß die Operation − im zu untersuchenden Ausdruck −(1/2) einstellig ist, also nur einen Operanden hat. Die durch (3) definierte Operation erwartet dagegen zwei Operanden, womit auch (3) wegfällt. Die nun folgende Abb. 3.6 zeigt den mit den Operatormengen assoziierten Ausdrucksbaum von −(1/2) nach der Initialisierung:

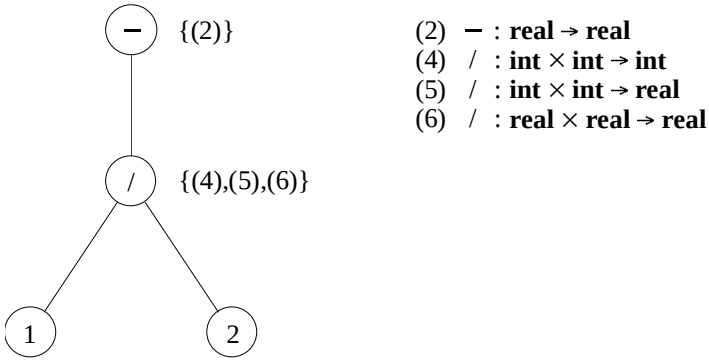


Abb. 3.6: Ausdrucksbaum nach der Initialisierung

Der Algorithmus berechnet bei der *bottom up*-Elimination die Typen der Operanden von /. Dabei handelt es sich jeweils um Konstanten des Typs **int**. Die Definition (6) in *ops*(/) erwartet jedoch Operanden mit dem Typ **real**. (6) wird aus der Menge entfernt. Betrachten wir nun die Operatormenge *ops*(−). Die Menge aller möglichen Resultatstypen des Operanden / ist {**int**, **real**}, wie sich aus (4) und (5) leicht ersehen läßt. Die einzige in *ops*(−) enthaltene Definition (2) hat als Parametertyp **real**, d.h. korrespondiert zu mindestens einem Typ in der Menge {**int**, **real**}. (2) kann also in *ops*(−) verbleiben (siehe Abb. 3.7):

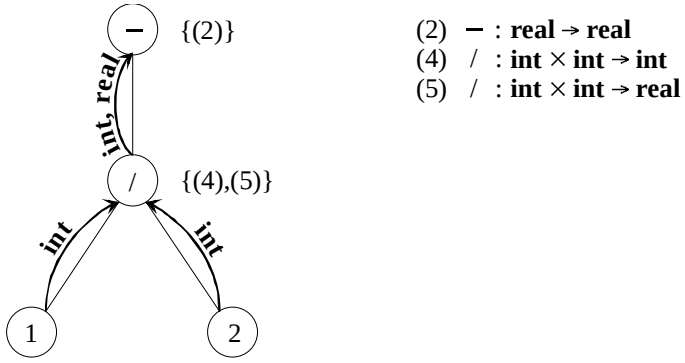


Abb. 3.7: Ausdrucksbaum nach der *bottom up*-Elimination

Wir kommen nun zur abschließenden *top down*-Eliminiation. *ops*(/) enthält nur noch zwei Definitionen (4) und (5). Die Menge aller Parametertypen (einstelliger Operator) aus der Menge *ops*(−) ist {**real**}. Der einzige dazu korrespondierende Resultatstyp der Operator-

definitionen aus *ops*(/) ist (5), (4) wird aus der Menge entfernt. Die Typen der Operanden von / liegen fest (vgl. Abb. 3.8):

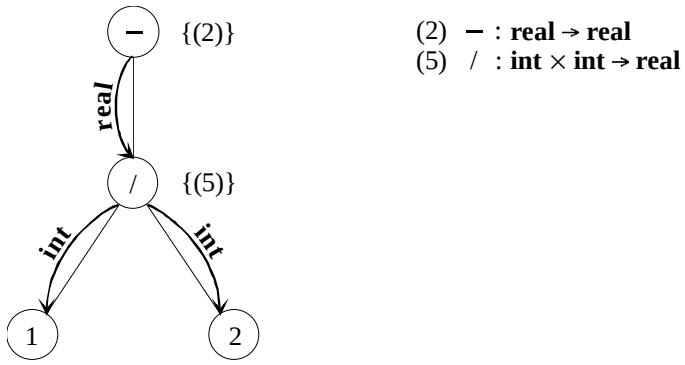


Abb. 3.8: Ausdrucksbaum nach der *top down*-Elimination

Alle *ops*-Mengen sind einelementig, d.h. die Überladung wurde vollständig aufgelöst.

3.5 Polymorphismus

Sprachen, die **parametrischen Polymorphismus** erlauben, gestatten die Definition von Funktionen, die für mehrere Kombinationen von Operanden- und Ergebnistypen dasselbe tun. Dieser Polymorphismus ist Bestandteil vieler funktionaler Programmiersprachen und der generischen Pakete in ADA.

Um dies zu bewerkstelligen, braucht man ein Modul, das feststellt, ob ein gemeinsames Typschema für die oben genannten Kombinationen von Typen existiert. Es ordnet jeder in einem typkorrekten Programm definierten, polymorph getypten Funktion ihren allgemeinsten Typ zu. Bei allen Compilern für Programmiersprachen mit polymorphen Typen ist ein solches Modul, der **Typinferenzalgorithmus**, implementiert.

Typterme (kürzer: **Typen**) werden aus Typvariablen, $\alpha, \beta, \gamma, ...$, die für beliebige Typen stehen, und aus Operatoren aufgebaut. Typterme heißen **polymorph**, wenn sie Typvariablen enthalten; solche Typterme ohne Typvariablen heißen **monomorph**. Nullstellige Operatoren sind beispielsweise **int** oder **bool**, die für die eingebauten Typen *int* bzw. *bool* stehen. Es ist zu beachten, daß mit dem Begriff *Typen* in diesem Fall keine Typtermdarstellung gemeint ist, sondern die eingebauten Typen der zugrundeliegenden Programmiersprache (hier die funktionale Sprache LAMA). Ein einstelliger Operator ist **list**. Zweistellige Operatoren sind der Konstruktor für Funktionstypen → und der Konstruktor für Paartypen ×. Typterme sind also z.B. Konstrukte der Form **list** $\alpha \rightarrow \beta$ oder $(\alpha \rightarrow \beta) \rightarrow \gamma$.

Hat man einen polymorphen Typ, so nennt man den Ersatz einer Typvariablen durch einen Typterm eine **Einsetzung**. Ist dieser Typterm monomorph, so spricht man über eine **monomorphe Einsetzung**. Alle vollständigen monomorphen Einsetzungen werden als **monomorphe Instantiierungen** bezeichnet. Hat man also beispielsweise den Funktionstypterm $\alpha \rightarrow \alpha$, so gibt es die monomorphen Instantiierungen **int → int**, **bool → bool**, **list bool → list bool**, usw.

3.5.1 Die Sprache LAMA

Wir verwenden als Beispielsprache die funktionale Sprache LAMA aus [WM92]. Die abstrakte Syntax der Programmiersprache LAMA sei, wie folgt, gegeben:

Elementname	Bereich	
e	E	Menge von Ausdrücken
v	V	Menge von Variablen
b	B	Menge von Basiswerten, z.B. boolesche Werte, integer, character, [],…
op_{un}	Op_{un}	Menge von unären Operatoren über Basiswerten z.B. $-$, **not**, **hd**, **tl**, **fst**, **snd**,…
op_{bin}	Op_{bin}	Menge von binären Operatoren über Basiswerten z.B. $+$, $-$, $=$, $\neq$, **and**, **or**, **cons**, **pair**,…

e = $b \mid v \mid (op_{un}\ e) \mid e_1\ op_{bin}\ e_2$

$\mid$ (**if** e_1 **then** e_2 **else** e_3)	bedingter Ausdruck
$\mid (e_1 e_2)$	Funktionsanwendung
$\mid (\lambda v.e)$	funktionale Abstraktion
$\mid$ (**letrec** $v_1 == e_1$; $v_2 == e_2$; ⋮ $v_n == e_n$ **in** e_0)	simultan rekursive Definitionen
$\mid$ (**let** $v_1 == e_1$; $v_2 == e_2$; ⋮ $v_n == e_n$ **in** e_0)	nicht rekursive Definitionen
$\mid [e_1,e_2,\ldots,e_n]$	Listenausdruck

Aus den Basiswerten und Variablen können mit bestimmten eingebauten Operatoren, sowie Funktionsabstraktion und Funktionsanwendung induktiv LAMA-Ausdrücke erzeugt werden. Es besteht die Möglichkeit, Variablen simultan rekursiv und nicht rekursiv zu definieren. Da die Sprache sehr einfach ist, sind nur wenige Erklärungen zum besseren Verständnis notwendig. Die Konstruktion $\lambda v.e$ definiert eine 1-stellige Funktion mit einem definierenden LAMA-Ausdruck e. Sie kann laut Syntax beliebig tief verschachtelt sein und läßt sich somit als n-stellige Funktion $\lambda v_1 \ldots v_n.e$ schreiben. Das trifft ebenso auf die Funktionsanwendung $(e_1 e_2)$ zu. Sie läßt sich als eine Anwendung $(e_1 \ldots e_m)$ auffassen und wird von rechts nach links ausgewertet. Ein *letrec*-Ausdruck führt ebenso wie ein *let*-Ausdruck n neue Namen $v_1,\ldots,v_n$ ein. Beide Definitionen unterscheiden sich durch den Gültigkeitsbereich G der neu eingeführten Variablen. Im *letrec*-Ausdruck umfaßt er für alle Variablen $G = \{e_0,e_1,\ldots,e_n\}$ und im *let*-Ausdruck für alle Variablen nur $G = \{e_0\}$.

3.5.2 Typinferenz

Wir führen das Prinzip der **Typinferenz** zunächst intuitiv ein und geben hierzu eine Reihe von Typkombinationsregeln für die zusammengesetzten LAMA-Konstrukte an. Die in die-

sem Abschnitt vorkommenden Begriffe und die Typkombinationsregeln werden im nächsten Abschnitt 3.5.3 formal definiert.

Eine Zuordnung von fest eingebauten Operatoren einer Programmiersprache mit polymorphen Typen zu Typtermen beschreibt eine **initiale Typumgebung** mit der die Typinferenz beginnt. Die initiale Typumgebung der Sprache LAMA ist, wie folgt, festgesetzt:

true, **false** :	**bool**	**plus**, **sub**, **mul**, **div** :	**int** → **int** → **int**
0, 1, 2, …:	**int**	**cons** :	$\alpha \rightarrow$ **list** $\alpha \rightarrow$ **list** α
succ, **pred**, **neg** :	**int** → **int**	**pair** :	$\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$
hd :	**list** $\alpha \rightarrow \alpha$	**fst** :	$(\alpha \times \beta) \rightarrow \alpha$
tl :	**list** $\alpha \rightarrow$ **list** α	**snd** :	$(\alpha \times \beta) \rightarrow \beta$
null :	**list** $\alpha \rightarrow$ **bool**		

Den zusammengesetzten LAMA-Konstrukten werden **Typkombinationsregeln** zugeordnet, durch deren Anwendung neue Typvariablen eingeführt werden. Eine solche Typkombinationsregel beschreibt Bedingungen auf den Typen der einzelnen Bestandteile des betreffenden LAMA-Konstrukts:

Das erste LAMA-Konstrukt, dem wir in diesem Zusammenhang begegnen, ist der **bedingte Ausdruck if** e_1 **then** e_2 **else** e_3. Er liefert die Typbedingungen e:**bool**, e_1:α und e_2:α mit der neu eingeführten Typvariable α. Die für e, e_1, e_2 berechneten Typen t, t_1, t_2 müssen diese Bedingungen erfüllen und an obige Typen durch *Unifikation* (vgl. Abschnitt 3.5.3.2) angepaßt werden. In den t, t_1, t_2 können sowohl α als auch Typvariable gebunden sein, wobei alle Vorkommen von α an den gleichen Typ gebunden werden müssen. Der Ergebnistyp des ganzen bedingten Ausdrucks ist α.

Die **Funktionsanwendung** e_1e_2 verlangt, daß e_1 einen Typ $\alpha \rightarrow \beta$ und e_2 den Typ α hat, und ergibt für die Anwendung den Typ β (Ergebnistyp), wieder mit neuen Typvariablen α und β.

Die **funktionale Abstraktion** $\lambda v.e$ hat den Ergebnistyp $\alpha \rightarrow \beta$, wobei α eine neue Typvariable für den Typ von v und β der Typ ist, der sich für e ergibt, wenn alle freien Vorkommen von v in e den Typ α zugeordnet bekommen.

In einem *let*-Ausdrucks **let** $v_1 == e_1;\ldots;v_n == e_n$ **in** e_0 werden zuerst alle Gleichungen typgeprüft. Das Resultat sind n Paare v_i:t_i, mit t_i Typ von e_i ($1 \le i \le n$). Für alle Vorkommen von v_i in e_0 setzt man verschiedene Versionen der t_i ein und berechnet dann den Typ t_0 von e_0, den Ergebnistyp des *let*-Ausdrucks. Zwei Versionen eines Typs t_i unterscheiden sich lediglich in der konsistenten Umbenennung aller Typvariablen in dem Typ. Wir nennen solche Typvariablen dann **generisch**.

In einer simultan rekursiven Definition **letrec** $v_1 == e_1;\ldots;v_n == e_n$ **in** e_0 bekommen die Vorkommen der v_i in den e_i ($1 \le i \le n$) nicht-generische Typvariablen und die in e_0 generische Typvariablen zugewiesen. Zuerst kreiert man eine Typumgebung $\{v_1:\alpha_1,\ldots,v_n:\alpha_n\}$ mit nichtgenerischen Typvariablen $\alpha_1,\ldots,\alpha_n$. Danach werden innerhalb dieser Typumgebung die Typen t_i der e_i ausgerechnet und diese Typen mit den α_i bzw. ihren Instanzen unifiziert. Der Typ t_0 für e_0 ergibt sich durch Einsetzung verschiedener (generischer) Versionen der t_i für alle Vorkommen der v_i in e_0. t_0 ist auch der Ergebnistyp des *letrec*-Ausdrucks.

Die Elemente eines **Listenausdrucks** $[e_1,e_2,\ldots,e_n]$ werden zuerst alle typgeprüft. Die für die e_i errechneten Typen $t_1,t_2,\ldots,t_n$ müssen durch Unifikation an die neu eingeführte Typvariable α angepaßt werden. Der Ergebnistyp des ganzen Ausdrucks $[e_1,e_2,\ldots,e_n]$ ist **list** α.

3.5.3 Formalisierung der Typinferenz

Um einen Typinferenzalgorithmus formulieren zu können, müssen die im letzten Abschnitt vorgestellten Regeln zunächst formalisiert werden. Es muß also zuerst formal spezifiziert werden, welche LAMA-Ausdrücke welchen Typ haben. Man bezeichnet eine solche Spezifikation als ein **Typinferenzsystem**. Ein darauf aufgebauter Typinferenzalgorithmus sollte dann den allgemeinsten Typ eines polymorphen Ausdrucks herausfinden oder einen Fehler ausgeben, wenn kein solcher Typ gefunden wird. Diese Aufgabe ist in [Mil78] beschrieben. Die hier gegeben Erläuterungen basieren auf [Rea89] und [DM82].

3.5.3.1 Grundlagen

Wir haben im letzten Abschnitt gesehen, daß die Typkombinationsregeln relativ zu Typumgebungen arbeiten. Eine solche Umgebung läßt sich als eine Menge A von **Annahmen** der Form $v{:}t$ formulieren, wobei v eine Variable und t ein Typterm ist.

Definition 3.5.1 (Typumgebung)
Eine **Typumgebung** A ist eine endliche Menge von Paaren $\{v_1{:}t_1,\ldots,v_n{:}t_n\}$, wobei die Variablen v_i und v_j für $i \neq j$ paarweise voneinander verschieden sind. Ein Paar $v_i{:}t_i$ bezeichnet man als eine **Annahme**. □

Betrachten wir die Regel zum bedingten Ausdruck **if** e_1 **then** e_2 **else** e_3, so kann man zur Beschreibung dieser Regel folgende Notation verwenden:

$$\frac{A \vdash e_1 : \text{bool} \quad A \vdash e_2 : t \quad A \vdash e_3 : t}{A \vdash (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) : t}$$

Die Aussage unter der Linie (Konklusion) folgt aus der Aussage über der Linie (Hypothese). Ein Ausdruck der Art $A \vdash e{:}t$ wird gelesen als „der Ausdruck e hat unter den Annahmen A den Typ t“. Eine einzelne Annahme sei mit $[v{:}t]$ und eine leere Menge von Annahmen mit $\vdash e{:}t$ beschrieben. $A(v) = t$ bedeutet, daß v durch A mit dem Typterm t assoziiert wird.

Um generische von nicht-generischen Typvariablen zu unterscheiden, führen wir den Begriff des **Typschemas** ein:

Definition 3.5.2 (Typschema)
Ein **Typschema** S hat die folgende Form:

$\forall\alpha_1\forall\alpha_2\ldots\forall\alpha_n.t$ (abgekürzt mit $\forall\alpha_1\alpha_2\ldots\alpha_n.t$)

Hier ist t ein beliebiger polymorpher Typ und die $\alpha_1,\ldots,\alpha_n$ seien **generische Typvariablen**. Ein Typ t ist unter dieser Definition ein Spezialfall eines Typschemas. □

Quantifizierte (in diesem Zusammenhang: $\forall$-quantifizierte) Typvariablen heißen **gebundene** Typvariablen. Hat man eine Quantifizierung $\forall\alpha_1\alpha_2\ldots\alpha_n.t$, so umfaßt die Bindung der $\alpha_1,\ldots,\alpha_n$ den **Gültigkeitsbereich** t. Ungebundene Typvariablen bezeichnet man als **frei**.

In einem *let*-Ausdruck **let** $v_1 == e_1;\ldots;v_n == e_n$ **in** e_0 konnten wir den Vorkommen der v_i in e_0 erlauben, verschiedene generische Instanzen der durch die jeweiligen e_i $(1 \leq i \leq n)$ gegebenen polymorphen Typen zu haben. Hierzu definieren wir zunächst einige notwendige Begriffe:

Definition 3.5.3 (Substitution)
Eine **Substitution (Ersetzung)** σ von Typen für Typvariablen ist eine endliche Menge von Paaren $[t_1/\alpha_1,\ldots,t_n/\alpha_n]$, wobei die Typvariablen α_i und α_j für $i \neq j$ paarweise voneinander verschieden sind. Jedes Paar t_i/α_i heißt eine **Bindung** von α_i. □

Definition 3.5.4 (Instanz)
Sei $\sigma = [t_1/\alpha_1,\ldots,t_n/\alpha_n]$ eine Substitution von Typen für Typvariablen und sei S ein Typschema. Eine **Instanz** von S ist ein Typschema S_σ, das man erhält, wenn man jedes freie Vorkommen der α_i in S durch t_i $(1 \leq i \leq n)$ ersetzt und, falls nötig, die generischen Variablen von S umbenennt. □

Nun können wir den Begriff der *generischen Instanzen* exakt definieren:

Definition 3.5.5 (generische Instanz)
Ein Typschema $S = \forall\alpha_1\ldots\alpha_m.t$ hat eine **generische Instanz** $S' = \forall\beta_1\ldots\beta_n.t'$, wenn $t' = [t_i/\alpha_i]t$ für einige Typen $t_1,\ldots,t_m$ und die β_j *nicht frei* in S sind. In diesem Fall werden wir $S \succ S'$ schreiben. □

Definition 3.5.6 (Komposition von Substitutionen)
Seien $\sigma = [t_1/\alpha_1,\ldots,t_m/\alpha_m]$ und $\psi = [t'_1/\beta_1,\ldots,t'_k/\beta_k]$ Substitutionen. Dann erhält man ihre **Komposition** ψ nach σ, in Zeichen $\psi\sigma$, aus der Menge $[t'_1/\beta_1,\ldots,t'_k/\beta_k,\psi t_1/\alpha_1,\ldots,\psi t_m/\alpha_m]$ durch Streichen der Paare t'_i/β_i mit $\beta_i \in \{\alpha_1,\ldots,\alpha_m\}$ und der Paare $\psi t_i/\alpha_i$ mit $\alpha_i = \psi t_i$. □

Die Streichungen sorgen dafür, daß das Ergebnis der Komposition wieder die minimale Darstellung einer Substitution ergibt. Wir benötigen die Komposition von Substitutionen für die Definition der Unifikation.

3.5.3.2 Unifikation

Das für die Implementierung eines Typinferenzalgorithmus benötigte Schlüsselwerkzeug ist ein **Unifikator**. Er basiert auf einem Standardalgorithmus, der zuerst in [Rob65] definiert wurde. Unifikation wird auch bei der Implementierung von logischen Programmiersprachen verwendet, etwa PROLOG. Formal ist ein Unifikator, wie folgt, definiert:

Definition 3.5.7 (Unifikator)
Eine Substitution σ ist ein **Unifikator** für zwei Typterme t_1 und t_2, wenn $\sigma t_1 = \sigma t_2$. Ein Unifikator σ für zwei Typterme t_1 und t_2 heißt ein **allgemeinster Unifikator** für die zwei Typterme, wenn für jeden Unifikator ψ dieser Typterme eine Substitution ϕ existiert, so daß $\psi = \phi\sigma$. □

In dieser Anwendung bekommt der Unifikator U als Eingabe zwei Typen t_1, t_2 und liefert eine **allgemeinste Ersetzung** σ für die in den Typen vorkommenden Typvariablen. Wendet man σ auf t_1 und t_2 an, so erhält man einen gemeinsamen Instanztyp dieser beiden Typen. Beispielsweise sind die beiden Typen

$$\alpha \to \alpha \quad \text{und} \quad (\beta \times \textbf{int}) \to \gamma$$

kompatibel und können mit der folgenden allgemeinsten Ersetzung σ unifiziert werden:

$$(\beta \times \textbf{int}) \text{ für } \alpha \quad \text{und} \quad (\beta \times \textbf{int}) \text{ für } \gamma \quad \Leftrightarrow \quad \sigma = [(\beta \times \textbf{int})/\alpha, (\beta \times \textbf{int})/\gamma]$$

Dies ergibt den unifizierten Typ ($\beta \times$ **int**) $\rightarrow$ ($\beta \times$ **int**). Eine Ersetzung $\sigma' = [$(**int** $\times$ **int**)/α, (**int** $\times$ **int**)/γ, **int**/β] unifiziert ebenfalls die o.g. Typen. Sie ist aber keine allgemeinste Ersetzung, denn der nun resultierende unifizierte Typ (**int** $\times$ **int**) $\rightarrow$ (**int** $\times$ **int**) ist eine Instanz des allgemeinsten Typ, der gefunden werden kann, nämlich ($\beta \times$ **int**) $\rightarrow$ ($\beta \times$ **int**). Formal ist σ' kein allgemeinster Unifikator der beiden Typen, da $\sigma' = \phi\sigma$ mit $\phi = [$**int**/$\beta]$. Im Gegensatz dazu, können die folgenden Typen

$\alpha \rightarrow \alpha$ und $(\beta \times$ **int**$) \rightarrow$ **bool**

nicht unifiziert werden, weil es keine Ersetzung zur Unifikation der beiden Typen gibt. Fehler wie dieser kommen vor, wenn zwei Typen entweder wie im vorigen Fall inkompatibel sind, oder wenn eine Variable mit einem Typterm identifiziert wird, der dieselbe Variable enthält. Ein Beispiel dieses Problems des **unendlichen Typs** ist etwa die Unifikation der beiden Typen:

$\alpha \rightarrow \alpha$ und $(\beta \times$ **int**$) \rightarrow \beta$

Um dieses Problem zu erkennen, ist ein sogenannter **Vorkommenstest** („occurs check“) nötig, der allerdings in den meisten Systemen aus Kostengründen nicht durchgeführt wird. Unsere Implementierung enthält jedoch diesen Test. Die eingegebenen Beispielprogramme sind meist so klein, daß dieser Test nicht ins Gewicht fällt. Stößt der Unifikator auf einen Fehler, so gibt er eine entsprechende Meldung aus, d.h. er terminiert immer.

3.5.3.3 Typinferenzsystem

Wir geben nun das verwendete **Typinferenzsystem** an. Dazu wollen wir zunächst eine spezielle Schreibweise einführen, um lediglich Regeln angeben zu müssen, welche die Ausdrükke der Sprache LAMA selbst betreffen und nicht etwa noch Regeln zu Instantiierung etc. Diese Schreibweise wurde in [Clé87] erstmals eingeführt.

Definition 3.5.8 (Abschluß eines Typs *t* unter der Typumgebung *A*)
Sei *S* ein Typschema, dann ist *FV* (*S*) die Menge aller freier (nicht quantifizierter) Variablen in *S*. Weiterhin bedeutet *FV* (*A*) für eine Menge von Annahmen (Typumgebung) *A*, die Vereinigung aller freien Typvariablen, die in den durch *A* mit Variablen assoziierten Typen vorkommen. Nun sei ***Gen* (*A, t*)** ein Typschema mit:

$Gen(A, t) = \forall\alpha_1\alpha_2\ldots\alpha_n.t$ mit $\{\alpha_1,\ldots,\alpha_n\} = FV(t) - FV(A)$

Gen (*A*, *t*) heißt der **Abschluß von *t* unter *A***. □

Die α_i ($1 \le i \le n$) sind also alle freien Typvariablen in *t*, die nicht frei in *A* sind. Gilt $S = Gen(A, t)$, dann sind alle in *t* nicht-generischen Variablen in *S* generisch gemacht worden, außer denjenigen, die schon in *A* nicht-generisch sind. Jetzt sind alle Voraussetzungen für die Angabe des Typinferenzsystems gegeben:

Typinferenzsystem	
VAR:	$A \vdash v : t \quad \text{mit } A(v) = S \succ t$
CON:	$A \vdash b : t \quad \text{mit } A(b) = S \succ t$
APP:	$\dfrac{A \vdash e : t' \rightarrow t \qquad A \vdash e' : t'}{A \vdash (ee') : t}$

Typinferenzsystem	
LIST:	$\dfrac{A \vdash e_1 : t \quad A \vdash e_2 : t \quad \ldots \quad A \vdash e_n : t}{A \vdash [e_1, e_2, \ldots, e_n] : \textbf{list}\ t}$
COND:	$\dfrac{A \vdash e_1 : \text{bool} \quad A \vdash e_2 : t \quad A \vdash e_3 : t}{A \vdash (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) : t}$
ABS:	$\dfrac{A + [v : t'] \vdash e : t}{A \vdash (\lambda v.e) : t' \to t}$
LET:	$\dfrac{A \vdash e_1 : t_1 \ldots A \vdash e_n : t_n \quad A + [v_1 : S_1] + \ldots + [v_n : S_n] \vdash e : t}{A \vdash (\textbf{let}\ v_1 == e_1\ ; \ldots ; v_n == e_n\ \textbf{in}\ e) : t}$ mit $S_i = Gen(A, t_i)$ und $1 \le i \le n$
LETREC:	$\dfrac{A + A' \vdash e_1 : t_1 \ldots A + A' \vdash e_n : t_n \quad A + A'' \vdash e : t}{A \vdash (\textbf{letrec}\ v_1 == e_1\ ; \ldots ;\ v_n == e_n\ \textbf{in}\ e) : t}$ mit $A' = [v_1 : t_1] + \ldots + [v_n : t_n]$ und $A'' = [v_1 : Gen(A, t_1)] + \ldots + [v_1 : Gen(A, t_2)]$

Der Typinferenzalgorithmus, der dieses Typinferenzsystem verwendet, ist im Anhang B.4 angegeben und erläutert. Wir geben ein einfaches Beispiel an, um zu demonstrieren, wie der Typinferenzalgorithmus funktioniert. Angenommen, wir wollen den allgemeinsten Typ des LAMA-Ausdrucks ((*λx.x*) 5) unter der initialen Typumgebung [5:**int**] herleiten. Wir müssen einen Beweis für die Aussage der Form

$[5:\textbf{int}] \vdash ((\lambda x.x)\ 5) : t$

konstruieren und dabei den Typ *t* bestimmen. Diese Aussage kann nur durch die Verwendung der Regel *APP* erzeugt werden:

$$\frac{[5:\textbf{int}] \vdash ((\lambda x.x)\ 5) : t' \to t \quad [5:\textbf{int}] \vdash 5 : t'}{[5:\textbf{int}] \vdash ((\lambda x.x)\ 5) : t}\ APP$$

Nun sind *t* und *t′* zu bestimmen. Zur Behandlung der zweiten Prämisse ist die Regel *CON* zu verwenden. Es gilt

$[5:\textbf{int}]\ (5) = \textbf{int} \succ t'$

Daraus folgt direkt, daß t′ = **int**. Die erste Prämisse wird mit der Regel *ABS* ausgewertet:

$$\frac{[5:\text{int}] + [x : t'] \vdash x : t}{[5:\text{int}] \vdash (\lambda x.x) : t' \to t}\ ABS$$

Deren Prämisse muß mit *VAR* behandelt werden:

$[5:\textbf{int}] + [x:t']\ (x) = t' \succ t$

Jetzt folgt $t = t' = \textbf{int}$. Der Beweis ist vollständig und $t = \textbf{int}$ wurde inferiert.

Kapitel 4

Prinzipien zur Erstellung von Animationssoftware

Wir wollen nun einige **Prinzipien und Merkregeln** für die Erstellung von Oberflächen und Animationen angeben. Sie basieren im Zusammenhang mit Animationen meist lediglich auf praktischen Erfahrungen und sind stark von dem Themengebiet, das dargestellt wird, abhängig. Einige Prinzipien können sich auch widersprechen, da sie gegenläufig angelegt sind.

Wir orientieren uns bei den Prinzipien zur Gestaltung von Oberflächen an Erkenntnissen aus dem Bereich der interaktiven Benutzerschnittstellen. Als weiterführende Literatur sei an dieser Stelle auf [Shn92] hingewiesen.

Die beschriebenen Prinzipien finden Anwendung in der *Animation der semantischen Analyse*, wenn auch alle Anforderungen in der Praxis nicht erreicht werden können. In den nachfolgenden Kapiteln verweisen wir beispielhaft auf einige Prinzipien, die beachtet wurden. Weiterhin machen wir auf einige Stellen aufmerksam, die gegen ein Prinzip verstoßen.

4.1 Ziele

Die Eigenschaften, die ein Lern- und Präsentationsprogramm, wie die *Animation der semantischen Analyse*, aufweisen sollte, lassen sich von zwei unterschiedlichen Standpunkten betrachten. Einerseits stellen die BenutzerInnen bestimmte Erwartungen an das Programm, andererseits stehen den AutorInnen nur begrenzte Ressourcen zur Verfügung, um diese Anforderungen zu erfüllen.

Da der Inhalt unseres Programms aus dem Bereich der Informatik stammt, können wir bei den BenutzerInnen Computergrundkenntisse voraussetzen, d.h. die Bedienung von Elementen der Windows-Oberfläche sollte bekannt sein, etwa das Aufrufen und die Bedienung einer Online-Hilfe.

4.1.1 Benutzersicht

Fehlerbehaftete Programme mindern die Motivation der AnwenderInnen. Daher sollte **Fehlerfreiheit** des Inhalts eines Lernprogrammes und **angemessene Funktionalität** der technischen Bestandteile das wichtigste Ziel sein. Weiterhin gibt es eine Vielzahl von Zielen, die Beachtung finden sollten:

- *Zuverlässigkeit*: Kommandos und Aktionen sollten genau das bewirken, wofür sie spezifiziert wurden. Aktualisierungen von Informationen oder Bildschirmbereichen müssen korrekt sein.
- *Verfügbarkeit*: Die Software-Architektur, Hardwarekomponenten und Netzwerkunterstützung muß eine hohe Verfügbarkeit des Programms sicherstellen.
- *Sicherheit*: Es sollte ein Schutzmechanismus vor unerwarteten bzw. unerlaubten Zugriffen, besonders im Netzwerk, sowie für die unbeabsichtigte Zerstörung von Daten existieren.
- *Integrität*: Es ist die Integrität der darzustellen Daten zu gewährleisten.
- *Standardisierung*: Das Fehlen von Standardisierungsmaßnahmen jeglicher Art führt zu längerer Einarbeitungszeit, Bedienungsfehlern und Inkompatibilitäten.
- *Integration*: Ein Programm sollte zusammen mit anderen Programmen verwendet werden können. Dies gilt auch hinsichtlich seiner Daten.
- *Konsistenz*: Gemeinsame Aktionssequenzen, Einheiten, Layouts, Farben, Typographie, etc. innerhalb einer Anwendung.
- *Portabilität*: Ein Programm sollte so konstruiert sein, daß es die Konvertierung von Daten und Oberflächen über mehrere Soft- und Hardwareumgebungen gestattet (z.B. über die Zwischenablage).

4.1.2 Autorensicht

Die EntwicklerInnen von Programmen haben als Ziel einerseits

- das Budget, andererseits
- die Zeitplanung (Schedules) und
- die Machbarkeit

zu beachten.

4.2 Design von Dialogen

Unter Dialogdesign sei nicht nur die Gestaltung von Dialogfenstern verstanden, sondern alle Bereiche in denen AnwenderInnen mit dem Programm interagieren können, also z.B. auch die Menüführung, Kommandosprachen, Erklärungen des Programms, etc.

(1) Streben nach Konsistenz: Darunter fallen konsistente Aktionssequenzen bei ähnlichen Situationen, identische Terminologie im Zusammenhang mit Erklärungen und Dialogfenstern, etc.

(2) Häufigen BenutzerInnen Shortcuts ermöglichen: Kürzere Antwortzeiten bei Anfragen und schnellere Bildschirmdarstellung sind neue Anreize für BenutzerInnen, die das Programm schon öfter verwendet haben.

(3) Informatives Feedback anbieten: Für jede Anwenderaktion sollte das Programm ein Feedback geben. Bei häufigen und untergeordneten Aktionen kann die Antwort klein ausfallen (z.B. das visuelle Feedback beim Klicken auf eine 3D-Schaltflächen durch

den Anschein des Eindrückens), bei seltenen und übergeordneten Aktionen jedoch ausführlich.

(4) In sich abgeschlossene Dialoge liefern: Organisation von Aktionssequenzen in Gruppen mit einem Anfang, einer Mitte und einem Ende. Dies gibt dem/der AnwenderIn das Gefühl des abgeschlossenen Ganzen und Definiertheit der einzelnen Elemente.

(5) Leichte Fehlerbehandlung anbieten: Programme sollten so konzipiert sein, daß Folgefehler vermieden werden. Falls ein Fehler passiert, so hat das Programm den Fehler zu lokalisieren und ein einfaches, verständliches Lösungsschema für diesen Fehler anzugeben.

(6) Zulassen, daß Aktionen rückgängig gemacht werden können: Soweit überhaupt möglich, sollten Aktionen zurückgenommen werden können. Deren Anzahl bzw. Umfang hängt von der Anwendung ab.

(7) Herstellung von Kausalzusammenhängen: Das Programm sollte keine überraschenden Aktionen durchführen. Die AnwenderInnen sollten sofort erkennen, warum sie beispielsweise etwas eingeben müssen. Sie sind die Initiatoren von Aktionen und nicht die Antwortenden unmotivierter Fragen des Programms.

(8) Wenig Informationen im Kurzzeitgedächtnis: Es sollte nicht verlangt werden, daß sich die AnwenderInnen Kodierungen, Regeln, o.ä. merken müssen. Es ist nötig, daß solche Informationen permanent, z.B. als Online-Hilfe, nachschlagbar sind. Weiterhin sind in diesem Kontext der Bildschirmaufbau einfach und die Notwendigkeit häufiger Fensterwechsel klein zu halten.

Diese Prinzipien müssen für jede Umgebung neu interpretiert, verfeinert und gegebenenfalls erweitert werden.

4.3 Oberflächen

Besonders wenn viele EntwicklerInnen an einem Programm arbeiten, müssen Richtlinien für die **Oberflächengestaltung** geschaffen werden. Wie in Abschnitt 4.2 sind die im folgenden genannten Richtlinien lediglich als Anhaltspunkte zu verstehen, die für jedes Projekt entsprechend anzupassen und zu erweitern sind:

(1) Konsistenz der Oberfläche: Schon während der Entwicklungsphase sollten Terminologien, Formate, Farbauswahl, Aufteilung, Größen etc. standardisiert und permanent kontrolliert werden.

(2) Effiziente Informationsaufnahme durch den/die AnwenderIn: Die Oberfläche sollte vertraut und gut strukturiert sein, damit die AnwenderInnen sie leicht erfassen und die dargestellten Informationen gut aufnehmen können.

(3) Minimale Merkanforderungen der AnwenderInnen: Es sollte nicht verlangt werden, daß man sich an Informationen eines Fensters (Seite) erinnern muß, um ein anderes Fenster (Seite) zu verstehen.

(4) Kompatibilität von Datendarstellung und Datenmaterial: Zwischen der Darstellung von Daten (z.B.: Syntaxbaum) und dem Datenmaterial (z.B.: Eingabeprogramm) muß ein klar nachvollziehbarer Zusammenhang bestehen.

(5) Flexibilität der Oberfläche für Benutzeranpassungen: BenutzerInnen haben unterschiedliche Vorlieben und Begabungen Informationen aufzunehmen. Daher kann es

angebracht sein, flexible Oberflächengestaltung zuzulassen. Fortgeschrittenen sollte es gestattet sein, selbst zu entscheiden, in welcher Weise und wie genau Informationen dargestellt werden.

(6) Initialisierter Zustand: Vorallem bei veränderbaren Oberflächen ist es notwendig, daß ein fest initialisierter Zustand eingestellt werden kann (z.B. Fokus auf wichtige Schaltflächen). Dies ist besonders bei neuen AnwenderInnen wichtig.

Einen Bestandteil der Oberfläche bilden die Programmelemente, deren Zweck es ist, die **Dateneingabe** zu ermöglichen, etwa Editierfenster oder Dialogboxen. Da ein schlechter Entwurf solcher Elemente unmittelbar die gute Bedienbarkeit des Programms herabsetzt, seien die hierfür geltenden Prinzipien gesondert aufgeführt:

(1) Konsistenz der Transaktionen für Dateneintrag: Unter allen Bedingungen sollten gleichartige Aktionssequenzen, Grenzen, Abkürzungen, usw. benutzt werden.

