Entwurf und Realisierung einer Verhaltensbeschreibungssprache für technische Expertensysteme

Diplomarbeit von

Uwe Husemeyer
Matr.-Nr. 3783010

vorgelegt bei

Herrn Prof. Dr. Hans Kleine Büning
Fachbereich 17 (Mathematik / Informatik)
Universität-GH Paderborn

24. April 1995

Hiermit versichere ich, daß ich diese Arbeit selbständig angefertigt und keine anderen als die angegebenen und kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Inhaltsverzeichnis

1 Einleitung **1**

2 Kontext und Zielsetzung **3**

2.1 Wissensbasierte Systeme im physikalisch-techn. Bereich 4

2.1.1 Wissensrepräsentation 4

2.1.2 Entwicklungsansätze 5

2.2 Das art*deco*-System 5

2.2.1 Konzept und Ablauf 5

2.2.2 Aufbau 7

2.2.3 Wissensrepräsentation 9

2.3 Zielsetzung 11

2.3.1 Anforderungen an die Verhaltensbeschreibungssprache 11

2.3.2 Anforderungen an den Übersetzer 15

3 Syntax der Verhaltensbeschreibungssprache **19**

3.1 Notation der Syntax 20

3.1.1 Lexikalische Ebene 20

3.1.2 Syntaktische Ebene 21

3.2 Lexikalische Konventionen 22

3.2.1 Kommentare 22

3.2.2 Bezeichner 22

3.2.3 Schlüsselwörter 24

3.2.4 Literale Konstanten 24

3.3 Anweisungen 25

3.3.1 Typdefinitionsanweisungen 26
3.3.2 Gleichungsanweisungen 26
3.3.3 Relationsanweisungen 26
3.3.4 Selektionsanweisungen 27
3.3.5 Boolsche Anweisungen 28
3.4 Ausdrücke 28
3.4.1 Arithmetische Ausdrücke 29
3.4.2 Boolsche Ausdrücke 30
3.4.3 Sonstige Ausdrücke 31

4 Semantik der Verhaltensbeschreibungssprache 33

4.1 Notation der Semantik 33
4.1.1 Attribute und ihre Notation 34
4.1.2 Semantische Regeln und ihre Notation 34
4.2 Bezeichner 36
4.3 Anweisungen 37
4.3.1 Typdefinitionsanweisungen 38
4.3.2 Gleichungsanweisungen 38
4.3.3 Relationsanweisungen 39
4.3.4 Selektionsanweisungen 40
4.3.5 Boolsche Anweisungen 42
4.4 Ausdrücke 43
4.4.1 Arithmetische Ausdrücke 43
4.4.2 Boolsche Ausdrücke 44
4.4.3 Sonstige Ausdrücke 44

5 Implementierungsansätze 47

5.1 Komponenten eines Übersetzers 47

5.1.1 Überblick 47

5.1.2 Komponenten der Analysephase 49

5.1.3 Komponenten der Synthesephase 51

5.1.4 Phasenübergreifende Komponenten 52

5.2 Grundsätzliche Implementierungsalternativen 53

5.3 Manuelle Ausprogrammierung 54

5.3.1 Der Scanner 55

5.3.2 Der Parser 58

5.4 Automatische Generierung 59

5.4.1 Übersetzerentwicklungsumgebungen 60

5.4.2 Lex und Yacc 61

6 Implementation des Übersetzers 67

6.1 Implementation des ^{art}*deco*-Scanners 67

6.1.1 Deklarations- und Definitionsteil 67

6.1.2 Übersetzungsregeln 68

6.1.3 Hilfsfunktionen 71

6.2 Implementation des ^{art}*deco*-Parsers 74

6.2.1 Deklarationen und Definitionen 74

6.2.2 Übersetzungsregeln 77

6.2.3 Hilfsfunktionen 81

7 Schlußbemerkungen 85

A Definition der Verhaltensbeschreibungssprache 87

A.1 Grammatiksymbole 87

A.2 Syntax 88

A.3 Semantik 90

B Beispiel einer Verhaltensbeschreibung **95**

B.1 Parameter 95

B.2 Graphische Darstellung 96

B.3 Mathematische Darstellung 96

B.3.1 Modelltiefe 0 (Statische Verhaltensbeschreibung) 96

B.3.2 Modelltiefe 1 (Berücksichtigung der Masse) 96

B.4 Externe Darstellung in $^{\text{art}}$*deco* 97

B.5 Interne Darstellung in $^{\text{art}}$*deco* 99

Abbildungsverzeichnis **101**

Literaturverzeichnis **103**

Kapitel 1

Einleitung

Die vorliegende Diplomarbeit umfaßt drei Teile:

1. Den Entwurf einer Verhaltensbeschreibungssprache für technische Expertensysteme,
2. die Implementierung eines Übersetzers für diese Sprache,
3. die vorliegende schriftliche Ausarbeitung.

Die Entwicklung der Verhaltensbeschreibungssprache wurde im Hinblick auf ihren Einsatz im technischen Expertensystem $^{\text{art}}$*deco*, das in Abschnitt 2.2 ab Seite 5 beschrieben wird, vorgenommen. Als Grundlage diente die in [Ste95], [KS94] und [Sue94] benutzte, sich im Prototypenstatus befindliche Sprache. Der Umfang dieser Sprache wurde erweitert und die Sprachdefinition völlig neu gestaltet, um eine klare Gliederung und Dokumentationsfähigkeit zu erzielen.

Bei der Implementation handelt es sich um einen mit Hilfe der UNIX-Werkzeuge Lex und Yacc generierten Übersetzer. Er transformiert die externe, auf den Anwender zugeschnittene Verhaltensbeschreibung einer Komponente in eine interne, effizient verarbeitbare Darstellung. Die Definition der internen Wissensrepräsentation war vorgegeben; sie besteht aus einer Lisp-ähnlichen Präfix-Notation und wird vom Constraint-System des $^{\text{art}}$*deco*-Deduktionsmoduls interpretiert (siehe Abschnitt 2.2). Im derzeitigen Entwicklungsstand ist der Übersetzer noch ein Prototyp.

Die Plattformen für den Einsatz des Übersetzers sind das UNIX-Betriebssystem, auf dem der Prototyp von $^{\text{art}}$*deco* unter der Benutzeroberfläche „OpenWindows" läuft, und MS-DOS; hierfür existiert eine weiterentwickelte Reimplementation von $^{\text{art}}$*deco* unter der Oberfläche „MS-Windows". Bei beiden Versionen sind die Spezifikationen für die Generatoren Lex und Yacc (und damit die Definition der externen und der internen Wissensrepräsentation) identisch. Minimale Abweichungen bestehen nur in der Implementation der benötigten C-Hilfsfunktionen.

Inhalte der schriftlichen Ausarbeitung

Kapitel 2 beschreibt zunächst die Besonderheiten der Wissensverarbeitung im physikalisch-technischen Bereich als Einsatzgebiet des Übersetzers. Hierbei wird insbesondere auf das ^art^*deco*-System eingegangen, um die Hintergründe der in Abschnitt 2.3 formulierten Anforderungen an die Wissensrepräsentation und ihrer Realisierung aufzuzeigen. Mit der Umsetzung der genannten Ziele befassen sich die darauffolgenden Kapitel.

In den Kapiteln 3 und 4 wird die Verhaltensbeschreibungssprache definiert. Die Ausführungen stellen eine Sprachreferenz dar; sie sollen die vorhandenen Formulierungsmöglichkeiten der Sprache aus Anwendersicht beschreiben und die wichtigsten Designentscheidungen beim Sprachentwurf begründen. Dabei werden die Aspekte der externen Darstellung in Kapitel 3 und die Aspekte ihrer Überführung in die interne Darstellung in Kapitel 4 behandelt. Darüber hinaus erfolgt die Definition der zur Sprachbeschreibung geeigneten Formalismen.

Die Kapitel 5 und 6 befassen sich mit der Realisierung des Übersetzers. Hierzu werden in Kapitel 5 einige grundsätzliche Implementierungsalternativen und ihre Eignung im Hinblick auf diese Diplomarbeit diskutiert. Diese Diskussion umfaßt sowohl eine grobe Skizzierung der unterschiedlichen Ansätze als auch die Erläuterung der dabei erforderlichen theoretischen Grundlagen. In Kapitel 6 folgt dann die Beschreibung der wichtigsten Aspekte der vorgenommenen Übersetzerimplementation. Die Intention ist hierbei, an einigen ausgewählten Beispielen die generelle Struktur des Übersetzers zu dokumentieren sowie spätere Änderungs- bzw. Erweiterungsmöglichkeiten aufzuzeigen.

Der Anhang enthält abschließend die zusammenfassende, unkommentierte Definition der Verhaltensbeschreibungssprache und ein anwendungsorientiertes Beispiel ihres Einsatzes.

Kapitel 2

Kontext und Zielsetzung

Wissensbasierte Systeme beschäftigen sich i. d. R. mit dem Einsatz geeigneter Wissensrepräsentationen und Inferenztechniken zur Lösung schlecht strukturierbarer Probleme. Es handelt sich hierbei zwar nur um relativ eng begrenzte Problembereiche, die von Experten routinemäßig gelöst werden können, dennoch fehlen klare – also im Rechner implementierbare – Algorithmen (vgl. [Pup89]). Derzeitige Expertensysteme werden die Leistungsfähigkeit eines Menschen nicht erreichen, denn der menschliche Experte verfügt über ein höheres Maß an Assoziationsbildung, Abstraktionsvermögen, Schlußfolgerungsfähigkeit und Hintergrundwissen, als diese Qualitäten durch ein Programm nachbildbar wären.

