Der $DoDL_C$-Compiler

Alexander Fronk Jörg Pleumann

MEMO
an der
Universität Dortmund
Lehrstuhl für Software-Technologie

Juli 1999

Die Sprache *DoDL* wurde 1996 erstmals vorgestellt. Sie propagiert die Möglichkeit, Hyperdokumente durch die Trennung ihres Inhalts von ihrer Struktur zu spezifizieren (**DoDL** = **Do**cument **D**escription **L**anguage). Dazu bedient sich *DoDL* objektorientierter Strategien und insbesondere einer Gastsprache, die, in *DoDL* integriert, die konstruktive Beschreibung hypermedialer Strukturen ermöglicht. Wir zeigen in dieser Arbeit einen Compiler für *DoDL*, wobei wir die Sprache C als Gastsprache herangezogen haben.

Mit $DoDL_C$ lassen sich Hyperdokumente spezifizieren, ohne direkt auf HTML-Ebene codieren zu müssen. Da Hyperdokumente jedoch im allgemeinen nicht ohne HTML auskommen, benötigt man ein Compilersystem, das eine $DoDL_C$-Spezifikation nach HTML transformiert. Dazu wird die Spezifikation in ein C-Programm übersetzt, welches einen Graphen erzeugt, der wiederum mit einem geeigenten Werkzeug traversiert und ausgelesen werden kann, um daraus HTML-Code zu generieren. Wir zeigen hier einen Compiler, der eine *DoDL*-Spezifikation in ein ANSI-C-Programm transformiert.

Das Memo gliedert sich in drei Teile: **Teil I** behandelt die Sprache $DoDL_C$. Sie wird anhand eines durchgängigen Beispiels erklärt. Weitere Schwerpunkte liegen auf den Sichtbarkeitsregeln und der formalen Syntax in Form leicht lesbarer Syntaxdiagramme.

Teil II beschreibt die Benutzung des Compilers als UNIX Shell-Tool, **Teil III** gibt Einblicke in die Compiler-Interna, und **Teil IV** schließt die Arbeit ab.

Damit kann diese Arbeit sowohl als Lernunterstützung für $DoDL_C$ als auch als eine Fallstudie für *angewandten Compilerbau* verstanden werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Teil I.

$DoDL_C$-Handbuch

1. Die Sprache *DoDL*$_C$

Wir beschreiben die Sprache *DoDL*$_C$, jedoch nicht so, wie sie in [Dob96b] eingeführt wurde. Als Gastsprache haben wir C implementiert. Die Sprachbeschreibung wird daher nicht, wie ursprünglich erdacht, auf Prolog [Bra88, CM81, NS93, Ban86] aufsetzen, sondern unsere Implementierung berücksichtigen.

1.1. Hyperdokumente und ihre Abstraktion durch Graphen

Die Möglichkeit, Informationen nicht-linear verknüpfen zu können, ist das offensichtlichste Merkmal von Hyperdokumenten. Zu den Informationen zählen Texte, aber auch Grafiken, Videos oder andere Medien (wir nennen diese auch *Medienobjekte*). Die Verknüpfungsstruktur, d.h. die Verbindung der Medien untereinander, läßt sich geeignet als Graph abbilden, wenn man *Links* als Kanten zwischen Knoten annimmt, die wiederum gerade durch „anklickbare", sagen wir Quell- und Zielobjekte innerhalb der Medien gebildet werden. Nehmen wir an, die Positionen dieser Objekte in bspw. einem Text eindeutig bestimmen zu können. Dann gelingt die Beschreibung von Hyperdokumenten gerade durch die Spezifikation resp. die Konstruktion der erwähnten Graphen. Ihren Knoten können eindeutig Positionen in Medienobjekten zugeordnet werden. Hält man die Konstruktion der Links möglichst einfach, lassen sich diese ebenfalls abstrakt, aber in Bezug auf ein konkretes Medium bestimmen. Gemeint sind hier Links der Art „Verknüpfung aller Vorkommen eines Strings in einem Text mit einem String eines anderen Textes, oder dem Beginn, dem Ende eines Medienobjekts", bzw. beliebige Variationen dieses Themas.

DoDL$_C$ ist zu dem Zweck entworfen worden, die Konstruktion solcher Graphen in objektorientierter Weise vornehmen zu können. Durch die Trennung von Inhalt und Struktur, sprich durch eine von den konkreten Medienobjekten separierte Beschreibung eines Graphen, gelingt ein flexibler und „sauberer" Ansatz im Sinne des Sprachentwurfs.

Im folgenden werden wir die Sprache *DoDL*$_C$ unter die Lupe nehmen und ihre Eigenschaften und Wirkungsweisen anhand der schrittweisen Entwicklung eines kleinen Hyperdokuments aufzeigen.

1.2. Klassenaufbau

Eine einfache *DoDL*$_C$-Klasse wie in Spezifikation 1.1 ist zunächst ein Konstrukt, das neben einem Klassenkopf aus mehreren Sektionen besteht, denen bestimmte Aufgaben zugeordnet werden. Konzeptionell gesehen werden eine *Datensicht*, eine *Hypersicht* und eine *Durchlaufsicht* formuliert. Prinzipiell kann man Klassen auf zwei Ebenen definieren: global, also für jede andere Klasse sichtbar, oder lokal, dann mit eingeschränktem Sichbarkeitsbereich. Dem Sichtbarkeitsbereich ist ein eigener Abschnitt vorbehalten (siehe dazu Kapitel 2).

```
class Geruest is
  declare class eineLokaleKlasse is // Lokale Klassen hier definieren
            ...
          end eineLokaleKlasse;
            ...
  documents ...  // Die Sicht auf die verwendeten Daten
  construct ...  // hier wird die Hypersicht beschrieben
  browsing ...   // Diese Sektion beinhaltet die Durchlaufsicht
end Geruest;
```

Quellcode 1.1: Ein Klassengerüst

Die `declare`-Sektion dient der Definition lokaler Klassen, gerade so, wie das in Java [AG96, Bis98, CW96, DD98, GJS97, SM] oder Beta [MMRN93, DD96] bekannt ist. Die Datensicht beinhaltet in der `document`-Sektion eine variable Anzahl von Platzhaltern, Variablen für Medienobjekte, auf die sich die Konstruktion des angesprochenen Graphen in der `construct`-Sektion bezieht. Die `browsing`-Sektion formuliert eine Selektionsregel, die konstruierte Links anhand von Attribut/Wert-Paaren zum Durchlaufen freigibt. Hier in der Hypersicht werden also dynamische Verhaltensweisen des spezifizierten Hyperdokuments festgelegt.

Schenken wir nun unsere Aufmerksamkeit den einzelnen Sektionen. Es gilt niemals zu vergessen, daß jede, ja wirklich jede Sektion rein optional ist. Die *leere Klasse* hat damit die einfachste Struktur:

```
  class empty is end empty;
```

Quellcode 1.2: Die leere Klasse

Der Optionalitätscharakter der Sektionen führt zu sehr einfach lesbaren Klassenhierarchien, aber das sehen wir später noch genauer. Kommentare werden zeilenweise hinter einem doppelten Schrägstrich notiert oder können auch als Block und dann an beliebigen Stellen in `/*` und `*/` eingefaßt werden.

1.2.1. Die `document`-Sektion

Die `document`-Sektion erlaubt die Deklaration von Variablen. Diese können drei unterschiedliche Typen annehmen: erstens einfache, atomare Typen, zweitens selbstdefinierte Typen (Klassen) sowie drittens Listen all dieser Typen. Atomare Typen wie `string, integer` oder auch `dbUnit` und die davon abgeleiteten Typen `text` und `graphics` sind in $DoDL_C$ vordefiniert. Klassen – und damit Typen – können aber, wie wir gerade sehen, selbst definiert werden und stehen dann als Typen zur Verfügung. Hierbei gilt es selbstverständlich, den Sichtbarkeitsbereich zu beachten, aber dazu später. Zunächst ist es wichtig zu wissen, welche Werte die genannten Typen annehmen dürfen. `string` und `integer` sind selbsterklärend, `dbUnit` und seine Derivate jedoch nicht.

Wir haben in der Einleitung darüber geredet, daß Medienobjekte zu Hyperdokumenten verknüpft werden. Eben diese Medienobjekte verbergen sich hinter `dbUnit`. Gemeint ist hier ganz allgemein ein persistentes Datum, also irgendein Medienobjekt, das sich z.Bsp. in einer Datenbank speichern läßt. Der Begriff *Datenbank* steht natürlich auch synonym für ein Dateisystem. Eine `dbUnit` kann als polymorpher Typ verwendet werden, meist jedoch in den speziellen Formen `text` bzw. `graphics`.

Eine Listenkonstruktion kann ebenfalls eingesetzt werden. Es wird dann eine Liste von Instanzen erzeugt, in der jede Instanz gerade denjenigen Typ besitzt, den die Liste definiert. Die einzelnen Elemente der Liste können durch Indizes angesprochen werden. Zu beachten ist dabei die Verwendung des Schlüsselwortes `list of` und eine geeignete Initialisierung durch die `binding`-Sektion, wie Abschnitt 1.2.4 das erklärt.

Das Beispiel in 1.3 zeigt eine Deklaration von Instanzen des atomaren Typs `dbUnit`. Selbstdefinierte Typen, wie in Abschnitt 1.2.2 zu sehen ist, werden analog deklariert.

```
class Decoration is
  documents greeting, go_home: dbUnit;
end Decoration;
```

Quellcode 1.3: Die `document`-Sektion

1.2.2. Die `declare`-Sektion

Die `declare`-Sektion erlaubt die Deklaration lokaler Klassen. Die darin deklarierten Klassen besitzen selbstverständlich das gleiche Gerüst, wie wir es vorhin gezeigt haben. Das bedeutet aber insbesondere, daß wir in der Lage sind, kleine lokale Spezifikationen zu schreiben, da für lokale Klassen ebenfalls sämtliche in $DoDL_C$ vorhandenen Modellierungskonzepte zur Verfügung stehen. Spezifikation 1.4 zeigt eine Klasse `SimpleHyperDoc`, die ein paar lokale Klassen und eine `document`-Sektion enthält. Diese Spezifikation wird im weiteren Verlauf erweitert werden.

```
  class SimpleHyperDoc is // eine einfache Klasse mit zwei lokalen Klassen
    declare class Decoration is
              documents greeting, go_home: dbUnit;
            end Decoration;
  /* es sollen die drei Dokumente homepage, aboutMe und Pic munter
     miteinander verlinkt werden. */
  documents homepage, aboutMe : text;     // zwei Texte
            reference         : string;   // ein String
            Pic               : graphics; // und eine Graphik
            deco              : list of Decoration;
            /* eine Liste von Instanzen der Klasse Decoration */
end SimpleHyperDoc;
```

Quellcode 1.4: Eine einfache Deklaration lokaler Klassen

Es wäre langweilig, nur Typen und Variablen zu deklarieren, ohne damit arbeiten zu wollen. Das tun wir im folgenden Abschnitt.

1.2.3. Die `construct`-Sektion

Die `construct`-Sektion wird mit dem Schlüsselwort `construct` eingeleitet. In dieser Sektion werden die Methoden derjenigen Klasse definiert, zu der diese Sektion gehört. Wir werden bei der Vererbung in Abschnitt 1.3 sehen, daß die Methoden der Superklasse redefiniert werden dürfen. Dies geschieht dann selbstverständlich in der erbenden Klasse.

Die Methoden werden nicht in der Sprache *DoDL* selbst implementiert, weil dazu keine Konstrukte definiert sind. Es gehört zur Philosophie von *DoDL*, hierzu eine sog. *Gastsprache* zu verwenden. In der vorliegenden Version ist das ANSI-C. Es ist sehr wichtig zu wissen, daß aufgrund der Objektorientierung von $DoDL_C$ die Gastsprache ANSI-C geringfügig erweitert werden mußte. So gibt es nun ein vordefiniertes Schlüsselwort `super`, mit dem der Aufruf von Methoden aus Superklassen realisiert werden kann. Ebenso ist es möglich, Methoden per *Qualifizierung*, also durch eine Punkt-Notation aufzurufen. Auch das `this`-Konstrukt zur Referenzierung von Attributen und Methoden des aufrufenden Objektes ist in $DoDL_C$ vordefiniert. Diese Erweiterungen sind Bestandteil von $DoDL_C$, auch wenn das widersprüchlich erscheinen mag. Um den objektorientierten Rahmen von $DoDL_C$ in der Gastsprache nutzen zu können, sind diese Erweiterungen unerläßlich. Sie bilden sozusagen den „Haken“, an dem die Gastsprache an $DoDL_C$ aufgehängt wird.

In der `construct`-Sektion werden aber wirklich nur Methoden definiert, nicht etwa auch Blöcke wie in BETA oder PASCAL [HS84, Ros89, Wir95], die mit `do` oder `begin` eingeleitet werden. Daraus ergibt sich eine Besonderheit: „Wo bringe ich denn mein Hauptprogramm unter?“ Wir wissen, daß in ANSI-C das Hauptprogramm ebenfalls eine Funktion ist, nämlich eine mit dem Schlüsselwort `main`. Genau dies hilft uns hier: die Namenskonvention wird schlichtweg beibehalten. Jede $DoDL_C$-Klasse kann analog zu Java eine parameterlose Methode `main` besitzen, durch die die Klasse ausgeführt werden kann. In einer Spezifikation kommen aber nun meist mehrere Klassen vor, die auch noch auf der gleichen, sagen wir einer äußeren globalen Ebene definiert sind. Welches `main` welcher Klasse ist das eigentliche Hauptprogramm? Um diese Frage zu beantworten, müssen wir die `binding`-Sektion beachten, die wir in Abschnitt 1.2.4 besprechen werden.

Wie erzeugt man denn nun einen Graphen? Man muß Knoten und Kanten erzeugen!

Es gibt in $DoDL_C$ vordefinierte Methoden, die für die Spezifikation von Hyperdokumenten unerläßlich sind. Diese Methoden konstruieren Start- und Zielpunkte (Knoten) von Links und natürlich Links (Kanten) selbst. Zum einen können wir als Knoten die Vorkommen eines Strings in einem Text sammeln. Das leistet eine Methode `PositionList getOcc(char *str)`, die auf ein Dokument vom Typ `dbUnit` angewendet werden kann und sämtliche Vorkommen von `str` in diesem sammelt und in einer Positionsliste zurückliefert. Die Struktur dieser Liste ist für die Verwendung völlig belanglos. Es genügt zu wissen, daß erstens jedes Element dieser Liste über einen Index angesprochen werden kann, und daß zweitens auf jede Position die Link-erzeugende Methode `void setLink(Position pos)` angewendet werden darf. Diese bildet einen Link von der aufrufenden Position zu `pos`. Wird diese Methode auf einer Positionsliste aufgerufen, werden alle Positionen dieser Liste durch einen Link mit `pos` verbunden, also eine $n:1$-Verbindung erzeugt. Auch $n:n$-Verbindungen sind beschreibbar, man muß ja lediglich `setLink` über einer Zielknotenmenge iteriert aufrufen. Solche Verbindungen bedürfen jedoch einer gesonderten Behandlung im Browser.

Wir können auch den Beginn und das Ende einer `dbUnit` explizit als Ankerpunkte für Links identifizieren. Dazu benutzen wir die Methoden `Position getBegin(void)` und `Position getEnd(void)`. Ihren Einsatz sehen wir im folgenden Beispiel.

Spezifikation 1.5 zeigt die Klasse `Decoration` aus Spezifikation 1.4, jetzt jedoch mit Methodenteil. Wir sehen, daß mehrere Methoden definiert sind, ganz so, wie wir das in ANSI-C gewohnt sind. Eine Methodendefinition beginnt stets mit dem Typ ihres Rückgabewertes, gefolgt vom Methodennamen und einer Parameterliste. Wir haben eine `main`-Methode in der Klasse `SimpleHyperDoc` definiert, die für die explizite Verknüpfung der definierten Dokumente sorgt. Die wesentlichen Bestandteile einer $DoDL_C$-Klasse und die Wirkungsweise von $DoDL_C$ zur Erzeugung von Hyperdokumenten können wir an diesem Beispiel schon sehr gut erkennen. Insbesondere sehen wir die Verwendung der Methoden `getBegin`, `getEnd` und `setLink`, sowie den Gebrauch der Variablen `homepage` aus der umgebenden Klasse `SimpleHyperDoc`.

Mit dieser Spezifikation haben wir eine Menge von Hyperdokumenten beschrieben, nicht nur ein einzelnes. Das liegt daran, daß wir die Variablen noch nicht mit konkreten Medienobjekten gefüttert haben. Aus jeder Belegung der Variablen resultiert möglicherweise ein anderes Hyperdokument, da ja zum Beispiel die Anzahl der Vorkommen von `reference` in den Dokumenten verschieden sein kann (dann entstehen unterschiedlich viele Links), oder ein anderes Bild wird verwendet. Was jedoch all diesen Hyperdokumenten gemeinsam ist, ist der Algoritmus, der ihre Verknüpfungsstruktur erzeugt. Der Algorithmus - und damit die Verknüpfungsstruktur - wird gerade durch die Spezifikation festgelegt. Man kann selbstverständlich den Aufruf der Positions- und Link-erzeugenden Methoden von beispielsweise dem Auffinden bestimmter Wörter in einem Text abhängig machen. Damit würden sich für den Betrachter recht unterschiedliche Verknüpfungsstrukturen auftun, Graphen also sichtbar werden, die insbesondere nicht isomorph zueinander sind. Dennoch ist all diesen Graphen und damit den Hyperdokumenten gemeinsam, daß sie durch eine einzige Spezifikation beschrieben werden können.

Binden wir nun konkrete Medienobjekte an unsere Variablen und schauen uns das erzeugte Hyperdokument resp. denjenigen Graphen an, der gerade dessen Verknüpfungsstruktur reflektiert.

1.2.4. Die `binding`-Sektion

Um eine Spezifikation an bestimmte Medienobjekte zu binden, müssen wir die Variablen der jeweiligen `document`-Sektionen aller beteiligten Klassen mit konkreten Werten belegen.

Das Schlüsselwort `binding` leitet eine `binding`-Sektion ein, die wohlgemerkt außerhalb der Klassendeklarationen vorgenommen wird, meist in einer eigenen Datei. Dadurch wird die einfache Austauschbarkeit der Variablenbelegung gewährleistet und der Trennung von Struktur und Inhalt der Hyperdokumente Rechnung getragen.

Binden wir also die Variablen der Spezifikation 1.5 an konkrete Werte. Dazu definieren wir eine `binding`-Sektion für die Klasse `SimpleHyperDoc` (siehe Binding 1.6).

Wir sehen in der `binding`-Sektion 1.6 schon alle Eigenschaften, die eine solche Sektion haben kann. Die Variablen vom Typ `dbUnit` werden mit Dateien belegt, die sich an den

```
class SimpleHyperDoc is
  declare class Decoration is
            documents greeting , go_home: dbUnit;
            construct
              void setGreeting (dbUnit page) {
                /* Eine Seitenbegruessung wird gesetzt . Ihr Ende wird
                   vor den Beginn der Datei page gehaengt . Wir erhalten
                   damit eine explizite lineare Verlinkung zweier
                   Dokumente.  page kann wie greeting ein beliebiges
                   Medienobjekt sein! */
                greeting.getEnd().setLink(page.getBegin());
              }

              void setHomeRef(dbUnit page) {
                /* Am Seitenende verweist eine Datei go_home auf
                   den Beginn der Startseite . */

                page.getEnd().setLink(go_home.getBegin());
                go_home.getEnd().setLink(homepage.getBegin());
                                          // homepage aus SimpleHyperDoc
              }

              void installDeco (dbUnit page) { // ein Faulenzer
                setGreeting (page);
                setHomeRef(page);
              }
          end Decoration;
  documents homepage, aboutMe : text;
            reference         : string;
            Pic               : graphics;
            deco              : list of Decoration;
  construct
    void main(void) { //das Hauptprogramm
      // erstmal den Dokumenten eine Deko verpassen
      deco[0].installDeco (homepage);
      deco[1].installDeco (aboutMe);
      deco[2].installDeco (Pic);

      // jedes Vorkommen von reference wird mit
      // der Deko von aboutMe verlinkt
      homepage.getOcc(reference).setLink(deco[1].greeting.getBegin());

      // das erste Vorkommen von "Uni" in aboutMe
      // fuehrt zur Deko von Pic
      aboutMe.getOcc("Uni")[1].setLink(deco[2].greeting.getBegin());
    }
end SimpleHyperDoc;
```

Quellcode 1.5: Die `construct`-Sektion

```
binding SimpleHyperDoc is
  homepage   = "~/texte/Welcome.txt";
  aboutMe    = "~/texte/Info.txt";
  reference  = "Arbeit";
  Pic        = "~/pics/Uni.jpg";
  in deco assign
    // eine Deko fuer die Startseite
    greeting = "~/texte/Welcome_greet.txt";
    go_home  = "~/texte/back.txt";
    | // neues Element besorgen
      // eine Deko fuer aboutMe
    greeting = "~/texte/aboutMe_greet.txt";
    go_home  = "~/texte/back.txt";
    |
      // eine Deko fuer das Bild
    greeting = "~/texte/Pic_greet.txt";
    go_home  = "~/texte/back.txt";
  end;                              // Ende von in
end;                                // Ende der Binding-Sektion
```

Quellcode 1.6: Die `binding`-Sektion

entsprechenden Stellen im Dateisystem befinden sollten. Man könnte hier anstatt `back.txt` ein Icon einsetzen, ohne die Klassenbeschreibung selbst ändern zu müssen!

Eine Liste von `Decoration` wird durch die Variable `deco` gebildet. Ihre Elemente werden durch Belegung der Variablen der Klasse `Decoration` erzeugt. Eingeleitet wird diese Belegung durch `in <Varname> assign`, die Elemente werden dabei durch einen senkrechten Strich voneinander getrennt. Es entstehen in der `binding`-Sektion also offensichtlich drei Instanzen. Genau soviele werden benötigt, um die Dokumente der Klasse `SimpleHyperDoc` zu dekorieren.

Die Belegung einer Variablen, die als Liste eines atomaren Typs deklariert wurde – also etwa `intList: list of integer;` – geschieht auf beinahe dieselbe Art und Weise. Der einzige Unterschied besteht darin, daß atomare Typen keine Struktur besitzen. Im Teil `in intList assign` wird daher lediglich ein Wert, aber keine Zuweisung gemäß `<varname> = <value>;` aufgeführt. Also würde beispielsweise die Anweisung

```
in intList assign
  7;
  |
  13;
  |
  1;
end;
```

eine Liste `intList` vom Typ `integer` mit drei Elementen, nämlich 7, 13 und 1 erzeugen.

1.2.5. Ein exemplarisches Hyperdokument

Damit wir das aus Spezifikation 1.5 und dem Binding 1.6 entstehende Hyperdokument und seinen Graphen zeigen können, müssen wir die in der `binding`-Sektion zugewiesenen Dateien kennen. Damit deren Umfang nicht auswuchert, haben wir uns bescheiden auf kleine Texte zurückgezogen.

Angenommen, die Datei `Welcome.txt` habe den Inhalt

> Ich erzähle Ihnen ein wenig über meine Arbeit, damit Sie wissen, womit ich mich beschäftige.

während `Info.txt` aus

> Ich arbeite am Lehrstuhl für Software-Technologie an der Uni Dortmund. Dort beschäftige ich mich mit der Sprache $DoDL_C$, ihrer Anwendung und ihrer formalen Semantik.

besteht. Ein Bild unserer Uni, das wir `Pic` zugewiesen haben, sei:

Der Inhalt der Dekorationsdateien ist nicht wesentlich und kann Abbildung 1.1 entnommen werden.

Jetzt sind wir in der Lage, das Hauptprogramm `main` der Klasse `SimpleHyperDoc` auszuführen. Das enstehende Hyperdokument sehen wir in Abbildung 1.1. Dabei sind die Links als Kanten gemalt. Sie beginnen in Teilen und zeigen auf Teile von Medienobjekten. Die Kanten, die von den Dekorationen gebaut werden, sind durchgezogen gemalt, gestrichelte Kanten zeigen die explizite Verlinkung durch Aufrufe von `getOcc`. Die ursprünglichen Medienobjekte haben wir gerahmt, die gestrichelten Rahmen sollen physikalische Seiten andeuten, wie sie später als HTML-Dateien abgelegt werden würden. Wir schauen uns den Graphen genauer an, da er die Verknüpfungsstruktur repräsentiert und erklären dabei schrittweise seine Entstehung. Zunächst gilt es, die drei Dateien aus `SimpleHyperDoc` mit geeigneten Dekorationen zu versehen. `Welcome.txt` erhält als Kopfzeile die Datei `Welcome_greet.txt`, als Fußzeile `go_home.txt`. Nachdem die Methode `deco[0].installDeco(homepage);` ausgeführt wurde, entsteht eine erste Teilstruktur (siehe Abb. 1.2) unseres Hyperdokuments. Wir erinnern uns,

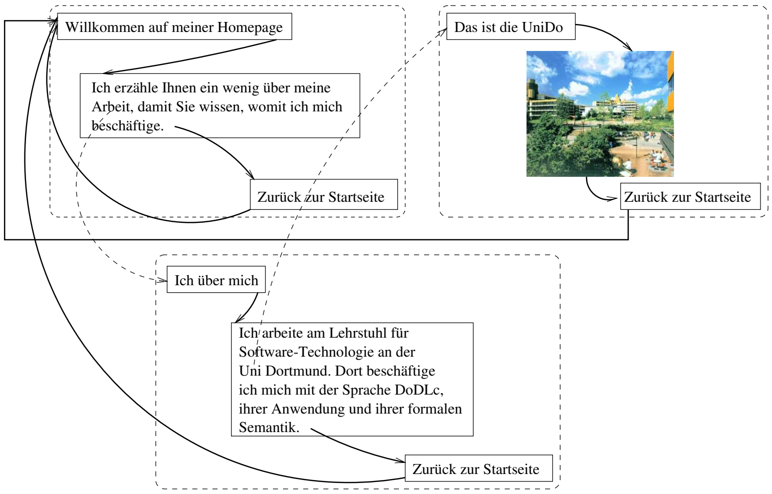

Abbildung 1.1.: Das entstehende Hyperdokument

daß die Knoten des nun entstehenden Graphen durch die Knoten-erzeugenden Methoden `getBegin, getEnd` und `getOcc` gebildet werden, während die Kanten durch `setLink` entstehen. Wir sehen in Abbildung 1.2 außerdem eine Attributierung der Knoten. Zu den Attributen zählen `File` und `Position`. Das erste Attribut sagt, zu welchem Medienobjekt der Knoten gehört, sprich, in welchem Medienobjekt sich dieser Ziel- oder Quellpunkt eines Links befindet. Das zweite Attribut gibt dann den genauen Ort innerhalb dieses Medienobjektes an. Die Namen der Knoten sind mnemonisch vergeben. Die Herkunft der Knoten ist daher recht einfach zu verstehen.

Die anderen Dekorationen verlaufen analog. Abbildung 1.3 zeigt die Dekoration der Datei `Info.txt`, repräsentiert in der Variablen `aboutMe`.

Wir schauen aber mal noch auf die Methoden

```
homepage.getOcc(reference).setLink(deko[1].greeting.getBegin());
```

und

```
aboutMe.getOcc('Uni')[1].setLink(deko[2].greeting.getBegin());
```

und was diese erzeugen (Abb. 1.4 und 1.5).

Es wird in der ersten Methode ein Knoten gebildet, der ein Vorkommen des Strings `reference` in der Datei `Welcome.txt` repräsentiert. Ein weiterer Knoten steht für den Beginn der Datei `Info.txt`. Dann wird ein Link erzeugt, der vom ersten Knoten zum dritten Knoten geht. Die zweite Methode erzeugt einen Knoten für das Vorkommen des Strings `Uni` in der Datei `aboutMe` und verlinkt es mit dem Beginn der Dekoration der Bildseite.

Wie nun das gesamte Hyperdokument aus Abbildung 1.1 als Graph aussieht, ist nicht mehr schwierig zu fassen, aber sehr umfangreich. Wir haben deswegen auf die Attributierung verzichtet. Abbildung 1.6 zeigt den gesamten Graphen.

