Entwicklung eines Modul–Informationssystems für den NoName–Editor

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von:

Erich Müller

1. Gutachter: *Prof. Dr. Heiko Vogler*
2. Gutachter: *Prof. Dr. Helmuth Partsch*

1995

Inhaltsverzeichnis

Teil I

Grundlagen, Entwurf und Implementierung

Dieser Teil behandelt zunächst die Problemstellung und führt in die benötigten Grundlagen ein. Im Anschluß daran wird die Funktionalität des zu erstellenden Systems sowie die Realisierung des dabei verwendeten Modells beschrieben. Die Vorstellung der Lösungsansätze und deren Implementierung beendet diesen Teil der Arbeit.

1 Einleitung

1.1 Hintergrund

Im Sommer des Jahres 1994 wurde an der Universität Ulm eine funktionale Programmiersprache namens NoName entwickelt. Die wesentlichen Bestandteile dieser Sprache sind die Spezifikationen von Datentypen, algebraischen Datentypen, Funktionen und tree transducern. Letztere sind spezielle Termersetzungssysteme. Zu den genannten Konstrukten kommt noch die Import–Spezifikation hinzu, mit welcher Konstrukte aus anderen NoName–Quelldateien eingelesen werden können.

Die Syntax von NoName orientiert sich stark an der Mathematik, entsprechend streng ist die Sprache typisiert. Als Beispiel ist die Spezifikation der Fakultätsfunktion `fac` in der Notation von NoName angegeben:

```
FUNC_SPEC
      fac: CARDINAL -> CARDINAL.
      fac(0) = 1
      fac(n+1) = (n + 1) * fac(n);
END_FUNC_SPEC
```

Für weitere Informationen zur Sprache NoName sei der interessierte Leser an dieser Stelle auf die NoName–Sprachbeschreibung [EGH+94a] verwiesen.

Im Rahmen eines Praktikums entstand für die Sprache NoName ein syntaxgesteuerter Editor. Er wurde mit dem Synthesizer Generator (SGen) der Firma GrammaTech entwickelt (weitere Informationen hierzu können [RT89] entnommen werden). Dieser sogenannte NoName–Editor war lediglich ein Prototyp, dessen Verwendbarkeit sehr eingeschränkt war. Parallel zur Erstellung der vorliegenden Arbeit wurde in einer Diplomarbeit ein stark verbesserter und erheblich erweiterter syntaxgesteuerter Editor entwickelt. Dieser nimmt nun z.B. die Typüberprüfung vor, auch kann ein NoName–Programm in die funktionale Programmiersprache Miranda übersetzt (und damit auch ausgeführt) werden. Einzelheiten hierzu können der Diplomarbeit von Dietmar Ernst ([Ern95]) entnommen werden.

Bereits während des oben genannten Praktikums wurde vorgeschlagen, die Funktionalität der Import–Spezifikation durch die Verwendung eines Informationssystems bzw. einer Datenbank zu erweitern, indem Sprachkonstrukte in geeigneter Form in der Datenbank abgelegt werden. Während beim bisher üblichen Import der Name des zu importierenden Sprachkonstruktes sowie der

Name der benötigten Quelldatei bekannt sein müssen, könnte man bei der Verwendung einer Datenbank über geeignete Suchkriterien interaktiv eine Auswahl aus allen vorhandenen Konstrukten treffen. Ein Beispiel wird den Unterschied klarer werden lassen:

```
FROM mathematik.nnm IMPORT fac, fibonacci;
```

Diese Import–Spezifikation importiert die Funktionen `fac` und `fibonacci` aus der Quelldatei `mathematik.nnm`. Der Programmierer kann die beiden Funktionen nur dann importieren, wenn er deren Namen sowie den Namen der Quelldatei kennt. Wenn nun ein Datenbanksystem vorhanden wäre, in welchem der Benutzer interaktiv aus den gespeicherten Sprachkonstrukten wählen kann, dann könnte z.B. ein Suchkriterium namens `Mathematik` oder dergleichen existieren, welches den beiden Funktionen zugeordnet ist. Durch die Angabe dieses Suchkriteriums erhielte der Benutzer die Namen aller zugeordneten Sprachkonstrukte angezeigt und könnte einfach das gewünschte Konstrukt auswählen. Allerdings wird der Einsatz eines Datenbanksystems erst bei relativ großen Datenbeständen sinnvoll. In kleineren Umgebungen ist die Import–Spezifikation sicherlich völlig ausreichend.

Diese Überlegungen trugen mit zur Entwicklung des NoName–Modul–Informationssystems (kurz NoMIS) bei, welches in dieser Arbeit vorgestellt wird. Die Aufgabenstellung ist somit relativ klar:

> Es ist ein Programm zu erstellen, mit welchem interaktiv Konstrukte der Sprache NoName in einer Datenbank abgelegt und daraus importiert werden können. Dabei sollen Suchkriterien zugeordnet bzw. angegeben werden, die das Auffinden gespeicherter Konstrukte erleichtern. Alle Tätigkeiten müssen in der Benutzeroberfläche des NoName–Editors angestoßen werden können.

Die Aufgabenstellung wurde bewußt flexibel formuliert, da zum Zeitpunkt der obigen Überlegungen noch keine konkreten Vorstellung für deren Realisierungsmöglichkeiten existierten. So war z.B. noch nicht abzusehen, welcher Aufwand allein für die Realisierung der Verbindung des Editors mit einer Datenbank entsteht. Auch war noch kein Weg bekannt, um den zu speichernden Programmtext aus dem Editor zu extrahieren. Die flexible Aufgabenstellung räumt auch gewisse Freiheiten bei der Modellierung des Problems ein. So wird z.B. noch nicht festgelegt, welche Daten zu speichern sind und in welcher Form dies zu erfolgen hat.

1.2 Datenmodell, Anfragesprache und Kopplung mit dem Editor

Bevor wir geeignete Daten– und Speicherungsstrukturen überlegen, müssen wir uns auf ein Datenmodell festlegen. Da in dieser Diplomarbeit keine Datenbank implementiert werden sollte, fiel die Wahl auf ein relationales Datenmodell. In der Anwendung kommt das Datenbanksystem Ingres zum Einsatz[1]. Ingres wurde an der University of Berkeley entwickelt und ist seit einigen Jahren auch kommerziell verfügbar (siehe [Pet92]).

Die bekannteste Datenbank–Anfragesprache für relationale Datenbanksysteme ist SQL (Simple Query Language, siehe [MS93]), die auch von Ingres als Benutzerschnittstelle angeboten wird. Diese Benutzerschnittstelle kann auch aus der Programmiersprache C angesprochen werden, d.h. es ist möglich, SQL–Anweisungen aus C heraus an die Datenbank zu schicken. Hierbei spricht man von Embedded SQL (ESQL). Dadurch können eigenständige Programme mit integrierten Datenbankoperationen erstellt werden, d.h. der Benutzer muß nicht selbst SQL–Anweisungen eintippen, sondern er kann z.B. durch die Wahl eines Menüpunktes eine Datenbankoperation anstoßen.

Der Synthesizer Generator bietet ebenfalls eine Schnittstelle zu der Programmiersprache C. Zusätzlich existieren einige Funktionen, mit welchen eigene Menüs und Dialogfenster im Stil der SGen–Benutzeroberfläche generiert werden können. Mit diesen Funktionen war es möglich, die Bedienung des NoName–Modul–Informationssystems in die Oberfläche des NoName–Editors zu integrieren.

Da sowohl der SGen als auch das Datenbanksystem Ingres über eine Schnittstelle zur Programmiersprache C verfügen, konnte die Kopplung der beiden Systeme über diese Schnittstellen realisiert werden. Die Abbildung 1.1 verdeutlicht dies.

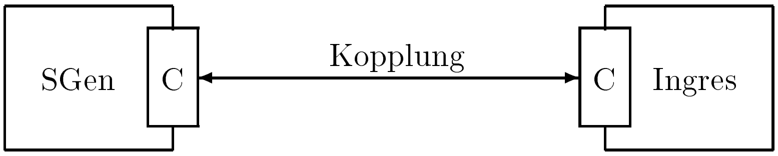


Abb. 1.1: Kopplung des Editors mit der Datenbank. C steht jeweils für das C–Interface.

[1] Es sollte ohne allzu großen Aufwand möglich sein, die Implementierung auch für andere relationale Datenbanksysteme wie z.B. Oracle umzuschreiben.

1.3 Die speicherbaren Konstrukte und die zugeordneten Daten

Nun legen wir fest, welche Konstrukte der Sprache NoName in der Datenbank abgelegt werden können. Dazu zählen mit Sicherheit die Spezifikationen von Datentypen, algebraischen Datentypen, Funktionen und tree transducern. Die Speicherung von Import–Spezifikationen ist meines Erachtens unnötig, da diese bei der Verwendung eines Datenbanksystems überflüssig sind. Wir werden somit im weiteren die Import–Spezifikationen nicht mehr betrachten. Der Begriff *Sprachkonstrukt* bezieht sich im folgenden immer nur auf die speicherbaren Konstrukte der Sprache NoName.

Die Vielfalt der Sprachkonstrukte läßt sich noch weiter einschränken, wenn diese zum Zeitpunkt der Speicherung die folgenden Kriterien erfüllen müssen:

1. Die Sprachkonstrukte, welche in der Datenbank abgelegt werden sollen, müssen korrekt bezüglich der Syntax von NoName sein. Die Erfüllung dieser Bedingung kann vom NoName–Editor überprüft bzw. sichergestellt werden.

2. Die zu speichernden Konstrukte müssen korrekt bezüglich der statischen Semantik von NoName sein. Auch dieses Kriterium kann vom NoName–Editor überprüft werden.

Manchmal kann es sinnvoll sein, auch nicht vollständig spezifizierte Sprachkonstrukte in der Datenbank abzulegen. Ein Beispiel hierfür sind Funktionsschemata, die lediglich den strukturellen Aufbau einer Funktionsspezifikation widerspiegeln und die konkrete Implementierung dem Benutzer überlassen. Ein Funktionsschema für die bereits angeführte Funktion `fac` könnte z.B. wie folgt aussehen:

```
FUNC_SPEC
      f: <type> -> <type>.
      f(0) = <expression>
      f(n+1) = <expression>;
END_FUNC_SPEC
```

Um auch derartige Sprachkonstrukte in der Datenbank ablegen zu können, müssen wir auch Platzhalter (z.B. `<type>`) in den Spezifikationen zulassen.

Wenn gespeicherte Sprachkonstrukte die oben genannten Kriterien erfüllen, dann können sie vom Benutzer des Systems wie eine „Black Box“ verwendet werden, denn die Korrektheit der Konstrukte bezüglich Syntax und statischer

Semantik wurde ja bereits zum Zeitpunkt der Speicherung sichergestellt. Selbstverständlich können dadurch dennoch nicht alle Konflikte ausgeschlossen werden. Typisch hierfür sind Namenskonflikte: In NoName müssen alle verwendeten Namen eindeutig sein. Wenn nun der Benutzer in seinem Programm einen Namen verwendet, der auch in einem importierten Sprachkonstrukt verwendet wird, dann erhält der Benutzer zwangsläufig eine Fehlermeldung, obwohl das importierte Konstrukt eigentlich als Black Box verwendet wurde. Derartige, dynamisch auftretende Konflikte können nicht ausgeschlossen werden. Abbildung 1.2 verdeutlicht die Entstehung eines Namenskonfliktes.

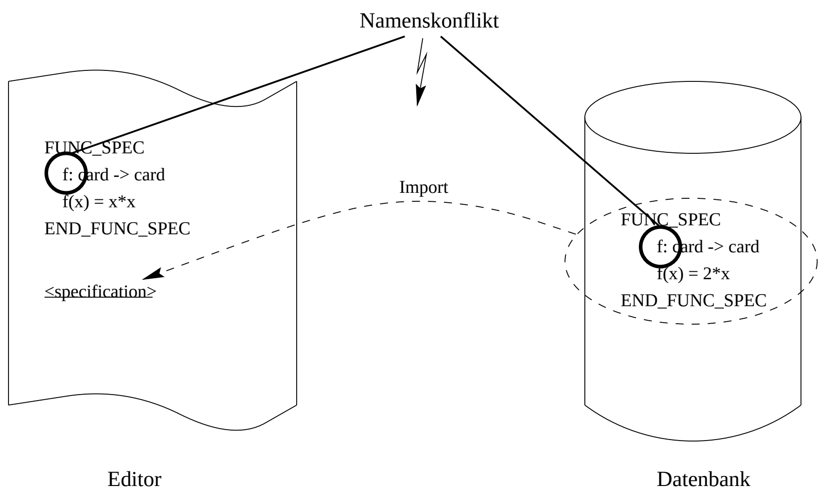


Abb. 1.2: Entstehung eines Namenskonfliktes durch Importieren eines Sprachkonstruktes

Nun legen wir fest, welche Daten zu jedem einzelnen Sprachkonstrukt in der Datenbank abgelegt werden sollen. Jedes Sprachkonstrukt erhält einen Namen und eine Version, die zusammen dieses Konstrukt eindeutig bestimmen. Daneben möchten wir einen optionalen Kommentar speichern, welcher der Beschreibung des Sprachkonstruktes dient. Vom System können der Typ der vorliegenden Spezifikation (d.h. ob es sich um die Spezifikation eines Datentyps oder einer Funktion etc. handelt), der Benutzer (über die UNIX–Umgebungsvariable `USER`) sowie das Systemdatum automatisch ermittelt und abgespeichert werden. Selbstverständlich muß auch der NoName–Code in der Datenbank abgelegt werden. Um die Verwendung einer Datenbank überhaupt rechtfertigen zu können, müssen ein oder mehrere Suchkriterien gespeichert werden. Mit diesen Suchkriterien wollen wir uns zunächst beschäftigen.

1.4 Die Verwaltung der Suchkriterien

Suchkriterien — wir wollen sie im folgenden auch *Topics* nennen — haben einen eindeutigen Namen und einen Kommentar. Der Name dient zur eindeutigen Identifizierung eines Topics. Um die Suchkriterien sinnvoll einsetzen zu können, benötigen wir eine Topic–Verwaltung. In dieser Verwaltung sind die folgenden Aktionen zusammengefaßt:

- Anlegen eines neuen Topics.
- Löschen nicht mehr benötigter Topics.
- Anzeige aller vorhandenen Topics.

Die Trennung der Topic–Verwaltung von der Sprachkonstrukt–Verwaltung, die wir später betrachten werden, erfolgte aus einer grundlegenden Überlegung heraus: Nur wenn Topics gezielt angelegt und gelöscht werden, dann ist auch eine gezielte Verwendung derselben möglich. Somit kann ein Benutzer nicht beim Speichern eines Sprachkonstruktes „eben mal schnell ein neues Topic generieren“, nur weil er nicht weiß, welche Suchkriterien bereits existieren und eventuell passend sind. Diese Aufgabentrennung verhindert auf benutzerfreundlichem Wege die Entstehung unzähliger Topics, die die Suche nach einem Sprachkonstrukt nicht vereinfachen, sondern lediglich erschweren würden. Spätestens dann, wenn genau so viele Topics wie Sprachkonstrukte vorhanden sind und jedes Sprachkonstrukt einem anderen Topic zugeordnet ist, dann haben die Topics ihre Existenzberechtigung verloren.

Topics sind prinzipiell nicht hierarchisch aufgebaut, um dem Benutzer bzw. den Benutzern einen möglichst hohen Freiheitsgrad in der Zuweisung von Topics zu Sprachkonstrukten zu gewähren. Der Benutzer muß auch selbst dafür sorgen, daß die Topics sinnvolle Namen erhalten und entsprechend ihrer zugedachten Bedeutung eingesetzt werden. Als Namen eignen sich beispielsweise Projektnamen, Bezeichnungen der charakteristischen Eigenschaften oder auch Zugehörigkeiten zu bestimmten Bereichen aus der Mathematik (z.B. `Trigonometrie`).

Beim Löschen eines Topics ist folgende Einschränkung zu berücksichtigen: Ein Topic kann nur dann aus der Datenbank entfernt werden, wenn kein Sprachkonstrukt existiert, das diesem Topic zugeordnet ist. Dadurch wird verhindert, daß ein Sprachkonstrukt ungewollt ohne zugeordnetes Suchkriterium existiert und dadurch nur noch erschwert aufgefunden werden kann.

1.5 Die Verwaltung der Sprachkonstrukte

Nach diesen Betrachtungen kommen wir nun zur Verwaltung der Sprachkonstrukte. Wie bereits angeführt, ist diese Verwaltung strikt getrennt von der Verwaltung der Suchkriterien. Die Sprachkonstrukt–Verwaltung umfaßt die folgenden Aufgaben:

- Speichern eines neuen Sprachkonstruktes in der Datenbank.
- Löschen eines gespeicherten Sprachkonstruktes aus der Datenbank.
- Importieren eines gespeicherten Sprachkonstruktes aus der Datenbank in den NoName–Editor.

Ablegen eines Sprachkonstruktes in der Datenbank

Wir wollen uns zunächst mit der Speicherung eines neuen Sprachkonstruktes in der Datenbank befassen. Wie wir bereits wissen, können nur komplette Sprachkonstrukte gespeichert werden, wobei bestimmte Kriterien erfüllt sein müssen. Nun muß das NoName–Modul–Informationssystem zunächst davon in Kenntnis gesetzt werden, welches Sprachkonstrukt des aktuellen Programms vom NoName–Editor in die Datenbank übertragen werden soll. Um die Auswahl so einfach wie möglich zu machen, gilt folgende Vereinbarung: In die Datenbank wird das Sprachkonstrukt aufgenommen, welches im NoName–Editor strukturell markiert ist[2]. Die Speicherung in der Datenbank wird durch die Auswahl eines Menüpunktes angestoßen. Dieser Menüpunkt ist allerdings nur dann wählbar, wenn das zu speichernde Sprachkonstrukt die bereits besprochenen Kriterien erfüllt.

In einem Dialogfenster muß der Benutzer einen Namen und eine Version für das zu speichernde Sprachkonstrukt angeben. Der Namen und die Version müssen zusammen eindeutig das Sprachkonstrukt bestimmen. Falls diese Bedingung verletzt ist, wird eine Fehlermeldung auf dem Bildschirm ausgegeben. Die Angabe eines Kommentars ist optional, wird aber aufgrund der besseren Dokumentation dringend empfohlen. Ebenfalls optional ist die Zuweisung beliebig vieler Topics. Da der Vorteil des Einsatzes einer Datenbank gegenüber der einfachen Import–Spezifikation aber genau in der vereinfachten Verwaltung von Sprachkonstrukten liegt — eben durch die Möglichkeit, Suchkriterien zu verwenden —, sollte in der Regel jedem Sprachkonstrukt mindestens ein Topic zugeordnet werden.

[2]Diese Markierung kann durch Anklicken eines umschließenden Schlüsselwortes (z.B. `FUNC_SPEC`) erreicht werden

Durch „Anklicken“ eines Schaltfeldes im Dialogfenster wird die Speicherung vollzogen. Dabei wollen wir noch die Speicherung des NoName–Codes näher betrachten. Ein NoName–Sprachkonstrukt kann — zumindest theoretisch — beliebig groß sein, d.h. es existiert im allgemeinen keine Obergrenze für die Länge des zugehörigen NoName–Codes. Nun sind relationale Datenbanksysteme aber nicht in der Lage, beliebig große Datensätze zu speichern. Stattdessen muß der Programmierer einer Datenbankanwendung bestimmte Maximalgrößen, die je nach verwendetem Datenbanksystem unterschiedlich sein können, berücksichtigen. Aus diesem Grund wird in der vorliegenden Arbeit der NoName–Code in Teile zerlegt, deren Größe maximal 1500 Bytes beträgt. Einträge dieser Größe kann das Datenbanksystem Ingres noch verarbeiten. Um die Teile auch wieder korrekt zusammensetzen zu können, werden diese fortlaufend numeriert. Aufgrund dieser Ordnung ist ein späteres Rekonstruieren des originalen NoName–Codes problemlos möglich. Diese Arbeiten — das Aufteilen und Numerieren sowie das Zusammensetzen in der korrekten Reihenfolge — werden vom Programm zu den entsprechenden Zeitpunkten jeweils automatisch durchgeführt.

Zusätzlich zu dem vom Benutzer gewählten Sprachkonstrukt werden auch alle Sprachkonstrukte, die von dem soeben gespeicherten Konstrukt benötigt werden, mit in der Datenbank abgelegt. Das folgende Programmbeispiel wird die Vorteile dieser Vorgehensweise klarer werden lassen:

```
TYPE_SPEC
     card = CARDINAL;
END_TYPE_SPEC

FUNC_SPEC
     fac: card -> card.
     fac(0) = 1
     fac(n+1) = (n + 1) * fac(n);
END_FUNC_SPEC
```

Die Funktion `fac` benötigt den Datentyp `card`. Wenn der Benutzer nun die Funktion `fac` in der Datenbank ablegen möchte, dann müßte er zum Erhalt der Black Box–Eigenschaft auch den Datentyp `card` in der Datenbank ablegen. Diese Arbeit übernimmt NoMIS, indem es die Bezeichner aller benötigten Konstrukte ermittelt und anschließend die Spezifikationen, in welchen diese Bezeichner definiert werden, in die Datenbank überträgt. Dabei wird eine redundante Abspeicherung von Sprachkonstrukten so weit wie möglich vermieden.

Entfernen eines Sprachkonstruktes aus der Datenbank

Nun wollen wir das Löschen eines Sprachkonstruktes aus der Datenbank näher betrachten. Üblicherweise können Löschvoränge nicht rückgängig gemacht werden und sind daher mit besonderer Vorsicht zu verwenden. Um auch in NoMIS versehentliches Löschen und mutwillige Zerstörung von Sprachkonstrukten anderer Benutzer möglichst auszuschließen, kann jeder Benutzer nur Sprachkonstrukte aus der Datenbank entfernen, die er zuvor selbst in der Datenbank abgelegt hat. Der Löschvorgang an sich bleibt aber irreversibel.

Um ein Sprachkonstrukt aus der Datenbank zu löschen, muß der Benutzer des NoName–Modul–Informationssystems zunächst den Spezifikationstyp des zu löschenden Sprachkonstruktes angeben. Dadurch wird der Suchraum für Sprachkonstrukte stark eingeschränkt. Anschließend müssen noch der Name und die Version des Sprachkonstruktes eingegeben werden. Hierfür bestehen zwei prinzipielle Möglichkeiten:

1. Der Benutzer kennt den Namen und die Version des Sprachkonstruktes und füllt in einem Dialogfenster die entsprechenden Felder mit diesen Daten aus.

2. Der Benutzer kennt den Namen und/oder die Version des Sprachkonstruktes nicht. Dann kann er aus einer Liste die Topics auswählen, die dem Sprachkonstrukt vermutlich zugeordnet sind. Nun erfolgt in der Datenbank eine automatische Suche nach Sprachkonstrukten, welche mindestens einem der ausgewählten Topics zugeordnet sind. Diese Sprachkonstrukte werden dem Benutzer in einer Liste angezeigt, aus der er das zu löschende Konstrukt auswählen kann.

Besonders bei der unter 2. angeführten Methode kann es vorkommen, daß sich der Benutzer nicht sicher ist, das gewünschte Sprachkonstrukt gewählt zu haben. Um sich davon überzeugen zu können, kann der NoName–Code des gewählten Konstruktes auf dem Bildschirm angezeigt werden.

Wenn der Benutzer nach der Auswahl des Sprachkonstruktes den Löschvorgang anstößt, dann werden auch alle Sprachkonstrukte, die von dem zu löschenden Konstrukt benötigt werden, aus der Datenbank entfernt. Da NoMIS aber nicht entscheiden kann, ob diese Sprachkonstrukte tatsächlich aus der Datenbank entfernt werden sollen, liegt die Entscheidung hierüber beim Benutzer. Dieser muß bei jedem Sprachkonstrukt, das automatisch gelöscht werden könnte, seine Zustimmung geben bzw. verweigern. Auch hier besteht als Entscheidungshilfe jeweils die Möglichkeit, den NoName–Code auf dem Bildschirm anzeigen zu lassen.

Wiederverwenden eines Sprachkonstruktes

Als letztes wollen wir nun noch den Import von Sprachkonstrukten aus der Datenbank in den NoName–Editor betrachten.

Die Auswahl des zu importierenden Sprachkonstruktes erfolgt analog der Auswahl eines Sprachkonstruktes beim Löschen aus der Datenbank. Wenn der Import des gewählten Sprachkonstruktes vom Benutzer angestoßen wird, dann werden auch alle davon benötigten Konstrukte in den Editor übertragen, sofern diese nicht schon dort vorhanden sind. Die Einhaltung dieser Einschränkung muß gewährleistet sein, damit der Benutzer nicht unzählige Fehlermeldungen wegen mehrfach verwendeter Namen erhält.

Die zu importierenden Sprachkonstrukte werden „von unten nach oben“ in den NoName–Editor eingefügt, d.h. das vom Benutzer gewählte Sprachkonstrukt steht in der Liste der importierten Konstrukte ganz unten. Dadurch wird sichergestellt, daß alle benötigten Funktionen, Datentypen etc., die soeben importiert wurden, vor deren Verwendung deklariert werden.

Sollten durch den Import Namenskonflikte aufgetreten sein — wir haben deren Zustandekommen bereits betrachtet —, dann muß der Benutzer für deren Auflösung sorgen. In der Regel sollten nicht die Namen der importierten Funktionen etc. geändert werden, sondern die Namen der selbstgeschriebenen Konstrukte. Dieses Vorgehen hat den Vorteil, daß importierte Konstrukte in allen Programmen denselben Namen haben.

1.6 Übersichten über den Datenbankinhalt

Zum Abschluß der Betrachtungen wollen wir noch eine Hilfestellung aufzeigen, die das NoName–Modul–Informationssystem für „Systemeinsteiger“ anbietet. Für Benutzer, die sich in ein bestehendes System und dessen Daten einarbeiten müssen, ist eine Übersicht über den vorhandenen Datenbestand immer von Vorteil. Dieser Gedanke war der Anlaß, in NoMIS eine Möglichkeit zu schaffen, mit welcher es dem Benutzer möglich ist, Listen in Form von LaTeX–Dateien mit den gespeicherten Daten zu generieren. Die folgenden drei Listen stehen hierfür zur Verfügung:

- Übersicht über alle gespeicherten Sprachkonstrukte. Diese Liste enthält den Namen, die Version, den Kommentar, den Spezifikationstyp, das Speicherungsdatum und den Namen des Programmierers.
- Übersicht über alle gespeicherten Topics. In dieser Liste sind die Namen und Kommentare aller Topics zusammengestellt.
- Übersicht über die Zuordnungen von Topics und Sprachkonstrukten. In dieser Liste sind für jedes Sprachkonstrukt der Namen, die Version sowie die Namen der zugeordneten Topics enthalten.

In seltenen Fällen kann eine Nachbearbeitung der erzeugten LaTeX–Dateien sinnvoll sein, etwa um mehrspaltigen Druck zu erzwingen.

1.7 Der Aufbau dieser Arbeit

Die weitere Arbeit ist wie folgt aufgebaut:

Im ersten Teil (Kapitel 1 bis 6) der Arbeit sind die theoretischen Aspekte, die Modellierung und Implementierung enthalten. In Kapitel 2 behandeln wir die Grundlagen aus den Bereichen Datenbanken und Synthesizer Generator. Die Modellierung der in der Datenbank darzustellenden Teilwelt wird in Kapitel 3 vorgestellt. Die Realisierung des Modells in der relationalen Datenbank ist in Kapitel 4 beschrieben, Kapitel 5 enthält die Beschreibung der Lösungsansätze und deren Implementierung. Ein Ausblick auf mögliche Erweiterungen wird in Kapitel 6 gegeben.

Der zweite Teil (Kapitel 7 bis 10) enthält ein Benutzerhandbuch, das unabhängig vom ersten Teil dieser Arbeit gelesen werden kann. Kapitel 7 gibt Hinweise zur Installation des NoName–Editors, die Verwendung dieses Editors wird in Kapitel 8 kurz erläutert. Kapitel 9 enthält eine komplette Beispielsitzung, die als „geführte Tour durch das NoName–Modul–Informationssystem“ verstanden werden kann. Das Kapitel 10 enthält eine Kurzreferenz, in welcher alle Menüpunkte des Menüs `NoMIS` sowie alle Buttons der in NoMIS verwendeten Dialogfenster aufgelistet sind.

Der Vollständigkeit halber sind die Quelldateien des Programms NoMIS im Anhang abgedruckt.

Danksagung

An dieser Stelle möchte ich Herrn Prof. Dr. H. Vogler und Herrn Prof. Dr. H. Partsch sowie Frau Dipl. Math. A. Mößle für ihre motivierte Betreuung und die Erstellung der Gutachten herzlich danken.

2 Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die für das Verständnis der weiteren Arbeit benötigt werden. Die angegebenen System–Voraussetzungen zielen bereits auf die in Kapitel 5 vorgestellte Implementierung. Der Abschnitt über Entity–Relationship–Modelle zeigt eine Möglichkeit, Teilwelten unserer realen Welt zu modellieren. Mit den relationalen Datenbanken werden wir uns ausführlicher beschäftigen und dabei auch einige Bezeichnungen und Schreibweisen einführen. Der letzte Teil dieses Kapitels beschreibt das Werkzeug zur Erstellung syntaxgesteuerter Editoren, den Synthesizer Generator. Dabei werden wir besonderen Wert auf die Schnittstelle zu C und die internen Datenstrukturen legen.

2.1 System-Voraussetzungen

Die in dieser Arbeit vorgestellte Software wurde mit dem C-Compiler `cc`, Version 1.122 vom 14. 04. 1989 auf einer SUN ELC erstellt. Unterstützt wird das Betriebssystem `SunOS` 4.2. Zur Laufzeit wird das relationale Datenbanksystem `Ingres` in Verbindung mit `OpenWin` bzw. `X11` benötigt.

2.2 Das Entity–Relationship–Modell

Entity–Relationship–Modelle (kurz: ER–Modelle) dienen zur Modellierung und Visualisierung eines Teils unserer realen Welt.

Entities, die durch beschriftete Rechtecke dargestellt werden, sind die identifizierbaren Objekte der zu modellierenden Teilwelt, also Autos, Häuser, Personen etc. Jedem Entity können mehrere Attribute zugeordnet werden, welche das Objekt näher spezifizieren. So kann das Entity *Auto* beispielsweise die Attribute Sitzplätze, Farbe und Hersteller besitzen, eine *Person* kann durch Alter, Haarfarbe und Namen beschrieben sein. Entities in der Notation des ER–Modells sind in der Abbildung 2.1 zu sehen.

Relationships (oder auch Relationen) werden durch beschriftete Rauten dargestellt und geben Auskunft über die Beziehung zweier Entities. Es kann z.B. eine Beziehung „besitzt“ existieren, welche die Besitzverhältnisse von Personen und deren Autos widerspiegelt. In Abbildung 2.2 ist diese Beziehung in die bereits genannten Entities eingebaut.

Eine wichtige Informationsquelle sind die Angaben bezüglich der Komplexität der Beziehungen. In Abbildung 2.2 steht neben dem Entity Auto der Buchstabe

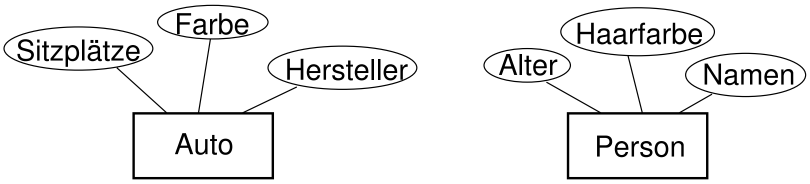


Abb. 2.1: Entities im ER–Modell

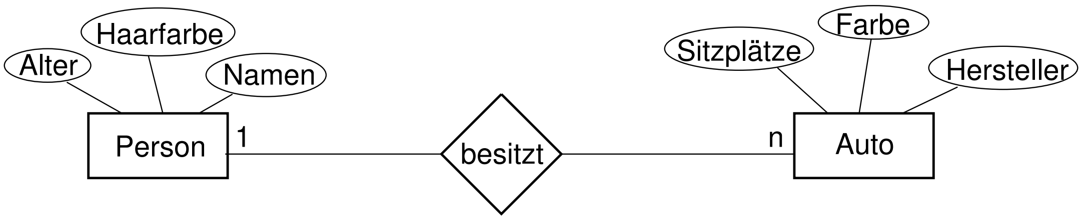


Abb. 2.2: Beziehung zwischen Entities

n, neben dem Entity Person ist die Ziffer 1 zu finden. Dies bedeutet, daß einer Person beliebig viele Autos gehörgen können (insbesondere auch kein Auto). Umgekehrt kann aber kein Auto existieren, das niemandem gehört. Neben diesen sogenannten 1:n–Beziehungen existieren noch n:m–Beziehungen. Eine n:m–Beziehung zwischen zwei Entities E_1 und E_2 bringt zum Ausdruck, daß beliebig viele Objekte des Entities E_1 beliebig vielen Objekten des Entities E_2 zugeordnet sein können. Hier besteht die Möglichkeit, einem Objekt des Entities E_1 *keine* Objekte des Entities E_2 zuzuordnen (und umgekehrt), d.h. es gilt $n \geq 0$ und auch $m \geq 0$.

In der Literatur finden sich noch weitere Notationen, z.B. die Min–Max–Notation zur genaueren Angabe der Komplexitäten. Mehr Informationen hierzu findet man in [EN89] und [Teo94].

2.3 Relationale Datenbanken

2.3.1 Hintergrund

Das *Relationale Datenmodell* wurde im Jahre 1970 von E. F. Codd vorgestellt. In den darauffolgenden Jahren wurden verschiedene Prototypen von relationalen Datenbanken vorgestellt, darunter auch Ingres, das an der University of California at Berkeley (UCB) entwickelt wurde. Das Ziel der Entwicklungen war, die bisher verwendeten sogenannten navigierenden Datenbanksprachen des hierarchischen Datenmodells (z.B. IMS) sowie des Netzwerk–Datenmodells (z.B. CODASYL) durch einfachere Konzepte abzulösen. Diese navigierenden

Sprachen arbeiteten auf baumartigen Datenstrukturen. In komplexen Anwendungen führten die verfügbaren Operationen wie „gehe zum ersten Sohn des aktuellen Knotens“ zu sehr umfangreichem und unübersichtlichem Programmcode. Von den Relationalen Datenbanken erhoffte man sich besonders in dieser Hinsicht eine enorme Verbesserung.

2.3.2 Das Basiskonzept Relationaler Datenbanken

Relationale Datenbanken stellen eine „Miniwelt“ — das ist ein kleiner Ausschnitt unserer realen Welt — in Tabellenform dar. Dabei ist die Tabelle die einzige verfügbare Datenstruktur. Eine Tabelle wird in der Fachsprache „Relation“ genannt. Die mathematische Grundlage für den Umgang mit Relationen bildet die Mengentheorie. Dabei sind nur einfache Operationen wie Durchschnitt, Vereinigung, Projektion, Selektion, Differenz und kartesisches Produkt zugelassen. Diese Operationen sind strukturerhaltend, d.h. sie führen Relationen wiederum in Relationen über. Mehr zum Thema „Relationen–Algebra“ findet sich in [DS94].