Vergleicht man Expertensysteme mit konventionellen Programmen, so verwendet der konventionelle Ansatz feste Algorithmen zur Bearbeitung der vorliegenden Daten; bei einer Änderung der Datenstrukturen müssen also auch die Algorithmen geändert werden. Dagegen ist für wissensbasierte Systeme zur Datenbearbeitung eine Trennung zwischen Problemlösungs*strategien* und Problemlösungs*wissen* charakteristisch. Hierdurch erreichen Expertensysteme u. a. folgende Eigenschaften:

- Flexibilität
  Das Wissen kann modifiziert werden, ohne daß dazu die Implementierung der Problemlösungsstrategien bekannt sein muß.

- Kompetenz
  Die Problemlösungsstrategien lassen sich vielseitig einsetzen und kombinieren; dadurch erreichen sie eine hohe Problemlösungsfähigkeit in ihrem Anwendungsbereich.

- Transparenz
  Der Vorgang der Problemlösung kann anhand des benutzten Wissens erklärt werden.

Die weiteren Ausführungen beschränken sich auf den Einsatz wissensbasierter Systeme im physikalisch-technischen Bereich.

2.1 Wissensbasierte Systeme im physikalisch-technischen Bereich

Gerade im physikalisch-technischen Bereich können die zu lösenden Probleme aufgrund der Komplexität des Gegenstands der Betrachtung sowie aufgrund des oft anspruchsvollen physikalischen Hintergrunds hohe Schwierigkeitsgrade erreichen. Typische Einsatzbereiche für wissensbasierte Systeme sind hier die Konfiguration einer technischen Anlage (Bildung einer komplexen Struktur durch das Auswählen und Zusammensetzen bekannter Basiskomponenten), die Diagnose (Erklärung von Fehlverhalten) sowie die Simulation bzw. Funktionsprüfung (Herleitung von Folgezuständen aus einem Anfangszustand).

2.1.1 Wissensrepräsentation

Um den Anforderungen im physikalisch-technischen Bereich gerecht zu werden, ist die Entwicklung geeigneter Wissensrepräsentationen von besonderer Bedeutung. Nach [Kip88] kann die Verwendung der elementaren Wissensrepräsentationsarten wie semantische Netze, Produktionsregeln oder objektorientierte Strukturen nur die Basis der benötigten Formalismen darstellen. Das hiermit abbildbare Wissen wird „Oberflächenwissen" genannt und umfaßt die assoziative Verknüpfung von Wissensinhalten (z. B. empirische oder heuristische Aussagen), ohne daß diese Inhalte näher begründet werden. Deshalb wird vor allem im technischen Bereich versucht, ein tiefergehendes Systemverständnis, sogenannte „tiefe Modelle", explizit in der Wissensbasis zu repräsentieren und mit Assoziativverknüpfungen zu versehen. Dieses kann beispielsweise durch eine Repräsentation von grundlegenden physikalischen Gesetzmäßigkeiten geschehen.

Neben einer geeigneten internen Wissensrepräsentation zur effizienten Verarbeitung ist die externe Wissensdarstellung wichtig, denn nur über diese ist in den einsatzbereiten Endsystemen („Running Systems"[1]) die Manipulation und Erweiterung der vorhandenen Wissensbasis durch einen Experten oder Wissensingenieur möglich. Auch hier müssen geeignete Konstrukte zur Formulierung „tiefer Modelle" vorhanden sein. Notwendig ist in diesem Zusammenhang eine syntaktische Annäherung an die vorliegende Domäne, um den traditionellen Weg der Wissensakquisition zu vereinfachen.

Der traditionelle Weg der Wissensakquisition ist eine Vorgehensweise, bei der ein Wissensingenieur den Experten, dessen Domänenwissen und -erfahrungen in der Wissensbasis abgebildet werden sollen, befragt und die so erhaltenen Wissensfragmente strukturiert. Dafür muß er oft implizites Wissen explizit machen. Anschließend übersetzt der Wissensingenieur das strukturierte Wissen in einen geeigneten Repräsentationsformalismus (vgl. [Sch91]). Im Gegensatz dazu soll mit Hilfe domänengerechter externer Wissensrepräsentationen der häufig über keine Erfahrungen im Umgang mit Programmiersprachen verfügende Experte in die Lage versetzt werden, direkt mit dem Expertensystem zu kommunizieren. Er soll also sein Wissen interaktiv im System formalisieren können. Werden zusätzlich graphische Ausdrucksmöglichkeiten für

[1]Running Systems sind Software-Systeme, die den Prototypenstatus erfolgreich verlassen haben und bei realen Problemstellungen eingesetzt werden.

Wissensbestandteile geschaffen, erzielen solche Expertensysteme eine hohe Benutzerfreundlichkeit; im Idealfall kann die Arbeit des Wissensingenieurs vollständig vom Experten übernommen werden. Die Vorteile dieser Vorgehensweise sind verkürzte Entwicklungszeiten für Expertensysteme und Wissensbasen sowie bessere Pflege- und Validierungsmöglichkeiten des Wissens.

Zusammengefaßt lauten also die Ziele in bezug auf die Wissensrepräsentation im physikalisch-technischen Bereich:

- Nachbildung einer möglichst ingenieursmäßigen, am Experten orientierten Vorgehensweise bei der Wissensakquisition;
- ergänzender Einsatz von Oberflächenwissen und „tiefen Modellen“ zur Fundierung des Problemlösungsvorgangs durch umfangreiches Bereichs- und Problemlösungswissen.

2.1.2 Entwicklungsansätze

Existierende Entwicklungen, die den Ansatz der Formalisierung „tiefer Modelle“ verfolgen, werden zum Beispiel in [Lü92] (CAMeL) und in [Kip88] (COMODEL) beschrieben. Die graphische Repräsentation von Wissensstrukturen wird im Zusammenhang mit dem wissensbasierten Diagnosesystem DIWA in [Sch91] und für das Konfigurationssystem AKON in [KS94] vorgestellt.

Bei ^{art}*deco*, einem System zur Konfigurationsprüfung hydraulischer Anlagen (siehe Abschnitt 2.2), werden über [Lü92] und [Kip88] hinausgehende Möglichkeiten der Verhaltensbeschreibung technischer Komponenten benötigt, so daß die Entwicklung einer neuen externen Wissensrepräsentation erforderlich war. Die Formalisierung dieser Repräsentation und die Implementation eines entsprechenden Übersetzers sind Gegenstand der vorliegenden Diplomarbeit.

2.2 Das ^{art}*deco*-System

Das Software-System ^{art}*deco* unterstützt wissensbasiert die Konfiguration und Inbetriebnahme von hydraulischen Anlagen. Es wurde am Lehrstuhl „Wissensbasierte Systeme“ der Universität-Gesamthochschule Paderborn (Prof. Dr. H. Kleine Büning) in Zusammenarbeit mit dem Institut „Meß-, Steuer- und Regelungstechnik“ der Universität-Gesamthochschule Duisburg (Prof. Dr. H. Schwarz) entwickelt.

2.2.1 Konzept und Ablauf

Ein charakteristisches Merkmal von ^{art}*deco* ist die graphische Komponente bei der Problemformulierung. Sie ermöglicht die Manipulation der Wissensbasen durch Interaktion mit dem Experten oder dem Benutzer[2].

[2]Als Benutzer des ^{art}*deco*-Systems wird der Ingenieur bezeichnet, der eine neue hydraulische Anlage mit Hilfe vorgegebener Komponenten modellieren möchte.

Die typische Vorgehensweise zur Prüfung einer hydraulischen Anlage besteht aus dem graphischen Aufbau eines hydraulischen Schaltkreises und der anschließenden Definition der Anforderungsparameter. In der Aufbauphase werden die geeigneten Objekte aus der sogenannten Komponentenbibliothek durch Mausoperationen ausgewählt und in einem Fenster angeordnet. Auch die Verknüpfung der Komponenten geschieht mit graphischen Operationen (z. B. „Drag and Drop"). Zusätzlich können mittels Dialogboxen die bekannten Zustandsgrößen als Anforderungsparameter eingegeben werden; sie werden in der Simulationsphase ausgewertet.

Das Ziel bei der Entwicklung von ^{art}*deco* war die Maximierung des aus diesen graphischen Informationen ableitbaren Wissens. Im Gegensatz zu CAD-Programmen wird abhängig von den Benutzeraktionen und den sich hinter den graphischen Objekten verbergenden Wissensbasen ein internes funktionales Schaltkreismodell der hydraulischen Anlage aufgebaut. Dabei müssen vom System folgende Aufgaben erfüllt werden:

- Schnittstellenprüfung
  Beispiel: Bei der Verknüpfung von zwei Komponenten muß die Kompatibilität der beteiligten Anschlüsse gewährleistet sein (Art, Maße, etc.).

- Instanziierung neuer Objekte
  Beispiel: Erzeugung eines Leitungsobjekts nach dem Ziehen einer Verbindungslinie zwischen zwei Komponentenanschlüssen.

Die anschließende Konfigurationsprüfung besteht aus einer Verhaltenssimulation der Anlage. Hier kann zwischen der statischen und der dynamischen Funktionsprüfung unterschieden werden (vgl. [Sue94]). Die statische Prüfung ist dabei der dynamischen zeitlich vorgelagert, um zu Beginn des Simulationszeitintervalls die Anfangswerte aller Zustandsgrößen bereitzustellen.

Statische Funktionsprüfung

Für die statische Prüfung wird das interne Modell durch ein nichtlineares Gleichungssystem repräsentiert, das die gegenseitigen Abhängigkeiten der lokalen Zustandsgrößen der Komponenten wiedergibt. Während der Berechnung versucht das System, aufgrund der bekannten Zustandsgrößen durch Constraint-Propagierung eine global konsistente Parameterbelegung für alle Zustandsvariablen zu ermitteln. Die berechneten Werte werden für spätere Verwendungen (z. B. in der dynamischen Simulation) gespeichert und können im Schaltkreis eingeblendet werden. Treten Inkonsistenzen auf, so erfolgt eine Meldung mit den widersprüchlichen Informationen an den Benutzer. Erst nach einer Fehlerkorrektur kann die dynamische Simulation gestartet werden.