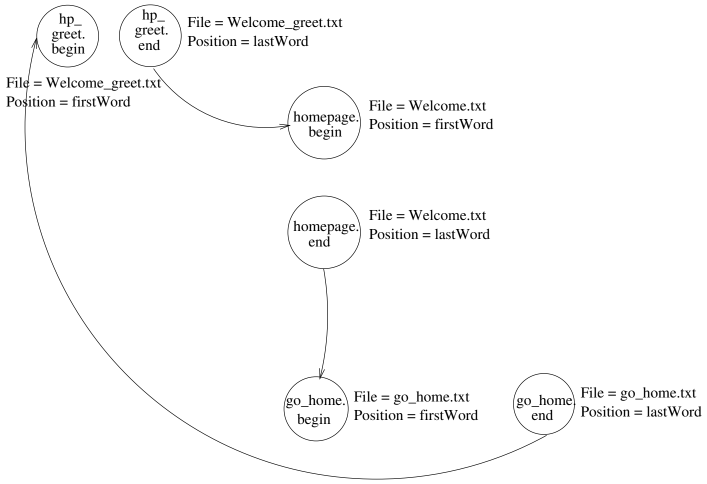

Abbildung 1.2.: Der Graph der dekorierten Startseite

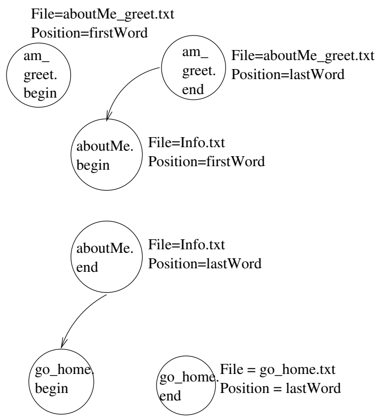

Abbildung 1.3.: Der Graph der dekorierten Datei aboutMe

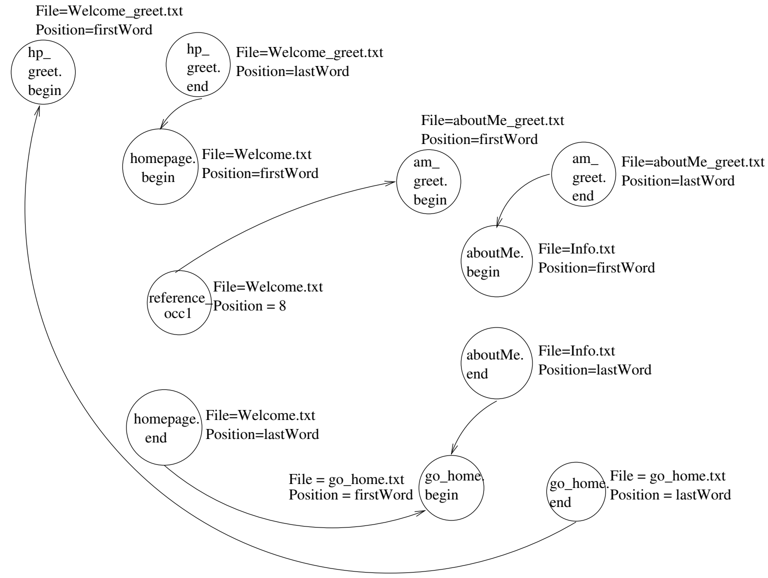

Abbildung 1.4.: Die Verlinkung der Startseite

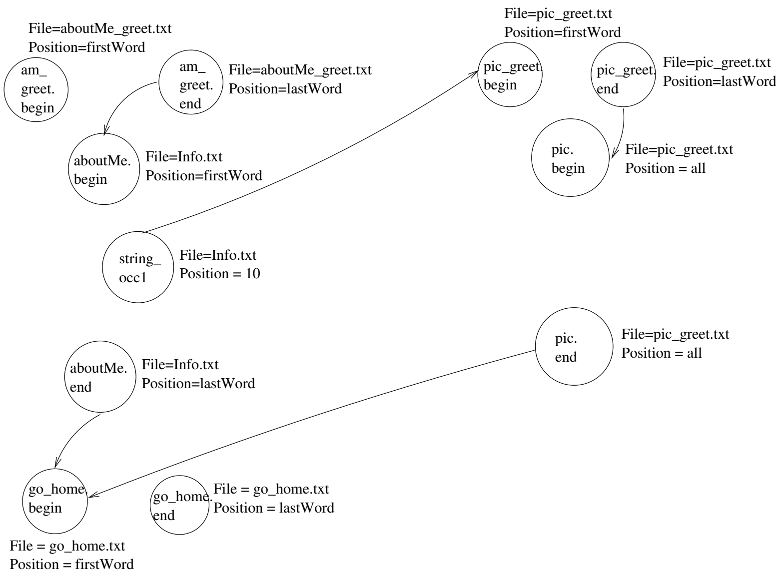

Abbildung 1.5.: Die Verlinkung des Bildes

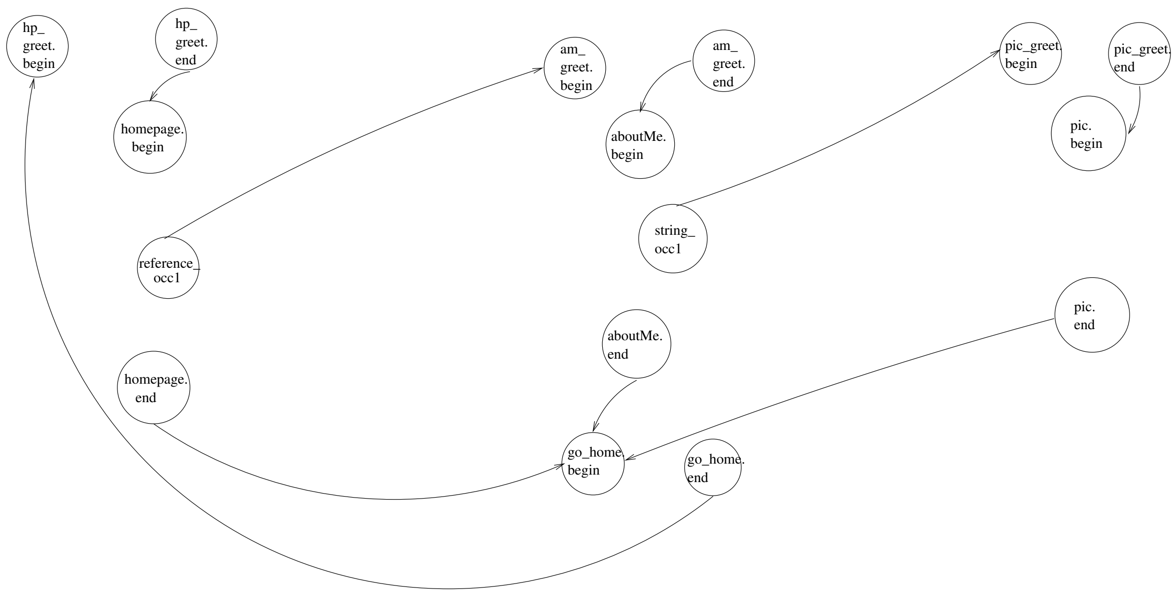

Abbildung 1.6.: Die Verknüpfungsstruktur eines Hyperdokuments

1.2.6. Die `browsing`-Sektion

Ein Hyperdokument besitzt neben seinen statischen Komponenten (nämlich seinem Inhalt und seiner Struktur) auch eine dynamische Seite: sein Verhalten zur Traversierzeit [FD97]. Man kann dann Fragen stellen wie „Wie wird das Hyperdokument benutzt, wie kann es benutzt werden und was geschieht, wenn es benutzt wird?“ Um diese Fragen zu beantworten, gibt es in $DoDL_C$ die `browsing`-Sektion [Dob96a]. Hierin werden *Feature-Terme* [Smo92, Zel97] notiert. Nur keine Panik, das sind schnuckelige Logik-basierte Konstrukte, die in unserem Fall nichts weiter tun, als furchtbar formal zu wirken (es dabei auch zu sein, was gerade das Schöne an ihnen ist) und die Menge der in der `construct`-Sektion erzeugten Links einfach wieder einzuschränken. Das geschieht so: In der `construct`-Sektion lassen sich Attribute für Knoten und Kanten explizit erzeugen und mit Werten belegen. So könnte jeder Link ein Attribut tragen, das an verschiedenen Links verschiedene Werte annehmen kann. Ein Feature-Term ist dann in der Lage, all diejenigen Links zu selektieren und damit zur Traversion freizugeben, die an diesem Attribut einen bestimmten Wert annehmen. Zum Setzen von Attributen bieten wir die Methode `setAttribute(char *attrID)` an, zum Setzen von Attributwerten `setAttributeValue(char *value)`.

Ein Beispiel: Wir definieren an jedem in der `construct`-Sektion erzeugten Link ein Attribut `userComesFrom`. Das kann zwei Werte tragen, einmal `UniDo` und `guest`. Der Link, der von `aboutMe` zu `Pic` führt, erhält den Wert `UniDo`, während alle anderen Links den Wert `guest` bei diesem Attribut tragen. In der `browsing`-Sektion können wir nun einen Feature-Term wie `userComesFrom: guest` notieren. Wird dieser Term ausgewertet, dann erhält man alle Links, die beim Attribut `userComesFrom` den Wert `guest` tragen. Das sind alle Links, bis auf den Link, der zu `Pic` führt. Dieser Link wird daher nicht generiert, das Bild kann also nicht erreicht werden. Hätten wir den Feature-Term `userComesFrom: {UniDo, guest}` angegeben, wären alle Links generiert worden.

Mit diesem Konzept lassen sich also statische Sichten auf das erzeugte Hyperdokument realisieren. Diese Sichten sind offensichtlich als eine Teilmenge des insgesamt erzeugbaren Hyperdokuments definiert und durch geeignetes Verändern eines Feature-Terms an individuelle Bedürfnisse anpaßbar, ohne den Rest der Spezifikation zu betreffen.

Geht man davon aus, daß in der Regel Links mit mehreren Attributen versehen werden, dann ist dieses Verfahren recht mächtig.

Spezifikation 1.7 zeigt die Umsetzung: die durch `getOcc` erzeugten Links werden durch `setAttribute` mit `userComesFrom` attributiert und diesem der Wert `UniDo` bzw. `guest` zugewiesen. Die `browsing`-Sektion zeigt einen Feature-Term, der sich wie im Beispiel beschrieben verhält. Nicht attributierte Links werden nicht durch den Feature-Term selektiert. Daher konnten wir auf die Attributierung der im Dekorationsteil erzeugten Links verzichten. Sie sollen (und sinnvollerweise müssen sie es) stets erzeugt werden.

```
class SimpleHyperDoc is
  declare class Decoration is ... end Decoration;
  documents ... ;
  construct
    void main(void) { // das Hauptprogramm

      // wie gehabt dekorieren
      deco[0].installDeco(homepage);
      deco[1].installDeco(aboutMe);
      deco[2].installDeco(Pic);

      // wie gehabt getOcc anwenden, Links attributieren und belegen
      homepage.getOcc(reference).setLink(deco[1].greeting.getBegin()).
        setAttribute(userComesFrom).setValue(guest);
      aboutMe.getOcc("Uni")[1].setLink(deco[2].greeting.getBegin()).
        setAttribute(userComesFrom).setValue(UniDo);
    }

  browsing
    userComesFrom: guest

end SimpleHyperDoc;
```

Quellcode 1.7: Die `browsing`-Sektion

1.2.7. Generische Klassen

Es ist relativ einfach, in $DoDL_C$ parametrisierte Klassen zu erzeugen. Dazu existiert das Schlüsselwort `generic`, das einfach nur vor eine Klassendefinition geschrieben wird. Hinter dem Klassennamen folgt in eckigen Klammern ein Parameter, der in der Klasse als Typ verwendet werden darf. Spezifikation 1.8 zeigt die Klasse `SimpleHyperDoc` parametrisiert.

Der Parameter wird in der Definition der Variable `Pic` verwendet. Hier soll erst bei der Verwendung der Klasse entschieden werden, ob `Pic` ein Bild oder ein Text ist. Dazu erweitern wir die Spezifikation 1.5 um zwei Klassen `InstHyperDocText` und `InstHyperDocGraphics`. Diese dienen als „Zwischenklassen" und werden in der Klasse `InstSimple` als Typ verwendet. Diese ist nur zur Instanziierung von `InstHyperDocText` resp. `InstHyperDocGraphics` gedacht, die von `SimpleHyperDoc` abgeleitet sind und den Variablentyp der Klasse konkretisieren. Ihre `binding`-Sektion (Quellcode 1.9) ist dann eine Erweiterung von Quellcode 1.6. Der hier für `Pic` angegebene Typ muß dann allerdings mit dem in der Klasse `InstSimple` übereinstimmen!

```
generic class SimpleHyperDoc[TextOrGraphic] is
   declare class Decoration is ... end Decoration;

  documents homepage, aboutMe : text;
            reference         : string;
            Pic               : TextOrGraphic; // hier der variable Typ
            deco              : list of Decoration;
   construct
     void main(void) {   // das Hauptprogramm bleibt unveraendert
        ...
     }
end SimpleHyperDoc;

/* Zwei Instanzen der generischen Klassen: */
 class InstHyperDocText is SimpleHyperDoc[text] with
end InstHyperDoc;

 class InstHyperDocGraphics is SimpleHyperDoc[graphics] with
end InstHyperDocGraphics;

 class InstSimple is
  documents textInst  : InstHyperDocText;      // eine Text-Instanz
            graphInst : InstHyperDocGraphics;  // eine Graphik-Instanz

end InstSimple;
```

Quellcode 1.8: Eine parametrisierte Klasse

1.3. Klassenhierarchien: Vererbung in $DoDL_C$

Was wäre eine objektorientierte Sprache ohne Vererbung? In $DoDL_C$ wird die einfache Erbung propagiert, Mehrfacherbung ist nicht möglich.

Syntaktisch wird eine erbende Klasse recht einfach aufgeführt, man muß lediglich die Kopfzeile erweitern. Nehmen wir wieder mal die lokale Klasse `Decoration` aus Spezifikation 1.5 her und definieren eine davon erbende Klasse `FowardDecoration` (Spezifikation 1.10). Semantisch gesehen werden sämtliche Attribute und Methoden aus der Klasse `Decoration` übernommen, sprich die `document`-Sektion und die `construct`-Sektion werden in die erbende Klasse kopiert.

```
binding InstSimple is
  in textInst assign
    homepage  = "~/texte/Welcome.txt";
    aboutMe   = "~/texte/Info.txt";
    reference = "Arbeit";
    Pic       = "~/pics/Uni.txt";    // Textdatei, kein Bild!!
    in deco assign ... end;
  end;
  in graphInst assign
    homepage  = "~/texte/Welcome.txt";
    aboutMe   = "~/texte/Info.txt";
    reference = "Arbeit";
    Pic       = "~/pics/Uni.jpg";    // hier das Bild!!
    in deco assign ... end;
  end;
end;
```

Quellcode 1.9: Binding für generische Klassen

Was aber geschieht, wenn die Subklasse ihrerseits neue Dokumente und Methoden definiert? Das ist zunächst kein Problem, wenn man weiß, daß $DoDL_C$ neu definierte Attribute und Methoden einfach hinzunimmt. Dabei sollten hinzukommende Attribute Namen besitzen, die in der Superklasse noch nicht verwendet wurden (sonst erhält man den Hinweis, daß etwas doppelt definiert wurde). Wird eine Methode in der Subklasse definiert, die in der Superklasse ebenfalls definiert ist, so wird die Methode der Subklasse diese verschatten, wenn die Signaturen der beiden Methoden identisch sind. Sind die Signaturen nicht identisch, liegt ein Fehler vor. Dabei beachten wir Polymorphien nicht.

In $DoDL_C$ ist also die *Redefinition* von Methoden realisiert, anders als die Realisierung der additiven Erbung in der ursprünglichen PROLOG-Variante von *DoDL*, wo bei Signaturgleichheit von Methoden die Methode der Superklasse übernommen und in der Subklasse um die dort definierten Anweisungen erweitert wird. Will man aber explizit auf eine namensgleiche Methode aus der Superklasse zugreifen, so verwendet man dazu das `super`-Konstrukt. Hierbei ist zu beachten, daß in diesem Anweisungsblock der Scope der Superklasse gilt! Der Vorteil dieser Variante liegt in ihrer sehr einfachen Handhabung. Sowohl das additive Prinzip aus PROLOG als auch das `inner`-Konstrukt aus BETA können hiermit realisiert werden.

Expandiert man etwa in Spezifikation 1.10 die Klasse `ForwardDecoration` (Spezifikation 1.11), dann sieht man die hinzugekommenen Attribute und redefinierten Methoden genauer. Das `super`-Konstrukt ist ebenfalls aufgelöst.

Vererbung kann übrigens auch auf generischen Klassen angewendet werden. Eine von einer generischen Klasse erbende Klasse kann ihrerseits wieder generisch sein. Das sieht im schlimmsten Fall dann so aus (Spezifikation 1.12): In Spezifikation 1.8 lassen wir `InstSimpleText` von der generischen Klasse `SimpleHyperDoc[TextOrGraphic]` erben und führen dabei einen Klassenparameter `mayBeGraphic` ein. Damit lassen sich dann Instanzen von `InstSimpleText` bilden, die, wie aus Spezifikation 1.8 bekannt, benutzt werden können. In `InstSimpleText` wird der formale Parameter der Klasse `SimpleHyperDoc` wegen der Erbung übernommen. Durch den aktuellen Parameter `text` wird er aber eindeutig ersetzt. Die Klasse `InstSimpleGraphic`

```
class SimpleHyperDoc is
  declare class Decoration is
            documents greeting, go_home: dbUnit;
             construct
               void setGreeting(dbUnit page) { ... }
               link setHomeRef(dbUnit page)  { ... }
               void installDeco(dbUnit page) { ... }
          end Decoration;

           class ForwardDecoration is Decoration with
          // eine Subklasse von Decoration}
            documents next: dbUnit; // next kommt hinzu
             construct
               void setForwardRef(dbUnit page) {
                 /* eine neue Methode zur Verlinkung:
                   Am Seitenende der Startseite verweisst ein Link auf
                   eine Seite page */
                 homepage.getEnd().setLink(page.getBegin());
               }

               void installDeco(dbUnit page) { // Redefinition
                 super.installDeco(page); // die alten Dekorationen
                 setForwardRef(page);
               }

          end ForwardDecoration;

  /* Der Rest ist klar, ForwardDecoration kann jetzt wie Decoration
     benutzt werden */
  documents ... ;
   construct ... ;
end SimpleHyperDoc;
```

Quellcode 1.10: Eine erbende Klasse

```
class SimpleHyperDoc is
  declare class Decoration is
            documents greeting, go_home: dbUnit;
             construct
               void setGreeting(dbUnit page) { ... }
               link setHomeRef(dbUnit page)  { ... }
               void installDeco(dbUnit page) { ... }
          end Decoration;

          class ForwardDecorationExpanded is // expandierte Version:
                                             // ohne Vererbung
            documents greeting, go_home, next: dbUnit;

             construct
               void setGreeting(dbUnit page) { ... } // aus Decoration
               link setHomeRef(dbUnit page)  { ... } // aus Decoration
               void setForwardRef(dbUnit page) {     // die neue Methode
                 homepage.getEnd().setLink(page.getBegin());
               }

               void installDeco(dbUnit page) { // die Redefinition
                 /* die beiden alten Anweisungen aus installDeco der
                    Klasse Decoration, die per super aufgerufen wurden */
                 setGreeting(page);
                 setHomeRef(page);
                 setForwardRef(page); // die neue Anweisung
               }
          end ForwardDecoration;

  // Der Rest ist wie gehabt
  documents ... ;
   construct ... ;
end SimpleHyperDoc;
```

Quellcode 1.11: Eine expandierte Klasse

ist eine Subklasse von `InstSimpleText` und damit eine Subklasse der generischen Klasse `SimpleHyperDoc`. Damit hat auch `InstSimpleGraphics` in der `document`-Sektion den Typ `TextOrGraphic`, ersetzt ihn aber mit `graphics`.

```
generic class InstSimpleText[mayBeGraphic] is SimpleHyperDoc[text] with
  documents textInst: TextOrGraphic; // hat den Typ text
end InstSimpleText;

class InstSimpleGraphic is InstSimpleText[graphics] with
  documents graphInst: TextOrGraphic; // hat den Typ graphics
end InstSimpleGraphic;
```

Quellcode 1.12: Eine von einer generischen Klasse erbende generische Klasse

2. Scoping-Regeln

Mit einem Satz könnte man sagen, daß $DoDL_C$ im Hinblick auf die Sichtbarkeit von Namen den ALGOL-Regeln [Nau63, May80] folgt. Damit wird ein Kontur-Modell etabliert, das wohlbekannt ist und nicht weiter diskutiert werden muß, wäre da nicht der Zeitgeist am Werk und ließe $DoDL_C$ eine objektorientierte Sprache sein. Im Gegensatz zu ALGOL gibt es hier Instanzen, auch lokale Klassen im Unterschied zu lokalen Prozeduren, die in objektorientierten Sprachen auch außerhalb der sie definierenden Klassen instanziiert werden können. Letztlich gibt es noch Vererbung und damit einen, sagen wir „Transport“ von Sichtbarkeitsbereichen. In ALGOL ist das undenkbar. Es lohnt also, genauer hinzuschauen und ein paar Fälle abzuklopfen, um Klarheit über die Sichtbarkeitsregeln in $DoDL_C$ zu schaffen.

2.1. Klassen auf gleicher Ebene

Werden zwei Klassen auf der gleichen Ebene definiert, wie das zum Beispiel Spezifikation 1.8 mit den Klassen `SimpleHyperDoc` und `InstSimple` zeigt, dann können später deklarierte Klassen die früher deklarierten instanziieren. Das Prinzip *declaration before use* wird damit in $DoDL_C$ eingehalten. Die Klasse `InstSimple` instanziiert zweimal die Klasse `SimpleHyperDoc`. Sie liegt im Sichtbarkeitsbereich von `InstSimple` und ist vor ihr deklariert.

Für die lokalen Klassen innerhalb einer `declare`-Sektion gilt natürlich das gleiche Prinzip. Zusätzlich müssen aber ein paar weitere Restriktionen beachtet werden, was die Sicht auf lokale Klassen resp. die Sicht lokaler Klassen auf umgebende Klassen betrifft.

2.2. Lokale Klassen

Lokale Klassen innerhalb einer `document`-Sektion können sich ebenfalls wie oben bereits beschrieben instanziieren.

Eine lokale Klasse sieht aber auch diejenige Klasse, in der sie deklariert ist sowie alle Klassen in deren Sichtbarkeitsbereich, die vor ihr deklariert wurden. In Spezifikation 1.8 wäre die lokale Klasse `Decoration` durchaus in der Lage, eine Instanz einer vor `SimpleHyperDoc` deklarierten Klasse zu definieren.

Lokale Klassen werden jedoch nur von der sie beinhaltenden Klasse gesehen. `InstSimple` sieht zwar `SimpleHyperDoc`, nicht jedoch deren lokale Klasse `Decoration`.

Eine lokale Klasse kann von außen jedoch indirekt instanziiert werden. Es gibt einen „Trick“, der auch beispielsweise in BETA und JAVA verwendet wird. Da `InstSimple` eine Instanz `textInst` von `SimpleHyperDoc` definiert hat, kann `InstSimple` nun über `textInst` an eine Instanz von `Decoration` herankommen:

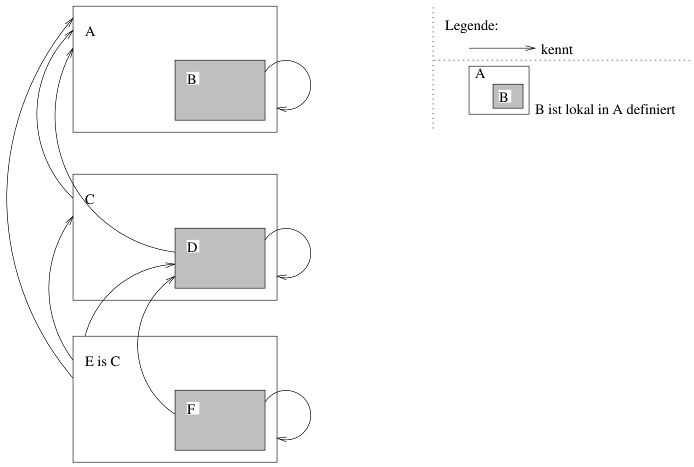

Abbildung 2.1.: Das $DoDL_C$-Kontur-Modell

```
textDeco :  textInst . Decoration ;
```

liefert eine Instanz von `Decoration`, die stets vom aktuellen Zustand von `textInst` abhängig ist, weil sie in ihm eingebettet ist. Verändert `textInst` seinen Zustand nach der Instanzierung von `textDeco`, wird `textDeco` diese Veränderung seiner Umgebung mitbekommen.

2.3. Erbende Klassen

Bei erbenden Klassen, so kann man feststellen, werden nicht nur die Variablen der `document`-Sektion und die Methoden der `construct`-Sektion auf die Subklassen übertragen, sondern auch der Sichtbarkeitsbereich. Alles, was in der Superklasse sichtbar ist, ist auch in der Subklasse sichtbar.

Interessant wird es, wenn eine Klasse mit lokalen Klassen von einer anderen Klasse mit lokalen Klassen erbt. Dann nämlich können die lokalen Klassen der Subklasse als Subklassen der lokalen Klassen der Superklasse definiert werden.

2.4. Zusammenfassung

Die einzelnen Fälle haben wir in einem $DoDL_C$-Kontur-Modell erfaßt, das in Abbildung 2.1 zu sehen ist.

Kurz in Punkten zusammengefaßt hat $DoDL_C$ die folgenden Eigenschaften, aus denen das Kontur-Modell abgeleitet wird: $DoDL_C$ verhält sich wie gängige objektorientierte Programmiersprachen, die

1. *declaration before use* einhalten, also mit single-pass-Compilern realisierbar sind und

2. bei denen alle Komponenten einer Klasse *public*, also bis auf Konturen uneingeschränkt nutzbar sind.

Für das Kontur-Modell gilt dann:

1. Klassen auf gleicher Ebene sehen sich gemäß *declare before use*;
2. lokale Klassen sehen ihre Umgebung gemäß *declare before use* und alles, was diese sieht;
3. lokale Klassen können von außen nur über ihre Umgebung erreicht werden;
4. bei erbenden Klassen wird der Sichtbarkeitsbereich der Superklasse auf die Subklasse übertragen.

3. Die $DoDL_C$-Grammatik

Abschließend wollen wir die Grammatik von $DoDL_C$ im Überblick betrachten. Dazu zeigen wir noch einmal alle Schlüsselwörter und deren Verwendung. Die Grammatik selbst geben wir dann in Form von Syntaxdiagrammen an. Diese gestalten die Arbeit mit $DoDL_C$ übersichtlicher als eine äquivalente EBNF.

3.1. Schlüsselwörter

Die Schlüsselwörter von $DoDL_C$ und ihre Verwendung können durch Tabelle 3.1 entschlüsselt werden.

Neben den Schlüsselwörtern gibt es noch ein einige Sonderzeichen. Ihre Verwendung läßt sich am einfachsten aus den Beispielspezifikationen ableiten. Hieran ist wirklich nichts Ungewöhnliches oder gar Geheimnisvolles zu entdecken.

3.2. Bezeichner

Bezeichner werden in $DoDL_C$ wie anderswo aus

- den Buchstaben `a..z` und `A..Z`,
- den Ziffern `0..9` und
- dem Unterstrich

gebildet. Ausgeschlossen werden alle weiteren Sonderzeichen. Außerdem beginnen Bezeichner immer mit einem Buchstaben oder dem Unterstrich.

3.3. Syntaxdiagramme

Abbildungen 3.1, 3.2 und 3.3 zeigen die Syntaxdiagramme von $DoDL_C$. Sie bleiben hier zugunsten der Übersicht unkommentiert.

Name	Inhalt
`class`	leitet eine Klassendefinition ein;
`is`	folgt einem Klassennamen in der Klassendefinition und der Bindingdefinition;
`with`	schließt den Kopf einer Subklassendefinition ab;
`generic`	leitet eine parametrisierte Klassendefinition ein;
`declare`	leitet die Sektion zur Definition lokaler Klassen ein; hier können wieder vollständige Klassenhierarchien deklariert werden;
`documents`	leitet die Sektion zur Deklaration von Variablen ein; mögliche Typen sind die vordefinierten Typen sowie selbstdefinierte Klassen und Listen all dieser Typen; die Wertebelegung erfolgt in der `binding`-Sektion;
`list of`	deklariert eine Liste mit Elementen des angebenen Typs; die Listengröße wird in der `binding`-Sektion definiert;
`construct`	leitet die Sektion zur Definition von Methoden der Klasse ein; die Methoden werden im Fall von $DoDL_C$ in C implementiert;
`browsing`	leitet einen Feature-Term ein; dieser dient als Selektionsregel für Links, die in der `construct`-Sektion durch Methoden erzeugt werden können;
`end`	schließt eine Klassendefinition, eine `binding`-Sektion oder ein `in`-Assigment der `binding`-Sektion ab;
`binding`	leitet die Definition einer `binding`-Sektion ein;
`in`	leitet die Zuweisung von Werten für einen zusammengesetzten Typ (Klasseninstanz oder Liste) ein (sog. `in`-Assignment);
`assign`	schließt die Einleitung eines `in`-Assingments ab;

Tabelle 3.1.: Schlüsselwörter von $DoDL_C$

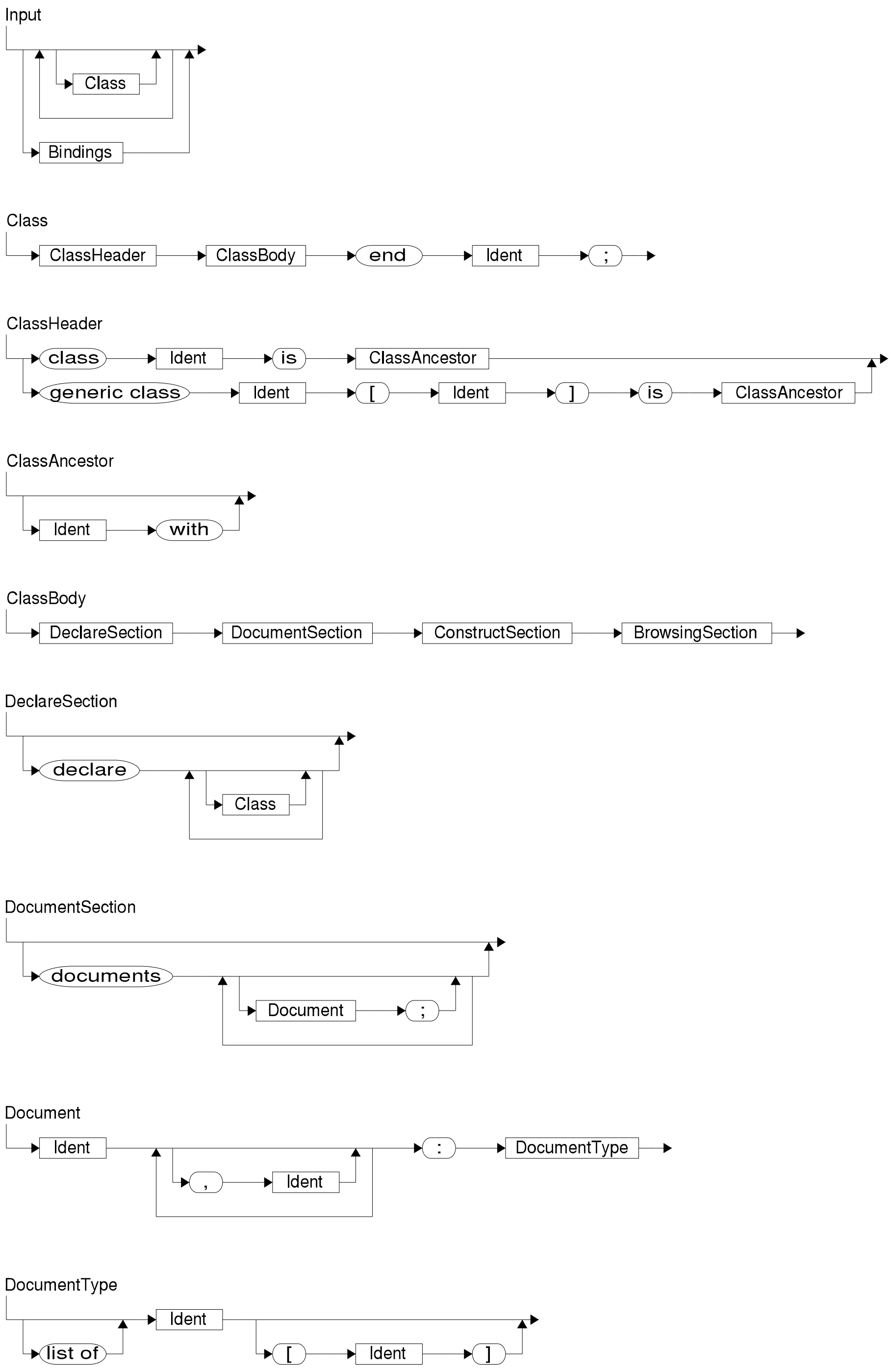

Abbildung 3.1.: Syntaxdiagramm der Sprache $DoDL_C$ (Teil 1 von 3)