(2) Geringe Anzahl von Eingabeaktionen durch die AnwenderInnen: Dieses Prinzip bewirkt eine kürzere Einarbeitungszeit und weniger Eingabefehler. Statt große Tastatureingaben machen zu müssen, sollte der/die AnwenderIn mit einem Zeigegerät (Maus, Lightpen) aus einer Auswahl die Daten selektieren können. Hierbei ist es wichtig, daß redundante Dateneinträge vermieden werden.

(3) Minimale Merkanforderungen an die AnwenderInnen: Die AnwenderInnen sollten sich nicht kryptische Symbole oder syntaktisch komplexe Kommandostrings merken müssen.

(4) Kompatibilität von Dateneinträgen und Datendarstellungen: Das Format der Eingabeinformationen und der schließlich angezeigten Informationen sollte von den AnwenderInnen als zusammengehörig assoziiert werden.

(5) Flexibilität der Dateneingabemöglichkeiten: Einige AnwenderInnen bevorzugen bestimmte Eingabeformen, etwa Drag-and-Drop statt über die Tastatur. Es sollte mehrere Eingabemöglichkeiten zur Verfügung stehen, was jedoch dem Konsistenzprinzip widerspricht.

4.4 Animationen

Animationen spielen sich ebenfalls auf der Benutzeroberfläche eines Programms ab, daher gelten hier auch die im vorhergehenden Abschnitt 4.3 erwähnten Prinzipien, z.B. das Konsistenzprinzip. Allerdings sollten für Animationen eigene Richtlinien geschaffen werden, welche die Dynamik von Animationen berücksichtigen:

(1) Flexible Steuerung: Animationen sollten in der Geschwindigkeit regelbar, schrittweise durchführbar, zu jedem Zeitpunkt zu stoppen und rückzusetzen sein. Ein Rückwärtslaufenlassen von Animationen ist nur bedingt sinnvoll und häufig auch technisch schwierig umzusetzen. Als Alternative ist eine *Undo*-Operation jedoch angebracht.

(2) Klar definierte Objektbewegungen: Bewegungen von Objekten sollten möglichst direkt zu ihrem Ziel, aber nicht über zu viele andere Objekte hinweg erfolgen. Es sollten nicht mehrere Objekte gleichzeitig bewegt werden und die Bewegung selbst nicht zu komplex und nicht ruckartig sein.

(3) Unmittelbares Feedback von Benutzeraktionen: Gerade bei Animationen ist ein optischen Feedback des Systems bei Benutzeraktionen wichtig. Wird eine Animation

angehalten, so sollte dieser Stopp sofort erfolgen und die Animation nicht erst noch eine Weile weiterlaufen.

(4) Minimale Merkanforderungen an die AnwenderInnen: Animationen und entsprechende Erklärungen sollten innerhalb eines räumlichen und logischen Rahmens ablaufen. Bei dynamischen Animationen kollidiert dieses Prinzip häufig mit zu hohem Platzanspruch.

(5) Highlights: Ab und zu können gewisse Teile von Animationen etwas spektakulärer ausfallen, als es der reine Informationsgehalt der Animation erfordert. Dies dient der Steigerung der Aufmerksamkeit, Motivation und Neugier auf spätere Animationen.

Der ein oder andere Punkt kann je nach Anwendungsgebiet eine stärkere bzw. schwächere Gewichtung erhalten. Kommerzielle Präsentationen auf Messen enthalten oft viele spektakuläre Animationen, die nur die wichtigsten Informationen übermitteln sollen, wogegen wissenschaftliche Lernsoftware mehr Wert auf didaktisch sinnvolle Animationen legt.

4.5 Aufmerksamkeit auf Objekte lenken

Gerade bei Lern- und Präsentationssoftware ist es notwendig, bestimmte Objekte, die beispielsweise gerade erläutert werden, in den **Blickpunkt** zu stellen. Hierfür eignen sich mehrere Techniken, die sowohl bei der Oberflächengestaltung, als auch beim Animationsdesign Anwendung finden:

- *Intensität*: Eine Veränderung der Intensität von Bildpunkten wird bei modernen Farbdisplays nur noch selten verwendet.
- *Markierung*: Unterstreichen, umrahmen, mit einem Pfeil bezeigen, mit Sonderzeichen (•, ∗, …) indizieren, usw.
- *Größe*: Vergrößern von Objekten.
- *Fontwahl*: Verwendung von mehreren verschiedenen Schriftarten.
- *Invertierung*: Benutzung von Farbinvertierungen.
- *Aufblitzen*: Man kann manche Displays oder Fenster samt Inhalt aufblitzen lassen.
- *Farbe*: Signalfarben ziehen Aufmerksamkeit auf sich. Bestimmte Farben sollten für fest definierte Zwecke reserviert sein.
- *Farbenblinken*: Wechsel von Farben, um ein Blinken zu simulieren.
- *Akustik*: Beispielsweise können „sanfte“ Töne ein positives Feedback symbolisieren, „harte“ Töne jedoch Warnungen.

Das zu häufige Anwenden dieser Techniken birgt jedoch auch Gefahren. Neue AnwenderInnen brauchen einfache, logisch organisierte und gut „etikettierte“ Oberflächen, die ihre Aktionen führen. Fortgeschrittene benötigen diese Art der Führung nicht. Sie bevorzugen subtilere Hervorhebungen. Die AutorInnen müssen bei der Entwicklung einen Mittelweg finden oder beides ermöglichen.

4.6 Farben

Im letzten Abschnitt haben wir gesehen, daß die Verwendung von **Farben** eine von mehreren Möglichkeiten ist, die Aufmerksamkeit auf Objekte zu lenken. Weil Farben bei der Gestaltung der *Animation der semantischen Analyse* eine der wohl wichtigsten Rollen gespielt haben, wollen wir sie hier etwas genauer betrachten.

Farben haben gewisse Eigenschaften, die dem Programm beim Gebrauch von Farben einen Vorteil verschaffen, sie

- beruhigen oder erregen das Auge,
- betonen eine einfache Oberfläche,
- erlauben feine Unterschiede in komplexen Oberflächen,
- heben die logische Organisation von Informationen hervor,
- verstärken die Beachtung von Warnungen,
- bewirken emotionalere Reaktionen.

Die große Verfügbarkeit von Farben, die durch die heutigen leistungsfähigen Farbbildschirme und Graphikkarten erreicht wird, verleitet leicht zu Farbmißbrauch. Deshalb bieten wir einige Anhaltpunkte zum Gebrauch von Farben an:

(1) Sparsame Benutzung von Farben: Zu viele Farben überfrachten den Bildschirm mit optischen Informationen leerer Semantik. Identische Farben sollten dann benutzt werden, wenn eine Beziehung zwischen den Objekten besteht.

(2) Farben ermöglichen die Beschleunigung bzw. die Verzögerung von Aufgaben: Beispielsweise bewirkt das sogenannte *Syntax-Coloring* neuerer Editoren für bestimmte Programmiersprachen das schnelle Erkennen syntaktischer Strukturen, wodurch Syntaxfehler vermieden werden können.

(3) Farbkodierungen erhöhen die Informationsdichte: In dichten Graphen können sich überschneidende Kanten z.B. unterschiedlich eingefärbt werden, um eine bessere Unterscheidung zu erreichen.

(4) Sicherstellung, daß die Farbkodierung die Aufgabe unterstützt: Keine willkürliche Färbung von Objekten.

(5) Farbkodierung soll konsistent sein: Die Regeln, die die EntwicklerInnen für die Farbkodierung von Objekten anwenden, sollten im gesamten Programm gelten, etwa daß alle anwählbaren Textelemente blau gefärbt sind

(6) Veränderung der Farbkodierung durch den/die AnwenderIn: In einigen Anwendungen kann es eventuell sinnvoll sein, daß die verwendete Farbkodierung veränderbar ist, etwa dann, wenn der/die AnwenderIn ein bestimmtes Objekt mit einer festen Farbe verbindet.

Bei der Auswahl der Farben ist darauf zu achten, daß diese auf möglichst vielen Ausgabegeräten gleich gut angezeigt werden. Hier fließt eine gewisse Hardwareabhängigkeit der Farbdarstellung in die Entscheidung mit ein.

Kapitel 5

Die animierte Präsentation

Die *Animation der semantischen Analyse* ist in zwei Bestandteile zerlegt: erstens die **animierte Präsentation** des Lehrstoffes, die Thema dieses Kapitels ist, und zweitens eine Erweiterung (ASA-Tool), die es erlaubt, die in der Präsentation vorgestellten Algorithmen an frei wählbaren Eingabebeispielen dynamisch zu testen und zu untersuchen. Diese Aufteilung der Aufgaben ist in den Einschränkungen des ToolBook-Systems begründet, etwa die Beschränkung der Seitengröße (siehe Abschnitt 2.6 und 6.1).

Die *animierte Präsentation* wurde mit Hilfe des Autorensystems *Multimedia ToolBook 3.0* implementiert. Sie stellt die Aufgaben der semantischen Analyse vor (in dem Umfang, wie in Kapitel 3 dargelegt), erläutert die Problematik an vielen kleinen Animationen und bietet Problemlösungen anhand von Algorithmen (Deklarations-Analysator, Typkonsistenz-Analysator, Auflösung der Überladung und Typinferenzalgorithmus; siehe Anhang B) an. Diese werden jeweils durch Algorithmenanimationen an einem festen, statischen Beispiel erklärt. Anhang C.1 zeigt einige Programmbeispiele der *animierten Präsentation*.

Eine Verknüpfung mit anderen ToolBook-Anwendungen (Büchern) ist ohne Probleme zu realisieren, da MTB 3.0 bereits ein entsprechendes Konzept dafür zur Verfügung stellt. Die *animierte Präsentation* der semantischen Analyse kann somit in eine größere Umgebung eingebunden werden, beispielsweise eine komplette Präsentation der wichtigsten Konzepte der Übersetzung von Programmiersprachen (lexikalische Analyse, syntaktische Analyse, etc.).

5.1 Starten des Programms

Wie bei jeder andere Window-Anwendung erfolgt der Start des Präsentationsprogramms nach der Installation des ToolBook-Laufzeitsystems entweder durch den Programm- oder den Dateimanager. Das Präsentationsprogramm ist diesbezüglich von seiner dynamischen Erweiterung vollständig unabhängig und lauffähig. Jeder Programmstart hat zwei aufeinanderfolgende Dialoge zur Folge (vgl. Abb. C.1.1):

1. *Eingabe des Benutzernamens*, um den/die BenutzerIn in darauffolgenden Dialogen direkt ansprechen und die verschiedenen BenutzerInnen unterscheiden zu können.
2. *Sprung an die zuletzt besuchte Seite*, wenn der/die BenutzerIn dem System nicht unbekannt ist. Standardmäßig eröffnet das Programm die Sitzung auf der Kapitelseite (Inhaltseite) der semantischen Analyse.

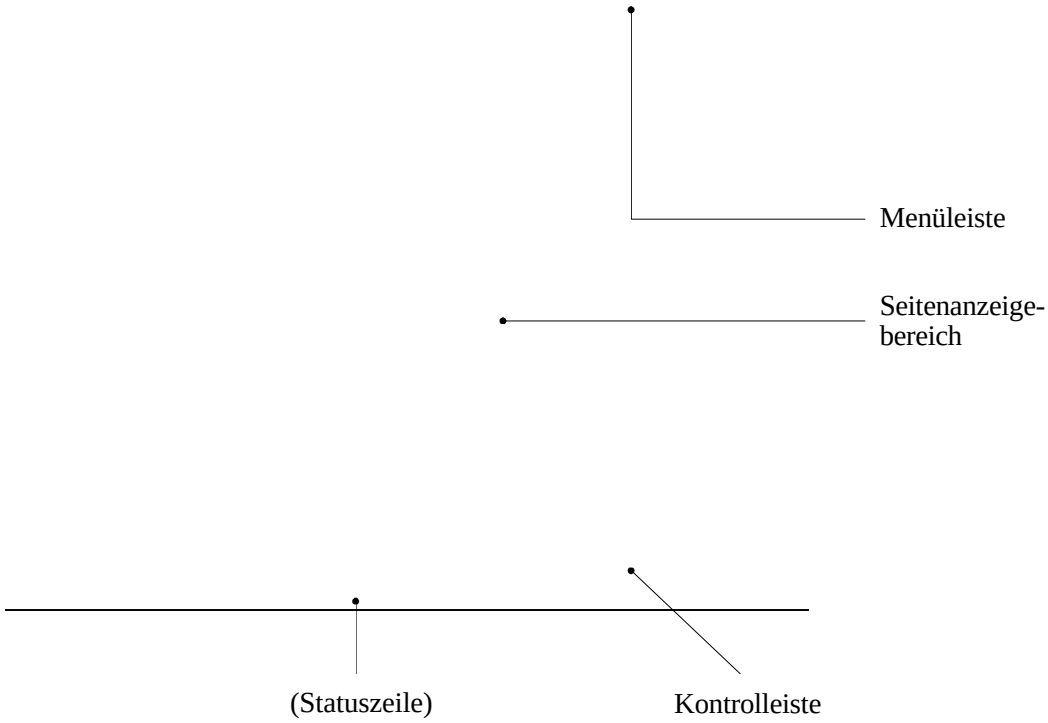


Abb. 5.1: Das Hauptfenster der *animierten Präsentation*

5.2 Aufbau

Die Präsentation besteht aus verschiedenen Fenstern: einem Hauptfenster, welches die Erklärungen und Animationen zur semantischen Analyse auf mehreren Seiten enthält, und fünf Hilfsmittelfenstern (die dynamische Erweiterung ausgeschlossen), die es den BenutzerInnen erlauben, auf bestimmte Informationen permanent zuzugreifen. In diesem Abschnitt werden wir den **Aufbau** der *animierten Präsentation* erläutern. Diese Darstellung hat Handbuchcharakter und kann somit auch als Nachschlagewerk für die Bedienung dienen.

5.2.1 Benutzeroberfläche und deren Eigenschaften

Die Benutzeroberfläche des Hauptfensters ist in vier Elemente eingeteilt: die Menüleiste, den Seitenanzeigebereich, eine Kontrolleiste und eine Statuszeile, die jedoch nicht standardmäßig angezeigt wird (siehe Abb. 5.1).

Das Hauptfenster hat eine fixe Größe von 21,15×15,9 cm, was etwas kleiner als die Oberfläche eines 15″ Monitors mit einer Auflösung von 800×600 Pixeln ist. Damit ist das Programm auf Systemen mit geringerer Auflösung nicht darstellbar. Wir mußten diese Mindestanforderungen an das System stellen, weil Animationen und Visualisierungen auf noch kleinerem Raum nicht vernünftig zu realisieren sind. Kommerzielle Softwareanbieter, die MTB 3.0 nutzen, gehen diesen Weg meist ebenfalls.

5.2.1.1 Menüleiste

Die **Menüleiste** enthält mehrere Unterpunkte, die für die Programmführung zuständig sind. Bei den Menünamen haben wir uns aus Konsistenzgründen (siehe Punkt (1) der Richtlinien

für die Oberflächengestaltung aus Abschnitt 4.3) an den Windows-Standard gehalten. Eine Ausnahme bildet das *Seite*-Menü, welches für ToolBook-Applikationen typisch ist.

Datei-Menü	
Beenden:	Beendet die Präsentation und schließt gegebenenfalls die Online-Hilfe.

Ansicht-Menü	
Statuszeile:	Blendet die Statuszeile ein oder aus. Ausgeblendet ist die Voreinstellung.

Optionen-Menü	
KFG:	Zeigt die *KFG*-Dialogbox an, mit welcher sich die BenutzerInnen die kontextfreien Grammatiken der Programmiersprachen PASCAL und LAMA ansehen können.
Algorithmen:	Zeigt die *Algorithmen und Regeln*-Dialogbox an, in welcher sich alle vorgestellten Algorithmen bzw. Regeln zu jeder Zeit ansehen lassen.
Geschwindigkeit:	Zeigt die *Geschwindigkeit einstellen*-Dialogbox an, in der sich die Animationsgeschwindigkeit verändern läßt.

Seite-Menü	
Nächste:	Zeigt die nächste Seite an.
Vorhergehende:	Zeigt die vorhergehende Seite an.
Zurück:	Springt von den Übersichten, Beispielen und Animationen wieder auf die Seite zurück, von wo aus man diese Elemente aufgerufen hat.
Inhalt:	Zeigt das Gesamtinhaltsverzeichnis an. (Dies ist für eine zukünftige größere Umgebung gedacht.)
Kapitel:	Zeigt das Kapitelverzeichnis an, in dem man sich gerade befindet.

Hilfe-Menü	
Inhalt:	Ruft die Online-Hilfe mit ihrem Inhaltsverzeichnis auf.
Hilfethema suchen:	Zeigt die *Hilfethema suchen*-Dialogbox, in welcher man mittels Schlüsselwort das gewünschte Thema aufrufen kann.
Hilfe zur Theorie:	Ruft die Online-Hilfe zur Compilerbau-Theorie auf.
Bedienelemente:	Ruft die Online-Hilfe zur Erklärung der Bedien- und Steuerungselemente auf.
Info:	Zeigt die Versionsnummer der *animierten Präsentation* an.

5.2.1.2 Seitenanzeigebereich

Im **Seitenanzeigebereich** spielt sich das Präsentations- und Animationsgeschehen ab. Dort werden alle Lerninhalte interaktiv erklärt, d.h. der/die BenutzerIn kann sich Beispiele, kleine Animationen, etc. anzeigen lassen. Alle anklickbaren Schaltflächen, Textelemente (sogenannte „Hotwords“) usw. sind blau eingefärbt bzw. besonders gekennzeichnet. Bei großen Animationen wird auf eine oder mehrere gesonderte Seiten „umgeblättert“. Abbildung 5.2 zeigt die Seitenorganisation im Hauptfenster. Fast alle Seiten sind vom Inhaltsverzeichnis aus erreichbar und umgekehrt. Ist ein Lerninhalt nicht auf eine Seite beschränkt, so

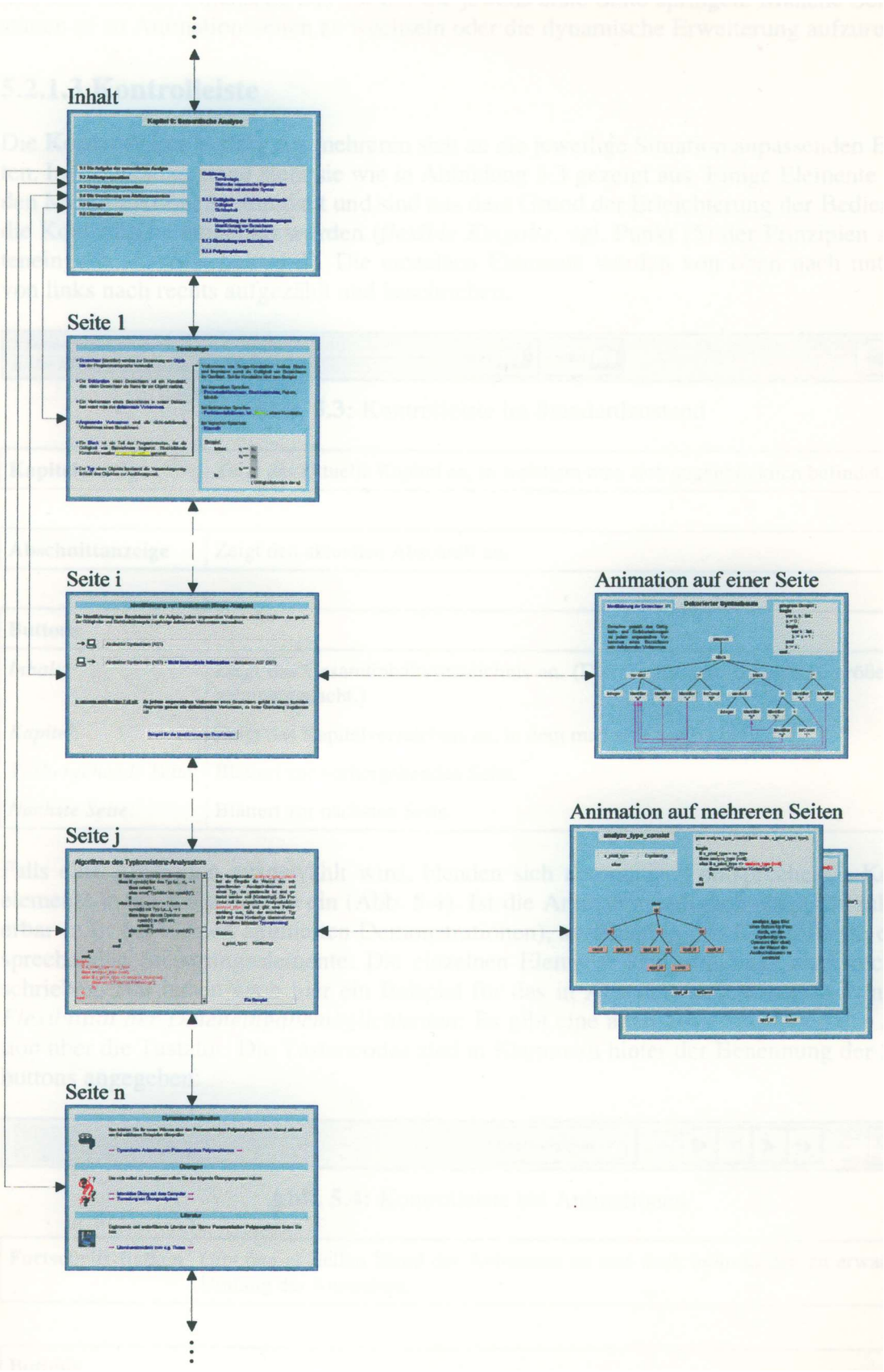


Abb. 5.2: Die Struktur der Seitenorganisation im Hauptfenster

läßt sich von der Inhaltseite aus nur auf die jeweils erste Seite springen. Manche Seiten gestatten es zu Animationsseiten zu wechseln oder die dynamische Erweiterung aufzurufen.

5.2.1.3 Kontrolleiste

Die **Kontrolleiste** besteht aus mehreren sich an die jeweilige Situation anpassenden Elementen. Im Standardzustand sieht sie wie in Abbildung 5.3 gezeigt aus. Einige Elemente sind zu den Menüfunktionen redundant und sind aus dem Grund der Erleichterung der Bedienung in die Kontrolleiste eingefügt worden (*flexible Eingabe*, vgl. Punkt (5) der Prinzipien zur Dateneingabe aus Abschnitt 4.3). Die einzelnen Elemente werden von oben nach unten und von links nach rechts aufgezählt und beschrieben:

Abb. 5.3: Kontrolleiste im Standardzustand

Kapitelanzeige	Zeigt das aktuelle Kapitel an, in welchem man sich augenblicklich befindet.

Abschnittanzeige	Zeigt den aktuellen Abschnitt an.

Buttons	
Inhalt:	Zeigt das Gesamtinhaltsverzeichnis an. (Dies ist für eine zukünftige größere Umgebung gedacht.)
Kapitel:	Zeigt das Kapitelverzeichnis an, in dem man sich gerade befindet.
Vorhergehende Seite:	Blättert zur vorhergehenden Seite.
Nächste Seite:	Blättert zur nächsten Seite.

Falls eine Animation ausgewählt wird, blenden sich automatisch entsprechende Kontrollelemente in die Kontrolleiste ein (Abb. 5.4). Ist die Animation lediglich eingeschränkt steuerbar (z.B. bei kleinen animierten Demonstrationen), so fehlen in der Kontrolleiste die entsprechenden Steuerungselemente. Die einzelnen Elemente sind von links nach rechts beschrieben. Wir haben auch hier ein Beispiel für das in Abschnitt 4.3 genannte Prinzip der *Flexibilität der Dateneingabemöglichkeiten*: Es gibt eine alternative Steuerung der Animation über die Tastatur. Die Tastencodes sind in Klammern hinter der Benennung der Steuerbuttons angegeben:

Abb. 5.4: Kontrolleiste bei Animationen

Fortschritt-Balken	Gibt den aktuellen Stand der Animation an und auch indirekt den zu erwartenden Umfang der Animation.

Buttons	
Geschwindigkeit:	Zeigt die *Geschwindigkeit einstellen*-Dialogbox an, in der sich die Animationsgeschwindigkeit verändern läßt.

Buttons	
Reset (←-*Taste*):	Die Animation wird neu initialisiert und auf einen Anfangszustand zurückgesetzt.
Stop (↓-*Taste*):	Die mit dem *Play*-Button gestartete Animation wird angehalten. Sie kann von diesem Haltepunkt beliebig mit dem *Play*- oder *Step*-Button fortgesetzt werden.
Play (↑-*Taste*):	Läßt die Animation mit Hilfe eines eingebauten einstellbaren Taktgebers laufen. Sie kann mit dem *Stop*-Button angehalten werden.
Step (→-*Taste*):	Führt immer nur einen Schritt der Animation aus. Der Knopf muß also immer wieder gedrückt werden. Mit dem *Play*-Button kann ein Taktgeber jederzeit zugeschaltet werden.
Zurück:	Springt von den Animationen wieder auf die Seite zurück, von wo aus man diese aufgerufen hat.

5.2.1.4 Statuszeile

Innerhalb der **Statuszeile** zeigt das Programm kurze Hilfetexte an. Wählt man einen Menüpunkt der Menüleiste aus oder überstreicht einen Button der Kontrolleiste, so schreibt das Programm eine entsprechende Erklärung in die Statuszeile. Sie läßt sich ein- bzw. ausblenden.

5.2.2 Hilfsmittel

Alle Hilfsmittel der *Animation der semantische Analyse* benutzen ein separates, vom Hauptfenster unabhängiges Fenster.

5.2.2.1 Online-Hilfe

Die **Online-Hilfe** der *animierten Präsentation* verwendet das integrierte Hilfesystem von Windows 3.1, um eine höchstmögliche Standardisierung und Bekanntheit im Umgang mit der Online-Hilfe zu erreichen. Es ist als Hypertext-Dokument angelegt und ermöglicht Sprünge in andere Hilfebereiche und das Suchen nach Stichwörtern. Außerdem ist die Online-Hilfe kontextsensitiv, d.h. je nachdem von welcher Stelle sie aufgerufen wird, erscheint der zu dieser Stelle passende Hilfetext (mit Bildern).

Abb. 5.5: Die oberste Ebene der Online-Hilfe und eine Verzweigungsmöglichkeit

Sie ist in drei Abschnitte unterteilt (siehe Abb. 5.5, links). Von der obersten Ebene kann man zu den folgenden Hilfethemen verzweigen:

Abb. 5.6: Das eingebaute Literaturverzeichnis

1. Vertiefung der Theorie
2. Hilfe zur dynamischen Erweiterung (ASA-Tool)
3. Hilfe zur *animierten Präsentation* (Lernprogramm)

5.2.2.2 Literaturangaben

Interessiert sich der/die BenutzerIn besonders für ein bestimmtes Thema, so unterstützt das Programm die Suche nach **Literaturangaben**. Auch hier wird das eingebaute Hilfesystem

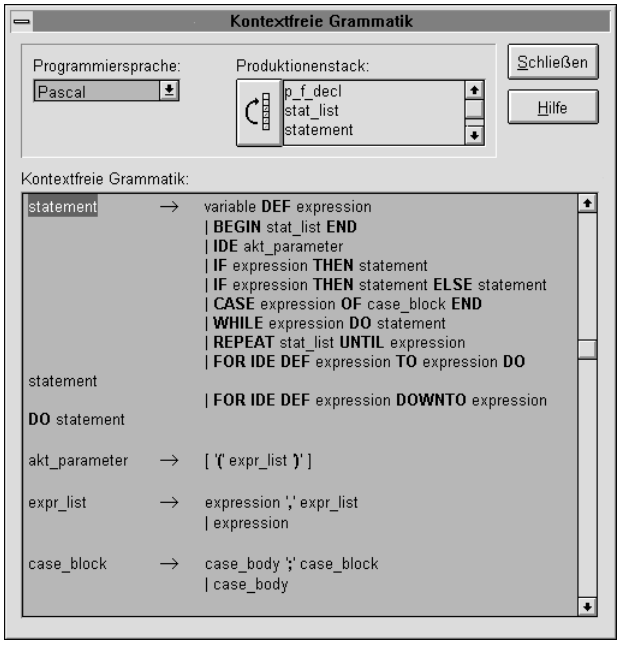


Abb. 5.7: Anzeige der verwendeten kontextfreien Grammatiken

von Windows genutzt. Abbildung 5.6 zeigt den Anfang des Literaturverzeichnisses für die semantische Analyse insgesamt.

5.2.2.3 Kontextfreie Grammatiken

Die in der *Animation der semantischen Analyse* verwendeten Beispielsprachen sind PASCAL und LAMA. Mit Hilfe der KFG-Dialogbox können die beiden dazugehörigen **kontextfreien Grammatiken** jederzeit angezeigt werden (siehe Abb. 5.7 und Anhang A).
Eine Vereinfachung ist der *Produktionenstack*, der dem/der BenutzerIn hilft, den Fluß der Produktionen zu verfolgen. Jedesmal, wenn man auf ein rechts stehendes Nichtterminal im unteren Feld doppelklickt, so wird dieses Nichtterminal im Produktionenstack oben eingetragen. Das Feld scrollt zu der entsprechenden Produktion und zeigt das angeklickte, links stehende Nichtterminal an. Nun kann man wieder ein rechts von dieser Produktion stehendes Nichtterminal anklicken und die entsprechende Produktion anzeigen lassen etc. Um in diesem Pfad wieder zurückzugehen, klickt man auf den Button links neben dem Stack. Dann wird das oberste Element auf dem Stack entfernt und im unteren Feld die vorhergehende Produktion (die, des nun obersten Nichtterminal auf dem Stack) markiert.

5.2.2.4 Animierte Algorithmen

Alle Quellcodes, der in der *Animation der semantischen Analyse* **animierten Algorithmen**, lassen sich durch die Algorithmen-Dialogbox darstellen (siehe Abb. 5.8 und Anhang B). Nach einer Auswahl wird der entsprechende Algorithmus direkt im unteren Feld angezeigt.

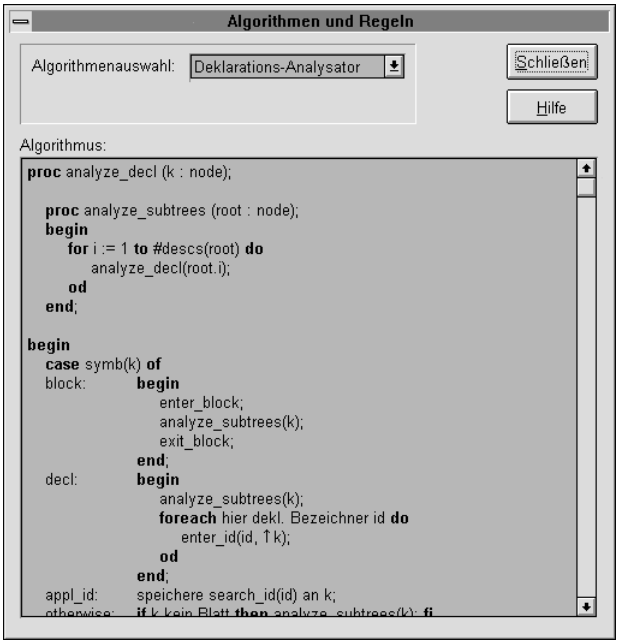


Abb. 5.8: Anzeige der Codes der animierten Algorithmen

5.2.2.5 Animationsgeschwindigkeit

Durch den Play-Button in der Kontrolleiste wird ein Windows-interner Taktgeber gestartet, der durch eine Geschwindigkeit-Dialogbox eingestellt werden kann (siehe Abb. 5.9).
Jede Animation verlangt in Abhängigkeit ihrer Komplexität eine spezifische Geschwindigkeitseinstellung. Zusätzlich muß die Aufnahmebereitschaft und -fähigkeit des/der Anwen-

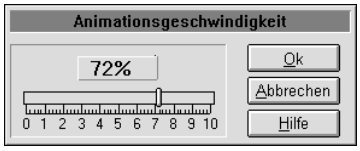


Abb. 5.9: Einstellen der Animationsgeschwindigkeit

derIn berücksichtigt werden. Daher ist es sinnvoll den AnwenderInnen zu gestatten, jede größere Animation mit einer jeweils eigenen Geschwindigkeit ablaufen zu lassen. Aus diesem Grund werden die unterschiedlichen Geschwindigkeitsniveaus mit dem Benutzernamen zu den jeweiligen Animationen abgespeichert. Diese Einstellung wird in einer Initialisierungsdatei (siehe Abschnitt 5.5.2) abgelegt.

5.3 Basisanimationen

ToolBook bietet eine Reihe von Techniken an, um Objekte zu animieren. Jede komplexe Animation ist aus vielen solcher grundlegenden Animationen aufgebaut:

- Verschieben von Objekten (Positionsveränderung).
- Ändern der Größe oder der Form eines Objekts durch die Änderung seines äußeren beschränkenden Rechtecks.
- Ändern der Größe oder der Winkel eines Objekts (z.B. einer Linie) durch Veränderung der Anzahl und Position der definierenden Winkelpunkte.
- Anzeigen und Verbergen einer Reihe von Objekten aus einem Stapel, wobei jedes Objekt eine geringfügige Abweichung desselben Bildes darstellt.
- Schnelles Durchblättern von Seiten, wobei jede Seite eine geringfügig andere Ansicht desselben Bildes enthält. (Je kleiner die Seitengröße ist, desto besser ist die Qualität der Animation)

Damit nicht jede zu realisierende Animation aus solchen Elementaranimationen konstruiert werden muß, ist es wünschenswert, eine Bibliothek von Funktionen zur Verfügung zu haben, mit deren Hilfe sich häufig verwendete Animationen mit einem Funktionsaufruf durchführen lassen. Da ToolBook eine solche Bibliothek nicht zur Verfügung stellt (außer einiger weniger Funktionen in Beispiel-Anwendungen), war es notwendig, eine solche Funktionsbibliothek in Eigenarbeit zu erstellen. Sie befindet sich im Buchskript unserer Präsentation und besteht aus folgenden Animationsroutinen:

zoomObjectCorner (obj, corner, sizeBegin, sizeEnd, steps): Zoomt in *steps* Schritten das Objekt *obj* von der Größe *sizeBegin* in die Größe *sizeEnd*, ausgehend von der Ecke *corner*. Die Ecken seien im Uhrzeigersinn von 1 bis 4 benannt.

zoomObjectSide (obj, side, sizeBegin, sizeEnd, steps): Zoomt in *steps* Schritten das Objekt *obj* von der Größe *sizeBegin* in die Größe *sizeEnd*, ausgehend von der Seite *side*. Die Seiten werden als Strings direkt übergeben („left“, „right“, „top“, „bottom“).

zoomObjectCentre (obj, sizeBegin, sizeEnd, steps): Zoomt in *steps* Schritten das Objekt *obj* von der Größe *sizeBegin* in die Größe *sizeEnd*, ausgehend vom Mittelpunkt des Objekts.

centre (pos, size): Berechnet den Mittelpunkt eines Rechtecks an der Position *pos* und der Größe *size*.

centreOfObject (obj): Berechnet den Mittelpunkt des Objekts *obj*.

rotate (obj, rotPoint, angle, direction): Dreht das Objekt *obj* in einem Schritt im Winkel von *angle* Grad um den Punkt *rotPoint*. Die Drehrichtung wird durch *direction* angegeben.

animRotate (obj, rotPoint, angle, direction, steps): Rotiert das Objekt *obj* in *steps* Schritten im Winkel von *angle* Grad um den Punkt *rotPoint*. Die Drehrichtung wird durch *direction* angegeben.

moveObject (obj, newPos, steps): Bewegt das Objekt *obj* in *steps* Schritten von seiner gegenwärtigen Position zu einer neuen Position *newPos*.

moveObjectAlongPath (obj, pathObj, fixPoint, steps): Bewegt das Objekt *obj* in *steps* Schritten entlang eines Pfades *pathObj* (Linie oder Winkellinie). *fixPoint* gibt an, welcher Punkt am Pfadobjekt laufen soll (Alle Ecken bzw. der Mittelpunkt als Angabe möglich).

expandLineAlongPath (lineObj, linePathObj, steps): Expandiert eine Linie *lineObj* in *steps* Schritten entlang einer unsichtbaren Pfadlinie *linePathObj*.

expandAngledLineAlongPath (angledLineObj, angledLinePathObj, steps): Expandiert eine Winkellinie *angledLineObj* in *steps* Schritten entlang einer unsichtbaren Pfadwinkellinie *angledLinePathObj*.

5.4 Botschaftsverarbeitung

In Kapitel 2 haben wir das Zusammenspiel von Ereignissen Botschaften und Botschaftbehandlungsroutinen innerhalb von Skripten kennengelernt. Wir haben auch gesehen, daß die Plazierung der Behandlungsroutinen bei den Objektskripten für eine gewisse Unübersichtlichkeit und damit Wartungsunfreundlichkeit sorgt. Um dieser Problematik zu entgehen, haben wir den größten Teil der Routinen, die mehrere Objekte gleichsam betreffen, im Buchskript zentralisiert. Dadurch wurde auch die Konsistenz der Funktionalität von Benutzeraktionen erhöht.

Betrachten wir als Beispiel die Betätigung eines Hotwords (anklickbares Textelement) mit der linken Maustaste. Abgesehen von der eigentlichen Funktionalität, färbt sich jedes Hotword (*konsistente Farbkodierung*, vgl. Punkt (5) der Prinzipien für den Farbgebrauch aus Abschnitt 4.6) beim Anklicken von blau auf gelb, um ein visuelles Feedback zu geben. Nachdem die Behandlungsroutine für die Botschaft *buttonDown* im Skript des Hotwords diese Botschaft erhalten hat, sendet sie eine spezielle benutzerdefinierte Botschaft für das Gelbfärben unter Angabe einer eindeutigen Identifikationsnummer (Absender) an eine entsprechende Behandlungsroutine im Buchskript. Dabei ist es sinnvoll, die Botschaft direkt an das Buch zu senden, und nicht in der Objekthierarchie aufsteigen zu lassen, um einen Geschwindigkeitsvorteil zu erzielen. Diese Behandlungsroutine sorgt nun für die Umfärbung des angeklickten Hotwords.