Dynamische Funktionsprüfung

Der dynamischen Konfigurationsprüfung stehen mehrere nichtlineare Differentialgleichungssysteme für dieselbe Komponente zur Verfügung. Es wird also nicht ein allgemeingültiges statisches Modell verwendet, sondern heuristisch jeweils ein geeignetes Modell ausgewählt und

berechnet. Je mehr dynamische Verhaltensbeschreibungen dem System zur Verfügung stehen (z. B. zur Berücksichtigung der Massenträgheit, des Druckaufbaus und der Reibungskräfte bei einem Zylinder), desto genauer ist die dynamische Simulation. Während der Simulation werden die einzelnen Berechnungsphasen in einer Statuszeile angezeigt und die aktuellen Werte der Zustandsgrößen graphisch dargestellt, soweit dieses sinnvoll ist (Beispiel: Position des Kolbens bei einem Zylinder). Darüber hinaus hat der Benutzer die Möglichkeit, sich für das Simulationszeitintervall die jeweiligen Werte einer Zustandsgröße als Funktionsgraph in Abhängigkeit von der Zeit anzeigen zu lassen. Dieses erfolgt in einem gesonderten Fenster.

Der Benutzer von *artdeco* kann jederzeit aus dem Simulationsmodus in den Bearbeitungsmodus wechseln und sowohl den aktuellen Aufbau des Schaltkreises als auch die Werte der Zustandsgrößen verändern und die Simulation erneut starten.

2.2.2 Aufbau

Der generelle Aufbau von *artdeco* besteht aus vier Komponenten, die im folgenden in Anlehnung an [Ste95] kurz erläutert werden. Ihre schematische Darstellung ist der Abbildung 2.1 zu entnehmen. Die Pfeile deuten dabei die Richtungen der wichtigsten Informationsflüsse an.

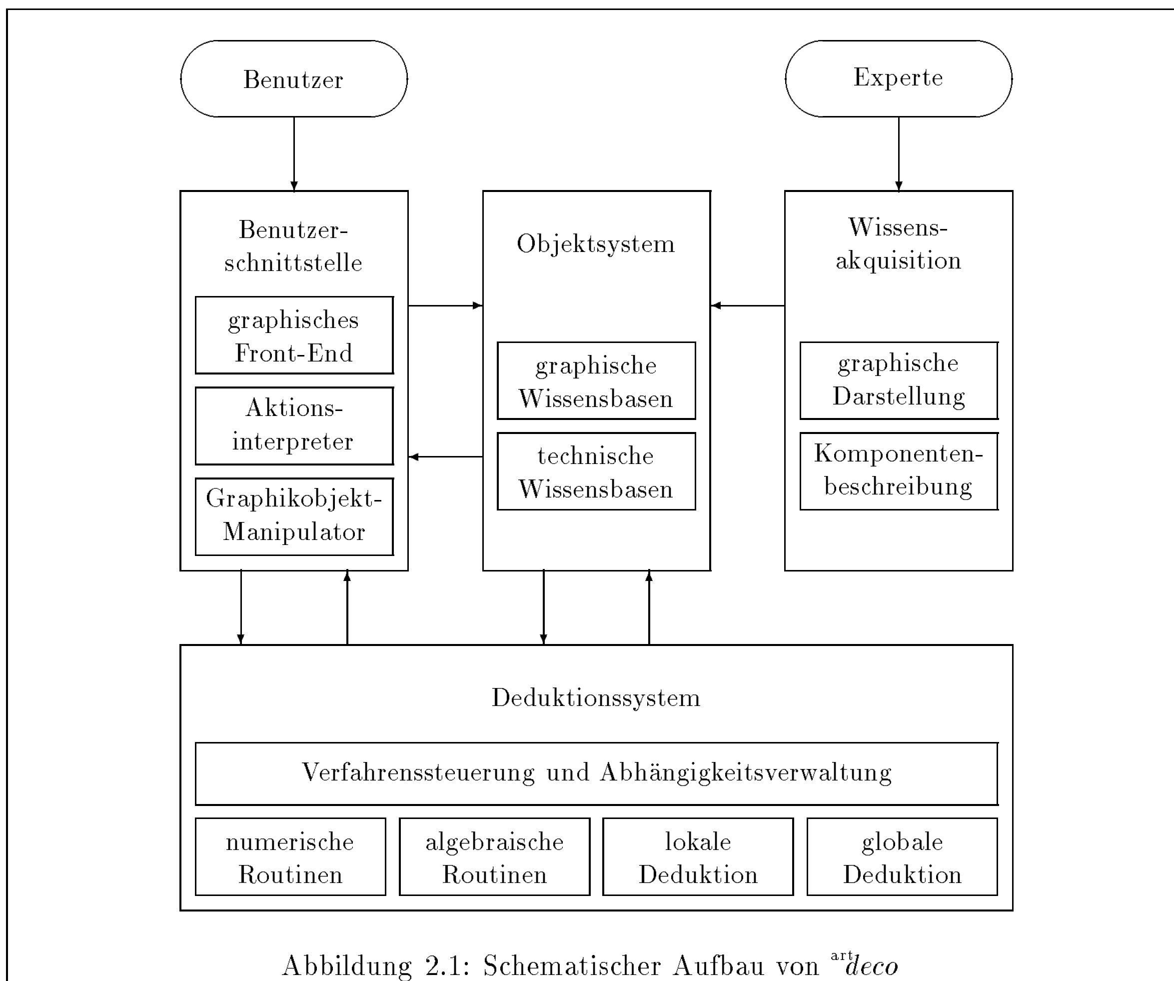


Abbildung 2.1: Schematischer Aufbau von *artdeco*

Wissensbasen

Das Wissen in den Wissensbasen läßt sich wie bei jedem Expertensystem nach seiner Herkunft in drei Bereiche aufteilen: Das problemspezifische Wissen stammt via graphischer Schnittstelle vom Benutzer, das bereichsspezifische Wissen über die Wissenserwerbskomponente vom Experten; die Zwischen- und Endergebnisse sind von der Deduktionskomponente hergeleitet worden.

Im ^{art}*deco*-System kann zusätzlich eine Unterteilung in graphische und technische Wissensbasen vorgenommen werden (siehe auch Abschnitt 2.2.3). In den graphischen Wissensbasen sind die Informationen für die visuelle Objektdarstellung und ihre Manipulationsmöglichkeiten festgelegt. Die Informationen der technischen Wissensbasen umfassen dagegen die Struktur und das Verhalten der jeweiligen Komponente.

Graphische Benutzerschnittstelle

Die graphische Schnittstelle ist für die Kommunikation des Systems mit dem Benutzer zuständig. Die Idee und die Grundzüge dieser Interaktion wurden in Abschnitt 2.2.1 skizziert. Hieraus leiten sich die drei Komponenten der Schnittstelle ab:

- Graphisches Front-End
  Diese Komponente ist als einzige oberflächenabhängig. Sie realisiert die visuellen Elemente der Kommunikation (Darstellung des Zeichenbereichs, der Graphikobjekte und Dialogboxen) und die Entgegennahme der Benutzeraktionen (Mausaktionen, Dateneingabe).

- Aktionsinterpreter
  Der Aktionsinterpreter analysiert die Benutzeraktionen und entscheidet dann kontextabhängig über ihre Zulässigkeit. Unzulässige Aktionen werden mit einer Fehlermeldung zurückgewiesen.

- Graphikobjekt-Manipulator
  Hier werden die Graphikobjekte mit dem internen Objektsystem in den Wissensbasen in Übereinstimmung gebracht, so daß zu jedem Zeitpunkt die Konsistenz zwischen der graphischen Darstellung und der internen Repräsentation gewährleistet ist.

Wissenserwerbskomponente

Mit Hilfe der Wissenserwerbskomponente manipuliert der Experte die technischen und graphischen Wissensbasen für die Objekte. Die graphischen Informationen können über Importfilter aus herkömmlichen CAD-Programmen eingelesen werden. Für die Verhaltensbeschreibung in den technischen Wissensbasen wurden die Wissensrepräsentationssprache und ihr Übersetzer, die Gegenstand dieser Diplomarbeit sind, entwickelt. Neue Komponenten können aufgrund der Struktur- und Verhaltensbeschreibungen instanziiert und in der Komponentenbibliothek angezeigt werden.

Deduktionskomponente

Die Deduktionskomponente enthält mehrere lokale und globale Deduktionsstrategien sowie numerische und algebraische Routinen. Ihr Einsatz wird durch Module zur Verfahrenssteuerung und Abhängigkeitsverwaltung kontrolliert.

2.2.3 Wissensrepräsentation

Im Rahmen der vorliegenden Diplomarbeit ist die Wissensakquisitionskomponente von besonderer Bedeutung. Aus diesem Grund wird im folgenden in Anlehnung an [Ste95] und [KS94] auf die relevanten Aspekte der internen und externen Wissensrepräsentation genauer eingegangen.

Internes Komponentenmodell

Das interne Modell einer hydraulischen Komponente in $^{\text{art}}$*deco* wird durch graphische und technische Informationen repräsentiert. Ein Teil der technischen Informationen bezieht sich auf das Verhalten und die physikalischen Eigenschaften der Komponente selbst (z. B. der hydraulische Widerstandswert eines Ventils). Ein anderer Teil beschreibt das Zusammenwirken von physikalischen Größen, die zwar in bezug auf die Komponente extern sind, sich aber beispielsweise an den Komponentenanschlüssen beobachten lassen. Hierzu wird das im folgenden erläuterte *Gate-Konzept* verwendet.