2.3.3 Bezeichnungen und Schreibweisen

Um im folgenden eine einheitliche Verwendung der „Fachbegriffe“ zu ermöglichen, sind nachfolgend die wichtigsten Begriffe und Schreibweisen kurz erläutert.

- Eine **Transaktion** ist eine Folge von Datenbank–Aktionen, die — eventuell zusammen mit anderen Aktionen — eine Einheit bilden, beispielsweise eine Kontobuchung oder Gehaltsabrechnung.

- Eine zweidimensionale Tabelle von Werten heißt **Relation**. Die Reihenfolge der Tabelleneinträge ist irrelevant. Eine Relation wird üblicherweise folgendermaßen visualisiert:

Name: R		
$Attr_1$	$Attr_2$	...
a	b	...
⋮	⋮	⋮

- Ein **Tupel** einer Relation R entspricht einer Zeile in einer Tabelle bzw. Relation. Ein Tupel enthält alle zusammengehörenden Werte bezüglich dieser Relation. Im obigen Beispiel ist

a	b	...

  ein Tupel. Auf den Tupeln in einer Relation ist keine Ordnung definiert.

- Ein **Attribut** einer Relation R entspricht einer Spalte in einer Tabelle. Alle Attribute einer Relation müssen einen eindeutigen Namen bezüglich dieser Relation haben. Im obigen Beispiel ist $Attr_2$ ein Attribut der Relation R. Wenn eine Relation R die Attribute A_i, $i = 1, 2, \ldots, n$ besitzt, dann schreiben wir für R auch $R(A_1, A_2, \ldots, A_n)$.

- Ein **Domain** (oder auch **Wertebereich**) ist eine Menge atomarer Werte, die aus der Sicht des Datenbanksystems „elementar" sind, d.h. die keine weitere Substruktur aufweisen. Jedem Attribut einer Relation R wird ein Domain zugeordnet. Für den Wertebereich D_j eines Attributs A_j einer Relation R schreiben wir $dom(A_j) = D_j$.

- Das **Schema** einer Relation R, bezeichnet mit $sch(R)$, ist die Menge der Attribute der Relation R. Wenn R die Attribute A_i, $i = 1, 2, \ldots, n$ besitzt, dann schreiben wir für das Schema von R: $sch(R) = \{A_1, A_2, \ldots, A_n\}$.

- $val(R) \subseteq dom(A_1) \times \ldots \times dom(A_n)$ entspricht dem **Wert einer Relation**.

- $t = \langle A_1{:}v_1, \ldots, A_n{:}v_n \rangle$ mit $v_i \in D_i$, $i = 1, 2, \ldots, n$ bezeichnet ein Tupel t aus einer Relation R mit $sch(R) = \{A_1, A_2, \ldots, A_n\}$, wobei die Reihenfolge der Attribute fest ist. Das Attribut A_i hat dabei den Wert v_i. Man schreibt mit gleicher Bedeutung auch $t = (v_1, \ldots, v_n), t \in val(R)$.

- Der Wert eines Attributes A_i, $i = 1, 2, \ldots, n$, in einem Tupel t wird mit $t(A_i) = v_i$ bezeichnet.

- $K \subseteq sch(R)$ heißt **Schlüsselkandidat** einer Relation R, wenn für alle Tupel t, t' aus R gilt:
  Wenn für alle Attribute $A_j \in K$ $t(A_j) = t'(A_j)$, dann ist $t = t'$ und K enthält nur die minimal mögliche Anzahl von Attributen.

- Ein ausgewählter Schlüsselkandidat heißt **Primärschlüssel**.

- F heißt **Fremdschlüssel** in der Relation R, falls F Primärschlüssel in der Relation S ist, und es muß gelten: $F \subseteq sch(R)$ und $F \subseteq sch(S)$.

- $A \in sch(R)$ ist **Schlüsselattribut**, wenn A Teil eines Schlüsselkandidaten ist.

- Attribute können niemals undefinierte Werte annehmen. Wenn einem Attribut kein expliziter Wert zugeordnet wird, dann enthält es den sogenannten **Nullwert**. Schlüsselattribute allerdings dürfen keine Nullwerte enthalten, sondern müssen immer mit expliziten Werten belegt werden.

Beziehungen zwischen Relationen werden ausschließlich über Attributwerte realisiert. Es existieren keine informationstragenden Verbindungen zwischen Relationen.

2.3.4 Die Anfrage–Sprache SQL

Es gibt mehrere Anfrage–Sprachen für relationale Datenbanken. Dazu zählen z.B. QBE (Query By Example, siehe [EN89]), das in DB2 von IBM Verwendung findet, QUEL (QUEry Language, siehe [EN89]), das zusammen mit Ingres entwickelt wurde, und SQL (Structured Query Language, siehe [MS93]), das inzwischen standardisiert und allgemein anerkannt ist.

Nachfolgend wird der Teil des SQL–Sprachumfanges beschrieben, der im Rahmen dieser Diplomarbeit benötigt wurde. Dazu zählen die folgenden Punkte:

- Beginn und Ende einer Transaktion.
- Das Select–From–Where–Konstrukt.
- Die Aggregat–Funktionen COUNT und MAX.
- Einfügen eines Tupels in eine Relation.
- Löschen eines Tupels aus einer Relation.

Innerhalb der vorgestellten SQL–Anweisungen bedeutet im folgenden:

|	Alternative
`[...]`	optionale Angabe
<u>Unterstreichung</u>	Default–Wert
serifenlose Schrift	Schlüsselwort aus SQL
kursive Schrift	Parameter der SQL–Anweisung

<u>Beginn und Ende einer Transaktion</u>

Transaktionen bilden eine „Klammer“ um zusammengehörende Datenbank–Aktionen. Eine derartige Klammerung ist nötig, damit sich mehrere gleichzeitig arbeitende Benutzer nicht gegenseitig stören bzw. damit die Datenbank konsistent gehalten werden kann.

Der Beginn einer Transaktion wird nicht explizit markiert. Jede Datenbank–Anfrage oder Datenbank–Manipulation impliziert den Beginn einer Transaktion, wenn die Anfrage bzw. Manipulation nicht innerhalb einer Transaktion stattfindet.

Das Ende einer Transaktion kann auf zweierlei Arten herbeigeführt werden. Entweder wird die Transaktion abgebrochen und die bereits erfolgten Änderungen zurückgenommen, oder die Transaktion wird korrekt beendet und die Änderungen konsistent in der Datenbank gehalten. Den ersten Fall bewirkt der Befehl ROLLBACK, den zweiten Fall bewirkt der Befehl COMMIT.

Das Select–From–Where–Konstrukt

Das zentrale Konstrukt in SQL ist das sogenannte „Select–From–Where“–Konstrukt, auch kurz mit „SFW“–Konstrukt bezeichnet.

> SELECT [ALL | DISTINCT] *Attributliste*
> FROM $R_1, \ldots, R_n$
> [WHERE *Prädikat*]
> [ORDER BY $Attr_1$ [ASC|DESC] [,$Attr_2$ [ASC|DESC]] ...]

Dabei bedeutet ALL, daß das Ergebnis Duplikate enthalten darf, DISTINCT eliminiert eventuell auftretende Duplikate.

Durch die Angabe einer *Attributliste* wird ein Teil des Schemas ausgewählt, welches durch das Produkt $R_1 \times \ldots \times R_n$ festgelegt wird. Die *Attributliste* kann unterschiedliche Form haben:

$R_i.A_{ij}, \ldots$	Dabei werden alle angegebenen Attribute A_{ij} der Relation R_i in die Ergebnismenge übernommen.
$R_i.*$	Alle Attribute der Relation R_i werden in die Ergebnismenge übernommen.
$*$	Alle Attribute von allen angegebenen Relationen werden in die Ergebnismenge übernommen.

Das *Prädikat* in der optionalen WHERE–Angabe kann unter anderem Vergleiche von Attributwerten und Konstanten mittels der Operatoren ($=, \neq, <$, AND, OR, ...) enthalten. Es werden nur Tupel in die Ergebnismenge aufgenommen, welche das angegebene Prädikat erfüllen.

Die optionale Angabe von ORDER BY bestimmt die Reihenfolge, in der die Ergebnistupel aus der Datenbank zurückgegeben werden sollen. Der Parameter $Attr_i$ muß dabei ein Attribut sein, das in der *Attributliste* aufgeführt ist. Mit den Parametern ASC bzw. DESC kann festgelegt werden, ob aufsteigend bzw. absteigend sortiert werden soll. Voraussetzung für die Verwendung von ORDER BY ist die Existenz einer Ordnung auf den Werten der angegebenen Attribute.

Die Aggregat–Funktion COUNT

SQL bietet standardmäßig einige sogenannte „Mengenfunktionen“ an, darunter auch COUNT. Mengenfunktionen treten im SFW–Konstrukt an die Stelle der *Attributliste* (siehe auch oben).

Die Funktion COUNT existiert in zwei Versionen:

COUNT($*$)

COUNT([DISTINCT] *Attributname*)

In der ersten Version wird die Mächtigkeit der Ergebnismenge berechnet. Zu beachten ist, daß die Ergebnismenge nicht duplikatfrei ist!

In der zweiten Version wird gezählt, wie viele verschiedene Werte das Attribut *Attributname* in der Ergebnismenge hat.

Ein Beispiel wird den Unterschied deutlich machen:
Nehmen wir eine Relation *Personal*, in welcher die Daten der Mitarbeiter einer Firma gespeichert werden. Sei das Schema der Relation *Personal* gegeben als $sch(Personal) = \{FamName, Vorname, GebDat, Beruf\}$. Desweiteren seien zehn Personen in dieser Firma beschäftigt, drei davon sollen Schmitt heißen (die anderen Namen seine paarweise disjunkt). Eine Anfrage unter Verwendung der Mengenfunktion COUNT(*) führt zu dem Ergebnis 10, die Verwendung von COUNT(DISTINCT FamName) ergibt 8, da nur acht verschiedene Familiennamen vorhanden sind.

Die Aggregat–Funktion MAX

Zu den Mengenfunktionen gehört auch die Funktion MAX. Sie wird verwendet wie folgt:

MAX(*Attributname*)

Diese Funktion liefert den maximalen Wert des Attributes *Attributname* in der Ergebnismenge.

Einfügen eines Tupels in eine Relation

Das Einfügen eines Tupels in eine Relation wird durch das folgende SQL–Konstrukt bewirkt:

INSERT INTO *Relationsname* [($Attr_1, Attr_2, \ldots$)]
VALUES ($Wert_1, Wert_2, \ldots$)

Dabei steht *Relationsname* für den Namen der Relation, in welche ein Tupel eingefügt werden soll. Wenn nicht alle Attribute der Relation mit Werten versehen werden sollen, dann kann eine Liste von Attributen $Attr_i$ angegeben werden, die mit Werten belegt werden müssen. Alle übrigen Attribute werden mit dem sogenannten Nullwert belegt. Da Schlüsselattribute keinen Nullwert enthalten dürfen, müssen diese in der Liste von Attributen zwingend angegeben werden!

Falls alle Attribute der Relation belegt werden sollen, dann kann die Angabe der Attributnamen entfallen; allerdings wird dann vorausgesetzt, daß die Reihenfolge der Werte im VALUES–Teil mit der definierten Reihenfolge der Attribute (bei CREATE TABLE, siehe [DS94]) übereinstimmt.

Löschen eines Tupels aus einer Relation

Tupel können aus einer Relation mit dem folgenden SQL–Konstrukt gelöscht werden:

DELETE FROM *Relationsname*
[WHERE *Prädikat*]

Dabei werden alle Tupel, welche das *Prädikat* erfüllen, aus der Relation *Relationsname* entfernt. Das *Prädikat* kann die gleichen Angaben umfassen wie im SFW–Konstrukt. Wenn die WHERE–Klausel weggelassen wird, dann werden alle Tupel der angegebenen Relation gelöscht. Die Relation selbst bleibt aber erhalten.

2.3.5 Embedded SQL

Embedded SQL (ESQL) ermöglicht die Verwendung von SQL in einer höheren Programmiersprache. Für diese sogenannte Host–Sprache muß ein Präprozessor existieren, der die SQL–Anweisungen in entsprechende Unterprogramm–Aufrufe der Host–Sprache übersetzt. Beim Übersetzen des Programms muß eine entsprechende Bibliothek eingebunden werden, welche die benötigten Funktionen zur Verfügung stellt. Die SQL–Syntax bleibt im wesentlichen identisch zu der in Abschnitt 2.3.4 beschriebenen Syntax. Zusätzliche Konstrukte ermöglichen die „Einbettung" der SQL–Anweisungen in das Programm. Auch kann die Behandlung von eventuell auftretenden Fehlern während der Ausführung einer SQL–Anweisung vom Programm erfolgen.

Die gemeinsame Nutzung von Variablen — d.h. Variablen, die sowohl in einer SQL–Anweisung als auch in anderen Teilen des Programmes Verwendung finden — wird durch eine besondere Kennzeichnung dieser Variablen ermöglicht. Möglichkeiten zur Kennzeichnung sind z.B. eine entsprechende Namensgebung oder das Voranstellen eines dafür reservierten Zeichens.

In der vorliegenden Arbeit wurde als Host–Sprache die imperative Programmiersprache C in Verbindung mit dem Datenbanksystem Ingres verwendet, da der Synthesizer Generator (siehe auch [Gra95]) eine Schnittstelle zu C anbietet und ein Präprozessor für Ingres verfügbar ist.

Der ESQL–Präprozessor für C erwartet vor jeder Anweisung, die er zu bearbeiten hat, die Markierung EXEC SQL. Die Variablen, die auch in SQL–Anweisungen verwendet werden, müssen separat deklariert werden und bei Verwendung in der SQL–Anweisung mit einem Doppelpunkt eingeleitet werden.

Wir werden die folgenden Punkte noch näher betrachten:

- Verwendung von ESQL vorbereiten.
- Verbindung mit der Datenbank aufbauen und abbrechen.
- Verwendung von Host–Variablen in SQL–Anfragen.
- Bearbeitung der Ergebnismenge in der Host–Sprache.
- Fehlerbehandlung in der Host–Sprache.

Dabei wird wiederum die Notation aus Kapitel 2.3.4 verwendet.

Verwendung von ESQL vorbereiten

Um von der Host–Sprache aus auf Embedded SQL zugreifen zu können, muß zuerst die sogenannte „Communication Area“ eingebunden werden. Dadurch wird der Übergabebereich für Daten von und zu der Datenbank eingerichtet. Eingebunden wird die Communication Area mit dem Befehl

EXEC SQL INCLUDE SQLCA;

Verbindung mit der Datenbank aufbauen/abbrechen

Um eine Verbindung mit der Ingres–Datenbank herzustellen, verwendet man folgenden Befehl:

EXEC SQL CONNECT *Datenbankname*;

Datenbankname ist dabei der vom Ingres–Systemadministrator vergebene Name für eine Datenbank.

Eine bestehende Verbindung zu einer Ingres–Datenbank wird mit

EXEC SQL DISCONNECT;

wieder abgebrochen.

Die Verwendung von Host–Variablen in SQL–Anfragen

Um Host–Variablen verwenden zu können, müssen diese gesondert deklariert werden. Dazu dient der folgende Deklarationsblock:

EXEC SQL BEGIN DECLARE SECTION;
    Deklarationen wie in C üblich
EXEC SQL END DECLARE SECTION;

Bei der Verwendung in einer SQL–Anweisung muß jeder in diesem Block deklarierten Variablen ein Doppelpunkt vorangestellt werden. Eine gültige SQL–Deklaration und SQL–Anfrage kann z.B. folgendermaßen aussehen:

EXEC SQL BEGIN DECLARE SECTION;
    domA my_a;
    domB my_b;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT A, B INTO :*my_a*, :*my_b* FROM *table*;

Den Variablen *my_a* und *my_b* werden Werte der Attribute A und B der Relation *table* zugewiesen. Daher müssen die Variablen *my_a* und *my_b* in einer DECLARE SECTION entsprechend den Attribut–Domains von A und B deklariert werden, d.h. der Wertebereich des Typs *domA* muß eine Obermenge von $dom(A)$ sein und der Wertebereich des Typs *domB* muß eine Obermenge von $dom(B)$ sein.

Bearbeitung der Ergebnismenge in der Host–Sprache

Die Einbettung von SQL–Anfragen in eine Host–Sprache wirft ein grundlegendes Problem auf: Wie können die Ergebnisse einer Anfrage in der Host–Sprache weiterverarbeitet werden? Die Ergebnismenge der SQL–Anfrage ist – wie der Name bereits sagt – eine *Menge*. Viele Host–Sprachen, darunter auch C, kennen keinen Mengentyp, sondern nur einfache und strukturierte Typen. Die Ergebnismenge muß daher von der Host–Sprache sequentiell bearbeitet werden. Hierfür existieren in Ingres zwei grundlegende Verfahrensweisen: das auch von anderen Systemen verwendete Cursor–Konzept sowie eine Ingres–spezielle Erweiterung des SFW–Konstruktes. Letzteres eignet sich immer dann, wenn z.B. die Ergebnisse einer Anfrage auf dem Bildschirm angezeigt werden sollen. Das erweiterte SFW–Konstrukt sieht wie folgt aus:

EXEC SQL SELECT [<u>ALL</u> | DISTINCT] *Attributliste*
FROM $R_1, \ldots, R_n$
[WHERE *Prädikat*]
[ORDER BY $Attr_1$ [ASC|DESC] [,$Attr_2$ [ASC|DESC]] . . .]

```
EXEC SQL BEGIN
Verarbeitung in der Host–Sprache
EXEC SQL END
```

Die Erweiterung beschränkt sich auf die letzten drei Zeilen. Die Anweisungen zwischen EXEC SQL BEGIN und EXEC SQL END werden automatisch für jedes Ergebnistupel ausgeführt. Dies bedeutet, daß der Programmierer keine explizite Schleife angeben muß. Allerdings dürfen in diesem Programmteil keine weiteren SQL–Anweisungen verwendet werden. Für Anwendungen, in welchen die Erweiterung des SFW–Konstruktes nicht ausreicht, muß das Cursor–Konzept verwendet werden.

Ein Cursor ist nichts anderes als ein Pointer auf ein Tupel des Anfrage–Ergebnisses. Auf einem Cursor können die Operationen DECLARE, OPEN, FETCH und CLOSE ausgeführt werden. Nachfolgend sind die Operationen näher beschrieben.

Die Deklaration eines Cursors geschieht mit der folgenden Anweisung:

```
EXEC SQL DECLARE cursorname CURSOR FOR
SQL–Anfrage;
```

Der *cursorname* muß ein im gesamten Programm eindeutiger Name sein, d.h. man kann nicht zwei verschiedene Cursor mit dem gleichen Namen versehen.

Durch die Deklaration eines Cursors wird die SQL–Anfrage nicht ausgeführt. Die Ausführung muß explizit angestoßen werden mit der Anweisung

```
EXEC SQL OPEN cursorname;
```

Anschließend kann in einer Schleife die Ergebnismenge abgearbeitet werden. Der Zugriff auf ein Tupel der Ergebnismenge wird durch folgende Anweisung ermöglicht:

```
EXEC SQL FETCH cursorname INTO :var1, :var2, ...;
```

Für jedes Tupel der Ergebnismenge muß genau ein FETCH durchgeführt werden. Die Abbruchbedingung der Schleife ergibt sich aus der Ingres–Systemvariablen *sqlca.sqlcode*, die auf die Existenz weiterer Ergebnis–Tupel abgefragt werden kann. Ein mögliches Programmfragment lautet etwa:

```
while(sqlca.sqlcode == INGRES_OK) {
    EXEC SQL FETCH cursorname INTO :var1, :var2, ...;
    Verarbeitung in der Host–Sprache
}
```

Bei der Verarbeitung der Rückgabewerte kann in der üblichen Weise auf die Variablen $var1$, $var2$, ... zugegriffen werden.

Nachdem alle Ergebnistupel auf die oben dargestellte Weise abgearbeitet sind, muß der Cursor geschlossen werden. Dies geschieht mit der folgenden Anweisung:

EXEC SQL CLOSE *cursorname*;

Fehlerbehandlung in der Host–Sprache

Im Gegensatz zu interaktiven SQL–Tools, wie etwa `sqlplus` oder `isql`, bei denen Fehlermeldungen einfach auf dem Bildschirm angezeigt werden, kann der Programmierer von Embedded–SQL–Anwendungen das Fehlerverhalten direkt beeinflussen. So kann z.B. im Falle eines Fehlers an eine konkrete Marke im Programm gesprungen werden. Dazu dient die folgende Anweisung:

EXEC SQL WHENEVER SQLERROR GOTO *marke*;

Ebenso kann auch eine Funktion aufgerufen werden:

EXEC SQL WHENEVER SQLERROR CALL *error_procedure()* ;

Oftmals ist es nützlich, den Fehler genauer bestimmen zu können. Dazu kann eine Beschreibung des aufgetretenen Fehlers vom Datenbanksystem erfragt werden:

EXEC SQL INQUIRE_SQL (*:msg* = ERRORTEXT);

Der Wert der Variablen *msg* kann z.B. dem Benutzer angezeigt werden. Es besteht auch die Möglichkeit, die Fehlerbehandlung wieder auszusetzen. Dies geschieht mit folgender Anweisung:

EXEC SQL WHENEVER SQLERROR CONTINUE;

Dadurch kann man sicherstellen, daß z.B. der Abbruch einer Datenbankverbindung auch trotz dem eventuellen Auftreten eines Fehlers durchgeführt wird. In den meisten Fällen kann bei einem fehlerhaften Abbruch der Datenbankverbindung auch keine korrekte Fehlerbehandlung mehr erfolgen, wodurch wiederum ein Fehler generiert wird. In diesem Fall befindet sich das Programm in einer Endlosschleife, die nur noch durch einen manuellen Eingriff beendet werden kann.

Weitergehende Informationen zu SQL und Embedded SQL können [MS93] und [DS94] entnommen werden.

2.4 Der Synthesizer Generator

2.4.1 Allgemeines

Der Synthesizer Generator ist ein Werkzeug zur Erstellung sogenannter syntaxgesteuerter Editoren. Eine relativ ausführliche Beschreibung findet sich in [Gra95], daher wird in dieser Arbeit nur das Nötigste beschrieben.

Ein syntaxgesteuerter Editor ist ein einfach zu bedienender, auf eine spezielle Programmiersprache zugeschnittener Editor, der dem Programmierer den menügesteuerten top–down Entwurf eines Programms ermöglicht. Dem Editor liegt eine kontextfreie Grammatik zugrunde, durch welche die Syntax der Programmiersprache — in unserem Fall ist dies die funktionale Programmiersprache NoName — festgelegt ist. Wir wollen den syntaxgesteuerten Editor für die Programmiersprache NoName im folgenden NoName–Editor nennen.

Bei einem top–down Entwurf eines Programms z.B. mit dem NoName–Editor werden Verletzungen der kontextsensitiven Nebenbedingungen der unterstützten Programmiersprache sofort erkannt und als Fehlermeldung dem Programmierer sichtbar gemacht. Hat er z.B. im Deklarationsteil einer Funktion einen Typbezeichner verwendet, der bisher selbst nicht deklariert war, so kann ihm dies direkt gemeldet werden. Eine nachfolgende Deklaration des unbekannten Bezeichners löscht die Fehlermeldung. Die kontextsensitiven Nebenbedingungen werden durch sogenannte Attributgrammatiken festgelegt.

Ein mit dem Synthesizer Generator erstellter syntaxgesteuerter Editor stellt das momentan editierte Programm als Syntaxbaum im Hauptspeicher dar. Der Syntaxbaum — und damit das Programm — ergibt sich aus den beim top–down Entwurf angewendeten Produktionen der zugrundeliegenden kontextfreien Grammatik.

Der Synthesizer Generator stellt ein Interface zu der Programmiersprache C zur Verfügung, mit welchem eigene Funktionen etc. in den Editor integriert werden können. Von diesem Interface wurde in der vorliegenden Arbeit Gebrauch gemacht, um den Datenbank–Anschluß (siehe dazu Kapitel 5.2) sowie eigene Dialogfenster zur Kommunikation mit der Datenbank zu realisieren. Wir werden in diesem Kapitel in knapper Form einen Ausschnitt aus dem C–Interface sowie die SGen–internen Datenstrukturen und Operationen auf diesen beschreiben, die z.B. zur Manipulation des Syntaxbaumes benötigt werden.

Die Synthesizer Specification Language (SSL), in welcher ein syntaxgesteuerter Editor spezifiziert wird, spielt in dieser Arbeit nur eine untergeordnete Rolle. Daher wird diese Sprache hier auch nicht vorgestellt.

2.4.2 Erzeugen eines neuen Menüs und der Menüpunkte

Das Interface zu C erlaubt die Deklaration von Menüs, welche in die Menüzeile des generierten syntaxgesteuerten Editors aufgenommen werden. Das Schema für eine derartige Deklaration ist im folgenden beschrieben.

Das Interface stellt eine Dummy–Funktion

init_sys3()

zur Verfügung, die für die Deklaration von Menüs und Dialogfenstern überschrieben werden kann. In dieser Funktion müssen die folgenden Schritte ausgeführt werden:

1. Zuerst benötigt man eine Variable des Typs ME, der vom Synthesizer Generator vorgegeben ist. Dieser Variablen wird der Rückgabewert der Funktion

   me_open(*Menü–Name*)

   zugewiesen.

2. Anschließend wird für jeden benötigten Menüpunkt die Funktion

   co_open(P_1,P_2,P_3)

   aufgerufen, wobei P_1 der Name des Menüpunktes, P_2 der Name der auszuführenden Kommando–Funktion, und P_3 der Name der sogenannten A–Testfunktion[3] sind. Die A–Testfunktion prüft, ob der Menüpunkt aktivierbar ist oder nicht (d.h. diese Funktion muß entweder true oder false zurückgeben). Die Kommando–Funktion wird ausgeführt, wenn der Anwender den Menüpunkt gewählt hat.

3. Zuletzt wird jeder deklarierte Menüpunkt einem Menü zugeordnet. Dazu wird für jeden Menüpunkt die Funktion

   MenuItem(P_1,P_2)

   aufgerufen, wobei P_1 ein Menüpunkt–Name und P_2 die oben genannte Variable des Typs ME sind.

Damit ist die Deklaration der Menüs beendet.

[3] A–Testfunktion steht für *activity test function.*

2.4.3 Erzeugen von neuen Dialogen

Analog zur Deklaration neuer Menüs müssen auch die Deklarationen neuer Dialoge in der Funktion init_sys3() erfolgen. Die Deklaration neuer Dialoge ist um einiges aufwendiger als die Deklaration neuer Menüs.

Jeder Dialog muß einen im ganzen Programm eindeutigen Namen erhalten. Der Grund liegt in der Verwaltung der Dialoge: Jeder deklarierte Dialog wird in einer Tabelle unter seinem Namen abgelegt. Wird später eine Instanz eines speziellen Dialoges erzeugt, so werden die benötigten Dialog–Daten aus dieser Tabelle extrahiert. Da der Zugriff auf die Tabelle über den Dialog–Namen erfolgt, muß dieser notwendigerweise eindeutig sein.

Die Registrierung eines neuen Dialoges erfolgt mit der Funktion

bd_register_dialog(P_1,...,P_{11});

P_1 ist dabei der eindeutige Dialog–Name. Für das Verständnis sind auch die Parameter P_4 bis P_9 wichtig. Sie enthalten Namen von Funktionen, welche zu definierten Zeitpunkten aufgerufen werden. P_4 verweist auf die Funktion, welche die Dialogelemente in dem Dialogfenster anordnet. Die Initialisierung dieser Dialogelemente wird in der von P_5 referenzierten Funktion vorgenommen. Beim Verlassen des Dialoges wird die Funktion P_6 aufgerufen. Sobald der Benutzer einen Button in dem Dialogfenster drückt, wird die Funktion P_7 aufgerufen. In dieser Funktion können die Werte der Dialogelemente abgefragt werden. P_8 verweist auf eine Funktion, die beim Schließen des Dialoges besondere Arbeiten (z.B. Freigabe von allokiertem Speicher) übernehmen kann. Der Parameter P_9 verweist auf eine Funktion, die nach dem Wählen eines Buttons im Dialog ausgeführt wird (dieser Parameter ist nur in wenigen Spezialfällen tatsächlich belegt, in der vorliegenden Arbeit wird er nicht verwendet). Wenn die Parameter P_6, P_7, P_8 und P_9 auf den Wert (void (*)()) 0 gesetzt werden, dann wird zu den entsprechenden Zeitpunkten keine Funktion aufgerufen. Die übrigen Parameter (P_2, P_3, P_{10} und P_{11}) geben lediglich Auskunft über bestimmte Dialogelemente und ähnliches; wir wollen sie daher nicht näher betrachten.

Ein registrierter Dialog wird vom Programm aus mit der Funktion

bd_open(*cmd, dialog_name, default, num_buttons, b_actions*);

geöffnet. Der Parameter *cmd* gibt die Überschrift des Dialogfensters an. In *dialog_name* wird der Name des registrierten Dialogs übergeben. *default* enthält den Namen des Buttons, der automatisch aktiviert wird, falls der Benutzer die RETURN–Taste drückt. Die Anzahl der Buttons steht in *num_buttons*. Der Parameter *b_actions* enthält ein Array von Tupeln, in denen für jeden Button festgelegt wird, welche Aktion bei Auswahl dieses Buttons durchgeführt werden soll. Wird für einen Button keine Aktion angegeben, so wird beim Drücken dieses Buttons das Dialogfenster nur geschlossen.

2.4.4 SGen–interne Datenstrukturen

Um verstehen zu können, wie ein Syntaxbaum des syntaxgesteuerten Editors manipuliert werden kann, ist die Kenntnis der Datenstrukturen des SGen eine unverzichtbare Voraussetzung. Der Leser aber, der sich nicht in die Details der Implementierung vertiefen möchte, kann dieses Unterkapitel getrost überspringen.

Da die Firma GrammaTech keine Dokumentation ihrer Implementation bereitstellen wollte oder konnte, wurde eine aufwendige Suche nach Funktionen, Typen und Vorgehensweisen zur Manipulation des Syntaxbaumes in deren Quellcode (ungefähr 4MB) unumgänglich. Die dabei gewonnenen Erkenntnisse, die mit Sicherheit nicht sämtliche Aspekte der SGen–Implementierung abdecken, aber für das Verständnis der Implementierung von NoMIS völlig ausreichend sind, werden in den Abbildungen 2.3 und 2.5 graphisch veranschaulicht.

In Abbildung 2.3 wird der am häufigsten benötigte „Pfad“ durch die Strukturen gezeigt. Alle Informationen, die der syntaxgesteuerte Editor zu einem Editorfenster gespeichert hat, werden durch die Struktur `_browser` zur Verfügung gestellt. Zu jedem derartigen Record existiert ein Zeiger des Typs `BROWSER`, der als „Anker“ in der abgebildeten Kette von Verweisen fungiert. Der Anker für die Datenstruktur des gerade aktuellen Editorfensters wird immer in der globalen Variablen `cur_browser` gespeichert.

Jeder `_browser`–Record enthält neben seinem Namen und Verweisen auf andere Browser auch eine Komponente namens `buf`, welche einen Zeiger des Typs `BUFFER` auf eine Struktur des Typs `_buffer` enthält. Um vom `_browser`–Record zum `_buffer`–Record zu gelangen, muß man nur diesem Verweis folgen. In der Abbildung 2.3 ist dieser Verweis mit `[1]` markiert. Das Verfolgen des Verweises kann betrachtet werden als das Selektieren einer Unterstruktur aus dem `_browser`–Record. Deshalb kann man man diese Selektion durch eine sogenannte *Zeiger–Selektion* realisieren. Zeiger–Selektionen sehen äußerlich wie eine Funktion aus, sie sind aber nur Definitionen für den Präprozessor, der sie in eine gewöhnliche Zeiger–Verfolgung übersetzt.

Der SGen bietet dem Programmierer für den Übergang vom `browser`–Record zum `_buffer`–Record die Zeiger–Selektion `br_buf` an, die folgendermaßen definiert ist[4]:

```
br_buf: BROWSER → BUFFER
#define br_buf(b) ((b)->buf)
```

[4]Im folgenden werden immer die typisierte Definition, wie sie z.B. von Funktionsdefinitionen bekannt ist, sowie die Definition im C–Quelltext angegeben.

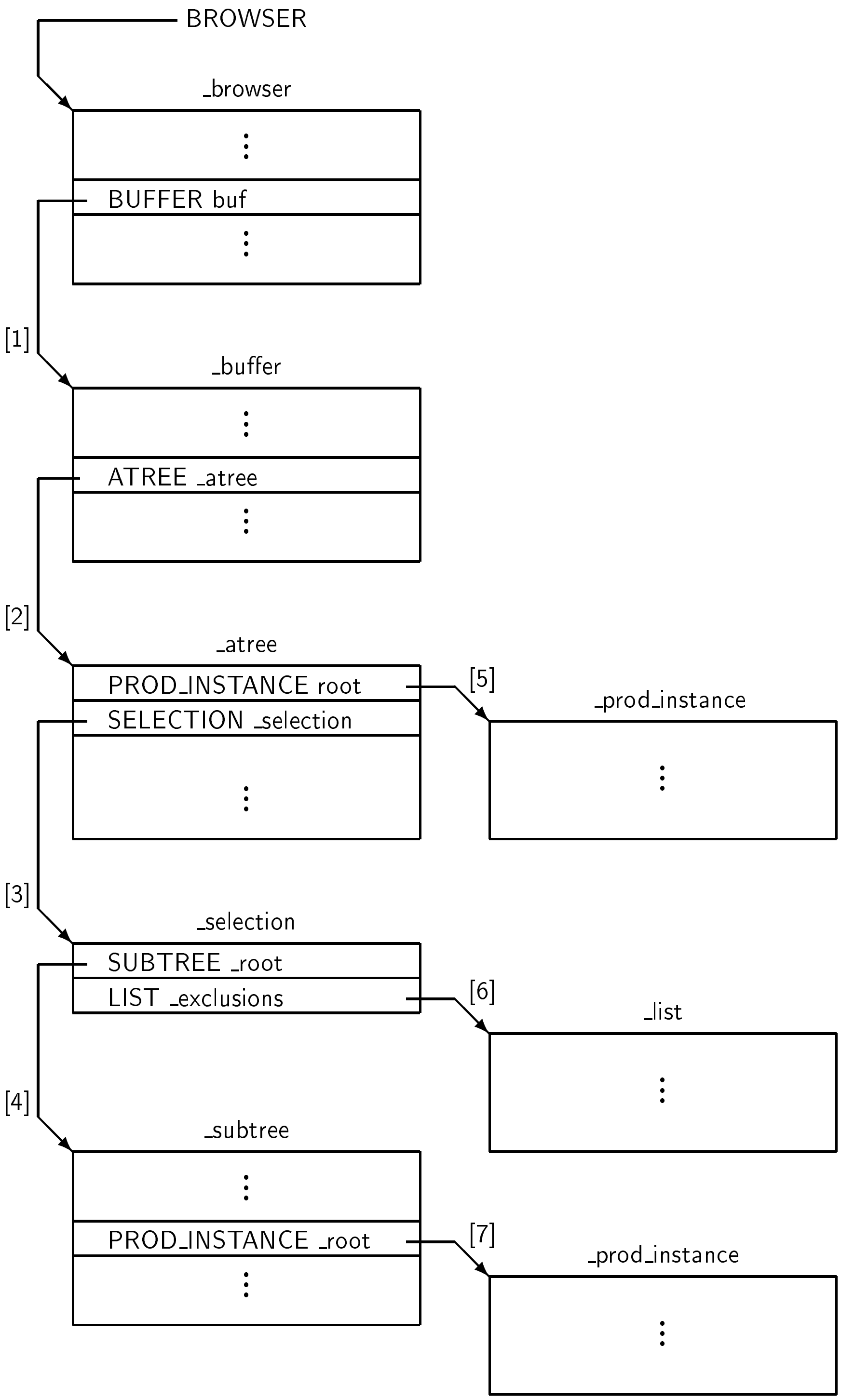


Abb. 2.3: SGen–Datenstruktur zur Verwaltung eines abstrakten Syntaxbaumes

In einem `_buffer`–Record werden unter anderem der Name des Buffers, der Name der editierten Datei, das Dateiformat und verschiedene Statusinformationen gespeichert. Daneben enthält jeder `_buffer`–Record einen Verweis auf eine Struktur des Typs `_atree`, in der das editierte Programm in der Form des zugehörigen abstrakten Syntaxbaumes gespeichert ist. Für diesen Übergang, der in der Abbildung 2.3 mit `[2]` markiert ist, steht ebenfalls eine Zeiger–Selektion zur Verfügung, die folgendermaßen definiert ist:

```
bu_atree: BUFFER → ATREE
#define bu_atree(b) ((b)->_atree)
```

Ein `_atree`–Record enthält den attributierten Syntaxbaum, der das editierte NoName–Programm repräsentiert. Dazu gehören auch Informationen über den momentanen Bearbeitungsstatus, z.B. die Information, ob der Syntaxbaum geändert wurde oder ob die Selektion neu gesetzt wurde.