5.5 Realisierung der Algorithmenanimationen

Die Vorstellung der Algorithmen findet in zwei Ebenen statt (siehe Abb. 5.10). Auf der ersten Ebene wird der Algorithmus als Ganzes erklärt. Man kann den Algorithmus schrittweise durchgehen. Dabei unterteilt das Programm den Algorithmus in logische Einheiten, also

z.B. die einzelnen Fälle einer *case*-Kontrollstruktur. Die aktuelle Einheit wird rot markiert und in einem Feld rechts vom Algorithmus erklärt. Nun kann man für jede Algorithmeneinheit in die zweite Ebene wechseln, in der diese Einheit zeilenweise, anhand eines meist alle Einheiten überspannenden Beispiels, animiert wird. Die Einheit wird hier ebenfalls in einem Feld dargestellt und eine aktuelle Zeile erscheint während der Animation auch in roter Farbe. Im Anhang C.1 ist anhand der Abbildungen C.1.6 bis C.1.11 ein Animationsausschnitt angegeben.

Abb. 5.10: Die zwei Ebenen der Algorithmenanimationen

Diese Unterteilung ist dahingehend motiviert, daß die Anwendung für Auflösungen von 800×600 Bildpunkten (15″ Monitore) konzipiert wurde. Das Hauptfenster der *animierten Präsentation* füllt diesen Raum fast vollständig aus. Eine andere Möglichkeit für die Lösung des Platzproblems wäre es, die Animationen in einem separaten Fenster abzuspielen. Allerdings müßte der/die AnwenderIn die Fenster dann immer abwechselnd in den Vordergrund bringen. Beide Lösungen widersprechen dem Prinzip der *Minimierung von Merkanforderungen* (siehe Punkt (3) der Richtlinien für die Oberflächengestaltung aus Abschnitt 4.3).

5.5.1 Möglichkeiten der Interaktion

Sobald man sich auf einer Seite befindet, die eine Animation enthält, hat man zwei Möglichkeiten, Einfluß auf die Animation zu nehmen. Durch die

- **Steuerelemente in der Kontrolleiste** (siehe Abschnitt 5.2.1.3) und die
- **Geschwindigkeit-Dialogbox** (siehe Abschnitt 5.2.2.5).

Für jede Animation kann die Geschwindigkeit separat eingestellt werden. Die *animierte Präsentation* speichert diese Geschwindigkeiten zusammen mit dem Anwendernamen in einer sogenannten INI-Datei ab, siehe nächsten Abschnitt.

5.5.2 Initialisierung

Um die Initialisierung der Animationen und den Namen der BenutzerInnen abzuspeichern, verwendet das Programm die von Windows bereitgestellten Konzepte zur Initialisierung von Applikationen über INI-Dateien. Das Windows-API stellt für die Erzeugung und Verwaltung von INI-Dateien mehrere Funktionen zur Verfügung, die nur noch vom Programm aufzurufen sind. Gleichzeitig hat die Verwendung dieser Funktionen den Vorteil, daß der Zugriff auf die Dateien über einen Cache erfolgt und damit die Zugriffsgeschwindigkeit sehr

hoch ist. Wir wollen an dieser Stelle einen möglichen Inhalt der INI-Datei (mit Namen COBAU9_1.INI) angeben:

```
[USERNAME]
Name=Andreas

[BOOKMARKS]
Andreas=16
Beatrix=10

[SPEED]
Andreas_Animation3=0.8
Andreas_Animation13=0.27
Beatrix_Animation3=0.79
Andreas_Animation29=0.5
```

Der Abschnitt [USERNAME] enthält den Namen des zuletzt aktuellen Benutzers. Er wird beim Start der *animierten Präsentation* abgefragt bzw. als möglicher Name vorgeschlagen.

[BOOKMARKS] beinhaltet erstens alle Namen der Benutzer, die jemals das Programms gestartet haben, und zweitens zu jedem dieser Namen die Seitenzahl der zuletzt besuchten Seite. Gibt ein/eine BenutzerIn zu Beginn der Sitzung einen dem Programm bekannten Namen an, so fragt das Programm, ob man auf dieser Seite die Sitzung fortführen möchte.

Für Animationen ist der nun folgende Abschnitt [SPEED] von Bedeutung. Jeder Animation ist eine voreingestellte relative Geschwindigkeit von 0.5 (50%) zugeordnet. Benutzerdefinierte Veränderungen der Geschwindigkeit werden in diesem Abschnitt aufgeführt. Der erste Eintrag besagt, daß der Benutzer *Andreas* die Geschwindigkeit der dritten Animation von 50% auf 80% erhöht hat.

5.5.3 Animationssteuerung und deren Implementierung

In diesem Abschnitt möchten wir etwas näher auf die Implementierung der Animationssteuerung und den Zusammenhängen zwischen den einzelnen Komponenten eingehen. Das Ziel bei der Entwicklung einer solchen Steuerung war es, ein Verfahren zu finden, welches für alle Animationen Anwendung findet. Es soll möglichst vielseitig, wartungsfreundlich und beliebig erweiterbar sein.

Unsere Realisierung ist in Abbildung 5.11 konzeptionell dargestellt. Die dunkelgrau hinterlegten Bereiche beschreiben Skripte von Objekten und die hellgrauen Bereiche entweder globale Variablen (sogenannte Systemvariablen) oder Dateien. Die Kästchen innerhalb der Skriptenbereiche symbolisieren Behandlungsroutinen. Unter den eingezeichneten Kanten, welche die Behandlungsroutinen verbinden, hat man sich das Senden von Botschaften vorzustellen. Das dick umrandete Kästchen ist der Einstiegspunkt.

Wechselt der/die AnwenderIn zu einer Animationsseite, dann sendet das ToolBook-System eine *enterPage*-Botschaft, die in einer entsprechenden Behandlungsroutine im Seitenskript abgefangen wird. Von dort aus wird eine Botschaft zu der Behandlungsroutine *Init* im selben Skript geschickt. Dies ist der Einstiegspunkt in der obigen Konzeptbeschreibung. *Init* besetzt nun das Feldelement *svMax*[*pageNumber*] mit der Anzahl der Animationsschritte für diese Animation und initialisiert *svAkt*[*pageNumber*] mit 0. Desweiteren werden über verschiedene Botschaftssendungen u.a. die Kontrollknöpfe in der Kontrolleiste initialisiert und die Systemvariable für die Geschwindigkeit *svSpeed*, entweder mit der Standardvorgabe 0.5, oder mit einem in der INI-Datei gespeicherten Wert besetzt. Verändert man die Ge-

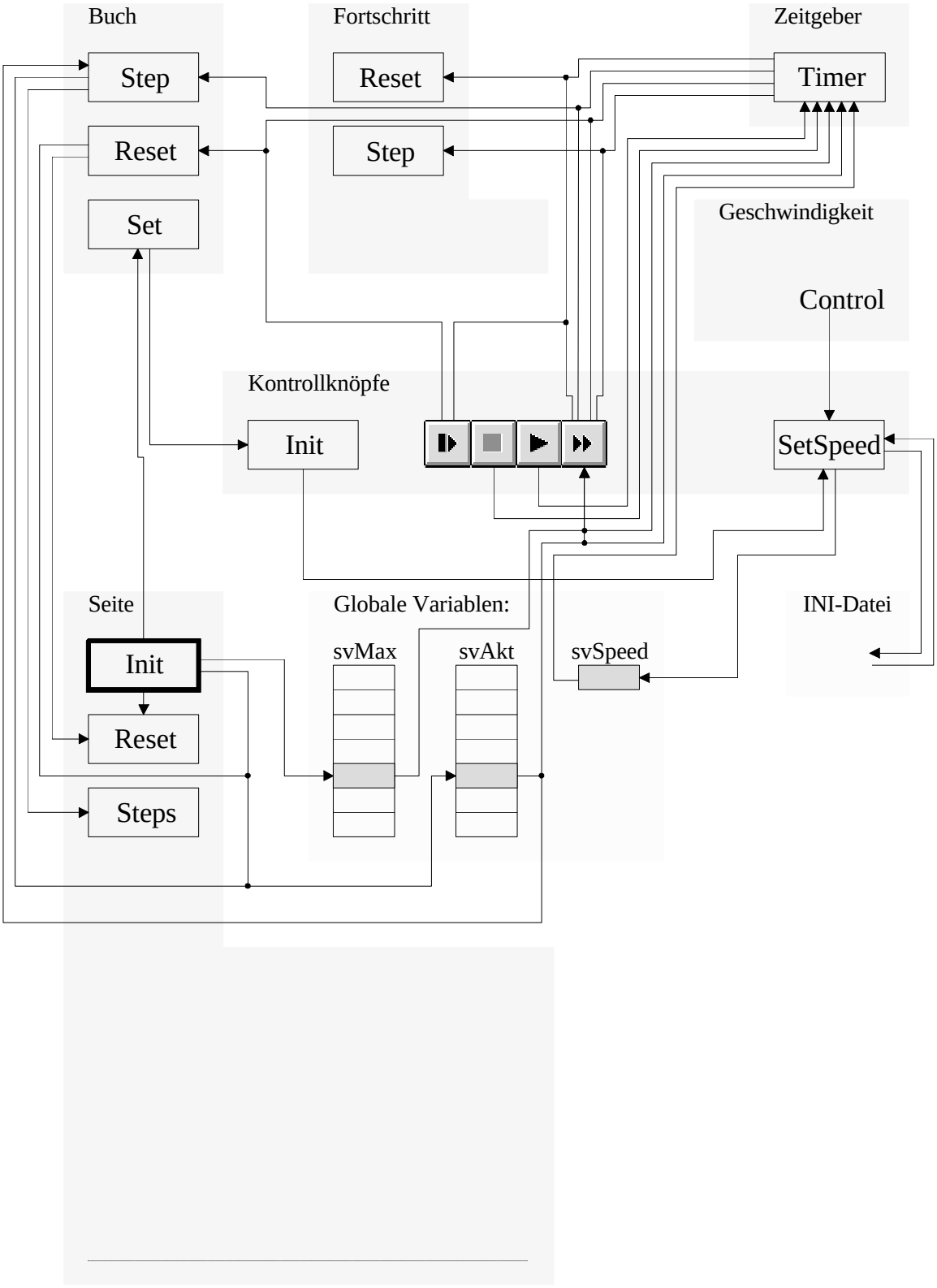


Abb. 5.11: Aufbau der Animationssteuerung

schwindigkeit, dann schreibt das Programm den neuen Wert in die INI-Datei und besetzt *svSpeed* neu. Ferner setzt *Init* die Animation zurück.

Durch die Betätigung des *Step*-Button liest die entsprechende Behandlungsroutine für das Klicken auf diesen Knopf die Systemvariablen *svMax*[*pageNumber*] und *svAkt*[*page-Number*] aus. Sind beide Einträge identisch, so wird die Fortschrittsleiste und die Animation zurückgesetzt. *svAkt*[*pageNumber*] erhält den Wert 0. Sind sie verschieden, dann macht die Fortschrittsleiste und die Animation einen Schritt. *svAkt*[*pageNumber*] wird um 1 inkrementiert.

Wenn der/die AnwenderIn auf den *Play*-Button drückt, dann sendet die betroffene Behandlungsroutine eine Botschaft zum *Timer*. Dieser aktiviert die gleiche Botschaftskaskade wie die Behandlungsroutine des *Step*-Buttons, bis auf das Auslesen von *svSpeed* und der Anforderung eines Windows-Timers (Abschnitt 1.2.3). Bei jedem empfangenen Timertick wird sozusagen ein Klicken auf den *Step*-Button simuliert. Der Wert von *svSpeed* bestimmt dabei die Länge des Zeitintervalls zwischen zwei Ticks. Die nun ablaufende Animation kann jederzeit durch die Betätigung des *Stop*-Buttons gestoppt werden. Dabei wird der *Timer* deaktiviert.

Um die Animation und damit auch die Fortschrittsleiste während des Ablaufs (0 < *svAkt*[*pageNumber*] < *svMax*[*pageNumber*]) zurückzusetzen, ist der *Reset*-Button zu betätigen. Damit stellt das Programm den Anfangszustand der Animation wieder her.

Das in Abbildung 5.11 gezeigte Konzept ist nicht das einzig denkbare. Die verschiedenen Behandlungsroutinen können in beliebigen Skripten untergebracht werden. Möchte man eine Animationsseite hinzufügen, so ist lediglich eine neues Seitenskript notwendig und im Extremfall eine Erweiterung der Arrays für *svAkt* und *svMax*, was aber auch automatisch erfolgen kann.

5.6 Aufruf der dynamischen Erweiterung

Jeweils am Ende jedes Themas, das in der *animierten Präsentation* vorgestellt wird, hat der/die BenutzerIn die Möglichkeit, ein Fenster der dynamischen Erweiterung ASA zu öffnen, um sich dieses Thema anhand von Beispielen näher zu verdeutlichen. Die Abbildung 5.12 zeigt den Aufruf des ASA-Tools aus einer Seite der *animierten Präsentation*.

Bevor das ASA-Tool endgültig startet, erscheint vorher eine systemmodale Dialogbox, die das ausgewählte Thema (von vier möglichen) anzeigt und es so dem/der AnwenderIn ermöglicht, die Auswahl zu ändern oder aber auch zurückzunehmen. Welche Themen von ASA behandelt werden können ist im nächsten Kapitel 6 beschrieben.

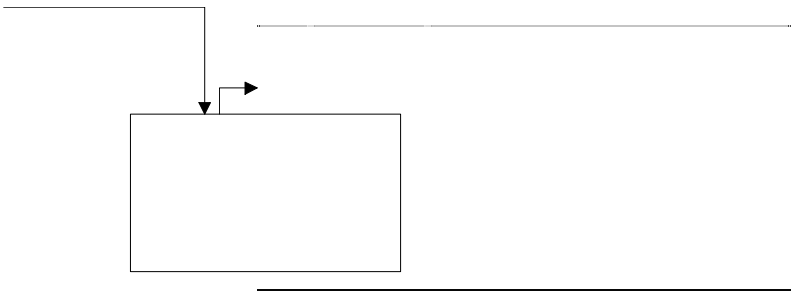


Abb. 5.12: Der Aufruf der dynamischen Erweiterung ASA (rechts)

5.6.1 Grundkonzept der Kommunikation

Wir verwenden zum Aufruf der dynamischen Erweiterung und zur weiteren Kommunikation nicht DDE, sondern definieren ein eigenes Kommunikationsprotokoll. Dieses Protokoll beruht auf der Möglichkeit, benutzerdefinierte Botschaften zu versenden. Der Vorteil eines eigenen Protokolls liegt in der Einfachheit der Implementierung, ohne unnötigen Überbau.

Windows definiert vier Bereiche von Botschaftsnummern. Dabei entspricht die Botschaft WM_USER dem Wert 0×0400 und dient der Trennung zwischen Botschaften, die ausschließlich von Windows benutzt werden dürfen und solchen, die auch Windows-Applikationen verwenden können:

Reichweite	**Bedeutung**
0 bis WM_USER – 1	Reserviert von Windows.
WM_USER bis 0×7FFF	Integer-Botschaften für die Verwendung innerhalb einer privaten Fensterklasse einer Applikation.
0×8000 bis 0×BFFF	Reserviert von Windows für den zukünftigen Gebrauch.
0×C000 bis 0×FFFF	String-Botschaften für den Gebrauch zur Kommunikation zwischen Applikationen.

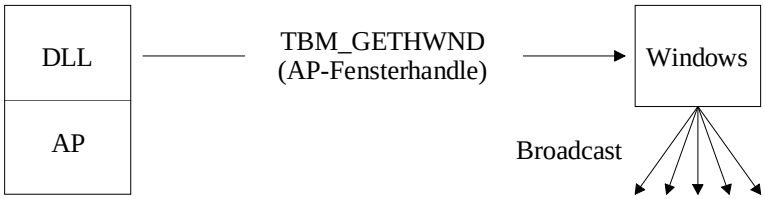
Durch einen Aufruf der Window-Funktion *RegisterWindowMessage* kann man sich vom System einen benutzerdefinierte Botschaft erzeugen lassen, die im vierten Bereich (0×C000 bis 0×FFFF) unserer Unterteilung liegt. Mit dieser generierten Botschaft hat man die Gewißheit, daß keine anderen Applikationen unabsichtlich dieselbe Botschaftsnummer verwenden.

5.6.2 Implementierung

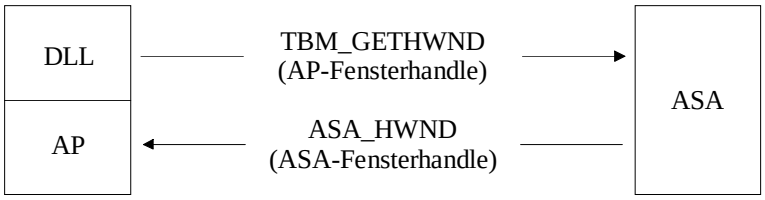
Die Kommunikationsfunktionen der *animierten Präsentation* (AP) sind in eine DLL mit Namen CONNECT.DLL ausgelagert worden. Von dort aus hat man die Möglichkeit, direkt die C-Schnittstelle des Windows-API anzusprechen. Diese DLL wird beim Startvorgang der *animierten Präsentation* in den Speicher geladen und die Funktionen im Buchskript bekannt gemacht.

Um die Kommunikation zu initialisieren, muß man unterscheiden welches Programm zuerst gestartet wurde. Beide Programme sind voneinander unabhängig lauffähig. Es ist nicht möglich, die *animierte Präsentation* vom ASA-Tool aus aufzurufen. Wir nehmen desweiteren an, daß alle benutzerdefinierten Botschaften dem System schon durch Aufrufe von *RegisterWindowMessage* bekannt gemacht wurden.

Beim Start der AP sendet eine Funktion der DLL einen globalen Broadcast mit der Botschaft TBM_GETHWND (TBM = „ToolBook Message“) an alle im Speicher befindlichen Programme. Dies ist notwendig, um den Fensterhandle des ASA-Tools zu erlangen, welcher der AP unbekannt ist:

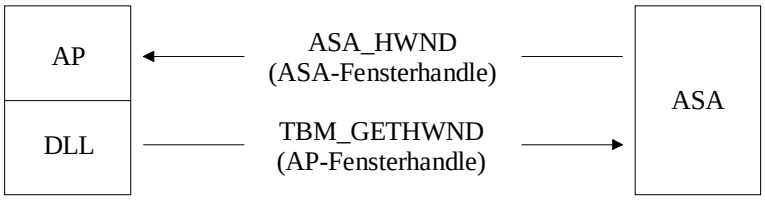


Falls sich das ASA-Tool noch nicht im Speicher befindet passiert nichts. Wenn doch, dann sendet es seinen Fensterhandle an die AP zurück, dessen Fensterhandle mit der Botschaft TBM_GETHWND als Attribut mitgesandt wurde. Die AP übersetzt die empfangene Botschaft ASA_HWND mit der ToolBook-Funktion *translateWindowMessage* in eine ToolBook-Botschaft und speichert den Fensterhandle des ASA-Tools in einer globalen Variablen ab:

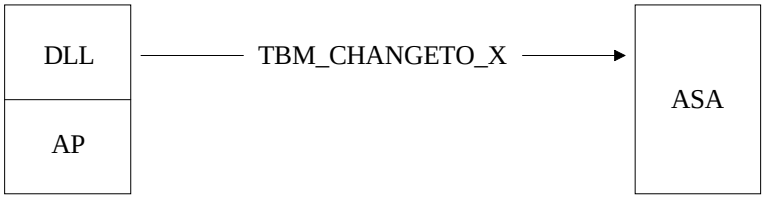


Wenn andererseits ASA manuell (über den Programm- oder Dateimanager) gestartet wird, weil es als eigenständiges Programm konzipiert wurde, dann muß es zuerst einmal feststellen, ob sich die AP im Hauptspeicher befindet. Dazu sendet ASA ebenfalls einen globalen Broadcast mit seinem Fensterhandle, allerdings mit der Botschaftsnummer ASA_HWND:

Wenn sich die AP nicht im Speicher befindet passiert nichts. Wenn doch, dann sendet die AP die Botschaft TBM_GETHWND mit ihrem Fensterhandle an ASA zurück (ASA sendet nun natürlich wieder die redundante Botschaft ASA_HWND, was durch den manuellen Aufruf von ASA bedingt ist):



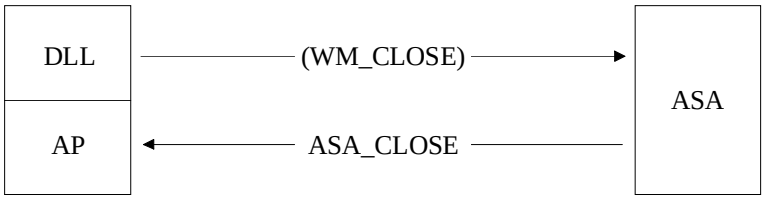
Ab hier nehmen wir an, daß beide Programme ihre Fensterhandles kennen und sich beide im Speicher befinden. Der/die BenutzerIn wählt nun auf einer Seite der *animierten Präsentation* das Hotword zur Behandlung der aktuellen Themas in der dynamischen Erweiterung ASA. (Falls sich diese nicht schon im Speicher befindet, wird sie automatisch gestartet.) In Abhängigkeit von den vier möglichen Themen gibt es nun auch vier verschiedene Botschaftstypen die versendet werden können. Das sind TBM_CHANGETO_CAS für den Vergleich der konkreten zur abstrakten Syntax, TBM_CHANGETO_CC für die Überprüfung der Kontextbedingungen, TBM_CHANGETO_OL für die Auflösung der Überladung und TBM_CHANGETO_PM für die Typinferenz. Alle diese möglichen Botschaften seien durch die generische Botschaftsbezeichnung TBM_CHANGETO_X abgekürzt:



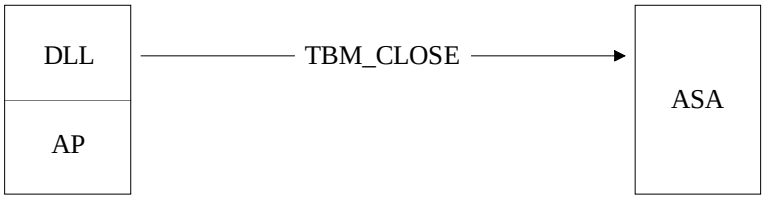
Das ASA-Tool reagiert daraufhin mit dem Einblenden der Dialogbox zur Themenauswahl mit dem aktuellen Thema als Vorauswahl. Akzeptiert der/die AnwenderIn das Thema, so

startet ASA mit dem aktuellen Thema bzw. wechselt von dem letzten zum aktuellen Thema.

Um das ASA-Tool zu beenden, wählt man entweder in der Menüleiste von ASA direkt die Beenden-Option oder drückt auf der entsprechenden Seite in der *animierten Präsentation* das nun gelb eingefärbte Hotword ein zweites Mal. Im zweiten Fall sendet AP direkt die Window-Botschaft WM_CLOSE an das ASA-Tool. In beiden Fällen meldet sich ASA durch das Senden der Botschaft ASA_CLOSE ab:



Beendet der/die AnwenderIn die *animierte Präsentation*, ohne vorher das ASA-Tool zu schließen, so meldet sie sich lediglich ab. Das ASA-Tool bleibt bis zur manuellen Schließung geöffnet:



Kapitel 6

Die dynamische Erweiterung ASA

Den zweiten Teil der *Animation der semantischen Analyse* bildet die dynamische Erweiterung **ASA** (ASA = Animationstool zur Semantischen Analyse). Das ASA-Tool stellt gegenwärtig vier unterschiedliche **Animationsthemen** zur Verfügung:

- Ein **Vergleich von konkreter und abstrakter Syntax** anhand der Eingabe eines beliebigen PASCAL-Programms.
- Die **Überprüfung der Kontextbedingungen** mit einem beliebigen PASCAL-Programm als Eingabe. Die Animation umfaßt die Visualisierung des Deklarations-Analysators sowie des Typkonsistenz-Analysators.
- Die **Auflösung der Überladung von Bezeichnern**. Als Eingabe erwartet ASA eine spezielle Spezifikation, welche die sichtbaren, evtl. überladenen Operatoren, einen Kontexttyp und einen Ausdruck umfaßt.
- Die **Typinferenz** bei einer polymorph getypten Programmiersprache. Für diese Animation ist ein LAMA-Programm anzugeben.

Jedes Thema wird an einem visualisierten Syntaxbaum animiert, welcher aus der Eingabe generiert und dessen Layout automatisch berechnet wird. Das Layout ist interaktiv und in (weicher) Echtzeit veränderbar. Ein zu großer Syntaxbaum läßt sich beispielsweise verkleinern oder stauchen.

Wir wollen zunächst kurz auf den Entscheidungsfindungsprozeß eingehen, der zur Zweiteilung des Programms *Animation der semantischen Analyse* führte.

6.1 Gründe für die Programmteilung

Die Trennung erfolgte, wie schon im letzten Kapitel beschrieben, aus Gründen der Effizienz, da dynamische, aufgrund beliebiger Benutzereingaben nicht vorherbestimmbare Visualisierungen und Animationen sehr rechenintensiv sind und hohe Anforderung an die Schnelligkeit bzw. Leistungsfähigkeit der Graphikausgaben stellen. Die in Abschnitt 2.6 angegebenen Restriktionen des Autorensystems MTB 3.0 waren diesbezüglich zu groß.

Grundsätzlich ist die in ToolBook eingebaute Sprache OPENSCRIPT nicht dafür geeignet, komplexe Algorithmen oder Datenstrukturen zu implementieren. Sie kennt weder Zeigerarithmetik noch Strukturen oder Unions. Außerdem ist die Durchführung einer Behandlungsroutine in OPENSCRIPT viel zu langsam, da ein Skript nicht kompiliert, sondern lediglich interpretiert wird. Deshalb mußten alle komplexeren Algorithmen, die über die Steue-

rung der graphischen Benutzeroberfläche hinausgehen, von vornherein in einer anderen Programmiersprache (in C) (z.B. als DLL's) implementiert werden. Dazu gehören die zu animierenden Algorithmen, Scanner, Parser, Aufbau der Syntaxbaumstruktur, Baumlayout, etc.

Es gab nun mehrere **Alternativen**, um der *animierten Präsentation* eine dynamische Komponente zu geben:

1. Ausschließliche Verwendung von ToolBook (Berechnungen und Algorithmen werden in C implementiert).
2. Verwendung von ToolBook, erweitert um ein unbewegliches Animationsfenster (externes Windows-Programm), das in dieselbe ToolBook-Instanz eingebaut ist.
3. Verwendung von ToolBook, erweitert um ein bewegliches Animationsfenster (externes Windows-Programm), das in dieselbe ToolBook-Anwendung eingebaut ist.
4. Verwendung von ToolBook und eines eigenständigen Windows-Animationsprogramms (in C), das alleine und auch von der in ToolBook implementierten *animierten Präsentation* aus aufgerufen werden kann.
5. Wie (4), aber das Animationsprogramm nutzt das MDI-Interface von Windows 3.1.

Alternative (1) hatten wir wegen den Restriktionen von ToolBook bereits oben abgelehnt. Eine gute Lösung wäre (2) gewesen. Diese Alternative ermöglicht eine konsistente Oberfläche, schnellen Bildaufbau im Animationsfenster und einer Animationssteuerung durch die ToolBook-Anwendung. Ein Nachteil ist der große Kommunikationsaufwand zwischen beiden Programmen, besonders bei der Steuerung der Animationen. Außerdem ist ein in der Größe festgelegtes Animationsfenster zu inflexibel. Eine Testimplementierung ergab zusätzlich große Probleme beim „Einhängen" des Fensters in die Oberfläche der ToolBook-Anwendung. (2) wurde aus diesen Gründen ausgeschlossen.

Alternative (3) läßt sich zu (2) reduzieren. Sie hat noch größere Nachteile bzgl. der Kommunikation. Ferner wären Animationsfenster und Steuerungskontrollen für die Animation voneinander getrennt. Die Alternative (3) wurde deshalb ebenso verworfen.

Überträgt man die gesamte Animationssteuerung aus der *animierten Präsentation* an das Animationsfenster, so ergibt sich damit ein interaktives Windows-Programm (4). Das ASA-Tool basiert auf diesem Konzept. Der Vorteil liegt in den Freiheiten (z.B. „unbegrenzte" Fenstergröße), die das Windows-System bietet. Die Kontrollen befinden sich in unmittelbarer Nähe zu den Animationen, wobei die Kommunikation zwischen den Kontrollen und den Animationen lediglich innerhalb von ASA abläuft. Nachteile sind der nach außen erscheinende konzeptionelle Bruch (Widerspruch zu Punkt (1) und (2) der Richtlinien zur Oberflächengestaltung aus Abschnitt 4.3) und die aufwendige Programmierung, besonders bei der Gestaltung der graphischen Oberflächen. ASA bietet aus diesem Grund auch nur einfachere Animationen, ohne komplexe Bewegungen an. Wir haben das ASA-Tool mithilfe des Windows-Compilers *MS Visual C++* übersetzt.

Schließlich bietet (5) die Möglichkeit, in mehreren Child-Fenstern Animationen von verschiedenen Eingaben zu zeigen. Der/die BenutzerIn könnte dann mehrere Animationsinstanzen miteinander vergleichen. (5) wäre eine geeignete Methode für eine verbesserte zukünftige Version des ASA-Tools.

6.2 Starten des Programms

Gestartet wird das ASA-Tool entweder, wie üblich, durch Anklicken der entsprechenden Ikone über den Programm- bzw. Dateimanager, oder über die *animierte Präsentation*. Bei jedem Programmstart erscheint eine systemmodale Dialogbox, in der das gewünschte Ani-

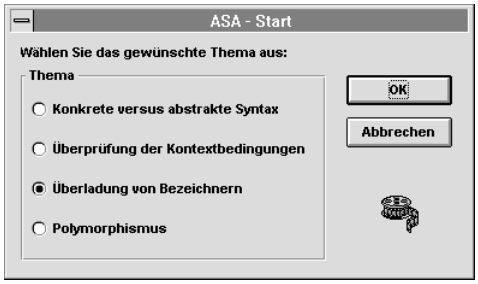


Abb. 6.1: Die Start-Dialogbox von ASA

mationsthema anzugeben ist (vgl. Abbildung 6.1).

Dieser Dialog kann jederzeit wieder aus dem laufenden Programm aufgerufen werden, um ein anderes Animationsthema auszuwählen. Ein Vorteil dieser Art der Auswahl ist die leichte Erweiterbarkeit von ASA mit neuen Animationen. Statt den vier automatischen Radiokontrollen ist es sinnvoll, in diesem Fall ein Listenfenster für die Auswahl einzusetzen.

6.3 Aufbau

ASA setzt sich aus einem Hauptfenster für die Visualisierungen, einem Editor für die Eingabe und diversen Dialogboxen zusammen. Die in diesem Abschnitt gegebenen Erläuterungen über den **Aufbau** des ASA-Tools haben Handbuchcharakter und können somit auch als Nachschlagewerk für die Bedienung dienen.

6.3.1 Benutzeroberfläche und deren Eigenschaften

Das Hauptfenster des ASA-Tools ist in vier Bereiche untergliedert: die Menüleiste, die

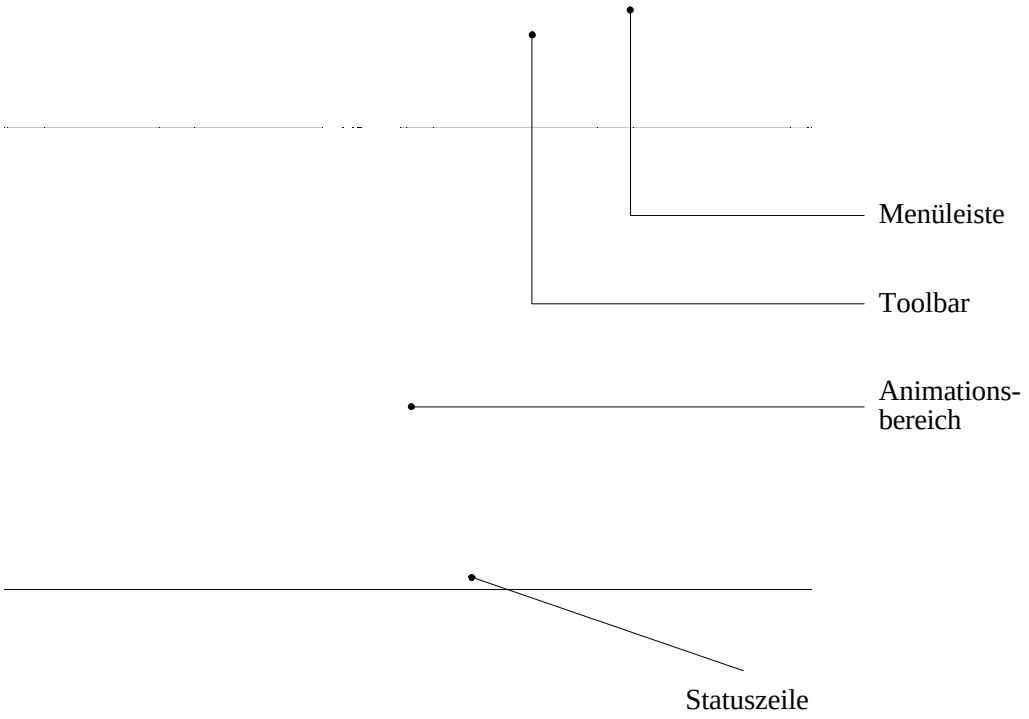


Abb. 6.2: Das Hauptfenster des ASA-Tools

Toolbar, den Animationsbereich und eine Statuszeile (vgl. Abb. 6.2). Falls der Animationsbereich maximiert werden soll, können Toolbar und Statuszeile ausgeblendet werden.

6.3.1.1 Menüleiste und Toolbar

Die **Menüleiste** enthält mehrere Unterpunkte, die teilweise, je nach Zustand des Programms und gewähltem Thema, gesperrt sein können:

Datei-Menü	
Start:	Öffnet die *Start*-Dialogbox zur Auswahl der Animationsthemen.
Neu:	Schließt die aktuelle Animation bzw. Visualisierung und löscht alle zu dieser gehörigen Einträge, sowohl im Editor (Eingabemaske bei dem Thema zur *Auflösung der Überladung*), als auch im Animationsbereich.
Öffnen:	Zeigt die *Quelldatei öffnen*-Dialogbox, aus dieser man sich eine Eingabedatei (*.PPT, *.LPT, *.OLS) auswählen kann.
Speichern:	Speichert die aktuelle Animationseingabe, falls die verändert wurde.
Speichern unter:	Zeigt die *Quelldatei speichern*-Dialogbox, mit deren Hilfe sich eine neue Eingabedatei, oder eine vorhandene sich unter neuen Namen speichern läßt.
Graphik kopieren:	Kopiert den Inhalt des mittels Mausklick ausgewählten sichtbaren Animationsbereichs in die Zwischenablage. Eine eingeblendete Dialogbox bietet den Start von *MS Paintbrush* zur eventuellen graphischen Weiterverarbeitung oder zum Ausdrucken an.
Beenden:	Beendet das ASA-Tool und gibt den Fokus an die *animierte Präsentation* zurück, falls dieses gestartet wurde.

Ansicht-Menü	
Toolbar:	Blendet die Toolbar (Schalterleiste) ein oder aus.
Statusbar:	Blendet die Statuszeile ein oder aus.
Animationsfenster:	Öffnet eine Dialogbox, mit deren Hilfe sich in Abhängigkeit von dem aktuellen Animationsthema zusätzliche Fenster einblenden (z.B. ein Fenster, das die initiale Typumgebung für einen polymorph getypten LAMA-Ausdruck anzeigt) oder sich die Knoten des Syntaxbaumes mit Attributen versehen lassen.
Hintergrundfarbe:	Ändert die Hintergrundfarbe im Animationsbereich. Dazu wird eine *Farbwahl*-Dialogbox eingeblendet, mit der sich auch Farben beliebig mischen lassen, solange die Graphikkarte (bzw. deren Treibersoftware) dieses zuläßt. Diese Funktion ist besonders sinnvoll, wenn man sich die Inhalte des Animationsbereichs in die Zwischenablage kopieren will, um sie eventuell auszudrucken. In diesem Fall ist ein weißer Hintergrund am sinnvollsten.

Eingabe-Menü	
Editor:	Blendet den Editor ein oder aus. Es steht bei allen Animationsthemen zur Verfügung, außer bei der *Auflösung der Überladung.*
Maske:	Zeigt die Dialogbox für die *Eingabespezifikation* an (siehe Abschnitt 6.2.2.3), in der man die Eingabe (Ausdruck, Kontexttyp, Operatoren) in einfacher Form festlegen kann. Sie ist nur bei der *Auflösung der Überladung* anzuwenden.

Parameter-Menü	
Skalierung:	Eröffnet ein neues Popup-Menü zur schnellen Skalierung des aktuellen Syntax- bzw. Ausdrucksbaumes. Das Skalierungsintervall reicht von 25% bis 400% in unterschiedlichen Schrittweiten.
Orientierung:	Eröffnet ein neues Popup-Menü zur Orientierung des aktuellen Baumes in alle vier Hauptachsenrichtungen.
Parameter:	Blendet die *Parameter*-Dialogbox ein, in der sich die wichtigsten Layoutparameter des Baumes einstellen lassen.

Animation-Menü	
Rücksetzen:	Setzt die Visualisierung auf den Anfangszustand zurück, also auf die bloße Darstellung des Syntax- bzw. Ausdrucksbaumes.
Endergebnis:	Führt den aktuellen Algorithmus (abhängig vom Thema) anhand des Eingabebeispiels aus und attributiert dementsprechend den Syntaxbaum.
Knoten färben:	Nur bei dem *Vergleich von konkreter und abstrakter Syntax* anwählbar. Koloriert die einzelnen Knoten des konkreten Syntaxbaums in Abhängigkeit davon, ob es sich bei ihnen um Terminale oder Nichtterminale handelt, und ob sie im gleichzeitig angezeigten abstrakten Syntaxbaum weggelassen werden können oder nicht.
Typkonsistenz prüfen:	Nur bei der *Überprüfung der Kontextbedingungen* erlaubt. Es wird ein Typkonsistenzcheck aller Ausdrucksunterbäume des aktuellen Syntaxbaums durchgeführt.