Die an einer Komponente befindlichen Gates sind zur einfacheren Identifizierung durchnumeriert und symbolisieren die Kontaktstellen dieser Komponente mit dem umliegenden Schaltkreis. Dabei müssen die Gates nicht unbedingt mit physikalischen Anschlüssen der Komponente korrespondieren (siehe das unten folgende Beispiel). Auch für die Gates sind graphische von technischen Informationen zu unterscheiden, denn sie erfüllen verschiedene Funktionen:

- Graphische Funktion
  Die Gates kennzeichnen bestimmte (u. U. maussensitive) Bereiche in der graphischen Komponentendarstellung. Für ein Gate, das mit einem physikalischen Komponentenanschluß korrespondiert, bestimmt ein solcher Bereich die graphische Verbindungsstelle der Komponente.

- Technische Funktion
  Mit Hilfe der Gates können externe physikalische Größen referenziert werden (z. B. der an Gate 2 eines Ventils anliegende Druck). Diese Möglichkeit wird in der Beschreibung des Komponentenverhaltens benötigt.

Die Abbildung 2.2 verdeutlicht den graphischen Aufbau einer Komponente in $^{\text{art}}$*deco* am Beispiel eines Gleichgangzylinders. Dabei sind die Gates durch gefüllte Kreise symbolisiert. Die Gates 3 und 4 sind hydraulische Anschlüsse. Im Gegensatz hierzu dienen die Gates 1 und 2 allein der Beschreibung von Kontaktstellen, an denen bestimmte physikalische Größen (hier u. a. Position, Geschwindigkeit und Kraft der Kolbenstange[3]) beobachtbar sind.

[3] Eine ausführliche Beschreibung der modellierbaren Zustandsgrößen eines Gleichgangzylinders ist in Anhang B zu finden.

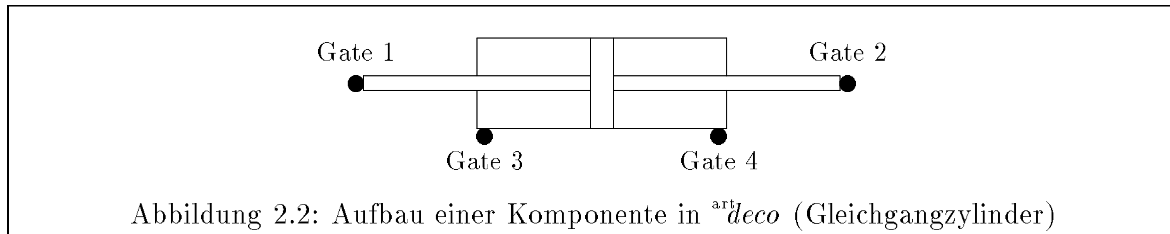


Abbildung 2.2: Aufbau einer Komponente in ^{art}*deco* (Gleichgangzylinder)

Verhaltensbeschreibung

Zur Definition einer hydraulischen Komponente durch den Experten müssen dem System die entsprechenden graphischen und physikalisch-technischen Informationen eingegeben werden[4]. Die technische Beschreibung der Komponente umfaßt die Definition der Objektstruktur mit ihren zu modellierenden physikalischen Größen und die Definition ihres statischen und dynamischen Verhaltens.

Der Experte hat die Möglichkeit, auf einem abstrakten Niveau durch die Angabe von physikalischen Zustandsgrößen und Gates zu beschreiben, welche Eigenschaften der Komponente problemadäquat modelliert werden sollen. Diese Strukturinformationen werden zunächst vom System zur Instanziierung eines entsprechenden ^{art}*deco*-Objekts verwendet. Darüber hinaus dienen sie der Fehlererkennung bei der Übersetzung der anschließend vom Experten vorgenommenen Verhaltensbeschreibung. Hier wird jede Zustandsgröße als Variable aufgefaßt, deren Existenz vorab bekannt sein muß. Auf diese Weise können eventuelle Fehler frühzeitig (d. h. schon während der Wissensakquisition, nicht erst zur Simulationszeit) entdeckt und gemeldet werden. Die ausführliche Beschreibung der Syntax für die Strukturinformationen befindet sich in Abschnitt 6.1.3.1 ab Seite 72.

Auch die Definition des Komponentenverhaltens soll vom Experten möglichst ingenieursmäßig und ohne Programmiererfahrung zu bewerkstelligen sein. Was hierunter in bezug auf das ^{art}*deco*-System zu verstehen ist, wird in Abschnitt 2.3 erläutert. Die ausführliche Darstellung der externen Wissensrepräsentation erfolgt in Kapitel 3. Der Experte kann dabei das Komponentenverhalten entsprechend unterschiedlicher Modelltiefen spezifizieren. Die Modelltiefe spiegelt das Maß der Detailgetreue bei der Simulation wider und ist durch die Menge der modellierten Einflußfaktoren auf die Komponente gekennzeichnet. Die so festgelegten Verhaltensbeschreibungen werden nach ihrer erfolgreichen Übersetzung im internen Objektsystem gespeichert. Spätere Erweiterungen können die Modelltiefe erhöhen.

Der Hauptbestandteil der Verhaltensbeschreibungen besteht aus der Formulierung von mathematisch-physikalischen Gleichungen und Relationen, die die Beziehungen zwischen den beteiligten physikalischen Größen repräsentieren. Diese Beziehungen sind ungerichtet und werden intern durch symbolische und numerische Constraints dargestellt. Die interne Wissensrepräsentation benutzt eine Lisp-ähnliche Präfix-Notation, weil sich dessen Verarbeitungsmöglichkeiten als vorteilhaft herausgestellt haben. Für den Experten und den Benutzer von ^{art}*deco* ist die interne Repräsentation und Verarbeitung des Wissens jedoch verborgen.

In Anhang B befindet sich ein ausführliches Beispiel zur Modellierung und Verhaltensbeschreibung einer hydraulischen Komponente. Hier erfolgt für einen Gleichgangzylinder neben der

[4]Im weiteren wird auf die graphischen Aspekte nicht näher eingegangen.

Angabe seiner graphischen Repräsentation die Gegenüberstellung der mathematischen Verhaltensdefinition mit ihrer externen und internen Darstellung in ${}^{art}deco$.

2.3 Zielsetzung

Der wichtigste Punkt bei der Entwicklung einer externen Wissensrepräsentation für die Wissenserwerbskomponente des ${}^{art}deco$-Systems ist die Berücksichtigung möglichst intuitiver, ingenieursmäßiger Ausdrucksmöglichkeiten, die den Experten bei der Wissensakquisition unterstützen. Diese Anforderung wird in Abschnitt 2.3.1 näher spezifiziert. Auch der Übersetzer kann dem Experten einige Routinetätigkeiten abnehmen. Damit beschäftigt sich Abschnitt 2.3.2.

Im folgenden werden sich die Ausführungen ausschließlich auf diejenigen Teile der externen Repräsentation beziehen, die das Komponentenverhalten definieren. Es handelt sich also um rein textuelle Aspekte der Wissensakquisition im ${}^{art}deco$-System.

2.3.1 Anforderungen an die Verhaltensbeschreibungssprache

Im Software-System ${}^{art}deco$ wird die Definition des Komponentenverhaltens in zwei – in bezug auf den Problemlösungsvorgang äquivalenten – Darstellungsformen gespeichert: sowohl in seiner internen als auch in seiner externen Repräsentation. Die zweifache Speicherung bedeutet zwar Redundanz, wird aber in Kauf genommen, weil semantisch nicht relevante Konstrukte wie Kommentare, textuelle Formatierungen etc. nicht in die interne Darstellung übersetzt werden. Diese Informationen sind jedoch für das Verständnis der externen Darstellung unverzichtbar und dürfen nicht verlorengehen.

Gerade die Art und Weise des Mensch-Maschine-Dialogs entscheidet häufig über die Akzeptanz eines Expertensystems. Wenn es nicht gelingt, das Wissen in einer für den Anwender intuitiven und problemadäquaten Form darzustellen, kann der Einsatz eines wissensbasierten Systems wegen mangelnder Handhabbarkeit in Frage gestellt werden. Das gleiche gilt für die Kommunikation während der Wissensakquisition: Soll der Umweg über einen Wissensingenieur (siehe Abschnitt 2.1.1) vermieden werden, muß die Wissensrepräsentation bestimmten Anforderungen genügen. Diese Anforderungen werden in den folgenden Abschnitten im Hinblick auf ${}^{art}deco$ skizziert.

2.3.1.1 Allgemeine Anforderungen

Die in diesem Abschnitt aufgeführten Anforderungen gelten grundsätzlich für jede Programmiersprache. Da auch die Verhaltensbeschreibungssprache von ${}^{art}deco$ programmiersprachliche Elemente enthält, werden sie kurz erläutert. Zusätzlich werden einige Beispiele aus der ${}^{art}deco$-Sprachentwicklung zur Verdeutlichung ausgeführt.

Angemessener Sprachumfang

Der Umfang der sprachlichen Ausdrucksmöglichkeiten hat Auswirkungen auf die weiter unten erläuterten Kriterien wie Wartbarkeit oder Erlernbarkeit. Dabei lassen sich gegenläufige Phänomene beobachten: Eine zu große Zahl verschiedener Sprachkonstrukte mag einem erfahrenen Anwender ausgefeilte Problemformulierungen eröffnen, erschwert jedoch die Erlernbarkeit der Sprache für einen Anfänger. Auf der anderen Seite kann ein zu kleiner Sprachumfang schnell beherrscht werden, verhindert aber benötigte Formulierungsmöglichkeiten oder erzwingt ineffiziente, schwer lesbare Konstruktionen.