Zwei Struktur–Komponenten von `atree` sind einer gesonderten Betrachtung wert: die Wurzel des Syntaxbaumes, repräsentiert durch die Komponente `root` des Typs `PROD_INSTANCE` (wird später erläutert), und der momentan markierte Teil des Programms im NoName–Editor, repräsentiert durch die Komponente `_selection` des Typs `SELECTION`. Der Übergang von dem `_atree`–Record zu dem `_selection`–Record wird wiederum durch eine Zeiger–Selektion ermöglicht (Markierung `[3]` in Abbildung 2.3). Die Definition dieser Zeiger–Selektion lautet folgendermaßen:

```
SE_selection: ATREE → SELECTION
#define SE_selection(t) ((t)->_selection)
```

Ein `_selection`–Record referenziert den im NoName–Editor aktuell markierten Teilbaum des gesamten Syntaxbaumes. Die Komponente `_root` des Typs `SUBTREE` zeigt auf die Wurzel des markierten Teilbaumes. Die Speicherung der Wurzel–Adresse allein reicht allerdings immer dann nicht aus, wenn der Benutzer ein Element einer Liste markiert. Der Grund liegt darin, daß die Listen des Synthesizer Generators intern als Baum repräsentiert werden, wie es in Abbildung 2.4 zu sehen ist.

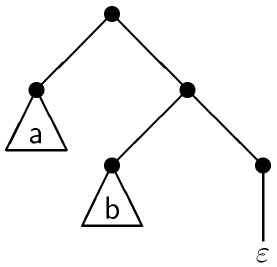


Abb. 2.4: Darstellung einer Liste als Baum

Die Wurzel dieses Baumes ist oben, die repräsentierte Liste enthält zwei Elemente, genannt `a` und `b`.

Wenn der Benutzer z.B. den Teilbaum `a` markieren möchte, dann wird — je nach Realisierung des sogenannten „unparsing scheme“, siehe [Gra95] — nicht der unmittelbare Wurzelknoten von `a`, sondern dessen Vaterknoten in der Komponente `_root` der `_selection`–Struktur gespeichert. Ohne weitere Vorkehrungen hätte dies zur Folge, daß *alle* Listenelemente beginnend bei dem gewünschten Element im Editor markiert wären. Deshalb existiert in der `_selection`–Struktur die Komponente `_exclusions` des Typs `LIST`, die auf eine Liste zeigt, in welcher alle nicht zu markierenden Listenelemente „unterhalb“ des gewünschten Elements enthalten sind. Wenn im Editor das letzte Element einer Liste oder ein strukturell anderes Konstrukt markiert ist, dann enthält die Komponente `_exclusions` einen Nullzeiger.

Der wichtige Teil der `_selection`–Struktur ist im folgenden der markierte Teilbaum, also die Struktur, auf welche die Komponente `_root` zeigt. Um zu der `_subtree`–Struktur zu gelangen, kann man wiederum eine Zeiger–Selektion verwenden (Markierung `[4]` in Abbildung 2.3), die folgendermaßen definiert ist:

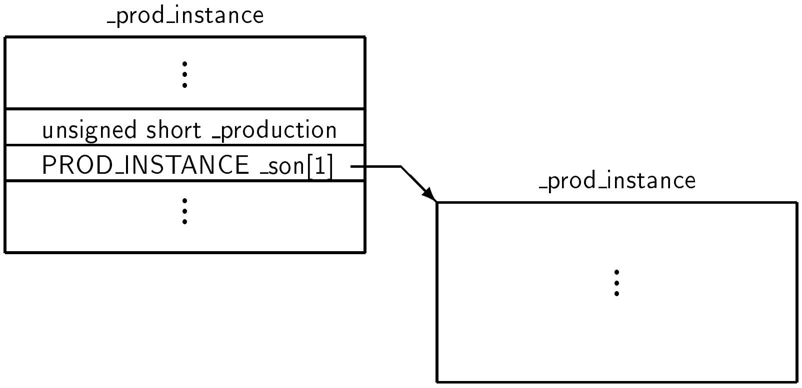
```
selection_root: SELECTION → SUBTREE
#define selection_root(s) ((s)->_root)
```

Ein `_subtree`–Record enthält einen Verweis auf die Wurzel des im NoName–Editor markierten Teilbaumes sowie alle weiteren Informationen, die zu diesem Teilbaum gehören. Der Verweis auf die Wurzel wird in der Komponente `_root` des Typs `PROD_INSTANCE` gespeichert. Der Aufbau einer `_prod_instance`–Struktur ist in Abbildung 2.5(a) zu sehen.

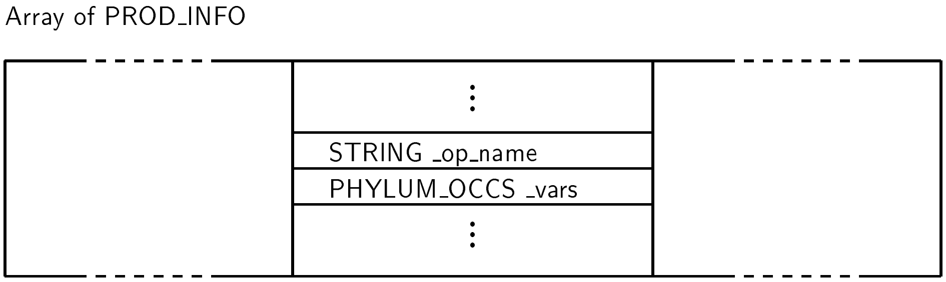
Nach unseren Erkenntnissen wird der abstrakte Syntaxbaum durch einen Baum von `_prod_instance`–Records aufgebaut. Das heißt, daß für jede Instanz einer Produktion der zugrundeliegenden Grammatik genau ein `_prod_instance`–Record vorhanden ist. Diese Erkenntnis stützt sich auf zwei Komponenten der `_prod_instance`–Struktur: Zum einen ist dies die Komponente `_production` des Typs `unsigned short`, welche die angewendete Produktion eindeutig numerisch spezifiziert. Zum anderen existiert die Komponente `_son[1]` des Typs `PROD_INSTANCE`, die auf eine Verkettung von `_prod_instance`–Records schließen läßt.

Der Zugriff auf den i-ten Sohn eines Knotens p, d.h. der Zugriff auf das Element `_son[i]` eines `_prod_instance`–Records, welches durch p referenziert wird, erfolgt mit der Zeiger–Selektion `son(p,i)`:

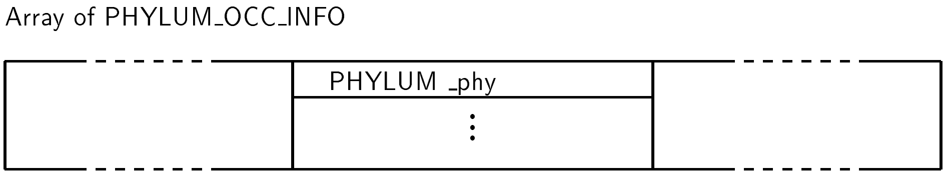
```
son: PROD_INSTANCE × CARDINAL → PROD_INSTANCE
#define son(p,i) (((p) -> _son)[i])
```



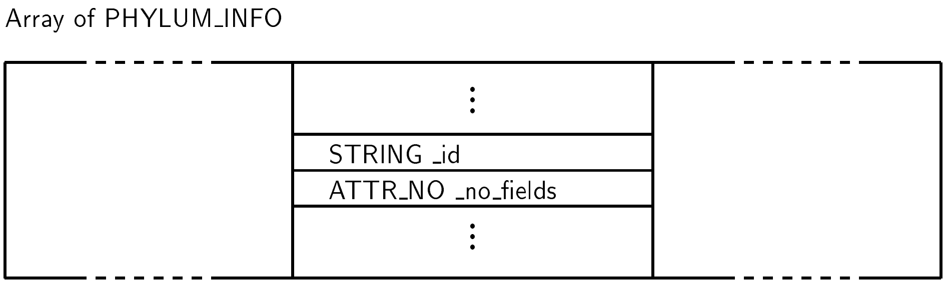
(a)



(b)



(c)



(d)

Abb. 2.5: Die `_prod_instance`–Struktur (a), ein `PROD_INFO`–Array (b), ein `PHYLUM_OCC_INFO`–Array (c) und ein `PHYLUM_INFO`–Array (d).

In den `_prod_instance`–Records wird, wie bereits angedeutet, eine numerische Codierung als Verweis auf die angewendete Produktion verwendet. Dadurch müssen produktionsspezifische Informationen, wie z.B. der Name, nur *einmal* gespeichert werden. Diese Informationen werden in einem Array namens `_production` abgelegt, das aus `PROD_INFO`–Strukturen besteht (siehe Abbildung 2.5(b)). Dabei wird jeder Produktion der Grammatik, welche dem syntaxgesteuerten Editor zugrunde liegt, genau ein `PROD_INFO`–Record und damit auch ein eindeutiger Index in diesem Array zugeordnet. Dieser Index wird von den `_prod_instance`–Records in der Komponente `_production` verwendet, um auf die angewendete Produktion zu verweisen. Der Zugriff auf diesen Index erfolgt durch die Struktur–Selektion `production`, die folgendermaßen definiert ist:

```
production: PROD_INSTANCE → CARDINAL
#define production(p) ((p) -> _production)
```

Die `PROD_INFO`–Struktur enthält eine Komponenete namens `_op_name` des Typs `STRING`, in welcher der Name der Produktion gespeichert wird. Um diesen Namen zu ermitteln, kann man die Struktur–Selektion `op_name(x)` verwenden, die folgendermaßen definiert ist:

```
op_name: CARDINAL → STRING
#define op_name(x) (_production[x]._op_name)
```

Neben der `PROD_INFO`–Struktur existieren noch die `PHYLUM_INFO`–Struktur und die `PHYLUM_OCC_INFO`–Struktur (siehe Abbildungen 2.5(d) und 2.5(c)). Für jedes Nichtterminalsymbol der zugrundeliegenden Grammatik — in der Sprache des SGen heißen die Nichtterminalsymbole *Phylum* — existiert genau ein `PHYLUM_INFO`–Record. Darin werden unter anderem der Name des Nichtterminalsymbols in der Komponente `_id` und die Anzahl der zugeordneten Attribute in der Komponente `_no_fields` gespeichert. Die `PHYLUM_INFO`–Records werden in dem Array `_phylum` abgelegt.

Damit die verschiedenen Vorkommen eines Nichtterminalsymbols unterschieden werden können, existiert für jedes Vorkommen genau ein `PHYLUM_OCC_INFO`–Record. Dieser enthält einen Verweis auf den zugehörigen `PHYLUM_INFO`–Record, der aus dessen Index im Array `_phylum` besteht und in der Komponente `_phy` abgelegt wird. Desweiteren werden in den `PHYLUM_OCC_INFO`–Records Verweise auf bzw. Informationen zu den entsprechenden Attributvorkommen gespeichert.

Die `PHYLUM_OCC_INFO`–Records werden in einem Array namens `_phylum_occs` angeordnet. Dabei wird eine bestimmte Ordnung eingehalten: Wenn in der Grammatik eine Produktion `A` → `BC` existiert und `A` am Index n im Array `_phylum_occs` abgelegt ist, dann werden die `PHYLUM_OCC_INFO`–Records für die

Nichtterminalsymbole `B` und `C` an den Positionen $n + 1$ und $n + 2$ im Array `_phylum_occs` gespeichert. Somit existiert auch hier ein eindeutiger Index für jedes Vorkommen eines Nichtterminalsymbols.

Wenn der Name eines Nichtterminals gesucht ist und der Index x desselben im Array `_phylum` bekannt ist, dann kann die Struktur–Selektion `sym_id(x)` verwendet werden:

```
sym_id: CARDINAL → STRING
#define sym_id(x) (_phylum[x]._id)
```

Neben diesen hier aufgeführten Zeiger– und Struktur–Selektionen, welche jeweils nur von einer Struktur zur nächsten wechseln, existieren noch mehrstufige Selektionen. Deren Definitionen sind nachfolgend mit einer kurzen Beschreibung aufgeführt:

```
br_atree: BROWSER → ATREE
#define br_atree(b) (bu_atree(br_buf(b)))
```

`br_atree(b)` liefert zu einem Zeiger `b` des Typs `BROWSER` den Verweis auf dessen zugehörigen attributierten Syntaxbaum, d.h. das Ergebnis ist ein Zeiger des Typs `ATREE`.

```
selection_apex: SELECTION → PROD_INSTANCE
#define selection_apex(s) (subtree_root(selection_root(s)))
```

Bei einem gegebenen Zeiger `s` auf einen `_selection`–Record liefert die Selektion `selection_apex(s)` den Verweis auf den `_prod_instance`–Record, welcher der Wurzel des betrachteten Baumes zugeordnet ist.

```
lhs: CARDINAL → STRING
#define lhs(x)   (_phylum_occs\
                 [((_production[x]._vars))]._phy)
rhs: CARDINAL × CARDINAL → STRING
#define rhs(x,i) (_phylum_occs\
                 [((_production[x]._vars)+i)]._phy)
```

`lhs(x)` liefert den Namen des Nichtterminalsymbols, welches auf der linken Seite der Produktion vorkommt, die in dem Array `_production` am Index x gespeichert ist. Analog hierzu arbeitet die Selektion `rhs(x,i)`, der Unterschied besteht lediglich darin, daß das gesuchte Nichtterminalvorkommen am Index `n+i` gespeichert ist. Es ist leicht einzusehen, daß `lhs(x)` nichts anderes ist als `rhs(x,0)`.

In der Abbildung 2.3 sind der Vollständigkeit halber noch weitere Zeiger angegeben (Markierungen `[5]`, `[6]` und `[7]`). Diese sind in der vorliegenden Arbeit jedoch nicht von Bedeutung und daher auch nicht näher beschrieben.

Wir wollen zum Abschluß dieser theoretischen Grundlagen die Verwendung einiger Selektionen anhand eines Beispiels betrachten. Nehmen wir an, wir möchten den Namen des Nichtterminalsymbols auf der linken Seite der Produktion ermitteln, die an der Wurzel des momentan markierten Teilbaumes im NoName–Editor angewendet ist. Dazu definieren wir uns zunächst die Variable `s` des Typs `STRING`. Die Ermittlung des Namens mit gleichzeitiger Zuweisung desselben an diese Variable lautet dann wie folgt:

```
s = lhs(selection_apex(SE_selection(br_atree(cur_browser))))
```

An diesem Beispiel wird deutlich, wie schnell Ausdrücke durch Schachtelung der verschiedenen Selektionen lang und unlesbar werden. In Abschnitt 5.4 sind einige Funktionen beschrieben, welche die Verwendung der hier vorgestellten Selektionen deutlich vereinfachen bzw. überflüssig machen.

3 Funktionalität des Informationssystems

Bevor wir auf die einzelnen Aspekte der Funktionalität des NoName–Modul–Informationssystems näher eingehen, wollen wir zwei Entity–Relationship–Modelle diskutieren, die beide zur Lösung unserer Aufgabe verwendet werden könnten. Eines davon werden wir der Implementierung in Kapitel 5 zugrunde legen.

3.1 Das ER–Modell

In diesem Abschnitt werden wir zwei ER–Modelle diskutieren. Eines davon wurde in der Implementierung verwendet. Zuerst wollen wir uns die etwas erweiterte Aufgabenstellung in Erinnerung rufen:

> Es soll ein Programm erstellt werden, mit dem Sprachkonstrukte der funktionalen Programmiersprache NoName in einer Datenbank verwaltet werden können.
>
> Der Benutzer soll durch Markieren eines Sprachkonstruktes und Selektion eines Menüpunktes im NoName–Editor in der Lage sein, das markierte Sprachkonstrukt in die Datenbank einzufügen. Das Sprachkonstrukt kann mehreren Suchkriterien zugeordnet werden, über welche zu einem späteren Zeitpunkt auch die Suche in der Datenbank erfolgt.
>
> Umgekehrt soll der Benutzer auf ebenso einfache Weise ein gespeichertes Sprachkonstrukt aus der Datenbank erfragen und an der aktuell markierten Stelle im Programm einfügen können. Dazu kann vorausgesetzt werden, daß der Benutzer eine Stelle im Programm markiert, an welcher ein Einfügen eines Sprachkonstruktes auch tatsächlich erlaubt ist.

Aus der Aufgabenstellung wird deutlich, daß das NoName–Informationssystem im wesentlichen aus zwei Teilen besteht. Zum einen ist dies eine Verwaltung der Suchkriterien, zum anderen ist dies die Verwaltung der NoName–Sprachkonstrukte. Die Zusammenhänge können wir uns am ER–Modell aus Abbildung 3.1 verdeutlichen.

Wir sehen die vier Entities

> *Sprachkonstrukt*,
> *SK–Version*,
> *Topic* und
> *NNM–Code*

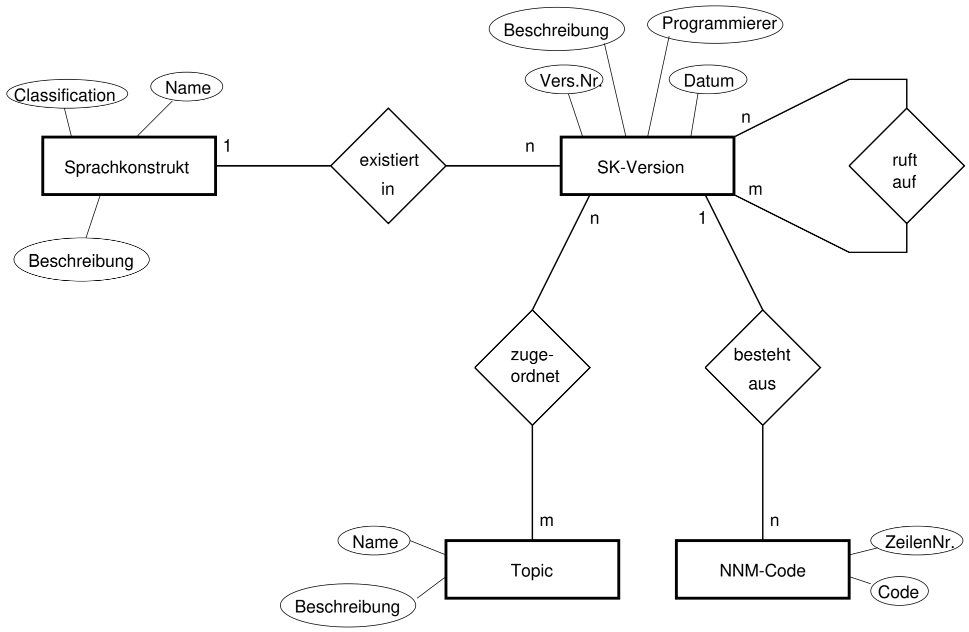


Abb. 3.1: ER–Modell, Version 1: Sprachkonstrukte und dessen Implementierungen sind getrennt modelliert.

sowie die vier Relationen

existiert in,
ruft auf,
zugeordnet und
besteht aus,

welche im folgenden näher erläutert werden.

Ein *Sprachkonstrukt* ist ein abstraktes Objekt, welches einen Namen, eine Beschreibung und eine Nummer (im ER–Diagramm als „Classification" bezeichnet) trägt. Unter einem Sprachkonstrukt wollen wir eine Typdeklaration, eine Spezifikation eines algebraischen Datentyps, eine Funktionsspezifikation und die Spezifikation eines tree transducers verstehen. Die Nummer dient zur Unterscheidung dieser vier „Spezifikationstypen".

Eine *SK–Version* ist eine Implementierung eines speziellen Sprachkonstruktes. Eine SK–Version ist gekennzeichnet mit einer Versionsnummer, einer Beschreibung, dem Namen des Programmierers sowie dem Speicherungsdatum. Da ein Sprachkonstrukt in mehreren Versionen implementiert sein kann, besteht zwischen den Entities „Sprachkonstrukt" und „SK–Version" die 1:n–Beziehung „existiert in".

Der NoName–Code einer SK–Version kann — zumindest theoretisch — beliebig lang sein. Da in der Datenbank aber nur Felder mit einer festen maximalen Größe angelegt werden können, existiert zu dem Entity *SK–Version* kein Attribut, welches diesen NoName–Code enthält. Stattdessen wird für den Code ein eigenes Entity verwendet, wobei der potentiell sehr lange Code in Teile fester Länge aufgespaltet wird. Diese Teile werden in der korrekten Reihenfolge numeriert, so daß der originale Code wieder restauriert werden kann. Dieser Überlegung entsprechend sind die Attribute des Entities *NNM–Code* gewählt. Die nötige Zuordnung von NoName–Code–Zeilen zu einer Implementierung geschieht mit der 1:n–Beziehung „besteht aus" zwischen den Entities „SK–Version" und „NNM–Code".

Das Entity *SK–Version* steht mit sich selbst in einer rekursiven Relation. Dies ist leicht einzusehen, wenn man bedenkt, daß eine SK–Version auf einer anderen basieren kann. So kann eine Funktion beispielsweise eine andere Funktion oder einen Datentyp benötigen, was durch diese rekursive Relation ausgedrückt wird. Die Rekursion selbst kann nicht in SQL, sondern nur in der Host–Sprache modelliert werden (siehe auch Kapitel 2).

Das letzte Entity ist der Suchkriterien–Verwaltung zuzuordnen. Ein *Topic* ist ein Suchkriterium, das lediglich einen Namen und eine Beschreibung besitzt. Ein Topic kann mehreren SK–Versionen zugeordnet sein, aber eine SK–Version kann auch mehreren Topics zugeordnet werden. Daher ist die Relation „zugeordnet" eine n:m–Beziehung.

Nun stellt sich die Frage, ob diese Modellierung bezüglich unserer realen Teilwelt sinnvoll ist. Dazu wollen wir die Verwaltung der Sprachkonstrukte, die Verwaltung der Topics sowie deren Relation zueinander betrachten.

Die Modellierung der Topics wie in Abbildung 3.1 ist sicherlich möglich, da ein Suchkriterium nichts weiter als einen eindeutigen Namen bzw. Schlüssel benötigt. Optional kann es zusätzliche Attribute bekommen, in unserem Fall ist dies eine kurze Beschreibung.

Bei der Modellierung der Sprachkonstruktverwaltung stellt sich die Frage, ob eine Trennung des abstrakten Entities „Sprachkonstrukt" von der realen Implementierung sinnvoll ist oder nicht. Die Trennung der beiden Entities hat einen großen Vorteil. Da der Spezifikationstyp dem Sprachkonstrukt zugeordnet ist, wird quasi automatisch diesbezüglich die richtige Zuordnung der Versionen vorgenommen. Dies bedeutet z.B., daß alle Versionen der Fibonacci–Funktion zu *einem* Spezifikationstyp gehören, nämlich zu dem, welcher dem entsprechenden Sprachkonstrukt — dieses wird hier wohl sinnvollerweise „Fibonacci" heißen — zugeordnet wurde. Desweiteren beschränken sich Bemerkungen bei den SK–Versionen auf die vorliegende Implementierung, weil allgemeine Bemerkungen ebenfalls dem Sprachkonstrukt zugeordnet sind. Jetzt allerdings sieht man auch schon einen Nachteil, denn der Benutzer sieht im syntaxgesteuerten Editor nur

Implementierungen, also SK–Versionen, aber niemals ein Sprachkonstrukt. Soll nun eine SK–Version gespeichert werden, zu der kein Sprachkonstrukt (bezogen auf das ER–Modell) existiert, dann muß dieses zuerst angelegt werden, damit anschließend die richtige Zuordnung der Implementierung erfolgen kann. Dies bedeutet, daß auch für die Sprachkonstrukte selbst eine eigene Verwaltung existieren müßte. Aus diesem Grund werden wir später ein anderes ER–Modell betrachten, das diesbezüglich modifiziert ist (siehe Abbildung 3.2).

Nun wollen wir noch kurz die Relation zwischen den Topics und der Verwaltung der Sprachkonstrukte betrachten. Statt einer Verbindung von Topics und SK–Versionen könnte man auch die Topics mit den Sprachkonstrukten in Beziehung setzen. Der daraus entstehende Vorteil gegenüber der vorgeschlagenen Modellierung bedingt aber gleichzeitig den größten Nachteil: Die Zuordnung der Suchkriterien müßte nur einmal auf Sprachkonstrukt–Ebene erfolgen und nicht separat für jede Implementierung desselben. Allerdings können durchaus zwei Topics „rekursive Implementierung“ und „iterative Implementierung“ existieren, die zwingend zu einer konkreten SK–Version gehören. Somit ist die gewählte Modellierung die sinnvollere Variante.

Wie bereits angedeutet, ist die in Abbildung 3.1 vorgeschlagene Modellierung nicht die einzig mögliche. Der oben erwähnte Nachteil der komplizierten Sprachkonstruktverwaltung kann durch ein reduziertes ER–Modell (siehe Abbildung 3.2) eliminiert werden.

Im Vergleich zu Abbildung 3.1 wurde das Entity „Sprachkonstrukt“ entfernt. Zugleich wurden dem Entity „SK–Version“ die Attribute *Classification* und *Name* hinzugefügt. Die Rolle des Sprachkonstruktes aus unserem vorigen ER–Modell übernimmt jetzt ein Topic. Der Rest des ER–Modells ist unverändert.

Nun wird zwar der Spezifikationstyp (d.h. das Attribut Classification) mit jeder Implementierung gespeichert, aber da die Bestimmung des Spezifikationstyps vom Programm erledigt werden kann, ist dies kein echter Nachteil. Die Eingabe eines Namens bei der Speicherung einer SK–Version stellt einen vertretbaren Aufwand dar, zumal eine geschickte Wahl von Namen das Wiederauffinden der SK–Versionen deutlich erleichtern kann.

Dieses reduzierte ER–Modell dient als Grundlage für die Implementierung des NoName–Modul–IS, da alle erforderlichen Informationen erfaßt und verwaltet werden können.

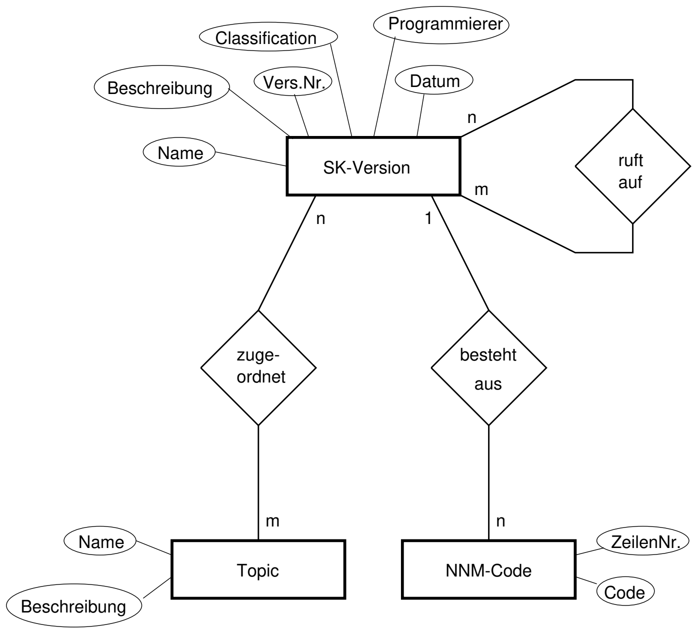


Abb. 3.2: ER–Modell, Version 2: Sprachkonstrukte fallen mit den Topics zusammen.

3.2 Die Topic–Verwaltung

3.2.1 Allgemeines

Unter einem Topic versteht man ein „Kennzeichen“ oder „Thema“, das Sprachkonstrukten[5] zugeordnet werden kann. Die Topics dienen dazu, ein Sprachkonstrukt näher zu spezifizieren und in einen gewissen Kontext einzuordnen. So kann ein Topic z.B. ein bestimmtes Projekt bezeichnen oder eine Gruppe von Funktionen, etwa die üblichen Listenfunktionen.

Topics können das Auffinden eines gesuchten Sprachkonstruktes erleichtern. Es ist nicht unbedingt nötig, den Namen des Sprachkonstruktes zu kennen, es läßt sich auch aufgrund der zugeordneten Topics relativ schnell finden. Durch die Topics grenzt sich die Menge der möglichen Sprachkonstrukte ein, so daß ohne große Schwierigkeiten das passende Konstrukt gewählt werden kann. Dies setzt natürlich voraus, daß die Topics möglichst vielsagende Namen erhalten und auch konsequent verwendet werden.

[5]Sprachkonstrukte werden in Abschnitt 3.3 behandelt.

Die Topic–Verwaltung gliedert sich in drei Aufgaben:

1. Anlegen eines neuen Topics (Save).
2. Löschen eines Topics (Delete).
3. Anzeigen von Topics (Browse).

Im Anschluß an diesen Abschnitt werden diese Teilaufgaben detailliert behandelt, im Kapitel 9 wird anhand eines Beispiels gezeigt, wie sich die Erledigung dieser Aufgaben für den Benutzer des Programms gestaltet.

3.2.2 Anlegen eines neuen Topics

Ein neues Topic muß einen Namen (maximal 20 Zeichen) erhalten, der noch an kein anderes Topic vergeben wurde. Optional hierzu kann eine Beschreibung des Topics (maximal 50 Zeichen) eingegeben werden. Dies erscheint besonders dann sinnvoll, wenn z.B. ein Projekt als Topic fungieren soll, der (evtl. mit Abkürzungen versehene) Projektname aber nicht gut auf den Projektinhalt schließen läßt. Dann kann der Topicname trotzdem dem Projektnamen entsprechen, in der Beschreibung zu dem Topic kann dann eine kurze Erläuterung dazu stehen. Somit kann auch ein „neuer“ Benutzer des Systems relativ leicht die Bedeutung von bereits existierenden Topics erkennen und diese Topics auch verwenden.

3.2.3 Löschen eines Topics

Bevor ein Topic gelöscht werden kann, muß festgestellt werden, ob dieses Topic einem Sprachkonstrukt zugeordnet ist. Wenn dies der Fall ist, dann darf das Topic nicht gelöscht werden. Es könnte sonst unter Umständen ein Sprachkonstrukt existieren, das nur diesem einen Topic zugeordnet war — und nun könnte es nicht mehr über die Angabe von Topics gefunden werden. Wenn kein Benutzer den Namen dieses Sprachkonstruktes kennt, dann ist ein Auffinden kaum mehr möglich.

Wenn das Löschen des Topics erlaubt ist, dann werden der Name und die Beschreibung aus der Topic–Relation entfernt.

3.2.4 Browsen von Topics

Beim Browsen werden die Namen aller gespeicherten Topics in alphabetisch sortierter Reihenfolge auf dem Bildschirm angezeigt. Wenn ein Topicname mit der Maus angeklickt wird, dann wird der zugehörige Beschreibungstext angezeigt.

3.3 Die Verwaltung der Sprachkonstrukte

3.3.1 Allgemeines

Unter einem Sprachkonstrukt der Programmiersprache NoName verstehen wir die Spezifikation von folgenden Konstrukten:

- Typen
- Algebraische Datentypen
- Funktionen
- tree transducer

Jedes Sprachkonstrukt gehört somit zu genau einer dieser vier „Klassen“, den sogenannten Spezifikationstypen. Wie bereits im Abschnitt 3.2 angedeutet, können einem Sprachkonstrukt mehrere Topics zugeordnet werden.

Die Verwaltung der Sprachkonstrukte gliedert sich in die folgenden drei Aufgaben:

1. Speichern eines Sprachkonstruktes in der Datenbank (Save).
2. Löschen eines Sprachkonstruktes aus der Datenbank (Delete).
3. Übertragen eines Sprachkonstruktes aus der Datenbank in den NoName–Editor (Retrieve).

Für jede dieser Teilaufgaben existiert ein Menüpunkt, dessen Aktivierungsmöglichkeit jeweils von bestimmten Bedingungen abhängt. Das Browsen von Sprachkonstrukten ist nicht als eigenständige Aufgabe realisiert, sondern in das Retrieve mit eingebunden.

In den folgenden Abschnitten sind diese drei Aufgaben näher erläutert. Im Kapitel 9 geben wir Beispiele für die Verwaltung von Sprachkonstrukten an.

3.3.2 Speichern eines Sprachkonstruktes in der Datenbank

Um den Menüpunkt für die Speicherung eines neuen Sprachkonstruktes aktivieren zu können, muß im NoName–Editor ein Sprachkonstrukt markiert sein, das sich korrekt parsen läßt und keine Fehler bezüglich der statischen Semantik besitzt[6]. Dies hat zur Folge, daß alle gespeicherten Sprachkonstrukte ähnlich

[6]Die statische Semantik ist in [EGH+94a] beschrieben.

einer Black Box zu verwenden sind. Der Benutzer kann sich also darauf verlassen, daß ein Konstrukt aus der Datenbank syntaktisch korrekt ist. Dennoch können bei der späteren Verwendung von gespeicherten Sprachkonstrukten Fehlermeldungen des NoName–Editors nicht ausgeschlossen werden. Es können z.B. Namenskonflikte auftreten, die zur Zeit der Speicherung noch nicht vorhanden waren.

Beim Speichern des Sprachkonstruktes muß zuerst der zugehörige Spezifikationstyp ermittelt werden. Dies geschieht durch das Programm. Das Speicherungsdatum sowie den Login–Namen des Benutzers ermittelt das Programm ebenfalls automatisch. Der Benutzer muß lediglich den Namen (maximal 30 Zeichen) und die Version des Sprachkonstruktes (maximal 8 Zeichen) über die Tastatur eingeben. Optional kann auch ein Kommentar aus maximal 50 Zeichen eingegeben werden.