Hilfe-Menü	
Inhalt:	Ruft die Online-Hilfe mit ihrem Inhaltsverzeichnis auf.
Hilfethema suchen:	Zeigt die *Hilfethema suchen*-Dialogbox, in welcher man mittels Schlüsselwort das gewünschte Thema aufrufen kann.
Theorie:	Ruft die Online-Hilfe zur Compilerbau-Theorie auf.
Bedienelemente:	Ruft die Online-Hilfe zur Erklärung der Bedien- und Steuerungselemente auf.
Algorithmus:	Ruft die Online-Hilfe zum Anzeigen des aktuell verwendeten Algorithmus an. (Nicht verfügbar beim *Vergleich von konkreter und abstrakter Syntax*).
Info:	Zeigt die Versionsnummer des ASA-Tools an.

Die **Toolbar** ermöglicht einen schnelleren Zugriff auf die wichtigsten Funktionalitäten als eine Menüauswahl. Ihre Funktionalität bildet also eine Untermenge der Funktionalität des sich darüber befindlichen Menüs. Das Auftreten der Toolbar-Elemente ist vom gewählten Animationsthema abhängig und paßt sich dementsprechend daran an. Aufgrund der zur Menüleiste redundanten Funktionalität verzichten wir hier auf eine detaillierte Erklärung der einzelnen Elemente. Abb. 6.3 zeigt als Beispiel die Toolbar für die Visualisierung des Unterschieds zwischen konkreter und abstrakter Syntax.

Abb. 6.3: Eine mögliche Anzeige der Toolbar

6.3.1.2 Animationsbereich

Im **Animationsbereich** des Hauptfensters findet die Visualisierung der einzelnen Themen statt. Basierend auf der graphischen Darstellung des aktuellen (abstrakten) Syntaxbaumes, kann der/die BenutzerIn auf diesem die Algorithmen starten und sich deren Funktionsweise verdeutlichen. Dabei ist es möglich, sich durch die Anwahl eines Syntaxbaumknotens die entsprechende zugehörige Annotation anzeigen zu lassen. Mehr zu den Visualisierungen im Abschnitt 6.4.

6.3.1.3 Statuszeile

Innerhalb der **Statuszeile** zeigt das ASA-Tool Hilfetexte an, die die Funktionalität der Menüelemente und der Steuerungsknöpfe der Toolbar kurz erklären. Dazu wählt man bei gedrückter linker Maustaste einen Menüpunkt aus oder drückt einen Knopf der Toolbar. Zieht man den Mauscursor wieder mit gedrückter Maustaste aus dem Knopfbereich heraus, so wird keine Aktion durchgeführt.

6.3.2 Hilfsmittel

ASA stellt mehrere **Hilfsmittel** zur komfortableren Nutzung des Tools zur Verfügung. Dazu gehören u.a. die Online-Hilfe und der ASA-Editor, die beide in einem normalen Fenster (Popup-Window) dargestellt werden, sowie mehrere modale Dialoge, etwa die Eingabemaske und der Dialog zum Einstellen der Layoutparameter für die visualisierten Bäume.

6.3.2.1 Online-Hilfe

Alle Komponenten der *Animation der semantischen Analyse* verwenden die gleiche **Online-Hilfe**. Wir haben sie bereits in Abschnitt 5.2.2.1 kennengelernt.

6.3.2.2 Editor

Der **ASA-Editor** ist in allen Animationsthemen, außer dem Thema der *Auflösung der Überladung* verfügbar. Mit seiner Hilfe lassen sich die Eingabeprogramme (in den Sprachen PASCAL oder LAMA) eingeben, anzeigen, verändern, speichern und syntaxtesten. Er wird vom Hauptfenster aus eingeblendet und bleibt solange als oberstes Fenster sichtbar, bis er manuell wieder ausgeblendet wird (vgl. Abb. 6.4).

Der Editor verfügt über alle für Windows-Editoren üblichen Editiermöglichkeiten des Textes, einschließlich Druckausgabe, Cut&Paste, Suchen bzw. Ersetzen und Schriftarten-

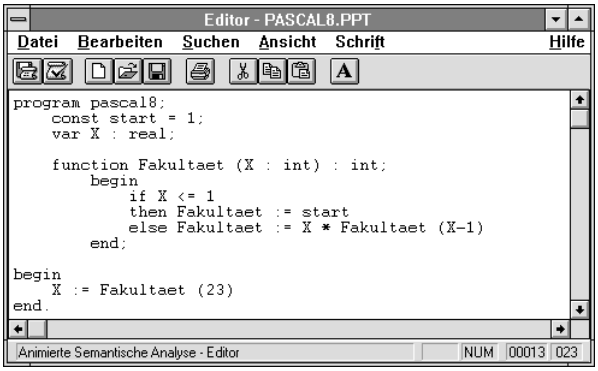


Abb. 6.4: ASA-Editor

auswahl. Diese Funktionalitäten sind über das Editor-Menü bzw. die Editor-Toolbar abrufbar. In der Statuszeile sind neben den gewohnten Hilfetexten auch Felder über die Tastaturzustände CAPS_LOCK bzw. NUM_LOCK und für eine Positionsangabe des Carets im Editfenster enthalten.

Eine Besonderheit ist der eingebaute Syntaxtester. Finden die implementierten Parser (vgl. die dazugehörigen Grammatiken in Anhang A) nach der Betätigung des entsprechenden Menüpunkts oder Buttons in der Toolbar einen Fehler im Quelltext, so erscheint eine Fehlermeldung über einen Dialog, der die Art und den Ort des Fehlers angibt. Zusätzlich markiert der Editor den Fehler im Quelltext direkt (*leichte Fehlerbehandlung*, siehe Punkt (5) aus Abschnitt 4.2).

6.3.2.3 Eingabemaske

Um die Eingabe einer Spezifikation für das Animationsthema *Auflösung der Überladung* zu vereinfachen, bietet das ASA-Tool eine **Eingabemaske** an (siehe Abb. 6.5). Die dort eingegebene Spezifikation läßt sich in einer ASCII-Datei ablegen.

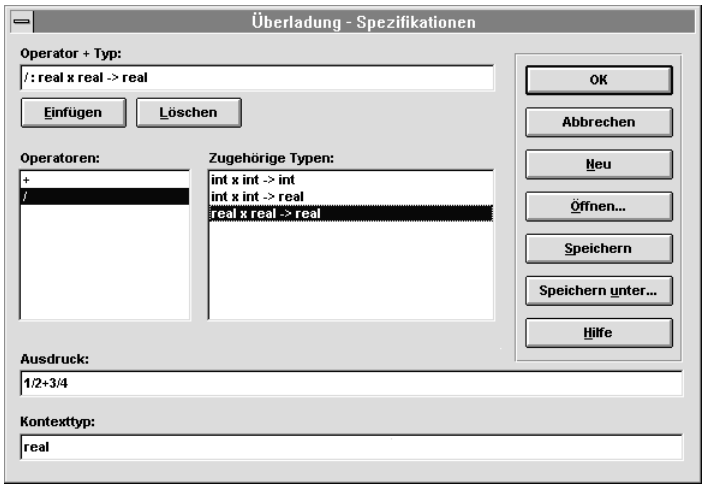


Abb. 6.5: Dialog für die Eingabemaske

Der Dialog ist in vier Bereiche aufgeteilt. Alle für die Spezifikation benötigten Eingaben können in einem einzigen Dialog vorgenommen werden (*abgeschlossener Dialog*, vgl. Punkt (4) der Prinzipien zum Dialogdesign aus Abschnitt 4.2):

Eingabe der überladenen Operatoren	
Operator + Typ:	In das Textfeld gibt man die Operatorspezifikation mit folgender Syntax ein: <Operatorname> : <Typ> wobei <Typ> = $t_1 \times \ldots \times t_m \rightarrow t$ mit t_i, $t \in$ {**int**, **real**, **char**, **bool**} für $1 \leq i \leq m$ Durch das Betätigen des *Einfügen*-Button ordnet die Eingabemaske die eingegebene Operatorspezifikation nach einem simultan ablaufenden Syntaxtest in die darunter befindlichen Listenfelder ein. Handelt es sich um einen neuen Operatornamen, so wird er neu in die Operatorliste aufgenommen. Der dazugehörige Typ kommt in die Typenliste, die sich für jeden angewählten Operator anpaßt. Falls der Operator schon in der Operatorenliste steht, so kommt der neu definierte Typ in die entsprechende Typenliste. Analog löscht der *Löschen*-Button die betreffenden Einträge in der Operator- bzw. Typenliste.
Operatoren:	Diese Liste beinhaltet alle eingegebenen (eventuell überladenen) Operatoren. Wenn auf einen Operatoreintrag geklickt wird, dann erscheint seine Typenliste.

Eingabe der überladenen Operatoren	
Zugehörige Typen:	Die Typenliste enthält alle Typen des links in der Operatorenliste markierten Operators. Wenn man auf einen Typ doppelklickt, so erscheint die entsprechende Operator/Typ-Kombination oben im Eingabefeld. Sie kann mit dem *Löschen*-Button aus den Listen entfernt werden.

Ausdruck	In das Textfeld gibt man einen Ausdruck nach folgenden Regeln ein: • Es dürfen Konstanten eines Typs aus {**int**, **real**, **char**, **bool**} verwendet werden (z.B.: 1, 6.23, 'W', …). • Jedes Zeichen bzw. jede Zeichenfolge, die keine Konstante ist, wird als Operator interpretiert. • Einstellige Operatoren werden mit ihrem Argument eingeklammert, also etwa (-23). • Zweistellige Operatoren stehen in der Mitte ihrer Argumente, beispielsweise 23 + 34.45. • Drei- und mehrstellige Operatoren werden ihren eingeklammerten, durch Kommata getrennten Argumenten vorangestellt, wie *add*(1, 5, 23).

Kontexttyp	Der Kontexttyp (*a-priori-Typ*) ist ein einfacher Typ aus der Typenmenge {**int**, **real**, **char**, **bool**}. Gemeint ist damit der Typ, den der bereits oben eingegebene Ausdruck als Rückgabetyp erhalten soll.

Buttons	
Ok:	Akzeptiert die getroffenen Eingaben und zeigt den eventuell neuen Ausdrucksbaum im Animationsbereich an.
Abbrechen:	Ignoriert die veränderten Spezifikationen und führt keine Änderungen aus.
Neu:	Schließt die aktuelle Animation und löscht alle zu dieser gehörigen Einträge sowohl in der Eingabemaske, als auch im Animationsbereich.
Öffnen:	Zeigt die *Quelldatei öffnen*-Dialogbox, aus dieser man sich eine Eingabedatei (*.OLS) auswählen kann, deren Spezifikation dann in den verschiedenen Feldern der Eingabemaske angezeigt wird.
Speichern:	Speichert die aktuelle Eingabespezifikation, falls sie verändert wurde.
Speichern unter:	Zeigt die *Quelldatei speichern*-Dialogbox, mit deren Hilfe sich eine neue Eingabedatei oder eine bereits vorhandene unter neuem Namen speichern läßt.
Hilfe:	Ruft die kontextsensitive Online-Hilfe für diesen Dialog auf.

Die der Eingabesyntax zugrundeliegende Grammatik ist im Anhang A.3 angegeben.

6.3.2.4 Layoutparameter

Der Dialog zur Einstellung der **Layoutparameter** besteht aus zwei Gruppen und vier Buttons (vgl. Abb. 6.6). Die *Layoutgruppe* enthält Einstellungen, welche das Aussehen des erzeugten Syntaxbaums zum Gegenstand haben. Andererseits dient die *Animationsgruppe* für Einstellungen Animationen betreffend (*abgeschlossener Dialog*, vgl. Punkt (4) der Prinzipien zum Dialogdesign aus Abschnitt 4.2):

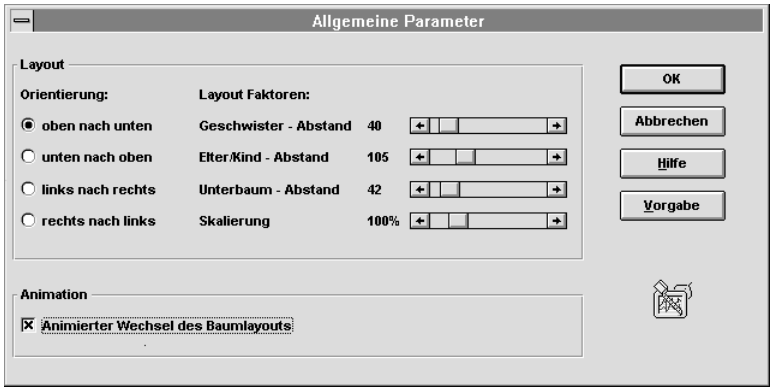


Abb. 6.6: Dialog zur Einstellung der Layoutparameter

Layoutgruppe	
Orientierung:	Hier stehen vier sich gegenseitig ausschließende Möglichkeiten zur Verfügung, die Ausrichtung des aktuellen Syntaxbaumes zu verändern. Dies ist dann sinnvoll, wenn der Baum verhältnismäßig sehr breit bzw. hoch ist, und man ihn sich komplett ansehen möchte, ohne die Skalierung zu ändern.
Layoutfaktoren:	Die Benutzung der Regler legt die verschiedenen Abstände (in logischen Einheiten, relativ zur gewählten Skalierung) zwischen den Knoten in Abhängigkeit ihrer Verwandtschaft neu fest, z.B. der Abstand zwischen Geschwisterknoten. Diese Freiheit erleichtert die Unterscheidung der Knotenattribute (z.B.: Typ). Bei einem sehr eng zusammenstehenden Baum überschneiden sich u.U. die Knotenattributierungen zweier Knoten. Durch das „Auseinanderziehen“ des Syntaxbaumes läßt sich dieses Problem schnell beheben. Schließlich besteht die Möglichkeit, den Baum im Bereich von 25% bis 400% zu skalieren. Vergleiche zu diesem Thema auch Abbildung C.2.6 in Anhang C.2.

Animationsgruppe	
Animierter Wechsel des Baumlayouts:	Ändert man das Baumlayout mit den in der Layoutgruppe aufgeführten Mitteln, so bewegen sich die Knoten (aus Effizienzgründen zu schwarzen Rechtecken reduziert) zu ihren neuen Positionen im Animationsbereich. Diese Animation ist dahingehend motiviert, daß die Änderung der Topologie des Baumes leichter nachzuvollziehen ist. Im VCG-Tool wurde diese Idee ebenfalls realisiert, siehe [San96].

Buttons	
Ok:	Akzeptiert die getroffenen Einstellungen und führt die Änderungen aus.
Abbrechen:	Ignoriert die getroffenen Einstellungen und führt keine Änderungen aus.
Vorgabe:	Setzt alle Einstellungen auf die Vorgabewerte zurück (*Rückname von Aktionen*, vgl. Punkt (6) aus Abschnitt 4.2).
Hilfe:	Ruft die kontextsensitive Online-Hilfe für diesen Dialog auf.

6.4 Visualisierungen

Wir wollen in diesem Abschnitt eine Übersicht davon geben, wie das ASA-Tool die in Kapitel 3 erläuterten Konzepte der semantischen Analyse visualisiert und welche Möglichkeiten die BenutzerInnen zur Interaktion haben. Beispiele sind im Anhang C.2 ausgedruckt.

6.4.1 Unterschied zwischen konkreter und abstrakter Syntax

Die Eingabe erfolgt entweder durch das Öffnen einer Eingabedatei (Dateiendung: *.PPT), die das Beispielprogramm (PASCAL) enthält, oder durch die Angabe eines PASCAL-Programms in den ASA-Editor. Falls es einen Syntaxfehler gibt, so wird der Editor aufgerufen, der Fehler in einer Benachrichtigung beschrieben, und die Position des Fehlers im Editor angezeigt. Hat man ihn korrigiert, so kann man das Beispielprogramm kurz noch einmal durch den Syntaxchecker testen. Somit wird ein eventueller Syntaxfehler entdeckt, der sich hinter dem bisher gefundenen Fehlern befindet, da der Syntaxchecker lediglich den ersten Fehler meldet und dann abbricht. Ist die Syntax des Eingabeprogramm korrekt, so stellt ASA den **abstrakten** und den **konkreten Syntaxbaum** im Animationsbereich graphisch dar.

Nun läßt sich die Ansicht der beiden Syntaxbäume durch das Ändern der *Layoutparameter* für den jeweiligen Fall optimieren. Die Bäume lassen sich durch die *Toolbar*-Buttons einzeln anzeigen, oder horizontal bzw. vertikal separieren (Anordnung der zwei Animationsfenster, in denen jeweils ein Baum enthalten ist). Eine für diese Funktionalität redundante Interaktionsmöglichkeit bietet die *Animationsansicht*-Dialogbox (vgl. Abb. 6.7), in der auch die Größe der unten beschriebenen Knoteninformation eingestellt werden kann. Man kann bei beiden Bäumen unabhängig voneinander zu einem beliebigen Teilbaum scrollen.

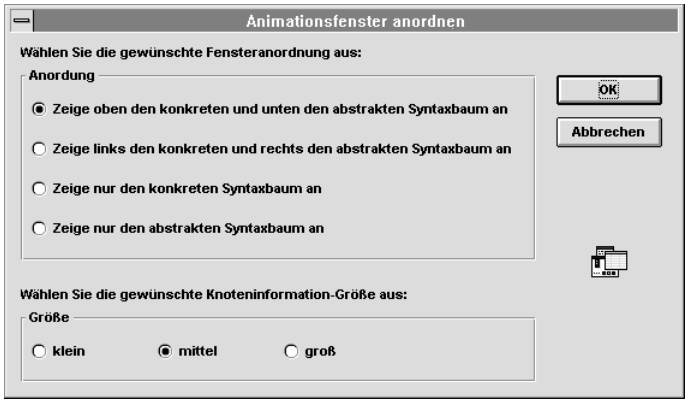


Abb. 6.7: *Animationsansicht*-Dialogbox (1)

Klickt man auf einen Knoten, so wird eine kurze Information über die Bedeutung des Knotens dahingehend gegeben, ob er im abstrakten Syntaxbaum benötigt wird oder nicht, und ob es sich bei ihm um ein Terminal oder ein Nichtterminal handelt. Dabei wird der ausgewählte Knoten rot hervorgehoben. Wählt man einen anderen Knoten aus oder ändert die *Layoutfaktoren*, so verschwindet die bisherige Information und der bis dahin aktuelle Knoten wird wieder normalfarbig (hellgrau) dargestellt. Gehört der Knoten zum abstrakten Syntaxbaum, dann beschränkt sich die angezeigte Information auf die Angabe, ob er ein Terminal- oder Nichtterminalknoten ist. Durch die *Knoten färben*-Funktionalität werden die Knoten im konkreten Syntaxbaum in solche unterteilt, die im abstrakten Syntaxbaum verschwinden (blau gefärbt) und solche, die übernommen werden (hellgrau-Standardknotenfarbe). Abbildung C.2.1 zeigt ein Beispiel dieser Visualisierung.

6.4.2 Überprüfung der Kontextbedingungen

Das Beispielprogramm (PASCAL, Dateiendung: *.PPT) wird wie im letzten Abschnitt 6.4.3 eingegeben und der entsprechende abstrakte Syntaxbaum im Animationsbereich gezeichnet. Wählt man aus dem Menü *Animation* den Punkt *Endergebnis*, so startet ASA die **Überprü-**

fung der Deklariertheitseigenschaften. Als Hinweis auf den Abschluß der Berechnungen auf allen Knoten färbt das ASA-Tool die Knoten dunkelgrau (*optisches Feedback*, siehe Punkt (3) der Richtlinien für Animationen aus Abschnitt 4.4). Die Knoten sind nun „aktiv“, d.h. sie enthalten Informationen. Weiterhin erhält der Mauscursor beim Überstreichen eines Knotens die Form einer Lupe (sonst hat er die Form eines Pfeiles). Bei einem Fehler erscheint eine Meldung und der Knoten, an dem der Fehler aufgetreten ist wird rot markiert. Klickt man auf ein angewandtes Vorkommen eines Bezeichners im Syntaxbaum, für den eine Deklaration entdeckt wurde (blau gefärbt), so weist ein roter Pfeil auf das definierende Vorkommen. Dabei wird der Knoten und seine Deklaration rot hervorgehoben. Wählt man einen anderen Knoten aus, so verschwindet der bisherige Pfeil und der alte Knoten wird wieder normalfarbig (hellgrau) dargestellt. Besitzt der Knoten keine Information, so erscheint die Information „Leer“.

Gehört der Knoten zu einem Ausdruck (auch blau gefärbt) und wurde noch kein **Typkonsistenzcheck** durchgeführt, dann erscheint ein Hinweis darauf, daß man zuerst einen solchen Test einleiten soll. Erst dann kann man auf das Typattribut, das dieser Knoten enthält, mittels Mausklick zugreifen. Diese Typattribute bleiben solange offen an dem Knoten, bis sie manuell geschlossen werden, indem man erneut auf den Knoten klickt oder die Animation zurücksetzt (*flexible Steuerung*, vgl. Punkt (1) aus Abschnitt 4.4). Sie passen sich auch simultan an ein neues Layout an, wenn sich dieses durch eine Änderung der *Layoutparameter* gewandelt hat. Abbildung C.2.2 im Anhang C.2 zeigt ein Programmbeispiel.

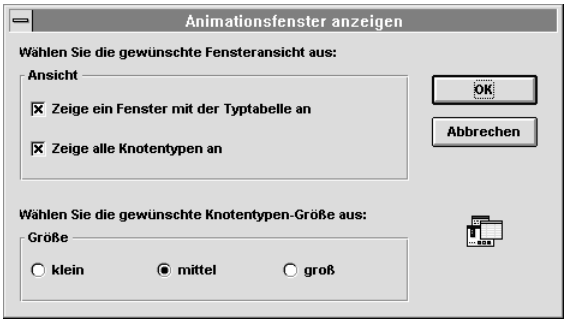


Abb. 6.8: *Animationsansicht*-Dialogbox (2)

Hierbei ist eine Besonderheit zu beachten: Besitzt der Knoten einen Typ und auch ein definierendes Vorkommen (z.B.: Variable vom Typ **real**) so bewirkt eine Mausklick mit der linken Maustaste einen Hinweis auf die Deklaration, ein Mausklick mit der rechten Maustaste das Erscheinen seines Typs. Möchte man alle Knotenattribute, hier den Typ, auf einmal anzeigen lassen, die Größe der Attribute verändern oder die Typtabelle aller im Beispiel verwendeten eingebauten Operatoren einblenden, dann lassen sich alle diese Einstellungen in einer speziellen Dialogbox (siehe Abb. 6.8) vornehmen, die durch den Menüpunkt *Animationsfenster* im Menü *Ansicht* gezeigt wird. ASA zeigt durch die Auswahl dieses Menüpunktes für jedes Animationsthema eine unterschiedliche (kontextsensitive) Dialogbox.

6.4.3 Auflösung der Überladung

Nach dem Öffnen einer Eingabedatei (Dateiendung: *.OLS), die die Beispielspezifikation enthält, oder der Eingabe der Spezifikation in die *Eingabemaske* (vgl. Abschnitt 6.3.2.3) wird der Ausdrucksbaum in dem Animationsbereich dargestellt. Falls das Programm einen Syntaxfehler erkennt, wird der Fehler in einer Benachrichtigung beschrieben und in der Maske angezeigt. Wie bei den anderen Animationsthemen auch, läßt sich die Ansicht des

Ausdrucksbaumes durch das Ändern der Layoutparameter für den jeweiligen Fall optimieren.

Wählt man aus dem Menü *Animation* den Punkt *Endergebnis*, so wird der Algorithmus zur **Auflösung der Überladung** gestartet. Daran folgt das Umfärben der Knoten zu dunkelgrau als Hinweis darauf, daß die Berechnung auf allen Knoten abgeschlossen ist. Bei einem Fehler erscheint eine Meldung und der Knoten, an dem der Fehler aufgetreten ist, wird rot markiert. Klickt man auf einen Knoten im Ausdrucksbaum, so wird seine sogenannte *ops*-Menge, in welcher alle potentiellen Typen zu diesem Operator enthalten sind, eingeblendet. Nach Beendigung des Algorithmus muß jede *ops*-Menge genau einen Typ enthalten, sonst wurde die Überladung nicht aufgelöst.

Diese *ops*-Mengen bleiben solange an dem Knoten sichtbar, bis sie durch Mausklick manuell geschlossen werden oder der/die AnwenderIn die Animation zurücksetzt. Die *ops*-Mengen passen sich auch simultan an ein neues Layout an. Abbildung C.2.3 zeigt ein Beispiel dieser Visualisierung.

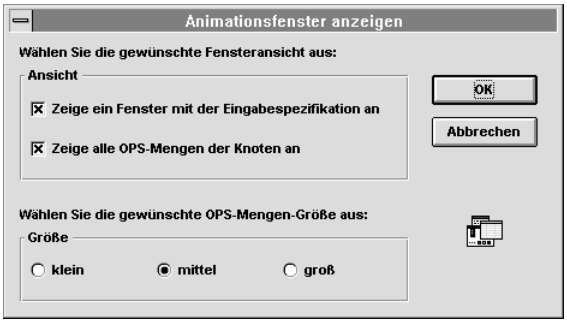


Abb. 6.9: *Animationsansicht*-Dialogbox (3)

Möchte man alle Knotenattribute, hier die *ops*-Mengen, auf einmal anzeigen lassen, die Größe der Attribute verändern oder die Eingabespezifikation in einem separaten Fenster einblenden, dann lassen sich alle diese Einstellungen in einer speziellen Dialogbox (vgl. Abb. 6.9) vornehmen, die durch den Menüpunkt *Animationsfenster* im Menü *Ansicht* gezeigt wird.

6.4.4 Typinferenz

Das Beispielprogramm (LAMA, Dateiendung: *.LPT) wird wie im Abschnitt 6.4.3 eingegeben und der entsprechende abstrakte Syntaxbaum des LAMA-Ausdrucks im Animationsbereich gezeichnet. Wählt man aus dem Menü *Animation* den Punkt *Endergebnis*, so wird die **Typinferenz** auf dem LAMA-Ausdruck gestartet. Als Hinweis auf den Abschluß der Berechnungen auf allen Knoten färbt das ASA-Tool die Knoten dunkelgrau. Bei einem Typfeh-

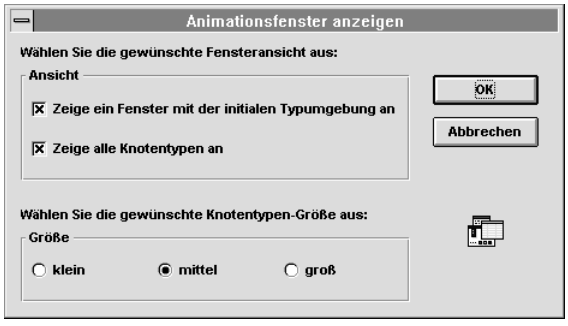


Abb. 6.10: *Animationsansicht*-Dialogbox (4)

ler erscheint eine Meldung. Klickt man auf einen Knoten im Syntaxbaum, so wird sein durch den Typinferenzalgorithmus berechneter polymorpher Typ eingeblendet.

Diese Typattribute bleiben solange an den Knoten sichtbar, bis sie durch erneuten Mausklick manuell geschlossen werden oder die Animation auf ihren Anfangszustand zurücksetzt. Sie passen sich wie üblich simultan an ein neues Baumlayout an. Möchte man alle Knotenattribute, hier die polymorphen Typen, auf einmal anzeigen lassen, die Größe der Attribute verändern oder die initiale Typumgebung aller im Beispiel verwendeten, eingebauten LAMA-Operatoren einblenden, dann lassen sich alle diese Einstellungen in einer speziellen Dialogbox (vgl. Abb. 6.10) vornehmen, die durch den Menüpunkt *Animationsfenster* im Menü *Ansicht* gezeigt wird. Zu dieser Visualisierung vergleiche auch die Abbildungen C.2.4 und C.2.5 im Anhang C.2.

6.5 Basisstruktur abstrakter Syntaxbaum

Es erschien dem Autor wichtiger, die Programmierung unter Windows in Kapitel 1 allgemein zu erläutern, als die Implementierung des ASA-Tools unter dem Windows-API im einzelnen. Dennoch gehen wir in diesem Abschnitt kurz auf die bedeutendste Datenstruktur bei der Programmierung von ASA ein, nämlich der Implementierung des Syntaxbaums. Er muß aus den Eingabebeispielen erzeugt und nach einer Layoutphase graphisch repräsentiert werden. Auf dieser Datenstruktur laufen die verschiedenen Algorithmen ab, die im Anhang B angeführt sind.

6.5.1 Erzeugung der Datenstruktur

Die Datenstruktur des Syntaxbaumes ist für PASCAL- und LAMA-Programme identisch, wobei die verschiedenen Knoten durch eine eindeutige Bezeichnung unterschieden werden. Ein Knoten besteht stark vereinfacht aus folgenden Einträgen:

- Informationen für den Aufbau der Syntaxbaum-Datenstruktur selbst, etwa Knotentyp, Anzahl der Kinder, Zeiger auf die Kinder, Elternknoten, etc.
- Für die graphische Repräsentation benötigte Informationen, z.B. *x*- und *y*-Koordinate, Knotendimension, Beschriftung, Anzahl der Beschriftungszeilen im Knoten, usw.
- Hinweise auf die Herkunft des mit dem Knoten symbolisierten Sprachkonstrukts im Quelltext des Beispielprogramms (s.u.).
- Das in Abhängigkeit vom Animationsthema mit dem Knoten assoziierte Attribut, also beispielsweise der Zeiger auf die Deklarationsstelle, der (polymorphe) Typ, die mit einem Operatorknoten assoziierte *ops*-Menge, etc.
- Informationen, die für die Visualisierung und Animation selbst gebraucht werden. Dazu zählen der Fensterhandle für das Knotenfenster (vgl. Abschnitt 6.5.3), Animationszustände und die Pfadangabe für den Pfeil vom angewandten zum entsprechenden definierenden Vorkommen bei der Animation zur *Überprüfung der Kontextbedingungen*.

Um die Syntax der Beispielprogramme zu testen, haben wir die Scanner- bzw. Parsergeneratoren *FLEX* und *BISON* verwendet (siehe [DS91]). Die zugrundeliegenden Grammatiken (und Symbolklassen für den Scanner) sind in Anhang A ausgedruckt. Der LALR(1)-Parsergenerator *BISON* erlaubt es, beliebige Aktionen durchzuführen, wenn der rechte Teil einer Produktionenregel zu einem entsprechenden Nichtterminal reduziert werden kann. Dies

nutzt man aus, um die Syntaxbaumstruktur simultan zum Analysevorgang aufzubauen. Wir geben als Beispiel eine Produktionenregel an:

```
statement : variable ':=' expression
              {$$ = build2 (AssignStat, $1, $3, @2);}
          | ...
          ;
```

Kann die rechte Seite dieser Regel (*variable* ':=' *expression*) zu dem Nichtterminal *statement* reduziert werden, dann erzeugt die Funktion *build2* einen neuen Syntaxbaumknoten des Typs *AssignStat* mit zwei Kindern. Das erste Kind ist der mit dem Nichtterminal *variable* assoziierte Teilbaum, wobei $1 (*BISON*-Notation) ein Zeiger auf diesen Teilbaum ist. Dementsprechend ist $3 ein Zeiger auf einen anderen Teilbaum, der bei der Reduktion zu einem *expression*-Nichtterminal generiert wurde. Wie oben wird dieser Teilbaum zum zweiten Kind des neuen Knotens *AssignStat*. Auf diese Art und Weise erzeugt der Parser nacheinander Teilbäume, die er dann als Kinder eines Elternknotens zusammengefaßt. @2 zeigt auf eine Struktur mit Positionsangaben des Terminals ':=' im zu analysierenden Quelltext des Beispielprogramms, mehr dazu weiter unten. $$ bekommt das Resultat der Funktion *build2* zugewiesen und zeigt auf den neu geschaffenen Knoten. Dieser Zeiger wird von einem Knoten weiter oben im Syntaxbaum verwendet, der wiederum dieses *statement* als Kind hat.

Gibt man ein bzgl. der Syntax fehlerhaftes Beispielprogramm ein, so verbleiben mit dem oben beschriebenen Verfahren eventuell Syntaxbaumfragmente im Speicher. In diesem Fall hat das ASA-Tool umsonst Speicherplatz für die Knoten des Syntaxbaums allokiert und Teile des Syntaxbaums aufgebaut. Dieses Problem umgehen wir, indem wir zwei Läufe über das Eingabeprogramm initiieren. Der erste testet sehr schnell die Korrektheit der Syntax, während im zweiten Durchlauf der Syntaxbaum erzeugt wird. Da die Beispielprogramme meist sehr klein sind, fällt dieser zusätzliche Test nicht ins Gewicht. Erwähnenswert ist, daß die Generatoren *FLEX* und *BISON* sicher nicht für die Anwendung für Windows-Programme konzipiert wurden. Ob die generierten Parser bei jeder Anwendung den allokierten Speicher wieder vollständig freigeben, ist dem generierten C-Code nur schwer zu ersehen. Eine Überprüfung des Speichers nach vielfachen Syntaxtests während des laufenden Programms hat allerdings auch keinen auffälligen Speicherplatzverlust ergeben.

Die implementierten Parser testen aber nicht nur die Syntax des Eingabeprogramms und bauen nicht nur die Datenstruktur des aus dem Eingabeprogramm resultierenden Syntaxbaumes auf. Gleichzeitig suchen sie die Komponenten der verschiedenen Sprachkonstrukte im Quelltext, z.B. die Einzelkomponenten (Schlüsselwörter) einer *if_then_else_fi*-Anweisung, und speichern deren Position im Quelltext an den entsprechenden Knoten im Syntaxbaum, etwa an einen Knoten *ifThenElseStat*. Das sind mehr Informationen, als das ASA-Tool in der gegenwärtigen Version verwenden kann. Aber für zukünftige Projekte könnten sich diese Textpositionen im Syntaxbaum eines Beispielprogramms als nützlich erweisen. Klickt z.B. der/die AnwenderIn auf einen Knoten (*ifThenElseStat*) in der Syntaxbaumvisualisierung, dann könnte das entsprechende (mehrteilige) Sprachkonstrukt im Quelltext als zusätzliche Information „randscharf" farbig markiert werden.

6.5.2 Layoutberechnung für Bäume

Um den sich im Speicher befindlichen Syntaxbaum auch graphisch im Animationsbereich anzuzeigen, muß ASA zunächst ein (möglichst ästhetisches) **Baumlayout** berechnen. In diesem Abschnitt geben wir den dazu notwendigen Layoutalgorithmus an. Dazu benötigen wir die Definition einiger Begriffe aus der Graphentheorie:

Definition 6.5.1 (Graph)
Sei V eine endliche Menge. Ein **Graph** ist ein Tupel $G = (V, E)$ mit $E \subseteq V \times V$. V ist die Menge der Knoten. E ist die Menge der Kanten. Für eine Kante $(v, w) \in E$ schreiben wir auch $v \longrightarrow w$.
Die Menge $pred(v) = \{w \in V \mid (w, v) \in E\}$ ist die Menge aller Vorgänger von v.
Die Menge $succ(v) = \{w \in V \mid (v, w) \in E\}$ ist die Menge aller Nachfolger von v.
Die Größen dieser Mengen sind $indeg(v) = |pred(v)|$ und $outdeg(v) = |succ(v)|$.
Der Grad eines Knotens v ist $degree(v) = indeg(v) + outdeg(v)$. □

Definition 6.5.2 (Pfad, Zyklus)
Sei $G = (V, E)$ ein Graph.
Ein **Pfad** $v \longrightarrow^n w$ der Länge n ist eine Folge von Knoten $v_0,\ldots,v_n$ mit $v_0 = v$, $v_n = w$, $(v_i, v_{i+1}) \in E$ für $0 \le i \le n-1$. Ein Pfad der Länge 0 heißt **leerer Pfad**.
Wir schreiben $v \longrightarrow^* w$, falls ein $n \ge 0$ existiert, so daß $v \longrightarrow^n w$.
Wir schreiben $v \longrightarrow^+ w$, falls der Pfad $v \longrightarrow^* w$ nicht leer ist.
Ein **Zyklus** ist ein Pfad $v \longrightarrow^+ v$. Ein Graph $G = (V, E)$ ist *azyklisch* (**DAG** = „Directed Azyclic Graph"), falls er keine Zyklen enthält. □

Definition 6.5.3 (Baum)
Ein (gerichteter) **Baum** T ist ein DAG $G = (V, E)$ mit n Knoten und $n-1$ Kanten, bei dem ein ausgezeichneter Wurzelknoten v_0 (genannt: *Apex*) existiert, so daß $v_0 \longrightarrow^* v$ für jedes $v \in V$.
In einem Baum T gilt $indeg(v) = 1$ für jeden Knoten $v \neq v_0$. □

Definition 6.5.4 (allgemeiner Baum)
Ein Baum T heißt **allgemeiner Baum**, wenn für jeden Knoten $v \in V$ die Anzahl seiner Nachfolger $outdeg(v)$ beliebig ist. □

Der hier beschriebene Algorithmus berechnet das **Layout von allgemeinen Bäumen**. Er entspricht dem Baumlayout-Algorithmus, der auch im VCG-Tool (vgl. [San96]) eingesetzt ist.

Bei der Entwicklung des ASA-Tools hatten wir zur Implementierung eines eigenen Layoutalgorithmus noch eine Alternative: die Verwendung des VCG-Tools mittels DDE. Leider war dessen Portierung auf Windows 3.1 noch nicht abgeschlossen, so daß dies zu Wartezeiten geführt hätte. Ein weiteres Problem ist der damit verbundene Speicherverbrauch. Im Minimalfall würden sich bei einer solchen DDE-Kommunikation drei Windows-Programme im Hauptspeicher befinden: die ToolBook-Anwendung zur *animierten Präsentation*, das ASA-Tool und das VCG-Tool. Eine Plattform mit wenig Hauptspeicher (≤ 8 MB) hätte dies sicherlich nur schwer verkraftet.

6.5.2.1 Layoutkriterien

Die **Layoutkriterien** geben an, wann ein Baumlayout ästhetisch wirkt:

- Knoten auf der gleichen Baumebene („Level") sollten entlang einer Geraden liegen und diese Geraden sollten zueinander parallel sein.
- Ein Elternknoten sollte über seinen Nachkommen zentriert liegen.