Bei der Festlegung des *angemessenen* Sprachumfangs muß als untere Schranke die minimale Mächtigkeit der Sprache in bezug auf ihren Einsatzbereich angesehen werden. Eine maschinelle Problemlösung kann nur dann erfolgen, wenn dem Software-System alle relevanten Informationen zur Verfügung stehen. Die Mächtigkeit der Sprachdefinition muß also zumindest so groß sein, daß alle zu lösenden Problemstellungen der Einsatzdomäne des Systems formulierbar sind. Dieses bedeutet z. B. für ^{art}*deco* die Möglichkeit der Modellierung von physikalischen bzw. mathematischen Gesetzmäßigkeiten, sowie von Daten des technischen Bereichs.

Die Bestimmung der sinnvollen, oberhalb dieser Schranke angesiedelten Komplexität einer Sprache erweist sich als schwieriger. Hier spielen sowohl die Aspekte der Realisierbarkeit einer entsprechenden Übersetzerimplementation als auch der Benutzerfreundlichkeit eine Rolle. Der Experte soll die jeweiligen Wissensinhalte so schnell wie möglich formulieren bzw. interpretieren können. Wünschenswert ist also eine Repräsentation, die so intuitiv ist, daß sich der Experte ohne Schwierigkeiten, die die Formulierung des Wissens oder des Problems betreffen, mit dem System kommunizieren kann. Sobald er weiß, *was* er eingeben will, sollte ihm die Frage, *wie* er dieses formulieren muß, keine Schwierigkeiten bereiten. Hierzu sind jedoch möglichst ausdrucksstarke Konstrukte erforderlich.

Ob es mit der ^{art}*deco*-Verhaltensbeschreibungssprache gelungen ist, einen angemessenen Sprachumfang zu definieren, kann nicht abschließend beurteilt werden. Hierzu sind weitere praktische Einsätze, die über das Beispiel in Anhang B hinausgehen, notwendig.

Gute Lesbarkeit

Läßt die Sprachdefinition die Einfügung von Kommentaren zu und verzichtet sie auf textuelle Formatierungsvorschriften, so sind wesentliche Voraussetzungen für eine gute Lesbarkeit der Quelltexte geschaffen. Für den menschlichen Leser von Quelltexten sind komplexe Sprachkonstrukte oft schwer verständlich, insbesondere, wenn eine andere Person für ihre Erstellung verantwortlich war. Kommentarzeilen sind also nicht nur für den Autor des Quelltextes nützlich, sondern z. B. auch für die Mitglieder seiner Arbeitsgruppe.

Bestehen keine Formatierungsvorschriften, dann kann der Anwender seinen Quelltext mit Leerzeichen, Tabulator- und Zeilenvorschüben (sogenannte Füllzeichen) so strukturieren, daß die Lesbarkeit erhöht wird. Denkbare und häufig verwendete Strukturierungen sind die Einrückungen von Sprachkonstrukten verschiedener Schachtelungstiefen, z. B. bei der Verwendung von IF-THEN-ELSE-Kaskaden. Auch können durch Einfügung von Leerzeilen inhaltlich zusammengehörige Quelltextteile optisch von anderen Teilen getrennt werden. Bei

^{art}*deco* werden die vom Anwender eingefügten Kommentare und Füllzeichen in der externen Verhaltensbeschreibung unverändert gespeichert und nur für die Erzeugung der internen Präfix-Darstellung eliminiert.

Gute Wartbarkeit

Eine wichtige Voraussetzung für die gute Wartbarkeit einer Sprache ist die Möglichkeit zur Modularisierung des Quelltextes. Ein Modul bezeichnet einen eigenständigen Programmteil, der über bestimmte Funktionalitäten verfügt. Auf diese Funktionalitäten kann von außerhalb des Moduls nur über eine genau spezifizierte Modul-Schnittstelle zugegriffen werden. Die einzelnen Module können separat getestet werden und müssen (im Idealfall) nach festgestellter Fehlerfreiheit nicht mehr verändert werden. Eine besondere Bedeutung kommt diesem sogenannten Black-Box-Prinzip wiederum in Projekten zu, in denen mehrere Personen zusammenarbeiten. Die Verhaltensbeschreibungssprache in ^{art}*deco* besitzt jedoch bewußt nicht die Komplexität einer Hochsprache, daher bechränken sich die Möglichkeiten auf die optische Blockung von Quelltextteilen.

Denkbar wäre auch der Einsatz von Makros. Sie tragen zu einer Verbesserung der Wartbarkeit bei, weil Änderungen nur an der Stelle der Makrodefinition notwendig sind (die Makroexpansion kann vom Übersetzer vorgenommen werden). Zum gegenwärtigen Zeitpunkt ist diese Möglichkeit jedoch noch nicht implementiert.

Schnelle Erlernbarkeit

Für die Erlernbarkeit einer Sprachdefinition ist neben dem angemessenen Sprachumfang (s. o.) die Möglichkeit zur erwartungsgerechten Verwendung der Sprachelemente von Bedeutung. Aus diesem Grund sollten die Schlüsselwörter der Sprache auch für ungeübte Anwender leicht mit ihrer Semantik in Verbindung zu bringen sein. Hierzu zählt auch eine konsistente Verwendung der terminalen Symbole. In der Sprache für ^{art}*deco* werden daher z. B. Parameterlisten immer mit runden Klammern, Mengen immer mit eckigen Klammern umschlossen. Dabei ist es unerheblich, ob es sich um eine Parameterliste für eine Funktion oder für eine Relation handelt, bzw. ob die Menge symbolische oder numerische Elemente enthält. Die Verwendung gleicher Terminale in unterschiedlichen Sprachkonstrukten vermindert die Komplexität der Sprachdefinition, sofern diese Konstrukte durch andere Merkmale (z. B. durch eindeutige einleitende Schlüsselwörter) zu unterscheiden sind.

2.3.1.2 Spezielle Anforderungen in ^{art}*deco*

Der Einsatz der ^{art}*deco*-Verhaltensbeschreibungssprache im physikalisch-technischen Bereich stellt an ihre Sprachdefinition weitere Anforderungen. So ist zusätzlich zu beachten, daß der Anwender der Sprache i. d. R. kein erfahrener Programmierer, sondern ein Ingenieur ist. Daher sollte er so weit wie möglich von programmiertechnischen Details befreit sein. Im folgenden wird erläutert, wie Syntax und Semantik der Sprache dieses Ziel unterstützen können.

Mathematische / physikalische Sprachkonstrukte

In einem technischen Expertensystem wie ^{art}*deco* müssen Sprachkonstrukte zur Beschreibung von physikalischen bzw. mathematischen Gesetzmäßigkeiten zur Verfügung stehen, um das Verhalten der Anlagenkomponenten adäquat formalisieren zu können. Diese Sprachkonstrukte stellen eine Erweiterung der Syntax von normalen Programmiersprachen dar. Die Intention ist hier, daß die aus den beteiligten Bereichen (Physik, Mathematik) stammenden Konstrukte in möglichst intuitiver Weise in die Sprachsyntax überführt werden können. Das heißt, die Syntax sollte so gut wie möglich den Konventionen entsprechen, die in diesen Bereichen für die Notation gelten.

Der Idealzustand im Sinne einer guten Lesbarkeit und leichten Verständlichkeit für einen Nicht-EDV-Fachmann wäre eine Syntax, die auf dem Rechnerbildschirm die gleiche Art und Anordnung der formalen Symbole des jeweiligen Wissensgebietes ermöglicht, wie der Experte sie auf einem Blatt Papier wählen würde. Dieses scheitert jedoch daran, daß auf dem Rechner nicht alle notwendigen textuellen Sonderzeichen zur Verfügung stehen und daß die Symbole nur zeilenorientiert positioniert werden können. Daher kann z. B. die Formulierung von Relationen und Differentialgleichungen in der Verhaltensbeschreibungssprache von ^{art}*deco* nur näherungsweise die Originalschreibweise widerspiegeln.

Quantitative und qualitative Beschreibungen

Um mit der Verhaltensbeschreibungssprache „tiefe Modelle" (siehe Abschnitt 2.1.1) formalisieren zu können, müssen funktionale, kausale und zeitliche Zusammenhänge repräsentierbar sein. Neben den im vorherigen Abschnitt erläuterten rein quantitativen Beschreibungen (z. B. Differentialgleichungen) müssen also auch qualititive Beschreibungen von Systemen und Abläufen modelliert werden können.

Qualitative Begriffe werden in ^{art}*deco* zur Zeit nur zur Darstellung von Zuständen benötigt, z. B. für ein 4/3-Wege-Proportionalventil die Schaltstellungsangabe „parallel", „blockiert" oder „gekreuzt". Denkbar ist nach [Frü88] aber auch die Formalisierung von Zusammenhängen zwischen einzelnen technischen Größen und deren Änderungen durch qualitative Funktionen (z. B. proportional, monoton, steigend) und deren Ableitungen (z. B. positiv). Die ^{art}*deco*-Sprache ermöglicht dieses durch die Verwendung symbolischer Konstanten.

Beschreibung von Objektstrukturen

Obwohl der Experte bei der Wissensakquisition keine Kenntnis über die interne Repräsentation der Objektstrukturen hat, benötigt er einen Formalismus zur Beschreibung ihres Aufbaus auf abstrakter Ebene. In ^{art}*deco* erfolgt die Definition einer hydraulischen Komponente durch die Angabe ihrer Gates und der modellierten Zustandsgrößen (vgl. Abschnitt 2.2.3). Diese Strukturinformationen werden vom System in das interne Objektsystem integriert; der Experte erhält damit durch die externe Wissensrepräsentationssprache die Möglichkeit, die interne Objektrepräsentation zu manipulieren.