Anschließend wählt der Benutzer aus einer Liste mittels Mausklick die Topics aus, die dem Sprachkonstrukt zugeordnet werden sollen. Dabei können beliebig viele Topics gewählt werden.

Die Bestätigung des Speichervorganges (mittels Mausklick) stößt die Übernahme des NoName–Codes in die Datenbank an, wobei folgende Schritte ausgeführt werden:

1. Falls das zu speicherne Sprachkonstrukt von anderen Sprachkonstrukten abhängt, dann werden auch diese Konstrukte in die Datenbank übernommen. Soll z.B. eine Funktion f gespeichert werden, die auf dem Datentyp $tree$ definiert ist, dann wird sowohl f als auch $tree$ in der Datenbank abgelegt. Dabei werden den „Hilfskonstrukten“, im Beispiel $tree$, dieselben Topics zugeordnet wie dem ursprünglich zu speichernden Sprachkonstrukt. Die Namensgebung für diese Hilfskonstrukte wird vom Programm erledigt, indem vor den eingegebenen Namen die Zeichenreihe „Hx“ gesetzt wird, wobei x durch eine fortlaufende Numerierung ersetzt wird („H“ steht für „Hilfskonstrukt“). Sollten Hilfskonstrukte selbst von anderen Sprachkonstrukten abhängen, dann wiederholt sich dieser Vorgang, bis die transitive Hülle der Abhängigkeiten gefunden ist. Dabei wird redundante Speicherung so weit wie möglich vermieden.

2. Der NoName–Code eines zu speichernden Sprachkonstruktes wird bei Bedarf automatisch auf Zeilen mit einer Länge von jeweils 1500 Zeichen aufgeteilt. Damit besteht prinzipiell keine Obergrenze für die Codelänge eines Sprachkonstruktes. Die Zeilen werden fortlaufend numeriert, um bei einem Retrieve (siehe Abschnitt 3.3.4) die korrekte Reihenfolge — und damit den originalen NoName–Code — rekonstruieren zu können.

Nach dem Speichern des Sprachkonstruktes kann der Benutzer sofort im Editor weiterarbeiten.

3.3.3 Löschen eines Sprachkonstruktes aus der Datenbank

Ein Benutzer kann nur Sprachkonstrukte löschen, die er auch selbst in der Datenbank abgelegt hat. Damit wird ausgeschlossen, daß ein Benutzer absichtlich oder unabsichtlich die Arbeit anderer Benutzer zerstört. Im folgenden verwenden wir den Begriff „Sprachkonstrukt" immer unter der Annahme, daß der Benutzer dieses Konstrukt in die Datenbank abgelegt hat und es daher hieraus auch entfernen darf.

Ein gespeichertes Sprachkonstrukt kann jederzeit aus der Datenbank gelöscht werden. Dazu muß der Benutzer zuerst den Spezifikationstyp des zu löschenden Sprachkonstruktes aus den vier gegebenen Möglichkeiten auswählen. Für die nun folgende Auswahl des Sprachkonstruktes existieren zwei mögliche Vorgehensweisen. Wenn der Benutzer den Namen und die Version des Sprachkonstruktes kennt, so kann er diese direkt in die vorgesehenen Dialogelemente eintragen. Andernfalls wählt der Benutzer aus einer Liste mit den vorhandenen Topics diejenigen aus, die vermutlich dem Sprachkonstrukt zugeordnet sind. Diese Topics dienen bei der darauffolgenden Suche in der Datenbank als Suchkriterium. Ein Konstrukt „paßt" auf das Suchkriterium, wenn ihm mindestens eines der gewählten Topics zugeordnet ist. Der Name und die Version dieser Konstrukte werden dem Benutzer in einer Liste angezeigt. Daraus wird durch Anklicken mit der Maus ein Sprachkonstrukt gewählt. Bevor tatsächlich gelöscht wird, kann sich der Benutzer den NoName–Code des gewählten Sprachkonstruktes noch einmal ansehen. Bestätigt er anschließend den Löschvorgang, so werden alle gespeicherten Informationen zu diesem Sprachkonstrukt aus der Datenbank entfernt.

Wurde ein Sprachkonstrukt gelöscht, so existieren möglicherweise andere Konstrukte in der Datenbank, die ausschließlich für dieses Sprachkonstrukt benötigt wurden. In den meisten Fällen können auch solche Konstrukte gelöscht werden. Da die Entscheidung hierüber aber letztendlich beim Benutzer liegen sollte, werden ihm die betroffenen Sprachkonstrukte automatisch nacheinander in einer Dialogbox angezeigt, in welcher er das Löschen bestätigen bzw. verhindern kann.

Nach dem Löschen eines bzw. mehrerer Sprachkonstrukte kann der Benutzer sofort im Editor weiterarbeiten.

3.3.4 Suchen eines Sprachkonstruktes und Übernahme des Codes in das NoName–Programm

Der gespeicherte NoName–Code aus der Datenbank ersetzt genau den Platzhalter im NoName–Editor, der aktuell markiert ist. Da aus der Datenbank nur komplette Spezifikationen importiert werden können, muß die aktuelle Selektion im Editor eine leere Spezifikation sein. Andernfalls kann ein Retrieve nicht ausgeführt werden.

Beim Retrieve wird ein Sprachkonstrukt auf die gleiche Weise ausgewählt wie beim Löschen eines Sprachkonstruktes: Zuerst muß der Spezifikationstyp gewählt, anschließend der Name und die Version direkt eingegeben oder über die Topics gewählt werden (siehe auch Abschnitt 3.3.3). In beiden Fällen hat der Benutzer die Möglichkeit, vor Ausführung des Retrieves den NoName–Code des gewählten Sprachkonstruktes anzusehen.

Wird der Retrieve durchgeführt, dann wird der NoName–Code aus der Datenbank importiert. Falls mehrere Code–Zeilen existieren, dann werden diese in der richtigen Reihenfolge zusammengesetzt. Anschließend wird der NoName–Code an den Parser übergeben. Der aus dem Parsing resultierende Syntaxbaum wird in den Syntaxbaum des momentan editierten NoName–Programmes an der markierten Stelle eingefügt. Sind in der Datenbank Sprachkonstrukte enthalten, welche von dem gesuchten Konstrukt benötigt werden, so werden diese ebenfalls importiert. Dabei bleiben die Sprachkonstrukte unberücksichtigt, welche bereits bei einem früheren Retrieve in derselben Arbeitssitzung importiert wurden.

Im Anschluß an einen Retrieve kann der Benutzer auf die gewohnte Weise weiterarbeiten.

3.4 Inhalt der Datenbank ausgeben

Mit dem NoName–Modul–Informationssystem können nahezu beliebig viele Sprachkonstrukte der Sprache NoName verwaltet werden[7]. Dabei ist es denkbar, daß die Übersicht über den aktuellen Datenbankinhalt verloren geht.

Um auch in dieser Hinsicht den Nutzen des Informationssystems nicht einzuschränken, existiert eine Möglichkeit, alle in der Datenbank verfügbaren Daten in Form von LaTeX–Dateien auszugeben. Diese Listen enthalten entweder alle Sprachkonstrukte oder alle Topics oder alle Zuordnungen zwischen Sprachkonstrukten und Topics.

[7] Die Obergrenze ergibt sich aus der Kapazität des verwendeten Speichermediums.

4 Realisierung des ER–Modells

4.1 Systematischer Entwurf von Relationen

ER–Modelle visualisieren Abhängigkeiten und Eigenschaften der Objekte aus einer realen Miniwelt. Dieser konzeptuelle Entwurf muß nun auf ein konzeptuelles Datenmodell abgebildet werden.

Der Entwurf von Relationen im relationalen Datenbankmodell aufgrund eines ER–Modells läuft nach folgendem Pseudo–Algorithmus ab (wir gehen davon aus, daß die Entities des ER–Modells ausschließlich atomare, d.h. strukturlose Attribute besitzen und den Beziehungen keine Attribute zugeordnet werden):

Transformation ER–Modell → relationales Datenmodell

Sei n_E die Anzahl der Entities im ER–Modell. Für alle i mit $1 \leq i \leq n_E$ sei m_i die Anzahl der Attribute des Entities E_i. Ferner bezeichne für alle j mit $1 \leq j \leq m_i$ A_{ij} ein Attribut des Entities E_i. Für eine n:m–Beziehung (bzw. eine 1:n–Beziehung[8]) zwischen den Entities E_i und E_j schreiben wir $B_{ij}^{n:m}$ (bzw. $B_{ij}^{1:n}$).

1. Jedes Entity E_i wird auf eine Relation $R_i, 1 \leq i \leq n_E$ abgebildet, wobei gilt:
$$sch(R_i) = \{A_{i1}, A_{i2}, \ldots, A_{im_i}\},\ 1 \leq i \leq n_E$$
   Der Primärschlüssel der Relation R_i setzt sich aus den Schlüsselattributen des Entities E_i zusammen. Die Wertebereiche der Attribute ändern sich durch diese Transformation nicht.

2. Sei $B_{ij}^{n:m}$ eine n:m–Beziehung zwischen den beiden Entities E_i und E_j $(1 \leq i,\ j \leq n_E)$ und seien $\{A_{i1}, \ldots, A_{ik}\}$ bzw. $\{A_{j1}, \ldots, A_{jl}\}$ die Primärschlüssel der Relationen R_i bzw. R_j. Dann wird $B_{ij}^{n:m}$ auf die Relation R_{ij} abgebildet, wobei gilt:
$$sch(R_{ij}) = \{A_{i1}, \ldots, A_{ik}, A_{j1}, \ldots, A_{jl}\}$$
   Die Wertebereiche der Attribute von R_{ij} werden von den Relationen R_i und R_j übernommen.

[8]Wir gehen im folgenden davon aus, daß in n:m–Beziehungen echte Mehrfachbeziehungen existieren, d.h. es gilt $n > 1$ und $m > 1$.

3. Sei $B_{ij}^{1:n}$ eine 1:n–Beziehung zwischen den beiden Entities E_i und E_j $(1 \leq i,\ j \leq n_E)$ und sei $\{A_{i1}, \ldots, A_{ik}\}$ der Primärschlüssel der Relation R_i. Ferner sei $sch(R_j) = \{A_{j1}, \ldots, A_{jm_j}\}$ das Schema von R_j. Dann wird $B_{ij}^{1:n}$ direkt in die Relation R_j aufgenommen durch folgende Modifikation:

$$sch(R_{j'}) = \{A_{j1}, \ldots, A_{jm_j}, A_{i1}, \ldots, A_{ik}\}$$

und $m_{j'} = m_j + k$. Die Wertebereiche der hinzugenommenen Attribute sind identisch mit den Wertebereichen derselben Attribute aus R_i.

Diese schematische Vorgehensweise hat einen Schwachpunkt. Wenn Schlüsselattribute sehr umfangreich sind — dies gilt z.B. für Text–Attribute —, dann sind in der Regel auch Relationen zur Realisierung von n:m–Beziehungen sehr groß. Der Bedarf an Speicherplatz kann in solchen Fällen meist verringert werden, wenn ein künstlicher, in der Regel numerischer Schlüssel eingeführt wird. Diese Optimierung wurde auch bei der vorliegenden Realisierung angewendet.

Wenn die erwartete Anzahl Tupel in einer Relation sehr groß sein wird, dann besteht die Möglichkeit, diese Relation nach bestimmten Kriterien horizontal zu partitionieren. Wenn z.B. in einer Relation alle Namen, Adressen (Ort, Straße und Hausnummer) und Telefonnummern der Einwohner von deutschen Millionenstädten gespeichert werden, dann könnte man diese Relation in mehrere kleinere Relationen zerlegen, die jeweils nur eine einzige deutsche Millionenstadt abdecken. Dabei bräuchte man den Namen des jeweiligen Wohnortes nicht in die Datensätze aufzunehmen, da die Städtenamen innerhalb einer Relation identisch wären. Der Vorteil dieser Partitionierung liegt in den schnelleren Zugriffszeiten auf einzelne Datensätze. Allerdings muß dann die Datenbankanfrage entsprechend modifiziert werden, so daß jeweils die gewünschte Partition verwendet wird. In der vorliegenden Arbeit wurde auch dieser Optimierungsansatz berücksichtigt und angewendet.

4.2 Entwurf der Ingres–Relationen

Mit dem in Abschnitt 4.1 gezeigten Verfahren ergeben sich aus dem ER–Modell in Abbildung 3.2 insgesamt fünf Relationen. Einige Attribute erhielten andere Namen, numerische Schlüssel wurden zwecks Optimierung mit aufgenommen.

In den folgenden Abschnitten sind die Relationsdefinitionen in Tabellenform dargestellt, die von dem Programm NoMIS verwendet werden. Die Schlüsselattribute der Relationen sind unterstrichen.

4.2.1 Die Relation skversion

Die Relation skversion entspricht dem Entity SK–Version.

skversion		
Feldname	Datentyp	Bedingung
classification	i2	not null
skname	varchar(30)	not null
version	varchar(8)	not null
skcode	i4	unique, not null
programmer	varchar(20)	–
storedat	date	–
comment	varchar(50)	–

Das Schema, das sich aufgrund des Entities SK–Version ergibt, wurde um das Attribut *skcode* erweitert. Dieses Attribut wird von anderen Relationen als Fremdschlüssel verwendet und wird hier deshalb mit der Bedingung „unique, not null“ deklariert.

4.2.2 Die Relation topic

Die Relation topic entspricht dem Entity Topic.

topic		
Feldname	Datentyp	Bedingung
name	varchar(30)	not null
topicno	i4	unique, not null
comment	varchar(50)	–

Hier wurde zusätzlich das Attribut *topicno* eingeführt, das als Fremdschlüssel verwendet werden kann. Daher ist es mit der Bedingung „unique, not null“ deklariert.

4.2.3 Die Relation sk_nnmcode

Die Relation sk_nnmcode steht für das Entity NNM_Code und die 1:n–Beziehung „besteht aus“. In der Relation wird nicht der Syntaxbaum, sondern dessen textuelle Repräsentation gespeichert.

sk_nnmcode		
Feldname	Datentyp	Bedingung
skcode	i4	not null
stringno	i4	not null
string	varchar(1500)	–

Das Attribut *skcode* dient hier als Fremdschlüssel und realisiert die 1:n–Beziehung zwischen den Relationen sk_version und sk_nnmcode.

4.2.4 Die Relationen sktopic1, sktopic2, sktopic3, sktopic4

Die n:m–Beziehung für die Zuordnung von Topics zu Sprachkonstrukten wurde in vier Relationen aufgeteilt. Die Partitionierung erfolgte dabei in Abhängigkeit des Spezifikationstyps der Sprachkonstrukte. Die Relation sktopic$x$ enthält somit nur Zuordnungen von Topics zu Sprachkonstrukten des Spezifikationstyps x mit $x \in \{1, 2, 3, 4\}$.

Die Aufteilung nach diesem Kriterium wurde gewählt, um Anfragen an die Datenbank beschleunigt ausführen zu können. Der Test, welche Partition zu verwenden ist, läßt sich in der Host–Sprache sehr einfach durchführen und erspart Anfragen an eine einzige, deutlich größere Relation.

sktopic$x$		
Feldname	Datentyp	Bedingung
skcode	i4	not null
topicno	i4	not null

Die Attribute *skcode* bzw. *topicno* sind Fremdschlüssel der Relationen skversion bzw. topic.

4.2.5 Die Relation dependencies

Die Abhängigkeiten von Sprachkonstrukten untereinander werden in der Relation dependencies gespeichert.

dependencies		
Feldname	Datentyp	Bedingung
dependent	i4	not null
required	i4	not null

Beide Attribute sind Fremdschlüssel der Relation skversion. Ein Tupel aus dieser Relation speichert die Abhängigkeit des „dependent"–Sprachkonstruktes vom „required"–Sprachkonstrukt.

4.2.6 Die Relation toptemp

Die Relation `toptemp` gehört nicht zu den Relationen, die durch das in Abschnitt 4.1 beschriebene Verfahren generiert werden. Sie wird in der Implementierung nur verwendet, um effizient arbeiten zu können. Die genaue Begründung hierfür kann im Abschnitt 5.8 nachgelesen werden.

toptemp		
Feldname	Datentyp	Bedingung
topicno	i4	-

Wie der Name dieser Relation bereits andeutet, werden hier nur vorübergehend Topics zwischengespeichert.

5 Lösungsansätze und deren Implementierung

In den folgenden Abschnitten werden die Lösungsansätze beschrieben, die letztendlich zur Implementierung führten. Die dabei angegebenen Algorithmen können aufgrund der Systemumgebung nicht immer direkt implementiert werden und geben somit nur die grundlegenden Ideen wieder. In der Implementierung wurden die Algorithmen an die Bedürfnisse der Fenster–Programmierung angepaßt, d.h. hinter einem einzigen „Befehl“ des Algorithmus können sich unter Umständen mehrere Funktionen verbergen. An den entsprechenden Stellen wird jeweils auf die Unterschiede hinsichtlich der Implementierung hingewiesen.

Die Algorithmen sind in einer prozedural orientierten Notation angegeben, wobei Kontrollstrukturen ähnlich denen in Pascal oder C verwendet werden. Für die Zuweisung des Wertes eines Ausdrucks b an die Variable a schreiben wir in den Algorithmen $a \leftarrow b$.

Um eine einheitliche Verwendung der Bezeichnungen zu ermöglichen, legen wir folgende Namenskonvention fest: Funktionen, welche die Aktivierbarkeit eines Menüpunktes testen, werden mit *A–Testfunktion* bezeichnet. Funktionen mit den Aktionen eines Menüpunktes heißen *Kommando–Funktionen*.

Wir werden uns zunächst mit der Modularisierung der Aufgabenstellung befassen und dabei die verschiedenen Module kurz vorstellen. Anschließend werden wir die Probleme und Lösungen ansprechen, die aufgrund der Systemumgebung entstanden. Mit den Realisierungen der einzelnen Teilaufgaben werden wir uns dann in den letzten Abschnitten dieses Kapitels befassen.

5.1 Die Modularisierung der Aufgabenstellung

In den vorangegangenen Kapiteln wurden die Aufgaben des NoName–Modul–Informationssystems bereits beschrieben. An dieser Stelle werden wir nun konkrete Teilaufgaben angeben, die nahezu unabhängig voneinander implementiert werden können. Da die Bedienung des Modul–Informationssystems über ein Menü vorgenommen werden soll, ist es naheliegend, die Aufgabenstellung so zu modularisieren, daß jeder Teilaufgabe ein Menüpunkt zugeordnet ist. Diese Vorgehensweise hat sich während der Implementierungsphase bewährt, da bereits programmierte Teilaufgaben unabhängig von den noch fehlenden Teilen getestet werden konnten.

Die Aufgabenstellung wird in die folgenden Teilaufgaben untergliedert:

- Verbindung zwischen Datenbank und Editor herstellen.
- Verbindung zwischen Datenbank und Editor abbrechen.
- Verwaltung der Topics:
  - Anzeige aller gespeicherten Topics.
  - Speichern eines neuen Topics.
  - Löschen eines nicht benötigten Topics.
- Speichern eines neuen Sprachkonstruktes.
- Importieren eines gespeicherten Sprachkonstruktes.
- Löschen eines gespeicherten Sprachkonstruktes.
- Erzeugen der LaTeX–Ausgabedateien:
  - Übersicht über alle gespeicherten Topics.
  - Übersicht über alle gespeicherten Sprachkonstrukte.
  - Übersicht über die Zuordnungen zwischen Topics und Sprachkonstrukten.

Wie in dieser Aufstellung zu sehen ist, werden die Verwaltung der Topics und die Erzeugung der Ausgabedateien noch weiter untergliedert. Diese kleinen Teilaufgaben erhalten allerdings keinen eigenen Menüpunkt, sondern werden über ein Dialogfenster ausgewählt, welches durch die Wahl des übergeordneten Menüpunktes geöffnet wird.

Aus der oben gezeigten Auflistung der Teilaufgaben ergibt sich fast direkt das Benutzermenü des Modul–Informationssystems (siehe Abbildung 5.1). Lediglich die Reihenfolge der Punkte wurde den erwarteten Benutzungshäufigkeiten entsprechend verändert. Häufig benötigte Menüpunkte sind eher im oberen Teil des Menüs angeordnet, die seltener verwendeten Punkte stehen dementsprechend weiter unten. Der Menüpunkt `db-exit-all` wurde noch hinzugefügt, um das Beenden des Editors nach dem Abbruch der Verbindung zur Datenbank etwas zu vereinfachen.

Die Modularisierung der Quelldateien orientiert sich an den Teilaufgaben (und somit auch an den Menüpunkten dieses Menüs). Lediglich die Realisierung der letzten drei Menüpunkte befindet sich — wegen der engen semantischen Zusammengehörigkeit — in einem einzigen Modul. Darüber hinaus existieren noch

NoMIS
db-retrieve db-store db-delete
topic-handling
db-output
db-connect db-disconnect db-exit-all

Abb. 5.1: Das Menü zur Bedienung von NoMIS.

Module, die allgemein benötigte Funktionen enthalten und daher keinem konkreten Menüpunkt zugeordnet werden können.

Die folgende Tabelle enthält eine Übersicht über die existierenden Module. Angegeben ist auch eine kurze Erläuterung, wofür das jeweilige Modul verwendet wird.

Modul	*Verwendung*
`nomis_widgets.sc`	Deklaration der Dialogfenster und Menüpunkte
`nomis_process.sc`	einfache Prozeß–Steuerung
`nomis_store.sc`	Speicherung eines Sprachkonstruktes
`nomis_retrieve.sc`	Übernahme eines Sprachkonstruktes aus der Datenbank
`nomis_delete.sc`	Löschen eines Sprachkonstruktes aus der Datenbank
`nomis_topic.sc`	Verwaltung der Topics
`nomis_output.sc`	Erstellung von Übersichtslisten in Form von LaTeX–Dateien
`nomis_dbconnect.sc`	Aufbau und Abbruch der Datenbank–Verbindung sowie Behandlung von Datenbankfehlern

Bei der Beschreibung der Realisierungen der Menüpunkte in den einzelnen Unterkapiteln gehen wir in der Reihenfolge vor, in welcher auch ein Benutzer die einzelnen Menüpunkte wählen wird. D.h. zunächst wird der Aufbau und Abbruch einer Verbindung zur Datenbank erläutert. Anschließend betrachten wir die Realisierung der Topicverwaltung und der Menüpunkte zur Verwaltung der Sprachkonstrukte. Zuletzt wird beschrieben, wie die Erstellung der LaTeX–Übersichtslisten realisiert ist.

Zuvor jedoch werden wir auf die allgemeinen Aspekte, darunter auch die Zusammenführung von SGen und Ingres, näher eingehen und die Funktionen der Toolbox zur einfacheren Programmierung kennenlernen.

5.2 Zusammenführung von SGen und Ingres

Zu Beginn der Arbeit stellte sich das Problem, wie der NoName–Editor und eine relationale Datenbank überhaupt zusammenzufügen sind. Die Grundlage hierfür kann nur die Programmiersprache C sein, da der NoName–Editor eine Schnittstelle zu C anbietet und die Datenbank mittels Embedded SQL (ESQL) und C als Host–Sprache angesprochen werden kann.

In allen vorangegangenen Tests und Arbeiten wurde der Code für benötigte C–Funktionen — durch SSL–spezifische Symbole markiert — direkt in den SSL–Quelldateien abgelegt. Erste Versuche mit Embedded SQL zeigten jedoch, daß der SSL–Präprozessor die Embedded–SQL–Anweisungen nicht bearbeiten konnte und abstürzte. Der Versuch, zuerst den ESQL–Präprozessor und dann die SSL–Übersetzung zu starten, mißlang ebenfalls. Somit blieb nur noch eine Möglichkeit offen: Der datenbankrelevante C–Code wird in separate Dateien gestellt, die dem ESQL–Präprozessor als Eingabedateien dienen. Anschließend können die daraus resultierenden Dateien mit dem C–Compiler in Object–Dateien übersetzt werden. Diese wiederum müssen nun lediglich beim Aufruf des SSL–Übersetzers als Parameter mit angegeben werden, der Linker fügt sie dann in den NoName–Editor ein. Ein angenehmer Nebeneffekt dieser Vorgehensweise ist, daß bei einer Änderung des datenbankrelevanten C–Codes die SSL–Quelldateien nicht mehr übersetzt werden müssen. Dadurch reduziert sich die Übersetzungszeit erheblich.

Die Namen aller Module mit datenbankrelevantem C–Code benötigen die Endung `.sc`, damit der Embedded–SQL–Präprozessor für Ingres die Dateien korrekt erkennt und übersetzt. Zu jedem Implementations–Modul existiert eine Deklarationsdatei (mit gleichem Namen, Endung `.sh`), in welcher die Prototypen der Funktionen enthalten sind.

Nachdem der prinizipielle Zusammenschluß von dem NoName–Editor und der Datenbank erfolgt ist, muß noch der Datenaustausch zwischen beiden Teilen ermöglicht werden. Wie bereits in Kapitel 4 beschrieben, werden die NoName–Sprachkonstrukte in ihrer textuellen Repräsentation in der Datenbank abgelegt. Dies setzt voraus, daß eine Möglichkeit existiert, entweder in der Programmiersprache C oder in SSL auf diese textuelle Repräsentation zugreifen zu können. Daß eine derartige Möglichkeit besteht, ist klar: Auch auf dem Bildschirm wird dem Benutzer des NoName–Editors die textuelle Repräsentation und nicht etwa der Syntaxbaum angezeigt. Um die hierfür nötige prinzipielle Vorgehensweise ausfindig zu machen, war eine aufwendige Suche im Quellcode der Firma GrammaTech unumgänglich[9]. Aufgrund der systematischen Analyse einiger Quell-

[9] C–Funktionen von GrammaTech, die nicht im bereitgestellten C–Interface enthalten sind, werden im folgenden *interne Funktionen* genannt. Ensprechend wird der Begriff *interne Datentypen* verwendet.

dateien entstand unter anderem die C–Funktion `get_text_of_selection()`[10]. Diese liefert die textuelle Repräsentation des im Editor gerade markierten Syntaxbaumes. Umgekehrt kann mit der Funktion `put_string_into_tree(...)` eine Zeichenfolge in den zugehörigen Syntaxbaum umgewandelt und in den Syntaxbaum des Programms an der aktuellen Markierung eingefügt werden.

5.3 Programmierung in der Fensteroberfläche

In Abschnitt 2.4.2 haben wir bereits die Funktion `init_sys3()` aus dem C–Interface des Synthesizer Generators angesprochen. Diese Funktion kann zur Deklaration eigener Menüs und Dialogfenster genau einmal vom Programmierer überschrieben, d.h. ersetzt werden. Da parallel zu meiner Arbeit auch Dietmar Ernst das C–Interface des SGen verwendete, entstand ein Konflikt durch mehrfaches Überschreiben dieser Initialisierungsfunktion. Wir einigten uns daher auf folgendes Vorgehen:

1. Die Fenster– und Menü–Deklarationen der Datenbankanbindung werden aus der Funktion `init_sys3()` ausgelagert und stattdessen in die Funktion `init_emdb()` geschrieben. Desweiteren existiert in dem Implementationsmodul `nomis_widgets.sc`, welches die Deklaration der Menüpunkte und Fenster enthält, die Funktion `init_sys3()`, die nur die Funktion `init_emdb()` aufruft und von der Compiler–Direktive `__MY_EDITOR__` umschlossen ist.

2. Soll bei der Erzeugung eines Editors nur die Datenbankanbindung eingebunden werden, so muß bei der Übersetzung die erwähnte Compiler–Direktive `__MY_EDITOR__` angegeben werden. Dadurch wird auch die Funktion `init_sys3()` aus dem Modul `nomis_widgets.sc` in den Code mit eingebunden.

3. Sollen sowohl die Datenbankanbindung als auch die Erweiterungen von Dietmar Ernst in den Editor eingebunden werden, so ist bei der Übersetzung die Compiler–Direktive `__MY_EDITOR__` nicht anzugeben. Somit wird die Funktion `init_sys3()` aus dem Modul `nomis_widgets.sc` nicht in den Code eingebunden. Die Funktion `init_emdb()` wird in diesem Fall durch die Funktion `init_sys3()` von Dietmar Ernst aufgerufen.

Die zentrale Funktion im Modul `nomis_widgets.sc` heißt somit `init_emdb()`. In ihr werden alle Menüpunkte des Menüs `NoMIS` deklariert und die Dialogfenster hierfür registriert.

[10] Die Funktionen, die auf internen Funktionen und Datentypen beruhen, sind in einer Toolbox (bestehend aus der C–Quelldatei `sgentools.c` sowie der Headerdatei `sgentools.h`) zusammengefaßt.

Die Namen der Kommando– und A–Testfunktionen leiten sich aus dem Namen des jeweils zugeordneten Menüpunktes ab. Heißt der Menüpunkt beispielsweise `db-connect`, dann heißen die Kommando– bzw. A–Testfunktionen entsprechend `db_connect_command` bzw. `db_connect_valid`. Diese Namenskonvention wird nur dann nicht beachtet, wenn dadurch Funktionen mit unterschiedlichem Namen, aber exakt derselben Funktionalität entstehen würden.

Die Registrierung der Dialog–Fenster hält sich mit einer geringen Abweichung an das in Abschnitt 2.4.3 beschriebene Schema. Die Abweichung bezieht sich auf die Version 4.1 des SGen. In dieser Version war noch keine sogenannte „finish_procedure" (Parameter P_6) vorgesehen. Da die Entwicklung des Modul–Informationssystems noch mit der SGen–Version 4.1 begann und die Version 4.2 während der Programmierphase noch nicht stabil lief, existiert die Compiler–Direktive `__VERSION4_2__`. Wird diese Direktive bei der Übersetzung nicht angegeben, so wird ein Editor für die Version 4.1 erzeugt.

Die in den folgenden Abschnitten angegebenen Algorithmen sind zumeist in den Kommando–Funktionen der entsprechenden Menüpunkte enthalten. Um die Fenster–Elemente mit Werten zu belegen, müssen bestimmte Datenstrukturen verwendet werden (diese können in [Gra95] nachgelesen werden). Da in einigen Dialogen eine oder mehrere Listen enthalten sind, wollen wir die dazu benötigte Datenstruktur kurz beschreiben. Die Einträge in einer Liste müssen Character–Felder sein, d.h. es ist nicht möglich, Strukturen in eine Liste aufzunehmen. Die Listenstruktur wird durch ein Feld nachgebildet, dessen Komponenten Zeiger auf jeweils ein Listenelement enthalten. Das letzte Feldelement muß einen NIL–Zeiger enthalten, der das Ende des Feldes markiert. Um diese „Liste" in einem Fenster anzuzeigen, muß dem entsprechenden Listen–Fensterelement lediglich der Zeiger auf das Feld übergeben werden. Abbildung 5.2 zeigt exemplarisch die Struktur einer derartigen Liste. Das erste Feldelement enthält einen Zeiger auf die Zeichenkette „Sinus", das zweite Element enthält einen Zeiger auf die Zeichenkette „Ackermann" etc.

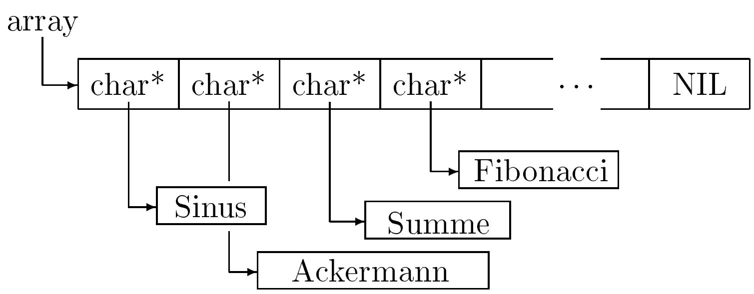


Abb. 5.2: Zeigerstruktur für Listenobjekte in Fenstern

Einen mächtigen Mechanismus stellt die Möglichkeit dar, einen Hauptspeicherbereich mit einem Dialogfenster zu verknüpfen. Dies ist besonders nützlich für

Speicherbereiche, auf die während der gesamten Lebensdauer eines Dialogfensters zugegriffen werden kann und die im Anschluß daran wieder freigegeben werden sollten. Da die Verknüpfung über einen Character–Zeiger abläuft, kann durch geschickt gewählte Zeigerstrukturen jede beliebige Information mit einem Fenster verbunden werden. Die dazu benötigten Funktionen stellt das C–Interface des SGen zur Verfügung. In den Dialogen des NoName–Modul–Informationssystems werden oft zwei und auch drei Listen des Typs aus Abbildung 5.2 benötigt, deren Speicher bei Beendigung des Dialogs wieder freigegeben werden muß. Um nun die Speicherbereiche für die zwei bzw. drei Listen an den Dialog zu knüpfen, werden in der Implementierung die Datenstrukturen aus Abbildung 5.3 verwendet.

```
typedef struct two_lists
{
        char **firlist;
        char **seclist;
        int  chosen;
        int  class;
} TWO_LISTS_ON_HEAP;
```

(a)

```
typedef struct three_lists
{
        char **firlist;
        char **seclist;
        char **thilist;
        int  chosen;
        int  class;
} THREE_LISTS_ON_HEAP;
```

(b)

Abb. 5.3: Datenstrukturen für (a) zwei bzw. (b) drei Listenobjekte in einem Dialogfenster

Die Einträge `**firlist`, `**seclist` und `**thilist` sind jeweils Zeiger auf ein Array von `char`–Zeigern, wie es in Abbildung 5.2 dargestellt ist. In der `int`–Variablen `chosen` kann der Index eines ausgewählten Listenelements festgehalten werden. Die `int`–Variable `class` enthält in numerischer Codierung den Spezifikationstyp des aktuell betrachteten Sprachkonstruktes.

In Abschnitt 2.4.3 haben wir bereits die verschiedenen Funktionen angesprochen, welche für die Fensterverwaltung und die Menüpunkterstellung benötigt werden. Die in den folgenden Abschnitten angegebenen Algorithmen sind in angepaßter Form in den Kommando–Funktionen der Menüpunkte, in den Initialisierungsfunktionen der Dialogfenster und in den Aktionsfunktionen von Buttons zu finden. Für detailierte Auskünfte sei der interessierte Leser auf den Quellcode des Programms NoMIS (siehe Anhang) und auf die Programmbeispiele in [Gra95] verwiesen.

5.4 Die Funktionen der Toolbox `sgentools.c`

In Kapitel 2.4.4 haben wir bereits einige interne Datenstrukturen des SGen erläutert. Besonders der komplexe Aufbau eines `_browser`–Records verhindert komfortables Arbeiten, obwohl einige Struktur–Selektionen vom System zur Verfügung gestellt werden. Aus diesem Grund entwickelten Dietmar Ernst und ich eine Toolbox, die häufig benötigte Funktionen zur Verfügung stellt. Bemerkenswert ist, daß alle Funktionen dieser Toolbox unabhängig vom aktuellen Editor sind. Dies bedeutet, daß alle diese Funktionen nicht nur bei der Erzeugung des NoName–Editors, sondern auch bei der Erstellung anderer syntaxgesteuerten Editoren in anderen Projekten verwendet werden können.