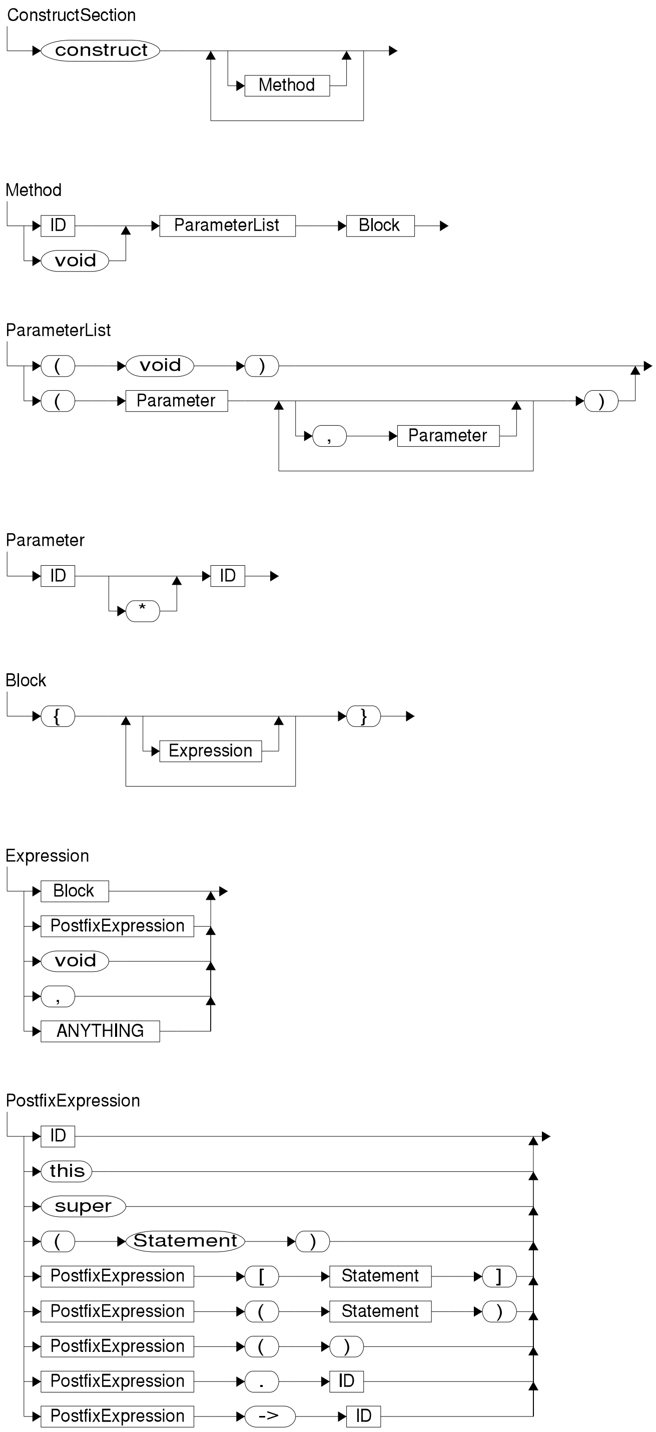

Abbildung 3.2.: Syntaxdiagramm der Sprache $DoDL_C$ (Teil 2 von 3)

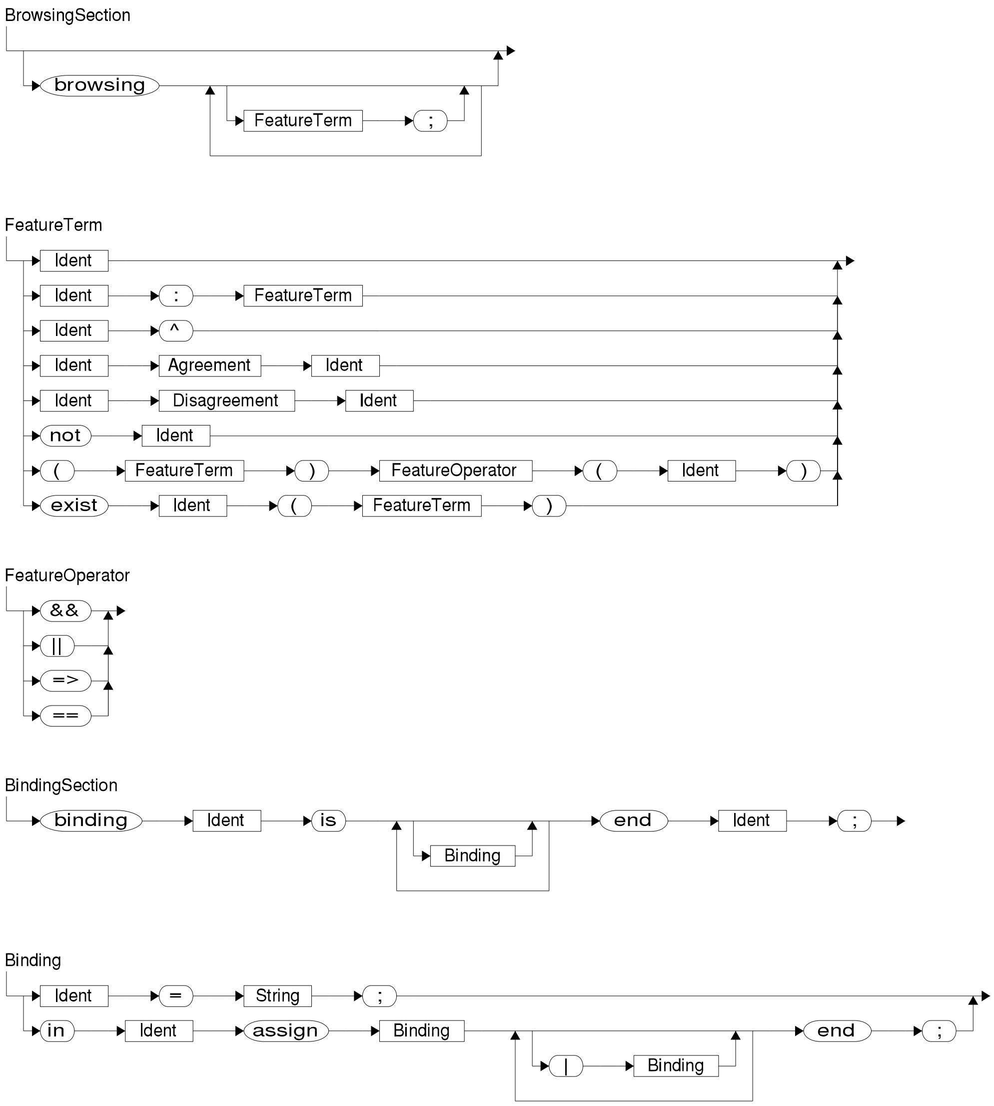

Abbildung 3.3.: Syntaxdiagramm der Sprache $DoDL_C$ (Teil 3 von 3)

Teil II.

Benutzungshandbuch

4. Installation

Jede Software, die etwas auf sich hält, will installiert werden, bevor sie benutzt werden kann. Das gilt natürlich auch für den $DoDL_C$-Compiler. Dieses Kapitel beschreibt den Aufbau der verwendeten Verzeichnisstruktur, das Anpassen der Systemkonfiguration an den Compiler sowie das Übersetzen und den anschließenden Test des Compilers für eine spezielle Plattform.

4.1. Aufbau der Verzeichnisstruktur

Der $DoDL_C$-Compiler setzt sich aus einer Vielzahl von Dateien und Unterverzeichnissen zusammen. Er sollte deshalb am besten in einem eigenen Verzeichnis installiert werden, dessen Name frei wählbar ist. Dieses Verzeichnis wird im folgenden als das *Basisverzeichnis* des $DoDL_C$-Compilers bezeichnet.

Unabhängig davon, ob der $DoDL_C$-Compiler aus einem `ZIP`-Archiv entpackt oder direkt einem CVS-Repository entnommen wird, sollte sich die in Tabelle 4.1 dargestellte Verzeichnisstruktur innerhalb des Basisverzeichnisses wiederfinden.

Diese Verzeichnisstruktur ist als fest zu betrachten. Es können zwar neue Verzeichnisse hinzugefügt werden, jedoch sollte keines der angegebenen Verzeichnisse gelöscht oder umbenannt werden, da dann der $DoDL_C$-Compiler mit hoher Wahrscheinlichkeit nicht mehr ordnungsgemäß funktionieren wird.

4.2. Übersetzen des Compilers

Der Compiler wird im Quellcode weitergegeben. Er muß für die Plattform, auf der er eingesetzt werden soll, übersetzt werden, bevor er tatsächlich verwendet werden kann. Dazu werden folgende Werkzeuge benötigt:

- Ein ANSI-C-Compiler[KR90], zum Beispiel der GNU C-Compiler `gcc`[Sta98]. Dieser Compiler wird nicht nur zum Übersetzen des Compilers benötigt, sondern auch zu dessen Laufzeit. Der $DoDL_C$-Compiler ruft den C-Compiler auf, um temporär erzeugten C-Quellcode in ein lauffähiges Programm zu übersetzen.

- GNU `flex`[Pax95] zum Übersetzen des Scanner-Teils des Compilers in C-Code. Möglicherweise kann stattdessen auch Unix `lex` benutzt werden; dies wurde jedoch nicht getestet.

- GNU `bison`[DS95] zum Übersetzen des Parser-Teils des Compilers in C-Code. Möglicherweise kann stattdessen auch Unix `yacc` benutzt werden; dies wurde jedoch nicht

Name	Inhalt
`bin`	Enthält die für die Ausführung des Compilers benötigten Dateien. Dieses Verzeichnis ist anfänglich leer. Das `makefile` kopiert während der Übersetzung des Compilers die notwendigen Dateien dorthin.
`doc`	Enthält die Dokumentation des Compilers.
`samples`	Enthält eine Reihe von Spezifikationen zur Demonstration und zum Testen des Compilers. Die Spezifikationen sind über verschiedene Unterverzeichnisse von `samples` verteilt.
`src`	Enthält den Quellcode des Compilers. Kapitel 7 beschreibt die Struktur des Quellcodes und die einzelnen Dateien im Detail.
`tmp`	Enthält temporäre Dateien. Dieses Verzeichnis ist anfänglich leer. Bei jedem Lauf des Compilers werden hier einige temporäre Dateien angelegt.

Tabelle 4.1.: Verzeichnisstruktur des $DoDL_C$-Compilers

getestet.

- GNU `make`[SM98] oder ein kompatibles Programm zum Verarbeiten des `makefile`. Das Programm muß in der Lage sein, die Präprozessor-Befehle `ifdef`, `else` und `endif` nach der Syntax von GNU `make` zu verstehen. Anderenfalls muß das `makefile` entsprechend angepaßt werden.

Diese Werkzeuge sind auf den meisten Unix-Systemen direkt verfügbar; für viele andere Plattformen existieren Portierungen. Der $DoDL_C$-Compiler wurde auf folgenden Systemen entwickelt und getestet:

- Auf den SunOS/Solaris-Rechnern des Lehrstuhls für Software-Technologie der Universität Dortmund unter Verwendung des GNU C-Compilers `gcc`.
- Auf einem Standard-PC unter IBM OS/2 Warp 4.0 unter Verwendung der EMX-Portierung des gleichen Compilers.

Das `makefile` erkennt diese Systeme automatisch und verhält sich entsprechend.
Zum Übersetzen des $DoDL_C$-Compilers sind die folgenden Befehle auszuführen:

```
cd src
make
cd ..
```

Wenn der Compiler zu einem späteren Zeitpunkt erneut übersetzt wird und dabei ein Compilieren aller beteiligten Dateien erzwungen werden soll, ist stattdessen `make all` auszuführen. Der Befehl `make clean` löscht temporäre Dateien, die während der Übersetzung erzeugt wurden, `make shred` entfernt zusätzlich auch die erzeugten Zieldateien. Bei geplanten Änderungen am `makefile` oder am Quellcode des Compilers sollte zunächst Kapitel 7 konsultiert werden.

4.3. Anpassen der Systemkonfiguration

Damit der Compiler korrekt funktioniert, ist es notwendig, eine Umgebungsvariable namens `DODL` mit dem Namen des Basisverzeichnisses zu belegen. Wird die Umgebungsvariable nicht gesetzt, kann der Compiler einige zur Laufzeit benötigte Dateien nicht finden bzw. er wird seine temporären Dateien nicht oder an einer falschen Stelle ablegen. Außerdem sollte das Verzeichnis `bin` in den Suchpfad des Systems aufgenommen werden, damit der Compiler von einer beliebigen Stelle aus gestartet werden kann.

Auf einem Unix-System werden zu diesem Zweck zum Beispiel der Datei `.bashrc` folgende Befehle hinzugefügt:

```
set DODL <Basisverzeichnis>
set path=( $path <Basisverzeichnis>/bin )
```

Bei Verwendung des $DoDL_C$-Compilers unter OS/2 sind stattdessen folgenden Zeilen in die Datei `config.sys` aufzunehmen:

```
set DODL=<Basisverzeichnis>
set PATH=%PATH%;<Basisverzeichnis>\bin
```

Anschließend ist eine Neuanmeldung am System (unter Unix) bzw. ein Neustart des gesamten Systems (unter OS/2) notwendig, damit die Änderungen wirksam werden.

4.4. Testen des Compilers

Für einen ersten einfachen Test des Compilers wird dieser zunächst ohne jegliche Parameter aufgerufen:

```
DoC
```

Auf dem Bildschirm sollten Informationen zu Version und Verwendung des Programms anzeigt werden. Falls das Programm nicht gefunden werden kann, wurde der Pfad nicht oder falsch in die Systemkonfiguration eingetragen. Dies ist entsprechend zu ändern. Falls das Programm ausgeführt wird, aber keine Ausgabe erzeugt, hat die Umgebungsvariable `DODL` keinen oder einen falschen Wert. Auch dies sollte den Anweisungen des vorangehenden Abschnitts entsprechend korrigiert werden.

Falls der Compiler unter einem Unix-System benutzt wird, kann nun beispielsweise folgender Befehl ausgeführt werden:

```
cd samples/hello
DoC hello
./hello
```

Bei Verwendung von OS/2 ist stattdessen folgendes einzugeben:

```
cd samples\hello
DoC hello
hello
```

Auf dem Bildschirm sollten die Worte *„Hello, DoDL world!“* erscheinen. Falls das so ist, können wir uns gratulieren, denn wir haben soeben unsere erste $DoDL_C$-Spezifikation erfolgreich übersetzt und ausgeführt.

5. Verwendung

Dieses Kapitel beschreibt die Verwendung des $DoDL_C$-Compilers. Es erläutert, welche Dateien der Compiler als Eingabe erwartet, wie er aufgerufen wird, was während der Verarbeitung geschieht und welche Ausgabedateien am Ende erzeugt werden. Es richtet sich an Leser, die den Compiler benutzen wollen, aber kein Interesse an technischen Details des Systems haben.

5.1. Eingabe

Der $DoDL_C$-Compiler verarbeitet Spezifikations- und Bindungsdateien, die der durch die Sprache $DoDL_C$ vorgegebenen Grammatik entsprechen. Teil I dieses Buches sowie die Veröffentlichungen [Dob96b, Dob96a] beschreiben diese Sprache im Detail. Die Syntaxdiagramme in den Grafiken 3.1 bis 3.3 auf den Seiten 27 bis 29 geben die Grammatik der Eingabedateien des $DoDL_C$-Compilers in komprimierter Form graphisch wieder. Es handelt sich bei diesen Diagrammen um eine vereinfachte Version der tatsächlichen Grammatik des Compilers. Die Diagramme entsprechen einer EBNF-Darstellung der Grammatik, die sich besser zur Veranschaulichung der Sprache eignet, während der Parser bzw. das verwendete Werkzeug `bison` auf einer (äquivalenten) Darstellung der Grammatik in purer BNF basiert, die zudem in LALR(1) liegt.

Wie aus den Syntaxdiagrammen zu ersehen ist, handelt es sich bei den Eingabedateien des Compilers um Textdateien. Als Zeilenende werden sowohl die Kombination `<cr> <lf>` als auch ein einzelnes `<lf>` erkannt. Zum Abschluß der Datei kann sowohl ein `<ctrl> + <d>` als auch ein `<ctrl> + <z>` werden werden. Mit anderen Worten: Der Compiler verarbeitet sowohl Unix- als auch DOS-artige Dateien.

5.2. Aufruf

Der Compiler wird im Normalfall mit drei Parametern gestartet: Der erste gibt den Dateinamen der zu compilierenden Spezifikation an, der zweite den Namen der Datei, in welcher die zu verwendenden Bindungen enthalten sind. Der dritte Parameter gibt den Namen der Ausgabedatei an, die vom Compiler erzeugt wird.

Für den Fall, daß die Eingabedateien `klassen.spec` und `dateien.bind` heißen und die Ausgabedatei `ausgabe.exe` erzeugt werden soll, sähe der Aufruf also wie folgt aus:

```
DoC klassen.spec dateien.bind ausgabe.exe
```

Besitzen die beiden Eingabedateien Namen, die sich nur im Suffix unterschieden, und soll die Ausgabedatei ebenfalls diesen Namen (mit dem für das Betriebssystem üblichen Suffix, also etwa `.exe` unter OS/2) tragen, dann kann auch ein kürzerer Aufruf mit nur einem Parameter verwendet werden. Der folgendende Aufruf verarbeitet zum Beispiel die beiden Dateien `test.spec` und `test.bind` und erzeugt als Ausgabe ein lauffähiges Programm `test` (bzw. `test.exe` unter OS/2):

```
DoC test
```

Ein Aufruf des Compilers ohne jegliche Parameter zeigt einen Hilfebildschirm an.

5.3. Verarbeitung

Abbildung 5.1 zeigt, welche Schritte der $DoDL_C$-Compiler während des Compilierungs-Prozesses ausführt und welcher Datenfluß von und zu den einzelnen am Prozeß beteiligten Dateien stattfindet. Abgerundete Rechtecke symbolisieren Aktionen des Compilers, durchgezogene Pfeile stehen für Kontrollfluß. Spitze Rechtecke repräsentieren Ein- und Ausgabedateien, gestrichelte Pfeile geben entsprechend den Datenfluß im System wieder.

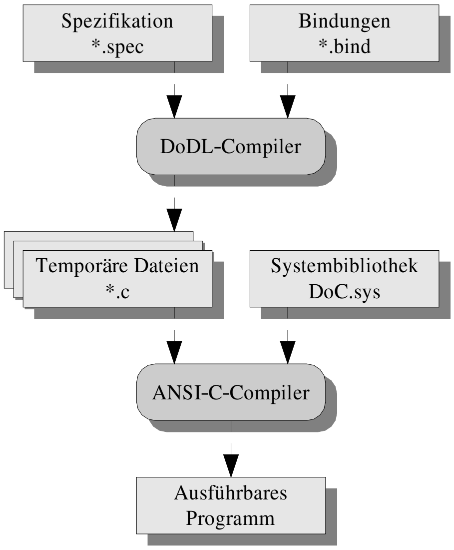

Abbildung 5.1.: Der Compilierungsprozeß

5.4. Ausgabe

Ausgabe des Compilers ist – einen fehlerfreien Durchlauf vorausgesetzt – ein ausführbares Programm, sozusagen eine „ausführbare $DoDL_C$-Spezifikation“. Dieses Programm erzeugt, wenn es aufgerufen wird, als Ausgabe das gewünschte, durch die Spezifikation beschriebene Hyperdokument beziehungsweise eine graphorientierte Repräsentation desselben. Diese Repräsentation, die nicht an ein spezielles Format für Hyperdokumente (wie etwa HTML) gebunden ist, muß in einem abschließenden Schritt von einem weiteren Werkzeug in das endgültige Hyperdokument (also etwa eine Sammlung von HTML-Dateien) überführt werden.

Teil III.

Technisches Handbuch

6. Architektur des Compilers

Dieses Kapitel gibt einen Überblick über die Architektur des $DoDL_C$-Compilers. Es richtet sich an Leser, die über die reine Benutzung hinaus das Innenleben des Compilers verstehen möchten, möglicherweise um Änderungen an dessen Quellcode vorzunehmen.

Insbesondere werden in diesem Kapitel die Vorgänge in der Symboltabelle erläutert. Die üblichen Techniken zum Umgang mit Symboltabellen sind zwar wohlbekannt, wir tragen jedoch den Feinheiten des speziellen Falles $DoDL_C$ Rechnung und zeigen die Arbeitsweise unserer Implementierung. Dies ist nicht zuletzt deshalb interessant, weil die Literatur häufig auf Symboltabellen imperativer Sprachen eingeht, aber objektorientierte Sprachen und deren besondere Probleme, etwa Vererbung und generische Klassen, gern außer Acht läßt.

Die Implementierung der Symboltabelle ist auch eng mit der Code-Generierung verknüpft. Deren Konzepte werden für verschiedene grundlegende Sprachelemente von $DoDL_C$ anhand eines Stück für Stück wachsenden Beispiels aufgezeigt. Dieses Beispiel ist in der Distribution des $DoDL_C$-Compilers enthalten und kann somit vom Leser nachvollzogen werden.

6.1. Aufbau

Der $DoDL_C$-Compiler besitzt einen weitgehend kanonischen Aufbau, wie in [ASU92, GJ90, KP84, SJ85, WG84] beschrieben: Er setzt sich aus einem Scanner, einem Parser und einem Emitter zusammen. Diesen drei Teilen liegt als wesentliche Datenstruktur die Symboltabelle zugrunde, in der alle notwendigen Informationen des Übersetzungsprozesses zusammenlaufen.

Der $DoDL_C$-Compiler ist ein *Single Pass Compiler*, es findet also nur ein Durchlauf des Parsers statt. Die Zielsprache des Compilers ist nicht der Assembler-Code irgendeines speziellen Prozessors, sondern ANSI-C[KR90]. Dazu stützt sich der $DoDL_C$-Compiler auf einen entsprechenden C-Compiler ab, den er nach dem erfolgreichen Übersetzen der Eingabedateien in eine Reihe von temporären Dateien mit C-Code automatisch aufruft. Die Grundgedanken, die diesem C-Code zugrunde liegen, sind sehr einfach, der Teufel steckt jedoch – wie wir noch sehen werden – im Detail:

- $DoDL_C$-Klassen werden in Typdeklarationen von C-Strukturen übersetzt. Das Objekt, welches das Hyperdokument repräsentiert, ist eine Instanz einer dieser $DoDL_C$-Klassen. Es wird naheliegenderweise durch eine entsprechende Variablen-Deklaration repräsentiert.

- Dokumente, also Platzhalter für Instanzen von Klassen, werden zu Feldern innerhalb der gerade erwähnten C-Strukturen.

- Methoden, also die C-Funktionen der `construct`-Sektion, bleiben im wesentlichen C-Funktionen, werden jedoch „aufbereitet“. So wird jeder Funktion zum Beispiel der in objektorientierten Sprachen übliche implizite Parameter `this` hinzugefügt, über den die Methode die Instanz erfragen kann, von welcher sie aufgerufen wurde.

- Für jede Klasse wird zusätzlich eine Struktur erzeugt, die Informationen über diese Klasse bereitstellt. Dazu gehören zum Beispiel Name und Vorfahr der Klasse. Wesentlicher Inhalt der Informationsstruktur ist jedoch eine Tabelle aller Methoden der Klasse. Methodenaufrufe finden – unsichtbar für den Benutzer – immer über diese Tabelle statt, wodurch Methodenzuteilung zur Laufzeit erreicht wird.

- Der gesamte erzeugte Quellcode wird über mehrere Dateien verteilt und in einer Reihenfolge erzeugt, die der Abfolge der Arbeitsphasen des Compilers entspricht. Die Teile des Quellcodes werden über `#include` miteinander verbunden. Abbildung 6.1 zeigt schematisch die temporären Dateien und deren wichtigste Beziehungen zueinander.

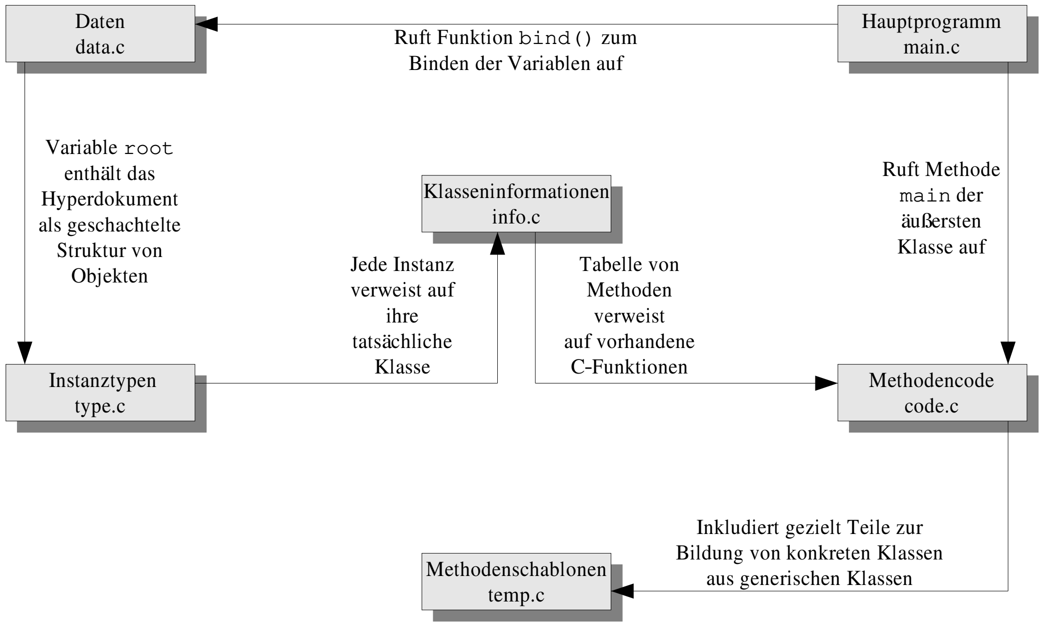

Abbildung 6.1.: Temporäre Dateien im Übersetzungsprozeß

Angesichts des dargestellten Aufbaus und der Abstützung auf den ANSI-C-Compiler steht es dem Leser frei, den $DoDL_C$-Compiler als einen erweiterten Präprozessor (oder Präcompiler) des C-Compilers zu betrachten oder – auch diese Sichtweise ist möglich – den C-Compiler als einen (erweiterten) Linker, der die Ausgabedateien des $DoDL_C$-Compilers zu einem ausführbaren Programm bindet.

6.2. Arbeitsweise

Die Arbeitsweise des Compilers läßt sich grob in drei Phasen unterteilen. In jeder Phase wird ein Teil der Ausgabedateien erzeugt.

6.2.1. Phase 1

Phase 1 eines Compiler-Laufes erstreckt sich vom Aufruf des Compilers bis unmittelbar vor den Start des Parsers. Sie beinhaltet die Aktionen, die zur Initialisierung des Compilers notwendig sind. In dieser Phase wird die Symboltabelle in ihren Ursprungszustand versetzt. Außerdem wird die Datei `main.c` erzeugt, die das für alle Spezifikationen identische Hauptprogramm enthält.

6.2.2. Phase 2

Phase 2 eines Compiler-Laufs umfaßt die syntaktische Analyse von Spezifikation und Bindungen. Während der Compiler diese beiden Dateien verarbeitet, füllt er die Symboltabelle mit allen Informationen, die er in ihnen vorfindet. Im wesentlichen sind dies die Deklarationen der einzelnen Klassen mit ihren jeweils lokalen Deklarationen, Dokumenten, etc. Aus dieser komplexen rekursiven Struktur muß nach der syntaktischen Analyse C-Code erzeugt werden, der jede Klasse und deren Instanzen modelliert und repräsentiert. Da die Beziehungen zwischen den Klassen keineswegs trivialer Natur sind (man denke an die Auswirkungen von Vererbung), kann dies nicht während der syntaktischen Analyse geschehen und wird dementsprechend auf einen Zeitpunkt verschoben, an dem der Parser seine Arbeit beendet hat.

Es gibt jedoch zwei Teile der Eingabe, die strukturell bereits so nahe am generierten Zielcode sind, daß ihre Übersetzung während der syntaktischen Analyse erzeugt werden kann: Die `construct`-Sektion enthält C-Code, der – wie zuvor erwähnt – aufbereitet wird. Die `construct`-Sektion jeder Klasse wird also in die entsprechenden Ausgabedateien `code.c` und `temp.c` geschrieben, während die Klasse gelesen wird. Ebenso die `binding`-Sektion: Obgleich diese sich syntaktisch stark vom generierten C-Code unterscheidet, läßt sie sich sehr einfach in eine Reihe von Zuweisungen übersetzen, weshalb dieser Teil der Ausgabe auch bereits beim Durchlauf des Parsers in die Datei `data.c` geschrieben wird. Für die beiden genannten Sektionen wird also nur wenig oder keine Information im Speicher gehalten.

6.2.3. Phase 3

Phase 3 eines Compiler-Laufes findet im Anschluß an die syntaktische Analyse statt, sofern diese erfolgreich war, und erstreckt sich bis zum Ende des Programms. In dieser Phase wird der Teil des Zielcodes erzeugt, der Informationen über alle in der Symboltabelle befindlichen Klassen und deren Eigenschaften benötigt. Erst an dieser Stelle werden also zum Beispiel die Deklarationen für die einzelnen Instanztypen sowie die Klasseninformationen in die Dateien `type.c` und `info.c` geschrieben.