Der syntaktische Aufbau der Komponentendefinition wird in Abschnitt 6.1.3.1 erläutert. Ein Teil hiervon ist die Verhaltensbeschreibung für die Komponente. Ihre Syntax und Semantik ist auf die Beschreibung der Objektstruktur ausgerichtet, d. h. die Sprache kann die hier festgelegten Zustandsgrößen sowohl akzeptieren, als auch ihre interne Darstellung erzeugen (siehe Abschnitt 2.3.2, Absatz „Bezeichneridentifikation“).

2.3.2 Anforderungen an den Übersetzer

Das Constraint-System in $^{\text{art}}$*deco* erwartet als Eingabe eine Lisp-ähnliche Sprache in Präfix-Notation (vgl. Abschnitt 2.2). Der Umgang mit einer Präfix-Notation ist jedoch einem Benutzer des Expertensystems nicht zuzumuten, denn in der Mathematik bedient man sich üblicherweise der Infix-Notation. Aus diesem Grund wird der Experte in die Lage versetzt, Verhaltensbeschreibungen in der gewohnten Infix-Notation zu formulieren; der in dieser Diplomarbeit implementierte Übersetzer transformiert sie nach Beendigung der Eingabe in die entsprechende interne Darstellung. Hierbei muß die Transformierung aller semantisch relevanten Informationen sichergestellt werden, damit das vom Experten formulierte Wissen korrekt abgebildet wird. Der Übersetzer darf also die Semantik nicht verändern.

Weitere Anforderungen an den Übersetzer ergeben sich aus der Zielsetzung, dem Experten die Wissensformulierung zu erleichtern. So können während des Transformationsprozesses Aufgaben übernommen werden, die den Abstraktionsgrad der Verhaltensbeschreibung erhöhen, oder die den Experten von Routinearbeiten entlasten. Dazu gehört auch eine Rückmeldung über evtl. vorhandene Fehler in der Komponentenbeschreibung. Die folgenden Abschnitte werden diese Anforderungen näher erläutern.

Algebraische Umformungen

Damit der Experte mathematische Konstrukte wie gewohnt formulieren kann, sollte der Übersetzer in der Lage sein, elementare Konventionen der Mathematik zu beherrschen (z. B. die sogenannte „Punkt-vor-Strich“-Regel). Auf diese Weise werden die Formulierungen übersichtlicher.

Eine weitere Anforderung an den $^{\text{art}}$*deco*-Übersetzer ist die Fähigkeit, bei Bedarf alle algebraischen Ausdrücke in seiner Ausgabe auf die drei Operatoren für die Addition, Multiplikation und Potenzierung zu reduzieren. Hierzu sind nur rudimentäre algebraische Umformungen notwendig, die aber die Effizienz des Constraint-Systems erhöhen können[5]. Wie in Abschnitt 4.4.1 gezeigt wird, können die notwendigen Umformungen sogar auf rein texueller Ebene vorgenommen werden.

Bezeichneridentifikation

Grundsätzlich ist bei der Formulierung des Wissens bzw. des Problems eine weitgehende Abstraktion von programmiertechnischen oder implementationsspezifischen Details anzustreben.

[5] Die Verarbeitungsalgorithmen des Constraint-Systems sind effizienter, wenn eine Beschränkung der algebraischen Operatoren auf Addition, Multiplikation und Potenzierung vereinbart ist.

Der mit dem System kommunizierende Experte sollte seine Eingabe also *problem*orientiert und nicht *implementations*orientiert formulieren können. Das hat den Vorteil, daß er seine physikalisch-technische Welt gedanklich nicht verlassen muß, um die betreffenden Wissensinhalte in die Syntax der Verhaltensbeschreibungsprache zu übertragen.

Neben der reinen Verhaltensbeschreibung hat der Übersetzer Zugriff auf die Definition der Struktur einer Komponente. Diese Informationen lassen sich ausnutzen, um den Abstraktionsgrad bei der Problemformulierung zu erhöhen. Ziel ist das vollständige Verbergen von internen Bezeichnern vor dem Experten. Im folgenden sei hierzu ein Beispiel ausgeführt.

Der Anwender von art*deco* möchte das Verhalten einer hydraulischen Komponente definieren. Er kennt ihre schematische Darstellung mit den Anschlüssen (Gates) und die physikalischen Gesetze, die ihr Verhalten modellhaft wiedergeben. Die beobachtbaren Zustandsgrößen an den Komponentenanschlüssen hat er mit Bezeichnungen versehen, wie sie in den Formeln der mathematischen Gleichungen zur Verhaltensbeschreibung auftreten. Da diese logischen Bezeichner dem Übersetzer aus der Komponentenstrukturbeschreibung bekannt sind (es handelt sich um die sogenannten Alias-Namen, siehe Abschnitt 6.1.3.1) kann er sie in ihre interne Darstellung überführen.

Ist zum Beispiel p_2 bei einem Ventil die Bezeichnung für den Druck an Gate 2, so sollte eine ähnliche Zeichenkette (z. B. `p2`) in den Formeln zur externen Verhaltensbeschreibung verwendbar sein. Der Übersetzer ist dann für die Ermittlung der korrespondierenden internen Bezeichnung `GATE2@pressure` verantwortlich.

Im Anhang B erfolgt am Beispiel eines Gleichgangzylinders die Gegenüberstellung der mathematischen Darstellung mit der äquivalenten Verhaltensbeschreibung in art*deco*. Zumindest in bezug auf die Bezeichnerdarstellung kann man fast von einer 1:1-Abbildung sprechen.

Fehlerbehandlung

Die Fehlerbehandlung ist ein wichtiges Element in der Kommunikation des Anwenders mit dem Software-System. Erfahrungsgemäß gibt es nur selten nicht-triviale Eingabesequenzen des Anwenders, die bereits bei der ersten Formulierung keine Fehler enthalten. Fehler sollten so früh wie möglich vom System erkannt und dem Anwender zur Korrektur mitgeteilt werden. Das heißt, die Benutzereingabe sollte sofort nach ihrer Beendigung auf fehlerhafte Elemente hin überprüft und im Fehlerfall gar nicht erst gespeichert werden.

Eine Fortsetzung des Übersetzungsvorgangs nach einer Fehlererkennung kann sinnvoll sein, um gegebenenfalls eine Fehlerliste für den Anwender zu erstellen. Dieses Vorgehen vereinfacht und beschleunigt u. U. die Fehlerkorrektur, wenn vom Anwender mehrere Fehler auf einmal beseitigt werden können. Ein Verfahren hierfür ist die Protokollierung des Übersetzungssvorgangs mit Kennzeichnung der fehlerhaften Formulierungen. Der Benutzer kann dann das Protokoll einsehen und die fehlerhaften Stellen gezielt korrigieren.

Nachteilig wirkt sich allerdings das Problem der Folgefehler aus. Folgefehler sind keine Fehler des Anwenders, sondern haben ihre Ursache in einer unzureichenden Fehlerreparatur durch den Übersetzer. Solche Fehlerreparaturen sind sehr aufwendig, da die Intention des Anwenders vom Übersetzer ermittelt werden muß. Häufig ist fraglich, ob diesem Aufwand ein

entsprechender Nutzen gegenübersteht. Die derzeitige Implementation des ^{art}*deco*-Übersetzers bricht daher den Übersetzungsvorgang schon nach der ersten Fehlererkennung unter Angabe der entsprechenden Quelltextzeile ab.

Zur Übersetzungszeit lassen sich jedoch nicht alle Fehler erkennen: Ein typisches Beispiel ist die Division durch Null. Bei diesen in den Bereich der sogenannten dynamischen Semantik fallenden Fehlern handelt es sich um Laufzeitfehler. Hier sind die Verarbeitungsalgorithmen und nicht der Übersetzer für eine sinnvolle Fehlererkennung und -behandlung zuständig.

Kapitel 3

Syntax der Verhaltensbeschreibungssprache

Die im Rahmen dieser Diplomarbeit entworfene Sprachdefinition verfügt über einige programmiersprachliche Konstrukte, ist aber wegen ihres wesentlich geringeren Umfangs mit allgemeinen Programmiersprachen wie C oder PASCAL nur schwer zu vergleichen. Auch ihre Charaktere weichen voneinander ab: Die genannten Programmiersprachen sind imperativ, d. h. sie beschreiben Zustandsänderungen und Ablaufstrukturen, während die ^{art}*deco*-Sprache eher zur Formalisierung von Relationen, also ungerichteten Beziehungen, eingesetzt wird. Ein weiterer Unterschied ist, daß die imperativen Programmiersprachen bis auf wenige Ausnahmen anweisungsorientiert sind, die ^{art}*deco*-Sprache hingegen ausdrucksorientiert[1] ist. Im weiteren Sinne dient jedoch auch die ^{art}*deco*-Verhaltensbeschreibungssprache im Rahmen der Wissensakquisition der *Programmierung* des Rechners. Daher sind die Formalismen zu ihrer Definition sowie die Vorgehensweisen zu ihrer Realisierung nicht grundlegend verschieden von denen bei der Entwicklung allgemeiner Programmiersprachen.

Die klare Trennung der Beschreibungen für die Sprachsyntax (dieses Kapitel) und ihre Semantik (Kapitel 4) wurde aus Gründen einer besseren Gliederung der Ausarbeitung vorgenommen. Zum besseren Verständnis der Intention der jeweiligen Konstrukte wird in diesem Kapitel zu ihrer Syntaxbeschreibung eine kurze Angabe der Semantik und evtl. ein Hinweis auf ihre ausführliche Erläuterung hinzugefügt.

Die Beschreibung der Sprachsyntax erfolgt in diesem Kapitel aus der Sicht des Experten und dient daher der anwendungsorientierten Definition der Sprachelemente. Die hierbei vorgenommene Unterteilung in die lexikalischen und die syntaktischen Aspekte wird zum Zweck der Vereinbarung einer sinnvollen Notation verwendet. In Abschnitt 5.1 wird sich diese Differenzierung in den Übersetzungsphasen „lexikalische Analyse“ und „syntaktische Analyse“ widerspiegeln.