Im folgenden sind die Funktionen der Toolbox aufgelistet und kurz erläutert. Dazu sei $p = (X_0 \rightarrow X_1 \ldots X_n)$ eine Produktion der kontextfreien Grammatik, welche der betrachteten Programmiersprache zugrunde liegt. X_0 sei das Nichtterminalsymbol, mit welchem die Wurzel der aktuellen Selektion im Editor beschriftet ist.

- `get_lhs_of_production()`
  Liefert das Nichtterminalsymbol X_0.

- `get_name_of_son(SON_NO son_no)`
  Liefert das Nichtterminalsymbol $X_{\texttt{son\_no}}$. Der Typ `SON_NO` entspricht dem Typ `int` aus der Programmiersprache C.

- `get_prodname_of_production()`
  Liefert den Namen der Produktion p.

- `get_prodname_of_son(SON_NO son_no)`
  Liefert den Namen der Produktion, die am `son_no`-ten Sohn der Produktion p angewendet wurde. Wenn `son_no` $> n$ oder `son_no` < 0, dann wird ein Fehler zurückgegeben. Der Typ `SON_NO` entspricht dem Typ `int` aus der Programmiersprache C.

- `get_text_of_selection(STRING viewname)`
  Liefert die textuelle Repräsentation der aktuellen Selektion in dem Editorfenster, welches die View `viewname` anzeigt.

- `set_selection(SON_NO son_no)`
  Ändert die aktuelle Selektion. Falls `son_no` $= 0$, dann wird der Vaterknoten von X_0 die Wurzel der aktuellen Selektion. Falls $0 <$ `son_no` $\leq n$, dann wird $X_{\texttt{son\_no}}$ die Wurzel der aktuellen Selektion. Falls `son_no` $> n$ oder `son_no` < 0, dann wird ein Fehler zurückgegeben.

- `error_in_selection()`
  Gibt `true` zurück, falls die aktuelle Selektion syntaktisch korrekt ist. Andernfalls wird `false` zurückgegeben. Für diesen Test wird die interne Funktion `TbtParseSelection(...)` verwendet, welche die textuelle Repräsentation des übergebenen Syntaxbaumes zu parsen versucht.

- `ascend_to_phylum(PROD_INSTANCE cur_node, char *phylum_name)`
  Liefert den kleinsten Teilbaum, dessen Wurzel mit dem Nichtterminalsymbol `phylum_name` beschriftet ist und den kompletten Teilbaum `cur_node` enthält.

- `get_unparse_of_prodinstance(PROD_INSTANCE root_of_syntree)`
  Liefert die textuelle Repräsentation des Syntaxbaumes `root_of_syntree`.

- `put_string_into_tree(char *ParseStr)`
  Ersetzt die aktuelle Selektion im Editor durch den Syntaxbaum, der sich durch Parsen der Zeichenfolge `ParseStr` ergibt.

- ```
  get_attrib_value(PROD_INSTANCE current_node,
                   char *attr_name)
  ```
  Ermittelt den Wert des Attributs `attr_name` am Knoten `current_node`.

- ```
  set_attrib_value(PROD_INSTANCE current_node,
                   char *attr_name,
                   PROD_INSTANCE new_value)
  ```
  Setzt den Wert des Attributs `attr_name` am Knoten `cur_node` auf den Wert `new_value`.

Weitere in der Toolbox vorhandene Funktionen sind lediglich Hilfsfunktionen und wurden direkt aus der Implementierung des SGen übernommen. Dieses Vorgehen war notwendig, da diese Funktionen in der SGen–Implementierung mit dem Zusatz `static` deklariert sind und somit aus der Toolbox nicht aufgerufen werden können.

5.5 Aufbau und Abbruch einer Verbindung zwischen dem Editor und der Datenbank

Der Aufbau einer Verbindung zwischen dem NoName–Editor und der Ingres–Datenbank wird durch den Menüpunkt `db-connect` angestoßen. Der Algorithmus 5.1 beschreibt die Vorgehensweise:

Algorithmus 5.1: (Verbindung zur Datenbank aufbauen)

Input: –
Output: –

```
 1  input database name into db_name;
 2  on errors goto connect_error;
 3  connect to database called db_name;
 4  db_connect_status ← connected;
 5  on errors call db_database_error();
 6  return;
 7  label connect_error:
 8     call db_database_error();
 9     db_connect_status ← disconnected;
10     return;
```

Die Eingabe des Datenbanknamens geschieht in einem eigenen Fenster. Dabei wird dem Benutzer defaultmäßig der Inhalt der UNIX–Umgebungsvariablen `NOMISDEFAULTDB` angezeigt. Dieser Name kann vom Anwender akzeptiert oder ersetzt werden.

Noch bevor der Anschluß an die Datenbank hergestellt wird, wird in Zeile 2 für ein eventuelles Scheitern eine Fehlerbehandlung vorgesehen. Dazu wird die folgende SQL–Anweisung verwendet:

EXEC SQL WHENEVER SQLERROR GOTO *connect_error*;

Dabei ist *connect_error* eine Marke, die dann angesprungen wird, wenn eine Datenbankverbindung nicht zustande kommen kann. An dieser Marke wird die Funktion `db_database_error()` aufgerufen, die den aufgetretenen Fehler von der Datenbank erfragt und in einem Fenster anzeigt. Hierfür wurde in der Implementierung die SQL–Anweisung

EXEC SQL INQUIRE_SQL (*:error_message* = ERRORTEXT);

verwendet, wobei die Variable *error_message* ein `char`–Array mit einer Größe von 1024 Bytes ist. Zu beachten ist, daß der Aufruf der Fehlerbehandlungsfunktion `db_database_error()` nicht direkt, sondern über den Umweg eines GOTOs erfolgt. Der Grund hierfür liegt in der unterschiedlichen Fortsetzung. Bei einem direkten Aufruf würde die Abarbeitung des Programms an der Stelle fortgesetzt, an welcher der Fehler aufgetreten ist. Dies bedeutet in diesem Fall, daß in Zeile 4 nicht unmittelbar unterscheidbar wäre, ob ein Fehler aufgetreten ist oder nicht. Man bräuchte dazu eine globale (boolesche) Variable oder dergleichen, die das Auftreten eines Fehlers anzeigen kann. Durch das GOTO dagegen kann eine Marke angesprungen werden, die ausschließlich im Fehlerfall erreicht wird. Somit ist eine Trennung zwischen erfolgreichem und nicht erfolgreichem Anschluß an die Datenbank möglich.

In Zeile 3 wird der Anschluß an die vom Benutzer angegebene Datenbank hergestellt. Die verwendete SQL–Anweisung lautet:

EXEC SQL CONNECT *:db_name*;

Die Variable *db_connect_status* gibt Auskunft über den Zustand der Verbindung zur Datenbank. Wenn der Anschluß an die Datenbank hergestellt werden konnte, dann wird *db_connect_status* auf *true* gesetzt, andernfalls erhält sie den Wert *false*. Diese Variable wird von den A–Testfunktionen der Menüpunkte abgefragt, die nur im Fall eines existierenden Datenbankanschlusses aktivierbar sein dürfen. In der Implementierung ist die Variable *db_connect_status* keine globale Variable, sondern eine `static`–Variable in der global bekannten Funktion `getset_db_connect_status(...)`, die als Parameter die Werte 0, 1 und 2 akzeptiert. Der Parameter 0 steht dabei für das Zurücksetzen des Wertes der Variablen auf *false* (gleichzusetzen mit *disconnected*), 1 steht für das Setzen auf den Wert *true* (entspricht *connected*), 2 steht für das Abfragen des aktuellen Wertes dieser Variablen. Über diesen Parameter können somit A–Testfunktionen den Zustand der Datenbankverbindung erfragen.

Wenn im Algorithmus 5.1 eine Verbindung zur Datenbank hergestellt werden konnte, dann wird für alle noch folgenden SQL–Anweisungen eine Fehlerbehandlung vereinbart (siehe Zeile 5). Bei jedem Auftreten eines Fehlers soll — wie schon beim Herstellen einer Datenbankverbindung — die zugehörige Ingres–Fehlermeldung angezeigt werden. Diese Vereinbarung wird mit der SQL–Anweisung

EXEC SQL WHENEVER ERROR CALL *db_database_error*;

getroffen. Im Gegensatz zu vorher wird hier kein GOTO, sondern das CALL verwendet, da die Bearbeitung an der Fehlerstelle fortgesetzt werden soll.

Eine existierende Verbindung kann auf zweierlei Arten vom Benutzer abgebrochen werden. Dafür stehen ihm die beiden Menüpunkte `db-disconnect` und `db-exit-all` zur Auswahl. Beide Punkte sind nur wählbar, wenn die Variable *db_connect_status* den Wert *connected* enthält. Die Kommando–Funktion des Menüpunktes `db-disconnect` arbeitet nach Algorithmus 5.2:

Algorithmus 5.2: (Verbindung abbrechen)

Input: –
Output: –

on errors continue;
disconnect from active database;
db_connect_status $\leftarrow$ *disconnected;*
return*;*

Der Menüpunkt `db-exit-all` arbeitet fast analog, es wird lediglich nach dem Abbruch der Verbindung auch noch die Sitzung mit dem Editor beendet:

Algorithmus 5.3: (Verbindung abbrechen und Editor beenden)

Input: –
Output: –

on errors continue;
disconnect from active database;
db_connect_status $\leftarrow$ *disconnected;*
exit(0)*;*
return*;*

In beiden Alternativen werden auftretende Fehler ignoriert, um den Abbruch der Datenbankverbindung zu garantieren. Würden Fehler beim Abbruch der Datenbankverbindung vom Programm bearbeitet, dann bestünde die Gefahr, daß das Programm nicht korrekt beendet werden kann. Daher wird bei der Verwendung von Embedded–SQL generell die Fehlerbehandlung vor dem Abbruch der Verbindung deaktiviert:

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

Die Abmeldung von der Datenbank erfolgt dann mit der bereits bekannten SQL–Anweisung

```
EXEC SQL DISCONNECT;
```

Durch die Zuweisung des Wertes 0 an die Variable *db_connect_status* wird sichergestellt, daß der Benutzer im Anschluß an die Abmeldung von der Datenbank nur den Punkt `db-connect` des Menüs `NoMIS` wählen kann, alle anderen Punkte dieses Menüs sind dadurch gesperrt.

5.6 Realisierung der Topicverwaltung

Die Topicverwaltung kann vom Anwender nur gestartet werden, wenn die Variable *db_connect_status* auf den Wert *connected* gesetzt ist. Dann kann der Benutzer aus einer Reihe von Schaltern auswählen, welche Aktion angestoßen werden soll. Im folgenden sind die Implementierungen der drei möglichen Aktionen — dies sind *Anzeigen, Hinzufügen und Löschen* von Topics — wieder im Pseudo–Code angegeben.

Für die **Anzeige aller gespeicherten Topics** wird ein Fenster geöffnet, in welchem eine Liste mit allen gespeicherten Topics angezeigt wird. Außerdem existiert ein Fensterelement zur Anzeige des Kommentars eines vom Benutzer mit der Maus gewählten Topics. Bei der Initialisierung der Fensterelemente wird nach Algorithmus 5.4 vorgegangen:

Algorithmus 5.4: (Anzeige aller gespeicherten Topics)

Input: –
Output: –

```
1   num_of_topics ← number of topics in the database;
2   allocate array for (num_of_topics + 1) char–pointers;
3   for each topic in the database do
4       allocate memory for topicname;
5       copy name of topic into allocated memory;
6       link it to next free array–element;
7   enddo;
8   sort array elements in alphabetical order;
9   link window element to the array;
10  link the array to the window;
11  return;
```

Für das Fensterelement, welches die Liste der verfügbaren Topics anzeigen soll, wird eine Datenstruktur aufgebaut, wie sie in Abbildung 5.2 zu sehen ist. In Zeile 1 des Algorithmus 5.4 wird zuerst die Anzahl der gespeicherten Topics ermittelt und festgehalten. Dies ist wie folgt implementiert:

```
EXEC SQL SELECT COUNT(*) INTO :num_of_topics
FROM topic;
```

Die Anzahl der benötigten Array–Elemente entspricht der Anzahl der gespeicherten Topics zuzüglich dem NIL–Zeiger als Abschluß–Element.

In den Zeilen 3 bis 7 wird jeder Topicname in den dazu allokierten Speicher kopiert und in das Feld eingehängt. Die anzuzeigenden Topicnamen werden aus der Datenbank erfragt. Dazu wird der SQL–Cursor *topiccursor* deklariert für die Anweisung

EXEC SQL SELECT *name* FROM *topic*;

In einer Schleife werden dann die Namen mittels

EXEC SQL FETCH *topiccursor* INTO *:esql_topic_name*;

in die Variable *esql_topic_name* eingelesen und anschließend weiterverarbeitet. In Zeile 9 wird durch eine C–Interface–Funktion das Fensterelement mit den Werten der Datenstruktur initialisiert. Damit später der allokierte Speicher wieder freigegeben werden kann, muß der Zeiger auf das Array gesichert werden. Dazu verwenden wir in Zeile 10 die in Abschnitt 5.3 beschriebene Möglichkeit, Speicherbereiche an ein Fenster zu knüpfen.

Das **Löschen eines gespeicherten Topics** erfolgt ähnlich der soeben beschriebenen Anzeige. Dabei wird die Liste aller verfügbaren Topics analog zu dem Algorithmus 5.4 erstellt. Wenn der Anwender aus dieser Liste ein Topic zum Löschen auswählt und den Löschvorgang durch Drücken des Buttons `delete` anstößt, dann wird nach dem Algorithmus 5.5 vorgegangen:

Algorithmus 5.5: (Löschen eines gespeicherten Topics)

Input: –
Output: –

```
1   topic_number ← unique code of topic;
2   if (topic_number is used) then
3      output("Topic is in use!");
4   else
5      delete topic;
6   endif;
7   return;
```

In Zeile 1 wird aus der Datenbank der Schlüssel des selektierten Topics erfragt. Dies geschieht über die SQL–Anweisung

EXEC SQL SELECT *topicno* INTO *:topic_number*
FROM *topic*
WHERE *name = :topic_name*;

Der Name des Topics wird durch den Benutzer vorgegeben, indem er aus der Liste aller verfügbaren Topics mit der Maus das zu löschende Topic auswählt.

In Zeile 2 wird getestet, ob das Topic gelöscht werden darf. Dazu werden Anfragen an die Datenbank gestellt, in denen die Anzahl derjenigen Sprachkonstrukte ermittelt wird, die diesem Topic zugeordnet sind:

```
EXEC SQL SELECT COUNT(*) INTO :usedx
FROM sktopicx
WHERE topicno = :topic_number;
```

wobei für `x` die Werte 1, 2, 3 und 4 eingesetzt werden, um alle Partitionen zu testen.

Der Test liefert genau dann `true`, wenn mindestens ein zugeordnetes Sprachkonstrukt existiert, d.h. wenn gilt:

$$used1 + used2 + used3 + used4 > 0$$

In diesem Fall wird dem Benutzer eine Fehlermeldung angezeigt, der Löschvorgang wird abgebrochen.

Wenn der Test den Wert `false` ergibt, dann wird das Sprachkonstrukt durch die folgende SQL–Anweisung aus der Datenbank entfernt:

```
EXEC SQL DELETE FROM topic WHERE name = :topic_name;
```

Die Dateneingabe beim **Anlegen eines neuen Topics** erfolgt in einem Fenster, welches jeweils ein Dialogelement für den Namen und den Kommentar enthält. Die Speicherung geht entsprechend dem Algorithmus 5.6 vor:

Algorithmus 5.6: (Neues Topic speichern)

Input: –
Output: –

```
input name of new topic;
input comment of new topic;
if (there exists a topic with this name) then
   output("The topic already exists!");
else
   topic_no ← unique code for new topic;
   store topic with (key = topic_no);
endif;
return;
```

Die Eingabe des Namens und des Kommentars in den Zeilen 1 und 2 kann in beliebiger Reihenfolge erfolgen. Der Test auf eindeutige Namen der Topics in Zeile 3 könnte auch der Datenbank überlassen werden, doch ist die Ingres–Fehlermeldung aufgrund deren numerischen Codierung nicht besonders aussagekräftig. Daher wird dieser Test explizit vom Programm durchgeführt. Dazu wird in der Relation `topic` nach dem Vorkommen des eingegebenen Namens gesucht. Bei einem Auffinden ergibt der Test den Wert `true`, andernfalls den Wert `false`. Verwendet wird dazu die folgende SQL–Anweisung:

```
EXEC SQL SELECT COUNT(*) INTO :existing
FROM topic
WHERE name = :topic_name;
```

Dabei enthält die Variable *topic_name* den vom Benutzer eingegbenen Namen des neuen Topics.

In Zeile 6 wird ein eindeutiger numerischer Code für das neue Topic ermittelt (dieser Code wird von anderen Relationen als Fremdschlüssel verwendet). Dazu wird der maximale bisher verwendete Code aus der Datenbank erfragt, um 1 erhöht und an das neue Topic vergeben. Dabei ist allerdings Vorsicht geboten, da Ingres im Falle einer leeren Relation einen beliebigen Wert als Maximum zurückliefert. Das bedeutet, daß vor der Maximumbestimmung noch getestet werden muß, ob die Relation `topic` leer ist.

Das neue Topic kann dann mit dem entsprechenden numerischen Schlüssel in der Datenbank abgelegt werden:

```
EXEC SQL INSERT INTO topic
VALUES (:topic_name, :num_key, :topic_comment);
```

5.7 Realisierung des Menüpunktes `db-store`

Da gespeicherte Sprachkonstrukte im Sinne einer „Black Box“ verwendet werden sollen, müssen mehrere Bedingungen erfüllt sein, um die Aufnahme eines Sprachkonstruktes in die Datenbank anstoßen zu können. Zuerst muß eine Verbindung an die Datenbank existieren. Dazu wird die Variable *db_connect_status* abgefragt. Desweiteren muß der Anwender eine Spezifikation im Editor markiert haben, wobei Import–Spezifikationen nicht berücksichtigt werden. Hierfür wird der Name der Produktion, die an der Wurzel des Teilbaumes der momentan aktuellen Selektion im NoName–Editor angewendet wurde, verglichen mit dem Namen „specList“. Nur bei Gleichheit ist eine Spezifikation markiert. Den Produktionsnamen erhält man durch die folgende Anweisung:

```
prod_name = sym_id(lhs(production(selection_apex(
            SE_selection(br_atree(cur_browser))))));
```

Um festzustellen, ob eine Import–Spezifikation markiert ist, wird der Namen der Produktion (analog zu obiger Anweisung) am ersten Sohn des aktuellen Teilbaums ermittelt. Wenn dieser Produktionsname gleich der Zeichenreihe „ImportSpec“ ist, dann liegt eine Importspezifikation vor, eine Speicherung ist somit nicht möglich.

Schließlich muß die markierte Spezifikation korrekt bezüglich der statischen Semantik der Sprache NoName sein. Für diesen Test existiert für jede Spezifikation ein SSL–Attribut namens `as_error`, das im Fehlerfall den Wert `true` und sonst den Wert `false` enthält. Die Abfrage des Attributwertes in der Programmiersprache C gestaltet sich etwas kompliziert. Prinzipiell übernimmt diese Abfrage die interne C–Funktion `get_attrib_value(tree,attr)`, deren erstes Argument ein Zeiger auf den Syntaxbaum ist, dessen Wurzel das Attribut `attr` besitzt. Da die Markierung im NoName–Editor eine Liste von Spezifikationen ist, muß die zu speichernde Spezifikation herausgefiltert werden, d.h. `tree` muß ein Zeiger auf den ersten Sohn der Spezifikationsliste sein. In einem Baum kann man von einem Knoten zu dessen $n-ten$ Sohn gelangen, indem die interne C–Funktion `son(tree,n)` verwendet wird.

Das Fenster, in welchem der Anwender die benötigten Eingaben tätigt, enthält eine Liste aller gespeicherten Topics (genannt `possible-topics`), die analog zum Algorithmus 5.4 erzeugt wird. Eine zweite Liste (`selected-topics` genannt) enthält alle Topics, die der Benutzer dem neuen Sprachkonstrukt zuordnen will. Diese Liste ist anfangs ebenso wie der Name, die Version und der Kommentar zum Sprachkonstrukt leer. Der Ablauf der Datenerfassung ist im Algorithmus 5.7 wiedergegeben:

Algorithmus 5.7: (Eingabe der Sprachkonstruktdaten)

Input: –
Output: –

input name of new sk;
optional input version and comment of new sk;
optional select topics;

Da die Eingaben in einem Dialogfenster erfolgen, existiert prinzipiell *keine* Reihenfolge, in welcher die Daten einzugeben sind. Der Einfachheit halber sind aber im Algorithmus 5.7 die Eingaben auf mehrere Befehlszeilen aufgeteilt.

Die Selektion von Topics aus der `possible-topics`–Liste verwendet eine Besonderheit, die vom C–Interface des SGen zur Verfügung gestellt wird. Das Anklicken eines Listeneintrages mit der Maus bewirkt den Aufruf einer vom Programmierer anzugebenden Funktion, die den Index des angeklickten Eintrages in der Liste als Parameter hat. Im Falle eines Mausklicks auf einen Eintrag in der `possible-topics`–Liste bewirkt die zugehörige Funktion, daß der angeklickte Eintrag in die `selected-topics`–Liste übernommen wird, falls er nicht schon darin enthalten ist. Wird auf die `selected-topics`–Liste geklickt, dann bewirkt die zuständige Funktion, daß der entsprechende Eintrag entfernt und somit die Selektion zurückgenommen wird. Bei jeder Änderung der `selected-topics`–Liste wird die resultierende Liste mittels des Sortierverfahrens Quicksort alphabetisch geordnet.

Die Angabe einer Version und eines Kommentars sind ebenso optional wie die Zuordnung von Topics. Der Name allerdings ist zwingend. Daher wird beim Auswählen des Buttons `save` in diesem Dialog zunächst getestet, ob tatsächlich ein Name eingegeben wurde. Wenn dies nicht der Fall ist, erhält der Benutzer eine entsprechende Fehlermeldung. Andernfalls wird die Speicherung entsprechend Algorithmus 5.8 und Algorithmus 5.9 durchgeführt:

Algorithmus 5.8: (Steuern des Speichervorganges)

recursive_storage(_actual_spec, func_counter_**)**

Input:	actual_spec:	Zeiger auf den Syntaxbaum des zu speichernden Sprachkonstruktes.
	func_counter:	Zähler für Hilfsfunktionen.
Output:	func_counter:	aktualisierter Zähler für Hilfsfunktionen.

```
sk_code ← store_sk(actual_spec, func_counter);
attr_instance ← instance of attribute asdb_code;
replace value of attribute instance attr_instance with sk_code;
nec_speclist ← speclist of necessary SK's;
for (each spec in nec_speclist) do
    if (spec is not saved in the database) then
        func_counter ← recursive_storage(spec, func_counter+1);
    endif;
    save dependence of actual_spec upon spec;
endfor;
return func_counter;
```

Im Abschnitt 3.3.2 wurde bereits auf die Speicherung aller benötigten Sprachkonstrukte, der sogenannten Hilfskonstrukte, und die damit verbundene Namensgebung hingewiesen. Hier stellt sich nun die Frage, wie diese Funktionalität implementiert werden kann.

Zuerst muß eine geeignete Strategie gefunden werden, mit welcher alle benötigten Sprachkonstrukte gefunden und gespeichert werden können. Wie der Name der Funktion `recursive_storage(...)` im Algorithmus 5.8 bereits andeutet, wird in der Implementierung auf eine Rekursion zurückgegriffen. Stellt man sich die Abhängigkeiten zwischen den Sprachkonstrukten als Baum vor, so wird prinzipiell nichts anderes als die Tiefensuche in einem „Abhängigkeitsgraph“ simuliert.

Für die fortlaufende Numerierung der Hilfskonstrukte benötigt man allerdings einen rekursionsunabhängigen Zähler. Dieser kann entweder über eine globale Variable oder über einen Funktionsparameter implementiert werden, wobei der — eventuell aktualisierte — Parameter auch als Rückgabewert fungiert. In der Implementierung wurde die Lösung über einen Parameter, $func\_counter$ genannt, gewählt.

Nun wurde im Abschnitt 3.3.2 auch auf die Vermeidung von Redundanz eingegangen. Die Frage, warum Redundanzvermeidung selbst bei nicht allzu umfangreichen Anwendungen erstrebenswert ist, scheint berechtigt. Doch die Antwort darauf ist leicht einzusehen. Die Erfahrung beim Programmieren in No-Name hat gezeigt, daß relativ viele Spezifikationen, insbesondere Funktionen, auf einigen anderen Sprachkonstrukten, den Hilfskonstrukten, basieren. Diese Hilfskonstrukte werden in der Regel aber nicht nur von einem, sondern von mehreren Sprachkonstrukten benötigt. Würde man bei jedem vom Benutzer initiierten Speicherungsvorgang *alle* Hilfskonstrukte in der Datenbank ablegen, ohne auf Redundanz zu testen, dann würde sich die Datenbank stark füllen. Dadurch werden Such– und Einfügezeiten unnötig lang, die Effizienz wäre nur noch ungenügend.

Diese leicht verständliche Betrachtung wird allerdings problematisch, wenn man versucht, Redundanz zu erkennen — und zwar automatisch. Da die Gleichheit zweier Funktionen im allgemeinen nicht entscheidbar ist, kann der Redundanzerkennung kein Funktionenvergleich zugrunde liegen. Die Lösung dieses Problems kann — zumindest teilweise — mittels eines Attributs erreicht werden, welches jeder Spezifikation zugeordnet wird. Dieses Attribut, welches in der Implementierung `asdb_code` genannt wird und den SSL–Typ `Int` besitzt, wird mit dem initialen Wert 0 belegt, d.h. beim Anlegen einer neuen Spezifikation durch den Benutzer erhält das Attribut den Wert 0. Soll ein Sprachkonstrukt nun in der Datenbank abgelegt werden, so ist dies nur möglich, wenn der Wert dieses Attributes gleich 0 ist. Bei einem erfolgreichen Speicherungsvorgang wird der Wert dieses Attributs durch den in der Datenbank eindeutigen numerischen Schlüssel des zugehörigen Sprachkonstruktes aus der Relation `skversion` ersetzt. Da dieser Schlüssel grundsätzlich von 0 verschieden ist, wird ein erneutes Ablegen in der Datenbank, das z.B. durch das Speichern eines abhängigen Sprachkonstruktes initiiert werden kann, unterbunden.

Statt der Lösung über das Attribut `asdb_code` wurde auch der Vorschlag in Betracht gezogen, den Syntaxbaum für gespeicherte Sprachkonstrukte zu modifizieren. Dies ist allerdings ohne eine Änderung der Sprache NoName nicht möglich, da Probleme beim Parsen bezüglich der Eingabe–Syntax auftreten. Deshalb wurde dieser Lösungsansatz nicht verwirklicht, auch wenn dessen Idee deutlich eleganter ist als die Lösung über ein entsprechendes Attribut.

Im folgenden wollen wir den Algorithmus 5.8, der den rekursiven Speicherungsvorgang steuert, noch etwas genauer betrachten.

Der Parameter *actual_spec* ist ein Zeiger auf den Teil des abstrakten Syntaxbaumes, welcher die zu speichernde Spezifikation darstellt. Wenn die Funktion `recursive_storage(...)` zum ersten Mal aufgerufen wird, dann zeigt *actual_spec* auf die Spezifikation, welche der Benutzer im Editor markiert hat. Der Parameter $func\_counter$ hat dabei den Wert 0.

In Zeile 1 des Algorithmus 5.8 wird zunächst die momentan aktuelle Spezifikation durch die Funktion `store_sk(...)` in der Datenbank abgelegt (siehe hierzu auch Algorithmus 5.9). Der dabei verwendete Schlüssel wird in der Variablen `sk_code` festgehalten. In Zeile 2 wird die Instanz des Attributes `asdb_code` ermittelt. Der Wert dieser Instanz wird anschließend durch den Schlüssel `sk_code` ersetzt. Nun wird eine Liste von Namen erstellt, die in der Definition des soeben gespeicherten Sprachkonstruktes angewandt vorkommen (Zeile 4). In der Implementierung wird dies durch drei SSL–Funktionen erledigt:

1. `GetLHID_of_Spec(actual_spec)` ermittelt alle links, d.h. definierend, vorkommenden Bezeichner der aktuellen Spezifikation. Diese werden als *IdentifierList* in der Variablen `left_id_list` gespeichert. *IdentifierList* ist dabei ein Syntaxbaum, welcher sich aus der kontextfreien Grammatik von NoName ableitet.

2. `GetIDsInSpec(actual_spec,left_id_list)` ermittelt zu allen Bezeichnern, die in der *IdentifierList* gespeichert sind, die Bezeichner, die definierend oder auch angewandt vorkommen und legt diese in der *IdentifierList* `all_id_list` ab.

3. `GetDifference(all_id_list,left_id_list)` bestimmt die Differenz der beiden Argumentlisten, d.h. das Ergebnis enthält alle angewandt vorkommenden Bezeichner, die für die aktuelle Spezifikation benötigt werden. Diese Ergebnisliste wird mit `right_id_list` bezeichnet und ist ebenfalls eine *IdentifierList*.

Da die *IdentifierList* ein Syntaxbaum ist, kann die an dessen Wurzel angewendete Produktion abgefragt werden. Wenn deren Name gleich der Zeichenreihe

„IdentifierListNil“ ist, dann ist die Liste leer, d.h. es existieren keine benötigten Spezifikationen. Andernfalls wird durch die SSL–Funktion `emGetImportantSpecs(...)` eine *SpecList* erzeugt, welche alle Spezifikationen enthält, in denen mindestens ein benötigtes Sprachkonstrukt definiert wird. Diese *SpecList* ist wiederum ein Syntaxbaum. Die Speicheradresse der Wurzel dieses Syntaxbaumes wird der Variablen `needed_speclist` zugewiesen. Da dieser Syntaxbaum einen Rechtsgrat darstellt, können die Spezifikationen ohne Probleme der Reihe nach bearbeitet werden.

Die `for`–Schleife testet nun für jede Spezifikation in dieser Liste, ob diese bereits in der Datenbank abgelegt ist. Dazu wird der Wert des Attributes `asdb_code` auf 0 getestet. Wenn der Wert gleich 0 ist, dann wird das eben betrachtete Hilfskonstrukt rekursiv mittels `recursive_storage(...)` in der Datenbank abgelegt. Dabei wird der Zähler für die Hilfskonstrukte um 1 hochgezählt, um die bereits beschriebene Namensgebung zu gewährleisten. Anschließend muß die Abhängigkeit zwischen der Spezifikation und deren Hilfskonstrukt in der Datenbank abgelegt werden (Zeile 9). Dazu wird folgende SQL–Anweisung verwendet:

EXEC SQL INSERT INTO *dependencies* VALUES (:skcode, :rel_skcode);

Dabei bezeichnet *rel_skcode* den numerischen Schlüssel des bereits in der Datenbank abgelegten Hilfskonstruktes.

Um das Konzept der Namensgebung stimmig abzuschließen, gibt die Funktion `recursive_storage(...)` den aktualisierten Zähler für Hilfsfunktionen zurück.

Die bereits angesprochene und auch im Algorithmus 5.8 verwendete Funktion `store_sk(...)` arbeitet entsprechend dem Algorithmus 5.9.

Algorithmus 5.9: (Speicherung eines Sprachkonstruktes)

store_sk(*nec_spec, func_counter***)**

Input:	nec_spec:	Zeiger auf den Syntaxbaum des zu speichernden Sprachkonstruktes.
	func_counter:	Zähler für Hilfsfunktionen.
Output:	skcode:	Eindeutiger Schlüssel für das gespeicherte Sprachkonstrukt.

if (*func_counter* $>$ 0) **then**
   skname $\leftarrow$ *new name dependent on func_counter;*
   class $\leftarrow$ *classification of nec_spec;*
endif*;*

```
5     if (there exists a name/version conflict with stored SK's) then
6        output("Cannot save because of name/version conflict!");
7        return 0;
8     endif;
9     skcode ← available code for new SK;
10    get date and programmer;
11    save data of SK;
12    save related topics;
13    save textual representation;
14    return skcode;
```

Die Parameter der Funktion `store_sk(...)` entsprechen in ihrer Bedeutung denen der Funktion `recursive_storage(...)`.

In der Funktion `store_sk(...)` muß nun die bereits beschriebene Namensgebung der Konstrukte realisiert werden. Dazu wird in Zeile 1 der Parameter `func_counter` zunächst auf 0 getestet. Nur wenn dieser Zähler größer als 0 ist, dann handelt es sich um ein Hilfskonstrukt, dessen Name wie oben beschrieben komponiert werden muß. Zusätzlich muß dann auch noch der zugehörige Spezifikationstyp ermittelt werden (Zeile 3). Dies geschieht durch Abfragen der angewendeten Produktion an der Wurzel des Teilbaumes, der die momentan betrachtete Spezifikation darstellt. Dazu wird in der Implementierung auf Selektionen zurückgegriffen, die in Abschnitt 2.4.4 beschrieben sind:

```
class = (char *) op_name(production(nec_spec));
```

In Zeile 5 wird auf Namens– und Versionskonflikte getestet. Derartige Konflikte können z.B. auftreten, wenn zufällig der Name eines Hilfskonstruktes mit einem bereits vergebenen Namen zusammenfällt. Da dies eher selten vorkommen wird, erhält der Benutzer lediglich eine Fehlermeldung mit dem Hinweis, daß er den Fehler manuell beheben muß. Da in diesem Fall keine Speicherung in der Datenbank erfolgen konnte, wird statt einem Schlüssel der Wert 0 zurückgegeben. Dadurch wird die Konsistenz des Verfahrens sichergestellt.

Wenn kein Namenskonflikt aufgetreten ist, dann wird in der Datenbank nach einem numerischen Schlüssel für das neue Sprachkonstrukt gesucht. Zeile 9 faßt einige Arbeitsgänge aus der Implementierung zusammen. Um möglichst kleine Schlüssel zu erhalten, werden Sprachkonstrukte beim Löschen nicht tatsächlich entfernt, sondern durch ein „Pseudo–Sprachkonstrukt“ ersetzt (Details hierzu siehe Abschnitt 5.9). Wenn ein solches „Pseudo–Sprachkonstrukt“ in der Datenbank enthalten ist, dann wird dieses beim Speichern eines neuen Konstruktes gelöscht und dessen Schlüssel weiterverwendet. Gibt es kein derartiges Konstrukt in der Datenbank, dann wird der maximale verwendete Schlüssel ermittelt, um 1 inkrementiert und dann für das neue Sprachkonstrukt verwendet. Anschließend werden die administrativen Daten in der Datenbank abgelegt:

EXEC SQL INSERT INTO *skversion*
VALUES (*:class, :sk_name, :sk_version, :sk_code, :programmer,*
:datum, :sk_comment);

Sodann werden alle zugeordneten Topics in einer Schleife entsprechend ihrem Auftreten in der `selected-topics`–Liste gespeichert. Für jede derartige Zuordnung wird die folgende SQL–Anweisung ausgeführt:

EXEC SQL INSERT INTO *sktopic*x
VALUES (*:skcode, :topno*);

Schließlich muß noch der NoName–Code in Stücken der Länge 1500 in der Datenbank abgelegt werden. Dazu muß zunächst die textuelle Repräsentation der Spezifikation `nec_spec` ermittelt werden. Dies geschieht durch die Funktion `get_unparse_of_prodinstance(...)`, die auch zu den selbstgeschriebenen Tool–Funktionen, aufbauend auf die interne Struktur des SGen, zählt:

```
nnm_code = get_unparse_of_prodinstance(nec_spec);
```

Die Aufteilung in einzelne Code–Fragmente erfolg in einer Schleife, in welcher für jedes Code–Fragment die folgende SQL–Anweisung ausgeführt wird:

EXEC SQL INSERT INTO *sk_nnmcode*
VALUES (*:skcode, :ctr, :nnm_code*);

Dabei ist die Variable `ctr` ein Zähler, der für die Numerierung der einzelnen Code–Fragmente benötigt wird.