- Ein Baum und sein Spiegelbild sollten so gezeichnet werden, daß die Zeichnungen Reflexionen voneinander sind. Überdies sollte ein Unterbaum in derselben Art und Weise gezeichnet werden, ohne Berücksichtigung, wo er sich im Baum befindet.
- Kleine Unterbäume sollten zwischen großen Unterbäumen nicht willkürlich positioniert erscheinen:
  - Kleine, innere Unterbäume sollten zwischen größeren Unterbäumen weiträumig verteilt werden (wobei die größeren Unterbäume auf einer oder mehreren Ebenen benachbart (adjazent) sind).
  - Kleine Unterbäume, die weit rechts bzw. weit links im Baum liegen, sollten zu größeren Unterbäumen benachbart sein.
- Das Baumlayout sollte eng sein, d.h. nicht unnötig viel Platz verbrauchen.

Der in ASA verwendete und nachfolgend beschriebene Layoutalgorithmus erfüllt diese Layoutkriterien. Er ist in [Wal90] angegeben und genauer beschrieben.

6.5.2.2 Arbeitsweise des Layoutalgorithmus

Wir nehmen bis auf weiteres an, daß sich der Apex des Baumes oben in der Zeichnung befindet und ein Knoten nur den Raum eines Punktes einnimmt. Der Algorithmus muß sich lediglich mit der Bestimmung der *x*-Koordinaten der Knoten beschäftigen, die *y*-Koordinate eines Knotens kann leicht aus seiner Ebene im Baum abgeleitet werden. Der Abstand der einzelnen Ebenen in der Zeichnung ist variabel und wird als *Ebenenseparation* bezeichnet. Unser Layoutalgorithmus verwendet die folgenden zwei Konzepte:

- *Unterbäume sind starre Einheiten*: Verschiebt der Algorithmus einen Knoten, so werden alle seine Nachkommen (wenn er welche hat) mitbewegt. Dies gilt ebenso für den ganzen Baum. Eine Wurzelverschiebung (Apex) hat die entsprechende Verschiebung des ganzen Baumes zur Folge.
- *Benutzung zweier Felder für die Positionierung jedes Knotens*: Diese zwei Felder sind für einen Knoten *k*
  - eine vorläufige *x*-Koordinate (*prelim*(*k*)) und
  - ein Modifizierer-Feld (*modifier*(*k*)), in dem Verschiebungen der *x*-Koordinate abgespeichert werden.

Um die endgültige *x*-Koordinate zu berechnen, muß der Baum **zweimal traversiert** werden. Die erste Traversierung weist für jeden Knoten *k* *prelim*(*k*) und *modifier*(*k*) zu; die zweite Traversierung berechnet die endgültige *x*-Koordinate eines jeden Knotens *k* durch die Summation von *prelim*(*k*) und den Modifizierer-Feldern aller Vorfahren von *k*. Dadurch können Unterbäume schnell und einfach verschoben werden. Der Algorithmus berechnet für einen Baum $T = (V, E)$ ein Layout in linearer Zeit $O(|V|)$.

Die **erste Traversierung** ist eine *Postorder*-Traversierung. Zunächst werden die kleinsten Unterbäume (Blätter) positioniert und dann rekursiv, von links nach rechts aufsteigend, die immer größer werdenden Unterbäume. Geschwisterknoten werden durch eine vordefinierte Minimaldistanz (*Geschwisterseparation*) separiert. Benachbarte Unterbäume separiert man in analoger Weise durch eine weitere vordefinierte Minimaldistanz (*Unterbaumseparation*). Beide Werte können unterschiedlich sein (vgl. Abb. 6.11).

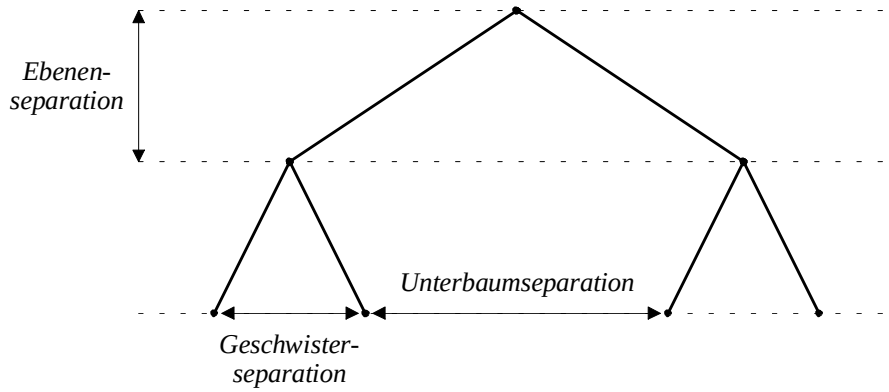


Abb. 6.11: *Ebenenseparation, Geschwisterseparation, Unterbaumseparation*

Währenddessen sich der Algorithmus von den Blättern zum Apex bewegt, kombiniert er kleinere Unterbäume zu größeren. Er positioniert die Unterbäume eines Knoten *k* nacheinander, von links nach rechts. Überlagert ein solcher Unterbaum seinen linken Nachbarn, so schiebt ihn der Algorithmus so lange nach rechts bis sich keine Punkte mehr berühren (siehe Abb. 6.12). Dazu werden zuerst ihre Wurzeln *k.i* um den Wert der *Geschwisterseparation* voneinander abgesondert (*prelim*(*k.i*) = *prelim*(*k.i*-1) + *Geschwisterseparation*). Dann werden die Unterbäume im nächstniedrigeren Level solange verschoben, bis zwischen den benachbarten Unterbäumen auf der unteren Ebene ein Abstand von *Unterbaumseparation* erreicht ist. Dieser Prozeß setzt sich sukzessive auf den unteren Ebenen fort, bis wir zu den Blättern der kürzeren Unterbäume kommen. Diese Verschiebungen werden in den Feldern *modifier*(*k.i*) abgespeichert. Hierbei ist zu beachten, daß der neue Unterbaum nicht zwangsläufig gegen einen Nachkommen seines unmittelbar linken Geschwisterknoten stoßen muß. Geschwisterknoten mit vielen Kindern weiter links, können Ursache dafür sein, daß der neue Unterbaum noch weiter nach rechts verschoben wird. Bewegungen dieser Art sind nicht immer nötig. Ist dieser Prozeß für alle Nachkommen des Knotens *k* abgeschlossen, dann wird der Knoten *k* über seinem linkesten und rechtesten Nachkommen zentriert. Hat *k* auf derselben Ebene keine linken Geschwister, so wird nur *prelim*(*k*) gesetzt und *modifier*(*k*) ist 0. Andernfalls ist der gesamte Unterbaum von *k* durch das entsprechende Setzen von *modifier*(*k*) zu verschieben.

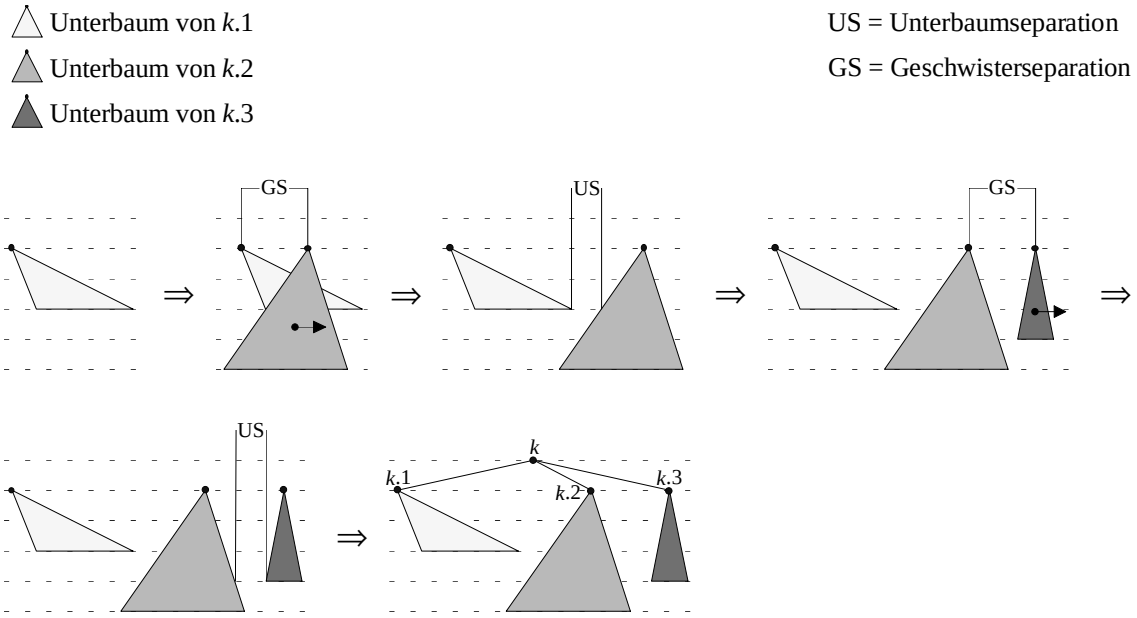


Abb. 6.12: Layoutvorgang am Beispiel eines Knotens *k*

Wenn man einen neuen, großen Unterbaum immer mehr nach rechts schiebt, so kann zwischen diesem großen Unterbaum und kleineren Unterbäumen, die zuvor korrekt positioniert waren, eine Lücke entstehen. Diese kleinen Unterbäume erscheinen nun wie angehäuft, während rechts von dieser Anhäufung ein leerer Raum entsteht. Der hier vorgestellte Algorithmus löst dieses Problem (vgl. Abb. 6.13). Er generiert einen gleichmäßig verteilten Raum um die Unterbäume. Während sich ein großer Unterbaum nach rechts bewegt, so wird die zurückgelegte Distanz gleichmäßig auf die kleineren inneren Unterbäume verteilt. Die Bewegung dieser Unterbäume wird wie oben beschrieben, durch Addition der proportionalen Werte zu den vorläufigen *x*-Koordinaten und Modifizierer-Feldern der Unterbaumwurzeln, durchgeführt. Haben wir beispielsweise auf der linken Seite drei kleine gebündelte Unterbäume, weil ein neuer großer Unterbaum rechts positioniert wurde, so wird der erste kleine Unterbaum um ¼ der entstandenen Lücke, der zweite kleine Unterbaum um ½ und der dritte kleine Unterbaum um ¾ nach rechts gependelt.

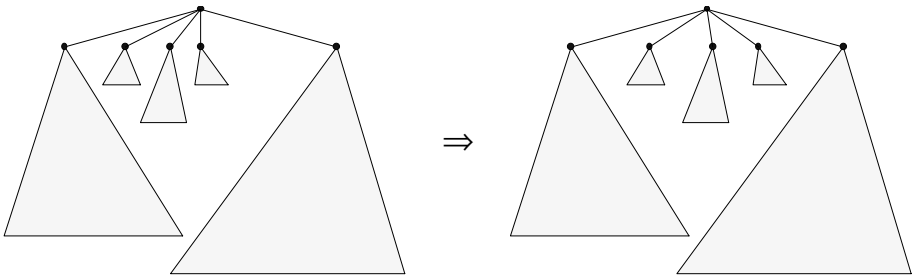


Abb. 6.13: Auspendeln von Unterbäumen

Die **zweite Traversierung**, eine *Preorder*-Traversierung, bestimmt die endgültige *x*-Koordinate von jedem Knoten. Sie startet beim Apex, summiert für jeden Knoten *k* den Wert *prelim*(*k*) mit der vereinigten Summe der Modifizierer-Werte seiner Vorfahren. Sie addiert dazu auch einen Wert, der die gewünschte Zentrierung des Baumes im Animationsbereich gewährleistet.

Unsere Implementierung dieses Layoutalgorithmus ist eine erweiterte Version des oben beschriebenen Algorithmus. Die Syntaxbaumknoten haben je nach Beschriftung eine unterschiedliche Höhe und Breite. Das Layoutverfahren mußte für diesen Umstand angepaßt werden. Weiterhin ist der Baum in vier Richtungen drehbar, d.h. der Apex des Baumes kann sich im Fenster oben, unten, rechts oder links befinden (siehe Abb. 6.14). Die Layoutparameter *Ebenenseparation*, *Geschwisterseparation* und *Unterbaumseparation* lassen sich beliebig verändern. Zusammen mit der Skalierungsmöglichkeit des Baumes wird den AnwenderInnen ermöglicht, die jeweilige optimale Darstellung des Syntaxbaumlayouts im Animationsbereich einzustellen (siehe auch Abschnitt 6.3.2.4).

a) Apex oben b) Apex links c) Apex unten d) Apex rechts

Abb. 6.14: Baumorientierungen

Da jeder Syntaxbaumknoten in ASA eine unterschiedliche Größe hat, muß diese vor dem eigentlichen Layoutvorgang ermittelt werden. Sie ist von der Beschriftung des Knotens und von dem jeweiligen Font (der sich auch je nach Ausgabegerät und System unterscheiden

kann) abhängig. Unter *Font* verstehen wir hier einen *logischen Font*, wie er in Kapitel 1 definiert wurde.

Dazu wird in einem Pass über die ganze Datenstruktur des Syntaxbaums die Beschriftung für jeden Knoten untersucht. Eine Window-Funktion *GetTextExtent* ermittelt die Breite und die Höhe einer Textzeile („string“), indem sie den in den Gerätekontext eingesetzten Font als Grundlage nimmt. Besteht die Knotenbeschriftung aus zwei Zeilen, so addiert das Programm die jeweiligen Höhen und bildet aus den Breiten das Maximum. Diese Dimensionen der Knoten werden bei den Separationen, die der Layoutalgorithmus ausführt, berücksichtigt.

Tatsächlich geschieht die schon erwähnte Skalierung der gezeichneten Knoten genau an obiger Stelle. Man muß lediglich die Größe des in den Kontext eingesetzten Fonts proportional verändern. Die Knoten werden demzufolge entsprechend kleiner bzw. größer. Ändert man nun proportional die Linienstärken der Knotenumrandung und der Baumkanten, sowie die Werte *Ebenenseparation*, *Geschwisterseparation* und *Unterbaumseparation*, dann hat man nach einer erneuten Layoutberechnung den ganzen Baum skaliert. Zu Beachten ist jedoch, daß die in der *Parameter*-Dialogbox einstellbaren Werte für diese *Layoutparameter* relativ gesehen gleich bleiben.

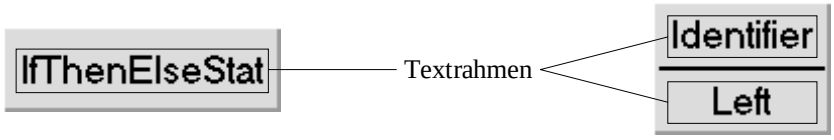
6.5.3 Graphische Darstellung

Nach der Berechnung des Baumlayouts enthalten die Knoten der Syntaxbaumstruktur ihre endgültige Positionsangabe. Soll der Baum gezeichnet werden, so erhält die Windows-Prozedur des Animationsfensters (Animationsbereich) die Botschaft WM_PAINT.

Zuerst werden die Baumkanten aus den Positionsangaben der Knoten berechnet und im Animationsbereich gezeichnet. Dazu versieht ASA die Kanten der Baumhierarchie entsprechend mit Pfeilspitzen. Für die Knoten des Baumes erzeugt das ASA-Tool allerdings eigene Fenster (Knotenfenster), in denen jeweils ein Knoten gezeichnet wird. Diese Fenster sind Child-Fenster des Animationsfensters. Vorteile dieses Verfahrens sind:

- *Knotenfenster sind Objekte*: Sie basieren auf einer eigenen Fensterklasse und haben damit auch eine eigene Window-Prozedur. Man kann sie im Anwendungsbereich ihres Parent verschieben, ohne sich um deren graphische Repräsentation zu kümmern.
- *Knotenfenster kann man anklicken*: Die Trefferprüfung für einen Mausklick übernimmt das Windows-System. Die Botschaft WM_LBUTTONUP wird direkt an das betroffene Knotenfenster geschickt, wenn z.B. die linken Maustaste auf dem Knoten losgelassen wurde.

Zur graphischen Baumausgabe werden diese Knotenfenster einfach zu den berechneten Positionen im Animationsbereich geschoben.



a) einteiliges Knotenfenster b) zweiteiliges Knotenfenster

Abb. 6.15: Knotenfenster

Erhält das Parent-Fenster (Animationsfenster) der Knotenfenster die Botschaft WM_-PAINT, so wird diese Botschaft auch an die Knotenfenster selbst versandt. Diese zeichnen ihren Fensterinhalt wie in Abb. 6.15 gezeigt. Dazu zentriert ASA eine Textzeile der Kno-

tenbeschriftung horizontal in einem Textrahmen, dessen Dimension die bereits oben beschriebene Funktion *GetTextExtent* liefert.

Nach dem Zeichnen des Syntaxbaumes kann der/die AnwenderIn nach der Durchführung des jeweiligen animierten Algorithmus auf die Knotenfenster klicken. Die Knotenattribute (Typen, *ops*-Mengen, etc.) erscheinen daraufhin ebenfalls in kleinen Fenstern, die identisch zu den Knotenfenstern aufgebaut und dementsprechend implementiert sind.

6.5.4 Einbettung von Wegen innerhalb der Baumdarstellung

Bei der *Überprüfung der Kontextbedingungen* können Syntaxbaumknoten angeklickt werden, die für ein angewandtes Vorkommen eines Bezeichners stehen. Es erscheint ein Pfeil auf den ein definierendes Vorkommen symbolisierenden Knoten. Dazu ist es nötig einen Algorithmus zu finden, der einen entsprechenden Polygonzug durch den Syntaxbaum berechnet. Zur Vereinfachung betrachten wir hier nur Baumlayouts, deren Apex oben im Fenster liegt und Knoten mit einer horizontalen bzw. vertikalen Ausdehnung von einem Pixel. Wir haben folgende Anforderungen an das Aussehen dieses Polygonzugs gestellt:

- Er sollte keine Baumkanten bzw. Knoten schneiden oder berühren.
- Es sollte immer nur ein Polygonzug gleichzeitig angezeigt werden. Damit ersparen wir es uns, mehrere Pfeile so zu berechnen, daß sie sich möglichst wenig kreuzen. Außerdem erhöht dies die Übersichtlichkeit im Animationsbereich.
- Sind Quell- und Zielknoten nicht benachbart und befindet sich zwischen ihnen ein Teilbaum, dessen Blätter auf einer tieferen Ebene liegen als die Ebenen des Quell- oder Zielknotens, so sollte der Polygonzug unter diesem Teilbaum herumgeführt werden.

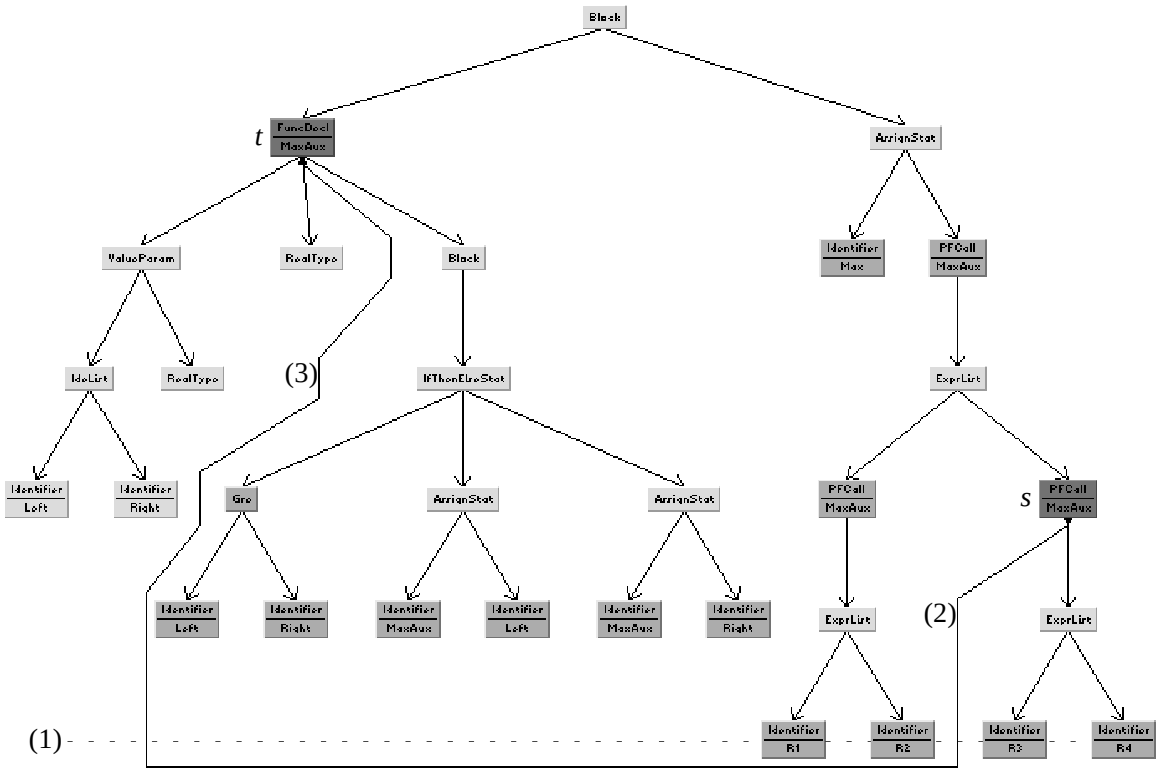


Abb. 6.16: Markierung eines definierenden Vorkommens durch einen Polygonzug

- Der erzeugt Polygonzug sollte möglichst kurz sein, aber dennoch ästhetisch wirken.

Der **Algorithmus** berechnet diesen Polygonzug, der durch ein Feld von Koordinatenpunkten repräsentiert wird, in vier Schritten:

1. Berechne den tiefsten Level (Ebene) unter dessen Knoten der Zug horizontal verlaufen kann, ohne eine Kante bzw. einen Knoten zu schneiden oder berühren.
2. Berechne den Teilzug von Quellknoten *s* zu dem in (1) ermittelten Level.
3. Berechne den Teilzug von Zielknoten *t* zu dem in (1) ermittelten Level.
4. Sortiere den in (3) gewonnenen Polygonzug in umgekehrter Reihenfolge und erweitere den Polygonzug in (2) um diesen Teilzug. Verbindet man nun alle Punkte im resultierenden Feld, so erhält man den gewünschten Pfeil.

Abbildung 6.16 zeigt einen auf diese Weise erzeugten Polygonzug. Um die Schritte (1) bis (3) zu implementieren sind umfangreiche Untersuchungen der zugrundeliegenden Baumstruktur erforderlich, deren Erklärung den knappen Rahmen dieser Arbeit sprengen würde.

Eine Umwandlung dieses Polygonzugs in eine Splinekurve würde den ästhetischen Anspruch weiter verbessern. In unserer Implementierung ist diese Umwandlung jedoch nicht enthalten.

Kapitel 7

Zusammenfassung und Ausblick

Mehrere Vorführungen vor Schülern, Lehrern und Studenten im Rahmen von Schülertagen und Präsentationsdemos haben großes Interesse und Neugier an diesem Projekt geweckt. Es bedurfte keiner großen Anstrengung, diese Personen für das Experimentieren mit den Animationen der lexikalischen und semantischen Analyse zu ermuntern. In nachfolgenden Diskussionen wurde uns auch der Vorteil von Animationen in diesem Bereich bestätigt. Ebenso ist die optische Gestaltung der Benutzeroberflächen und Animation auf Zuspruch gestoßen.

Weniger befriedigend ist der konzeptionelle Bruch, der durch die Trennung der beiden Programmteile *animierte Präsentation* und *ASA* entsteht. Wir haben gezeigt, daß dieses Problem von den Schwächen des zugrundeliegenden Autorensystems *Multimedia ToolBook 3.0* erzwungen wurde. Die alleinige Verwendung von MTB 3.0 hätte unter den gestellten Anforderungen an das fertige Programm zu keinem Erfolg geführt. Andererseits ist eine direkte Programmierung des Windows-API (wie bei ASA durchgeführt) für das ganze Programm zu komplex und zeitaufwendig. Es wäre sowohl für die EntwicklerInnen als auch für die AnwenderInnen besser, wenn die gesamte *Animation der semantischen Analyse* mit einem einzigen Tool erstellt worden wäre. Dieses Tool sollte die Eigenschaften und Stärken von Autorensystemen (MTB 3.0) mit denen von Programmen zur Animationsgenerierung verbinden. Vielleicht erfüllt ein Nachfolgeversion von MTB 3.0 diesen Anspruch.

Der Autor möchte an dieser Stelle nicht unerwähnt lassen, daß ein großer Teil der Entwicklungszeit des Programms mit dem Abwägen der in Kapitel 4 vorgestellten Prinzipien zur Erstellung von Animationssoftware verwendet wurde. Es gibt leider kein Patentrezept dafür, wie man z.B. die Benutzeroberfläche designen muß oder Animationen entwirft. Verwendet man zuviel Farbe oder zuwenig. Hat man zu viele Objekte auf einer ToolBook-Seite plaziert oder wirkt eine Seite zu langweilig, etc. Letztendlich spielen viele verschiedene, eventuell sich gegenseitig ausschließende Faktoren eine Rolle, bis hin zum persönlichen Geschmack und Vorlieben bzw. Abneigungen in der Darstellung.

Während der Entwicklung unseres Programms ist der 32 Bit-Nachfolger von Windows 3.1, Windows 95, erschienen. Die *Animation der semantischen Analyse* ist als 16 Bit-Anwendungen ohne Probleme darauf lauffähig. Eine Ausnahme bildet die ToolBook-Applikation (*animierte Präsentation*), die kleinere Darstellungsprobleme unter Windows 95 hat, etwa leichte Farbunterschiede. MTB 3.0 befindet sich nämlich auf der schwarzen Liste der nicht vollständig kompatiblen 16 Bit-Anwendungen zu Windows 95 (siehe [Wir95a]). Erwähnenswert ist, daß sich unter Windows 95 das Aussehen der Dialogfenster des ASA-Tools an die graphischen Konventionen dieser Umgebung anpaßt. Die für den 3D-Effekt der Dialogfenster zuständige dynamische Bibliothek CTL3DV2.DLL (siehe Abschnitt 1.3.4) schaltet sich unter Windows 95 selbständig ab.

Optimierungen

Gerade bei der Güte der **Graphikausgabe** sind unter Windows 95 Verbesserungen des Programms möglich. Ein Weg, die Graphikausgabe im Vergleich zu Windows 3.1 zu beschleunigen, ist die neue Graphikschnittstelle *Win G* von Windows 95. Sie ist alternativ zum GDI zu verwenden und ermöglicht einen Direktzugriff auf die Graphikhardware. Dies ist vor allem für die Darstellung von schnellen und ruckelfreien Animationen nützlich.

Schließlich bietet Windows 95 ein verbessertes Multitasking, das eventuell für bessere (dynamische) **Algorithmenanimationen** sorgen könnte (wir konnten leider noch keine Programmiererfahrung auf diesem System sammeln). Für diesen Gebrauch wäre ein *preemptives* Multitasking am besten geeignet, da man dort leichter zwischen mehreren Prozessen umschalten kann. Mit *preemptiven* Multitasking ist das schrittweise „Ablaufenlassen" der animierten Algorithmen in ASA leicht zu realisieren. Das *non-preemptive* Multitasking von Windows 3.1 erschwert die Realisierung dieser Idee sehr. Wir haben unsere Implementierung bereits so vorgenommen, daß für diesen Zweck keine großen Änderungen am bestehenden Quelltext vorgenommen werden müssen und die hierfür benötigten Steuerungselemente im Programm nur noch sichtbar zu machen sind.

Zusätzliche Möglichkeiten der **Modularisierung** vom Programmen bieten die sogenannten VBX-Kontrollen von *Visual Basic,* sowie die neuen OCX-Kontrollen unter Windows 95. Unter diesen Begriffen versteht man Module, die einfach in ein Programm eingebunden werden können, um spezielle Aufgaben zu übernehmen. Dazu gehören z.B. eine kleine primitive Datenbank, Tachometeranimationen, etc. Während VBX-Kontrollen nur mit *Visual Basic* erstellt werden können, ist die Erzeugung von OCX-Kontrollen mit mehreren Compilern möglich.

In Kapitel 5 haben wir mehrere **Basisanimationen** kennengelernt, deren Geschwindigkeit bei Bewegungen von Objekten noch von der Leistungsfähigkeit des Prozessors bzw. der Graphikkarte abhängen. Wie schnell solche Elementaranimationen hintereinander abgespielt werden, ist jedoch timergesteuert und somit vom System unabhängig. Die Geschwindigkeit von bewegten Objekten sollte aber auf allen Plattformen gleich sein. Optimal wäre es, wenn das Programm mit wachsender Prozessorleistung eine Bewegung immer feiner abstuft.

Wir haben das ASA-Tool nur für kleine Eingabebeispiele konzipiert. Große Eingaben haben aufgrund der benötigten Graphikleistung bei der graphischen Ausgabe einen hohen Speicherverbrauch zur Folge. Weiterhin wirken sehr große Syntaxbäume in einem Fenster leicht unübersichtlich. Eine möglich Verbesserung ist die Verwendung von **Sammelknoten**, ähnlich zum VCG-Tool. Um die Übersicht zu verbessern, können damit mehrere Knoten (oder Unterbäume) zusammengefaßt werden.

Ausblick

Am Lehrstuhl für Compilerbau und Programmiersprachen wird auf dem Gebiet der *Visualisierung von Übersetzern* weiter geforscht. Neben dem Ansatz, die Animationen und Visualisierungen mit dem Autorensystem MTB 3.0 und seiner Nachfolger unter Windows zu implementieren, gibt es ein weiteres Medium: das Internet. Mit Hilfe der Sprache *Java* lassen sich ebenfalls gut Animationen erstellen. Verfügt man über einen Internetzugang, so können die Animationen über WWW („World Wide Web") mit einem geeigneten Browser (z.B.: *Netscape, Internet Explorer,* etc.) auf unterschiedlichen Plattformen abgespielt werden.

Anhang A

Grammatiken

Wir listen hier die zugrundeliegenden Grammatiken der verwendeten Beispielsprachen LAMA und PASCAL auf, sowie die Syntax der *Eingabespezifikation* für die dynamische Animation der *Auflösung der Überladung*. Die gewählte Darstellungsart entspricht den Konventionen für Eingabefiles des Scannergenerators *LEX* (bzw. GNU-Klon *FLEX*) und des Parsergenerators *YACC* (bzw. GNU-Klon *BISON*). Nichtterminale sind *kursiv* dargestellt. Terminalsymbole sind **GROSS UND FETT** gedruckt. Das jeweilige Startsymbol ist ***fett und kursiv*** dargestellt. Die implementierten Scanner akzeptieren für die Terminalsymbole hinsichtlich der Groß- und Kleinschreibung verschiedene und auch abkürzende Schreibweisen. Kommentare sind in zwei Formen möglich, entweder in geschweiften Klammern „{…}“ oder in runden Klammern mit Stern „(*…*)“. Um die Lesbarkeit der Grammatiken zu verbessern, sind im Gegensatz zur gewählten Eingabekonvention die zweiteiligen Schlüsselsymbole, wie etwa „<>“, ebenfalls in Hochkommata gesetzt, z.B. ’<>’.

A.1 LAMA

A.1.1 Symbolklassen

BU	[a-zA-Z]
ZI	[0-9]
IDE	{**BU**} ({**BU**} | {**ZI**})*
INTEGER	([1-9]{**ZI**}*) | 0
REAL	{**INTEGER**}.{**ZI**}+
CHAR	’[^’]’

A.1.2 Grammatik

lama_programm : *expression*
;

let_list : *let_def* ’**;**’ *let_list*
| *let_def*
;

let_def : **IDE** ’==’ *expression*
;

var_list : **IDE** *var_list*
| **IDE**
;

arg_list : *expression arg_list*
| *expression*
;

list_list : *expression* ’**,**’ *list_list*
| *expression*
;

expression : *basiswert*
| **IDE**
| *expression* ’**+**’ *expression*
| *expression* ’**-**’ *expression*
| *expression* ’***’ *expression*
| *expression* ’/’ *expression*
| *expression* **OR** *expression*
| *expression* **AND** *expression*
| *expression* ’**=**’ *expression*
| *expression* ’<’ *expression*
| *expression* ’>’ *expression*
| *expression* ’<>’ *expression*
| *expression* ’<=’ *expression*
| *expression* ’>=’ *expression*
| **CONS** *expression expression*
| **PAIR** *expression expression*
| ’**[**’ *list_list* ’**]**’
| ’**[]**’
| ’**(**’ *op_un* ’**)**’
| ’**!**’ *var_list* ’**.**’ *expression* **END**
| ’**(**’ *expression arg_list* ’**)**’
| **IF** *expression* **THEN** *expression* **ELSE** *expression* **FI**
| **LETREC** *let_list* **IN** *expression* **END**
| **LET** *let_list* **IN** *expression* **END**
;

op_un : ’**+**’ *expression*
| ’**-**’ *expression*
| **NOT** *expression*
| **SUCC** *expression*
| **PRED** *expression*
| **NEG** *expression*
| **HD** *expression*
| **TL** *expression*

| **NULL** *expression*
| **FST** *expression*
| **SND** *expression*
;

basiswert : **INTEGER**
| **REAL**
| **CHAR**
| **TRUE**
| **FALSE**
;

A.2 PASCAL

A.2.1 Symbolklassen

BU [a-zA-Z]
ZI [0-9]
IDE {**BU**} ({**BU**} | {**ZI**})*
INTEGER ([1-9]{**ZI**}*) | **0**
REAL {**INTEGER**}.{**ZI**}+
STRING ’[^’]*’

CHAR_TYPE „char“ oder „character“ Typbezeichnung
BOOL_TYPE „bool“ oder „boolean“ Typbezeichnung
INT_TYPE „int“ oder „integer“ Typbezeichnung
REAL_TYPE „real“ Typbezeichnung

A.2.2 Grammatik

pascal_program : **PROGRAM IDE** ’**;**’ *block* ’**.**’
;

block : *const_decl*
type_decl
var_decl
p_f_decl
BEGIN *stat_list* **END**
;

const_decl :
| **CONST** *const_list* ’**;**’
;

const_list : *const_list* ’**;**’ *const_def*
| *const_def*
;

```
const_def       : IDE '=' constant
                ;

type_decl       :
                | TYPE type_list ';'
                ;

type_list       : type_list ';' type_def
                | type_def
                ;

type_def        : IDE '=' type
                ;

var_decl        :
                | VAR var_list ';'
                ;

var_list        : var_list ';' var_def
                | var_def
                ;

var_def         : id_list ':' type
                ;

id_list         : IDE ',' id_list
                | IDE
                ;

p_f_decl        :
                | p_f_list ';'
                ;

p_f_list        : p_f_list ';' p_f_def
                | p_f_def
                ;

p_f_def         : PROC IDE parameters ';' block
                | FUNC IDE parameters ':' type_ide ';' block
                ;

stat_list       : statement ';' stat_list
                | statement
                ;

parameters      :
                | '(' parameter_list ')'
                ;
```

parameter_list : *parameter* '**;**' *parameter_list*
| *parameter*
;

parameter : *id_list* '**:**' *type*
| **FUNC** *id_list* '**:**' *type*
| **VAR** *id_list* '**:**' *type*
| **PROC** *id_list*
;

statement : *variable* ':=' *expression*
| **BEGIN** *stat_list* **END**
| **IDE** *akt_parameter*
| **IF** *expression* **THEN** *statement*
| **IF** *expression* **THEN** *statement* **ELSE** *statement*
| **CASE** *expression* **OF** *case_block* **END**
| **WHILE** *expression* **DO** *statement*
| **REPEAT** *stat_list* **UNTIL** *expression*
| **FOR IDE** ':=' *expression* **TO** *expression* **DO** *statement*
| **FOR IDE** ':=' *expression* **DOWNTO** *expression* **DO** *statement*
;

akt_parameter :
| '**(**' *expr_list* '**)**'
;

expr_list : *expression* '**,**' *expr_list*
| *expression*
;

case_block : *case_body* '**;**' *case_block*
| *case_body*
;

case_body : **INTEGER** '**:**' *statement*
;

expression : *simple_expr*
| *expression* '=' *expression*
| *expression* '<' *expression*
| *expression* '>' *expression*
| *expression* '<=' *expression*
| *expression* '>=' *expression*
| *expression* '<>' *expression*
| *variable*
| **IDE** '**(**' *expr_list* '**)**'
| '**(**' *expression* '**)**'
| *expression* '+' *expression*
| '+' *expression*
| *expression* '-' *expression*

| '-' *expression*
| *expression* **OR** *expression*
| *expression* **AND** *expression*
| **NOT** *expression*
| *expression* '*****' *expression*
| *expression* '/' *expression*
| *expression* **MOD** *expression*
;

simple_expr : **INTEGER**
| **STRING**
| **REAL**
| *boolean*
;

variable : **IDE**
| **IDE** *design_list*
;

design_list : *design design_list*
| *design*
;

design : '**[**' *expr_list* '**]**'
| '**.**' **IDE**
| '^'
;

type : *type_ide*
| '^'*type_ide*
| **ARRAY** '**[**' *dim_list* '**]**' **OF** *type*
| **RECORD** *field_list* **END**
;

dim_list : *dim* '**,**' *dim_list*
| *dim*
;

dim : *bound* '**..**' *bound*
;

field_list : *field* '**;**' *field_list*
| *field*
;

field : *id_list* '**:**' *type*
;

bound : **IDE**
| **INTEGER**

| ’+’ **INTEGER**
| ’-’ **INTEGER**
;

constant : **IDE**
| **REAL**
| ’+’ **REAL**
| ’-’ **REAL**
| **INTEGER**
| ’+’ **INTEGER**
| ’-’ **INTEGER**
| *boolean*
| **STRING**
;

boolean : **TRUE**
| **FALSE**
;

type_ide : **IDE**
| *types*
;

types : **CHAR_TYPE**
| **BOOL_TYPE**
| **INT_TYPE**
| **REAL_TYPE**
;

A.3 Eingabesyntax der Überladungsspezifikation

A.3.1 Symbolklassen

BU	[a-zA-Z]
ZI	[0-9]
IDE	{**BU**} ({**BU**} | {**ZI**})*
INTEGER	([1-9]{**ZI**}*) | 0
REAL	{**INTEGER**}.{**ZI**}+
CHAR	’[^\’]’

CHAR_TYPE	„char“ oder „character“ Typbezeichnung
BOOL_TYPE	„bool“ oder „boolean“ Typbezeichnung
INT_TYPE	„int“ oder „integer“ Typbezeichnung
REAL_TYPE	„real“ Typbezeichnung

A.3.2 Grammatik

overload_spec : *operator_part*
expression_part
type_part
;

operator_part : **OPS_BEGIN** *operator_types* **OPS_END** (Operator-Regeln)
;

operator_types : *operator* '**;**' *operator_types*
| *operator* '**;**'
;

operator : *op_id* '**:**' *type_list*
;

op_id : **IDE**
| '**=**'
| '**<**'
| '**>**'
| '**<>**'
| '**>=**'
| '**<=**'
| **OR**
| **NOT**
| **AND**
| '**%**'
| '**+**'
| '**-**'
| '*****'
| '**/**'
;

type_list : *type* '**,**' *type_list*
| *type*
;

type : *par_type_list* '**->**' *types*
;

par_type_list : *types* '**x**' *par_type_list*
| *types*
;

types : **CHAR_TYPE**
| **BOOL_TYPE**
| **INT_TYPE**
| **REAL_TYPE**

```
                ;

type_part       : TYPE_BEGIN types TYPE_END          (Kontexttyp-Regeln)
                ;

expression_part : EXPR_BEGIN expression EXPR_END     (Ausdruck-Regeln)
                ;

expr_list       : expression ',' expr_list
                | expression
                ;

expression      : simple_expr
                | expression '=' expression
                | expression '<' expression
                | expression '>' expression
                | expression '<=' expression
                | expression '>=' expression
                | expression '<>' expression
                | IDE
                | IDE '(' expr_list ')'
                | '(' expression ')'
                | expression '+' expression
                | '+' expression
                | expression '-' expression
                | '-' expression
                | expression OR expression
                | expression AND expression
                | NOT expression
                | expression '*' expression
                | expression '/' expression
                | expression '%' expression
                ;

simple_expr     : INTEGER
                | CHAR
                | REAL
                | boolean
                ;

boolean         : TRUE
                | FALSE
                ;
```

Anhang B

Animierte Algorithmen

Alle hier gedruckten Algorithmen laufen auf der Darstellung der Eingabeprogramme als abstrakte Syntaxbäume ab. Um sie zu formulieren, führen wir die folgende Notation ein: Sei *k* ein Knoten (Datentyp: *node*) des abstrakten Syntaxbaums, dann beschreibt

#*descs*(*k*)	die Zahl der Kindknoten von *k*,
symb(*k*)	das Symbol mit dem *k* markiert ist,
type(*k*)	den mit *symb*(*k*) assoziierten Typ,
k.*i*	das *i*-te Kind von *k*.