Abbildung 6.2 zeigt schematisch den Daten- und Kontrollfluß innerhalb des Compilers. Rechtecke stellen Ein- und Ausgabedateien dar, Ovale repräsentieren Aktionen des Compilers. Durchgezogene Pfeile symbolisieren den Kontroll-, gestrichelte Pfeile entsprechend den

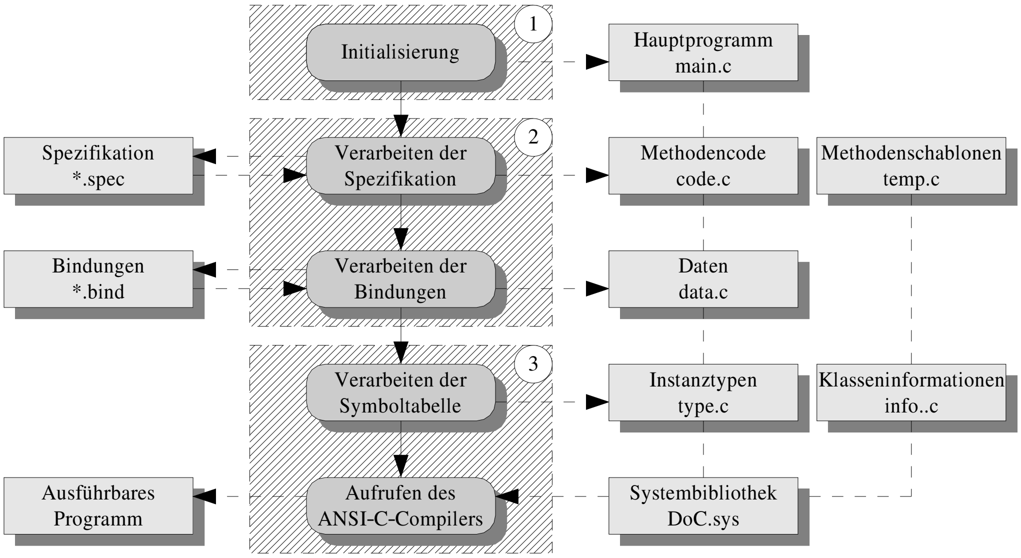

Abbildung 6.2.: Kontroll- und Datenfluß im Compiler

Datenfluß. Die drei Phasen sind jeweils schraffiert hinterlegt. Aus Gründen der Übersichtlichkeit wurde in dieser Grafik darauf verzichtet, den Datenfluß von der und in die Symboltabelle darzustellen. Es ist jedoch leicht einzusehen, daß in den Phasen 1 und 2 große Datenmengen in die Symboltabelle geschrieben werden, die dann in Phase 3 ausgewertet werden.

6.2.4. Vergleich mit der Prolog-Variante

Die ursprüngliche, auf Prolog basierende Variante von *DoDL* konstruierte aus der Klassen- und Instanzenstruktur der Spezifikation einen Baum, dessen Knoten unter anderem mit dem Inhalt der `constraints`-Sektion einer Klasse dekoriert wurden. Diese formulierte die Konstruktion der Verknüpfungsstruktur mithilfe von Fakten und Regeln. Der Baum wurde in Post-Order traversiert, wobei sich ein vollständiges Prolog-Programm ergab, das in eine Datei geschrieben wurde. Ein Prolog-Interpreter sollte dieses Programm ausführen und daraus eine Faktenbasis gewinnen, die das Hyperdokument abstrakt repräsentierte. Ein weiteres Programm sollte diese abstrakte Repräsentation in ein konkretes Hyperdokument, also zum Beispiel eine Menge von HTML-Seiten, umwandeln.

Diese Vorgehensweise konnte aufgrund der Unterschiede zwischen dem allein auf Fakten und Regeln basierenden Prolog und der imperativen Sprache C nur teilweise beibehalten werden. Wirft man einen Blick auf den Quellcode, dann scheint es zudem so, als würde der Parser von $DoDL_C$ überhaupt keine Baumstruktur erzeugen, sondern nur die Symboltabelle füllen und den C-Code bearbeiten, aber das ist nicht der Fall: Die Symboltabelle *ist* die Baumstruktur. Sie ist als Baum implementiert, der exakt die verschachtelte Struktur der einzelnen Klassen, Instanzen etc. wiedergibt. Es besteht also durchaus eine Parallele zur ursprünglichen Vorgehensweise.

Die Prolog-Variante benötigt allerdings keinerlei Datentypen. $DoDL_C$ hingegen definiert – wie bereits erwähnt – für jede Klasse mehrere Strukturen, die voneinander abhängig sind. Nimmt man die Effekte von Vererbung und generischen Klassen hinzu, dann sind diese Strukturen auch über eine Klasse hinaus voneinander abhängig. Es gibt nur sehr wenige Möglichkeiten, die einzelnen Stücke C-Quellcode, die sich aus einer komplexen Spezifikation ergeben, in eine Reihenfolge zu bringen, die der C-Compiler verarbeiten kann – ein einfacher Post-Order-Durchlauf durch die Symboltabelle reicht dazu nicht aus.

Die generelle Vorgehensweise innerhalb von $DoDL_C$ ist die, mehrere Pre- und Post-Order-Durchläufe durch die Symboltabelle durchzuführen und dabei die einzelnen Datenstrukturen zu erzeugen.

6.3. Übersetzung einfacher Klassen

Wir wollen in diesem Abschnitt zunächst Quellcode und Zielcode einer sehr einfachen Klasse unter die Lupe nehmen, bevor wir diese Klasse in den folgenden Abschnitten Stück für Stück ausbauen, um die Übersetzung fortgeschrittener Sprachkonstrukte zu beleuchten.

6.3.1. Quellcode der Spezifikation

Quellcode 6.1 zeigt die erste Spezifikation, mit der wir den $DoDL_C$-Compiler füttern wollen. Die Spezifikation enthält eine Klasse `Book` mit einem Dokument `title`, das den Namen des Buches aufnimmt, und einer Methode `main`, die diesen Namen ausgibt. Der Name wird innerhalb der Bindungen als „The Art of DoDL“ festgelegt.

```
class Book is
  documents
    title : string;

   construct
    void main(void) {
      printf("Contents of book \"%s\":\n", title);
    }
end Book;

binding Book is
  title = "The Art of DoDL";
end;
```

Quellcode 6.1: Spezifikation einer einfachen Klasse

6.3.2. Inhalt der Symboltabelle

Abbildung 6.3 gibt den Inhalt der Symboltabelle nach der syntaktischen Analyse dieser Spezifikation graphisch wieder. Jedes Rechteck stellt einen Eintrag der Symboltabelle und potentiell auch einen eigenen Namensraum dar, da die meisten Symbole über untergeordnete Symbole verfügen können. Symbole, die durch kleinere Rechtecke repräsentiert werden, sind jeweils im

Namensraum des umschließenden Symbols beheimatet. Durchgezogene Pfeile verweisen auf den Typ eines Symbols. Offensichtlich besitzen nicht alle Symbole einen Typ, für Dokumente und für Parameter von Methoden ist er jedoch erforderlich.

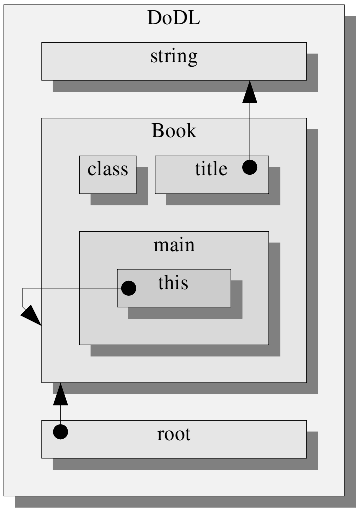

Abbildung 6.3.: Symboltabelle für eine einfache Klasse

Die Grafik zeigt einen äußeren Namensraum mit dem Namen `DoDL`, der in der Spezifikation scheinbar keine Entsprechung besitzt. Dieser Namensraum ist immer vorhanden. Er stellt die Wurzel der Symboltabelle und somit die Begrenzung des $DoDL_C$-Universums dar. Innerhalb dieses äußersten Namensraums finden wir zunächst den vordefinierten Typ `string`. Diesem Typ folgt die Definition der Klasse `Book`, die der Beispiel-Spezifikation entspringt. Innerhalb dieser Klasse finden wir – wie zu erwarten – das Dokument `title` sowie die Methode `main` wieder. `title` ist vom Typ `string`, deshalb existiert ein Pfeil, der auf diesen Typ zeigt.

Die Methode `main` besitzt einen Parameter `this`, den wir ebenfalls nicht deklariert haben. Diese Parameter wird für jede Methode automatisch angelegt. Er verweist jeweils auf eine Instanz der Klasse – der Typ-Pfeil deutet es an – und zwar auf genau die Instanz, welche die Methode aufgerufen hat. Dieser implizit vorhandene `this`-Parameter erlaubt uns in der `construct`-Sektion den Zugriff auf das Feld `title` innerhalb von `main`, ohne daß eine Instanz angegeben werden muß.

Es wäre aber – dies sei hier der Vollständigkeit halber erwähnt – trotzdem möglich gewesen, stattdessen `this.title` zu schreiben. Wir hätten das gleiche Ergebnis erzielt, da der Compiler erkennt, daß mit `title` ein Feld der aufrufenden Instanz gemeint ist, und den Code entsprechend ändert. Genau das war gemeint, wenn in den vorangehenden Abschnitten vom „Aufbereiten“ des Codes der `construct`-Sektion die Rede war.

Die Grafik enthält außerdem ein Symbol `root`, das nicht explizit deklariert wurde. Bei diesem Symbol handelt es sich um die Wurzel des Hyperdokumentes oder – wenn man so will – das Hyperdokument selbst. Der Name ist fest und für den Benutzer nicht von Bedeutung. Der Typ des Hyperdokumentes ergibt sich durch die Bindungen, da erst hier feststeht, für welche äußere Klasse tatsächlich eine Instanz erzeugt werden soll. Wie man anhand des Pfeils sieht, ist unser Beispiel-Hyperdokument vom Typ `Book`.

6.3.3. Struktur des Zielcodes

Abbildung 6.4 zeigt schematisch die Struktur des Codes, der für unser Beispiel erzeugt wird. Jeder schraffiert hinterlegte Bereich steht für eine Datei, deren Namen jeweils in einem kleinen Oval angegeben ist. Jedes Rechteck repräsentiert einen Bezeichner im C-Code. Ähnlich wie bei der Symboltabelle werden auch hier Unterstrukturen als kleinere Rechtecke innerhalb von größeren dargestellt. Jeder Pfeil entspricht einem Zeiger, der eine der Strukturen mit einer anderen verbindet. Diese Zeiger werden entweder bereits im C-Code fixiert oder erst während der Ausführung des Programms gesetzt.

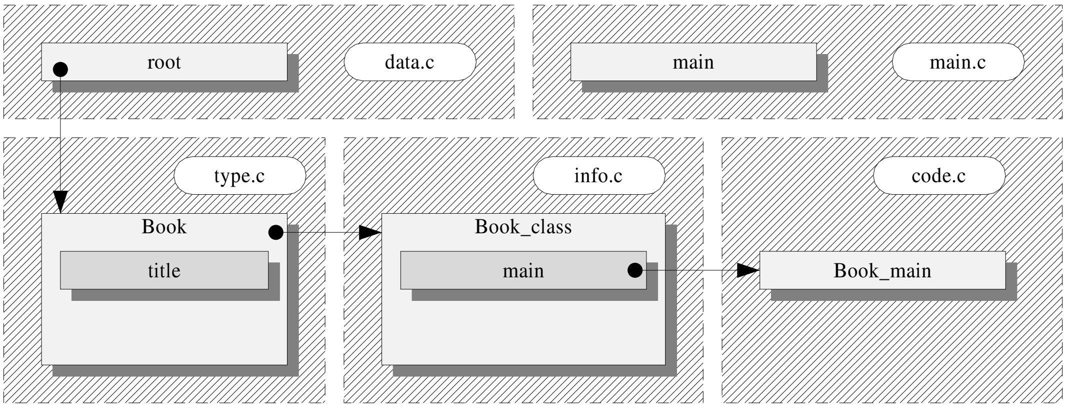

Abbildung 6.4.: Struktur des Zielcodes für eine einfache Klasse

Wir sehen die übliche C-Funktion `main` und eine Variable `root`, die das Hyperdokument repräsentiert. `root` verweist auf die Klasse `Book`, die über das Feld `class` auf eine weitere Struktur `Book__class` verweist. Letztere enthält die Klasseninformationen der Klasse `Book`, die wir in der Grafik auf die Tabelle der Methoden reduziert haben. In der Tabelle finden wir einen Zeiger auf eine Funktion `main`, die – nicht dargestellt – genau die Signatur unserer gewünschten Methode hat. Dieser Zeiger verweist auf die tatsächliche Implementierung der Methode, die sich in der globalen Funktion `Book__main` verbirgt. Während in einem Hyperdokument beliebig viele verschiedene Instanzen einer Klasse erzeugt werden können, existiert die Struktur mit Klasseninformationen nur einmal. Alle Instanzen verweisen also – naheliegenderweise – auf die gleiche Struktur.

Da jeder Bezeichner im C-Code nur einmal verwendet werden darf – und da C leider nur ausgesprochen rudimentäre Namensräume bietet – mußten die ursprünglichen, aus der Spezifikation stammenden Bezeichner einige Veränderungen erfahren. So wird zum Beispiel an den Namen der Informationsstruktur der Suffix `class` angehängt. `class` ist ein reserviertes Wort der Sprache $DoDL_C$ und kann somit keinesfalls mit einem vom Benutzer gewählten Bezeichner kollidieren. Ebenso mußte die Implementierung der Methode `main` in der Datei `code.c` einen neuen Namen erhalten, damit sie nicht mit der globalen Funktion `main` oder einer gleichnamigen Methode einer anderen Klasse kollidiert.

Der doppelte Unterstrich in den Bezeichnern dient zur Abbildung der hierarchischen Namensräume von $DoDL_C$ in den flachen Namensraum von C: Jedem Bezeichner wird – falls nötig – der Name des umschließenden Namensraums mit einem trennenden doppelten Unterstrich vorangestellt. Nötig ist dies, wenn sich das C-Konstrukt, das sich aus einem Bezeichner ergibt, im globalen Namensraum von C wiederfindet. Für Bezeichner innerhalb von Strukturen ist keine Umbenennung notwendig. Innerhalb einer Spezifikation dürfen Unterstriche in Bezeichnern nicht verwendet werden. Dadurch erhalten wir eine injektive Abbildung zwischen den Namensräumen von $DoDL_C$ und dem von ANSI-C.

Der aufmerksame Leser wird sich wundern, wohin denn der äußerste Namensraum `DoDL` entschwunden ist, den wir erst vor kurzem kennengelernt haben. Dieser wurde zur Verbesserung der Übersichtlichkeit in dieser und auch in den folgenden Grafiken weggelassen. Die Bezeichner müßten also eigentlich `DoDL__Book`, `DoDL__Book__class` etc. heißen. Im folgenden Abschnitt, der den Inhalt des Zielcodes zeigt, werden sie uns mit den erwarteten Namen begegnen.

6.3.4. Inhalt des Zielcodes

Schauen wir uns in den folgenden Abschnitten an, wie der C-Code aussieht, den der $DoDL_C$-Compiler für unsere Spezifikation erzeugt.

Inhalt der Datei `type.c`

Quellcode 6.2 enthält einige Zeilen aus der Datei `type.c`, in welcher die Instanztypen deklariert werden. Die Struktur, die für unsere Klasse `Book` an dieser Stelle erzeugt wird, heißt `DoDL__Book`. Sie beinhaltet zunächst einen Zeiger, der beim Initialisieren einer Instanz auf die entsprechende Struktur mit Klasseninformationen gesetzt werden muß. Wir werden später noch sehen, wie dies funktioniert. Dem Zeiger folgt das erwartete Feld `title` vom Typ `string`. Letzterer ist im C-Code unter dem Namen `DoDL__string` bekannt.

```
typedef struct {
  struct DoDL__Book__class * __class;   // Verweis auf Klasse
  DoDL__string title;                    // Dokument "title"
} *DoDL__Book;
```

Quellcode 6.2: Ausschnitt aus der Datei `type.c`

Inhalt der Datei `info.c`

Quellcode 6.3 zeigt einen Ausschnitt aus der Datei `info.c`, die Informationen über alle Klassen bereitstellt. Wir sehen die Deklaration der Struktur `DoDL__Book__class`. Es existiert ein Feld zur Aufnahme des Klassennamens, das im unteren Teil der Struktur mit dem passenden Namen belegt wird. Ebenso finden wir ein Feld, das auf die entsprechende Superklasse verweist – überraschenderweise enthält es nicht den Wert `NULL`, sondern die Adresse einer Klasse

`DoDL__Object`, die zur Systembibliothek des Compilers gehört. `DoDL__Object` ist sozusagen „die Mutter aller Klassen“.

```
struct DoDL__Book__class {
  char *__name;                            // Name der Klasse
  struct DoDL__Book__class *__class;       // Verweis auf sich selbst
  struct DoDL__Object__class *__super;     // Verweis auf Vorfahrenklasse
  struct {
    void (*main)(void *__this);            // Signatur der Methode "main"
  } __methods;
} DoDL__Book__class = {                    // Ab hier Belegung mit Werten
  "Book",
  &DoDL__Book__class,
  &DoDL__Object__class,
  {
    &DoDL__Book__main,                     // Implementierung von "main"
  }
};
```

Quellcode 6.3: Ausschnitt aus der Datei `info.c`

Klassenname und Superklasse haben keine wirkliche Bedeutung innerhalb des Systems, es hat sich jedoch während der Entwicklung des Compilers als hilfreich erwiesen, daß diese Informationen bereitstehen. Wesentlich interessanter ist die Unterstruktur `__methods`, die eine Tabelle aller Methoden der Klasse enthält. Wir finden in dieser Tabelle einen Zeiger auf eine Funktion, welche die komplette Signatur unserer Methode `main` inklusive des implizit vorhandenen Parameters `this` besitzt. Im unteren Teil wird dieser Zeiger mit der Adresse einer Funktion namens `DoDL__Book__main` belegt, hinter der sich die Implementierung unserer Methode `main` verbirgt.

Inhalt der Datei `code.c`

In Quellcode 6.4 sehen wir einen Ausschnitt aus der Datei `code.c`, der deutlich macht, wie der C-Code für die `construct`-Sektion unseres Beispiels aufgebaut ist.

Zunächst werden einige Hilfsmakros definiert, die uns den Namen der aktuellen Klasse und den der Superklasse sowie eine Zugriffsmöglichkeit auf den impliziten Parameter `this` liefern. Ein weiteres Makro `scope` dient zur Umwandlung des $DoDL_C$-Bezeichners einer Methode in den C-Bezeichner. Die Makros sollen helfen, die bereits mehrfach erwähnte „Aufbereitung“ des Quellcodes in einem erträglichen und vor allem lesbaren Rahmen zu halten.

`super` kommt zum Beispiel unmittelbar dann zum Einsatz, wenn im Quellcode eine Methode eines Vorfahren aufgerufen wird. Es wäre sicherlich auch möglich gewesen, jedes Vorkommen von `super` in der Methode durch den $DoDL_C$-Compiler ersetzen zu lassen.

Das etwas komplexer aufgebaute Makro `this` sorgt dafür, daß beim Zugriff auf den Instanz-Parameter der Methode der korrekte Typ verwendet wird. Wie wir sehen, heißt der Parameter selbst nämlich `__this` und ist untypisiert. Dies erlaubt uns, die gleiche (Implementierung der) Methode auch für Nachkommen der Klasse `Book` zu verwenden, da alle Zeigertypen zuweisungskompatibel zum untypisierten Zeiger sind. Man stelle sich hierzu eine Klasse `NewBook`

```
#define class DoDL__Book                    // Makros definieren
#define super (&DoDL__Object__class)
#define scope(id) DoDL__Book ## __ ## id
#define this ((class)(__this))
#define call(caller, callee) caller->__class->__methods.callee

// Methode "main" mit einem Zugriff auf Feld "title"
void scope(main)(void *__this) {
  printf("Contents of book \"%s\":\n", this->title);
}

#undef call                                 // Makros loeschen
#undef scope
#undef this
#undef class
#undef super
```

Quellcode 6.4: Ausschnitt aus der Datei `code.c`

vor, die von der Klasse `Book` abstammt und deren Methode `main` nicht überschreibt. Beide Klassen würden sich also eine Implementierung von `main` teilen, und die Informationsstrukturen beider Klasse würden auf die gleiche C-Funktion verweisen. Wäre der Instanzzeiger typisiert, dann hätten die Methoden unterschiedliche Signaturen, und der C-Compiler würde eine Struktur wie die aus Quellcode 6.4 (siehe dort *Signatur* und *Implementierung* von `main`) als fehlerhaft anmahnen oder gar abweisen.

Das Makro `scope` hängt vor den als Parameter übergebenen Bezeichner mithilfe des C-Makro-Operators `##` den Namen des umschließenden Namensraums, in unserem Beispiel also `DoDL__Book`.

Das Makro `call` dient zum Aufruf von Methoden über die Methodentabelle. Die Arbeitsweise läßt sich im Zusammenhang mit den beiden vorangehenden Ausschnitten des Zielcodes gut nachvollziehen. Es ist für alle Klassen identisch.

Abschließend werden alle Makros wieder entfernt, da eine möglicherweise folgende Klasse die gleichen Makros anders definieren würde und der C-Compiler das Redefinieren von Makros nicht besonders mag, was er in jedem solchen Fall durch eine entsprechende Warnung lautstark kundtut.

Inhalt der Datei `data.c`

Schauen wir uns als nächstes an, welchen Zielcode der Compiler für die Bindungen erzeugt. Dazu betrachten wir in Quellcode 6.5 einen Ausschnitt aus der Datei `data.c`.

Wie bereits erwähnt, liegt der Bindungsabschnitt strukturell nahe am endgültigen Code, weshalb er bereits vom Parser in eine Reihe von Zuweisungen übersetzt wird. Wir sehen zunächst eine Variable `root`, die unser Hyperdokument enthält. Diese Variable korrespondiert unmittelbar mit dem gleichnamigen Eintrag, den wir in der Symboltabelle gefunden haben. Außerdem wird eine Funktion `bind` erzeugt, welche die einzelnen Zuweisungen durchführt: Zunächst wird von einem Makro `init` der Verweis auf die Klasse des Dokumentes gesetzt

```
DoDL__Book root;                    // Variable fuer Hauptobjekt

void bind(void)                     // Funktion fuer Bindungen
{
  init(root, DoDL__Book__class);    // Variable initialisieren
  root->Title = "The Art of DoDL";  // Feld "title" mit Wert belegen
}
```

Quellcode 6.5: Ausschnitt aus der Datei `data.c`

– man kann dies als eine Art Konstruktor-Aufruf betrachten – anschließend erhält unser Feld `title` den gewünschten Wert.

Inhalt der Datei `main.c`

Quellcode 6.6 zeigt einen Ausschnitt aus der Datei `main.c`. Diese Datei stellt eine Funktion `main` bereit, die nichts weiter tut, als zunächst die Bindungen durchzuführen und anschließend die Methode `main` des äußeren, das Hyperdokument erzeugenden Objektes aufzurufen. Der Inhalt dieser Datei ist unabhängig von Spezifikation und Bindungen, also stets gleich.

```
int main(int argc, char *argv[])   // Feste Funktion "main"
{
  bind();                          // Bindungen ausfuehren
  call(root, main)(root);          // Methode "main" von "root" aufrufen
}
```

Quellcode 6.6: Ausschnitt aus der Datei `main.c`

Wir sehen hier zum ersten Mal die Verwendung des Makros `call` zum Aufruf von Methoden: Über den ersten Parameter findet das Makro (zur Laufzeit des übersetzten Programms!) die richtige Methodentabelle. Es kann sowohl auf Instanzen als auch auf Klassen aufgerufen werden. Es folgt der Name der Methode und die Liste der aktuellen Parameter. In unserem Beispiel ist dies das äußere Objekte, das zum `this`-Parameter der Methode wird.

6.4. Übersetzung lokaler Klassen

In diesem Abschnitt wollen wir die Behandlung lokaler Deklarationen aus Sicht des Compilers, also insbesondere den Zielcode, der für diesen Fall erzeugt wird, betrachten.

6.4.1. Quellcode der Spezifikation

Bauen wir zunächst unser bekanntes Beispiel so aus, daß es eine lokale Klasse deklariert und ein entsprechendes Dokument, also eine Instanz dieser Klasse verwendet. Quellcode 6.7 zeigt die Änderungen.

Die lokale Klasse trägt den Namen `Chapter` – sie soll folglich ein Kapitel unseres Buches repräsentieren. Das Kapitel verfügt über einen eigenen Titel `title` und zwei Methoden `head`

```
class Book is
  declare
    class Chapter is
      documents
        title : string;

      construct
        void head(void) {
          printf("*␣%s\n", title);
        }

        void body(void) {
          puts("␣␣Abstract:␣Valuable␣information␣about␣DoDL");
        }

    end Chapter;

  documents
    title :     string;
    chapter1 :  Chapter;

  construct
    void main(void) {
      printf("Contents␣of␣book␣\"%s\":\n", title);
      chapter1.head();
      chapter1.body();
    }
end Book;

binding Book is
  title = "The␣Art␣of␣DoDL";

  in chapter1 assign
    title   = "DoDL␣-␣Principles,␣Techniques,␣and␣Tools";
  end;
end;
```

Quellcode 6.7: Spezifikation einer lokalen Klasse und `body`. Die Methode `head` gibt den Titel des Kapitels aus, `body` soll eine kurze Inhaltsangabe anzeigen. Letztere ist der Einfachheit halber fest codiert. Die Hauptklasse ruft in ihrer Methode `main` die beiden Methoden der lokalen Klasse auf.

6.4.2. Inhalt der Symboltabelle

Abbildung 6.5 gibt den Inhalt der Symboltabelle nach der syntaktischen Analyse der Spezifikation graphisch wieder. Hinzugekommen sind die lokale Klasse `Chapter` mit ihrem Feld `title` und ihren beiden Methoden `head` und `body` sowie das Dokument `chapter1` in `Book`.

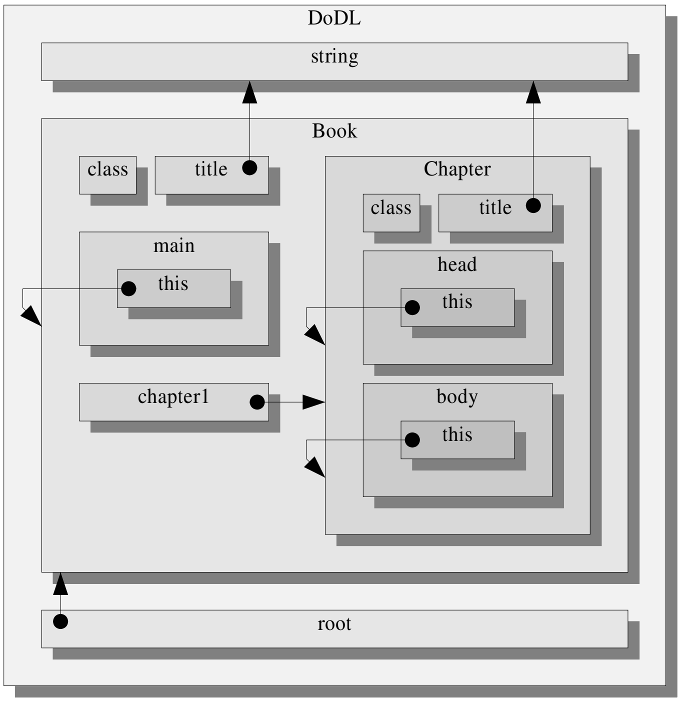

Abbildung 6.5.: Symboltabelle für eine lokale Klasse

6.4.3. Struktur des Zielcodes

Abbildung 6.6 zeigt schematisch die Struktur des Zielcodes, der nun für unser Beispiel erzeugt wird. Die Grafik zeigt noch deutlicher als die des ersten Beispiels die Abbildung der hierarchischen Namensräume von $DoDL_C$ in den flachen Namensraum von C: Die Namen der Funktionen, die den Methoden der lokalen Klasse zugrundeliegen, setzen sich nun aus drei Teilen zusammen.

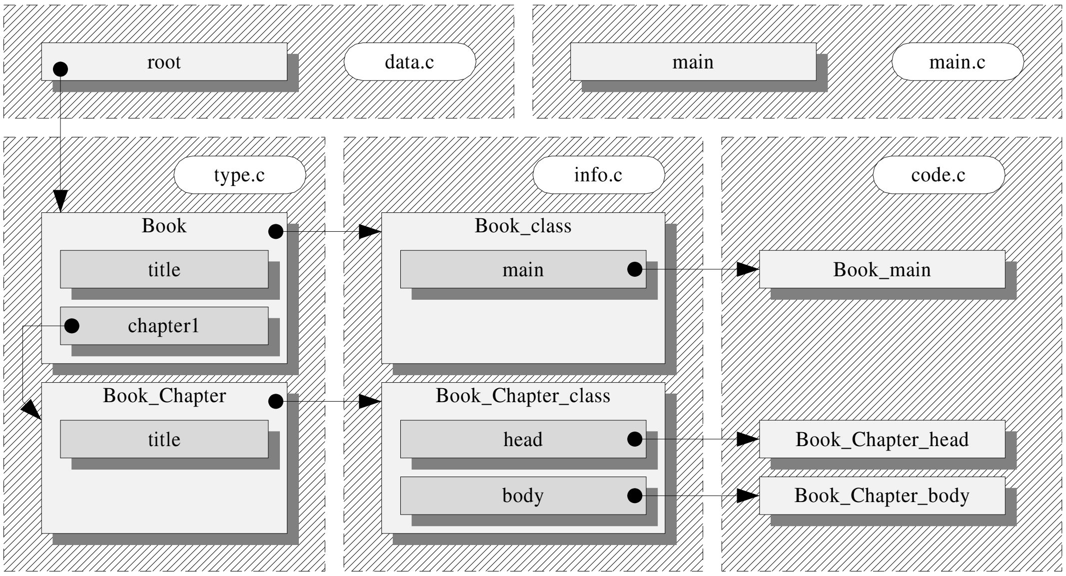

Abbildung 6.6.: Struktur des Zielcodes für eine lokale Klasse

6.4.4. Inhalt des Zielcodes

Schauen wir uns in den folgenden Abschnitten wieder im Detail an, wie der C-Code aussieht, den der *DoDL*$_C$-Compiler für unsere geänderte Spezifikation erzeugt. Die folgenden Quellcode-Stücke 6.8, 6.9 und 6.10 zeigen Ausschnitte aus den Dateien `type.c`, `info.c` und `code.c`, die für das neue Beispiel erzeugt werden. Durch die Verwendung der Vorwärts-Deklarationen in den Dateien `type.c` und `info.c` ist die Reihenfolge der sich gegenseitig benutzenden Strukturen bedeutungslos.

```
typedef struct DoDL__Book__Chapter__struct *DoDL__Book__Chapter;
typedef struct DoDL__Book__struct *DoDL__Book;

struct DoDL__Book__Chapter__struct {              // Klasse "Chapter"
  struct DoDL__Book__Chapter__class * __class;
  DoDL__string title;                             // mit Feld "title"
};

struct DoDL__Book__struct {                       // Klasse "Book"...
  struct DoDL__Book__class * __class;
  DoDL__string title;                             // mit Feld "title"
  DoDL__Book__Chapter chapter1;                   // und Feld "chapter1"
};
```

Quellcode 6.8: Ausschnitt aus der Datei `type.c`

Auch hier sieht man deutlich, wie die Bezeichner bei der Verwendung lokaler Klassen immer länger werden. Die Implementierung der Methode `main` (die sich in der Funktion

```
struct DoDL__Book__class;
struct DoDL__Book__Chapter__class;

struct DoDL__Book__class {
  char *__name;
  struct DoDL__Book__class *__class;
  struct DoDL__Object__class *__super;
  struct {                                   // Methodensignaturen
    void (*main)(void *__this);
  } __methods;
} DoDL__Book__class = {
  "Book",
  &DoDL__Book__class,
  &DoDL__Object__class,
  {
    &DoDL__Book__main,                       // Methodenimplementierungen
  }
};

struct DoDL__Book__Chapter__class {
  char *__name;
  struct DoDL__Book__Chapter__class *__class;
  struct DoDL__Object__class *__super;
  struct {                                   // Methodensignaturen
    void (*main)(void *__this);
    void (*head)(void *__this);
    void (*body)(void *__this);
  } __methods;
} DoDL__Book__Chapter__class = {
  "Chapter",
  &DoDL__Book__Chapter__class,
  &DoDL__Object__class,                      // Methodenimplementierungen
  {
    &DoDL__Object__main,
    &DoDL__Book__Chapter__head,
    &DoDL__Book__Chapter__body,
  }
};
```

Quellcode 6.9: Ausschnitt aus der Datei `info.c`

```
...
void scope(main)(void *__this) {
  printf("Contents of book \"%s\":\n", this->title );
  call(this->chapter1, head)(this->chapter1);   // Zugriff auf "chapter1"
  call(this->chapter1, body)(this->chapter1);   // Zugriff auf "chapter1"
}
...
```

Quellcode 6.10: Ausschnitt aus der Datei `code.c`

DoDL__Book__main verbirgt) zeigt außerdem sehr schön die Verwendung des Makros call zum Aufruf der beiden Methoden der lokalen Klasse.