Nach dem Speichern aller Daten wird der verwendete Schlüssel `skcode` an die aufrufende Funktion zurückgegeben.

5.8 Realisierung des Menüpunktes `db-retrieve`

Es müssen mehrere Bedingungen erfüllt sein, um den Menüpunkt `db-retrieve` aktivieren zu können. Zum einen muß ein Anschluß an die Datenbank existieren. Um dies zu testen, muß lediglich die Variable *db_connect_status* abgefragt werden. Zum anderen muß der Anwender im Editor einen Platzhalter markiert haben, an dem eine Spezifikation eingefügt werden kann[11]. Dabei muß der aktuelle Kontext `spec` heißen (Eigenschaften des Editors können [EGH+94b] entnommen werden). Um diese Bedingung zu testen, wird der Name der Produktion ermittelt, die an der Wurzel der aktuellen Markierung im Editor angewendet wurde:

```
prod_name = sym_id(lhs(production(selection_apex(
            SE_selection(br_atree(cur_browser))))));
```

In der Implementierung entspricht die rechte Seite dieser Zuweisung dem Aufruf der Tool–Funktion `get_lhs_of_production()`.

Wenn der Menüpunkt `db-retrieve` gewählt wurde, dann wird der Benutzer zunächst in einem Fenster aufgefordert, den gewünschten Spezifikationstyp auszuwählen. Dazu wird eine sogenannte Toggle–Box verwendet, deren vier Schaltfelder mit den vier möglichen Spezifikationstypen beschriftet sind. Toggle–Boxen haben die Eigenschaft, daß genau ein Feld aus allen angebotenen Schaltfeldern ausgewählt werden muß. In unserer Anwendung hier bedeutet dies, daß der Benutzer genau einen Spezifikationstyp auswählen kann bzw. muß.

Im Anschluß an die Auswahl des Spezifikationstyps wird ein Fenster geöffnet, in welchem die Wahl des gesuchten Sprachkonstruktes getroffen wird. Dieses Fenster enthält drei Listen. Die Liste mit allen gespeicherten Topics wird analog zum Algorithmus 5.4 aufgebaut. Ebenso funktioniert die Selektion und damit die Verwaltung der Liste mit den selektierten Topics so, wie es in Abschnitt 5.7 beschrieben wurde. Die dritte Liste bedarf der Erläuterung. Ihre Einträge setzen sich aus den Namen und zugehörigen Versionen aller Sprachkonstrukte zusammen, die den selektierten Topics zugeordnet sind. Wir werden diese Liste im folgenden `matching`–Liste nennen. Da die Listenelemente nur Character–Felder sein dürfen, müssen der Name und die Version eines Sprachkonstruktes in *eine* Zeichenfolge kopiert werden. Wenn der Benutzer aus dieser Liste ein Sprachkonstrukt auswählt, dann muß der Name von der Version wieder getrennt werden. Nun fehlt aber die Information, wo der Name endet bzw. wo die Version beginnt. Aus diesem Grund wird in jedem Listenelement die Länge des Sprachkonstrukt–Namens festgehalten. Dazu werden bei der Allokation des benötigten Speichers für jeden Listeneintrag fünf Bytes mehr bereit gestellt, als

[11] In der Version 1.0 des NoName–Editors heißt dieser Platzhalter `<specification>`.

die Länge des Namens und der Version zusammen benötigen. Da die Längenangabe aber nicht auf dem Bildschirm erscheinen soll, wird diese *nach* dem Begrenzer für Zeichenketten — in C ist das bekanntlich das Zeichen `'\0'` — in den allokierten Speicherbereich geschrieben. Eine andere Möglichkeit wäre die Einführung eines speziellen Zeichens, welches das Ende des Namens kennzeichnet. Dann allerdings dürfte dieses Zeichen nicht im Namen auftreten. Aufgrund dieser Beschränkung wurde diese Methode nicht implementiert. In Abbildung 5.4 ist eine Listenstruktur mit Einträgen der oben genannten Art visualisiert. Darin existiert z.B. ein Sprachkonstrukt namens `Sinus`, welches in der Version `1` existiert. Die Länge des Namens beträgt fünf, deshalb ist nach dem Begrenzer für Zeichenfolgen die Zahl `5` abgelegt.

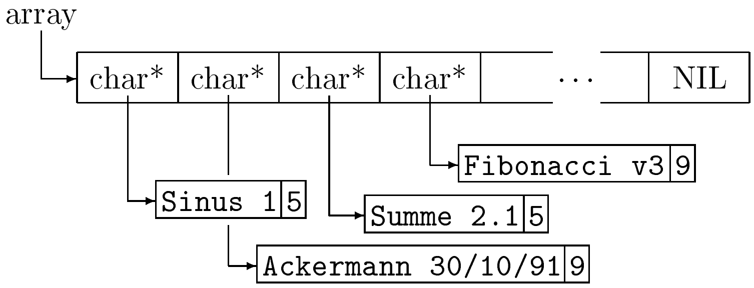


Abb. 5.4: Name und Version eines Sprachkonstruktes in *einer* Zeichenfolge, die Längenangabe ist nach dem Begrenzer für Zeichenfolgen abgelegt.

Der Algorithmus 5.10 gibt die Vorgehensweise beim Aufbau der Liste mit den passenden Sprachkonstrukten wieder. Diese Liste wird immer dann aufgebaut, wenn der Benutzer den Button `scan` drückt, um die Datenbank nach passenden Sprachkonstrukten zu durchsuchen.

Algorithmus 5.10: (Liste mit Sprachkonstrukten erstellen, die den ausgewählten Topics zugeordnet sind)

Input: –
Output: –

if (*list is already existing*) **then**
  free memory of all list elements;
endif;
if (*no topics selected*) **then**
  output(*"Please select at least one topic!"*);
else
  insert all selected topics into toptemp;
  matching_SKs ← *number of matching SKs;*
  allocate array for $(matching\_SKs + 1)$ *char–pointers;*

```
10        for (each matching SK) do
11            len ← length_of(name) + length_of(version) + 5;
12            allocate len bytes;
13            copy name and version into allocated memory;
14            save length of name behind string delimiter;
15            link allocated memory to next free array element;
16        enddo;
17        link the array to the window;
18        delete all tuples from toptemp;
19    endif;
```

Im Algorithmus 5.10 wird zuerst getestet, ob bereits eine nicht–leere `matching`–Liste existiert. Da der Anwender in der Regel nur den Button `scan` drückt, wenn er die Liste der selektierten Topics verändert hat, ist diese Liste nicht mehr auf dem aktuellen Stand. Somit kann der von ihr belegte Speicher wieder freigegeben werden. Wenn der Benutzer kein Topic ausgewählt hat, d.h. wenn die `selected-topics`–Liste leer ist, dann erhält er eine Fehlermeldung (Zeile 5). Andernfalls werden die numerischen Schlüssel aller selektierten Topics in der temporären Relation `toptemp` zwischengespeichert und Speicher für ein Array mit der benötigten Anzahl Elemente allokiert.

Der Grund für die Zwischenspeicherung aller selektierten Topics liegt in der einfacheren Weiterverarbeitung. Die passenden Sprachkonstrukte, d.h. Sprachkonstrukte, die mindestens einem der gewählten Topics zugeordnet sind, können recht einfach vom Datenbanksystem durch ein kartesisches Produkt von Relationen mit anschließender Selektion ermittelt werden, sofern auch die gewählten Topics in einer Relation abgelegt sind. Andernfalls müsste für *jedes* Sprachkonstrukt quasi „manuell", d.h. durch entsprechenden C–Code, getestet werden, ob es einem der gewählten Topics zugeordnet ist und sich somit qualifiziert, oder nicht.

Um für die `for`–Schleife von Zeile 10 bis Zeile 16 die passenden Sprachkonstrukte zur Verfügung stellen zu können, wird in der Implementierung zunächst ein Cursor deklariert, über welchen die Sprachkonstrukte abgearbeitet werden können (`x` wird dabei je nach Spezifikationstyp durch 1, 2, 3 oder 4 ersetzt):

```
EXEC SQL DECLARE skcursorx CURSOR FOR
SELECT DISTINCT skversion.skname, skversion.version
FROM skversion, toptemp, sktopicx
WHERE toptemp.topicno = sktopicx.topicno
AND sktopicx.skcode = skversion.skcode
ORDER BY skversion.skname, skversion.version;
```

Für jedes Sprachkonstrukt, das sich aufgrund dieses Cursors qualifiziert, wird entsprechend der Länge des Namens und der Version Speicher allokiert. In den Zeilen 13 und 14 wird genau das oben beschriebene Verfahren zur Speicherung der Namenlänge angewendet. Nachdem alle Character–Felder in das Array eingehängt wurden, kann das Array an das Fensterelement gebunden werden (Zeile 17). Schließlich können in Zeile 18 alle Tupel aus der temporären Relation `toptemp` gelöscht werden.

Wenn der Benutzer ein Sprachkonstrukt aus der `matching`–Liste auswählt, dann werden der Name, die Version und der entsprechende Kommentar in den dafür vorgesehenen Fensterelementen angezeigt. Um aus der in der `matching`–Liste angezeigten Zeichenfolge den Namen und die Version zu extrahieren — diese werden ja als Schlüssel für die Datenbankanfrage bezüglich des Kommentars verwendet —, benötigt man die Länge des Namens, die nach oben beschriebenem Verfahren in der Zeichenfolge abgelegt wurde.

Wenn ein Sprachkonstrukt ausgewählt wurde, dann kann der Benutzer durch Drücken des Buttons `retrieve` den Übertragungs–Vorgang anstoßen. Die dabei durchgeführten Arbeitsschritte sind im Algorithmus 5.11 wiedergegeben.

Algorithmus 5.11: (Rekursives Importieren aller benötigten Sprachkonstrukte)

recursive_retrieve(*skcode*)
Input: skcode: Numerischer Schlüssel des Sprachkonstruktes, das in das aktuelle NoName–Programm einzufügen ist.
Output: –

```
if (program needs SK identified by skcode) then
    set editor selection to an empty spec;
    retrieve NoName–code from database;
    put string into syntax tree;
    replace value of attribute asdb_code with skcode;
endif;
neccessary ← number of neccessary SKs for actual specification;
allocate memory for the keys of neccessary SKs;
copy keys of neccessary SKs in allocated memory;
for (each neccessary SK) do
        recursive_retrieve(SK);
enddo;
free allocated memory;
return;
```

Beim ersten Aufruf der Funktion `recursive_retrieve(...)` hat der Parameter *skcode* den Wert des numerischen Schlüssels des Sprachkonstruktes, welches der Anwender aus der `matching`–Liste ausgewählt hat. In Zeile 1 des Algorithmus 5.11 wird zuerst getestet, ob das momentan betrachtete Sprachkonstrukt dem aktuellen NoName–Programm hinzugefügt werden muß, oder ob es darin bereits enthalten ist. In der Implementierung wird dabei jede Instanz des Attributes *asdb_code* mit dem hier betrachteten Schlüssel (*skcode*) verglichen. Dazu werden einige interne C–Funktionen benötigt, die wir nicht näher betrachten wollen, weil sie nur eine untergeordnete Rolle spielen. Wenn kein Attribut mit dem Wert des Schlüssels existiert, dann muß das Sprachkonstrukt aus der Datenbank in das NoName–Programm importiert werden. Dazu muß zuerst eine „leere" Spezifikation markiert werden. Beim ersten Aufruf dieser Funktion ist diese Bedingung noch durch den Benutzer erfüllt, bei allen späteren Aufrufen wird vom Programm eine freie Spezifikation geschaffen. Dies geschieht durch folgende Anweisung:

```
Execute_command(Str(str_to_str0("specList")));
```

Die Funktion `Execute_command` simuliert im wesentlichen das Betätigen eines Transform–Buttons, der mit „specList" beschriftet ist. Da die Markierung im Editor allerdings keine Liste, sondern eine einzelne Spezifikation sein muß, wird die Selektion noch in den ersten Sohn der Liste verschoben:

```
set_selection(1);
```

Der gespeicherte NoName–Code wird in Zeile 3 aus der Datenbank geholt. Wenn der Code dabei aus mehreren Teilen besteht, so werden diese in der richtigen Reihenfolge zusammengesetzt. Die daraus resultierende Zeichenkette wird durch die Funktion `put_string_into_tree(...)` geparst und an der aktuell markierten Editor–Position in den Syntaxbaum eingetragen. Schließlich muß der Wert des Attributs *asdb_code* auf den Wert des numerischen Schlüssels des Sprachkonstruktes gesetzt werden.

Die noch verbleibenden Anweisungen im Algorithmus 5.11 dienen dem rekursiven Import der benötigten Hilfskonstrukte. Hierfür wird in den Zeilen 7 bis 9 eine Liste mit den Schlüsseln aller nötigen Sprachkonstrukte erstellt. Die Daten werden dabei aus der Relation `dependencies` ermittelt:

```
EXEC SQL SELECT required INTO :required_sk
FROM dependencies
WHERE dependent = :skcode;
```

Für jedes dieser Hilfskonstrukte, deren Schlüssel in einer Schleife in die Variable `required_sk` übertragen wird, wird in den Zeilen 10 bis 12 des Algorithmus 5.11 die Funktion `recursive_retrieve(...)` aufgerufen. Wenn alle Hilfskonstrukte bearbeitet sind, dann kann der allokierte Speicher wieder freigegeben werden.

5.9 Realisierung des Menüpunktes `db-delete`

Der Menüpunkt `db-delete` kann ausgewählt werden, falls eine Anbindung an die Datenbank existiert. Dazu wird lediglich die Variable *db_connect_status* auf den Wert *connected* abgefragt.

Analog zum Menüpunkt `db-retrieve` wird der Benutzer zuerst in einem Fenster aufgefordert, den gewünschten Spezifikationstyp auszuwählen. Auch hier wird eine Toggle–Box verwendet, deren Schaltfelder mit den vier möglichen Spezifikationstypen beschriftet sind.

Die Auswahl des zu löschenden Sprachkonstruktes erfolgt ebenfalls analog dem in Abschnitt 5.8 beschriebenen Verfahren. Es handelt sich hierbei um dasselbe Fenster, lediglich der Button `retrieve` wurde durch den Button `delete` ersetzt. Nach dem Drücken dieses Buttons wird gemäß Algorithmus 5.12 verfahren.

Algorithmus 5.12: (Rekursives Löschen von Sprachkonstrukten)

recursive_delete(*sk_code***)**
Input: sk_code: Numerischer Schlüssel des Sprachkonstruktes, das aus der Datenbank gelöscht werden soll.
Output: –

```
neccessary ← number of neccessary SKs for actual specification;
allocate memory for the keys of neccessary SKs;
copy keys of neccessary SKs in allocated memory;
if ( sk_code is required for other SKs) then
   delete ← input("delete anyway? (y/n)");
else
   delete ← input("delete? (y/n)");
endif;
if ((user is allowed to delete SK) and (delete = true)) then
   delete all dependencies of SK;
   delete connections to all topics;
   delete NoName–code;
   mark SK as "DeLEteD" in table skversion;
   if (deleted SK is part of program) then
      replace value of attribute asdb_code with 0;
   endif;
endif;
```

```
18   for (each neccessary SK) do
19       if (SK is not deleted yet) then
20          recursive_delete(SK);
21       endif;
22   enddo;
23   free allocated memory;
24   return;
```

Das Löschen eines Sprachkonstruktes beginnt mit der Ermittlung aller benötigten Hilfskonstrukte (Zeilen 1 bis 3 des Algorithmus 5.12). Dazu wird zunächst die Anzahl der benötigten Sprachkonstrukte aus der Tabelle `dependencies` ermittelt, dann entsprechend Speicher allokiert und die Schlüssel der Hilfskonstrukte in den allokierten Speicher kopiert.

Um versehentliches Löschen zu vermeiden, muß der Benutzer jeden einzelnen Löschvorgang bestätigen. Dies geschieht in einem separaten Fenster, in welchem Name, Version, Datum und Programmierer des Sprachkonstruktes angezeigt werden. Da die Behandlung dieses Fensters die Abarbeitung des Algorithmus 5.12 unterbrechen soll, entsteht das folgende Problem:

In grafischen Benutzeroberflächen wird jeder Mausklick, das Drücken eines Buttons, das Öffnen eines Fensters usw. als *Event* bezeichnet. Alle Events werden vom System in einer Event–Schleife sequentiell abgearbeitet. Dies bedeutet, daß ein Event komplett bearbeitet sein muß, bevor der nächste aus der Warteschlange geholt und bearbeitet werden kann. Nun läuft aber der Algorithmus 5.12 selbst in einer Fensterfunktion ab, die auf den Event „Drücken des Buttons `delete`“ hin aufgerufen wird. Diese Funktion kann nicht direkt durch einen neuen Event unterbrochen werden, weshalb das Öffnen des Fensters verzögert wird, bis der Löschvorgang abgeschlossen ist. Diese Problematik führte zur Entwicklung der beiden Funktionen `Sleep()` und `Wake()` aus dem Modul `nomis_process.sc`, die folgende Funktionalität haben:

- `Sleep()` extrahiert aus der Event–Warteschlange den nächsten Event und gibt ihn zur Bearbeitung an das System weiter, solange die globale Variable *sleepend* den Wert *false* enthält. Dies bedeutet, daß mindestens ein Event, der in dieser Schleife bearbeitet wird, den Wert der Variablen *sleepend* auf *true* setzen muß. Dann bricht die Funktion `Sleep()` die Bearbeitung der Warteschlange ab und kehrt zur aufrufenden Funktion zurück.

- `Wake()` setzt den Wert der Variablen *sleepend* auf *true* und kehrt sofort zur aufrufenden Funktion zurück.

In unserem Fall geben uns diese beiden Funktionen die Möglichkeit, den Löschvorgang für den Benutzer kontrollierbar zu machen. In der Implementierung folgt auf den Befehl zum Öffnen des Fensters der Aufruf der Funktion `Sleep()`. Dadurch wird das Fenster sofort geöffnet, der Benutzer wählt zwischen den Buttons `yes` und `no` und in der jeweils aktivierten Funktion wird die Funktion `Wake()` aufgerufen, durch welche der normale Fortgang der Event–Bearbeitung wieder hergestellt wird. Die Zeilen 5 und 7 im Algorithmus 5.12 sind starke Vereinfachungen der Implementierung. Die Unterscheidung zwischen Sprachkonstrukten, die selbst Hilfskonstrukt sind für andere Sprachkonstrukte, und solchen, die keine Hilfskonstrukte sind, ermöglicht die Verwendung verschiedener Dialogfenster zur Bestätigung des Löschvorganges. Falls ein Sprachkonstrukt gelöscht werden soll, welches von anderen Konstrukten benötigt wird, so existiert in dem entsprechenden Fenster ein zusätzlicher Button, mit welchem die abhängigen Sprachkonstrukte angezeigt werden können. Welches der beiden Dialogfenster geöffnet werden soll, entscheidet sich in Zeile 4. Dazu wird folgende SQL–Anfrage an die Datenbank gestellt:

EXEC SQL SELECT COUNT(*) INTO *:req_ctr*
FROM *dependencies*
WHERE *required = :skcode*

Wenn abhängige Sprachkonstrukte existieren, dann enthält die Variable *:req_ctr* einen Wert größer Null, andernfalls ist der Wert gleich Null.

In Zeile 9 wird getestet, ob der Benutzer die Erlaubnis besitzt, das gewählte Sprachkonstrukt zu löschen. In der vorliegenden Implementierung ist das Löschen eines Sprachkonstruktes dem Programmierer vorbehalten, der es in der Datenbank abgelegt hat. Für den Test muß lediglich die Übereinstimmung des Programmierernamens aus der Datenbank mit dem aktuellen Benutzernamen geprüft werden. Um den aktuellen Benutzer zu ermitteln, wird auf die Umgebungsvariable `USER` zugegriffen. In der Programmiersprache C erfolgt dieser Zugriff über die Funktion `getenv("USER")`. Selbstverständlich wird durch diese Vorgehensweise kein vollständiger Schutz vor unerlaubten Löschoperationen erreicht, da über das UNIX–Kommando `setenv` der Umgebungsvariablen `USER` jederzeit ein anderer Wert zugewiesen werden kann.

Wenn der Benutzer das Sprachkonstrukt löschen möchte und auch dazu autorisiert ist, dann werden aus der Relation `dependencies` alle Abhängigkeiten gelöscht, d.h. es werden sowohl die Tupel entfernt, welche eine Abhängigkeit von anderen Konstrukten repräsentieren, als auch die Tupel, welche die Notwendigkeit eines Sprachkonstruktes für andere Konstrukte festhalten. Dies geschieht durch die folgende SQL–Anweisung:

EXEC SQL DELETE FROM *dependencies*
WHERE *dependent = :skcode*
OR *required = :skcode;*

Ebenso werden die zugehörigen Tupel aus der Relation `sktopic`x (x ersetzt durch den zugehörigen Spezifikationstyp) und der NoName–Code aus der Relation `sk_nnmcode` mit den folgenden SQL–Anweisungen entfernt:

EXEC SQL DELETE FROM *sktopic*x
WHERE *skcode = :skcode;*

EXEC SQL DELETE FROM *sk_nnmcode*
WHERE *skcode = :skcode;*

Im Abschnitt 5.7 wurde bereits darauf hingewiesen, daß Sprachkonstrukte aus der Relation `skversion` nicht gelöscht, sondern nur durch ein Pseudo–Sprachkonstrukt mit dem Namen „DeLEteD" ersetzt werden[12]. Dies ermöglicht bei nachfolgenden Speicherungsvorgängen die Wiederverwendung nicht mehr benötigter Schlüssel, ohne diese separat, z.B. in einer eigenen Relation, speichern zu müssen. Da Löschvorgänge in der Datenbank relativ selten sein werden, hält sich der Platzverbrauch in vertretbaren Grenzen. Für diese Änderungsoperation wird die folgende SQL–Anweisung verwendet:

EXEC SQL UPDATE *skversion*
SET *skname = 'DeLEteD', version = 'DeLEteD',*
    programmer = 'DeLEteD', comment = 'DeLEteD'
WHERE *skcode = :skcode;*

Wenn das gelöschte Sprachkonstrukt in dem aktuell editierten NoName–Programm zuvor aus der Datenbank importiert wurde, dann muß das zugehörige Attribut *asdb_code* auf den Wert 0 gesetzt werden (Zeilen 14 bis 16). In dem hierfür benötigten Test werden alle Instanzen des Attributes *asdb_code* des aktuellen NoName–Programms mit dem Schlüssel des gelöschten Sprachkonstruktes verglichen.

Bei dem nun folgenden Abstieg in die Rekursion in den Zeilen 18 bis 22 muß noch eine Besonderheit beachtet werden. Man kann die Abhängigkeiten der Sprachkonstrukte untereinander als Baum darstellen. Die Knoten entsprechen dabei den Sprachkonstrukten, wobei Sprachkonstrukte mehrmals im Baum vorkommen dürfen, die Kanten entsprechen den Abhängigkeiten. Dann entspricht

[12] Der Name „DeLEteD" wurde in der Annahme gewählt, daß wohl kein Benutzer diesen Namen für eigene Sprachkonstrukte verwenden wird.

die implementierte Vorgehensweise einem depth–first–Durchlauf dieses Baumes. Somit besteht die Möglichkeit, daß aufgrund der Information im Baum ein Sprachkonstrukt gelöscht werden soll, das bereits in einer früheren Inkarnation der Rekursionsfunktion gelöscht wurde. Deshalb wird das rekursive Löschen eines Sprachkonstruktes nur dann angestoßen, wenn es noch in der Datenbank enthalten ist (Zeilen 19 bis 21).

Zum Schluß muß noch der allokierte Speicher für die Liste der benötigten Hilfskonstrukte freigegeben werden. Dann wird die aktuelle Inkarnation verlassen.

5.10 Realisierung des Menüpunktes `db-output`

Der Menüpunkt `db-output` kann nur dann ausgewählt werden, wenn die Variable *db_connect_status* den Wert *connected* enthält, d.h. es muß ein Anschluß an die Datenbank existieren.

Zur Wahl stehen drei Listen, die aus einer Toggle–Box ausgewählt werden können. Im folgenden sind die Algorithmen für die Erzeugung der drei Listen angegeben.

Der Algorithmus 5.13 zeigt die Erstellung einer LaTeX–Datei, welche eine Übersicht über die gespeicherten Sprachkonstrukte enthält.

Algorithmus 5.13: (Liste mit Sprachkonstrukten erzeugen)

Input: –
Output: –

```
1   open file constructs.list;
2   if (error occurred while opening file) then
3      output("Cannot open file constructs.list!");
4   endif;
5   print LaTeX–document header;
6   print headline;
7   print name of database and current date;
8   for (each SK in the database) do
9          print name, version, classification, programmer, date, comment;
10  enddo;
11  print LaTeX–document closing;
12  close file constructs.list;
```

Die Daten eines Sprachkonstruktes werden in Zeile 9 ausgegeben. In der Implementierung ist hierfür eine `tabular`–Umgebung vorgesehen, damit die Daten eines Sprachkonstruktes nicht aufgrund eines Seitenumbruches auf zwei

Seiten verteilt werden können. Als gelöscht markierte Sprachkonstrukte, also Sprachkonstrukte mit dem Namen „DeLEteD“, werden nicht in die Liste aufgenommen. Die benötigten Sprachkonstruktdaten werden alle aus der Relation `skversion` extrahiert.

Im Algorithmus 5.14 ist die Erstellung einer LaTeX–Datei mit den Daten aller gespeicherten Topics wiedergegeben.

Algorithmus 5.14: (Liste mit Topics erzeugen)

Input: –
Output: –

```
open file topic.list;
if (error occurred while opening file) then
   output("Cannot open file topic.list!");
endif;
print LaTeX–document header;
print headline;
print name of database and current date;
for (each topic in the database) do
        print name, comment;
enddo;
print LaTeX–document closing;
close file topic.list;
```

Algorithmus 5.14 arbeitet analog dem Algorithmus 5.13. Auch hier werden die Daten in eine `tabular`–Umgebung eingeschlossen, um unerwünschte Umbrüche zu vermeiden. Die Topicdaten kommen alle aus der Relation `topic`.

Der Algorithmus 5.15 zeigt die Vorgehensweise bei der Erstellung einer LaTeX–Datei, welche die Zuordnungen zwischen Sprachkonstrukten und Topics enthält.

Algorithmus 5.15: (Liste mit Zuordnungen zwischen Sprachkonstrukten und Topics erzeugen)

Input: –
Output: –

```
open file assign.list;
if (error occurred while opening file) then
   output("Cannot open file assign.list!");
endif;
```

```
print LaTeX–document header;
print headline;
print name of database and current date;
for (each SK in the database) do
        print name, version;
        for (each assigned topic) do
            print topicname;
        enddo;
enddo;
print LaTeX–document closing;
close file assign.list;
```

Auch der Algorithmus 5.15 verfährt im wesentlichen analog zum Algorithmus 5.13. Die Daten für die Sprachkonstrukte werden aus der Relationen `skversion` entnommen, die Namen der zugeordneten Topics ergeben sich aus den Relationen `topic` und `sktopic`x (x wird dabei wiederum durch die numerische Codierung des Spezifikationstyps des jeweils betrachteten Sprachkonstruktes ersetzt).

Die LaTeX–Dateien, welche durch die drei Algorithmen 5.13, 5.14 und 5.15 erzeugt werden, können ohne Nachbearbeitung an den LaTeX–Übersetzer weitergegeben werden, falls die Schriftart `cmssbx10` in den Skalierungen 1000 und 1440 sowie die Stil–Datei `a4.sty` verfügbar sind. Beispielausdrucke finden sich auf den Seiten 116 und 117 dieser Arbeit.

6 Erweiterungen zu NoMIS

Im Rahmen dieser Diplomarbeit wurde eine gewisse „Grundfunktionalität“ des Systems implementiert. Es sind durchaus Erweiterungen denkbar, die dem Benutzer weitere Vorteile verschaffen könnten. Dazu zählen auch die folgenden Vorschläge.

- Die Benutzeroberfläche des NoName–Editors ist optisch nicht auf dem aktuellen Stand. Die Verwendung anderer Fensterelemente wäre sicherlich von Vorteil. Die Firma GrammaTech hat dieses Problem auch erkannt und will in der Version 5 des Synthesizer Generators eine andere Oberfläche anbieten, so daß sich die Erweiterung in dieser Hinsicht möglicherweise relativ einfach bewerkstelligen läßt.

- Der Anschluß an andere Datenbanksysteme, z.B. Oracle, wäre wünschenswert. In der vorliegenden Arbeit wurde kein Interface zu Oracle entwickelt, da dieses System nur auf einem anderen Betriebssystem verfügbar war und somit keine Testumgebung vorhanden war.

- Die Verwaltung der unterschiedlichen Versionen eines Sprachkonstruktes könnte eventuell verbessert werden. Es ist z.B. denkbar, daß immer nur die neueste Version eines Sprachkonstruktes importiert werden darf, die älteren Versionen aber dennoch benötigt werden für Programme, die ebenfalls schon älter sind. Ein derartiger zeitlicher Aspekt wurde in der vorliegenden Arbeit nicht berücksichtigt.

- Die Suche nach einem Sprachkonstrukt in der Datenbank könnte erweitert werden. Mit der vorliegenden Version des Informationssystems können Sprachkonstrukte über die zugeordneten Kriterien gefunden werden. Sinnvoll wäre eventuell, auch die Suche nach Bezeichnern in den Sprachkonstrukten zu ermöglichen. Damit könnten z.B. Sprachkonstrukte aufgefunden werden, die eine bestimmte Funktion verwenden. Allerdings zählt die effiziente textuelle Suche in Datenbanken zu den problematischeren Aufgaben und ist daher nicht einfach zu implementieren.

- Zur Zeit wird auch an einem Übersetzer für die Sprache NoName gearbeitet. Wenn dieser fertiggestellt ist, dann könnte man das ER–Modell dahingehend erweitern, daß auch übersetzter Code in der Datenbank abgelegt werden kann. Die Modellierung könnte analog zur 1:n–Beziehung zwischen den Entities `SK-Version` und `NNM-Code` erfolgen.

Teil II

Bedienungsanleitung

Dieser Teil ist als Bedienungsanleitung gedacht, die unabhängig vom Teil I der Diplomarbeit gelesen werden kann. Daher sind auch Informationen enthalten, die dem Leser des ersten Teiles bereits bekannt sind.

7 Installation des NoName–Editors

Um den NoName–Editor installieren zu können, benötigten Sie das UNIX–Programm `gunzip` zum Entpacken der Dateien auf Ihren Installationsdisketten. Für die spätere Verwendung des NoName–Modul–Informationssystems NoMIS wird eine Installation des relationalen Datenbanksystems Ingres benötigt. Der Ingres–Systemadministrator muß die Relationen, deren Definitionen sich in einer Datei auf den Installationsdisketten befinden, in einer Ingres–Datenbank anlegen.

Das NoName–Editor–System befindet sich auf zwei Installationsdisketten und besteht aus den folgenden Dateien:

- **nnm.gz**
  Dies ist das gepackte Shell–Skript, das den Editor mit den korrekten Parametern startet.

- **NoName.style.gz**
  Dies ist die gepackte Datei mit den Startparametern für das Shell–Skript. Die (entpackte) Datei NoName.style können Sie auch selbst verändern (siehe hierzu auch [EGH+94b]).

- **NoName.syngen_resources.gz**
  Dies ist die gepackte Datei mit Parametern zur Tastatur–Belegung etc. Im Normalfall sollten Sie hier keine Änderungen vornehmen müssen.

- **NoName_1.gz**
  Dies ist Teil 1 des Editors in gepackter Form.

- **NoName_2.gz**
  Dies ist Teil 2 des Editors in gepackter Form.

- **nnm_install**
  Dies ist ein Skript, mit welchem die Installation des Editors einfach durchgeführt werden kann.

- **nnm_create.tables.gz**
  In dieser Datei sind die Definitionen für die benötigten Relationen in gepackter Form enthalten.

Alle Dateien mit der Endung „.gz“ liegen in gepackter Form vor. Um den Editor benützen zu können, müssen die gepackten Dateien entpackt und in einem Verzeichnis, dem sogenannten Installationsverzeichnis, abgelegt werden. Kopieren

Sie dazu alle Dateien der beiden Disketten in Ihr Installationsverzeichnis und rufen Sie dann das Installationsskript auf wie folgt:

```
nnm_install
```

Wenn das UNIX–Programm `gunzip` vorhanden und in Ihrem Pfad enthalten ist, dann wird durch das Installationsskript der ausführbare Editor erzeugt. Andernfalls müssen Sie das Programm `gunzip` von Hand aufrufen und anschließend den Editor erzeugen:

```
<pfad>/gunzip *.gz
cat NoName_1 NoName_2 > NoName
chmod 755 NoName
chmod 755 nnm
\rm NoName_1
\rm NoName_2
```

Sie müssen `<pfad>` dabei durch den Namen des Pfades ersetzen, unter welchem das Programm `gunzip` abgelegt ist.

Damit ist die Installation des Editors bereits abgeschlossen.

Hinweise für den Ingres–Systemadministrator

Das Anlegen der benötigten Relationen kann mit dem Ingres–Tool `isql` erfolgen. Nach dem Starten dieses Tools muß lediglich die Datei `nnm_create.tables` geladen und die darin enthaltenen SQL–Anweisungen ausgeführt werden. Anschließend müssen alle Benutzer des NoName–Modul–Informationssystems zumindest SELECT–Rechte auf allen Relationen sowie INSERT– und DELETE–Rechte auf der Relation `toptemp` erhalten. Damit können die Benutzer alle Daten lesen, Einfügungen und Löschoperationen sind aber nicht gestattet. Die Benutzer, welche auch Einfüge– und Löschoperationen in der Datenbank vornehmen sollen, benötigen auch auf allen anderen Relationen INSERT–, DELETE– und UPDATE–Rechte.

Weitere administrative Tätigkeiten sind nicht erforderlich.

8 Der NoName–Editor

8.1 NoMIS — Eine Erweiterung des NoName–Editors

Das NoName–Modul–Informationssystem, kurz NoMIS, ist eine Datenbankerweiterung des NoName–Editors. Dieser Editor wurde für die funktionale Programmiersprache NoName entwickelt. Diese Sprache ist in [EGH+94a], der syntaxgesteuerte Editor dazu in [EGH+94b] ausführlich beschrieben. Daher werden im folgenden nur die Besonderheiten im Umgang mit der Erweiterung NoMIS beschrieben.