B.1 Implementierung des Deklarations-Analysators

Aus Gründen der Einfachheit werden für den hier vorgestellten Algorithmus ADA-Gültigkeitsregeln zugrundegelegt. Diese drücken sich darin aus, daß im *decl*-Fall der **case**-Anweisung von *analyze_decl* erst alle Deklarationen rekursiv abgearbeitet werden, bevor die lokalen Deklarationen eingetragen werden. Die Gültigkeitsregeln von PASCAL (vor allem die Möglichkeit, Bezeichner rekursiv zu definieren) fordern dagegen zuerst den Eintrag aller Bezeichner in die Symboltabelle und dann erst die Analyse aller Unterbäume. Für jeden Block ist dann eine Menge von allen lokal angewandten Bezeichnern zu bilden. Trifft der Deklarations-Analysator auf eine Deklaration, so testet er für jeden dort deklarierten Bezeichner, ob er nicht schon in dieser Menge enthalten ist. Dieses Verfahren hat jedoch den Nachteil, daß manche angewandten Vorkommen von Bezeichnern in inkorrekten Programmen nicht richtig identifiziert werden. Der Algorithmus zur Identifizierung der Bezeichner (Deklarations-Analysator) ist in Abschnitt 3.3.1 erklärt:

```
proc analyze_decl (k: node);

proc analyze_subtrees (root: node);
begin
   for i := 1 to #descs(root) do
      analyze_decl (root.i)
   od
end;

begin
   case symb(k) of
```

```
   block:      begin
                  enter_block;
                  analyze_subtrees (k);
                  exit_block
               end;
   decl:       begin
                  analyze_subtrees (k);
                  foreach hier dekl. Bezeichner id do
                     enter_id (id, ↑k)
                  od
               end;
   appl_id:    speichere search_id (id) an k;
   otherwise:  if k kein Blatt then analyze_subtrees (k) fi
   od
end
```

Die Hauptfunktion *analyze_decl* verwendet mehrere Operationen zur Verwaltung einer Symboltabelle. Deren Implementierung sei wie folgt realisiert: *enter_block* vermerkt das Öffnen eines neuen Blocks. Die Prozedur *create_symb_table* kreiert eine leere Symboltabelle. *exit_block* setzt die Symboltabelle auf den Stand zurück, den sie vor dem letzten *enter_block* hatte:

```
proc enter_block;
begin
   if Schachtelungstiefe ist 0
   then create_symb_table
   fi;
   kellere neuen Eintrag für den neuen Block
end;

proc create_symb_table;
begin
   kreiere leeren Keller von Blockeinträgen
end;

proc exit_block;
begin
   foreach Deklarationseintrag des aktuellen Block do
      lösche Eintrag
   od;
   entferne Blockeintrag aus dem Keller
end
```

Die nächste Gruppe von Operationen bzw. Funktionen beschäftigt sich mit den gefundenen Bezeichnern direkt. *enter_id* fügt einen Eintrag für den Bezeichner *id* in die Symboltabelle ein. Dieser enthält den Verweis auf seine Deklarationsstelle, die in *decl_ptr* übergeben wird. *search_id* sucht das definierende Vorkommen zu *id* und gibt den Verweis auf die Deklarationsstelle zurück, wenn er existiert. Beide Routinen arbeitet relativ zum aktuellen Block:

```
proc enter_id (id: idno; decl_ptr: ↑node);
begin
   if exist. bereits ein Eintrag für id in diesem Block
   then error („Doppeldeklaration“)
   fi;
   kreiere neuen Eintrag mit decl_ptr und Nr. des akt. Blocks;
   füge diesen Eintrag hinten an die lineare Liste für id an;
   füge diesen Eintrag hinten an die lineare Liste für diesen Block an
end;

func search_id (id: idno) : ↑node;
begin
   if Zeile für id ist leer
   then error („undeklarierter Bezeichner“)
   else return (Wert des decl-Feldes aus erstem Eintrag in Zeile id)
   fi
end
```

B.2 Implementierung des Typkonsistenz-Analysators

Der Algorithmus zur Überprüfung der Typkonsistenz kontrolliert, ob das Quellprogramm die Typregeln der verwendeten Programmiersprache erfüllt. Weiterhin löst er die eventuelle (triviale) Überladung eingebauter Operatoren (z.B. die arithmetischen Operatoren) auf. Dazu verwendet er einen *bottom up*-Pass über einen Ausdrucksbaum, siehe Abschnitt 3.3.2. Bevor die Hauptprozedur *analyze_type_consist* aufgerufen wird, ist aus dem Kontext des zu untersuchenden Ausdrucks ein Kontexttyp (*a_priori_type*) zu errechnen:

```
proc analyze_type_consist (root: node, a_priori_type: type);

func analyze_type (k: node) : type;
begin
   case symb(k) of
   const:      begin
                  return type(k)
               end;
   appl_id:    begin
                  if id ist Operand
                  then return type(k)
                  else error („Modus-Fehler“)
                  fi
               end;
   operator:   begin
                  for i := 1 to #descs(k) do
                     t_i := analyze_type (k.i)
                  od;
                  if Tabelle von symb(k) einelementig
                  then  if symb(k) hat den Typ t_1 × … × t_n → t
                        then return t
                        else error („Typfehler bei symb(k)“)
```

```
                    fi
                else if exist. Operator in Tabelle von symb(k) mit Typ t1 × … × tn → t
                    then trage diesen Operator anstatt symb(k) in AST ein;
                         return t
                    else error („Typfehler bei symb(k)“)
                    fi
                fi
            end
    od
end;

begin
    if a_priori_type = no_type
    then analyze_type (root)
    else if a_priori_type <> analyze_type (root)
        then if Typkonsistenz nicht durch Typanpassung erreicht werden kann or
                die Sprache keine Typanpassung erlaubt
             then error („Falscher Kontexttyp“)
             fi
        fi
    fi
end
```

B.3 Algorithmus zur Auflösung der Überladung

Der Auflösungsalgorithmus benutzt zwei Läufe über einen Ausdrucksbaum, zuerst *bottom up* und dann *top down* (siehe Abschnitt 3.4.1). Um die Formulierung des Algorithmus zu vereinfachen, führen wir folgende Notation ein: An jedem Knoten k des zugrundeliegenden abstrakten Syntaxbaums erhalten wir über

vis(k)	Menge der an k sichtbaren Definitionen von *symb*(k),
ops(k)	die Menge der augenblicklichen Kandidaten für das überladene Symbol *symb*(k).

Weiterhin haben wir für jedes definierende Vorkommen eines überladenen Symbols *op* mit dem Typ $t_1 \times \ldots \times t_m \rightarrow t$

rank(*op*)	=	m
res_typ(*op*)	=	t
par_typ(*op*, i)	=	t_i ($1 \le i \le m$).

Die beiden letzteren seien auf Mengen von Operatoren erweiterbar. Bevor der Auflösungsalgorithmus durch die Routine *resolve_overloading* aufgerufen wird, sei für den Ausdruck, in welchem die Überladung von Operatoren aufgelöst werden soll, der Kontexttyp (*a_priori_type*) berechnet. Die Hilfsfunktion *pot_res_types* liefert alle potentielle Typen des Resultats der Eingabeoperation *symb*(k). *act_par_types* liefert die Menge aller i-ten Parametertypen der augenblicklichen Kandidaten für die Eingabeoperation *symb*(k). Zu Anfang assoziiert *init_ops* mit jedem Knoten eine Menge aller möglichen Definitionen des Operators:

```
proc resolve_overloading (root: node, a_priori_type: type);

func pot_res_types (k: node) : set of type;
begin
   return {res_typ (op) | op ∈ ops(k)}
end;

func act_par_types (k: node, i: integer) : set of type;
begin
   return {par_typ (op, i) | op ∈ ops(k)}
end;

proc init_ops;
begin
   foreach k
      ops(k) := {op | op ∈ vis(k) und rank(op) = #descs(k)}
   od;
   ops(root) := {op ∈ ops(root) | res_typ(op) = a_priori_type}
end;

proc bottom_up_elim (k: node);
begin
   for i := 1 to #descs(k) do
      bottom_up_elim (k.i);
      ops(k) := ops(k) – {op ∈ ops(k) | par_typ(op, i) ∉ pot_res_types (k.i)}
   od
end;

proc top_down_elim (k: node);
begin
   for i := 1 to #descs(k) do
      ops(k.i) := ops(k.i) – {op ∈ ops(k.i) | res_typ(op) ∉ act_par_types (k, i)};
      top_down_elim (k.i)
   od
end;

begin
   init_ops;
   bottom_up_elim (root);
   top_down_elim (root);
   prüfe, ob jetzt alle ops-Mengen einelementig sind, sonst Fehlermeldung
end
```

B.4 Typinferenzalgorithmus

Dieser Algorithmus stammt aus dem Buch [Rea89]. Er wurde dort in der funktionalen Sprache *Standard ML* (*SML*) vorgestellt und für die *Animation der semantischen Analyse* in C übersetzt und erweitert. Hier ist er in einem PASCAL-ähnlichen Code angegeben und basiert auf dem in Abschnitt 3.5.3.3 gezeigten Typinferenzsystem. Aus Gründen der Über-

sichtlichkeit enthält der Typinferenzalgorithmus keine Regeln für LAMA-Konstrukte der Form ($\lambda v_1 \ldots v_n.e$) und ($e_1 \ldots e_n$), sondern nur für deren einfache Formen ($\lambda v.e$) und ($e_1 e_2$). Seine Implementierung in der *Animation der semantischen Analyse* kann jedoch mit diesen komplexeren Ausdrücken umgehen.

Zunächst geben wir zwei Funktionen an, welche die Typen bzw. Typterme *t* (Datentyp: *typeExp*) direkt manipulieren. Die Datenstruktur zur Implementierung eines Typterms ist ein Baum. Ein Knoten dieses Baumes ist entweder ein null-, ein- oder zweistelliger Typoperator, d.h. ein Operator aus {**int**, **bool**, **real**, **char**, **list**, $\times$, $\rightarrow$}, oder eine Typvariable. Dabei entspricht die Anzahl der Kinder eines Typknotens seiner Stelligkeit. Wir führen folgende Notation im Zusammenhang mit Typtermen ein. Sei *t* ein Typ, dann ist

rank(*t*)	die Anzahl der Kinder der Wurzel des Typbaums von *t*,
part_typ(*t*, *i*)	das *i*-te Kind der Wurzel des Typbaums von *t*,
op(*t*)	der durch die Wurzel symbolisierte Operator bzw. Typvariable.

Die Funktion *occurs* testet, ob in einem beliebigen Typterm eine bestimmte Typvariable (Datentyp: *typeVar*) enthalten ist („occurs check“). Die zweite Funktion *isTypeVar* überprüft, ob es sich bei einem gegebenen Typ um eine Typvariable handelt:

func *occurs* (α: *typeVar*, *t*: *typeExp*) : **bool**;
begin
  if α ist in *t* enthalten
  then return *true*
  else return *false*
  fi
end;

func *isTypeVar* (*t*: *typeExp*) : **bool**;
begin
  if *t* ist eine Typvariable
  then return *true*
  else return *false*
  fi
end

Die folgenden Funktionen bearbeiten Ersetzungen (Substitutionen) (Datentyp: *sub*). Eine Ersetzung ist als ein Konstrukt der Form $\sigma = [t_1/\alpha_1, \ldots, t_n/\alpha_n]$ repräsentiert. Die Funktion *emptySub* erstellt eine leere Substitution, d.h. eine identische Ersetzung einer Typvariable auf sich selbst. *isEmptySub* testet, ob es sich bei einer gegebenen Ersetzung um eine leere Ersetzung handelt. Die Funktion *newSub* erstellt aus einer Typvariablen und einem Typterm eine neue Substitution. *substitute* wendet eine Ersetzung auf alle in einem Typterm auftretenden Variablen an. Das Ergebnis dieser Operation ist ein neuer Typterm. Schließlich führt *compose* eine Verknüpfung zweier Substitutionen σ_1, σ_2 durch und zwar so, daß zuerst σ_2 und dann σ_1 auf das Resultat angewandt wird:

func *emptySub* (α: *typevar*) : *sub*;
begin
  return die leere Ersetzung [α/α]
end;

```
func isEmptySub (σ: sub) : bool;
begin
   if σ ist eine leere Ersetzung
   then return true
   else return false
   fi
end;

func newSub (α: typeVar, t: typeExp) : sub;
begin
   return [t/α]
end;

func substitute (σ: sub, t: typeExp) : typeExp;
begin
   if isTypeVar (t)
   then  if es existiert eine passende Ersetzung für t in σ
         then wende die Ersetzung σ auf t an
         fi
   else  for i := 1 to rank(t) do
            part_typ(t, i) := substitute (σ, part_typ(t, i))
         od
   fi;
   return t
end;

func compose (σ1: sub, σ2: sub) : sub;
begin
   return [σ1, σ2]
end
```

Der Unifikator besteht aus zwei Funktionen: Die Funktion *unify* unifiziert zwei Typterme und liefert eine allgemeinste Ersetzung zurück. Sie verwendet u.a. zur Behandlung von Listentypen die zweite Funktion *unifyall*:

```
func unify (t1: typeExp, t2: typeExp) : sub;
begin
   if isTypeVar (t1)
   then  if occurs (t1, t2)
         then  if isTypeVar (t2)
               then return emptySub (t2)
               else error („Unendlicher Typ“)
               fi
         else  return newSub (t1, t2)
         fi
   else  if isTypeVar (t2)
         then return unify (t2, t1)
         fi
   fi;
```

```
   if op(t1) = op(t2)
   then return unifyall (t1, t2)
   else error („Typfehler, da op(t1) ≠ op(t2)“)
   fi
end;

func unifyall (t1: typeExp, t2: typeExp) : sub;
begin
   if t1 und t2 sind Typvariablen oder einfache Typen
   then return emptySub (0)
   else  if rank(t1) = rank(t2)
         then  σ1 := unify (t1.1, t2.1);
               if rank(t1) = 2
               then  t'1 := substitute (σ1, t1.2);
                     t'2 := substitute (σ1, t2.2);
                     σ2 := unify (t'1, t'2);
                     return compose (σ2, σ1)
               else  if rank(t1) = 1
                     then return σ1
                     fi
               fi
         fi
   fi;
   error („Unifikationsfehler, da inkonsistente Anzahl der Typargumente für
         Listentypen“)
end
```

Mit Mengen von Annahmen (Datentyp: *assum*) werden die folgenden Funktionen assoziiert. Wir repräsentieren eine Menge von Annahmen (Umgebung) durch ein Paartupel $A = (pairs, nongen)$, wobei *pairs* die Form $[(v_1, t_1),\ldots,(v_n, t_n)]$ hat, und *nongen* genau die Typvariablen in den t_i spezifiziert, die nicht-generisch sind. *nongen* ist eine Liste von Typtermen (nicht nur Typvariablen), so daß alle in diesen Typtermen vorkommenden Typvariablen nicht-generisch sind. Dies vereinfacht das Hinzufügen einer neuen Annahme zu *A*. Für eine Menge von Annahmen *A* gelten die Notationen:

pairs(*A*)	die Menge der Annahmen *pairs* von *A*,
nongen(*A*)	die Liste der Typterme *nongen* von *A*,
$a_i \in pairs(A)$	eine Annahme (v_i, t_i),
node(a_i)	der Knoten *k* mit *symb*(*k*) = v_i,
type(a_i)	den Typ t_i der Annahme a_i.

Die erste Funktion *addAssum* fügt eine neue Annahme (*v*, *t*) in eine Umgebung *A* ein, wobei alle im Typ *t* enthaltenen Typvariablen als nicht-generisch angesehen werden. *addGenAssum* hat die gleiche Funktionalität wie *addAssum*, außer dem Unterschied, daß alle im Typ *t* enthaltenen Typvariablen als generisch angesehen werden. Die Funktion *lookup* liefert für einen Knoten den entsprechenden Typterm aus der Umgebung. *getNG* gibt eine Liste aller Typen aus, die nicht-generische Typvariablen enthalten. Schließlich erzeugt die Funktion *emptyAssum* eine leere Typumgebung. Darunter versteht man eine Typumgebung ohne Annahmen und ohne nicht-generischen Variablen. Alle im Verlauf der Typinferenz kreierten

Annahmen werden aus diesem Initialwert generiert, falls keine initiale Typumgebung angegeben wurde:

```
func addAssum (k: node, t: typeExp, A: assum) : assum;
begin
   füge das neue Paar (k, t) einer Annahme zu pairs(A) hinzu;
   füge den Typterm t zu nongen(A) hinzu;
   return A
end;

func addGenAssum (k: node, t: typeExp, A: assum) : assum;
begin
   füge das neue Paar (k, t) einer Annahme zu pairs(A) hinzu;
   return A
end;

func lookup (k: node, A: assum) : typeExp;
begin
   foreach in pairs(A) enthaltenen Annahmen a_i do
      if node(a_i) = k
      then return type(a_i)
      fi
   do;
   return no_type
end;

func getNG (A: assum) : list of typeExp;
begin
   return nongen(A)
end;

func emptyAssum () : assum;
begin
   return neue Umgebung A
end
```

Der Algorithmus benutzt Zustände (Datentyp: *state*), die die aktuelle Situation beschreiben. Ein solcher Zustand *s* besteht aus einem Paartupel (σ, α), dessen erster Eintrag eine akkumulierte Ersetzung σ, der zweite Eintrag die nächste neue generische Typvariable α enthält. An einem Zustand *s* erhalten wir über

sub(*s*)	die akkumulierte Ersetzung σ von *s*,
var(*s*)	die nächste neue generische Typvariable α von *s*.

Die Funktion *newTypeVar* erzeugt aus einem gegebenen Zustand eine neue Typvariable und liefert den neuen Zustand zusammen mit der neuen Typvariable zurück. *extendSub* erweitert einen Zustand um eine neue Ersetzung (durch Verknüpfung der Ersetzungen). Um die aktuelle Ersetzung aus einem Zustand zu extrahieren, wird die Funktion *getSub* verwendet. *state0* erzeugt einen neuen leeren Zustand:

```
func newTypeVar (s: state) : tuple of typeExp and state;
begin
   var(s) := α, wobei α neue Typvariable;
   return (α, s)
end;

func extendSub (σ: sub, s: state) : state;
begin
   if ¬(isEmptySub (σ))
   then sub(s) := compose (σ, sub(s))
   fi;
   return s
end;

func getSub (s: state) : sub;
begin
   return sub(s)
end;

func state0 () : state;
begin
   sub(s) := emptySub (0);
   var(s) := initiale neue Typvariable;
   return s
end
```

Die nächste Gruppe von Funktionen bearbeitet die Typinstantiierung. *rawType* liefert den neuen Typterm, der sich durch die Anwendung der aktuellen Substitution auf den Typ der Umgebung ergibt, der sich auf den aktuellen Knoten bezieht. Die Funktion *generic* stellt fest, ob eine Typvariable in Abhängigkeit der aktuellen Umgebung und Ersetzung generisch ist oder nicht. *freshInst* ersetzt die in einem Typterm enthaltenen generischen Variablen durch neue und liefert den resultierenden Typterm und einen neuen Zustand zurück. *newTypeInst* verwendet die zuletzt erwähnten Funktionen, um den mit einem Knoten assoziierten Typ unter Berücksichtigung der Umgebung und des Zustandes eine neue Instanz zu geben:

```
func rawType (k: node, A: assum, s: state) : typeExp;
begin
   return substitute (getSub (s), lookup (k, A))
end;

func generic (A: assum, s: state, α: typeVar) : bool;
begin
   (t1,…,tm) := getNG (A);
   foreach ti in der Typenliste (t1,…,tm) do
      if occurs (α, substitute (getSub (s), ti)) = true
      then return false
      fi
   od;
```

```
   return true
end;

func freshInst (t: typeExp, A: assum, s: state) : tuple of typeExp and state;
begin
   foreach Typvariable α in t do
      if generic (A, s, α) = true
      then  (α', s) := newTypeVar (s);
            σ := [α'/α];
            t := substitute (σ, t)
      fi
   od;
   return (t, s)
end;

func newTypeInst (k: node, A: assum, s: state) : tuple of typeExp and state;
begin
   return freshInst (rawType (k, A, s), A, s)
end
```

Es folgt die Hauptanalysefunktion *analyse* des Typinferenzalgorithmus. Diese Funktion spiegelt das in Abschnitt 3.5.3.3 angegebene Typinferenzsystem wider. Sie traversiert den Syntaxbaum des LAMA-Ausdrucks rekursiv und berechnet für jeden Knoten in Abhängigkeit des aktuellen Zustands und der aktuellen Umgebung den entsprechenden Typterm und den neuen Zustand:

```
func analyse (k: node, A: assum, s: state) : tuple of typeExp and state;
begin
   case symb(k) of
   VAR:     return newTypeInst (k, A, s);
   CON:     return newTypeInst (k, A, s);
   APP:     begin                          (symb(k) = (e1e2))
               (t1, s1) := analyse (e1, A, s);
               (t2, s2) := analyse (e2, A, s1);
               (t3, s3) := newTypeVar (s2);
               σ := unify (t2 → t3, t1);
               s4 := extendSub (σ, s3);
               t4 := substitute (σ, t3);
               return (t4, s4)
            end;
   COND:    begin                                  (symb(k) = if e1 then e2 else e3)
               (t1, s1) := analyse (e1, A, s);
               σ1 := unify (t1, 'bool');
               (t2, s2) := analyse (e2, A, extendSub (σ1, s1));
               (t3, s3) := analyse (e3, A, s2);
               σ2 := unify (t2, t3);
               s4 := extendSub (σ2, s3);
               t4 := substitute (sub(s4), t3);
               return (t4, s4)
```

end;

LET: **begin** (*symb*(*k*) = **let** $v_1 == e_1;\ldots;v_n == e_n$ **in** e_0)
  $s_0 := s$; $s_1 := s$; $A_1 := A$;
  for $i := 1$ **to** n **do**
    $(t_i, s_i) :=$ *analyse* (e_i, A, s_{i-1});
    $A_1 :=$ *addGenAssum* (v_i, t_i, A_1)
  od;
  $(t_0, s_0) :=$ *analyse* (e_0, A_1, s_n);
  return (t_0, s_0)
end;

LETREC: **begin** (*symb*(*k*) = **letrec** $v_1 == e_1;\ldots;v_n == e_n$ **in** e_0)
  $s_1 := s$; $A_1 := A$;
  for $i := 1$ **to** n **do**
    $(t_1, s_1) :=$ *newTypeVar* (s_1);
    $A_1 :=$ *addAssum* (v_i, t_1, A_1);
  od;
  $A_2 := A$; $(t_2, s_2) := (t_1, s_1)$;
  for $i := 1$ **to** n **do**
    $(t_2, s_2) :=$ *analyse* (e_i, A_1, s_2);
    $\sigma :=$ *unify* $(t_2,$ *lookup* $(v_i, A_1))$;
    $s_2 :=$ *extendSub* (σ, s_2);
    $t_2 :=$ *substitute* (σ, t_2);
    $A_2 :=$ *addGenAssum* (v_i, t_2, A_2)
  od;
  $(t_0, s_0) :=$ *analyse* (e_0, A_2, s_2);
  return (t_0, s_0)
end;

ABS: **begin** (*symb*(*k*) = $\lambda v.e$)
  $(t_1, s_1) :=$ *newTypeVar* (s);
  $A_1 :=$ *addAssum* (v, t_1, A);
  $(t_2, s_2) :=$ *analyse* (e, A_1, s_1);
  $t_3 :=$ *substitute* $(sub(s_2), t_1) \rightarrow t_2$;
  return (t_3, s_2)
end;

OP_{UN}: **begin** (*symb*(*k*) = $(op_{un}\ e_1)$ und *type*$(op_{un}) = t_{.1} \rightarrow t_{.2}$)
  $(t_1, s_1) :=$ *newTypeInst* (k, A, s);
  $(t_2, s_2) :=$ *analyse* (e_1, A, s_1);
  $\sigma :=$ *unify* $(t_{1.1}, t_2)$;
  $s_3 :=$ *extendSub* (σ, s_2);
  $t_3 :=$ *substitute* $(\sigma, t_{1.2})$;
  return (t_3, s_3)
end;

OP_{BIN}: **begin** (*symb*(*k*) = $e_1\ op_{bin}\ e_2$ und *type*$(op_{bin}) = t_{.1} \rightarrow t_{.2} \rightarrow t_{.3}$)
  $(t_1, s_1) :=$ *newTypeInst* (k, A, s);
  $(t_2, s_2) :=$ *analyse* (e_1, A, s_1);
  $(t_3, s_3) :=$ *analyse* (e_2, A, s_2);
  $\sigma_1 :=$ *unify* $(t_{1.1}, t_2)$;
  $s_4 :=$ *extendSub* (σ_1, s_3);
  $\sigma_2 :=$ *unify* $(t_{1.2}, t_3)$;

```
                $s_5$ := extendSub ($\sigma_2$, $s_4$);
                $t_5$ := substitute (sub($s_5$), $t_{1.3}$);
                return ($t_5$, $s_5$)
            end;
   LIST:    begin                                   (symb(k) = [$e_1$,$e_2$,…,$e_n$])
                $s_0$ := s; $s_1$ := s;
                for i := 1 to n do
                    ($t_i$, $s_i$) := analyse ($e_i$, A, $s_{i-1}$);
                    if i > 1
                    then σ := unify ($t_{i-1}$, $t_i$);
                         $s_i$ := extendSub (σ, $s_i$);
                         $t_i$ := substitute (σ, $t_i$)
                    fi
                od;
                $t_n$ := list $t_n$;
                return ($t_n$, $s_n$)
            end;
   od
end
```

Die letzte Funktionsgruppe beinhaltet Funktionen, die den Typinferenzalgorithmus aufrufen und die initiale Typumgebung erstellen. *lookupInitial* testet, ob ein Operatorknoten schon in der initialen Typumgebung eingetragen ist. Sie wird von der Funktion *getInitialAssumption* aufgerufen, welche die initiale Typumgebung des betrachteten LAMA-Ausdrucks generiert. Die Hauptfunktion *type_checker* erhält als Eingabe den Syntaxbaum des LAMA-Ausdrucks, läßt die initiale Typumgebung durch *getInitialAssumption* generieren und ruft *analyse* auf:

```
func lookupInitial (k: node, A: assum) : bool;
begin
   if ∃ a ∈ pairs(A) mit k = node(a)
   then return true
   else return false
   fi
end;

func getInitialAssumption (k: node, A: assum) : assum;
begin
   if lookupInitial (k, A) = false
   then  t := erzeuge den initialen Typ von k, wenn er existiert;
         if ¬(t = no_type)
         then A := addGenAssum (k, t, A)
         fi
   fi;
   for i:= 1 to #descs(k) do
      A := getInitialAssumption (k.i, A)
   od;
   return A
end;
```

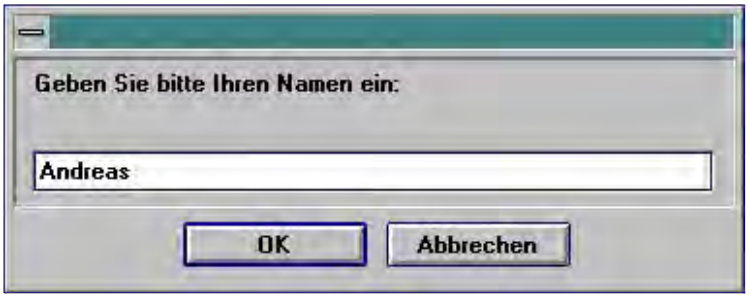
```
proc type_checker (root: node);
begin
    A := getInitialAssumption (root, emptyAssum ());
    type(root) := analyse (root, A, state0 ())
end
```

Anhang C

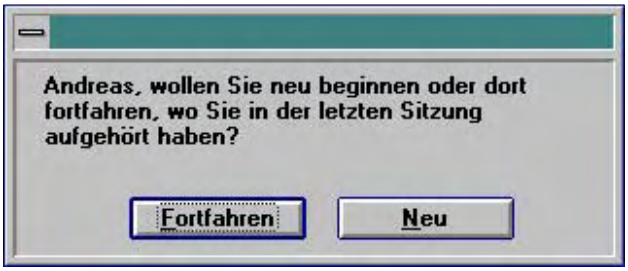
Beispiele für Animationen

Die in diesem Anhang ausgedruckten Bilder geben von den Visualisierungen und Animationen dieser Arbeit naturgemäß nur eine sehr unvollkommene Vorstellung. Sie spiegeln z.T. auch lediglich die Situation wider, in denen der/die BenutzerIn eine Animation einfrieren kann. Eine Animation in Bewegung kann in einem gedruckten Text nur sehr schlecht simuliert und dargestellt werden.

C.1 Visualisierungen aus der animierten Präsentation

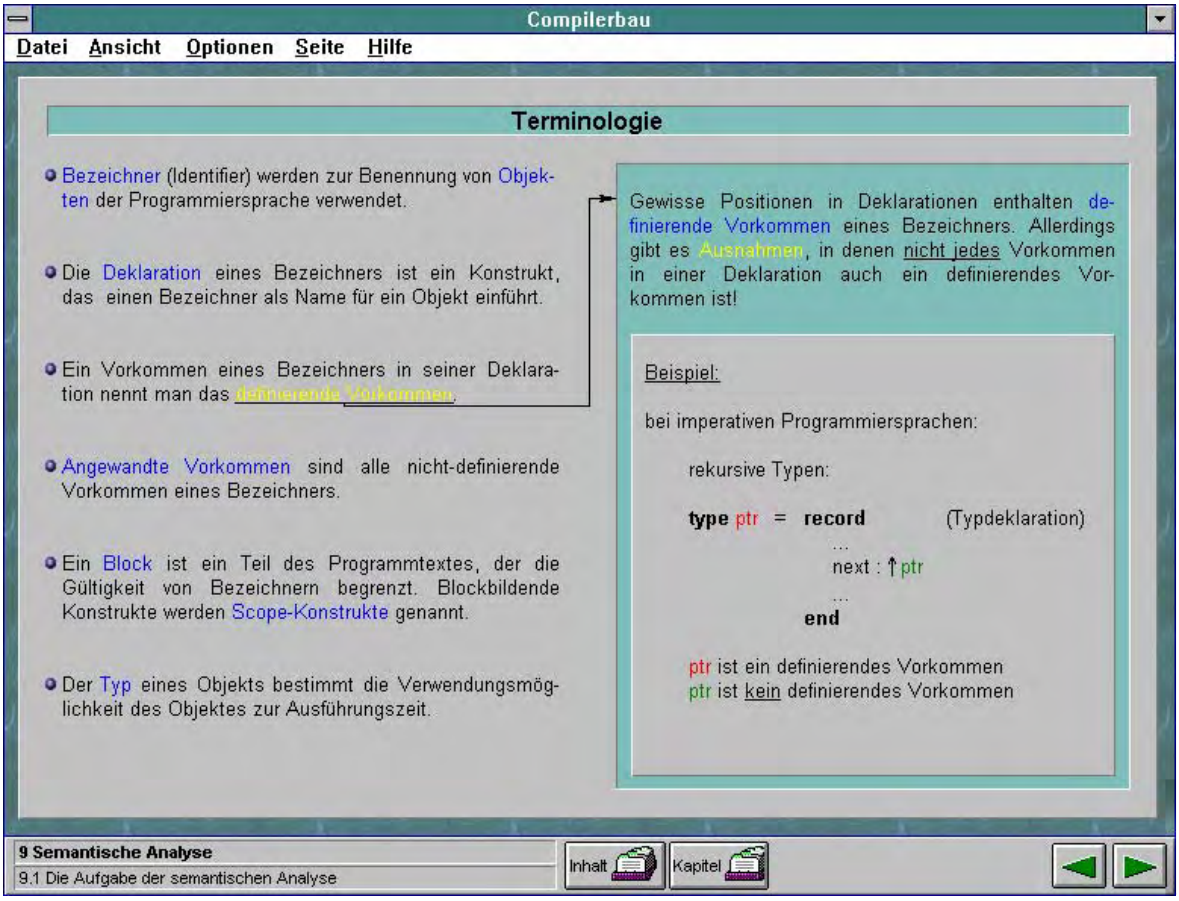


Zuerst wird der/die BenutzerIn nach dem Namen gefragt. Alle im Programm auftretenden Dialoge verwenden diesen Namen als persönliche Anrede.



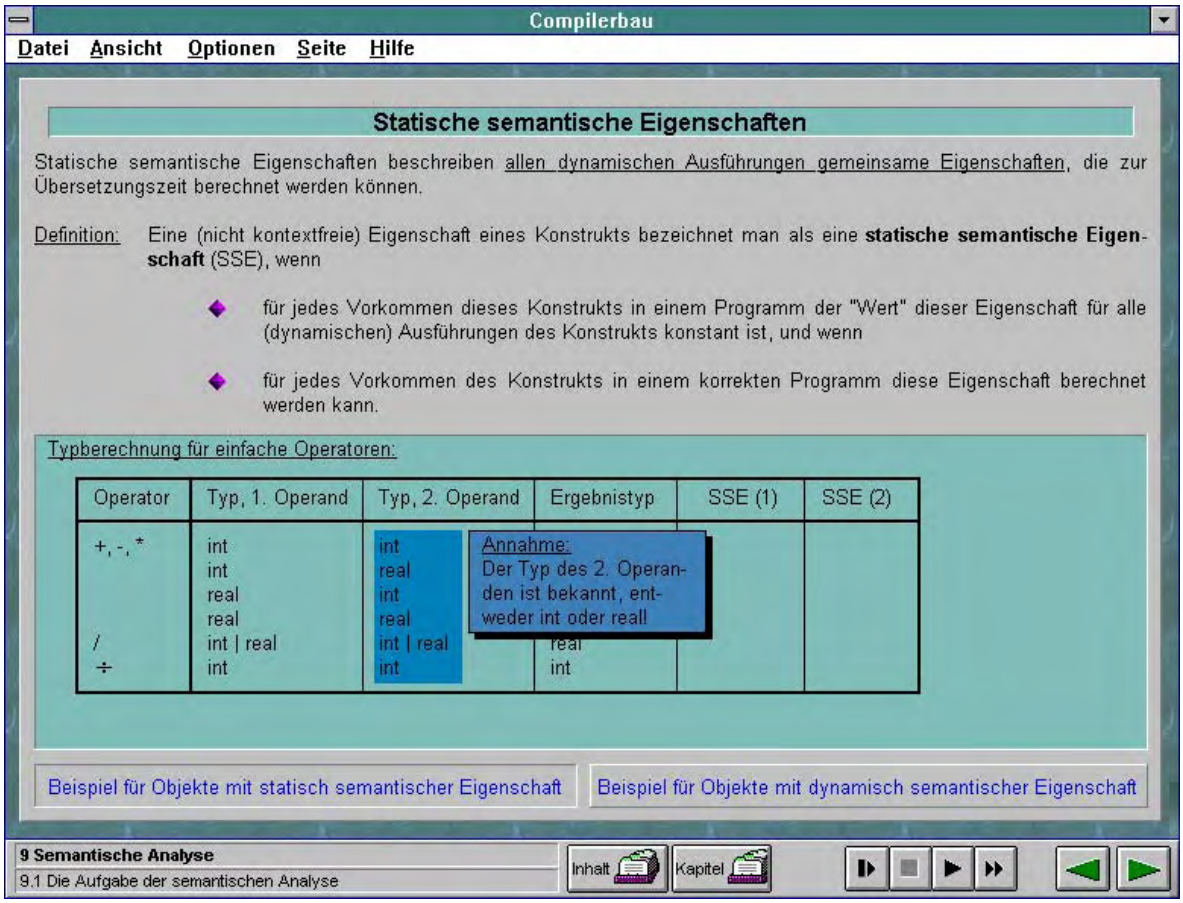
Ist der Name dem System bereits bekannt, so wird der/die AnwenderIn danach gefragt, ob das Programm an der zuletzt besuchten Seite fortfahren soll oder nicht. Wird *Neu* ausgewählt, dann zeigt das Programm die Seite mit dem Inhaltsverzeichnis an.