6.5. Übersetzung des Vererbungsmechanismus

Wir wollen nun sehen, wie der Vererbungsmechanismus in $DoDL_C$ implementiert ist, also insbesondere, welche Auswirkungen Vererbung auf Symboltabelle und Zielcode hat.

6.5.1. Quellcode der Spezifikation

Quellcode 6.11 zeigt die angepaßte Spezifikation. Sie entspricht weitgehend der Spezifikation aus dem vorangehenden Abschnitt. Hinzugekommen ist die lokale Klasse `AuthorChapter`, die Nachkomme von `Chapter` ist und zusätzlich ein Feld `author` bereitstellt, in welchem der Autor des Kapitels gespeichert werden kann. Damit der Autor auch ausgegeben wird, mußte die Methode `head` überschrieben – also redefiniert – werden. Die neue Implementierung ruft zunächst gezielt über `super.head()` den von `Chapter` geerbten Code auf und zeigt anschließend den Namen des Autors an. Das Dokument `chapter1` verwendet die neue Klasse anstelle der alten.

6.5.2. Symboltabelle

Abbildung 6.7 zeigt die Symboltabelle, die sich aus dem Beispiel ergibt. Die neue Klasse `AuthorChapter` findet sich rechts neben der bekannten Klasse `Chapter` im Namensraum von `Book` wieder. Die Grafik macht einen wesentlichen Aspekt der Modellierung von Vererbung in der Symboltabelle deutlich: Die Nachkommen-Klasse enthält nur die neu deklarierten Dokumente und Methoden. Geerbte Komponenten werden nicht explizit aufgeführt. Stattdessen verweist die Klasse – der gestrichelte Pfeil deutet es an – auf ihren Vorfahren. Ebenso verweist die Methode `head` auf die gleichnamige Methode des Vorfahren, die sie redefiniert.

Die Symboltabelle speichert offenbar keine redundaten Informationen über verwandte Klassen. Wenn Bezeichner in der Symboltabelle gesucht werden, wandert die Suchfunktion von der aktuellen Klasse über alle Vorfahren bis zur Wurzelklasse. Diese Art der Impementierung wurde gewählt, weil sie zum einen Speicherplatz spart, zum anderen Probleme vermeidet, die sich aus möglichen Inkonsistenzen zwischen einem an mehreren Stellen gespeicherten identischen Datum ergeben.

6.5.3. Struktur des Zielcodes

Die Art und Weise, wie die Symboltabelle mit Vererbung umgeht, kann leider für den erzeugten Zielcode nicht unmittelbar verwendet werden: Hier müssen natürlich in Klassen und Instanzen alle geerbten Informationen explizit aufgeführt werden. Grafik 6.8 und die zusätzlichen Strukturen `Book__Author__Chapter` und `Book__Author__Chapter__class` zeigen dies.

Sehr aufschlußreich ist die Grafik außerdem, was die Beziehung zwischen den Methodentabellen der beiden lokalen Klassen und den Implementierungen der Methoden betrifft:

```
class Book is
  declare
    class Chapter is
      documents
        title : string;

      construct
        void head(void) {
          printf("* %s\n", title);
        }

        void body(void) {
          puts("  Abstract: Valuable information about DoDL");
        }
    end Chapter;

    class AuthorChapter is Chapter with
      documents
        author: string;

      construct
        void head(void) {
          super.head();
          printf("  By: %s\n", author);
        }
    end AuthorChapter;

  documents
    title:     string;
    chapter1: AuthorChapter;

  construct
    void main(void) {
      printf("Contents of book \"%s\":\n", title);
      chapter1.head();
      chapter1.body();
    }
end Book;

binding Book is
  title = "The Art of DoDL";

  in chapter1 assign
    title  = "DoDL - Principles, Techniques, and Tools";
    author = "A. V. Aho, R. Sethi, J. D. Ullman";
  end;
end;
```

Quellcode 6.11: Spezifikation einer geerbten Klasse

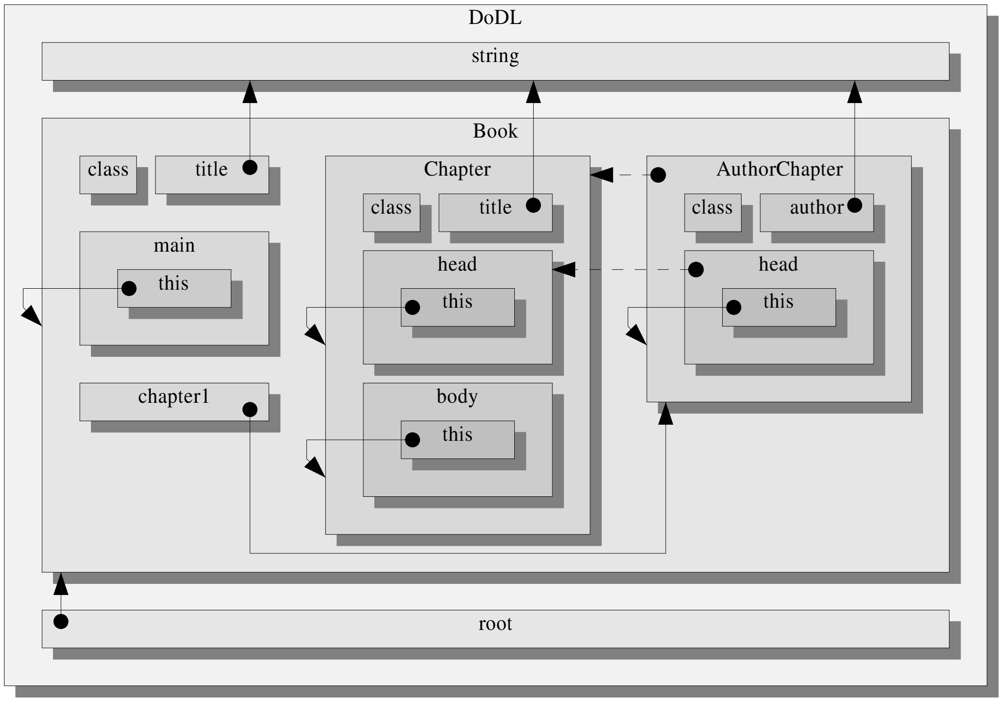

Abbildung 6.7.: Symboltabelle für eine geerbte Klasse

- Die Methoden `head` und `body` befinden sich in beiden Klassen an der gleichen Position der Methodentabelle. Dies ist wichtig für den Mechanismus der Methodenzuteilung zur Laufzeit, der ja letztlich nichts weiter als ein indizierter Zugriff auf die Methoden ist.
- Die Einträge der Methode `head` verweisen – wie zu erwarten war – in beiden Tabellen auf unterschiedliche Implementierungen der Methode. Das bereits vorgestellte Makro `call` findet zur Laufzeit stets die richtige.
- Die Einträge der Methode `body` verweisen auf die gleiche Implementierung. Offenbar teilen sich also beide Klassen die gleiche Implementierung der Methode. Es wird keinerlei Code kopiert. Genau aus diesem Grund wurde der Parameter `__this` im C-Code untypisiert deklariert und erst über das Makro `this` innerhalb der Methode selbst in den richtigen Typ umgewandelt.

6.5.4. Inhalt des Zielcodes

Schauen wir uns nun den Zielcode für unsere neue Spezifikation an. Die Quellcode-Stücke 6.12, 6.13, 6.14 und 6.15 zeigen Ausschnitte aus den Dateien `type.c`, `info.c` und `code.c`, die für das neue Beispiel erzeugt werden. Die Ausschnitte zeigen deutlich die Gedanken aus dem letzten Abschnitt; insbesondere sieht man sehr schön, wie beide lokalen Klassen auf die gleiche Implementierung von `body`, aber auf unterschiedliche Implementierungen von `head` verweisen.

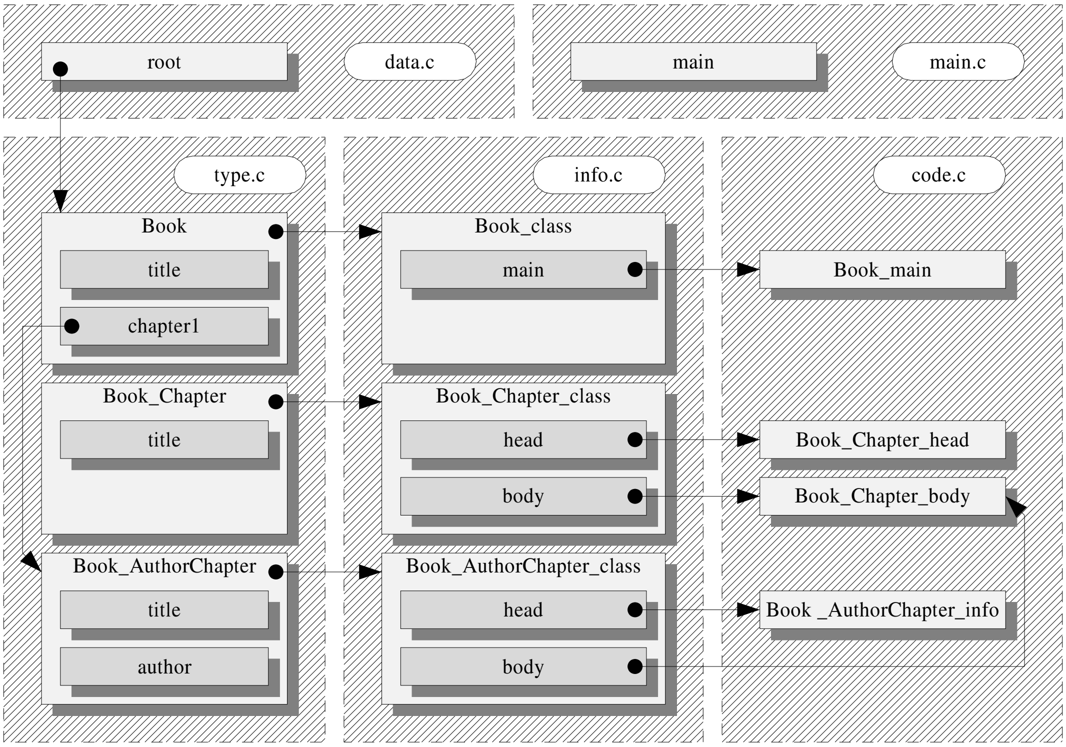

Abbildung 6.8.: Struktur des Zielcodes für eine lokale Klasse

```
struct DoDL__Book__Chapter__struct {              // Vorfahr
  struct DoDL__Book__Chapter__class * __class;
  DoDL__string title;
};

struct DoDL__Book__AuthorChapter__struct {        // Nachkomme
  struct DoDL__Book__AuthorChapter__class * __class;
  DoDL__string title;  // geerbtes Feld "title"
  DoDL__string author; // neues Feld "author"
};

struct DoDL__Book__struct {
  struct DoDL__Book__class * __class;
  DoDL__string title;
  DoDL__Book__AuthorChapter chapter1;
};
```

Quellcode 6.12: Ausschnitt aus der Datei `type.c`

```
struct DoDL__Book__class;
struct DoDL__Book__Chapter__class;
struct DoDL__Book__AuthorChapter__class;

struct DoDL__Book__class {
  char *__name;
  struct DoDL__Book__class *__class;
  struct DoDL__Object__class *__super;
  struct {
    void (*main)(void *__this);
  } __methods;
} DoDL__Book__class = {
  "Book",
  &DoDL__Book__class,
  &DoDL__Object__class,
  {
    &DoDL__Book__main,
  }
};
```

Quellcode 6.13: Ausschnitt aus der Datei `info.c` (Teil 1 von 2)

6.6. Übersetzung des Listenkonstruktes

Das Listenkonstrukt von $DoDL_C$ bietet eine Möglichkeit, die tatsächliche Anzahl der Instanzen eines Dokumentes erst durch die `binding`-Sektion bestimmen zu lassen. Dieser Abschnitt zeigt wie das Listenkonstrukt übersetzt wird. Die Realisierung basiert im wesentlichen auf einem dynamischen Array, dessen Länge im „minus-ersten“ Element gespeichert wird. Das Array wird von speziellen Bibliotheksfunktionen für den Benutzer unsichtbar gewartet.

6.6.1. Quellcode der Spezifikation

Um das Listenkonstrukt zu verwenden, deklarieren wir nicht mehr ein einzelnes Dokument vom Typ `AuthorChapter`, sondern direkt eine ganze Liste. Innerhalb der Bindungen erzeugen wir dann drei Elemente für diese Liste. Die Methode `main` von `Book` ruft für alle Elemente der Liste die Methoden `head` und `body` auf. Die Funktion `size` ist Bestandteil der Systembibliothek. Sie ermittelt die Anzahl der Elemente einer Liste.

6.6.2. Inhalt der Symboltabelle

Die Symboltabelle in Abbildung 6.9 entspricht weitgehend der aus dem letzten Beispiel. Das Dokumente `chapter1` ist einem Dokument `chapters` gewichen, das eine Liste von Instanzen von `AuthorChapter` aufnehmen kann.

6.6.3. Struktur des Zielcodes

Auch die Struktur des Zielcodes hat sich für unser neues Beispiel nicht wesentlich geändert, wie Abbildung 6.10 deutlich macht. Im Detail hat sich der erzeugte Zielcode jedoch sehr wohl

```
struct DoDL__Book__Chapter__class {
  char *__name;
  struct DoDL__Book__Chapter__class *__class;
  struct DoDL__Object__class *__super;
  struct {                              // Methodensignaturen
    void (*main)(void *__this);
    void (*head)(void *__this);
    void (*body)(void *__this);
  } __methods;
} DoDL__Book__Chapter__class = {        // Methodenimplementierungen
  "Chapter",
  &DoDL__Book__Chapter__class,
  &DoDL__Object__class,
  {
    &DoDL__Object__main,
    &DoDL__Book__Chapter__head,
    &DoDL__Book__Chapter__body,
  }
};

struct DoDL__Book__AuthorChapter__class {
  char *__name;
  struct DoDL__Book__AuthorChapter__class *__class;
  struct DoDL__Object__class *__super;
  struct {
    void (*main)(void *__this);         // Methodensignaturen
    void (*head)(void *__this);         // sind exakt so wie
    void (*body)(void *__this);         // beim Vorfahren.
  } __methods;
} DoDL__Book__AuthorChapter__class = {
  "AuthorChapter",
  &DoDL__Book__AuthorChapter__class,
  &DoDL__Object__class,
  {
    &DoDL__Object__main,                // Implementierungen
    &DoDL__Book__AuthorChapter__head,   // unterscheiden sich
    &DoDL__Book__Chapter__body,         // jedoch.
  }
};
```

Quellcode 6.14: Ausschnitt aus der Datei `info.c` (Teil 2 von 2)

```
#define class DoDL__Book__Chapter
#define scope(id) DoDL__Book__Chapter ## __ ## id
#define super (&DoDL__Object__class)

void scope(head)(void *__this) {
 printf("*␣%s\n", this->title);
}

void scope(body)(void *__this) {
  puts("Abstract:␣Valuable␣information␣about␣DoDL");
}

#undef class
#undef scope
#undef super

#define class DoDL__Book__AuthorChapter
#define scope(id) DoDL__Book__AuthorChapter ## __ ## id
#define super (&DoDL__Book__Chapter__class)

void scope(head)(void *__this) {       // Das Makro "call" sucht
  call(super, head)(this);             // die passende Methode
  printf("By:␣%s\n", this->author);    // aus der Tabelle und ruft
}                                      // sie auf.

#undef class
#undef scope
#undef super
...
```

Quellcode 6.15: Ausschnitt aus der Datei `code.c`

```
class Book is
  declare
    class Chapter is
      ...
    end Chapter;

    class AuthorChapter is Chapter with
      ...
    end AuthorChapter;

  documents
    title:      string;
    chapters:  list of AuthorChapter;

  construct
    void main(void) {
      int i;

      printf("Contents of book \"%s\":\n", title);
      for (i = 0; i < size(chapters); i++) {
        chapters[i].head();
        chapters[i].body();
      }
    }
end Book;

binding Book is
  title = "The Art of DoDL";

  in chapters assign
    title  = "DoDL - Principles, Techniques, and Tools";
    author = "A. V. Aho, R. Sethi, J. D. Ullman";
  | title  = "Algorithms in DoDL";
    author = "R. Sedgewick";
  | title  = "DoDL for the working programmer";
    author = "L. C. Paulson";
  end;
end;
```

Quellcode 6.16: Spezifikation einer Klasse mit Listenkonstrukt

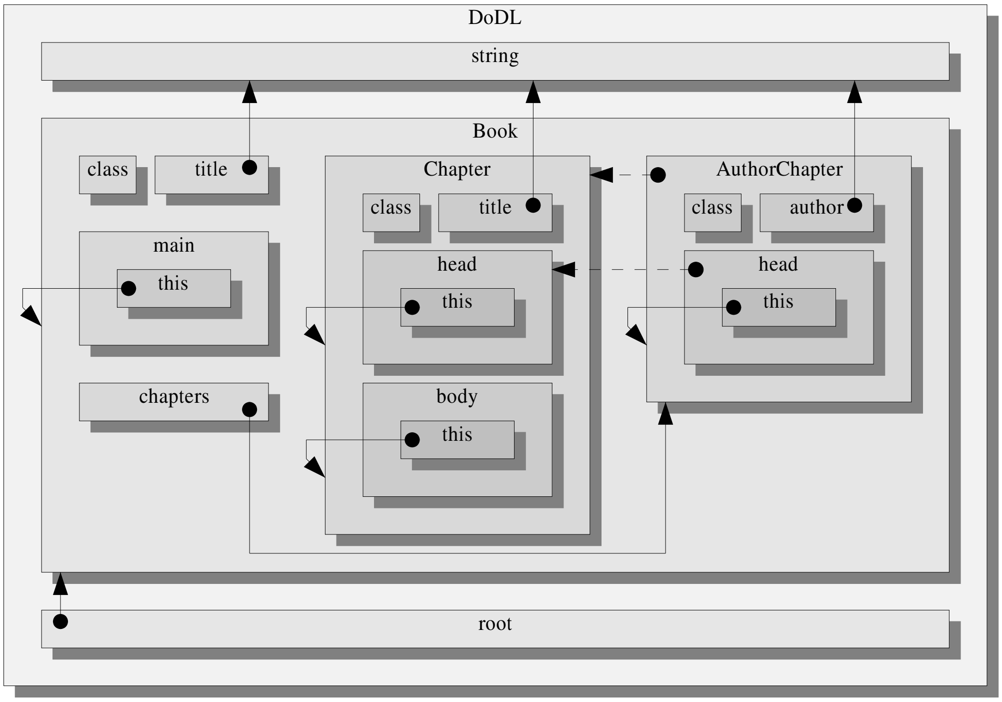

Abbildung 6.9.: Symboltabelle für das Listenkonstrukt

geändert, wie der folgende Abschnitt zeigen wird.

6.6.4. Inhalt des Zielcodes

Quellcode 6.17 zeigt einige Zeilen aus der Datei `type.c`. Die Struktur `DoDL__Book` enthält einen Zeiger auf eine Struktur vom Typ `DoDL__Book__AuthorChapter`, wo im vorangehenden Beispiel noch direkt ein Feld von diesem Typ deklariert wurde. Dieser Zeiger verweist auf das dynamische Array, das unserer Liste zugrunde liegt. Das schöne an dieser – zugegebenermaßen recht einfachen – Implementierung ist die Tatsache, daß Zeiger und Arrays in C kompatible Strukturen sind. Insbesondere kann auf einen Zeiger wie auf ein Array zugegriffen werden, wie der Ausschnitt 6.18 aus der Datei `code.c` zeigt: Wir sehen die Übersetzung der Methode `main` der Klasse `Book`. Die Zugriffe auf die Listenelemente wurden unverändert aus der Spezifikation übernommen.

```
struct DoDL__Book__struct {
  struct DoDL__Book__class *__class;
  DoDL__string title;
  DoDL__Book__AuthorChapter *chapters;   // Zeiger bzw. Array
};
```

Quellcode 6.17: Ausschnitt aus der Datei `type.c`

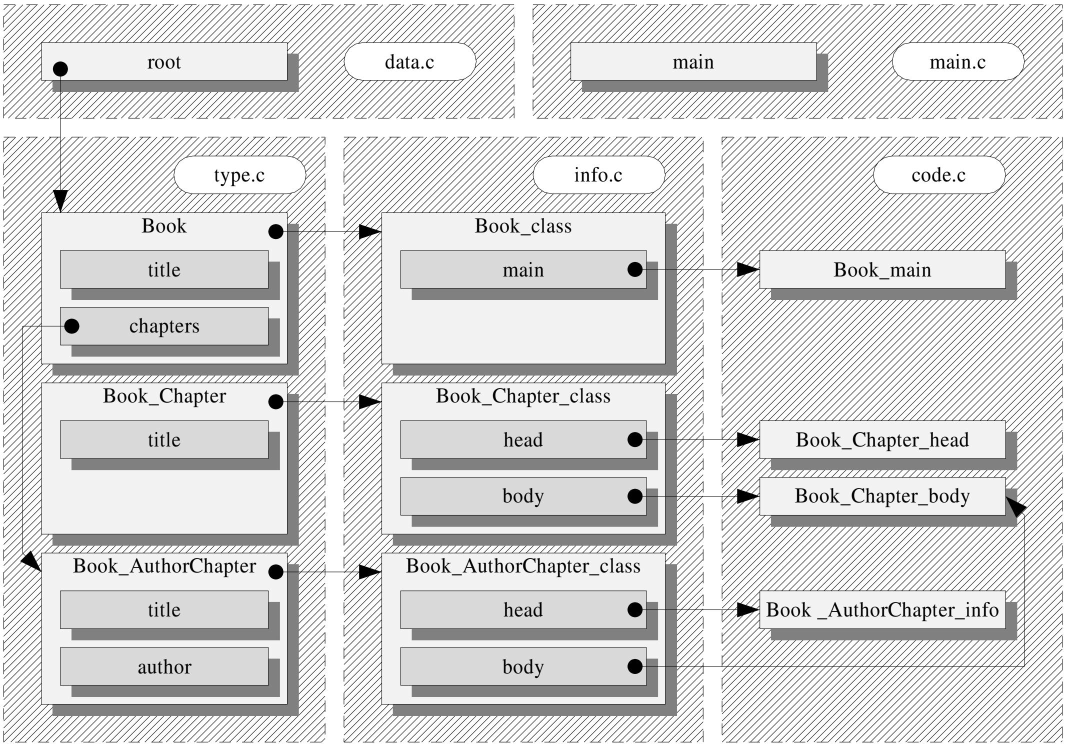

Abbildung 6.10.: Struktur des Zielcodes für das Listenkonstrukt

```
...
void scope(main)(void *__this) {
  DoDL__int i;

  printf("Contents of book \"%s\":\n", this->title );
  for(i = 0; i < size(this->chapters); i++) {
    call(this->chapters[i], head)(this->chapters[i]);   // Zugriff auf
    call(this->chapters[i], body)(this->chapters[i]);   // Listenelemente
  }
}
...
```

Quellcode 6.18: Ausschnitt aus der Datei `code.c`

Quellcode 6.19 enthält einen Ausschnitt aus der Datei `data.c`, also der Übersetzung der `binding`-Sektion. Der Ausschnitt zeigt, wie die Liste beim Erzeugen des ersten Elementes mit einer Funktion `init_list` initialisiert wird. Anschließend wird vor dem Hinzufügen jedes weiteren Elementes die Bibliotheksfunktion `grow_list` aufgerufen, um die Liste um ein Element zu erweitern. Jedes Element der Liste wird einzeln mit dem Makro `init` initialisiert.

```
DoDL__Book root;

void bind(void)
{
  init(root, DoDL__Book__class);
  root->title = "The Art of DoDL";

  // Erzeugen der Liste
  root->chapters = init_list(sizeof(*root->chapters));
  // Hinzufuegen eines Elementes
  root->chapters = grow_list(root->chapters);
  // Initialisieren des Elementes
  init(root->chapters[0], DoDL__Book__AuthorChapter__class);
  // Setzen der Felder
  root->chapters[0]->title = "DoDL - Principles, Techniques, and Tools";
  root->chapters[0]->author = "A. V. Aho, R. Sethi, J. D. Ullman";
  // Weiter mit dem naechsten Element

  root->chapters = grow_list(root->chapters);
  init(root->chapters[1], DoDL__Book__AuthorChapter__class);
  root->chapters[1]->title = "Algorithms in DoDL";
  root->chapters[1]->author = "R. Sedgewick";
  root->chapters = grow_list(root->chapters);
  init(root->chapters[2], DoDL__Book__AuthorChapter__class);
  root->chapters[2]->title = "DoDL for the working programmer";
  root->chapters[2]->author = "L. C. Paulson";
}
```

Quellcode 6.19: Ausschnitt aus der Datei `data.c`

6.7. Übersetzung generischer Klassen

Generische Klassen erlauben die Bildung von allgemeinen, parameterisierten Schablonen, von denen beliebig viele konkrete Klassen geerbt werden können. Jeder konkrete Nachkomme bindet den freien Parameter der Schablone an einen beliebigen Typ.

6.7.1. Quellcode der Spezifikation

Quellcode 6.20 zeigt unsere erneut erweiterte Beispiel-Spezifikation. Anstatt das Listenkonstrukt direkt zu verwenden, deklarieren wird nun eine generische Listenklasse `GenericList`, die über den Typnamen `SomeType` parametrisiert ist. Die Klasse enthält neben einer Liste von Elementen dieses Typs auch zwei Funktionen `info` und `show`. `show` ruft für alle Elemente der Liste die Methode `info` auf, eine Methode, die in der generischen Klasse leer implementiert ist, weil naheliegenderweise keine Annahmen über mögliche Belegungen von `SomeType` in konkreten Nachkommen getroffen werden können.

Die Klasse `ChapterList` ist ein direkter Nachkomme von `GenericList`, wobei der freie Parameter `SomeType` hier an die Klasse `AuthorChapter` gebunden wird. Die Klasse enthält also eine Liste von Elementen des Typs `AuthorChapter`. Gleichzeitig wird die Methode `info` derart überschrieben, daß die bereits bekannten Methoden `head` und `body` für das übergebene Element aufgerufen werden.

Im Endeffekt leistet das Beispiel also das gleiche wie bei der Veranschaulichung des Listenkonstruktes – nur verwendet es hierzu jetzt eine generische Klasse.

6.7.2. Inhalt der Symboltabelle

Mit den beiden zusätzlichen Klassen ist die Symboltabelle jetzt soweit gewachsen, daß wir an einigen Stellen mit Details sparen müssen, um den Rahmen einer Seite nicht zu sprengen. Die Klassen `Chapter` und `AuthorChapter` werden deshalb nicht weiter untergliedert. Das gibt uns den Platz, den wir benötigen, um die neuen Klassen `GenericList` und `ChapterList` in all ihrer Schönheit darzustellen.

Wie sieht nun die Symboltabelle für unsere generische Klasse aus? Wir würden erwarten, daß die Klasse `ChapterList` sich in der Symboltabelle als direkter Nachkomme von `GenericList` wiederfindet, daß dort der tatsächliche Typ von `SomeType` angegeben ist und die Methode `info` überschrieben wird. Abbildung 6.11 zeigt die Symboltabelle, die sich aus unseren Annahmen ergibt.

Die von uns angenommene Modellierung funktioniert prinzipiell, bringt aber in speziellen Fällen Probleme mit sich, die mit dem unterschiedlichen Charakter von Methoden bei generischen und nicht-generischen Klassen zusammenhängen: Der Compiler schreibt die Methoden nicht-generischer Klassen bereits während der syntaktischen Analyse in die Ausgabedatei `code.c`. Er speichert – wie bereits an anderer Stelle erwähnt – außer ihrer Signatur keine besonderen Informationen in der Symboltabelle und wirft insbesondere während des weiteren Vorgehens keinen „zweiten Blick“ auf den Code der Methoden.

Diese Vorgehensweise möchten wir natürlich auch für die Methoden generischer Klassen beibehalten. Hier stoßen wir aber auf ein Problem: Die Methoden sind *unvollständig*, sie basieren auf einem freien Parameter – in unserem Beispiel `SomeType` – dessen tatsächliche Belegungen in eventuellen konkreten Nachkommen nicht bekannt sind, während die generische Klasse verarbeitet wird.

Die Behandlung des Codes für die Methoden generischer Klassen soll deshalb auf einem Makro-Mechanismus basieren. Wenn eine konkrete Klasse von einer generischen Klasse erbt, geschehen bei der Behandlung der geerbten Methoden zwei Dinge:

1. Es wird ein Makro definiert, das den Namen des generischen Parameters besitzt und diesen mit dem tatsächlichen Typ belegt.