Die Aufgabe des NoName–Modul–Informationssystems besteht in der Verwaltung der NoName–Sprachkonstrukte. Ein NoName–Sprachkonstrukt ist entweder eine Typspezifikation, die Spezifikation eines algebraischen Datentyps, eine Funktionsspezifikation oder die Spezifikation eines tree transducers. Diese vier „Spezifikationstypen“ werden in NoMIS *classifications* genannt. Die Import–Spezifikationen werden nicht unterstützt, da diese bei der Verwendung des Modul–Informationssystems nicht benötigt werden. Der erweiterte NoName–Editor bietet die Möglichkeit, Sprachkonstrukte in der Datenbank abzulegen, mit Suchkriterien zu versehen, gespeicherte Sprachkonstrukte mit geringem Aufwand in das aktuelle NoName–Programm aufzunehmen und diverse Verwaltungen einfach durchzuführen. Außerdem können verschiedene Listen in Form von LaTeX–Dateien erzeugt werden. Im Kapitel 9 werden die Vorgehensweisen anhand eines Beispiels gezeigt, auch sind in diesem Abschnitt derartige LaTeX–Dokumente abgebildet.

Die speicherbaren Daten sind folgendermaßen organisiert:

Sprachkonstrukte haben einen Namen (max. 30 Zeichen) und eine Version (max. 8 Zeichen). Diese sind vom Benutzer einzugeben und müssen zusammen eindeutig ein Sprachkonstrukt bestimmen. Optional kann ein Kommentar von maximal 50 Zeichen eingegeben werden, um ein Sprachkonstrukt näher beschreiben zu können. Das Speicherungsdatum und der aktuelle Benutzername werden automatisch in der Datenbank vermerkt.

Die Suchkriterien, im folgenden *Topics* genannt, besitzen einen eindeutigen Namen (max. 20 Zeichen) sowie optional einen Kommentar (max. 50 Zeichen). Topics können beliebig vielen Sprachkonstrukten zugeordnet werden, ebenso können einem Sprachkonstrukt beliebig viele Topics zugeordnet werden. Abbildung 8.1 zeigt an einem Beispiel mit drei Sprachkonstrukten und vier Topics, welches Netz dadurch entstehen kann. Dabei repräsentieren die Pfeile Zuordnungen aus der Sicht der Sprachkonstrukte. Aus der Sicht der Topics sind diese Pfeile lediglich in umgekehrter Richtung zu betrachten.

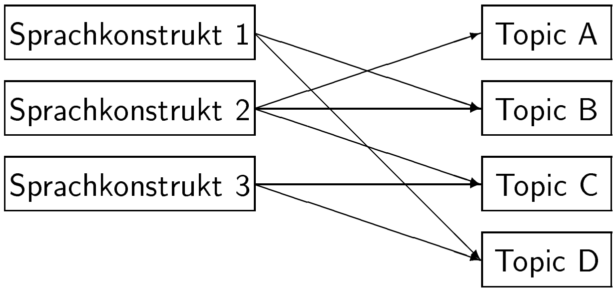


Abb. 8.1: Mehrere Zuordnungen zwischen Sprachkonstrukten und Topics

8.2 Die Verwendung des Editors

Die Verwendung des NoName–Editors ist in [EGH⁺94b] beschrieben. Dort sind auch die Bedeutungen der Menüpunkte, die Verwendung der Tastatur, die Anzeige verschiedener Sichten sowie die Bedeutung der Transform–Buttons erklärt. Neu im Zusammenhang mit NoMIS ist lediglich das Menü `NoMIS`, das im folgenden vorgestellt wird.

In dem Menü `NoMIS` sind alle Menüpunkte enthalten, welche der Verwaltung von Sprachkonstrukten dienen.

NoMIS
db-retrieve db-store db-delete
topic-handling
db-output
db-connect db-disconnect db-exit-all

Die Menüpunkte sind so angeordnet, daß häufig benötigte Aktionen im oberen Teil des Menüs und weniger oft gewählte Aktionen im unteren Bereich zu finden sind. So ist der Punkt db-connect (Verbindung zur Datenbank herstellen) relativ weit unten angeordnet, da er in der Regel nur einmal pro Editorsitzung benötigt wird. Das Speichern eines Sprachkonstruktes dagegen wird häufiger gewünscht werden, deshalb findet sich der entsprechende Menüpunkt db-store weiter oben. Der besseren Übersichtlichkeit wegen sind zusammengehörende Menüpunkte durch horizontale Trennstriche optisch gruppiert.

Im folgenden sind alle Punkte dieses Menüs erläutert. Angegeben sind jeweils die Funktion sowie die Voraussetzungen, die erfüllt sein müssen, um den Menüpunkt auswählen zu können.

Menüpunkt: **db-retrieve**
Funktion: Suchen eines Sprachkonstruktes und Übernahme desselben in das aktuell editierte NoName–Programm. Die Suche kann direkt durch die Angabe des Namens und der Version eines Sprachkonstruktes oder durch die Angabe von Topics angestoßen werden.
Voraussetzung:
- Existenz einer Verbindung zur Datenbank.
- Markierung einer „leeren" Spezifikation im Editor.

Menüpunkt: **db-store**
Funktion: Speichern eines NoName–Sprachkonstruktes in der Datenbank. Es können der Name, die Version und ein Kommentar eingegeben werden. Außerdem können Topics als Suchkriterien zugeordnet werden, damit später ein leichteres Auffinden ermöglicht wird. Vom System werden automatisch noch der Name des Benutzers, das aktuelle Datum sowie der Typ des Sprachkonstruktes (Funktionsspezifikation oder Typspezifikation etc.) ermittelt und abgespeichert. Zu beachten ist, daß das zum Zeitpunkt der Menüpunkt–Auswahl markierte Sprachkonstrukt in der Datenbank abgelegt wird. Eine Änderung der Markierung ist nachträglich nicht möglich.
Voraussetzung:
- Existenz einer Verbindung zur Datenbank.
- Die aktuelle Markierung im Editor muß eine Spezifikation sein, die sich korrekt parsen läßt und keine Fehler bezüglich der statischen Semantik aufweist.

Menüpunkt: **db-delete**
Funktion: Löschen eines Sprachkonstruktes aus der Datenbank. Die Suche nach dem zu löschenden Sprachkonstrukt kann durch die Angabe des Namens und der Version oder durch Angabe von Topics angestoßen werden. Letzteres ermöglicht die Auswahl des gewünschten Sprachkonstruktes aus einer Liste von Sprachkonstrukten, die alle mindestens einem angegebenen Topic zugeordnet sind. Das Löschen eines Sprachkonstruktes kann nicht durch einen einfachen „Knopfdruck" rückgängig gemacht werden! Zu beachten ist auch, daß ein Sprachkonstrukt nur von dem Benutzer gelöscht werden kann, der dieses auch in der Datenbank abgelegt hat.
Voraussetzung:
- Existenz einer Verbindung zur Datenbank.

Menüpunkt:	**topic-handling**
Funktion:	Starten der Verwaltung von Topics. Es wird ein Fenster geöffnet, in welchem der Benutzer die Wahl trifft zwischen *Topics anzeigen* (browse), *Topic neu anlegen* (new) und *Topic löschen* (delete). Wählt der Benutzer den Punkt browse, so erhält er eine Liste aller gespeicherten Topics. Durch Anklicken eines Topics in der Liste wird der zugehörige Kommentar angezeigt. Die Wahl des Punktes new ermöglicht die Erstellung eines neuen Topics. Dazu muß ein eindeutiger Name und optional eine kurze Beschreibung eingegeben werden. Mit dem Punkt delete kann ein gespeichertes Topic aus der Datenbank entfernt werden. Dabei ist zu beachten, daß Topics, die mindestens einem Sprachkonstrukt zugeordnet sind, nicht gelöscht werden können.
Voraussetzung:	• Existenz einer Verbindung zur Datenbank.

Menüpunkt:	**db-output**
Funktion:	Der Benutzer kann mehrere LaTeX–Dateien erzeugen lassen, in welchen Übersichten über gespeicherte Daten in der Datenbank enthalten sind. Die Auswahl einer Liste erfolgt dabei in einem Fenster, das durch diesen Menüpunkt geöffnet wird. Es stehen folgende Listen zur Verfügung: *Übersicht aller gespeicherten Sprachkonstrukte* (erzeugt `constructs.list`), *Übersicht aller gespeicherten Topics* (erzeugt `topic.list`) und *Zuordnungen zwischen Sprachkonstrukten und Topics* (erzeugt `assign.list`). Je nach Anzahl der in der Datenbank gespeicherten Sprachkonstrukte und Topics benötigt die Ausführung dieser Punkte relativ viel Zeit (bis hin zu einigen Minuten), weshalb die Auswahl dieser Punkte nur gezielt erfolgen sollte.
Voraussetzung:	• Existenz einer Verbindung zur Datenbank.

Menüpunkt:	**db-connect**
Funktion:	Herstellen einer Verbindung zwischen dem NoName–Editor und der Datenbank. Der Benutzer kann in einem Fenster den Namen der Datenbank angeben, wobei defaultmäßig der Inhalt der UNIX–Umgebungsvariablen `NOMISDEFAULTDB` angezeigt wird.
Voraussetzung:	• Keine Verbindung zur Datenbank existent.

Menüpunkt:	**db-disconnect**
Funktion:	Abbrechen einer Verbindung zwischen dem NoName–Editor und der Datenbank.
Voraussetzung:	• Existenz einer Verbindung zur Datenbank.

Menüpunkt:	**db-exit-all**
Funktion:	Abbrechen einer Verbindung zwischen dem NoName–Editor und der Datenbank mit anschließendem Verlassen des Editors. Dabei werden eventuell geänderte NoName–Programme nicht gesichert! Dieser Menüpunkt ist nur dafür gedacht, beim „Spielen“ mit dem System den Editor schnell beenden zu können. Wenn der Editor allerdings produktiv eingesetzt wird, dann sollte mit dem Menüpunkt db-disconnect die Datenbankverbindung abgebrochen und anschließend mit dem Menüpunkt exit aus dem Menü File der Editor beendet werden.
Voraussetzung:	• Existenz einer Verbindung zur Datenbank.

9 Beispielsitzung mit NoMIS

In den folgenden Abschnitten wird der Umgang mit NoMIS Schritt für Schritt erklärt. Wenn die einzelnen Abschnitte der Reihe nach gelesen und nachvollzogen werden, dann sollte der Benutzer im Anschluß daran in der Lage sein, ohne weitere Anleitung mit NoMIS zu arbeiten. Sollten Unklarheiten in der Bedienung des NoName–Editors auftreten, so kann das Handbuch zum Editor weiterhelfen (siehe [EGH+94b]). Die in den folgenden Abschnitten durchgeführten Schritte gehen von einer anfangs „leeren“ Datenbank aus, in welcher nach und nach Daten abgelegt werden. Sollten in Ihrer Datenbank bereits Datensätze vorhanden sein, so weichen die Abbildungen der Bildschirmmasken teilweise etwas von Ihrer tatsächlichen Bildschirmdarstellung ab.

Wir werden zunächst ein paar Topics kreieren und dabei die gesamte Topic–Verwaltung kennenlernen. Anschließend werden wir die Speicherung eines neuen Sprachkonstruktes betrachten. Dann werden wir ein gespeichertes Sprachkonstrukt aus der Datenbank in den NoName–Editor importieren und das gespeicherte Sprachkonstrukt aus der Datenbank entfernen. Schließlich betrachten wir noch die Erzeugung der LaTeX–Dateien, welche die verschiedenen Übersichten enthalten. Zuletzt finden sich noch Hinweise, die beim Beenden des Editors beachtet werden sollten.

9.1 Starten des NoName–Editors

Der NoName–Editor wird wie gewöhnlich mit dem Kommando

```
nnm
```

in einem XTerm– oder CommandTool–Fenster gestartet. Daraufhin wird das Hauptfenster geöffnet, von dem aus alle weiteren Aktionen ausgehen. In Abbildung 9.1 ist dieses Fenster abgebildet, die Fensterbereiche sind darin kurz erklärt.

In der Menüzeile befindet sich das Menü `NoMIS`, dessen Punkte in Abschnitt 8.2 erläutert sind. In den folgenden Abschnitten werden wir uns mit der Verwendung dieses Menüs detailliert beschäftigen.

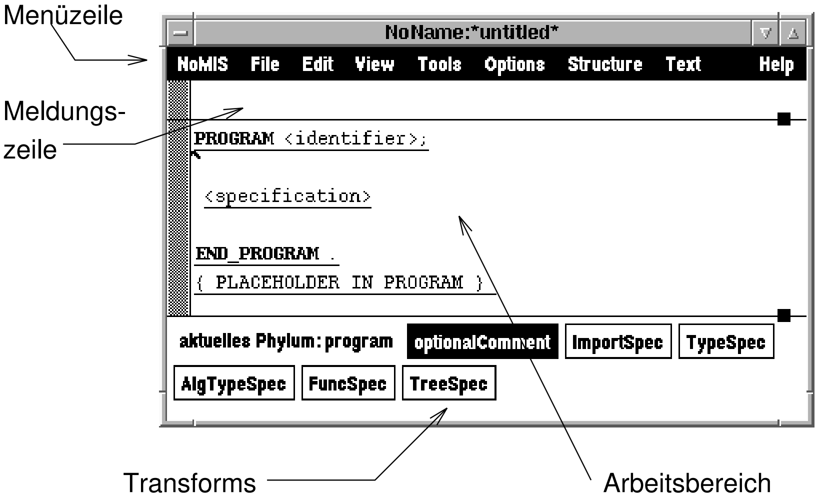


Abb. 9.1: Hauptfenster des NoName–Editors

9.2 Datenbankanschluß herstellen

Um mit der Datenbank arbeiten zu können, benötigen wir zunächst eine Verbindung zwischen dem NoName–Editor und der Datenbank. Dazu wählen wir den Punkt db-connect aus dem Menü `NoMIS` aus. Wir erhalten ein Fenster, wie es in Abbildung 9.2 dargestellt ist.

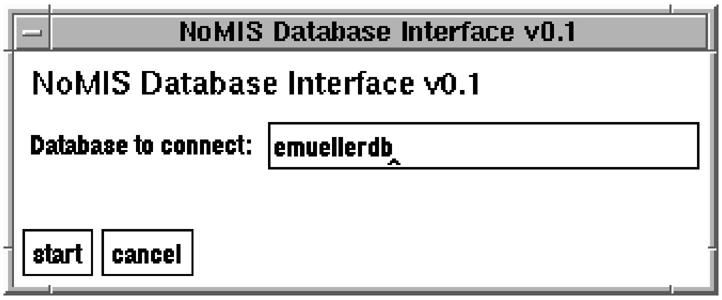


Abb. 9.2: Eingabe des Datenbanknamens

Der im Fenster angezeigte Datenbankname — in der Abbildung ist dies der Name `emuellerdb` — entspricht dem Wert der UNIX–Umgebungsvariablen `NOMISDEFAULTDB`. Wenn Ihnen der bei Ihnen angezeigte Name nicht zusagt, dann können Sie dieser Umgebungsvariablen einen anderen Namen zuweisen oder einfach in diesem Fenster einen anderen Namen eingeben.

Die Betätigung des Buttons `start` bewirkt die Herstellung einer Verbindung zur angegebenen Datenbank. Dieser Vorgang kann einige Sekunden dauern, falls der Datenbankserver ein anderer Rechner im Netzwerk ist als der von

Ihnen benutzte Rechner. Bei erfolgreichem Aufbau der Verbindung kehren Sie zum Fenster aus Abbildung 9.1 zurück, in der Meldungszeile erhalten Sie den Hinweis, daß Sie an die Datenbank angeschlossen sind.

9.3 Die Benutzung der Topic–Verwaltung

Die Topic–Verwaltung enthält drei Teilaufgaben:

1. Topics anzeigen
2. Neues Topic anlegen
3. Topic löschen

Wenn Sie den Menüpunkt topic-handling auswählen, dann sehen Sie ein Fenster ähnlich Abbildung 9.3.

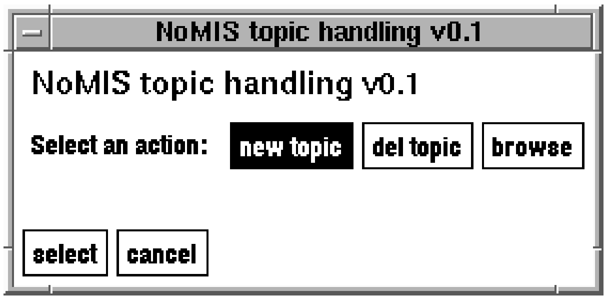


Abb. 9.3: Auswahl einer Aktion der Topic–Verwaltung

Wir möchten zunächst ein neues Topic anlegen. Dazu wählen wir aus den drei angebotenen Möglichkeiten den Schalter mit der Beschriftung `new topic` durch einfaches Anklicken mit der Maus aus. Wir bestätigen die Auswahl durch Betätigung des Buttons `select` und erhalten damit das Fenster aus Abbildung 9.4.

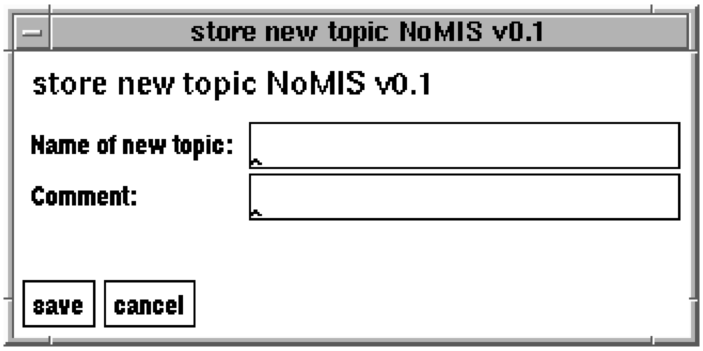


Abb. 9.4: Fenster zum Anlegen eines neuen Topics

Nun müssen wir einen Namen für das neue Topic vergeben. Dazu bewegen wir den Mauszeiger in das Dialogelement rechts neben dem Text `Name of new topic:` und schreiben z.B. `Arithmetik`. Nun versehen wir das neue Topic noch mit einem Kommentar, indem wir den Mauszeiger in das Feld rechts neben dem Text `Comment:` bewegen und schreiben beispielsweise `einfache arithmetische Funktionen etc.`. Sodann betätigen wir den Button mit der Bezeichnung `save`. Das Fenster schließt automatisch, Sie befinden sich wieder im Fenster von Abbildung 9.3. Wir wiederholen das Anlegen eines neuen Topics mit den Topicnamen `System`, `Trigonometrie` und `Alles` sowie beliebigen Kommentaren.

Nach der Erzeugung neuer Topics wollen wir uns nun die gespeicherten Topics anzeigen lassen. Wir wählen dazu in dem Fenster aus Abbildung 9.3 den Schalter mit der Beschriftung `browse` und erhalten dadurch ein Fenster entsprechend Abbildung 9.5.

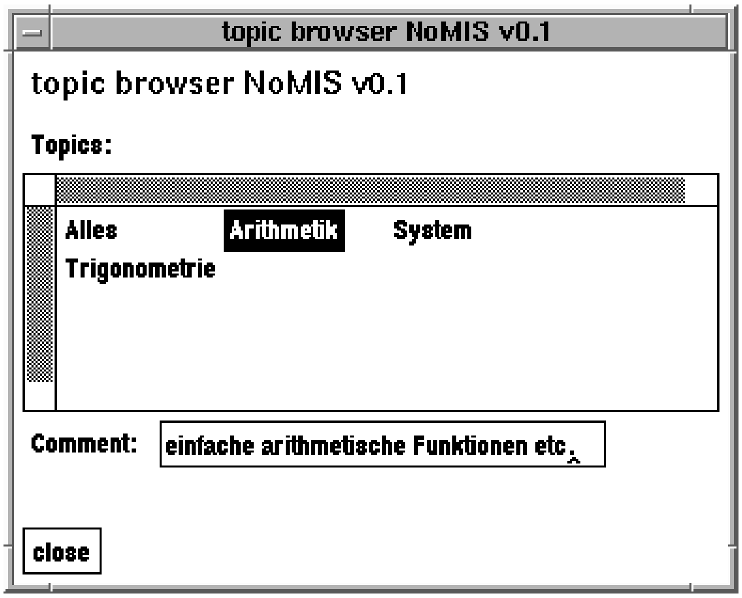


Abb. 9.5: Anzeige aller gespeicherten Topics

Der größte Teil des Fensters wird von der Liste eingenommen, welche die Namen aller gespeicherten Topics alphabetisch sortiert anzeigt. Wenn mehr Topics gespeichert sind, als in dieser Liste Platz haben, dann werden an den Listenrändern sogenannte *Scrollbars* erzeugt, durch welche der sichtbare Listenausschnitt mit der Maus verschoben werden kann. Durch einfaches Anklicken eines Topicnamens mit der Maus wird der zugehörige Kommentar in dem Fensterelement unterhalb der Liste angezeigt. In Abbildung 9.5 wurde bereits auf den Topicnamen `Arithmetik` geklickt.

Durch die Betätigung des Buttons `close` wird dieses Fenster wieder geschlossen, Sie kehren zum Fenster aus Abbildung 9.3 zurück.

Als letzter Punkt der Topic–Verwaltung bleibt nun noch das Löschen eines Topics. Dazu wählen wir im Fenster gemäß Abbildung 9.3 den Schalter `del topic`. Nach der Bestätigung durch Drücken des Buttons `select` erhalten wir ein Fenster entsprechend der Abbildung 9.6.

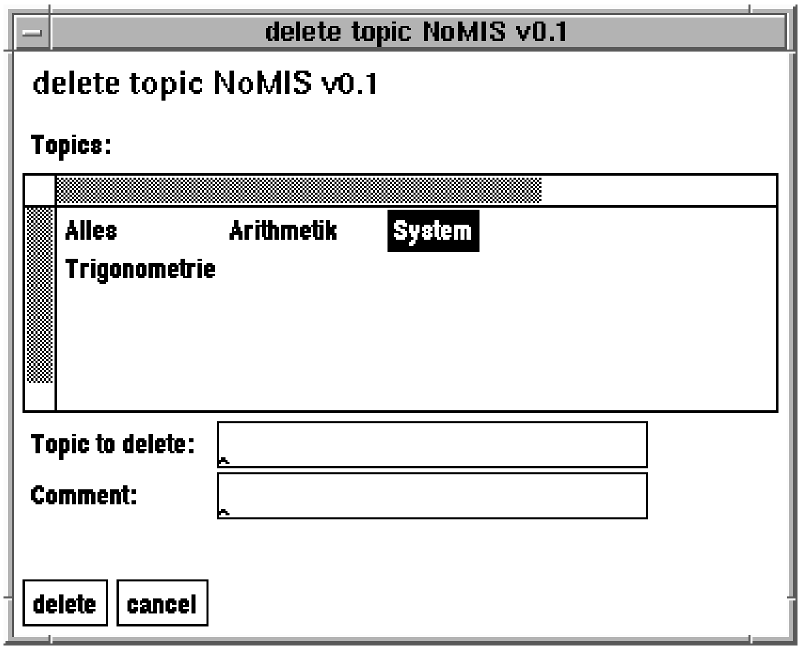


Abb. 9.6: Löschen eines Topics

Wie bei der Anzeige aller gespeicherten Topics erhalten wir auch hier eine Liste mit den Namen aller verfügbaren Topics angezeigt. Daraus wählen wir nun den Namen `Alles` durch einfachen Mausklick aus. Wenn wir anschließend den Button `delete` betätigen, wird das Topic aus der Datenbank gelöscht. Dabei wird geprüft, ob ein Sprachkonstrukt existiert, welches diesem Topic zugeordnet ist. Trifft dies zu, so erhalten Sie eine entsprechende Fehlermeldung, das Topic kann dann nicht gelöscht werden. In unserem Fall trifft dieses nicht zu, daher wird das Topic mit dem Namen `Alles` problemlos aus der Datenbank entfernt. Mit der Anzeige aller gespeicherten Topics können Sie dies verifizieren!

Nach dem erfolgreichen Löschen des Topics kehren Sie wiederum zum Fenster aus Abbildung 9.3 zurück. Dieses Fenster können Sie durch Drücken des Buttons `cancel` verlassen, Sie befinden sich dann wieder im Fenster aus Abbildung 9.1.

Damit haben wir jetzt alle Aufgaben der Topic–Verwaltung kennengelernt. Im folgenden beschäftigen wir uns mit der Verwaltung der Sprachkonstrukte.

9.4 Speicherung eines neuen Sprachkonstruktes

In diesem Abschnitt wollen wir ein neues Sprachkonstrukt in der Datenbank ablegen. Dazu schreiben wir uns zuerst ein NoName–Programm, das in Abbildung 9.7 zu sehen ist.

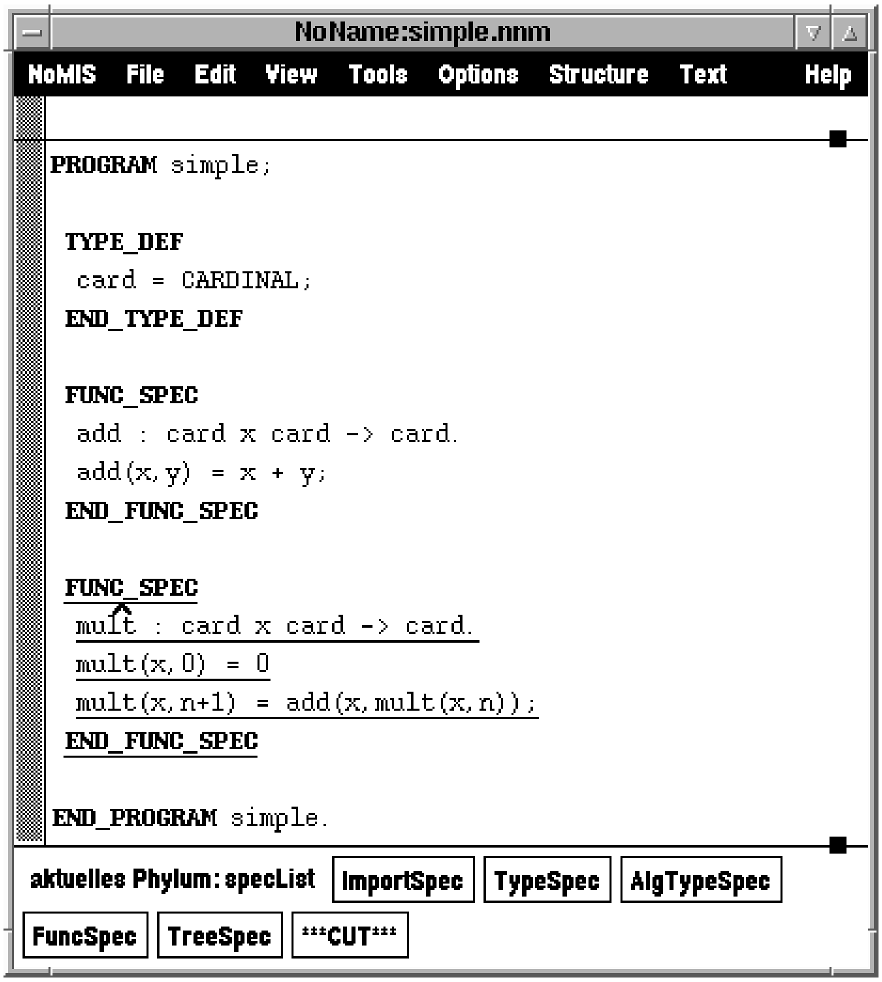


Abb. 9.7: Beispielprogramm zur Speicherung eines neuen Sprachkonstruktes

Wir möchten nun die Funktion `mult`, die den Datentyp `card` und die Funktion `add` benötigt, in der Datenbank ablegen. Dazu markieren wir mit der Maus die Funktions–Spezifikation, welche die Funktion `mult` enthält. Wenn diese Funktions–Spezifikation fehlerfrei geparst werden kann und auch keine Fehler bezüglich der statischen Semantik enthält, dann können wir den Menüpunkt db-store aus dem Menü `NoMIS` auswählen. Wir erhalten dadurch ein Fenster, wie es in Abbildung 9.8 zu sehen ist.

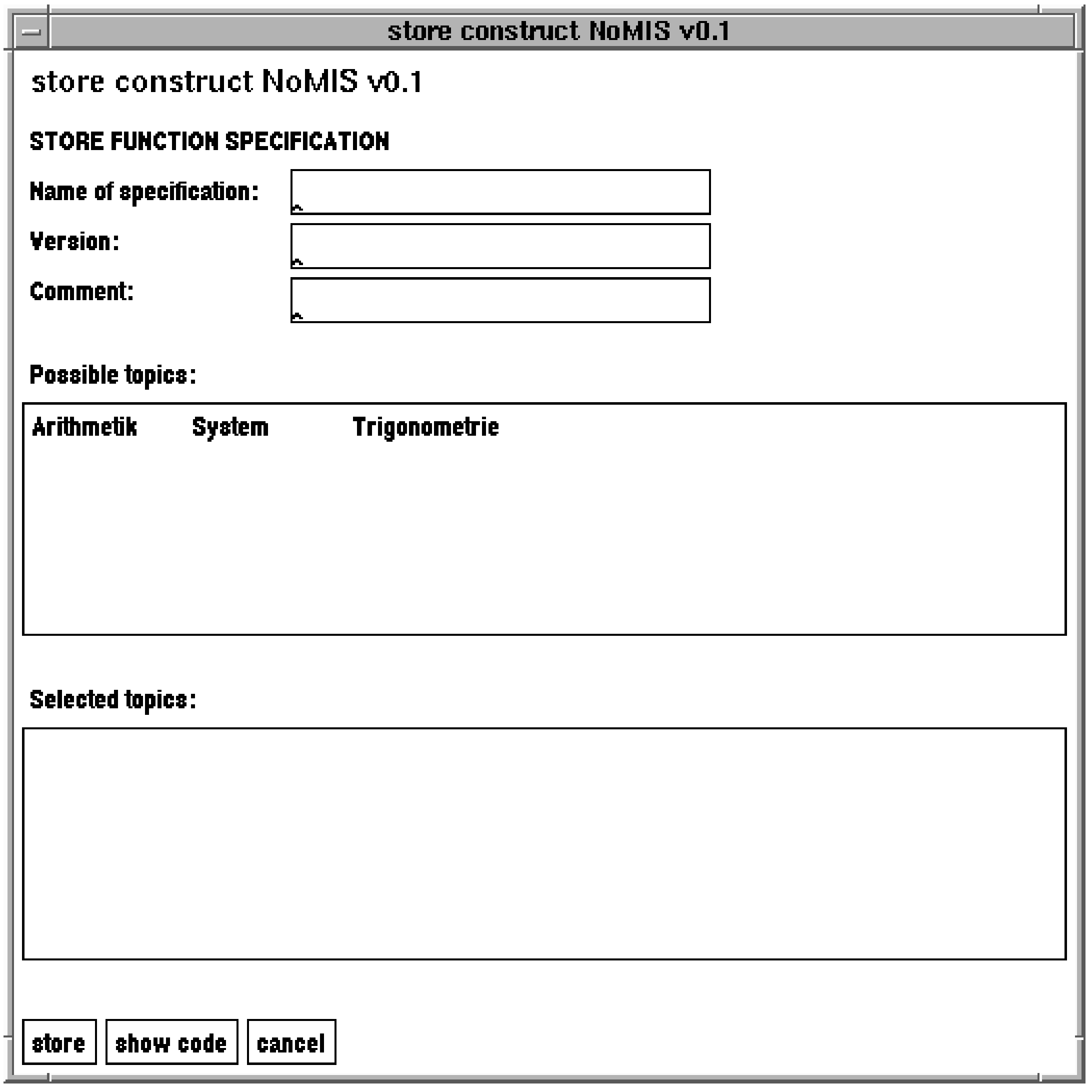


Abb. 9.8: Fenster zur Dateneingabe bei der Speicherung eines neuen Sprachkonstruktes

Im oberen Teil des Fensters stehen nun zwei Felder zur Eingabe des Namens und der Version zur Verfügung. Wir geben dem neuen Sprachkonstrukt den Namen `mult` in der Version `1a`. In dem Feld unmittelbar darunter können Sie bei Bedarf noch einen Kommentar für das neue Sprachkonstrukt eingeben.

Den größten Teil des Fensters nehmen zwei Listen ein, die der Zuordnung des neuen Sprachkonstruktes zu bestehenden Topics dienen. In der oberen Liste sind alle gespeicherten Topics alphabetisch sortiert aufgeführt. Die untere Liste ist nach dem Öffnen des Fensters zunächst noch leer. Wenn wir nun aus der oberen Liste ein Topic mittels Mausklick auswählen, d.h. wir ordnen dieses dem neuen Sprachkonstrukt zu, dann wird der gewählte Namen in die untere Liste eingefügt. Somit sind in der unteren Liste alle bereits zugeordneten Topics zu sehen. Wenn Sie ein Topic versehentlich zugeordnet haben, dann können Sie dieses wieder zurücknehmen, indem Sie in der unteren Liste auf dessen Namen klicken. Dadurch wird es aus dieser Liste wieder entfernt. Wir wollen unser neues Sprachkonstrukt dem Topic `Arithmetik` zuordnen und klicken daher auf diesen Listeneintrag.

Nun können wir durch Drücken des Buttons `store` das Sprachkonstrukt in der Datenbank ablegen. Dabei werden auch die von `mult` benötigten Sprachkonstrukte, die wir im folgenden *Hilfskonstrukte* nennen wollen, automatisch mit gespeichert. Da der Name und die Version eines Sprachkonstruktes dieses Sprachkonstrukt eindeutig identifizieren müssen, können die Hilfskonstrukte nicht unter dem gleichen Namen abgelegt werden wie unsere Funktion `mult`. Daher gilt folgende Namenskonvention:

Die zu einem Sprachkonstrukt erforderlichen Hilfskonstrukte werden fortlaufend numeriert. Um diese Hilfskonstrukte als solche zu kennzeichnen, erhalten sie einen Namen, der mit dem Buchstaben ‚`H`' beginnt, gefolgt von der fortlaufenden Nummer und dem vom Benutzer eingegebenen Namen des zu speichernden Sprachkonstruktes. In unserem Beispielprogramm erhält die Typ–Spezifikation den Namen `H1mult` und die Funktions–Spezifikation für `add` den Namen `H2mult`. Als Version und Kommentar werden die vom Benutzer eingegebenen Werte übernommen.

Aufgrund dieser Namenskonvention ist dringend zu empfehlen, selbst keine Namen zu vergeben, die mit dem Buchstaben ‚`H`' gefolgt von einer Nummer beginnen!

Bei der Importierung eines Sprachkonstruktes aus der Datenbank werden wir sehen, daß die Speicherung aller benötigten Hilfskonstrukte eine große Erleichterung für den Programmierer darstellt und daher äußerst sinnvoll ist.

Sollten Sie sich nicht sicher sein, ob Sie tatsächlich das gewünschte Sprachkonstrukt im NoName–Editor markiert haben, dann können Sie sich durch Drücken des Buttons `show code` den NoName–Programmcode des aktuell markierten Sprachkonstruktes in einem separaten Fenster anzeigen lassen. Dieses Fenster verlassen Sie durch Betätigen des Buttons `ok`.