Abb. C.1.1: Startsequenz der *animierten Präsentation*



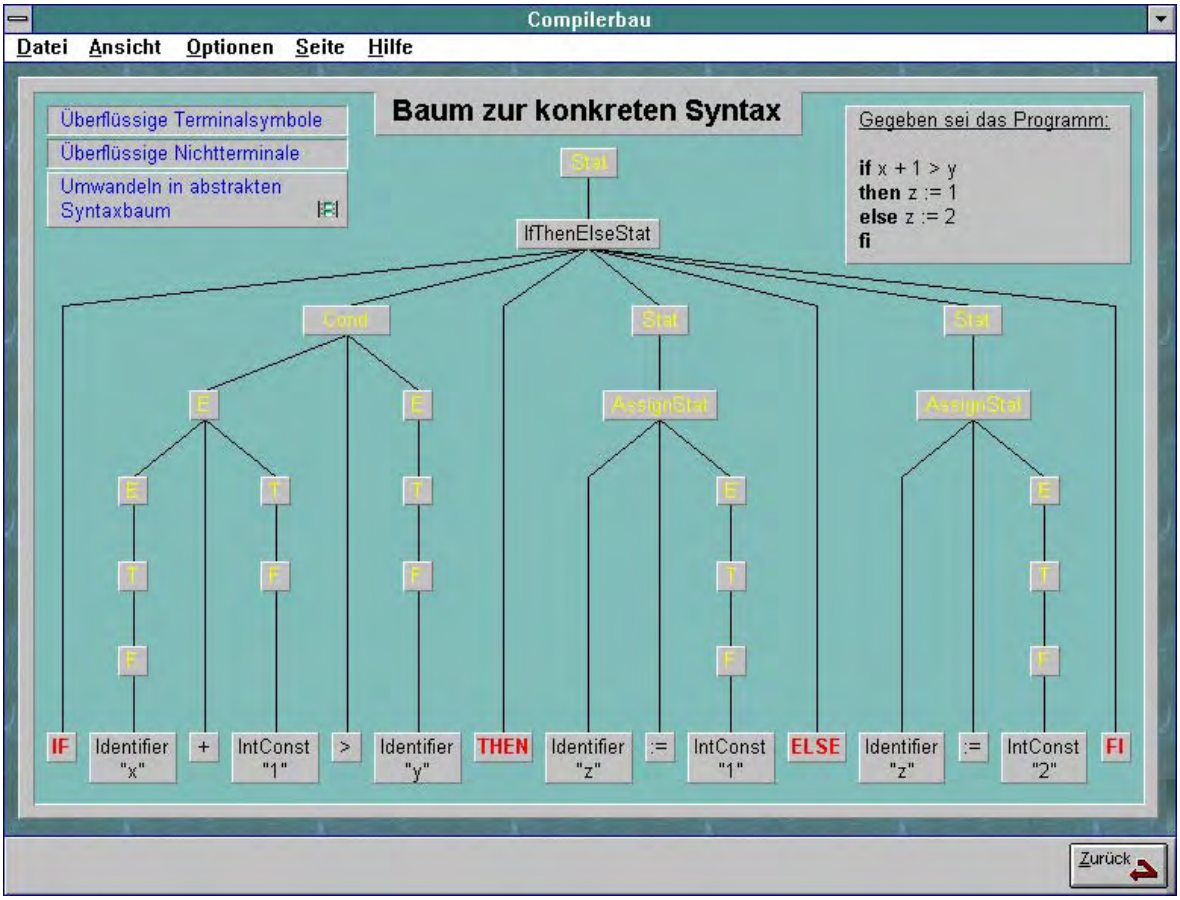
Auf der linken Hälfte der Seite sind die in der semantischen Analyse verwendeten Begriffe kurz definiert. Klickt man auf einen blau gefärbten Textteil, so erscheint auf der rechten Hälfte eine detailliertere Erklärung des gewählten Begriffs. Innerhalb dieses Bereichs lassen sich auch ein oder mehrere Beispiele zu dem ausgewählten Begriff anzeigen.

Abb. C.1.2: Terminologie zur semantischen Analyse



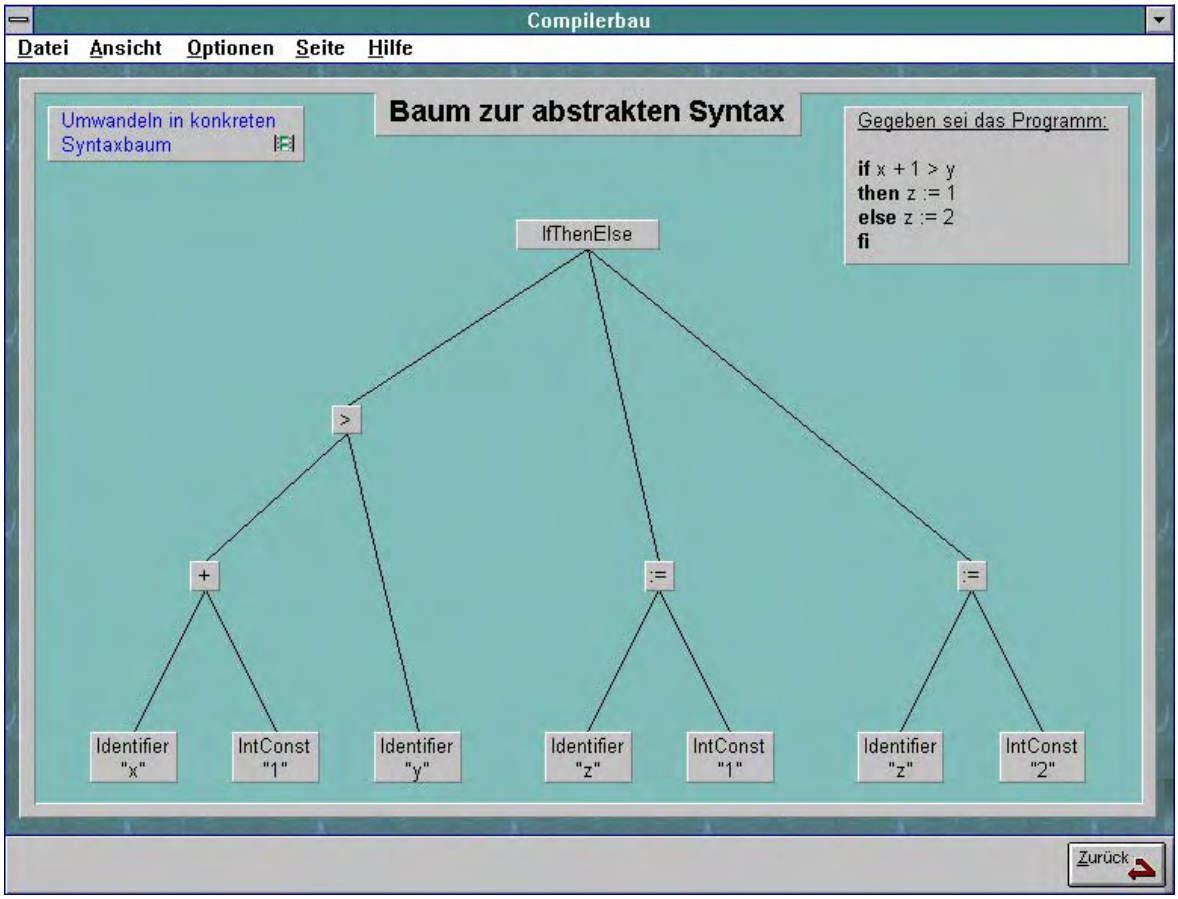
Zunächst wird der Begriff der *statischen semantischen Eigenschaften* im oberen Bereich der Seite definiert. Danach besteht die Möglichkeit, eine kleine Animation ablaufen zu lassen, die den Unterschied zwischen statischen und dynamischen semantischen Eigenschaften anhand der Typberechnung einfacher arithmetischer Operatoren verdeutlicht.

Abb. C.1.3: Statische semantische Eigenschaften



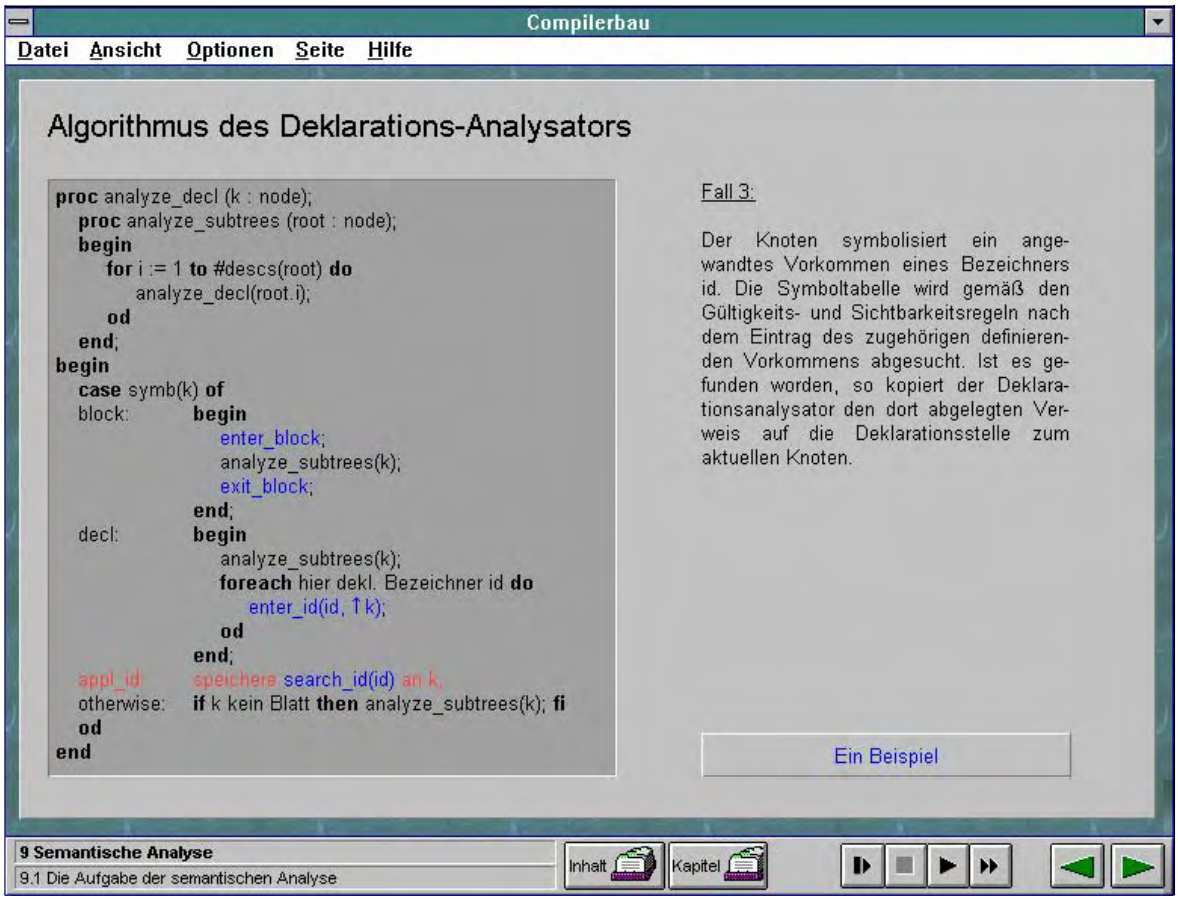
Gezeigt wird der konkrete Syntaxbaum zu dem rechts oben angegebenen Programmfragment. Für die semantische Analyse sind die rot gefärbten Terminalsymbole, sowie die gelb gefärbten Nichtterminale nicht mehr notwendig. Klickt man auf das Feld *Umwandeln in abstrakten Syntaxbaum*, so entfernt die *animierte Präsentation* die überflüssigen Knoten und wandelt den konkreten Syntaxbaum animiert in den abstrakten Syntaxbaum um (vgl. Abb. C.1.5).

Abb. C.1.4: Baum zur konkreten Syntax



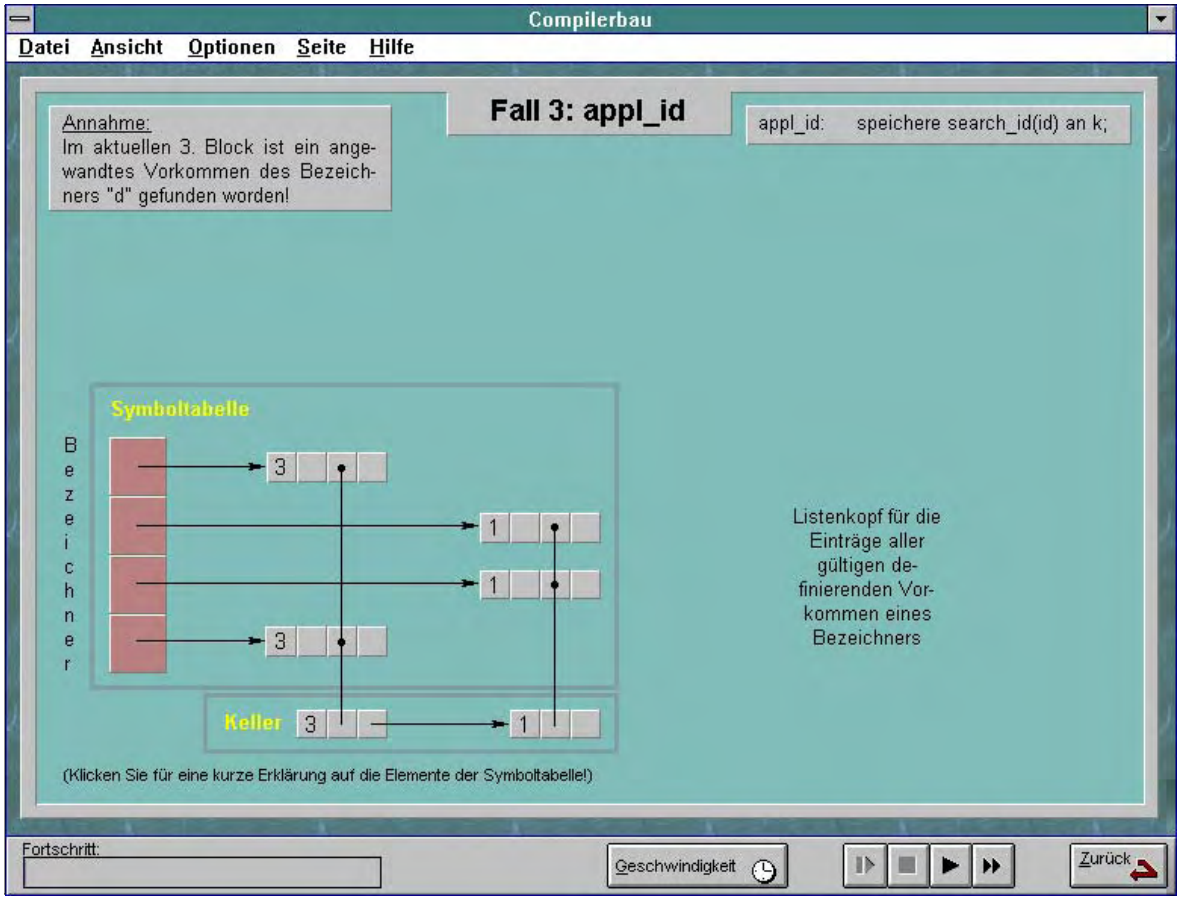
Aus dem Baum zur konkreten Syntax (siehe Abb. C.1.4) wurde dieser abstrakte Syntaxbaum erzeugt. Er läßt sich wieder in einen konkreten Syntaxbaum animiert zurückverwandeln, um die Unterschiede noch besser darzustellen.

Abb. C.1.5: Baum zur abstrakten Syntax



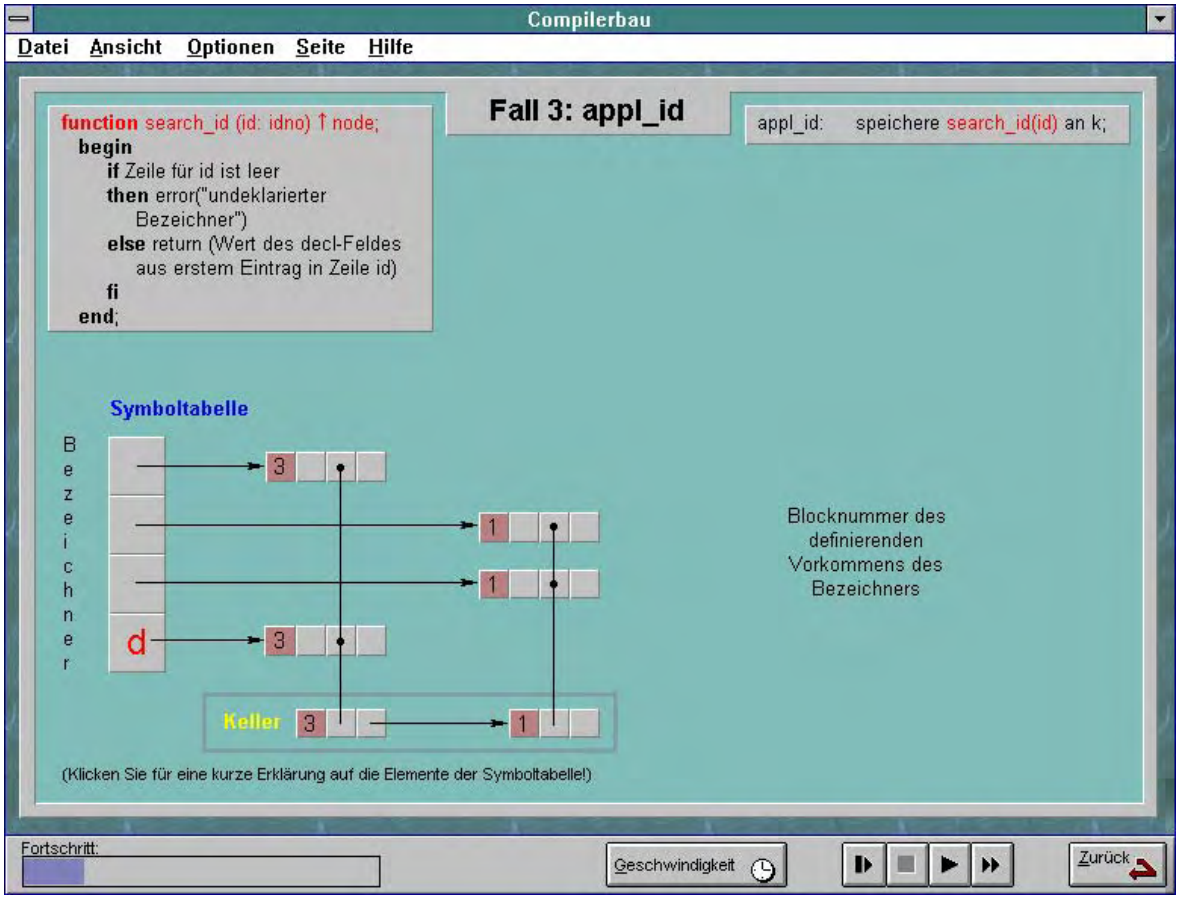
Der rot gefärbte Quelltext des Algorithmus kennzeichnet die aktuelle Stelle, die auf der rechten Hälfte der Seite erklärt wird. Mit den Steuerungsbuttons unten in der Kontrolleiste kann der Algorithmus komplett durchlaufen werden (Textanimation). Zu allen wichtigen Bestandteilen des Algorithmus lassen sich Beispielanimationen aufrufen, die ihrerseits wieder aus mehreren Seiten bestehen können. Die Abbildungen C.1.7 bis C.1.11 zeigen die wichtigsten Schritte einer solchen Beispielanimation.

Abb. C.1.6: Algorithmenanimation für den Deklarations-Analysator



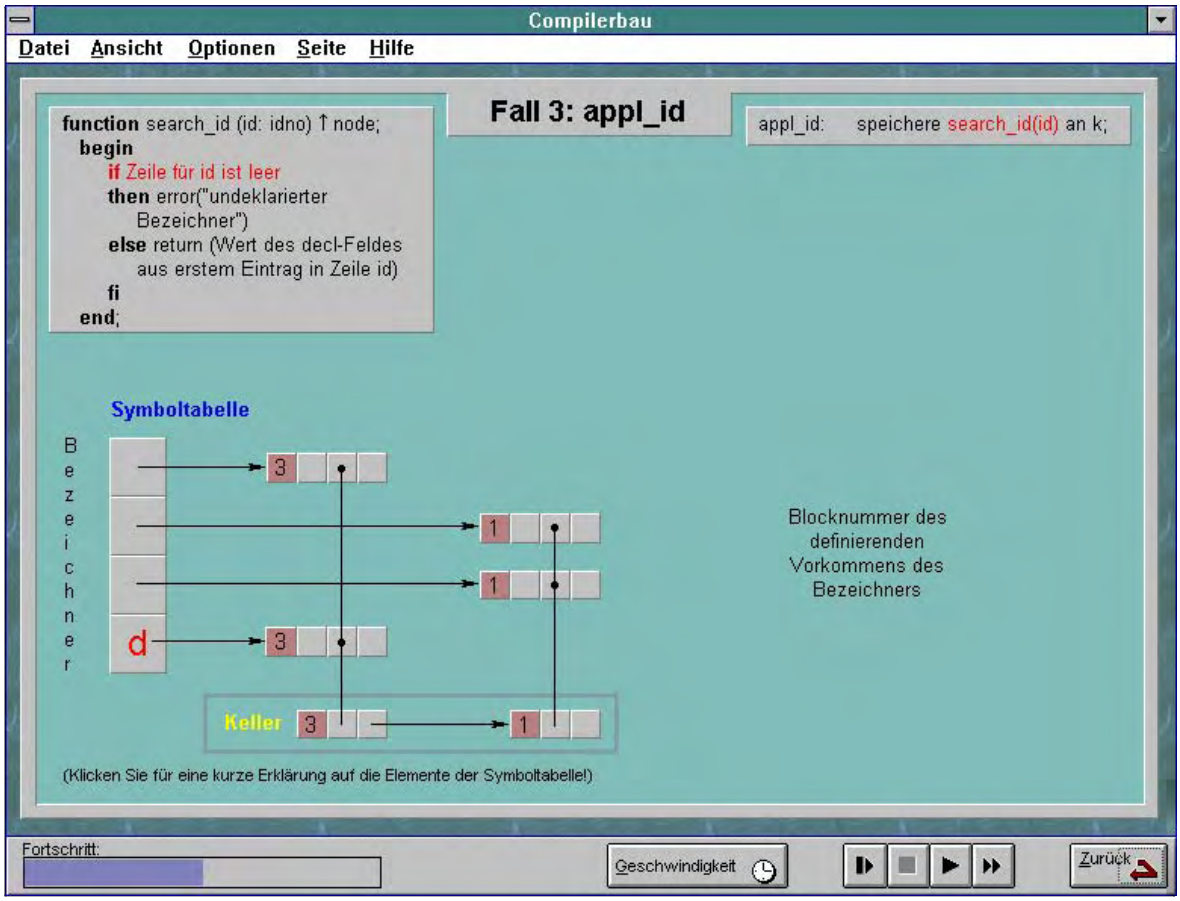
Rechts oben befindet sich die aktuelle Stelle im Quelltext zum Algorithmus für den Deklarations-Analysator. Man kann die Elemente der Symboltabelle anklicken, um eine kurze Erklärung zu dem ausgewählten Element zu erhalten. Es wird angenommen, daß im aktuellen dritten Block ein angewandtes Vorkommen eines Bezeichners „d“ gefunden wurde.

Abb. C.1.7: Erster Schritt der Beispielanimation zu einer Symboltabellenoperation



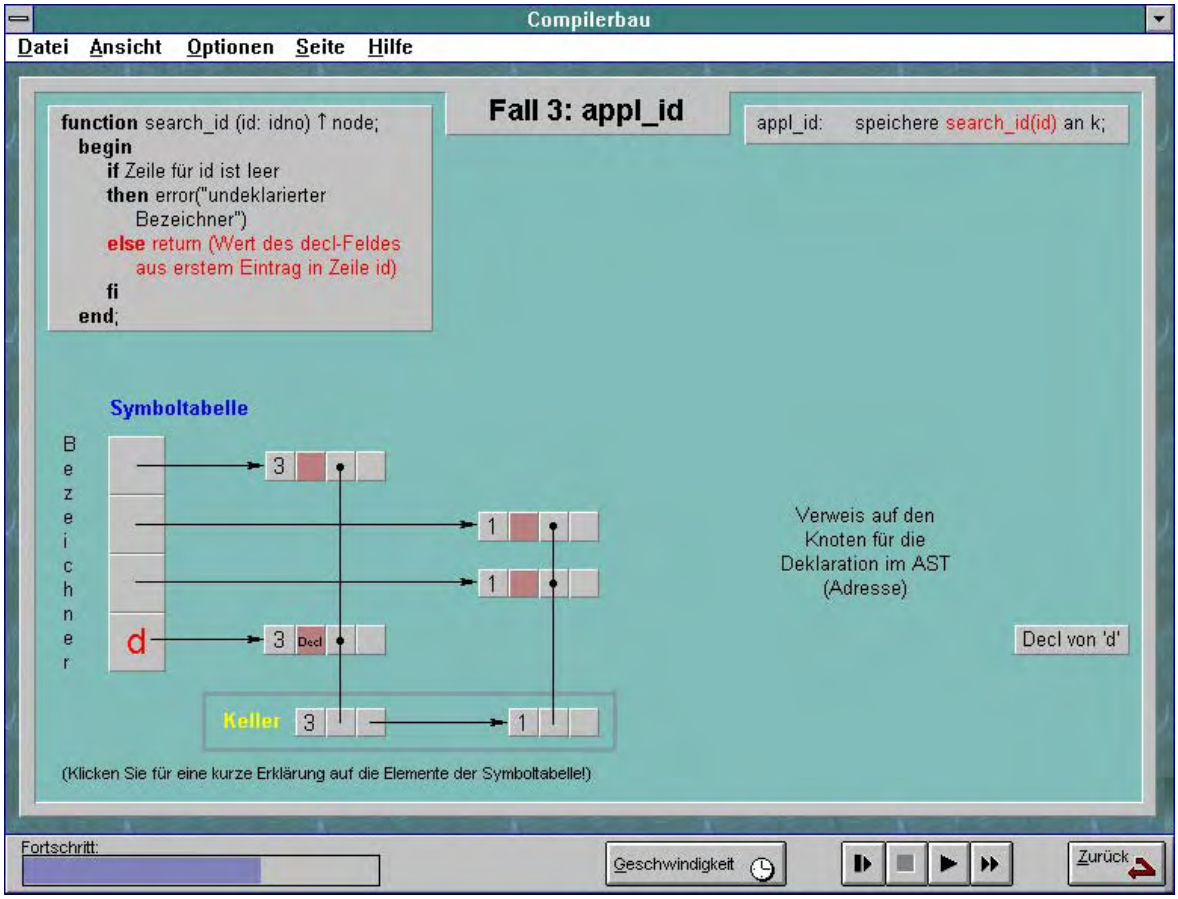
In der Quelltextzeile rechts oben ist ein Funktionsaufruf *search_id (id)* enthalten. Diese Funktion ist nun links oben zu sehen. Der/die AnwenderIn kann den Quelltext für diese Funktion zeilenweise abarbeiten lassen (Eine Zeile, die beispielsweise nur ein **begin** enthält, hat keine Animation zur Folge). Weiterhin hat der Autor auf die erste Zelle eines Eintrags geklickt, um sich deren Bedeutung anzeigen zu lassen.

Abb. C.1.8: Zweiter Schritt der Beispielanimation zu einer Symboltabellenoperation



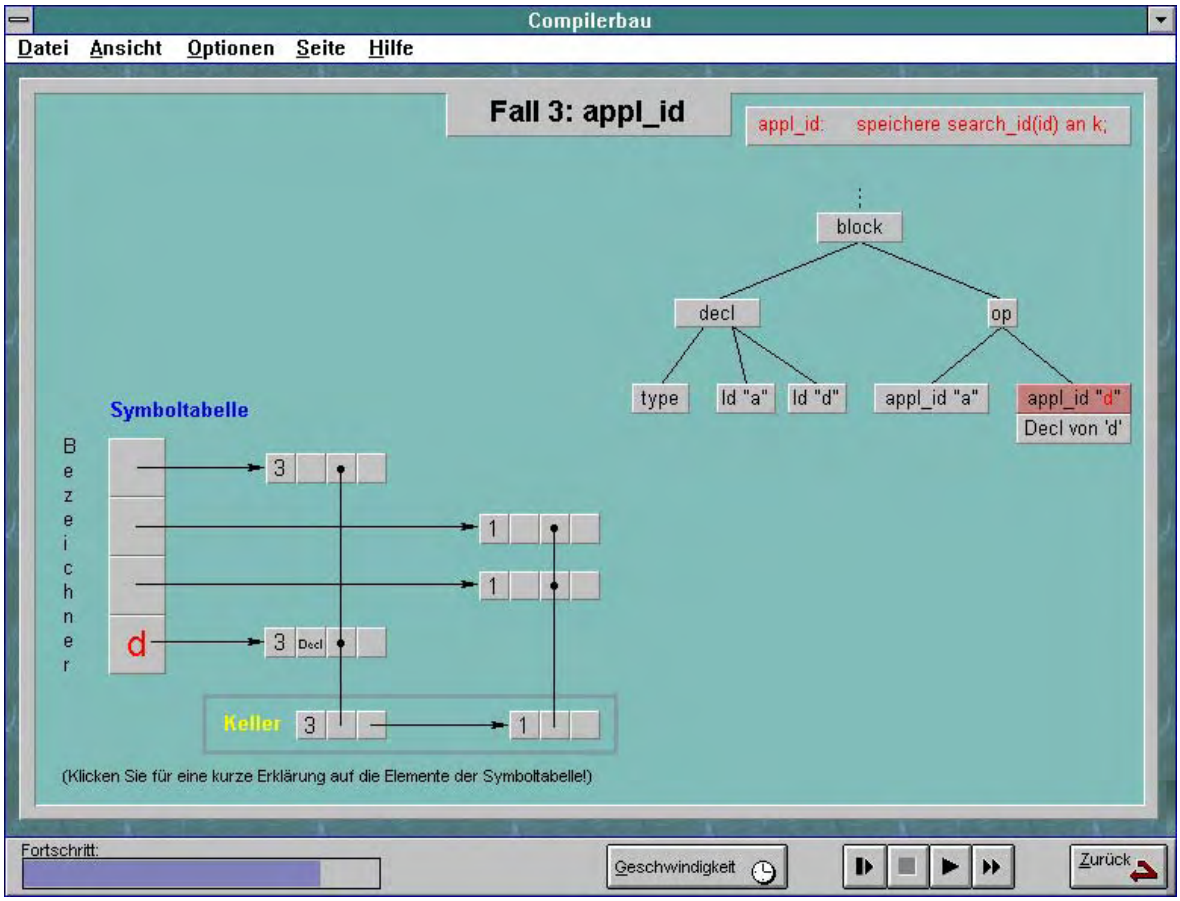
Die Bedingung der rot eingefärbten Zeile für die **if**-Anweisung ist nicht erfüllt.

Abb. C.1.9: Dritter Schritt der Beispielanimation zu einer Symboltabellenoperation



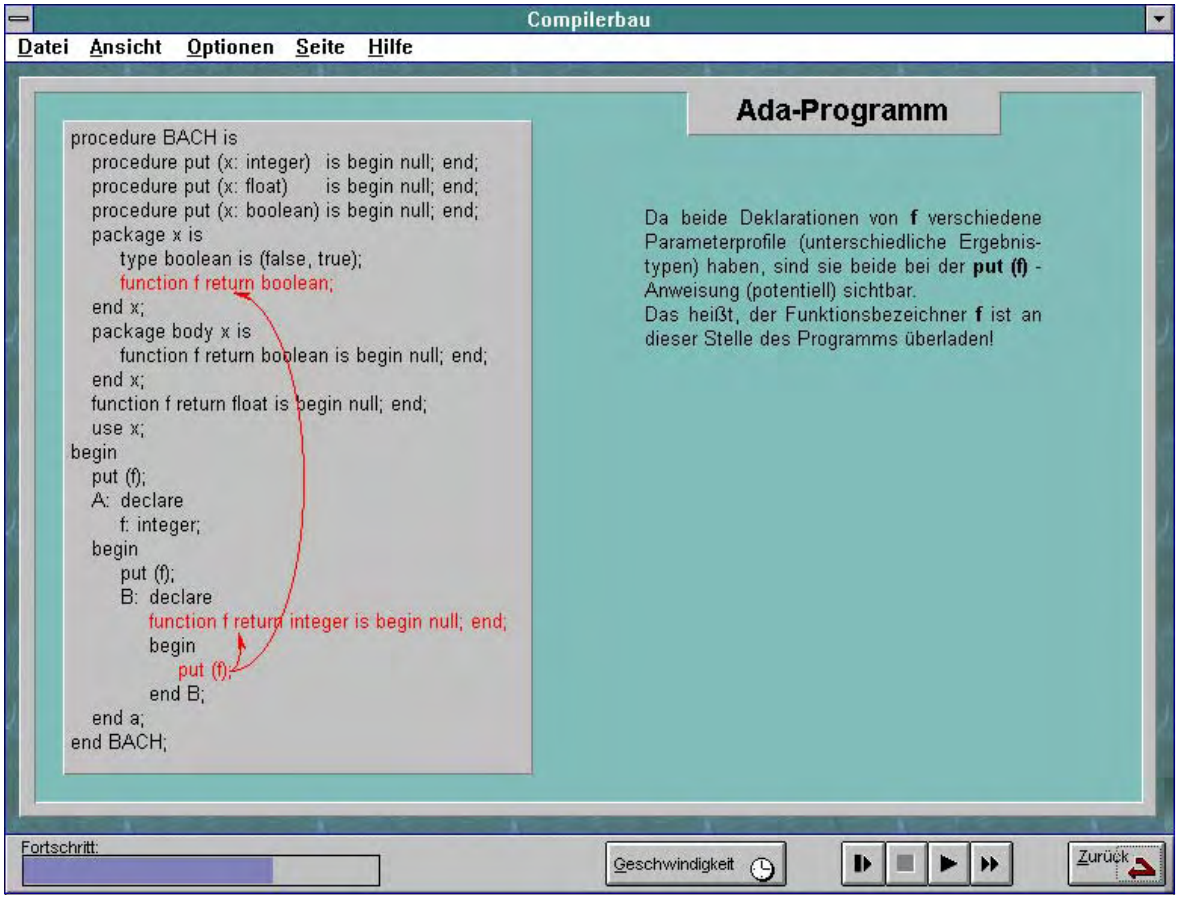
Der Beispielbezeichner „d“ wurde im dritten Block deklariert. Der entsprechende Zeiger (Adresse) des entsprechenden definierenden Vorkommens wird ausgelesen.

Abb. C.1.10: Vierter Schritt der Beispielanimation zu einer Symboltabellenoperation



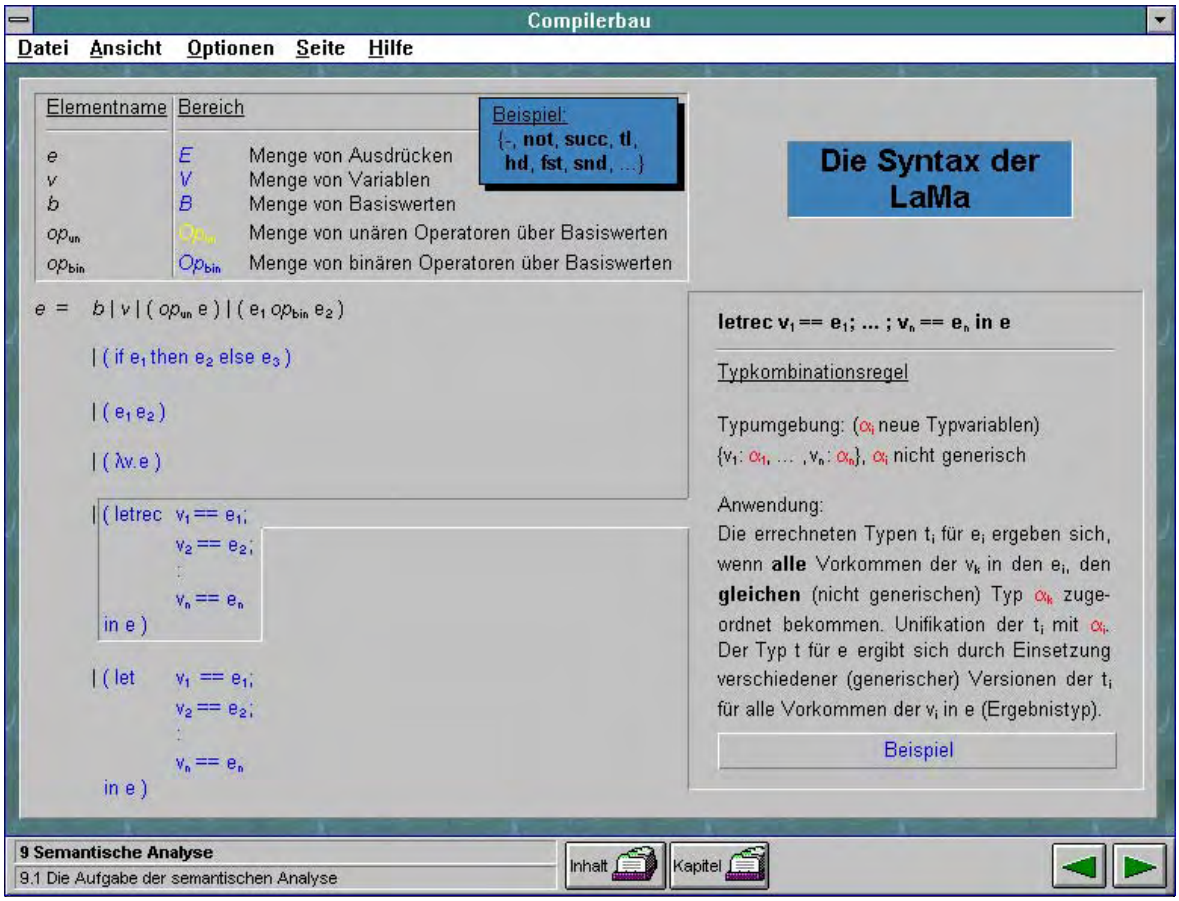
Der Knoten im zugrundeliegenden abstrakten Syntaxbaum, der dieses angewandte Vorkommen des Bezeichners „d“ symbolisiert, wird mit dem im vierten Schritt ausgelesenen Zeiger attributiert. Damit ist diese kleine Beispielanimation beendet und man kann zur Algorithmenseite (vgl. Abb. C.1.6) zurückkehren, um weitere Beispielanimationen auszuwählen.