2. Die Methoden des generischen Vorfahren werden per `#include` eingebunden. Dadurch, daß der Name des Parameters zuvor bereits über das Makro umdefiniert wurde, wird der Typ überall korrekt ersetzt.

```
class Book is
  declare
    class Chapter is
      ...
    end Chapter;

    class AuthorChapter is Chapter with
      ...
    end AuthorChapter;

    generic class GenericList[SomeType] is
      documents
        items: list of SomeType;

      construct
        void info(SomeType item) {
        }

        void contents(void) {
          int i;

          for (i = 0; i < size(items); i++) {
            info(items[i]);
          }
        }
    end GenericList;

    class ChapterList is GenericList[AuthorChapter] with
      construct
        void info(AuthorChapter item) {
          item.head();
          item.body();
        }
    end ChapterList;

  documents
    title:    string;
    chapters: ChapterList;

  construct
    void main(void) {
      int i;

      printf("Contents of book \"%s\":\n", title);
      chapters.contents();
    }
end Book;

binding Book is
  ...
end;
```

Quellcode 6.20: Spezifikation einer generischen Klasse

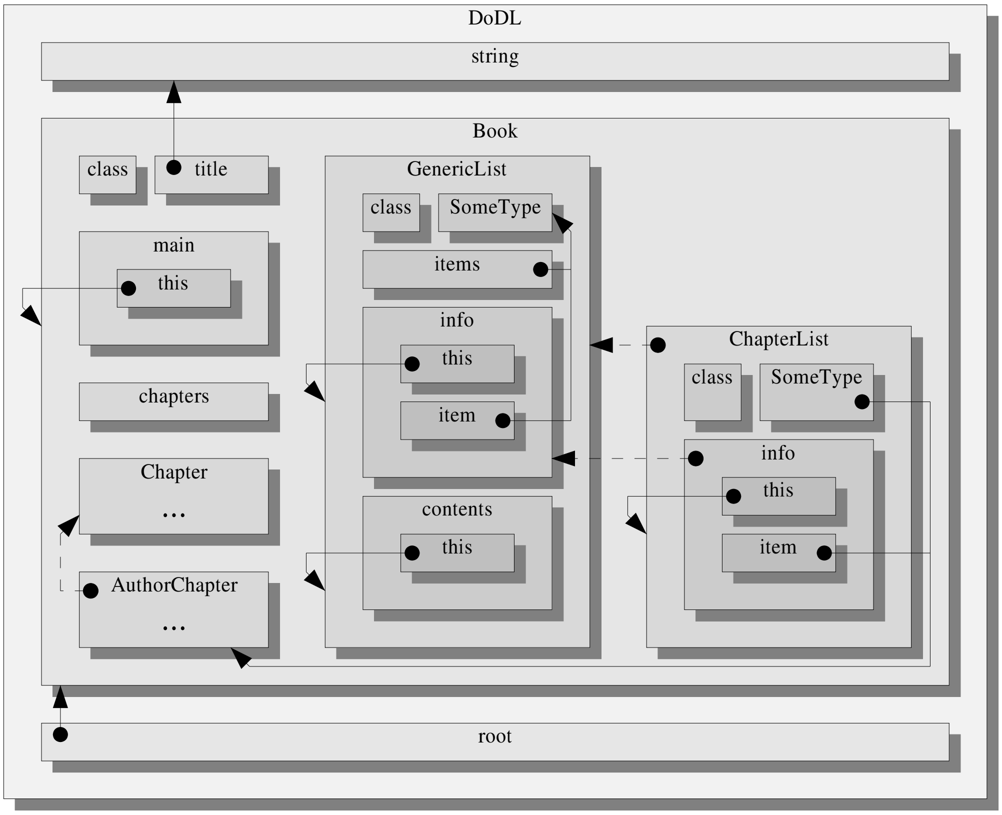

Abbildung 6.11.: Symboltabelle für generische Klassen (theoretisch)

Durch das Makro arbeiten also offenbar alle geerbten Methoden auf dem gewünschten Typ. Was geschieht aber nun, wenn der konkrete Nachkomme eine gerade erst geerbte Methode wieder überschreibt? In unserem Beispiel geschieht genau das: `ChapterList` überschreibt die Methode `info` von `GenericList`. Wenn nun zuerst sämtliche Methoden von `GenericList` per `#include` eingebunden werden und anschließend die von `ChapterList`, dann besitzt die entstehende Klasse offenbar zwei Implementierungen von `info`, die auch noch identische Namen tragen, nämlich in beiden Fällen `Book__ChapterList__info`. Wir haben also gleich an zwei Stellen einen Namenskonflikt: In der Methodentabelle und in der Datei `code.c`.

Schlimmer noch: Wenn wir – naheliegenderweise – die in `ChapterList` implementierte Methode als die richtige ansehen und diese in die Methodentabelle der entstehenden Klasse eintragen, was passiert dann beim Aufruf von `super.info`? Im Normalfall würde die Methodentabelle des Vorfahren zum Aufruf der Methode herangezogen, aber für `GenericList` existiert überhaupt keine Methodentabelle, da ja – wie bereits erwähnt – keine Klasseninformationen erzeugt werden können. Wie läßt sich dieses Problem nun ohne übermäßigen Aufwand lösen?

Die Lösung basiert auf der Idee, unsichtbar für den Benutzer eine weitere Zwischenklasse – sozusagen ein Bindeglied zwischen der generischen Klasse und der Klasse des Benutzers – zu erzeugen. Diese Klasse ist nichts weiter als die Konkretisierung der generischen Klasse, entspricht also dem, was wir erhalten, wenn der freie Parameter der generischen Klasse gebunden wird, aber noch keine Methode überschrieben wird. Für diese Klasse können wir Klasseninformationen inklusive einer Methodentabelle erzeugen. Diese Klasse ist es auch, die den Methodencode der generischen Klasse per `#include` einbindet.

Die von Benutzer deklarierte Klasse – also unsere `ChapterList` – wird direkter Nachkomme der Zwischenklasse. Folglich treten auch keinerlei Problem beim Überschreiben der Methoden auf, denn die Methoden befinden sich ja nun in getrennten Klassen und somit auch in getrennten Namensräumen. Auch das Aufrufen der Methoden des Vorfahren über `super` stellt keinerlei Problem mehr dar.

Ein kleines Detail-Probleme bleibt noch, aber das läßt sich glücklicherweise leicht lösen: Unsere Zwischenklasse muß einen eindeutigen Namen haben, der mit keinem vom Benutzer erzeugten Bezeichner kollidieren kann. Dazu hängen wir einfach den Suffix `generic` mit einem doppelten Unterstrich an den Namen der Klasse an, ganz so wie wir es zuvor schon mit `class` für die Klasseninformationen getan haben. Auch `generic` ist ein reserviertes Wort von $DoDL_C$ und kann als solches nicht in einem Bezeichner des Benutzers auftauchen. In unserem Beispiel heißt die Zwischenklasse also `Book__ChapterList__generic`.

Abbildung 6.12 zeigt die Symboltabelle, die der $DoDL_C$-Compiler während der syntaktischen Analyse unserer Beispiels tatsächlich aufbaut. Wie wir sehen, entspricht sie genau den Überlegungen, die wir in den vorangehenden Abschnitten angestellt haben.

6.7.3. Struktur

Abbildung 6.13 zeigt die Struktur des erzeugten Quellcodes. Auch hier mußten aus Platzgründen einige bereits aus den vorangehenden Beispielen bekannte Details weggelassen werden. Die Grafik gibt genau die Beziehungen wieder, die im letzten Abschnitt beschrieben wurden. Man sieht deutlich die Zwischenklasse und die Klasse des Benutzers sowie die Methodenta-

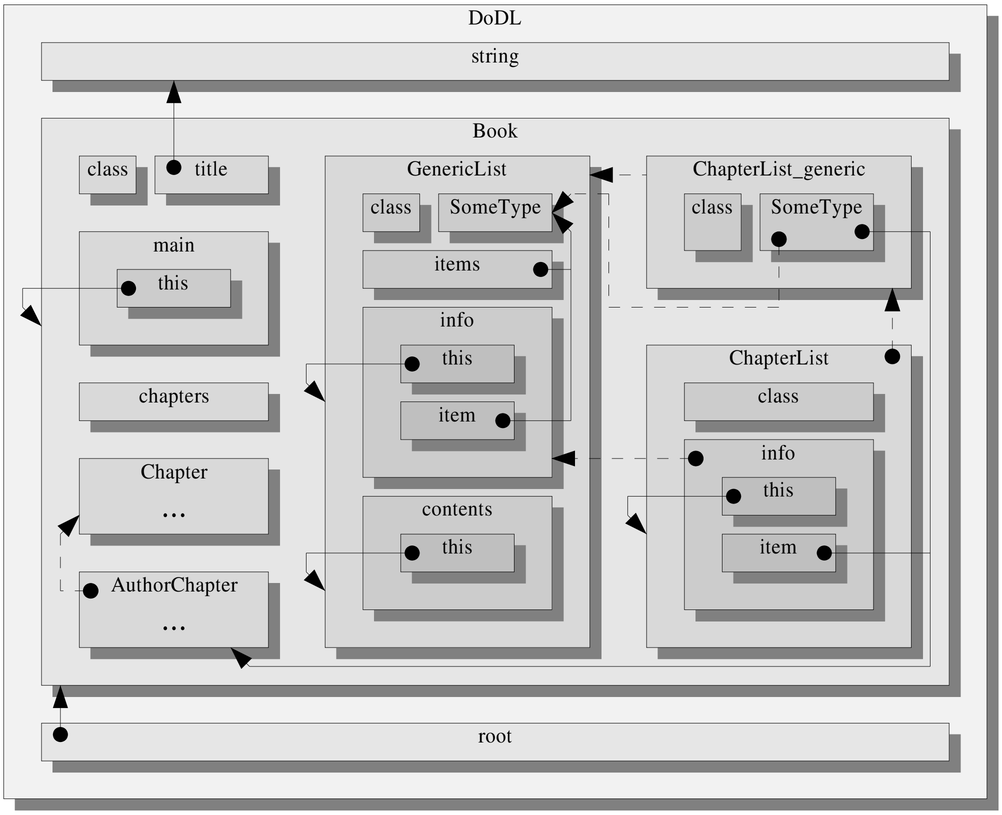

Abbildung 6.12.: Symboltabelle für generische Klassen (tatsächlich)

bellen der beiden Klassen: Es treten keinerlei Probleme mit mehrdeutigen Bezeichnern auf, alle Methoden lassen sich eindeutig zuweisen. Offenbar greift der beschriebene Mechanismus.

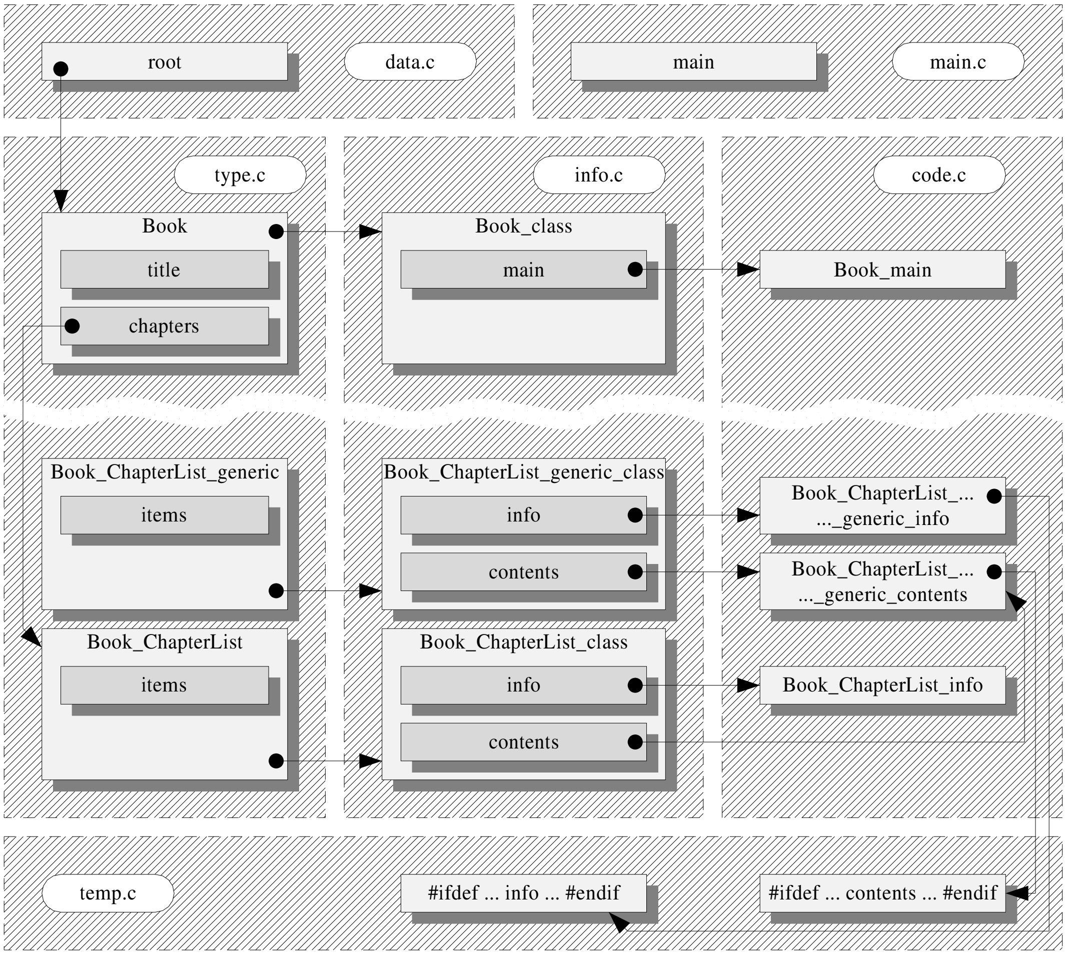

Abbildung 6.13.: Struktur des Zielcodes für generische Klassen

6.7.4. Inhalt des Zielcodes

Werfen wir abschließend noch einen kleinen Blick auf den Zielcode, der für unsere Zwischenklasse `Book__ChapterList__generic` erzeugt wird. Quellcode 6.21 entstammt der der Datei `temp.c`, in welche die Methodenschablonen geschrieben werden.

Der Ausschnitt zeigt, wie das Makro `DoDL__Book__GenericList__SomeType` verwendet wird, um gezielt diesen Teil der Datei per `#ifdef` auszuwählen. Anderenfalls müßte für jede generische Klasse eine eigene Datei erzeugt werden, was aus Sicht des C-Compilers sicherlich effizienter wäre, aber für den $DoDL_C$-Compiler mehr Verwaltungsaufwand mit sich bringen würde, da die Anzahl der Ausgabedateien nicht mehr fest wäre. Außerdem sehen wir, wie hilfreich es ist, daß wir das Makro `scope` verwenden, um die Bezeichner von $DoDL_C$ in den Na-

```
#ifdef DoDL__Book__GenericList__SomeType // Diesen Teil nur compilieren,
                                         // wenn richtiges Makro
                                         // definiert.
...
void scope(info)(void *__this, DoDL__Book__GenericList__SomeType item) {
 }

void scope(contents)(void *__this) {   // Makro "scope" sorgt fuer
 {                                     // tatsaechlichen Namen der
   DoDL__int i;                        // Funktion im Programm.

   for(i = 0; i < size(this->items); i++) {
     call(this, info)(this, this->items[i]);
   }
}
...
#endif
```

Quellcode 6.21: Ausschnitt aus der Datei `temp.c`

mensraum von C abzubilden: Es entspricht der Anforderung, daß der Name einer generischen Methode (im C-Code) davon abhängt, von welcher nicht-generischen Klasse sie eingebunden wird.

Quellcode 6.22 entstammt der Datei `code.c`. Er zeigt, wie die einzelnen Makros definiert sind und wie anschließend die Datei `temp.c` über `#include` eingebunden wird. Wie bereits erwähnt, wird vom Präprozessor nicht wirklich die ganze Datei eingebunden, sondern nur der gewünschte Ausschnitt.

```
// Zunaechst die Zwischenklasse bauen...

#define class DoDL__Book__ChapterList__generic
#define scope(id) DoDL__Book__ChapterList__generic ## __ ## id
#define DoDL__Book__GenericList__SomeType DoDL__Book__AuthorChapter
...
#include <temp.c>  // Einbinden der passenden generischen Schablone.
                   // Damit ist die Zwischenklasse schon fertig.
...
#undef class
#undef scope
#undef DoDL__Book__GenericList__SomeType

// Die tatsaechliche Klasse ist Nachkomme der Zwischenklasse...

#define class DoDL__Book__ChapterList
#define scope(id) DoDL__Book__ChapterList ## __ ## id
...
void scope(info)(void *__this, DoDL__Book__AuthorChapter item) {
  call(item, head)(item);
  call(item, body)(item);
}

#undef class
#undef scope
...
```

Quellcode 6.22: Ausschnitt aus der Datei `code.c`

7. Der Quellcode im Detail

Dieses Kapitel beleuchtet den Quellcode des $DoDL_C$-Compilers im Detail. Alle zum Compiler gehörenden Module, deren Aufgaben sowie die wesentlichen enthaltenen Datenstrukturen und Funktionen werden in jeweils einem gesonderten Abschnitt beschrieben. Das Kapitel richtet sich an Leser, die Änderungen am Compiler vornehmen oder den Compiler für eine spezielle (bislang nicht unterstützte) Plattform übersetzen wollen.

Der Quellcode folgt – soweit dies möglich und sinnvoll ist – den folgenden einfachen Konventionen:

- Jedes Modul, zu dem eine Header-Datei existiert, bindet diese per `#include` ein.
- Bezeichner für Typen beginnen stets mit Großbuchstaben (also etwa `Symbol` für den Basistyp der Symboltabelle).
- Bezeichner für globale Funktionen und Variablen beginnen stets mit einem Präfix, der sich aus den ersten drei Zeichen des Moduls zusammensetzt, in dem sie deklariert sind (also etwa `symInstall` für die fundamentale Funktion der Symboltabelle).
- Mehrzeilige Kommentare stehen unmittelbar vor dem Quellcode, den sie kommentieren. Kommentare, die sich bis zum Zeilenende erstrecken, stehen in der Zeile, die sie kommentieren.

Ausgenommen von der Namenskonvention sind Bezeichner, die von den Werkzeugen `flex` und `bison` erzeugt werden (also etwa die Funktion `yyparse`).

7.1. Die Projektdatei (`makefile`)

Dieser erste Abschnitt beschreibt die Projektdatei des $DoDL_C$-Compilers, das sogenannte `makefile`. Die Projektdatei enthält die Abhängigkeiten zwischen allen Quelldateien und die Schritte, die notwendig sind, aus den einzelnen Dateien den Compiler zu konstruieren. Die Projektdatei muß mit dem Werkzeug GNU `make` ausgeführt werden. Für andere Varianten dieses Programms sind wahrscheinlich Anpassungen erforderlich, da die Datei bedingte Anweisungen enthält, deren Syntax nicht festgelegt ist.

Abbildung 7.1 zeigt die wesentlichen Beziehungen zwischen dem `makefile` und den anderen Dateien. Alle Dateien sind als Rechtecke dargestellt. Die beiden gestrichelten Rechtecke repräsentieren temporäre Dateien, die von den Werkzeugen `flex` und `bison` erzeugt werden. Ein Pfeil symbolisiert eine Quelltext-Abhängigkeit in Pfeilrichtung. Das Modul, von dem der Pfeil ausgeht, benutzt also Teile des Moduls, auf welches der Pfeil verweist. Zur Verbessung der

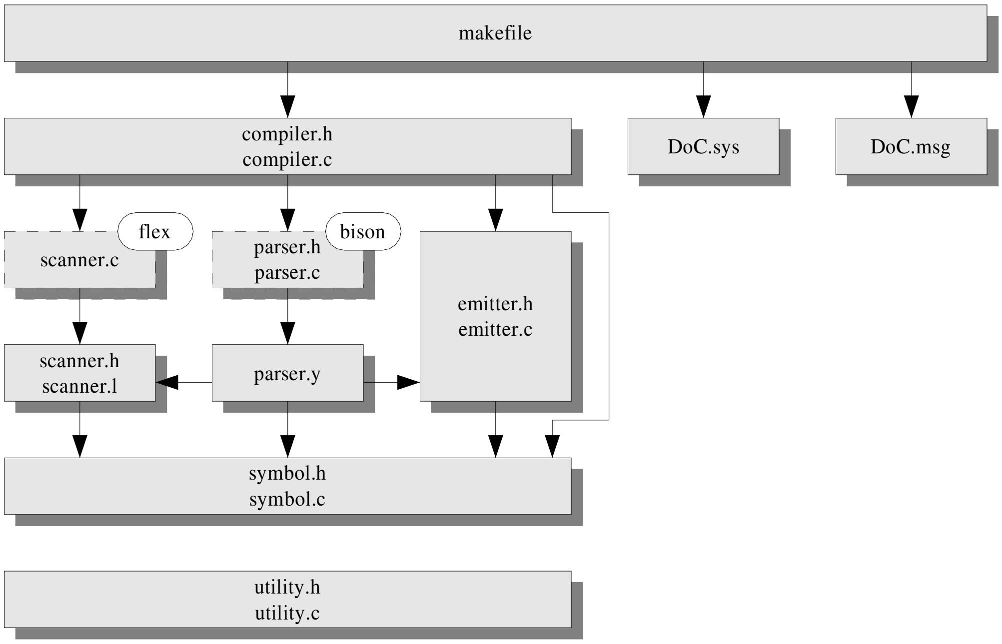

Abbildung 7.1.: Abhängigkeiten der Quelldateien

Übersicht wurden die Abhängigkeiten zu den Dateien `utility.h` und `utility.c` weggelassen. Alle C-Quellcode-Dateien sind von diesen beiden Dateien abhängig.

Beim Ausführen erkennt das `makefile` zunächst, ob es unter einem Unix-System oder OS/2 ausgeführt wird und setzt einige Dateinamen entsprechend. Anschließend werden die Aktionen ausgeführt, die den Compiler erzeugen. Ist die Übersetzung des Compilers erfolgreich, dann kopiert das `makefile` diesen sowie die Dateien `DoC.sys` und `DoC.msg` in das Verzeichnis `bin`, das sich unterhalb des $DoDL_C$-Basisverzeichnisses befindet. Tabelle 7.1 enthält eine Liste der möglichen Aufrufe von `make` für den $DoDL_C$-Compiler. Die Quellcodes 7.1 bis 7.3 zeigen das komplette `makefile`.

7.2. Der Compiler (`compiler.c`)

Das Compiler-Modul `compiler.c` ist das oberste Modul in der Hierarchie der C-Quellcode-Dateien. Es verbindet die von den anderen Modulen bereitgestellten Datenstrukturen und Funktionen zu einem vollständigen, lauffähigen Programm. Außer der – zugegebenermaßen etwas länglichen – Funktion `main` und einigen Variablen zur Aufnahme von Dateinamen stellt dieses Modul keine wesentlichen Datenstrukturen oder Funktionen bereit.

Zu den Aufgaben von `main` gehört zum Beispiel die Analyse der Parameter, die dem aufgerufenen Programm übergeben wurden, das Initialisieren der verschiedenen Untermodule und das Anstoßen der Funktion `yyparse` zur syntaktischen Analyse sowie verschiedener Funktionen des Emitters zum Generieren von Zielcode. Ist das Übersetzen des $DoDL_C$-Quelltextes in C-Dateien erfolgeich, dann ruft der $DoDL_C$-Compiler als letzte Aktion einen ANSI-C-Compiler

```
###########################################################################
#
# "makefile" - A master project file for the DoDL compiler
#
# $Id: Quellcode.tex,v 1.6 1999/08/26 10:22:13 pleumann Exp $
#
# Usage:
#
#   make        does what is necessary
#   make clean  removes temporary files
#   make shred  removes temporary and resulting files
#   make all    rebuilds the whole project
#
# The first section defines several variables depending on the current
# machine. Please use GNU MAKE or some other tool that supports IFDEFs.
#
###########################################################################

ifdef OS2_SHELL

  #########################################################################
  # If this is an OS/2 machine, use the EMX port of the GNU C compiler.
  #########################################################################

  CC=gcc
  CP=copy
  RM=-del

  EXE=..\bin\DoC.exe
  SYS=..\bin\DoC.sys
  RTL=..\bin\DoC.rtl
  MSG=..\bin\DoC.msg

else

  #########################################################################
  # Assume a Unix machine and the GNU C compiler in the default case.
  #########################################################################

  CC=gcc
  CP=cp
  RM=-rm -f

  EXE=../bin/DoC
  MSG=../bin/DoC.msg
  SYS=../bin/DoC.sys
  RTL=../bin/DoC.rtl

endif
```

Quellcode 7.1: Das `makefile` des $DoDL_C$-Compilers (Teil 1 von 3)

```
###########################################################################
# Define object files. Please don't change makefile beyond this point.
###########################################################################

OBJ=parser.o scanner.o symbol.o emitter.o compiler.o utility.o

###########################################################################
# Define rules used for making the files.
###########################################################################

default:        $(EXE) $(MSG) $(SYS) $(RTL)

$(EXE): $(OBJ)
                $(CC) $(OBJ) -o $(EXE)

$(MSG): DoC.msg
                $(CP) DoC.msg $(MSG)

$(SYS): DoC.sys
                $(CP) DoC.sys $(SYS)

$(RTL): DoC.rtl
                $(CP) DoC.rtl $(RTL)

parser.c:       parser.y symbol.h emitter.h
                bison -o parser.c -d parser.y

symbol.o:       symbol.c symbol.h utility.h

emitter.o:      emitter.c emitter.h

parser.o:       symbol.o utility.o

utility.o:      utility.c utility.h

scanner.c:      scanner.l scanner.h parser.y emitter.h
                flex -oscanner.c scanner.l
```

Quellcode 7.2: Das `makefile` des $DoDL_C$-Compilers (Teil 2 von 3)

```
############################################################################
# Define special rules used to clean up and remake everything.
############################################################################

clean:
                $(RM) parser.c
                $(RM) parser.h
                $(RM) scanner.c
                $(RM) *.o

shred:          clean
                $(RM) $(EXE)
                $(RM) $(MSG)
                $(RM) $(SYS)
                $(RM) $(RTL)

all:            shred default

############################################################################
# End of file.
############################################################################
```

Quellcode 7.3: Das `makefile` des $DoDL_C$-Compilers (Teil 3 von 3)

Aufruf	Erläuterung
`make`	Führt die nötigen Schritte zum Konstruieren des Compilers aus.
`make all`	Erzwingt die Neuübersetzung aller Dateien, unabhängig davon, ob dies notwendig ist.
`make clean`	Löscht alle Objektdateien und andere temporäre Dateien, die während der Übersetzung erzeugt wurden.
`make shred`	Entspricht dem Aufruf `make clean`, löscht aber zusätzlich den erzeugten Compiler sowie die Dateien `DoC.sys` und `DoC.msg` aus dem Binärverzeichnis.

Tabelle 7.1.: Aufrufmöglichkeiten von `make`

zum Erzeugen eines lauffähigen Programms auf.

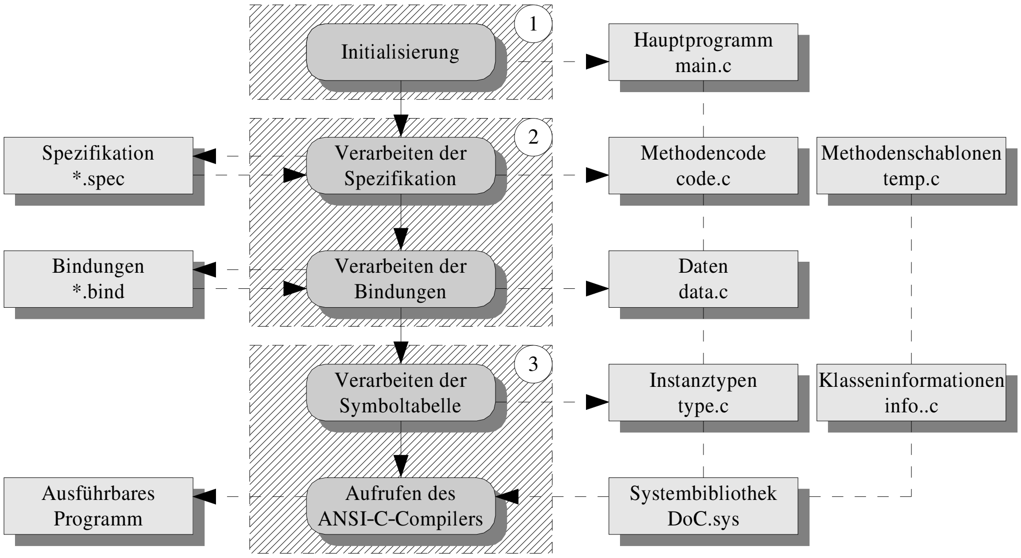

Abbildung 7.2.: Daten- und Kontrollfluß im $DoDL_C$-Compiler

Abbildung 7.2 zeigt die wesentlichen Schritte, die während eines Compiler-Laufes ausgeführt werden. Abgerundete Rechtecke repräsentieren Aktionen des Compilers, spitze Rechtecke stehen für die Symboltabelle und die verschiedenen Ausgabedateien. Durchgezogene Pfeile markieren den Kontrollfluß, und gestrichelte Pfeile symbolisieren den Datenfluß innerhalb des Systems.

7.3. Der Scanner (`scanner.h`, `scanner.l`)

Dieser Abschnitt beschreibt den Scanner des $DoDL_C$-Compilers, also den Teil, der für die lexikalische Analyse zuständig ist. Der Quellcode des Scanners ist in den Dateien `scanner.h` und `scanner.l` enthalten. Das Werkzeug `flex` wird benutzt, um aus `scanner.l` die Datei `scanner.c` zu generieren.

Der Aufbau des Scanners ist weitgehend kanonisch: Es werden reguläre Ausdrücke für Freiraum, Bezeichner, einzelne Zeichen, Zeichenketten und Zahlen definiert. Auf diesen regulären Ausdrücken werden Regeln aufgebaut, die das Verhalten des Scanners implementieren. Die meisten Regeln liefern ein erkanntes Token an den Parser, und zwar als Rückgabewert der Funktion `yylex`, die von `flex` generiert wird.