---

[1] In einer ausdrucksorientierten Sprache liefern auch Anweisungen einen Wert, d. h. sie können wie Ausdrücke verwendet werden.

3.1 Notation der Syntax

3.1.1 Lexikalische Ebene

Der Quelltext ist eine Folge von Zeichen eines Zeichensatzes (bei artdeco wie allgemein üblich des ASCII-Zeichensatzes). Teilfolgen des Quelltextes stellen sogenannte Grundbezeichner wie z. B. Wortsymbole oder Literale dar. Diese Grundbezeichner werden durch Füllzeichen voneinander getrennt. Zur Festlegung der Menge der in einer Programmiersprache zulässigen Zeichenfolgen sind reguläre Ausdrücke üblich.

In Anlehnung an [HU90] gilt für die Bildung regulärer Ausdrücke und die damit repräsentierten Zeichenfolgen:

> Sei Σ ein Zeichensatz, d. h. eine endliche Folge von Zeichen. Die regulären Ausdrücke über Σ und die von ihnen beschriebenen Mengen werden wie folgt rekursiv definiert:
>
> 1. $\emptyset$ ist ein regulärer Ausdruck und bezeichnet die leere Menge.
> 2. Die leere Zeichenfolge ε ist ein regulärer Ausdruck und bezeichnet die Menge $\{\varepsilon\}$.
> 3. Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck und bezeichnet die Menge $\{a\}$.
> 4. Wenn a und b reguläre Ausdrücke für die Mengen A und B sind, so sind (ab), $(a|b)$ und (a^*) reguläre Ausdrücke und bezeichnen die Mengen $\{\alpha\beta \mid \alpha \in A, \beta \in B\}$, $A \cup B$ und $\{\varepsilon, a, aa, aaa, ...\}$.
> (ab) heißt die Konkatenation von a und b und $(a|b)$ die Alternative von a oder b.

Zur Vereinfachung der Schreibweise werden weitere Vereinbarungen getroffen:

> Sei a ein regulärer Ausdruck. Dann gelte:
>
> - $(a^+) := (aa^*)$ bezeichnet die Menge $\{a, aa, aaa, ...\}$
> - $(a^?) := (\varepsilon|a)$ bezeichnet die Menge $\{\varepsilon, a\}$
> - (a) bezeichnet die Menge $\{a\}$.
>
> Weiter gelte für $x, y \in \Sigma$:
>
> - $(x \mid \cdots \mid y)$ sowie $x \mid \cdots \mid y$ bezeichnen die Menge $\{x, \ldots, y\}$, wobei x die untere Grenze und y die obere Grenze innerhalb des Zeichensatzes angibt.

Der vorletzte Punkt zielt auf eine Reduzierung der Klammerverwendung hin. Hierzu ist jedoch noch eine Festlegung der Operator-Prioritäten notwendig: Die höchste Priorität sollen die Operatoren *, + und ? haben, gefolgt von der Konkatenation und die gefolgt von der Alternative. Alle Konstrukte seien links-assoziativ.

3.1.2 Syntaktische Ebene

Während sich die lexikalische Ebene auf die Beschreibung des Aufbaus einzelner Zeichenfolgen im Quelltext bezieht, läßt sich mit einer Notation für die syntaktische Ebene die Struktur des Programms wiedergeben. Viele Konstrukte sowohl in allgemeinen Programmiersprachen als auch in der Verhaltensbeschreibungssprache von [art]*deco* haben eine inhärent rekursive Struktur, die sich mit Hilfe kontextfreier Grammatiken definieren läßt.

Die folgende Definition kontextfreier Grammatiken lehnt sich an [ASU88a] an:

> Zu einer kontextfreien Grammatik $G = (T, N, P, S)$ gehören vier Komponenten:
>
> 1. Eine Menge T von Terminalen.
> 2. Eine Menge N von Nichtterminalen.
> 3. Eine Menge P von Produktionen der Gestalt $A \rightarrow x$, wobei $A \in N$ ein Nichtterminal ist und $x \in N \cup T$ ein Grammatiksymbol aus der Vereinigungsmenge von Terminalen und Nichtterminalen.
> 4. Ein ausgezeichnetes Nichtterminal S als Startsymbol.

Dabei bedeuten die Begriffe im einzelnen (in Anlehnung an [ASU88a] und [Kas90]):

- *Terminale* sind die Grundsymbole, aus denen der Quelltext besteht.
- *Nichtterminale* sind syntaktische Variablen, die Mengen von Zeichenfolgen bezeichnen. Sie vereinfachen die Definition der durch die Grammatik erzeugten Sprache und geben der Sprache eine hierarchische Struktur.
- Die *Produktionen* bestimmen, wie die Terminale und Nichtterminale zur Bildung von korrekten Zeichenfolgen miteinander kombiniert werden können. Das Grundprinzip zur Bildung korrekter Zeichenfolgen ist dabei die Ableitung: Sei $w = uAv$ eine beliebige Folge von Zeichen aus der Vereinigungsmenge von Terminalen und Nichtterminalen und $p = A \rightarrow x$ eine Produktion, dann wendet man p in w an, indem das Nichtterminal A durch x ersetzt wird.
- Das *Startsymbol* ist ein ausgezeichnetes Nichtterminal. Die durch das Startsymbol bezeichnete Menge von Terminalfolgen ist die von der kontextfreien Grammatik definierte Sprache, also die Menge aller syntaktisch korrekten Programme.

Auch hier sei eine Vereinfachung der Schreibweise vereinbart: Rechte Seiten von Produktionen, die das gleiche Nichtterminal als linke Seite haben, werden zusammengefaßt, wobei die einzelnen Alternativen der rechten Seiten durch | getrennt werden. Um eine klare Trennung zwischen Terminalen und Nichtterminalen auch optisch zu erreichen, werden die Nichtterminale kursiv und die Terminale in Schreibmaschinenschriftart dargestellt.

3.2 Lexikalische Konventionen

In den folgenden Abschnitten werden die Grundsymbole definiert, aus denen die Verhaltensbeschreibungssprache für ${}^{\text{art}}$*deco* aufgebaut ist. Da hier nicht die Struktur des Quelltextes beachtet werden muß, erfolgt die Definition durch reguläre Ausdrücke und den in Abschnitt 3.1.1 auf Seite 20 vereinbarten abkürzenden Schreibweisen. Zur besseren Lesbarkeit wird in den regulären Ausdrücken das Leerzeichen, das ein Element des ASCII-Zeichensatzes ist, durch das Symbol ␣ dargestellt.

3.2.1 Kommentare

Sobald in einer Quelltextzeile das Kommentarzeichen `%` auftritt, werden alle folgenden Zeichen bis zum Zeilenende als Kommentar interpretiert und zur weiteren Analyse der Verhaltensbeschreibung eliminiert. Ein wiederholtes Auftreten des Zeichens `%` innerhalb eines Kommentars hat keine besondere Bedeutung.

In Abbildung 3.1 sind die zwei Möglichkeiten des Kommentareinsatzes – der ganzzeilige Kommentar und der Kommentaranhang – beispielhaft aufgeführt.

```
...
% (1) ganzzeiliger Kommentar
x = 0; % (2) Kommentaranhang
...
```

Abbildung 3.1: Beispiele des Kommentareinsatzes

3.2.2 Bezeichner

Einfache Bezeichner

Ein einfacher Bezeichner (*simple-id*) ist eine beliebig lange Folge von Buchstaben, Ziffern und bestimmten Sonderzeichen. Um während der lexikalischen Analyse eine effiziente Arbeitsweise zu ermöglichen (Vermeidung von Backtracking), sind als erstes Zeichen der Punkt oder eine Ziffer nicht zugelassen. Aus diesem Grund erfolgt in der Definition von *simple-id* eine Unterteilung in *first-char* (erstes Zeichen des Bezeichners) und *following-char* (Zeichen ab zweiter Stelle).

Weiter gilt:

- Groß- und Kleinbuchstaben werden unterschieden,
- alle Zeichen sind signifikant,
- Schlüsselwörter (siehe Abschnitt 3.2.3) sind nicht als Bezeichner erlaubt.

Abbildung 3.2 enthält die Definition des einfachen Bezeichners *simple-id*. Das Nichtterminal *digit* ist in Abbildung 3.6 definiert und symbolisiert eine Ziffer; ε ist die leere Zeichenfolge (vgl. Abschnitt 3.1.1).

simple-id	⟶	*first-char following-char*
first-char	⟶	`!` | `#` | `$` | `&` | `/` | `?` | `@` | `A` | ··· | `Z` | `\` | `_` | `a` | ··· | `z` | `{` | `|` | `}` | `~`
following-char	⟶	*first-char following-char*
	|	*digit following-char*
	|	`.` *following-char*
	|	ε

Abbildung 3.2: Definition des einfachen Bezeichners

Komplexe Bezeichner

Die hier vorgestellte Definition des einfachen Bezeichners sollte für den typischen Einsatzbereich der Wissensrepräsentationssprache ausreichend sein. „Typisch“ bedeutet hierbei, daß der Experte sein technisches Wissen bzw. Problem vollständig ohne Kenntnis der internen Formalismen des Systems ausdrücken kann (problemorientierte statt implementationsorientierte Formulierung, vgl. Abschnitt 2.3.2). Trotzdem sollte die Referenzierung physikalischer Zustandsgrößen auf einem niedrigeren Abstraktionsniveau (sogenannte Low-Level-Zugriffe) in bestimmten Fällen möglich sein.