Abb. C.1.11: Fünfter Schritt der Beispielanimation zu einer Symboltabellenoperation



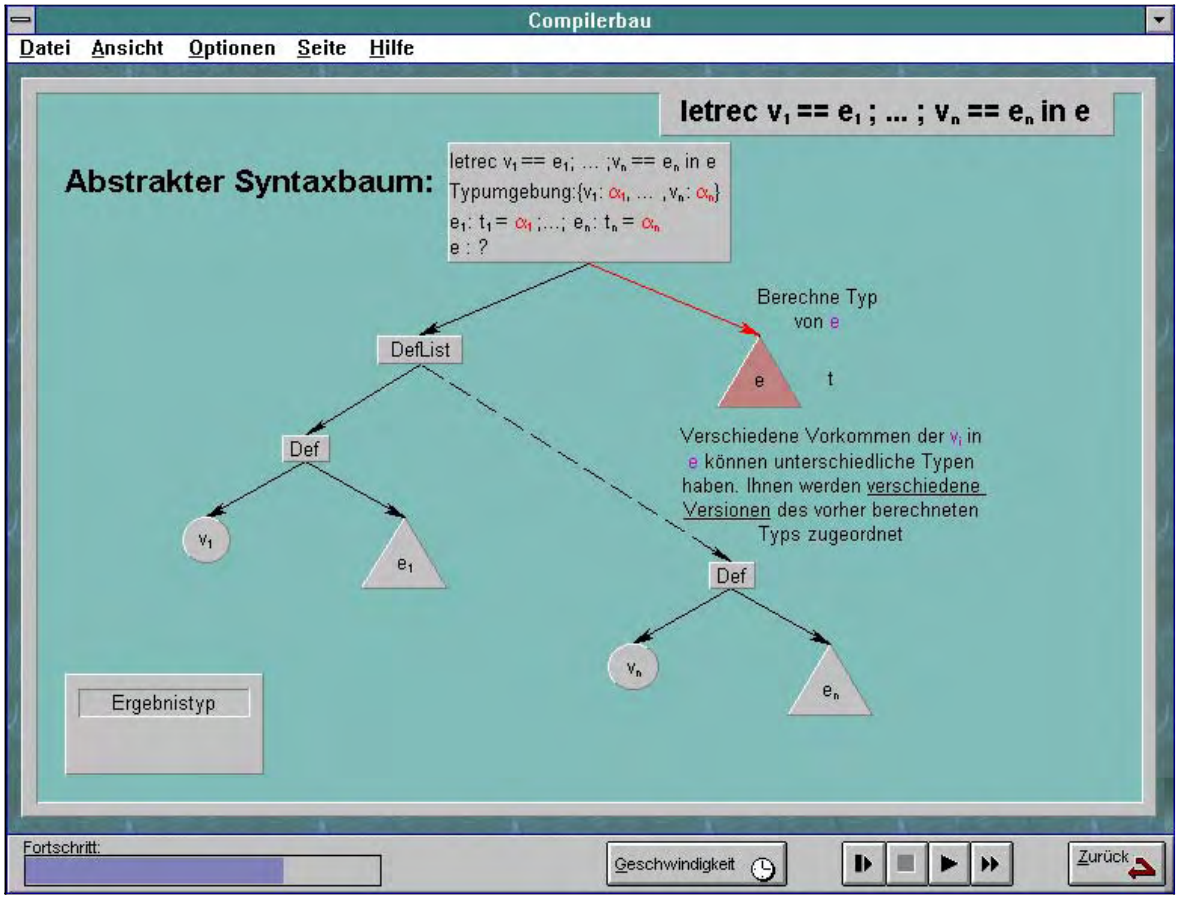
Diese Textanimation versucht das Prinzip der Überladung von Bezeichnern anhand eines einfachen ADA-Programms zu erklären. Es wird gezeigt, daß zu einem angewandten Vorkommen eines Funktionsbezeichners mehrere definierende Vorkommen existieren.

Abb. C.1.12: Textbeispiel für die Überladung von Bezeichnern



Zu den einzelnen zusammengesetzten LAMA-Ausdrücken lassen sich die entsprechenden Typkombinationsregeln anzeigen. Oben wurde der Begriff Op_{un} angewählt, um einige Beispiele für die Elemente aus der Menge zu erhalten, welche der Begriff symbolisiert. Wie in Abb. C.1.6 gezeigt, kann man sich auch hier Beispiel-animationen zu den einzelnen Typkombinationsregeln ansehen.

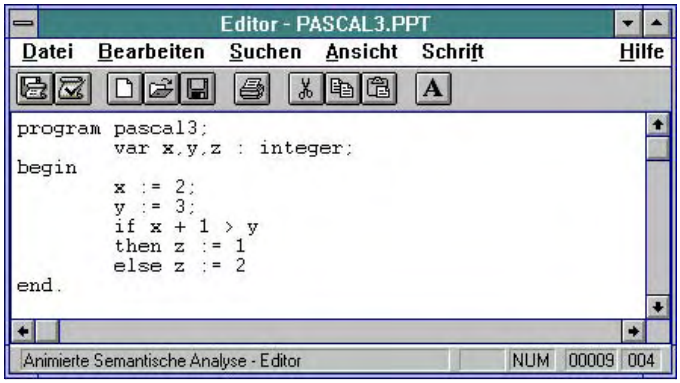
Abb. C.1.13: Die Syntax des funktionalen Sprache LAMA



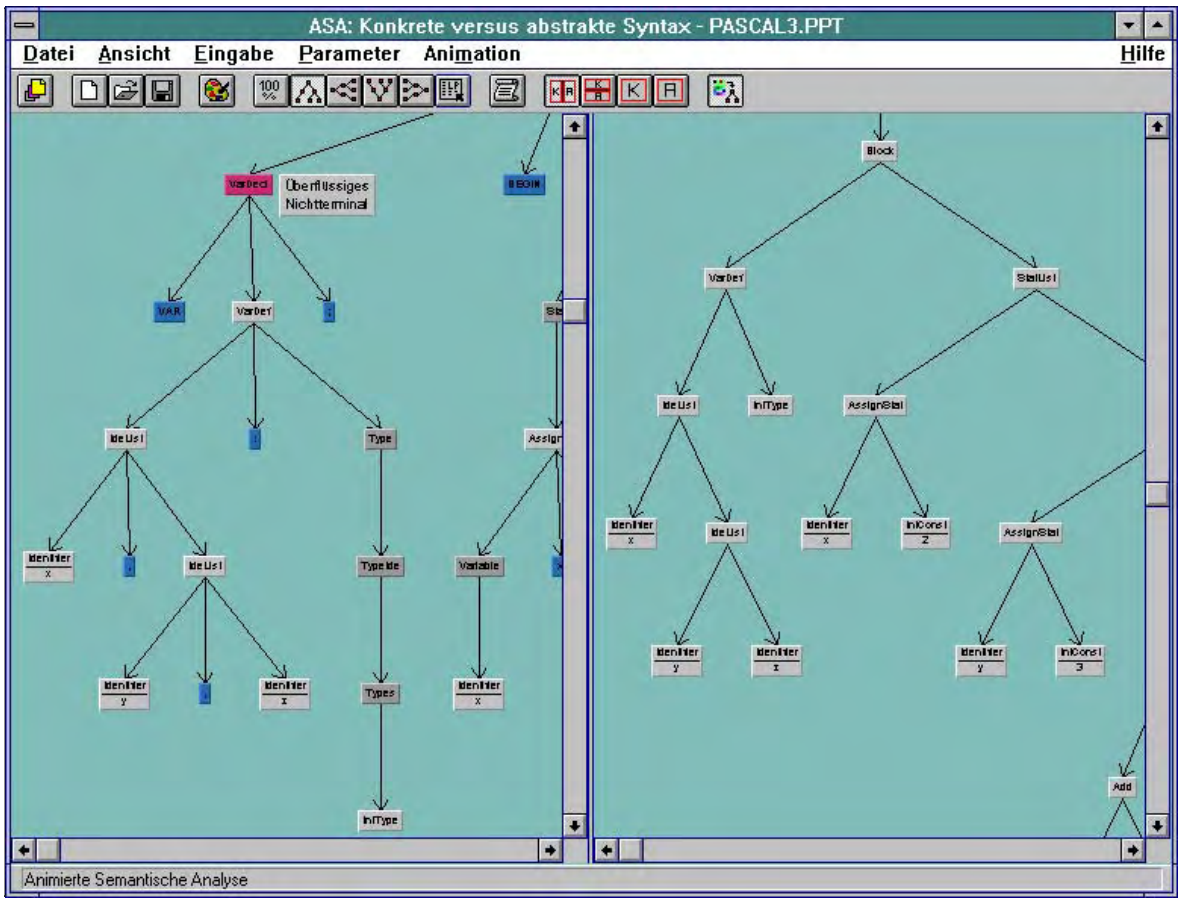
Aus Platzgründen zeigen wir nicht alle Schritte dieses Beispiels. In Abhängigkeit von der Typkombinationsregel werden die einzelnen polymorphen Typen der Unterbaumwurzeln berechnet, eine Typumgebung erzeugt, etc.

Abb. C.1.14: Animation zur Typkombinationsregel eines *letrec*-Ausdrucks

C.2 Visualisierungen aus dem ASA-Tool



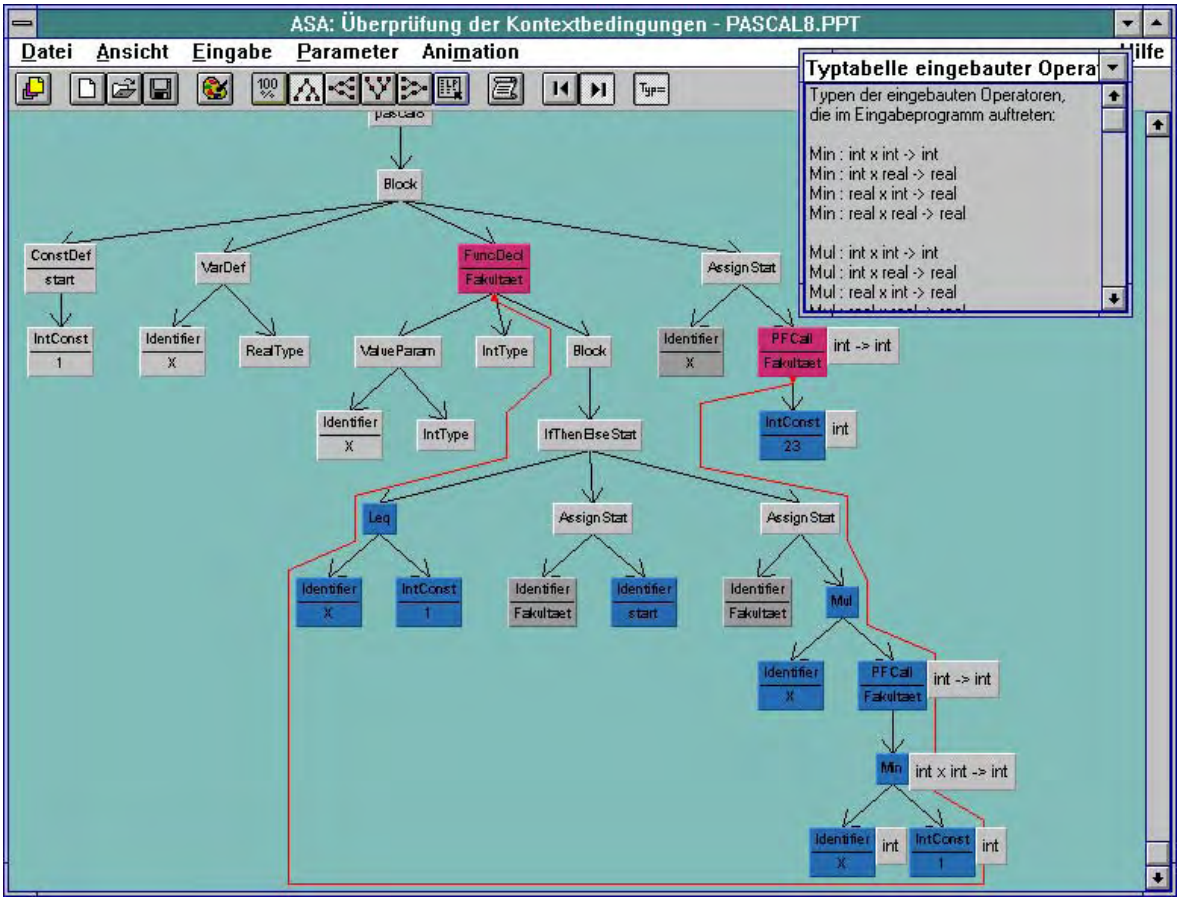
Ein PASCAL-Beispielprogramm im ASA-Editor



Visualisierung des oben im ASA-Editor gezeigten Beispielprogramms. Links ist ein Teilbaum zur konkreten Syntax und rechts der entsprechende Teilbaum zur abstrakten Syntax zu sehen (vertikale Teilung). Beide Syntaxbäume können auch horizontal geteilt oder einzeln angezeigt werden. ASA hat nach einem Mausklick auf den rot markierten Knoten eine Knoteninformation (Standardgröße) angezeigt. Die Layoutparameter entsprechen außer dem Skalierungsfaktor (60%) den Voreinstellungen.

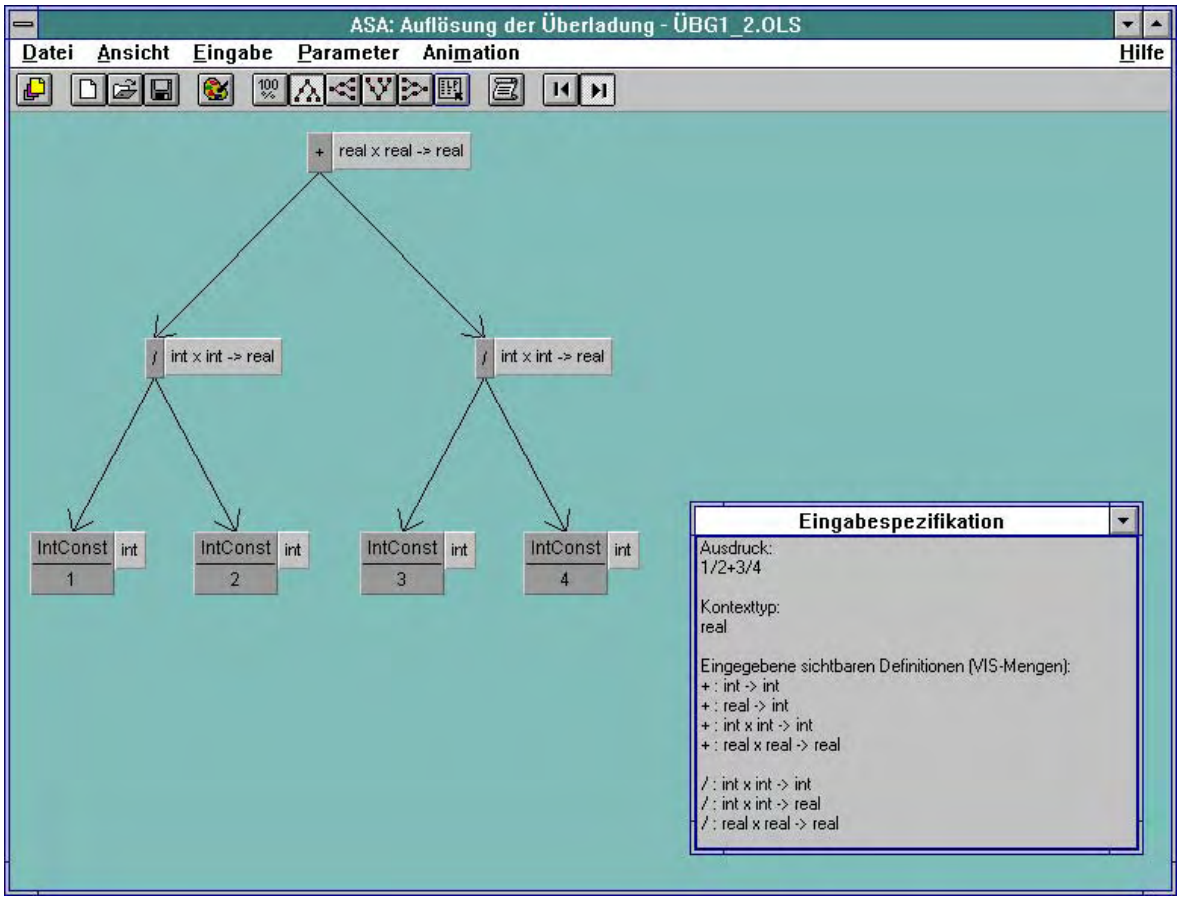
Abb. C.2.1: Visualisierung zum Vergleich der konkreten zur abstrakten Syntax

Ein weiteres PASCAL-Beispielprogramm im ASA-Editor



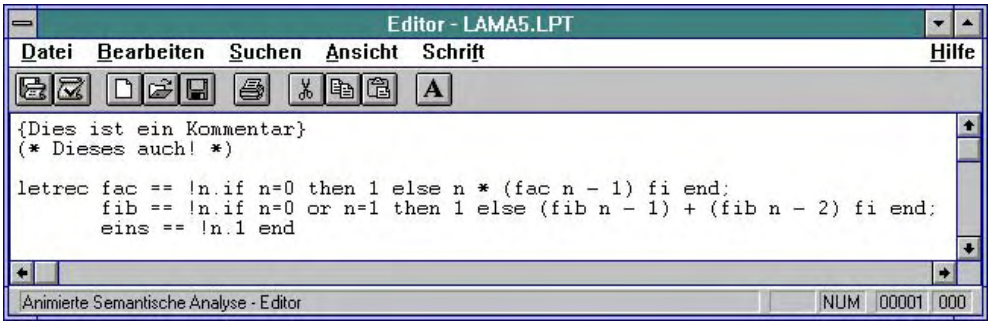
Visualisierung des oben im ASA-Editor gezeigten Beispielprogramms. Der abstrakte Syntaxbaum ist fast vollständig dargestellt. Zu einigen Knoten sind die Typattribute (große Form) zu sehen. Grundlage hierfür sind die in einem Hilfsfenster angegebenen Typen für die im Beispielprogramm verwendeten eingebauten Operatoren. Für ein angewandtes Vorkommen des Bezeichners *Fakultaet* hat ASA das nach den Gültigkeits- und Sichtbarkeitsregeln errechnete definierende Vorkommen markiert (nach dem Mausklick auf das angewandte Vorkommen). Der Skalierungsfaktor des Baumlayouts beträgt 100%. Um den Baum passend im Fenster zu plazieren, wurden die Werte für die verschiedenen Separationen (siehe Abschnitt 6.5.2.2) verkleinert.

Abb. C.2.2: Visualisierung der Überprüfung der Kontextbedingungen

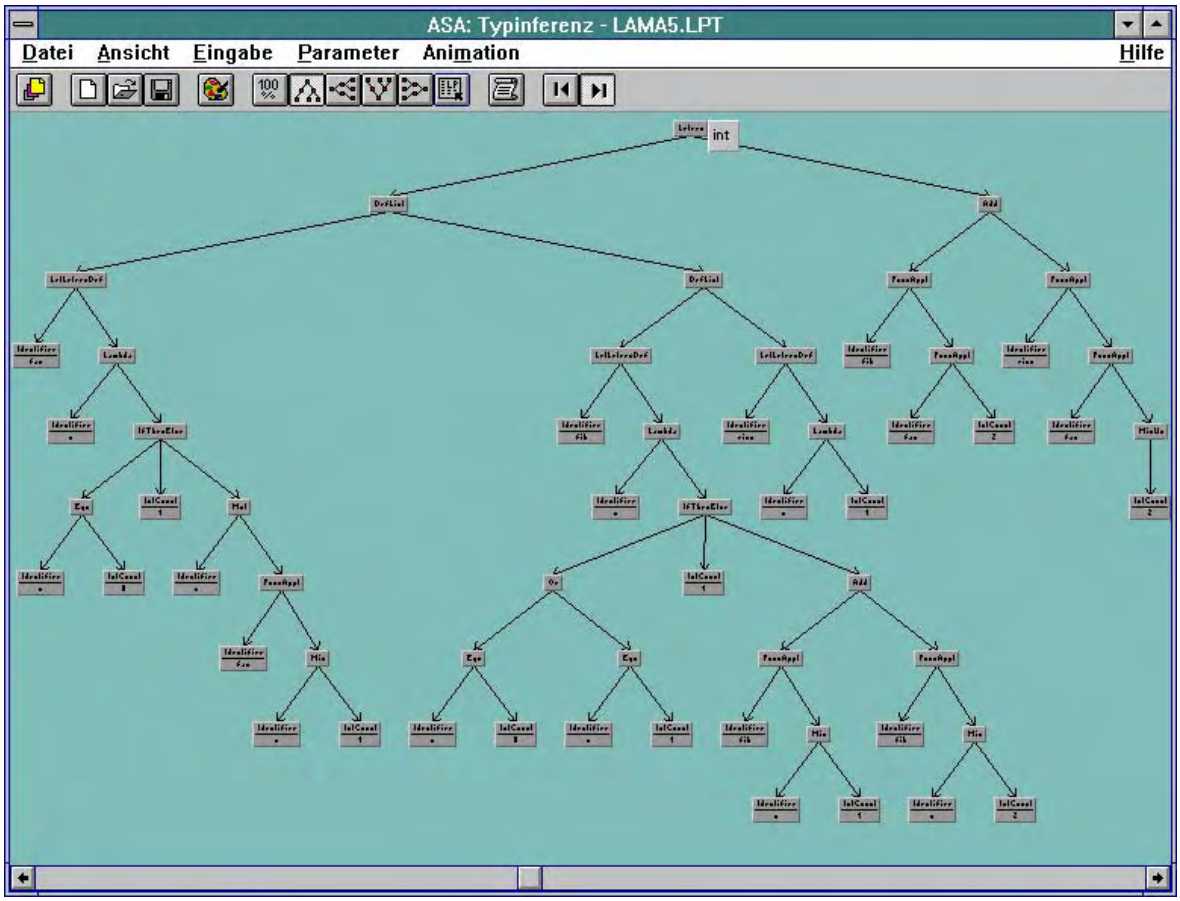


Rechts unten ist ein Hilfsfenster mit der aktuellen Eingabespezifikation eingeblendet, die zuvor mit Hilfe der Eingabemaske (vgl. Abschnitt 6.3.2.3) erstellt worden war. Im Animationsbereich ist der auf 120% vergrößerte Ausdrucksbaum mit den zu jedem Operatorknoten assoziierten *ops*-Mengen zu sehen. Alle *ops*-Mengen sind einelementig. Die Überladung der Operatoren wurde also erfolgreich aufgelöst. Die Eingabespezifikation entspricht dem Beispiel aus Abschnitt 3.4.1.

Abb. C.2.3: Visualisierung der Auflösung der Überladung

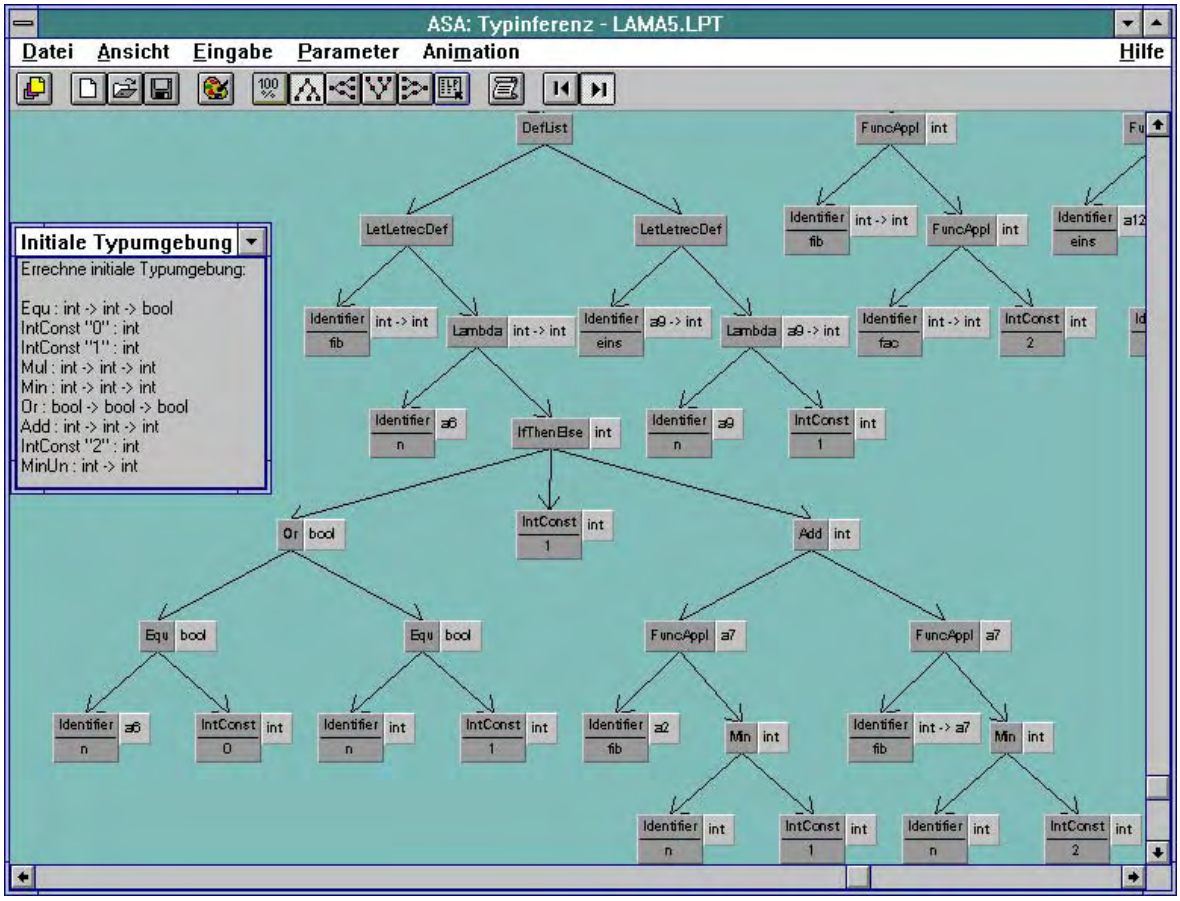


Ein LAMA-Beispielprogramm im ASA-Editor



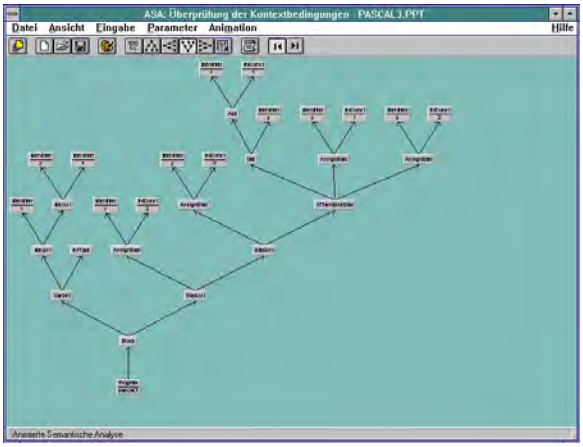
Der zu obigem Beispielprogramm generierte abstrakte Syntaxbaum; auf 50% skaliert. Der Ergebnistyp des gesamten LAMA-Ausdrucks ist der Typ **int**.

Abb. C.2.4: Visualisierung der Typinferenz (1)

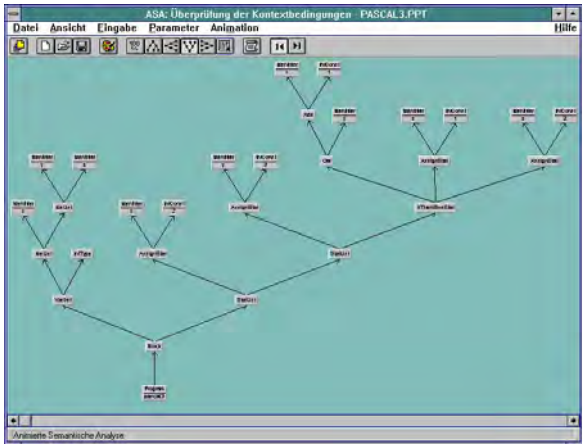


Diese Sicht stellt einen vergrößerten (100%) Teil des in Abbildung C.2.4 gezeigten Syntaxbaum dar. Alle berechneten (polymorphen) Typen (Standardgröße) sind eingeblendet. Das Hilfsfenster links zeigt die für diesen LAMA-Ausdruck gültige initiale Typumgebung. Um den Teilbaum passend im Fenster zu plazieren, wurden die Werte für die verschiedenen Separationen zusätzlich verkleinert.

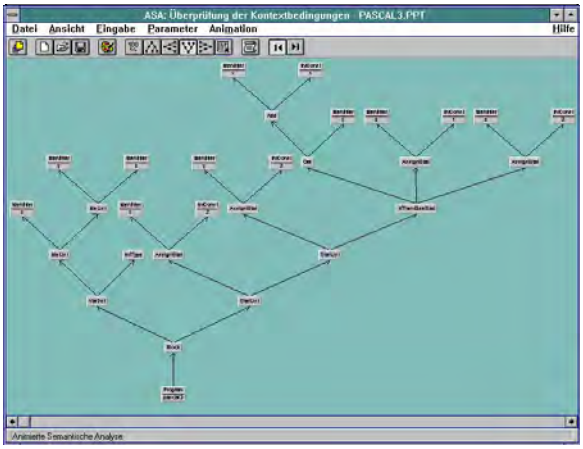
Abb. C.2.5: Visualisierung der Typinferenz (2)



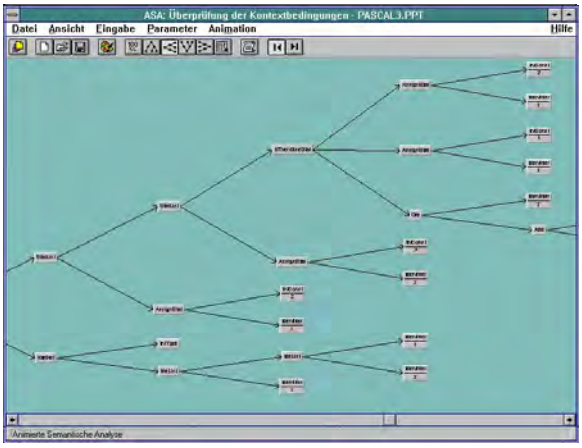
Apex unten, Skalierungsfaktor 60%,
Standardwerte für die Separationen der Knoten.



Apex unten, Skalierungsfaktor 60%,
Standardwerte für die Knotenseparationen bis auf eine Vergrößerung der *Unterbaumseparation*.



Apex unten, Skalierungsfaktor 60%,
Standardwerte für die Knotenseparationen bis auf eine Vergrößerung der *Geschwisterseparation*.



Apex links (nicht sichtbar), Skalierungsfaktor 60%,
Standardwerte für die Knotenseparationen bis auf eine Vergrößerung der *Ebenenseparation*.

Abb. C.2.6: Verschiedene Syntaxbaumlayouts am Beispiel aus Abbildung C.2.1

Literaturverzeichnis

[Alt93] A. Alteneder. *Visualisieren mit dem Computer: Computergraphik und Computeranimation*. Siemens, VCH, 1993.

[Asy94a] Asymetrix. *ToolBook, Benutzerhandbuch*. 1994.

[Asy94b] Asymetrix. *Multimedia ToolBook, Benutzerhandbuch und OpenScript-Referenz*. 1994

[Blo93] A. Bloesch. *Aesthetic Layout of Generalized Trees*. In Software-Practice and Experience 23(8), pp. 817-827, 1993.

[Bro87] M.H. Brown. *Algorithm Animation*. MIT Press, 1987.

[Bro93] M.H. Brown. *The 1992 SRC Algorithm Animation Festival*. In IEEE Symp. on Visual Languages, pp. 116-123, 1993.

[BS84] M.H. Brown, R. Sedgewick. *A System for Algorithm Animation*. In SIGGRAPH ’84, Computer Graphics 18(3), pp. 177-186, 1984.

[Clé87] D. Clément. *The Natural Dynamic Semantics of Mini-Standard ML*. In TAPSOFT ’87, Vol. 2, LNCS 250, pp. 67-81, Springer, 1987.

[Det93] J. DeTreville. *The GraphVBT Interface for Programming Algorithm Animations*. In IEEE Symp. on Visual Languages, pp. 26-31, 1993.

[DH91] P. Deuflhard, A. Hohmann. *Numerische Mathematik, Eine algorithmisch orientierte Einführung*. De-Greuter-Lehrbuch, 1991.

[DM82] L. Damas, R. Milner. *Principal Type Schemes for Functional Programs*. In 9th ACM Symp. on Principles of Programming Languages, pp. 207-212, 1982.

[DPM80] F.L. DeRemer, T.J. Pennello, R. Meyers. *A Syntax Diagram for (Preliminary) ADA*. In ACM SIGPLAN Notices 15(7,8), pp. 36-47, 1980.

[DS91] C. Donnelly, R. Stallman. *BISON Documentation*. Free Software Foundation, Cambridge, USA, 1991.

[ELL93] P. Eades, T. Lin, X. Lin. *Two Tree drawing Conventions*. In International Journal of Computational Geometry and Applications 3(2), pp. 133-153, 1993.

[FA94] M. Fritz, R. Aumiller. *Das Visual C++ 1.5 Buch*. Sybex, 1994.

[Gla93] S.C. Glassman. *A Turbo Environment for Producing Algorithm Animations*. In IEEE Symp. on Visual Languages, pp. 32-36, 1993.

[KKUG83] Kaucher, Klatte, Ullrich, v. Gudenberg. *Programmiersprachen im Griff, Band 4: ADA*. BI-Hochschultaschenbücher, 1983.

[KM90] B.W. Kernighan, D.M. Ritchie. *Programmieren in C*. Hanser, Prentice-Hall International, 1990.

[Koh95] G. Kohlmann. *Visualisierung der abstrakten P-Maschine*. Diplomarbeit, Universität des Saarlandes, 1995.

[Lor84] B. Lorho (Hrsg.). *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.

[Lee92] M. Lee. *An Algorithm Animation Programming Environment*. In ICCAL '92, LNCS 602, pp. 367-379, 1992.

[Lem94] I. Lemke. *Entwicklung und Implementierung eines Visualisierungswerkzeugs für Anwendungen im Übersetzerbau*. Diplomarbeit, Universität des Saarlandes, 1994.

[LS94] I. Lemke, G. Sander. *Visualisation of Compiler Graphs*. User Documentation, 1994.

[Mi93a] Microsoft Corporation. *Visual Workbench User's Guide*. 1993.

[Mi93b] Microsoft Corporation. *Programming Techniques*. 1993.

[Mi93c] Microsoft Corporation. *Win3.1 SDK Programmers Reference*. Vol. 1, 1993.

[Mi93d] Microsoft Corporation. *Win3.1 SDK Programmers Reference*. Vol. 2, 1993.

[Mi93e] Microsoft Corporation. *Win3.1 SDK Programmers Reference*. Vol. 3, 1993.

[Mi93f] Microsoft Corporation. *Win3.1 SDK Programmers Reference*. Vol. 4, 1993.

[Mil78] R. Milner. *A Theory of Type Polymorphism in Programming*. In Journal of Computer and Systems Sciences 17, pp. 348-375, 1978.

[Möh92] M.G. Möhrle. *Im Wettbewerb: Klassische Autorensysteme versus objektorientierte Oberflächen*. In Proceedings des 3. Internationales Symposium für Informationswissenschaften (ISI92), pp. 119-129, 1992.

[Nag82] M. Nagl. *Einführung in die Programmiersprache ADA*. Vieweg, 1982.

[Pax90] V. Paxson. *FLEX Documentation*. 1990.

[PDM80] T.J. Pennello, F.L. DeRemer, R. Meyers. *A Simplified Operator Identification Scheme for ADA*. In ACM SIGPLAN Notices 15(7,8), pp. 82-87, 1980.

[Pet92] C. Petzold. *Programmierung unter Microsoft Windows 3.1*. Microsoft Press, 1992.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Rea89] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[Rob65] J.A. Robinson. *A Maschine_Oriented Logic Based on the Resolution Principle*. In Journal ACM 12, pp. 23-41, 1965.

[San95] G. Sander. *Visualisation of Compiler Graphs*. Technical Report, Universität des Saarlandes, 1995.

[San96] G. Sander. *Visualisierungstechniken für den Compilerbau*. Dissertation, Universität des Saarlandes, 1996.

[Sed91] R. Sedgewick. *Algorithmen*. Addison-Wesley, 1991.

[Shn92] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer* Interaction. 2nd Edition, Addison-Wesley, 1992.

[SR83] K.J. Supowit, E.M. Reingold. *The Complexity of Drawing Trees Nicely*. In Acta Informatica 18, pp. 377-392, 1983.

[Sta89] G. Staubach, *UNIX-Werkzeuge zur Textmusterverarbeitung Awk, Lex, und Yacc*. Springer, 1989.

[Ste92] B. Steiner. *Visualisierung der abstrakten Maschine MaMa*. Diplomarbeit, Universität des Saarlandes, 1992.

[Wal90] J. Walker. *A node-Positioning Algorithm for General Trees*. In Software-Practice and Experience 20(7), pp. 685-705, 1990.

[Wat84] D.A. Watt. *Contextual Constraints*. In [Lor84], 1984.

[Wir95] S. Wirtz. *Visualisierung der abstrakten Maschine WiM*. Diplomarbeit, Universität des Saarlandes, 1995.

[Wir95a] W. Wirth. *Windows 95 intern*. In PC Professionell 11, pp. 168-226, Ziff-Davis, 1995.

[WM92] R. Wilhelm, D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, 1992.