Das Lesen von Freiraum und Kommentaren hat zwar nicht die Rückgabe eines Tokens zur Folge, jedoch werden diese Teile der Eingabe im Gegensatz zu den meisten anderen Scannern bei uns nicht einfach ignoriert. Stattdessen werden sie in einem speziellen Feld der – relativ komplex aufgebauten – Struktur für lexikalische Werte gesammelt und zusammen mit dem nächsten erkannten Token zurückgegeben. Der Parser verfügt also zu jedem Token, das der Scanner ihm liefert, auch über den kompletten vorangehenden Freiraum. Diese Vorgehensweise

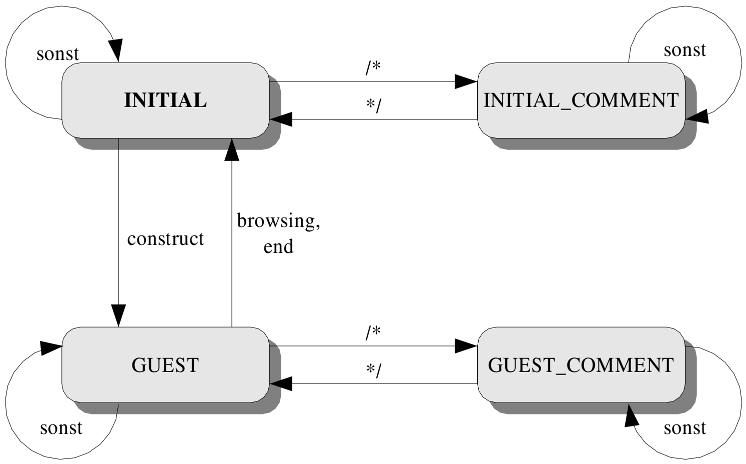

Abbildung 7.3.: Zustandsübergänge des Scanners

hat den Vorteil, daß sich die Formatierung und die Kommentare der Quelldatei auch im temporären C-Code wiederfinden, was die Fehlersuche in diesem Code wesentlich erleichtert.

Da die Wirtssprache $DoDL_C$ und die Gastsprache C lexikalisch disjunkt sind, war es zudem notwendig, mehrere Zustände einzuführen, in denen sich der Scanner befinden kann. Viele Regeln sind abhängig vom aktuellen Zustand; zum Beispiel werden die Token für die Feature-Terme nur im Bereich der Wirtssprache erkannt, nicht aber innerhalb der Gastsprache.

Zustand	Erläuterung
`INITIAL`	Dies ist der Startzustand des Scanners. Er repräsentiert die Wirtssprache $DoDL_C$.
`INITIAL_COMMENT`	Dieser Zustand wird zeitweise angenommen, wenn der Scanner einen mehrzeiligen Kommentar innerhalb der Wirtssprache $DoDL_C$ liest.
`GUEST`	Dieser Zustand repräsentiert die Gastsprache C. Er wird angenommen, solange sich der Scanner innerhalb der `construct`-Sektion befindet.
`GUEST_COMMENT`	Dieser Zustand wird zeitweise angenommen, wenn der Scanner einen mehrzeiligen Kommentar innerhalb der Gastsprache liest.

Tabelle 7.2.: Zustände des Scanners

Tabelle 7.2 erläutert die vier Zustände, in denen sich der Scanner befinden kann. Abbildung 7.3 zeigt zudem, bei welchen erkannten Token der Scanner einen Zustandswechsel vornimmt.

7.3.1. Datenstrukturen

Neben den internen, vom Werkzeug `flex` generierten Datenstrukturen – die hier nicht weiter beschrieben werden sollen – definiert der Scanner im wesentlichen die beiden strukturierten Typen `Expression` und `Token`.

Feld	Typ	Erläuterung
`caller`	`String`	Enthält den Teil eines Ausdrucks, der als Aufrufer gilt. Dieser erstreckt sich im allgemeinen bis unmittelbar vor den letzten Punkt-Qualifizierer des Ausdrucks. In den meisten Fällen entspricht dieser Wert dem von `Instance`. Ausnahme ist der Fall, daß eine Methode eines Vorfahren aufgerufen wird – dann verweist `caller` auf die Klasse des Vorfahren und `Instance` auf die Instanz, also das `this`.
`operator`	`String`	Enthält den Operator, der Aufrufer und Aufgerufenen voneinander trennt, meist ein Punkt.
`instance`	`String`	Enthält den Teil eines Ausdrucks, der als Instanz gilt. Dieser erstreckt sich im allgemeinen bis unmittelbar vor den letzten Punkt-Qualifizierer des Ausdrucks. In den meisten Fällen entspricht dieser Wert dem von `caller`. Ausnahme ist der Fall, daß eine Methode eines Vorfahren aufgerufen wird – dann verweist `caller` auf die Klasse des Vorfahren und `instance` auf die Instanz, also das `this`.
`callee`	`String`	Enthält den Teil eines Ausdrucks, der als Aufgerufener gilt. Dieser erstreckt sich meist vom letzten Punkt-Qualifizierer des Ausdrucks bis zu dessen Ende.
`instance`	`String`	Enthält den Teil eines Ausdrucks, der als Instanz gilt, für die der Ausdruck ausgewertet wird. Bei Methodenaufrufen ist dies der Teil, der im (implizit vorhandenen) Parameter `this` übergeben wird.
`type`	`Symbol`	Enthält den Typ des Ausdrucks, sofern dieser vom Parser ermittelt werden konnte.

Tabelle 7.3.: Aufbau des Typs `Expression`

Der Typ `Expression` dient zur Repräsentation beliebig komplexer Ausdrücke, die auf Bezeichnern basieren oder solche enthalten. Der Parser benötigt diese Struktur, um aus einfachen Bezeichnern zusammengesetzte Ausdrücke zu bilden, etwa beim Zugriff auf Unterdokumente oder Listenelemente sowie beim Aufruf von Methoden. Tabelle 7.3 erläutert die einzelnen Felder dieses Typs.

Der strukturierte Typ `Token` dient sowohl zur Rückgabe lexikalischer Werte vom Scanner an den Parser (ist somit identisch mit `YYSTYPE`) als auch zum Propagieren solcher Werte

Feld	Typ	Erläuterung
`lineNumber`	`int`	Die Zeilennummer, in der dieses Token gelesen wurde.
`space`	`String`	Sämtliche Leerzeichen und Kommentare, die diesem Token im Quelltext vorangehen.
`token`	`String`	Der Text des Tokens ohne vorangehende Leerzeichen und Kommentare.
`source`	`String`	Der Quelltext, der für dieses Token in den Zielcode geschrieben werden soll. Der Wert dieses Feldes entspricht anfänglich der Konkatenation von `Space` und `Token`, wird aber innerhalb des Parsers manipuliert. Zum Beispiel werden dort die `Source`-Werte mehrerer Token konkateniert, um komplexere Ausdrücke und größere Quellcode-Blöcke zu bilden.
`expr`	`Expression`	Enthält detaillierte Informationen zu lexikalischen Werten, die auf Bezeichnern basierende Ausdrücke repräsentieren. Der Scanner initialisiert diese Unterstruktur mit leeren Strings, innerhalb des Parser wird sie – möglicherweise – manipuliert. Tabelle 7.3 beschreibt diesen Typ.

Tabelle 7.4.: Aufbau des Typs `Token`

innerhalb des (nur implizit vorhandenen) Syntaxbaumes, der sich während der syntaktischen Analyse ergibt. Die einzelnen Felder der Struktur werden in Tabelle 7.4 detailliert beschrieben.

Schließlich besitzt der Scanner noch eine Variable `scaPreprocessing`, die widerspiegelt, ob im letzten Analyseschritt ein Token akzeptiert (`False`) oder Freiraum ignoriert (`True`) wurde. Die Variable wird benötigt, um zu entscheiden, ob gelesener Freiraum zum gleichen Token gehört oder nicht.

7.3.2. Funktionen

Der Scanner besitzt nicht viele Funktionen, die für die anderen Module oder für den Programmierer von Bedeutung sind. Wesentlich ist die von `flex` automatisch erzeugte Funktion `yylex`, die vom Parser aufgerufen wird, um ein Token zu lesen.

Zusätzlich definiert der Scanner zwei Makros, die von den einzelnen Regeln verwendet werden. Das Makro `ACCEPT` wird aufgerufen, wenn ein Token akzeptiert und an den Parser zurückgegeben werden soll. Es baut auf einer Funktion namens `scaDoAccept` auf. Das Makro `IGNORE` wird aufgerufen, wenn Freiraum oder Kommentare ignoriert werden sollen. Es baut auf einer Funktion namens `scaDoIgnore` auf.

7.4. Der Parser (parser.y)

In diesem Abschnitt werfen wir einen Blick auf den Parser des *DoDL*$_C$-Compilers, also den Bestandteil, der sich um die syntaktische Analyse kümmert. Der Quellcode des Parsers ist in der Datei `parser.y` enthalten. Wie man an der Endung dieser Datei unschwer erkennen kann, handelt es sich um eine Grammatik, die zunächst mit einem Parser-Generator – in unserem Fall GNU `bison` – in die Dateien `parser.h` und `parser.c` übersetzt werden muß, bevor der C-Compiler zum Einsatz kommen kann.

Der Quellcode des Parsers bietet trotz sorgfältiger Einrückung und großzügiger Kommentierung keinen besonders schönen Anblick, was im wesentlichen den zahlreichen semantischen Aktionen zu verdanken ist, mit denen die eigentliche Grammatik angereichert wurde. Der Parser trägt zum Beispiel die Verantwortung für die Steuerung der Symboltabelle, er ist also dafür zuständig, neue Klassen, Dokumente etc. in die Symboltabelle einzutragen. Der Parser übernimmt auch einen Teil der Code-Generierung, und zwar genau den Teil, der sich mit dem Erzeugen des Zielcodes der `construct`-Sektion beschäftigt. Diese Aufgabe kommt ihm zu, weil die `construct`-Sektion ja bereits als C-Code vorliegt, den wir nicht bis ins kleinste Detail analysieren wollen – schließlich war es nicht unser Ziel, einen kompletten C-Compiler in den *DoDL*$_C$-Compiler zu integrieren. Der Parser schreibt also innerhalb der `construct`-Sektion den gelesenen Quellcode einfach mit „kleinen Änderungen“ in die Ausgabedatei.

Die Grafiken 7.4, 7.5, 7.6 und 7.7 zeigen die Grammatik des *DoDL*$_C$-Compilers. Die Grammatik ist in LALR(1), wie von `bison` gefordert. Die nichtterminalen Symbole `ID` und `STR` repräsentieren einen Bezeichner und ein String-Literal. Das Symbol `ANYTHING` wird innerhalb der `construct`-Sektion benutzt, um ein Quellcode-Element zu repräsentieren, das der *DoDL*$_C$-Compiler nicht weiter beachtet und unverändert in den Zielcode schreibt.

7.4.1. Datenstrukturen

Neben den internen, von `bison` zur Realisierung des LALR(1)-Algorithmus generierten Datenstrukturen verwendet der Parser nur Datenstrukturen, die von anderen Modulen bereitgestellt werden: Er greift massiv auf die Symboltabelle zu (siehe Abschnitt 7.6), und er benutzt die vom Scanner definierten Typen `Expression` und `Token` (siehe Abschnitt 7.3) zum Propagieren von Werten innerhalb des (impliziten) Syntaxbaums.

7.4.2. Funktionen

Die wesentliche, nach außen sichtbare Funktion des Parsers ist `yyparse`. Diese Funktion wird von `bison` aus der Grammatik erzeugt. Zusätzlich befinden sich innerhalb des Parser-Moduls `parser.y` einige Funktionen, die sehr umfangreiche semantische Aktionen kapseln. Die Implementierung befindet sich am Ende der Datei, wodurch die eigentliche Grammatik lesbarer wird. Für diese Funktionen gilt eine einfache Namenskonvention: Die semantische Aktion einer (hypothetischen) Regel `Something` befindet sich in der Funktion `parDoSomething`. Möglicherweise wird ein numerischer Suffix angehängt, wenn mehrere semantische Aktionen für ein Nichtterminal existieren. In diesem Fall würden die Funktionen also `parDoSomething1`, `parDoSomething2` etc. heißen.

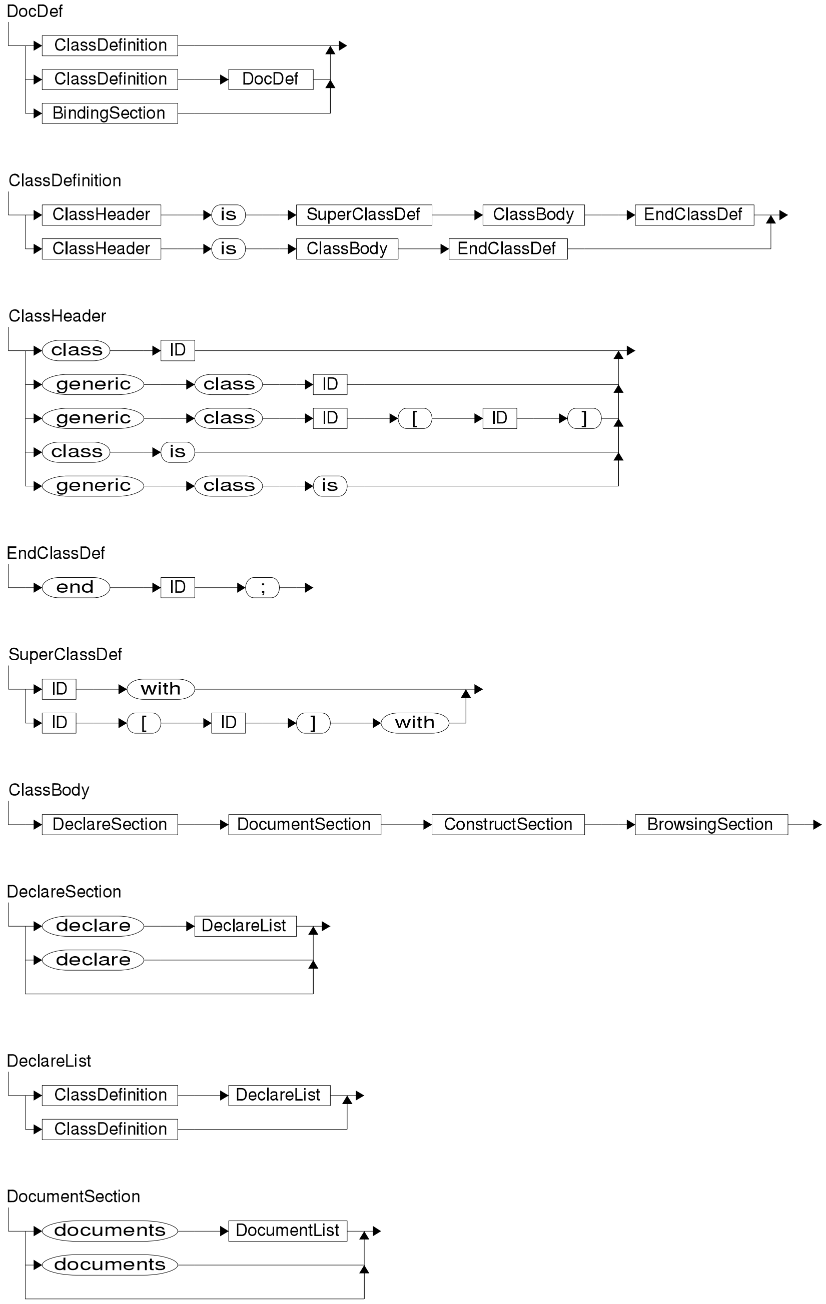

Abbildung 7.4.: Grammatik des $DoDL_C$-Compilers (Teil 1 von 4)

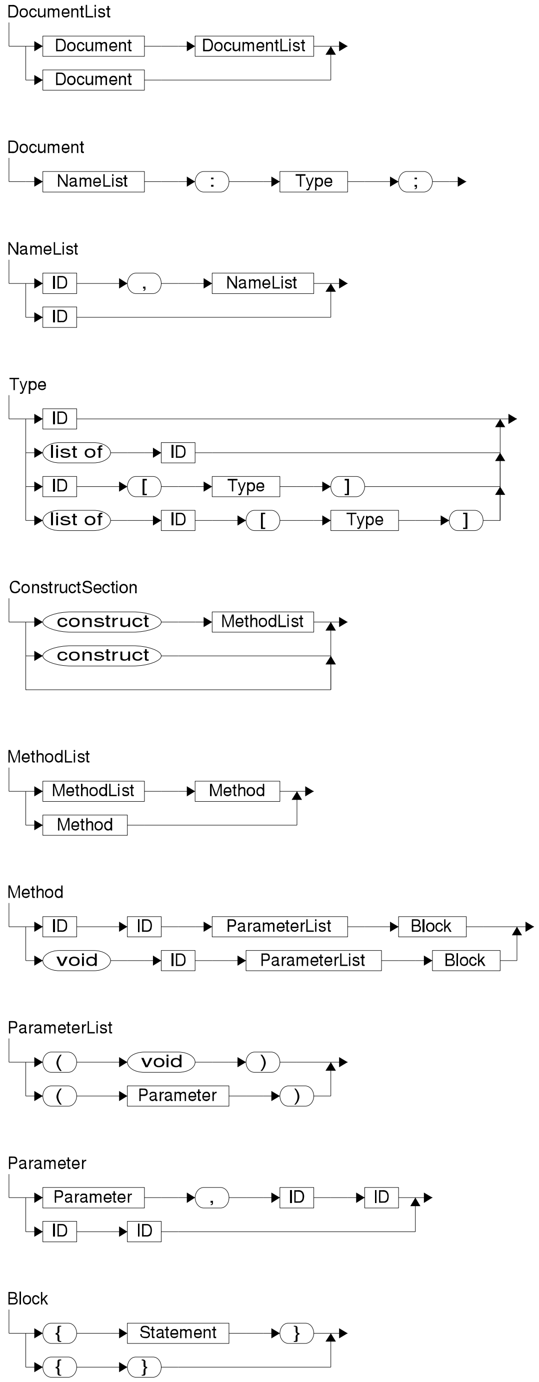

Abbildung 7.5.: Grammatik des $DoDL_C$-Compilers (Teil 2 von 4)

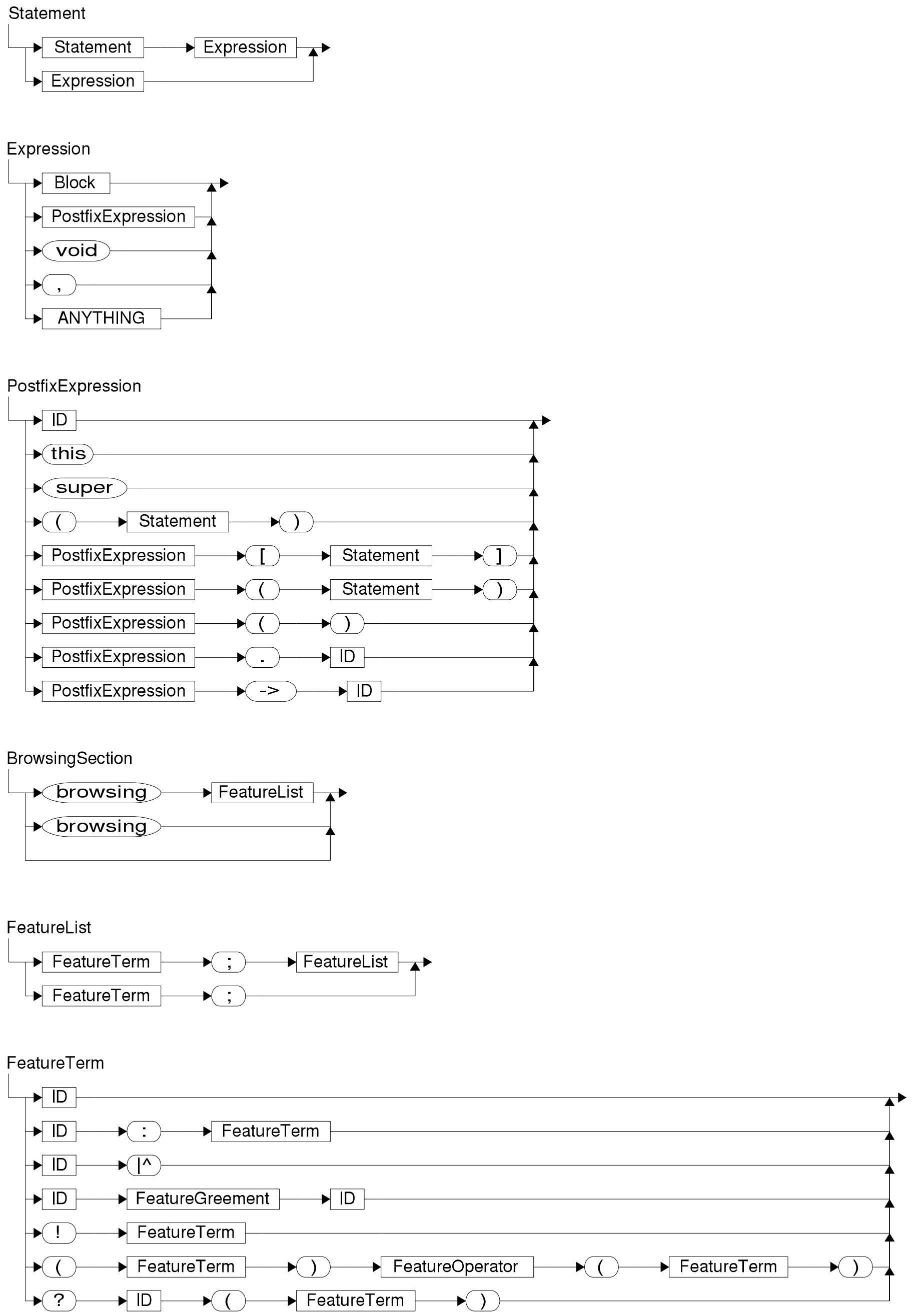

Abbildung 7.6.: Grammatik des $DoDL_C$-Compilers (Teil 3 von 4)

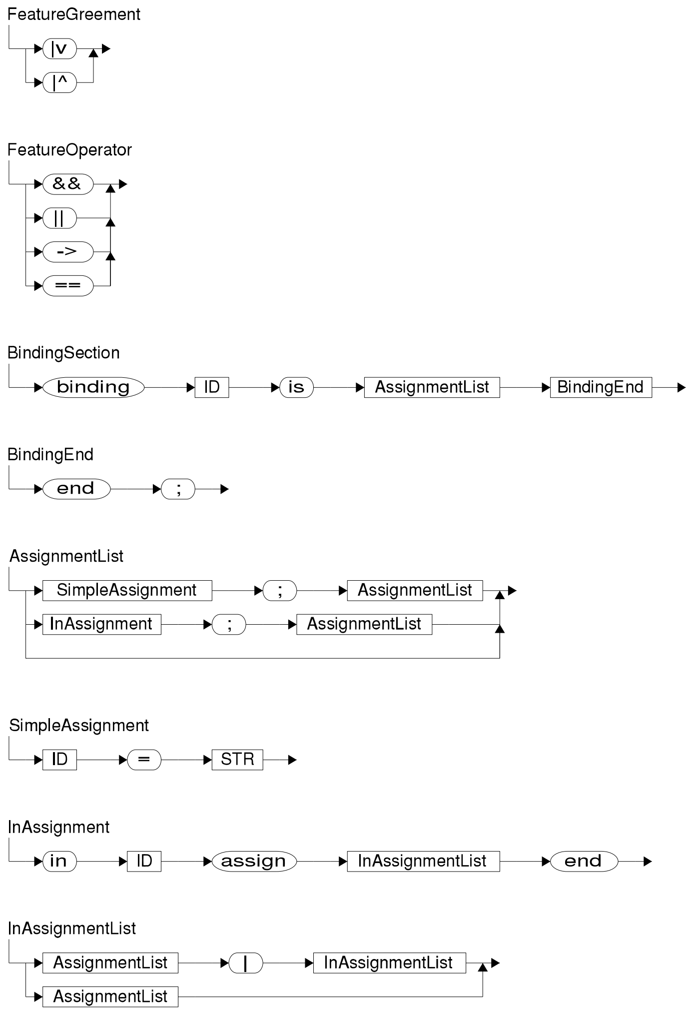

Abbildung 7.7.: Grammatik des $DoDL_C$-Compilers (Teil 4 von 4)

7.5. Der Emitter (`emitter.h`, `emitter.c`)

Der Emitter ist der für die Generierung des Zielcodes zuständige Teil des $DoDL_C$-Compilers. Alle Ausgaben in die verschiedenen erzeugten Dateien laufen entweder direkt oder indirekt über den Emitter. Das Modul stellt dazu sowohl eine Reihe von Low-Level-Funktionen zur Ausgabe von Zeichenketten, Umbrüchen und Kommentaren bereit, als auch Funktionen auf höherem Abstraktionsniveau, welche selbständig die Symboltabelle traversieren und spezielle Strukturen in C-Code übersetzen. Der Quellcode des Emitters befindet sich in den Dateien `emitter.h` und `emitter.c`.

7.5.1. Datenstrukturen

Der Emitter selbst stellt keine besonderen Datenstrukturen bereit, abgesehen von einigen Variablen, in denen er zum Beispiel über die gerade geöffnete Quelldatei Buch führt. Einige Funktionen des Emitters greifen jedoch auf die Symboltabelle zu. Um diese Funktionen zu verstehen, ist die Kenntnis der Datenstrukturen der Symboltabelle (siehe Abschnitt 7.6) erforderlich.

7.5.2. Funktionen

Das Emitter-Modul definiert eine Reihe von Funktionen zur Ausgabe des Zielcodes. Diese Funktionen lassen sich – wie bereits erwähnt – anhand des Abstraktionsniveaus, auf dem sie operieren, gliedern. Die Funktionen zur Ausgabe von primitiven Code-Elementen (etwa Zeichenketten) werden vom Parser aufgerufen, während dieser die `construct`-Sektion verarbeitet. Die Funktionen zur Ausgabe komplexerer Strukturen (etwa aller Strukturen zur Repräsentation von Objekten) werden vom (Haupt-) Compiler-Modul aus angestoßen, nachdem die syntaktische Analyse beendet ist. Die Tabellen 7.5 und 7.6 geben einen Überblick über die Funktionen des Emitter-Moduls.

7.6. Die Symboltabelle (`symbol.h`, `symbol.c`)

Wie in einem Compiler üblich, dient die Symboltabelle zur Verwaltung aller Bezeichner, die in einem Programm – in unserem Fall einer Kombination aus $DoDL_C$-Spezifikation und Bindungsdatei – auftreten. Die Symboltabelle verwaltet den Namen und den Typ jedes Symbols sowie dessen Beziehungen zu anderen Symbolen (etwa Vererbung). Außerdem sorgt die Symboltabelle dafür, daß Bezeichner eindeutig bleiben, wobei dies zugegebenermaßen eine ihrer trivialeren Aufgaben ist. Die Symboltabelle muß auch dem Umstand Rechnung tragen, daß innerhalb von $DoDL_C$ hierarchische Namensräume existieren, woraus sich eine – je nach Spezifikation beliebig umfangreiche – baumartige Struktur ergibt. Dementsprechend ist die Symboltabelle auch als Baum implementiert, bildet also die Namensräume der Sprache strukturell nach.

Die Symboltabelle ist der Dreh- und Angelpunkt des gesamten Compilers, da hier alle wesentlichen Informationen zusammenlaufen. Informationen für die Symboltabelle werden

Funktion	Erläuterung
`emiInitialize`	Initialisiert den Emitter. Diese Funktion setzt im wesentlichen den Basispfad für die Ausgabedateien (normalerweise das Verzeichnis `tmp` unterhalb des $DoDL_C$-Basisverzeichnisses) und löscht die eventuell vorhandenen Ausgabedateien eines vorherigen Compiler-Laufes.
`emiTerminate`	Schließt die Benutzung des Emitters ab. Diese Funktion wird am Ende des Übersetzungprozesses aufgerufen. Sie erzeugt die Hauptdatei `main.c`.
`emiSetFile`	Wechselt die Ausgabedatei. Diese Funktion wird aufgerufen, um die folgenden Ausgaben des Emitters in eine andere Datei zu lenken. Es wird nur der Hauptteil des Dateinamens angegeben (etwa `type`), Verzeichnis und Suffix werden automatisch angehängt (etwa `/dodl/tmp/type.c`).
`emiWriteString`	Schreibt einen String in die aktuelle Zieldatei.
`emiWriteStringFmt`	Schreibt einen formatierten String in die aktuelle Zieldatei. Diese Funktion formatiert den angegebenen String zunächst analog zu `sprintf` und ruft dann `emiWriteString` auf.
`emiWriteNewLine`	Schreibt einen Zeilenumbruch in die aktuelle Zieldatei. Die Benutzung dieser Funktion ist dem direkten Schreiben eines Zeilenumbruches vorzuziehen, da die Funktion erkennt, ob die Ausgabe innerhalb eines Kommentares stattfindet, und entsprechend reagiert (d.h. für die entsprechende Einrückung sorgt, was lesbareren Code ergibt).
`emiWriteSourceInfo`	Schreibt Informationen in die aktuelle Zieldatei. Diese Funktion sollte nur vom Parser aufgerufen werden. Sie schreibt Namen und aktuelle Zeilennummer der Quelldatei mit einer `#line`-Anweisung in die Ausgabedatei, wodurch Fehler besser lokalisiert werden können.
`emiWriteCommentBegin`	Schreibt den Beginn eines Kommentars in die Ausgabedatei und schaltet auf den Kommentarmodus um. Innerhalb dieses Modus arbeitet die Funktion `emiWriteNewLine` anders (siehe dort).
`emiWriteCommentEnd`	Schreibt das Ende eines Kommentars in die Zieldatei und schaltet auf den normalen Ausgabemodus zurück.

Tabelle 7.5.: Funktionen des Emitters (Teil 1 von 2)

Funktion	Erläuterung
`emiWriteClassForwards`	Schreibt Forward-Deklarationen für Klasseninformationen in die aktuelle Ausgabedatei. Aufgrund der teilweise rekursiven Struktur des generierten Codes ist die Ausgabe dieser Informationen notwendig, bevor `emiWriteClasses` die eigentlichen Klasseninformationen erzeugt. Die Funktion wird vom Compiler-Modul aus angestoßen; als aktueller Parameter wird die Wurzel der Symboltabelle übergeben.
`emiWriteClasses`	Schreibt Informationen über alle in der Symboltabelle enthaltenen Klassen in die aktuelle Ausgabedatei. Die Funktion wird vom Compiler-Modul aus angestoßen; als aktueller Parameter wird die Wurzel der Symboltabelle übergeben.
`emiWriteMethods`	Schreibt Informationen über die Methoden einer Klasse in die aktuelle Zieldatei. Diese Funktion schreibt die (lokalen) Namen und Signaturen aller Methoden der übergebenen Klasse. Sie wird von `emiWriteClassInfos` aufgerufen.
`emiWriteMethodTable`	Schreibt Informationen über die Methoden einer Klasse in die aktuelle Zieldatei. Diese Funktion bindet die (lokalen) Namen der Funktionen einer Klasse über eine Tabelle an (globale) Implementierungen von Funktionen. Das Erzeugen der Tabelle ist notwendig, damit die Methodenzuteilung zur Laufzeit funktionieren kann. Die Funktion wird von `emiWriteClassInfos` aufgerufen.
`emiWriteObjectForwards`	Schreibt Forward-Deklarationen für die Objekttyp-Deklarationen in die aktuelle Ausgabedatei. Diese Funktion arbeitet analog zu `emiWriteClassForwards`.
`emiWriteObjects`	Schreibt die Objekttyp-Deklarationen für alle in der Symboltabelle enthaltenen Klassen in die aktuelle Ausgabedatei. Die Funktion wird vom Compiler-Modul aus angestoßen; als aktueller Parameter wird die Wurzel der Symboltabelle übergeben.
`emiWriteDocuments`	Schreibt die Namen und Typen der Dokumente der übergebenen Klasse in die aktuelle Zieldatei. Diese Funktion wird von `emiWriteObjects` aufgerufen.