Wenn Sie die Speicherung eines neuen Sprachkonstruktes abbrechen wollen, dann betätigen Sie in dem Fenster aus Abbildung 9.8 den Button `cancel`, Sie gelangen dadurch ebenso wie nach dem Betätigen des Buttons `store` in das Fenster aus Abbildung 9.1.

9.5 Importieren eines gespeicherten Sprachkonstruktes

Wir möchten nun das im Abschnitt 9.4 in der Datenbank abgelegte Sprachkonstrukt wieder importieren. Dazu löschen wir alle Sprachkonstrukte im NoName–Editor und markieren eine „leere“ Spezifikation, wie es in Abbildung 9.9 zu sehen ist.

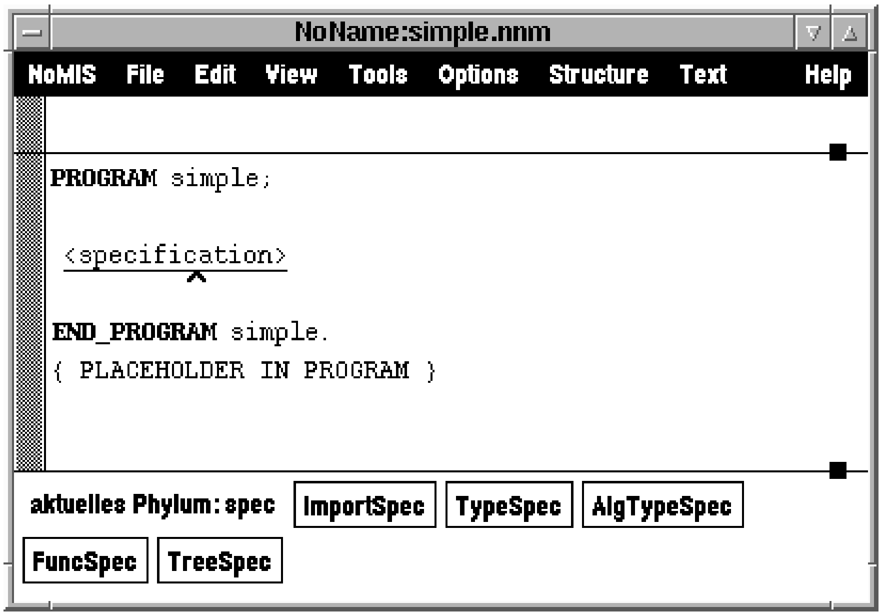


Abb. 9.9: Markieren einer „leeren“ Spezifikation als Vorbereitung für den Import eines Sprachkonstruktes

Warum ist das Löschen aller Sprachkonstrukte im NoName–Editor nötig? Ganz einfach, NoMIS importiert nur solche Sprachkonstrukte aus der Datenbank, die noch nicht im Editor vorhanden sind. Das heißt, wenn Sie nur die Typ–Spezifikation aus dem NoName–Editor entfernen und die Funktions–Spezifikation `mult` aus der Datenbank importieren, dann wird nur die Typ–Spezifikation eingefügt, denn die anderen beiden Spezifikationen befinden sich ja noch immer im Editor. Leider hat die Erkennung bereits importierter oder noch vorhandener Sprachkonstrukte einen kleinen Haken. Wenn Sie die Funktions–Spezifikation `mult` nach der bekannten Vorgehensweise in der Datenbank abgelegt haben und anschließend eine Änderung des Quellcodes in dieser Funktion vornehmen, dann glaubt NoMIS bei einem darauffolgenden Import noch immer, daß das Sprachkonstrukt nicht eingefügt werden muß. Wir löschen also alle Sprachkonstrukte aus dem NoName–Editor und wollen nun die Funktion `mult` aus der Datenbank importieren.

Wir gehen nun von dem Fenster aus Abbildung 9.9 aus. Durch die Wahl des Menüpunkts db-retrieve aus dem Menü `NoMIS` wird ein Fenster geöffnet, das dem in Abbildung 9.10 entspricht.

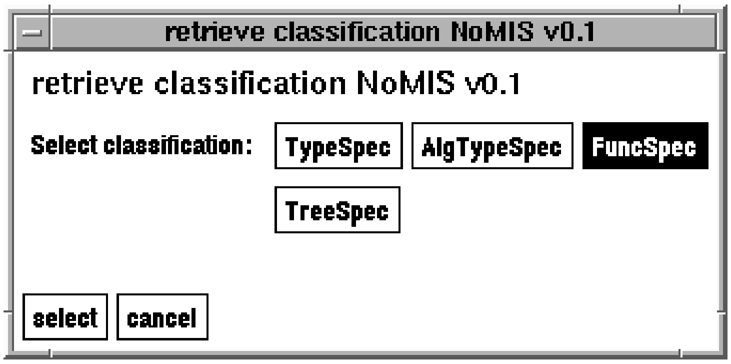


Abb. 9.10: Auswahl eines Spezifikationstyps

In diesem Fenster muß NoMIS mitgeteilt werden, welcher Spezifikationstyp importiert werden soll. Wir können wählen zwischen Typ–Spezifikationen, algebraischen Datentyp–Spezifikationen, Funktions–Spezifikationen und Spezifikationen von tree transducern. In unserem Fall wollen wir eine Funktion importieren und wählen daher den Schalter, der mit `FuncSpec` beschriftet ist. Durch Betätigung des Buttons `select` erhalten wir ein Fenster entsprechend Abbildung 9.11, in welchem wir die Selektion des zu importierenden Sprachkonstruktes durchführen werden[13].

Es bestehen zwei Möglichkeiten, das gewünschte Sprachkonstrukt in der Datenbank zu finden und für den Import auszuwählen. Die erste Möglichkeit besteht darin, direkt den Namen und die Version in die dafür vorgesehenen Fensterelemente in der oberen Hälfte des Fensters einzugeben. Was machen wir aber, wenn wir den Namen und/oder die Version nicht oder nicht genau kennen? Dann verwenden wir die zweite Methode. Diese besteht in der Angabe der Topics, die dem gesuchten Sprachkonstrukt zugeordnet sein könnten. Genau wie beim Speichern eines neuen Sprachkonstruktes funktioniert auch hier die Auswahl aus der Liste aller verfügbaren Topics. Durch Drücken des Buttons `scan` wird die dritte Liste in diesem Fenster mit Namen und Versionen aller Sprachkonstrukte gefüllt, die mindestens einem der angegebenen Topics zugeordnet sind. Aus dieser Liste wiederum kann durch einen einfachen Mausklick ein spezielles Sprachkonstrukt ausgewählt werden. Dessen Name, Version und Kommentar werden automatisch in die oben genannten Fensterelemente übertragen. Um sich zu vergewissern, daß Sie tatsächlich das gewünschte Sprachkonstrukt angegeben haben, können Sie auch hier den Button `show code` drücken. Dadurch wird Ihnen der gespeicherte NoName–Programmcode in einem Fenster angezeigt, das Sie durch Betätigen des Buttons `ok` schließen können. Sollten Sie ein falsches Sprachkonstrukt gewählt haben, so können Sie einfach aus der Liste aller passenden Sprachkonstrukte ein anderes anklicken, oder Sie wählen

[13] In den Abbildungen 9.11 und 9.12 sind aus Platzgründen die Listenelemente verkürzt dargestellt.

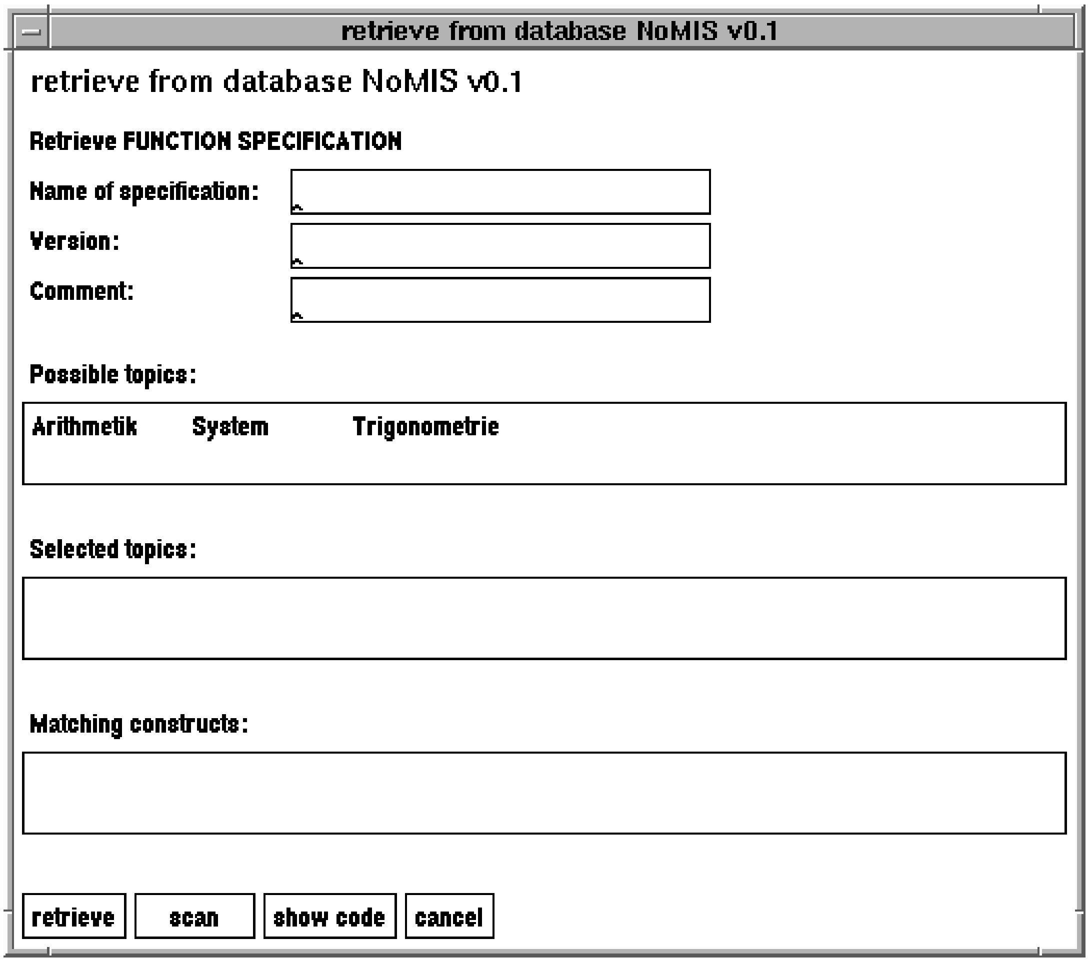


Abb. 9.11: Importieren eines Sprachkonstruktes

andere Topics und starten die Suche in der Datenbank erneut durch Drücken des Buttons `scan`. Dieses Vorgehen können Sie beliebig oft wiederholen.

In unserem Beispiel gestaltet sich die Suche noch sehr einfach, da nur ein Sprachkonstrukt vorhanden ist und dessen Name und Version sicherlich noch bekannt sind. Dennoch wollen wir die zweite Methode zur Auswahl eines Sprachkonstruktes wählen, weil dies in der Praxis der häufiger verwendete Weg sein wird. Wir können z.B. die Topics `System` und `Arithmetik` auswählen und mit dem Button `scan` die Suche in der Datenbank starten. Wie erwartet wird in der Liste aller passenden Sprachkonstrukte auch die Funktion `mult` angezeigt. Wenn wir nur das Topic `System` ausgewählt hätten, dann wäre die Liste mit den passenden Sprachkonstrukten leer geblieben. So aber können wir nun „unsere“ Funktion `mult` durch einen einfachen Mausklick auswählen, der Name, die Version und der Kommentar werden in den entsprechenden Feldern angezeigt.

Durch die Betätigung des Buttons `retrieve` wird nun der Import–Vorgang angestoßen. Jetzt wird auch der Vorteil deutlich, den die Speicherung aller Hilfskonstrukte mit sich bringt: Es wird nicht nur die Spezifikation für die Funktion `mult` in den Editor eingefügt, sondern auch alle Hilfskonstrukte, die benötigt werden. Das erspart uns das Suchen nach Hilfskonstrukten in der Datenbank!

Wenn Sie den Import eines Sprachkonstruktes durch Betätigung des Buttons `cancel` abbrechen, dann gelangen Sie ebenso wie beim Drücken des Buttons `retrieve` in das Editor–Fenster zurück. Sie können nach dem Importieren in der gewöhnlichen Art und Weise mit dem NoName–Editor weiterarbeiten.

9.6 Löschen eines Sprachkonstruktes aus der Datenbank

Das Löschen eines Sprachkonstruktes ist die dritte Möglichkeit, die NoMIS zur Verwaltung der Sprachkonstrukte anbietet. Wir wollen in diesem Abschnitt das im Abschnitt 9.4 erstellte Sprachkonstrukt aus der Datenbank löschen. Da die Löschoperation eine reine Datenbankoperation ist, d.h. es wird kein Sprachkonstrukt vom oder zum NoName–Editor übertragen, spielt die aktuelle Selektion im Editor für den Verlauf des Löschvorganges keine Rolle. Wir wählen lediglich den Punkt `db-delete` aus dem Menü `NoMIS` und geben in dem dadurch geöffneten Fenster den Spezifikationstyp an, welcher dem Typ des zu löschenden Sprachkonstruktes entspricht (siehe auch Abbildung 9.10). In unserem Beispiel ist dies wiederum eine Funktions–Spezifikation, daher wählen wir den Schalter mit der Beschriftung `FuncSpec` und bestätigen mit dem Button `select`. Damit erhalten wir ein ähnliches Fenster, wie es Abbildung 9.11 zeigt. Wir müssen dort lediglich an allen Stellen das Wort `retrieve` durch das Wort `delete` ersetzen.

Zur Auswahl des zu löschenden Sprachkonstruktes stehen uns auch hier wieder die in Abschnitt 9.5 beschriebenen zwei Möglichkeiten zur Verfügung: Entweder geben wir den Namen und die Version direkt ein, oder wir suchen uns über die zugeordneten Topics das gewünschte Sprachkonstrukt heraus. Nachdem wir das Sprachkonstrukt ausgewählt haben, sieht unser Fenster ähnlich wie das in Abbildung 9.12 gezeigte Fenster aus.

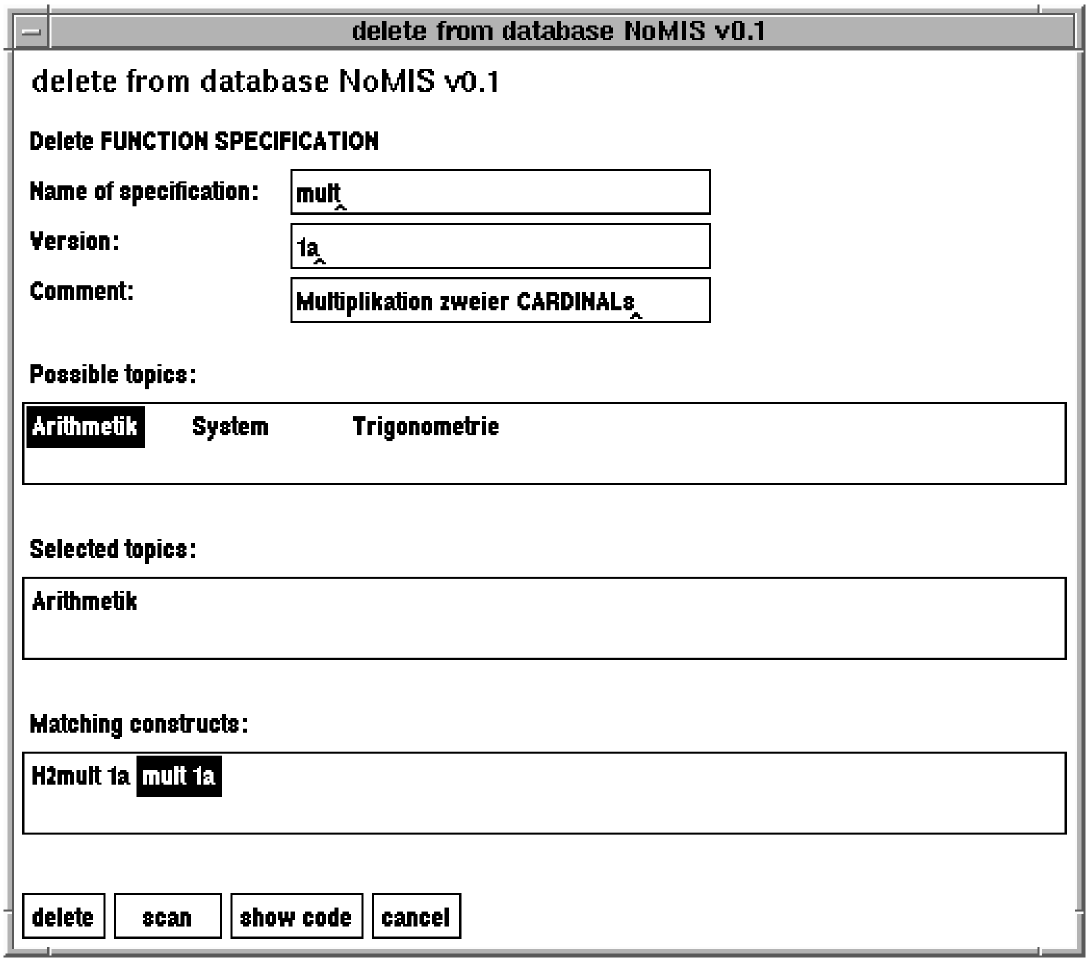


Abb. 9.12: Löschen eines gespeicherten Sprachkonstruktes

Die Betätigung des Buttons `delete` stößt den Löschvorgang des selektierten Sprachkonstruktes an. Dabei werden prinzipiell auch alle Hilfskonstrukte aus der Datenbank entfernt! Da NoMIS aber nicht entscheiden kann, ob Sie das eine oder andere Hilfskonstrukt noch anderweitig verwenden wollen, werden Sie bei jedem Hilfskonstrukt aufgefordert, das Löschen desselben zu bestätigen beziehungsweise zu unterbinden. Dabei werden zwei Arten von Hilfskonstrukten unterschieden:

1. Hilfskonstrukte, die auch noch von anderen Sprachkonstrukten benötigt werden.
2. Hilfskonstrukte, die nicht von anderen Sprachkonstrukten benötigt werden.

Beim Löschen von Hilfskonstrukten, die zur Kategorie 1 gehören, wird Ihnen die Notwendigkeit des Hilfskonstruktes für andere Sprachkonstrukte explizit mitgeteilt. Beispiele für das Aussehen der Fenster, in welchen Sie zu entscheiden haben, ob das Hilfskonstrukt gelöscht werden darf oder nicht, sind in den Abbildungen 9.13 und 9.14 zu sehen. Es werden jeweils der Name, die Version, der Programmierer und das Speicherungsdatum angezeigt.

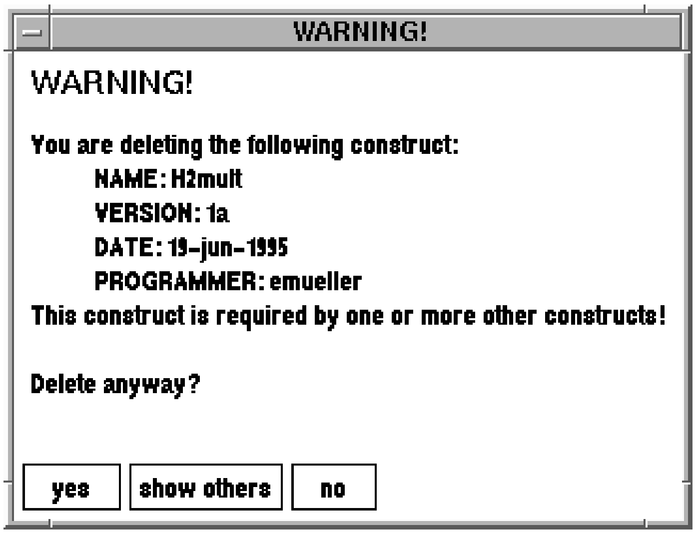


Abb. 9.13: Bestätigen des Löschens eines Hilfskonstruktes der Kategorie 1

Abb. 9.14: Bestätigen des Löschens eines Hilfskonstruktes der Kategorie 2

Bei Hilfskonstrukten der Kategorie 1 empfiehlt es sich in der Regel, den Löschvorgang zu unterbinden, damit die anderen Sprachkonstrukte, die dieses Hilfskonstrukt benötigen, weiterhin komplett in der Datenbank vorhanden sind.

Hinweis: Sie dürfen nur Sprachkonstrukte aus der Datenbank entfernen, die Sie selbst auch in der Datenbank abgelegt haben! Sollten Sie versuchen, ein Sprachkonstrukt zu löschen, das Sie nicht selbst in der Datenbank abgelegt haben, dann erhalten Sie eine Fehlermeldung entsprechend Abbildung 9.15.

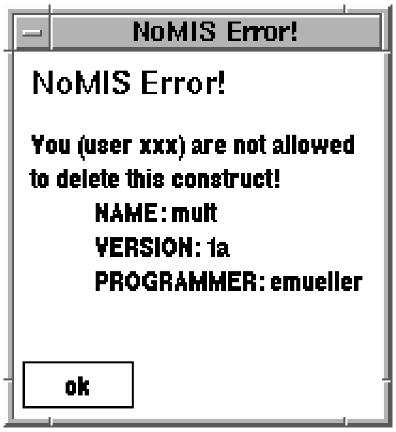


Abb. 9.15: Fehlermeldung beim unerlaubten Löschen eines Sprachkonstruktes

Nach dem Löschen aller Sprachkonstrukte, bzw. wenn Sie den Löschvorgang abgebrochen haben, kehren Sie in das Editor–Fenster zurück und können in der gewohnten Art mit dem NoName–Editor weiterarbeiten.

9.7 Die Erzeugung von LaTeX–Dateien

NoMIS ermöglicht die Erstellung von Übersichtslisten des Datenbankinhaltes in Form von LaTeX–Dateien. Dazu dient der Menüpunkt db-output aus dem Menü NoMIS. Wenn wir diesen Menüpunkt wählen, dann erscheint ein Fenster entsprechend Abbildung 9.16.

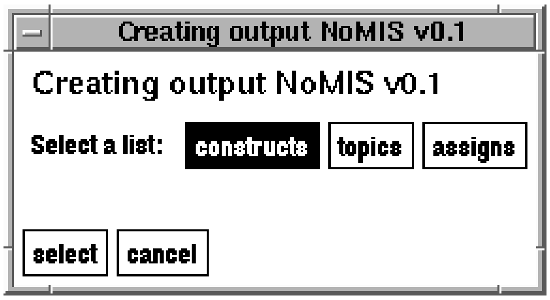


Abb. 9.16: Auswahl einer Übersichtsliste

Es stehen uns drei mögliche Listen zur Verfügung:

1. Auflistung aller in der Datenbank gespeicherten Sprachkonstrukte, sortiert nach dem Namen.

2. Auflistung aller in der Datenbank gespeicherten Topics, ebenfalls sortiert nach dem Namen.

3. Auflistung der Zuordnungen zwischen Sprachkonstrukten und Topics. Die Ausgabe erfolgt sortiert nach dem Namen der Sprachkonstrukte.

Wir gehen im folgenden davon aus, daß das Beispiel aus Abbildung 9.7 so in der Datenbank abgelegt wurde, wie es in Abschnitt 9.4 beschrieben wurde.

Wir wollen zunächst eine Auflistung aller gespeicherten NoName–Sprachkonstrukte erstellen. Dazu wählen wir im Fenster aus Abbildung 9.16 den Schalter `constructs` und bestätigen mit dem Button `select`. Damit erhalten wir ein Fenster, wie es in Abbildung 9.17 zu sehen ist.

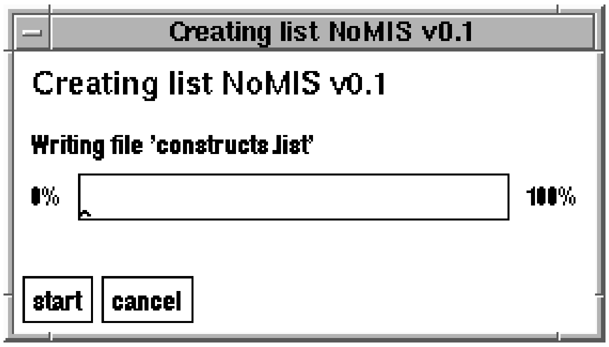


Abb. 9.17: Erzeugen einer Übersichtsliste

Das Fensterelement in der Mitte des Fensters zeigt anteilsmäßig an, wieviel Prozent der Ausgabe bereits erfolgt ist. Wenn dieses Fensterelement noch leer ist wie in der Abbildung 9.17, dann wurde noch keine Ausgabe vorgenommen, d.h. Sie müssen die Ausgabe erst noch durch die Betätigung des Buttons `start` beginnen. Dann wird eine Folge von dem Buchstaben ‚`X`' angezeigt, die immer länger wird, bis sie den rechten Rand des Fensterelements erreicht. Dann ist die Ausgabe abgeschlossen, Sie können durch Drücken des Buttons `cancel` in das Fenster aus Abbildung 9.16 zurückkehren.

In Abbildung 9.18 ist die bereits übersetzte Version des LaTeX–Dokuments dargestellt, welches durch diese Vorgehensweise erzeugt wurde.

NoMIS output: constructs.list

Output date: 19.06.95
Database: emuellerdb

Name:	H1mult
Version:	1a
Classification:	type specification
Programmer:	emueller
Date:	19-jun-1995
Comment:	Multiplikation zweier CARDINALs
Name:	H2mult
Version:	1a
Classification:	function
Programmer:	emueller
Date:	19-jun-1995
Comment:	Multiplikation zweier CARDINALs
Name:	mult
Version:	1a
Classification:	function
Programmer:	emueller
Date:	19-jun-1995
Comment:	Multiplikation zweier CARDINALs

Abb. 9.18: Übersicht aller gespeicherten Sprachkonstrukte

Wenn wir im Fenster aus Abbildung 9.16 den Schalter `topics` auswählen, dann erhalten wir wieder das Fenster aus Abbildung 9.17. Durch Betätigen des Buttons `start` wird ein LaTeX–Dokument erzeugt, welches in Abbildung 9.19 in der bereits übersetzten Version zu sehen ist.

NoMIS output: topic.list

Output date: 19.06.95
Database: emuellerdb

Name:	Arithmetik
Comment:	einfache arithmetische Funktionen etc.
Name:	System
Comment:	systemseitig verfuegbare Konstrukte
Name:	Trigonometrie
Comment:	trigonometrische Funktionen etc.

Abb. 9.19: Übersicht aller gespeicherten Topics

Schließlich haben wir noch die Möglichkeit, die Zuordnung von Sprachkonstrukten und Topics ausgeben zu lassen. Dazu wählen wir im Fenster aus Abbildung 9.16 den Schalter `assigns`. Wiederum erhalten wir ein Fenster analog Abbildung 9.17. Durch die Betätigung des Buttons `start` wird ein LaTeX–Dokument generiert, welches in Abbildung 9.20 ebenfalls in der übersetzten Version dargestellt ist.

NoMIS output: assign.list

Output date: 19.06.95
Database: emuellerdb

Name:	H1mult
Version:	1a
Topics:	Arithmetik
Name:	H2mult
Version:	1a
Topics:	Arithmetik
Name:	mult
Version:	1a
Topics:	Arithmetik

Abb. 9.20: Zuweisungen von Sprachkonstrukten und Topics

Die jeweils erzeugten LaTeX–Quelldateien können direkt dem LaTeX–Übersetzer gegeben werden. Unter Umständen kann es sinnvoll sein, diese Dateien von Hand etwas nachzuarbeiten, z.B. für zweispaltigen Ausdruck, falls dies aufgrund der Länge der Namen etc. möglich ist.

Hinweis: Die Prozentanzeige im Fenster aus Abbildung 9.17 ist nicht korrekt, falls nur sehr wenige Daten in der Datenbank enthalten sind. In diesem Fall erreicht der Balken nicht den rechten Rand der Anzeige, sondern stoppt schon früher. Dann ist die Ausgabe ebenfalls schon beendet. Sind mehr als 25 Sprachkonstrukte und Topics in der Datenbank enthalten, dann funktioniert die Anzeige in der gewünschten Weise.

9.8 Beenden einer Sitzung mit dem NoName–Editor

Vor dem Beenden des Editors durch den Menüpunkt exit aus dem Menü `File` sollten Sie immer die Verbindung zur Datenbank korrekt abbrechen. Dazu dient der Menüpunkt db-disconnect aus dem Menü `NoMIS`. Wenn Sie keine wichtigen Programme im NoName–Editor geöffnet haben bzw. keine Änderungen darin vorgenommen haben, dann können Sie auch den Punkt db-exit-all aus dem Menü `NoMIS` verwenden. Sie werden dann aber nicht auf eventuell ungespeicherte Änderungen eines NoName–Programmes aufmerksam gemacht! Daher empfiehlt sich immer die zweistufige Beendigung des Editors über die Menüpunkte db-disconnect mit anschließendem exit.

10 Kurzreferenz

Dieses Kapitel enthält eine alphabetisch angeordnete Kurzbeschreibung aller Menüpunkte des Menüs `NoMIS` (siehe Kapitel 10.1) und alle Buttons der Dialogfenster, die in NoMIS Verwendung finden (siehe Kapitel 10.2).

10.1 Menüpunkte des Menüs `NoMIS`

Für jeden Menüpunkt sind die Gliederungsnummer und die Nummer der Seite angegeben, auf welcher dieser Menüpunkt in der vorliegenden Arbeit erläutert wird. Diese Angabe erfolgt in der Form „(Gliederungsnummer) – Seite". Z.B. bedeutet (8.2) – 98 damit „Kapitel 8.2, Seite 98".

db-connect . . . . (8.2) – 98
Verbindung zwischen dem NoName–Editor und der Datenbank herstellen.

db-delete . . . . (8.2) – 97
Löschen eines Sprachkonstruktes aus der Datenbank.

db-disconnect . . . . (8.2) – 99
Abbrechen einer Verbindung zwischen dem NoName–Editor und der Datenbank.

db-exit-all . . . . (8.2) – 99
Verlassen des NoName–Editors nach Abbruch der Verbindung zur Datenbank.

db-output . . . . (8.2) – 98
Erzeugung der Übersichtslisten in Form von LaTeX–Dateien.

db-retrieve . . . . (8.2) – 97
Übernahme eines gespeicherten Sprachkonstruktes in den NoName–Editor.

db-store . . . . (8.2) – 97
Speichern des im NoName–Editor aktuell markierten Sprachkonstruktes in der Datenbank.

topic-handling . . . . (8.2) – 98
Anstoßen der Topic–Verwaltung: Anlegen eines neuen Topics, Löschen eines nicht mehr benötigten Topics, Anzeige aller gespeicherten Topics.

10.2 Buttons der Dialogfenster

Für jeden Button ist eine Liste von Abbildungsnummern angegeben. In den zugehörigen Abbildungen ist ein Fenster zu sehen, das diesen Button enthält.

cancel 9.2, 9.3, 9.4, 9.6, 9.8, 9.10, 9.11, 9.12, 9.16, 9.17
Abbrechen des aktuellen Bearbeitungsschrittes bzw. Schließen des Dialogfensters nach Beendigung des Bearbeitungsschrittes.

close 9.5
Schließen des Dialogfensters zur Anzeige aller gespeicherten Topics.

delete 9.6, 9.12
Anstoßen des Löschvorganges beim Löschen eines Topics bzw. eines Sprachkonstruktes.

no 9.13, 9.14
Löschen von Hilfskonstrukten unterbinden.

ok 9.15
Schließen des Dialogfensters, welches den Benutzer darauf hinweist, daß er das gewünschte Sprachkonstrukt nicht löschen darf. Auch: Schließen des Dialogfensters zur Anzeige des NoName–Codes (ohne Abbildung).

retrieve 9.11
Übernahme des gewählten Sprachkonstruktes und aller seiner Hilfskonstrukte aus der Datenbank in den NoName–Editor.

save 9.4
Anstoßen des Speichervorganges beim Speichern eines neuen Topics.

scan 9.11, 9.12
Suche nach Sprachkonstrukten in der Datenbank, die zu den gewählten Topics passen.

select 9.3, 9.10, 9.16
Auswahl aus einer Toggle–Box bestätigen.

show code 9.8, 9.11, 9.12
Anzeigen des NoName–Codes des Sprachkonstruktes, welches gespeichert, importiert oder gelöscht werden soll.

show others 9.13
Anzeige der Sprachkonstrukte, welche von dem zu löschenden Sprachkonstrukt abhängig sind.

start 9.2, 9.17
Herstellung einer Verbindung zur Datenbank (in Abbildung 9.2) bzw. Starten der Ausgabe in eine LaTeX–Datei (in Abbildung 9.17).

store 9.8
Anstoßen des Speichervorganges beim Speichern eines neuen Sprachkonstruktes und aller seiner benötigten Hilfskonstrukte.

yes 9.13, 9.14
Löschvorgang des angezeigten Hilfskonstruktes bestätigen.

Teil III
Anhang

In der Original–Arbeit enthält dieser Teil die Listings der Quelldateien des NoName–Modul–Informationssystems. Der interessierte Leser kann diese Dateien unter der E-Mail–Adresse

mueller@dbis.informatik.uni-ulm.de

anfordern.

Literaturverzeichnis

[DS94] P. Dadam und M. Scholl. *Blätter zur Vorlesung „Datenbanksysteme I: Grundlagen, Datenmodelle und Systemkonzepte“*. Universität Ulm, WS 1993/94.

[EGH+94a] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mößle, E. Müller, S. Sablatnög, H. Vogler und E. Wieser. *NoName — Eine funktionale Sprache mit Unterstützung der primitiven Rekursion*. Universität Ulm, 1994.

[EGH+94b] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mößle, E. Müller, S. Sablatnög, H. Vogler und E. Wieser. *Handbuch zum NoName–Editor, Version 1.0*. Universität Ulm, 1994.

[EN89] R. Elmasri und S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1989.

[Ern95] D. Ernst. *Entwicklung und Implementierung eines syntaxgesteuerten Editors für die Sprache NoName*. Diplomarbeit, Universität Ulm, 1995.

[Gra95] GrammaTech. *The Synthesizer Generator 4.2 Reference Manual*, 1995.

[Ing91] Ingres Corporation. *INGRES/Embedded SQL Companion Guide for C*, 1991. Release 6.4.

[MS93] J. Melton und A. R. Simon. *Understanding SQL: A complete guide*. Morgan Kaufmann Publishers, Inc., 1993.

[Pet92] D. Petković. *INGRES — Das relationale Datenbanksystem mit Knowledge–Base und Object–Base*. Addison Wesley, erste Auflage, 1992.

[Reg91] Regionales Rechenzentrum für Niedersachsen/Universität Hannover. *Die Programmiersprache C — Ein Nachschlagewerk*, 1991.

[RT89] T. W. Reps und T. Teitelbaum. *The Synthesizer Generator — A system for constructing language–based editors*. Springer–Verlag, 1989.

[Teo94] T. J. Teorey. *Database Modeling & Design*. Morgan Kaufmann Publishers, Inc., zweite Auflage, 1994.