Tabelle 7.6.: Funktionen des Emitters (Teil 2 von 2)

weitgehend von Scanner und Parser bereitgestellt, wobei dem Parser die Steuerung der Symboltabelle obliegt, und später vom Emitter für die Erzeugung des Zielcodes ausgewertet.

Die Implementierung der Symboltabelle befindet sich in den beiden Dateien `symbol.h` und `symbol.c`.

7.6.1. Datenstrukturen

Die beiden wesentlichen Datenstrukturen zur Realisierung der Symboltabelle sind die Typen `Symbol` und `Scope`. Der Typ `Symbol` repräsentiert einen Eintrag der Symboltabelle mit allen zugehörigen Informationen. Unter anderem existieren Verweise auf das übergeordnete Symbol (bzw. den übergeordneten Namensraum), den nächsten Bruder des Symbols (im gleichen Namensraum) und das erste Kind des Symbols (das den ersten untergeordneten Namensraum bildet). Der Typ `Scope` wird verwendet, um innerhalb von Spezifikation oder Bindungsdatei betretene und noch nicht wieder verlassene Namensräume zu verwalten. Dadurch ergibt sich ein Stapel von Namensräumen, die automatisch traversiert werden, wenn ein Symbol gesucht wird.

Die Tabellen 7.7 und 7.8 erläutern den Aufbau des Typs `Symbol` im Detail. Tabelle 7.9 zeigt die möglichen Werte des Aufzählungstyps `SymbolKind`, mit welchem die Art eines Symbols festgelegt wird. Tabelle 7.10 enthält den Aufbau des Typs `Scope`.

7.6.2. Funktionen

Es existiert eine Reihe von Funktionen zur Verwaltung der Symboltabelle und der Namensräume. Die Tabellen 7.11 und 7.12 enthalten eine Übersicht mit einer kurzen Aufgabenbeschreibung jeder Funktion.

7.7. Das Hilfsmodul (`utility.h, utility.c`)

Das unterste Modul in der Hierarchie der C-Quellcodes des Compilers stellt grundsätzliche Datentypen und Funktionen bereit. Diese dienen im wesentlichen dazu, einige Unzulänglichkeiten[1] von C zu umschiffen. Als ein Beispiel sei hier die in C nur ausgesprochen rudimentär unterstützte und dementsprechend fehleranfällige Verarbeitung von Strings genannt.

7.7.1. Datenstrukturen

Tabelle 7.13 zeigt die Basistypen, die von diesem Modul definiert werden.

7.7.2. Funktionen

Tabelle 7.14 enthält eine Auflistung der verschiedenen Hilfsfunktionen des Moduls.

[1] Zugegebenermaßen ein ziemlicher Euphemismus.

Feld	Typ	Erläuterung
`localName`	`String`	Enthält den lokalen Namen des Symbols. Dieser ist innerhalb des Namensraumes, in dem das Symbol deklariert ist, eindeutig. Der lokale Name wird direkt dem $DoDL_C$-Quellcode entnommen.
`globalName`	`String`	Enthält den globalen Namen des Symbols. Dieser ist innerhalb der gesamten Symboltabelle eindeutig. Für Klassen setzt sich der globale Name zum Beispiel aus der Konkatenation der Namen aller umschließenden Klassen zusammen. Der globale Name wird benötigt, um eine Abbildung der hierarchischen Namensräume von $DoDL_C$ in den flachen Namensraum von C zu schaffen.
`kind`	`SymbolKind`	Gibt die Art des Symbols an. Tabelle 7.9 zeigt die möglichen Werte.
`type`	`Symbol`	Wird nur für Dokumente und Parameter von generischen Klassen verwendet. Enthält einen Verweis auf das Symbol, das den Typs des Dokumentes oder Parameters beschreibt.
`classAncestor`	`Symbol`	Wird nur für Klassen verwendet. Enthält einen Verweis auf die Klasse, die unmittelbarer Vorfahr dieser Klasse ist. Alle Klassen außer der internen Basisklasse `Object` besitzen einen Vorfahren.
`signature`	`String`	Wird nur für Methoden verwendet. Enthält den gesamten Kopf der Methode in textueller Form, so wie er später in den Zielcode geschrieben werden soll.
`isOverride`	`Boolean`	Wird nur für Methoden verwendet. Gibt an, ob die Methode eine Methode gleichen Namens im Vorfahren überschreibt.
`isInternal`	`Boolean`	Wird nur für interne Zwecke verwendet. Wenn das Feld den Wert `True` hat, wird für das Symbol kein Code generiert.

Tabelle 7.7.: Aufbau des Typs `Symbol` (Teil 1 von 2)

Feld	Typ	Erläuterung
`parent`	`Symbol`	Enthält eine Referenz auf das übergeordnete Symbol, also den Namensraum, in dem dieses Symbol deklariert wurde.
`next`	`Symbol`	Enthält eine Referenz auf den nächsten Bruder des Symbols im gleichen Namensraum. Über dieses Feld sind alle Kinder eines Symbols zu einer Liste verkettet.
`firstChild`	`Symbol`	Enthält eine Referenz auf das erste Kind eines Symbols, also das erste untergeordnete Symbol, das im Namensraum dieses Symbols deklariert wurde.
`lastChild`	`Symbol`	Enthält eine Referenz auf das letzte Kind eines Symbols, also das untergeordnete Symbol, das zuletzt im Namensraum dieses Symbols deklariert wurde. Dieses Feld wird nicht wirklich benötigt. Es wurde aber aus Effizienzgründen eingeführt, damit neue Symbole schneller (am Ende) der Kinderliste angehängt werden können.

Tabelle 7.8.: Aufbau des Typs `Symbol` (Teil 2 von 2)

7.8. Die Systembibliothek (`DoC.sys`)

Die Systembibliothek enthält das Laufzeitsystem eines $DoDL_C$-Programms. Sie stellt vordefinierte Typen, eine Basisklasse sowie einige unterstützende Funktionen (z.B. zur Verwaltung von Listen) zur Verfügung, ohne die keine $DoDL_C$-Spezifikation auskommt. Der Code der Systembibliothek ist unabhängig von der übersetzten Spezifikation, deshalb wäre es unpraktisch, diesen Code etwa vom Emitter generieren zu lassen. Stattdessen wurde der Weg gewählt, die Bibliothek über eine `#include`-Anweisung in jede übersetzte Spezifikation einzubinden. Diese Vorgehensweise hat auch den Vorteil, daß Änderungen an der Basisklasse nicht notwendigerweise Änderungen am Compiler nach sich ziehen.

Die Systembiliothek ist in der Datei `DoC.sys` enthalten.

7.9. Die Meldungsdatei (`DoC.msg`)

Die Meldungsdatei `DoC.msg` enthält sämtliche Texte, die während eines Compiler-Laufes auf dem Bildschirm ausgegeben werden können. Die Hilfsfunktionen `utiLoadStr`, `utiMessage` und `utiError` laden diese Texte und zeigen sie gegebenenfalls an. Durch das Auslagern der Meldungen in eine zentrale Datei wurde zum einen der Quellcode übersichtlicher, zum anderen kann die Meldungsdatei leicht angepaßt oder in eine andere Sprache übersetzt werden, ohne daß dazu der Compiler geändert oder gar neu übersetzt werden müßte.

Das Format des Meldungsdatei ist sehr einfach. Jede Meldung besteht aus einer numerischen Kennung, einem Leerzeichen und dem eigentlichen Meldungstext, der sich bis zum

Wert	Erläuterung
`skRoot`	Äußerstes Symbol sowohl der Symboltabelle als auch des hierarchischen Namensraums von $DoDL_C$. Wird nur intern verwendet.
`skKeyword`	Schlüsselwort der Gastsprache. Kann verwendet werden, um Schlüsselwörter vorab in die Symboltabelle einzutragen, so daß sie nicht versehentlich als Bezeichner benutzt werden können.
`skPrimitiveType`	Primitiver Typ, etwa `string` oder `integer`.
`skGenericClass`	Generische Klasse.
`skConcreteClass`	Konkrete (nicht-generische) Klasse.
`skFormalParam`	Formaler Parameter einer generischen Klasse. Findet sich in der generischen Klasse selbst wieder.
`skActualParam`	Aktueller Parameter einer generischen Klasse. Findet sich in der Klasse wieder, die den Parameter einer generischen Klasse mit einem Wert belegt.

`skUnknownDoc`	Ein Dokument, dessen Typ noch nicht bekannt ist. Alle Dokumente haben zunächst diese Kennzeichnung und bekommen erst später eine der Kennzeichnungen `skSingleDoc` oder `skListOfDoc`. Dadurch wird das Konstrukt
	`documents`
	`  A, B, C: TypeName;`
	leichter handhabbar.

`skSingleDoc`	Ein einzelnes Dokument.
`skListOfDoc`	Eine Liste von Dokumenten.
`skHyperDoc`	Ein Hyperdokument, also eine Instanz der Klasse, die auf der obersten Ebene der Bindungsdatei erzeugt wird (und deren Methode `main` aufgerufen wird). Aus jedem Paar von Spezifikation und Bindungsdatei kann nur genau ein Hyperdokument hervorgehen.
`skMethod`	Eine Methode.
`skParameter`	Ein Parameter einer Methode.
`skClassInfo`	Eine Struktur von Klasseninformationen. Wird nur intern verwendet.

Tabelle 7.9.: Aufbau des Typs `SymbolKind`

Feld	Typ	Erläuterung
`symbol`	`Symbol`	Enthält das Symbol, dessen Namensraum betreten wurde.
`next`	`Scope`	Referenziert den Namensraum, der unmittelbar zuvor betreten wurde. Dieser Namensraum wird wieder aktiv, wenn der aktuelle Namensraum verlassen wird. Effektiv bilden die Namensräume über dieses Feld einen Stapel, auch wenn dieser als Liste implementiert ist.
`instanceCount`	`int`	Enthält die Anzahl der Instanzen des Symbols, dessen Namensraum betreten wurde. Dieses Feld wird innerhalb der Bindungen für Listen verwendet. Bei jedem Auftreten eines senkrechten Striches wird der Instanzzähler um eins erhöht.

Tabelle 7.10.: Aufbau des Typs `Scope`

Funktion	Erläuterung
`symInitialize`	Initialisiert die Symboltabelle.
`symQueryRoot`	Gibt das äußerste Symbol des $DoDL_C$-Namensraums zurück, das gleichzeitig die Wurzel des Symboltabelle ist.
`symInstall`	Installiert ein neues Symbol in der Symboltabelle. Es werden nur Name und Art des Symbols festgelegt, alle weiteren Infomationen müssen nachträglich über spezialisierte Funktionen spezifiziert werden.
`symLookup`	Sucht ein Symbol anhand von dessen Namen. Es kann kontrolliert werden, welche Namensräume bei der Suche traversiert werden sollen. Die folgenden Werte sind möglich: • `lsLocal` – Durchsucht nur den aktuellen Namensraum. • `lsClass` – Durchsucht den gesamten Namensraum einer Klasse. Die Suche erstreckt sich also über alle Vorfahren bis zum Basisobjekt. • `lsGlobal` – Durchsucht zunächst den gesamten Namensraum der Klasse. Anschließend werden alle umschließenden Namensräume bis hin zur Wurzel der Symboltabelle durchsucht.

Tabelle 7.11.: Funktionen der Symboltabelle (Teil 1 von 2)

Funktion	Erläuterung
`symQueryFullName`	Erfragt den globalen Namen eines Symbols, also den eindeutigen Namen, der innerhalb des C-Zielcodes für das Symbol verwendet wird.
`symSetClassAncestor`	Weist einer Klasse einen Vorfahren zu.
`symSetFormalClassParam`	Weist einer generischen Klasse einen formalen Parameter zu.
`symSetActualClassParam`	Weist einer Klasse, die von einer generischen Klasse erbt, einen aktuellen Parameter zu.
`symSetDocumentType`	Weist allen bisher untypisierten Dokumenten im aktuellen Namensraum einen Typ zu.
`symQueryScope`	Gibt das Symbol zurück, dessen Namensraum zuletzt betreten wurde.
`symEnterScope`	Betritt den Namensraum eines Symbols.
`symLeaveScope`	Verläßt den Namensraum eines Symbols.
`symSetInstanceCount`	Ändert die Anzahl der Instanzen des Symbols, dessen Namensraum zuletzt betreten wurde.
`symQueryInstanceCount`	Erfragt die Anzahl der Instanzen des Symbols, dessen Namensraum zuletzt betreten wurde.

Tabelle 7.12.: Funktionen der Symboltabelle (Teil 2 von 2)

Funktion	Erläuterung
`Boolean`	Definiert einen Aufzählungstyp zur Repräsentation von Wahrheitswerten. Die beiden möglichen Werte sind `True` und `False`.
`Integer`	Entspricht dem Typ `int`.
`String`	Definiert einen Zeiger auf ein Array von Zeichen. Die meisten Funktionen zur String-Verarbeitung erwarten als ersten Parameter einen Zeiger auf diesen Typ, was (leider) die bestmögliche Approximation eines `var`-Parameters aus Pascal ist, die sich in C finden läßt.

Tabelle 7.13.: Datentypen des Hilfsmoduls

Funktion	Erläuterung
`utiInitialize`	Initialisiert das Modul mit den Hilfsfunktionen.
`utiNewStr`	Erzeugt einen neuen String. Die Funktion allokiert den nötigen Speicher automatisch.
`utiNewStrFmt`	Erzeugt einen neuen formatierten String. Die Funktion allokiert den nötigen Speicher automatisch.
`utiDisposeStr`	Gibt den Speicher für einen nicht mehr benötigten String frei.
`utiAssignStr`	Ändert den Inhalt eines Strings.
`utiAppendStr`	Hängt Text an das Ende eines bestehenden Strings an.
`utiPrependStr`	Hängt Text vor den Anfang eines bestehenden Strings.
`utiDeleteStr`	Löscht eine Anzahl von Zeichen aus einem String.
`utiInsertStr`	Fügt Text an einer wählbaren Position eines Strings ein.
`utiPos`	Ermittelt die Position des ersten Vorkommens eines Teilstrings in einem String.
`utiChangeFileExt`	Ändert den Suffix eines Strings, der einen Dateinamen enthält. Der Suffix erstreckt sich vom letzten Punkt bis zum Ende des Strings. Existiert bislang kein Suffix, wird er einfach angehängt.
`utiExtractFileDir`	Extrahiert den Teil eines Dateinamens, der das Verzeichnis repräsentiert.
`utiLoadStr`	Lädt einen String aus der Nachrichtendatei des Programms und formatiert ihn mit den gegebenen Argumenten.
`utiMessage`	Lädt einen String aus der Nachrichtendatei des Programms, formatiert ihn mit den gegebenen Argumenten und gibt das Ergebnis auf dem Bildschirm aus.
`utiError`	Lädt einen String aus der Nachrichtendatei des Programms, formatiert ihn mit den gegebenen Argumenten und gibt das Ergebnis auf dem Bildschirm aus. Anschließend wird das Programm beendet.

Tabelle 7.14.: Funktionen des Hilfsmoduls

Ende der Zeile erstreckt. Mehrzeilige Meldungen werden erreicht, indem in mehreren Zeilen die gleiche Kennung verwendet wird. Leerzeilen und Zeilen, die mit einem Semikolon beginnen, werden ignoriert.

Teil IV.

Ausblick

8. Ein paar abschließende Gedanken

Wir wollen diese Arbeit nicht schließen, bevor wir nicht einen Blick über den berühmten Tellerrand geworfen haben. Wir wollen auch nicht verheimlichen, was wir nicht können, geschweige denn, daß wir uns Gedanken über die Zukunft machen.

8.1. Was haben wir jetzt eigentlich?

Im Jahr 1996 entstand $DoDL_C$ als ein Vorschlag, Hyperdokumente zu spezifizieren, d.h. einen Graphen konstruktiv zu beschreiben, der die Verknüpfungsstruktur des gewünschten Hyperdokuments repräsentiert. Die Spezifikation ist objektorientiert aufgebaut. Durch Compilierung entsteht ein Programm, das ausführbar ist und den beschriebenen Graphen erzeugt. Dieser muß jedoch noch mit einem geeigneten Werkzeug in eine aktuell verfügbare Sprache übersetzt werden, die in einem gängigen Browser verarbeitet werden kann. Das ist nunmal der Weg, den alle Hyperdokumente gehen müssen. Der Vater des Gedankens will Methoden und Werkzeuge der Software-Technik nutzen, um von den Struktur und Inhalt vermischenden HTML-Konstrukten Abstand zu nehmen und große Systeme leicht handhabbar zu machen.

Im Laufe des Jahres 1998 entstand dann ein Compiler, der eine $DoDL_C$-Spezifikation nach Ansi-C übersetzt, so daß ein Programm entsteht, welches ..., aber das ist ja ausführlich beschrieben worden.

Damit existiert ein Compiler-System, daß Hyperdokument-Graphen erzeugen kann. Der jetzt noch fehlende Generator, also diejenige Software, die den Graphen traversiert und nach HTML übersetzt, ist im Jahre 1999 als Diplomarbeit des jüngeren der beiden Autoren entstanden. Zum Ende des Jahrtausends existiert ein Software-System, das die Machbarkeit des 1996 erdachten Ansatzes beweist, das die Möglichkeit aufzeigt, Hyperdokumente in Inhalt und Struktur systematisch auf hoher Sprachebene zu beschreiben und aus der Beschreibung die Dokumente zu generieren.

8.2. Was kann man noch machen?

Wenn man diese Frage stellt, sind die Antworten abzählbar, aber beinahe unendlich. Es ist noch vieles möglich, aber es sollte auch auf Sinnhaftigkeit überprüft werden. Meist entpuppen sich die vermeindlichen Antworten als Fragen:

- Gelingt es, nicht nur Texte und Graphiken einzubinden, sondern auch komplexe Medienobjekte wie Videos?
- Kann man strukturierte Dokumente, etwa Tabellen, genauer betrachten?

- Lassen sich auch Dateien mit bestimmten Formaten, zum Beispiel PDF, einbinden?

Und darüber hinaus:

- Kann man die Beschreibung des Graphen auf diese Dokumenttypen beziehen?
- Gelingt es also, eine PDF-Datei direkt zu verlinken und in einem geeigneten Browser zu betrachten?
- Dieselbe Frage läßt sich auch für dvi-Dateien stellen.

Die Kombination der beiden Fragenklassen endet in einer, nennen wir es *$DoDL_C$-Normalform*. Diese soll dann benutzt werden, um eine vom Dokumenttyp bzw. Dokumentformat und Browser unabhängige Beschreibung zu erhalten. Ein geeignetes Frontend übersetzt eine $DoDL_C$-Spezifikation, egal auf welchen Dokumenttypen und Formaten sie arbeitet, in diese Normalform, ein geeignetes Backend erzeugt entweder HTML oder *traversierbares PDF* oder anderes.

Ein anderer Fragenkomplex betrachtet Werkzeuge, die die Arbeit mit $DoDL_C$ unterstützen. Man sehne sich nach einem Debugger, der es zuläßt, den spezifizierten Graphen nach Eigenschaften zu untersuchen, ihn zu traversieren, bevor er zu HTML wird. Im Frontend kann man eine Unterstützung zur Modellierung mit $DoDL_C$ verlangen, etwa eine semiformale graphische Notation, verbunden mit einem syntaxgesteuerten Editor für diese Notation.

Das führt, konsequent weitergedacht, zu einer Entwicklungsumgebung für $DoDL_C$-Spezifikationen. Es wäre durchaus überlegenswert, die in der Projektgruppe $\mathcal{HEU}$ entstandene Architektur einer erweiterbaren Entwicklungsumgebung zu diesen Zwecken zu nutzen! Dazu finden sich in Abschnitt 8.5 ein paar Worte.

8.3. Was können wir nicht?

Wir sind keine Universalgenies. Und zugegeben, Texte zu verknüpfen ist noch ziemlich einfach. Strings suchen ist nämlich ein lineares Verfahren, den richtigen Algorithmus vorausgesetzt.

Bilder hingegen machen die Suche von Teilobjekten schon ziemlich kompliziert. Und Bilderkennungsalgorithmen zu implementieren, war nicht gerade unser vorrangiges Ziel. Dennoch, am Konzept ändert sich nichts. Für einen beliebigen Medienobjekttyp müssen „nur“ die Methoden `getBegin`, `getEnd` und `getOcc` implementiert werden, und schon sehen wir, wie $DoDL_C$ viele bunte Bilder miteinander verdrahtet.

Wir haben die Erweiterung von $DoDL_C$, die `browsing`-Sektion, nicht implementiert. Wir hätten dies „naiv“ tun können, indem wir einen Constraint-Solver für Feature-Terme [Zel97] von Andreas Zeller integiert hätten. Schaut man jedoch konzeptionell genauer hin, erheben sich eine Menge wichtiger Fragen, die erst noch geklärt werden müssen. Diese betreffen den Umgang mit den Termen innerhalb einer objektorientiert strukturierten Spezifikationshierarchie, nämlich

- wie vererbt man die `browsing`-Sektion?

aber auch ihren pragmatischen Sinn und Zweck, nämlich

- Dient die `browsing`-Sektion nur bei der Generierung des Graphen dazu, nur diejenigen Links zu erzeugen, die von den Feature-Termen erfaßt werden, oder soll diese Sektion dynamisch, sprich erst zur Traversierzeit ausgewertet werden?

Man kann Medienobjekte nicht nur strukturiert betrachten, sondern auch rekursiv zusammengesetzt. Ein Medienobjekte könnte beispielsweise aus einem Text gefolgt von einem Bild gefolgt von einem anderen Text bestehen. Der erste Text könnte eine Gruppierung zweier weiterer Texte sein, und so fort. Medienobjekte dieser Art sind in unserer Implementierung von $DoDL_C$ nicht handhabbar, hier gibt es lediglich `dbUnit` und die davon abgeleiteten, nicht-zusammengesetze Typen `text` und `graphics`. Wir können diese zusammengesetzten Strukturen jedoch nachbauen. Man muß lediglich fordern, daß eine Verknüpfungskaskade der Art $\mathsf{Ende(Text\ 1)} \rightarrow \mathsf{Beginn(Bild)}$; $\mathsf{Ende(Bild)} \rightarrow \mathsf{Beginn(Text)}$ usw. als ein komponiertes Medienobjekt betrachtet wird. Dann ist jedoch die Anwendung positionsfindender Methoden an diese „inhomogenen“ Medienobjekte anzupassen.

8.4. Was werden wir nie können?

$DoDL_C$ ist zwar eine objektorientiert arbeitende Sprache, jedoch ist sie akademischer Natur. Das bedeutet, daß sie mit der Absicht konstruiert wurde, einen Beitrag zum Verständnis der Konstruktion von Hyperdokumenten zu liefern. Die Sprache wurde also möglichst schlank und ohne Balast in Hinblick auf das genannte Ziel entworfen. Damit wurden nur die zur Zielerreichung nötigsten Sprachkonstrukte aufgenommen. Was der Sprache fehlt und sie dadurch von vielen anderen objektorientierten Sprachen unterscheidet, ist das Fehlen von Referenzen auf Objekte und deren Bestandteile und, damit verbunden, das Fehlen dynamisch erzeubarer Objekte. Wir können Instanzen nur über die `binding`-Sektion erzeugen, dies jedoch nur zur Compilezeit. Somit gelingt es uns nicht, Hyperdokumente zu beschreiben, deren Struktur nicht bereits zur Spezifikationszeit bekannt ist.

Ein Beispiel: Das beliebte Listenkonstrukt zur Erzeugung mehrerer Instanzen einer Klasse wird in der `binding`-Sektion mit Leben gefüllt. Hier entscheidet sich die Länge der Liste. Nun könnte man sich ein Hyperdokument wünschen, daß an jedes Vorkommen eines Strings in einem Text eine andere Instanz einer solchen Liste knüpft. Die Länge der Liste und damit die Anzahl der Instanzen wäre erst zur Compilezeit bekannt, da dann erst feststeht, wieviele Vorkommen des gesuchten Strings es denn nun gibt. Hyperdokumente, die solche erst zur Compilezeit bekannten Bedingungen an die Existenz von Instanzen stellen, sind in $DoDL_C$ nicht realisierbar.

8.5. Und dann ist da noch die $\mathcal{HEU}$

In den Semestern Winter '98/'99 und Sommer '99 hat eine Projektgruppe am Lehrstuhl für Software-Technologie stattgefunden, die das häre Ziel hatte, eine Hypermediaentwicklungsumgebung ($\mathcal{HEU}$) prototypisch zu implementieren. Die Idee dahinter war, ein grafisches Frontend für $DoDL_C$ zu schaffen, eine grafische Notation, die von syntaxgesteuerten Editoren unterstützt $DoDL_C$ modelliert und „per Knopfdruck“ in textuelles $DoDL_C$ übersetzt. Die Anforderungen an die Entwicklungsumgebung (EU) waren jedoch so umfangreich und konzeptionell höchst nicht-trivial, daß sich im Laufe der Zeit der Schwerpunkt auf die architektonischen Grundlagen der EU verschob. Will heißen, das grafische Frontend wurde nicht realisiert. Es entstanden lediglich kleine, zwar syntaxgesteuerte Editoren für jedoch „nur" eine UML-artige Notation. Diese Editoren freuten sich, die Machbarkeit des Architekturkonzepts mit all seinen Schikanen und Schönheiten gezeigt zu haben. $DoDL_C$ jedoch blieb dabei ein wenig auf der Strecke.

Ein sehr schöner Gedanke ist es zu erleben, wie die EU um das gewünschte Frontend erweitert wird, die Arbeit mit $DoDL_C$ innerhalb der EU geleistet werden kann, der Compiler integriert wird, und letztlich der ja schon oft zitierte Generator den gesamten, weiten und steinigen Weg vom spezifizierten Hyperdokument bis hin zu seiner HTML-Repräsentation komplettiert.

Wie heißt es akademisch so schön:

> This is beyond the scope of this paper and will be described somewhere else.

Literaturverzeichnis

[AG96] K. Arnold und J. Gosling. *Die Programmiersprache Java.* Addison Wesley, 1996.

[ASU92] A. V. Aho, R. Sethi, und J. D. Ullman. *Compilerbau*, Band 1, 2. Addison Wesley, 1992.

[Ban86] F. Bancilhon. A Logic-Programming/Object-Oriented Cocktail. *SIGMOD RECORD*, 15(3):11–21, September 1986.

[Bis98] J. M. Bishop. *Java Gently.* Addison-Wesley, 1998.

[Bra88] I. Bratko. *PROLOG.* Addison-Wesley, 1988.

[CM81] W. F. Clocksin und C. S. Mellish. *Programming in Prolog.* Springer, 1981.

[CW96] M. Campione und K. Walrath. *Das Java Tutorial.* Addison-Wesley, 1996.

[DD96] E.-E. Doberkat und S. Dissmann. *Einführung in die objektorientierte Programmierung in* BETA. Addison-Wesley, 1996.

[DD98] H. M. Deitel und P. J. Deitel. *Java: how to program.* Prentice Hall, 1998.

[Dob96a] E.-E. Doberkat. Browsing a Hyperdocument. MEMO 87, Lehrstuhl für Software-Technologie, Uni Dortmund, September 1996.

[Dob96b] E.-E. Doberkat. A Language for Specifying Hyperdocuments. *Software - Concepts and Tools*, 17:163–172, April 1996.

[DS95] C. Donelly und R. Stallman. *Bison, The YACC-compatible Parser Generator*, November 1995.

[FD97] A. Fronk und E.-E. Doberkat. Durchlaufverhalten von Hyperdokumenten. Technischer bericht, Lehrstuhl für Software-Technologie, Universität Dortmund, April 1997.

[GJ90] D. Grune und C. Jacobs. *Parsing Techniques - A Practical Guide.* Ellis Horwood, 1990.

[GJS97] J. Gosling, B. Joy, und G. Steele. *Java - Die Sprachspezifikation.* Addison Wesley, 1997.

[HS84] E. Horowitz und S. Sahni. *Fundamentals of Data Structures in Pascal.* Ptiman, 1984.

[KP84] B. W. Kernighan und R. Pike. *The UNIX Programming Environment.* Prentice Hall, 1984.

[KR90] B. W. Kernighan und D. M. Ritchie. *Programmieren in C,.* Carl Hanser Verlag, 1990. Ansi-C.

[May80] G. May. *Strukturiertes Programmieren mit ALGOL 60.* Hanser, 1980.

[MMRN93] O. L. Madsen, B. Moeller-Redersen, und K. Nygaard. *Object-Oriented Programming in the* BETA *Programming Language.* Addison-Wesley, 1993.

[Nau63] P. Naur. Revised report on the algorithmic language Algol 60. *Comm. ACM*, 6(1):1–17, 1963.

[NS93] A. Nerode und R. A. Shore. *Logic For Applications.* Springer, 1993.

[Pax95] V. Paxson. *Flex, a fast scanner generator*, Februar 1995.

[Ros89] M. Rosenstein. *Datenstrukturen - effektiv programmieren.* McGraw-Hill, 1989.

[SJ85] A.T. Schreiner und E. Janich. *Compiler bauen mit UNIX.* Carl Hanser Verlag, 1985.

[SM] Inc Sun Microsystems. The Java Technology Home Page. http://www.javasoft.com/.

[SM98] R. Stallman und R. McGrath. *GNU Make – A Program For Directing Recompilation.* Free Software Foundation, 3.77 Auflage, 1998.

[Smo92] G. Smolka. Feature-Constraint Logics for Unification Grammars. *Logic Programming*, (12):51–87, 1992.

[Sta98] R. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 2.8.1 Auflage, 1998.

[WG84] W. Waite und G. Goos. *Compiler Construction.* Springer-Verlag, 1984.

[Wir95] N. Wirth. *Algorithmen und Datenstrukturen.* Teubner, 1995.

[Zel97] A. Zeller. Unified Versioning Through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, Oktober 1997.

Wie so oft, steckte auch in dieser Arbeit der Teufel im Detail.