Dieses hat zwei Gründe:

1. Es können a priori nicht alle zukünftigen Einsatzmöglichkeiten des $^{\text{art}}$*deco*-Systems und die damit verbundenen Formulierungsprobleme vorausgesehen werden. Unter Umständen ist die externe Wissensrepräsentation schon für leicht veränderte Aufgabenstellungen unzureichend. Es ist denkbar, daß in diesen Fällen Konstrukte auf niedrigerem Niveau einen Einsatz des Expertensystems ohne Änderungen an der Verhaltensbeschreibungssprache, ihres Übersetzers oder den Inferenzmechanismen ermöglichen.

2. Zeitweise sind in der Testphase der Expertensystementwicklung noch nicht alle benötigten semantischen Informationen zur Umsetzung der abstrahierten, problemorientierten Formulierungen verfügbar. Die Testphase wird hier durch Low-Level-Zugriffe vereinfacht und beschleunigt.

Durch die syntaktische Variable *identifier* werden komplexer aufgebaute Bezeichner für solche Low-Level-Zugriffe definiert. Mit einem *identifier* können gezielt die Zustandsgrößen von Komponenten und ihrer Gates referenziert werden (näheres zur Semantik in Abschnitt 4.1, Seite 37).

identifier	⟶	*simple-id* `[` *simple-id* `]`
	|	*simple-id* `[` *number* `]`
	|	*simple-id*

Abbildung 3.3: Definition des komplexen Bezeichners

Die Syntax für *identifier* zeigt Abbildung 3.3. Der Referenzierungsparameter ist dabei in eckige Klammern (`[` und `]`) eingeschlossen. Die syntaktische Variable *number* stellt eine numerische Konstante dar und wird in Abschnitt 3.2.4 definiert.

3.2.3 Schlüsselwörter

Die in Abbildung 3.4 aufgelisteten Zeichenketten sind als Schlüsselwörter reserviert. Sie dürfen daher nicht als Bezeichner verwendet werden. Zulässig ist nur ihre Schreibweise in Großbuchstaben.

```
BEGIN      CASE     DEFAULT   ELSE     END      IF
RELATION   THEN     TYPE
```

Abbildung 3.4: Liste der Schlüsselwörter

Die Liste der ebenfalls nicht als Bezeichner erlaubten Operatoren und Interpunktionszeichen befindet sich in Abbildung 3.5.

```
AND    IN     NOT    OR     "      %
'      (      )      *      +      ,
-      .      ..     /      :      ;
<      <=     <>     =      =?     >
>=     [      ]      ^
```

Abbildung 3.5: Liste der Operatoren und Interpunktionszeichen

3.2.4 Literale Konstanten

In der Verhaltensbeschreibungssprache von $^{\text{art}}$*deco* werden drei verschiedene Konstantentypen unterschieden:

1. numerische Konstante (*number*),
2. symbolische Konstante (*symbol*),
3. String-Konstante (*string*).

Der Aufbau der numerischen Konstanten ist im wesentlichen vergleichbar mit der Definition für Gleitkommazahlen in allgemeinen Programmiersprachen. Der optionale Exponent wird mit dem Zeichen `E` eingeleitet. Für die symbolischen Konstanten sind alle druckbaren ASCII-Zeichen mit Ausnahme von ' zugelassen. Das Begrenzerzeichen ' umschließt das Symbol, gehört aber nicht dazu. Die Konventionen für String-Konstanten sind analog zu denen für symbolische Konstanten, jedoch ist hier das Begrenzerzeichen "; es ist daher nicht als Teil des Strings erlaubt. Die Länge der Zeichenketten für numerische Konstanten, Symbole und Strings ist nur durch die Pragmatik der Implementation begrenzt.

Die Definitionen für die drei literalen Konstanten sind in Abbildung 3.6 zu finden.

$$
\begin{array}{rcl}
\mathit{number} & \longrightarrow & \mathit{digit}^{+}\ (\ \texttt{.}\ \mathit{digit}^{+}\ )^{?}\ (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-}\ )^{?}\ \mathit{digit}^{+}\ )^{?} \\
\mathit{digit} & \longrightarrow & \texttt{0} \mid \cdots \mid \texttt{9} \\
\mathit{symbol} & \longrightarrow & \texttt{'}\ (\ \texttt{␣} \mid \cdots \mid \texttt{\&} \mid \texttt{(} \mid \cdots \mid \texttt{\~{}}\ )^{*}\ \texttt{'} \\
\mathit{string} & \longrightarrow & \texttt{"}\ (\ \texttt{␣} \mid \texttt{!} \mid \texttt{\#} \mid \cdots \mid \texttt{\~{}}\ )^{*}\ \texttt{"}
\end{array}
$$

Abbildung 3.6: Definition der literalen Konstanten

3.3 Anweisungen

Jede zulässige Verhaltensbeschreibung im ${}^{\text{art}}$*deco*-System besteht aus einer Anweisung (*statement*). Diese Anweisung ist aus einer von drei Kategorien:

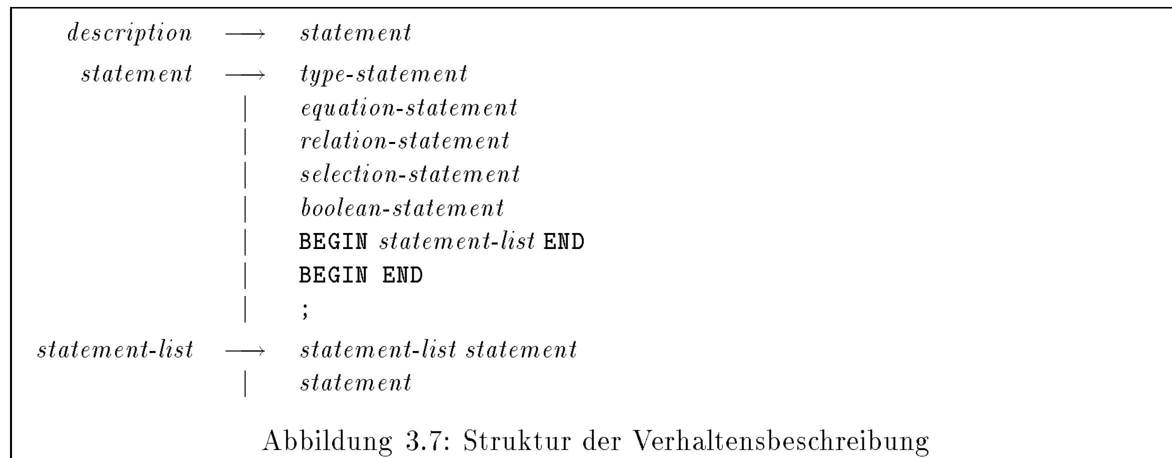
- Einzelne Anweisung
  Eine einzelne Anweisung wird realisiert durch *type-statement*, *equation-statement*, *relation-statement*, *selection-statement* oder *boolean-statement*.

- Verbundanweisung
  Mit Hilfe von Verbundanweisungen können *mehrere* Anweisungen an Stellen plaziert werden, an denen nur *eine* Anweisung erwartet wird. Sie werden daher durch die Schlüsselwörter `BEGIN` und `END` geklammert. Eine Klammerung durch kürzere Zeichen wie z. B. `{` und `}` in Anlehnung an die Programmiersprache C schien hingegen im Hinblick auf ungeübte Anwender weniger sinnvoll zu sein, da die Lesbarkeit des Quelltextes durch eine zu häufige Verwendung von Sonderzeichen leidet.

- Leere Anweisung
  Leere Anweisungen realisieren das andere Extrem: Sie gewährleisten die syntaktische Korrektheit an Stellen, wo eine Anweisung gefordert wird, haben aber keine semantische Bedeutung. Eine leere Anweisung wird durch eine leere Klammerung `BEGIN END` oder das einzelne Terminatorzeichen `;` (s. u.) symbolisiert.

Grundsätzlich gibt es in allgemeinen Programmiersprachen zwei Möglichkeiten, Anweisungen voneinander abzugrenzen: Jeweils zwei Anweisungen werden durch ein *Separator*zeichen voneinander getrennt (z. B. in Pascal mit dem Semikolon) oder jede Anweisung wird mit einem *Terminator*zeichen abgeschlossen (z. B. in C, auch mit dem Semikolon). Für die Verhaltensbeschreibungssprache in ${}^{\text{art}}$*deco* ist die zweite Möglichkeit vorgesehen, weil die Ansicht vertreten wird, daß die generelle Anfügung eines Terminatorzeichens hinter jede Anweisung für den Benutzer eher zur Routine wird und auf diese Weise eine potentielle Fehlerquelle schneller an Bedeutung verliert. Um programmiererfahrene Benutzer nicht zu verwirren, wird auch in dieser Sprache das Semikolon (`;`) als Terminatorzeichen verwendet.

In den Definitionen der Abbildung 3.7 wird die grundlegende Syntax für eine korrekte Verhaltensbeschreibung in der Wissensrepräsentationssprache festgelegt. Das Startsymbol für die kontextfreie Grammatik ist *description*. Die rekursive Definition von *statement-list* verdeutlicht die theoretisch unbegrenzte Verkettung von einzelnen Anweisungen (*statement*) in einer Vebundanweisung.



description	⟶	*statement*
statement	⟶	*type-statement*
	|	*equation-statement*
	|	*relation-statement*
	|	*selection-statement*
	|	*boolean-statement*
	|	`BEGIN` *statement-list* `END`
	|	`BEGIN END`
	|	;
statement-list	⟶	*statement-list statement*
	|	*statement*

Abbildung 3.7: Struktur der Verhaltensbeschreibung

3.3.1 Typdefinitionsanweisungen

Typdefinitionsanweisungen (*type-statement*) vereinbaren einen Namen für die mit dem Quelltext definierte Verhaltensbeschreibung. Syntaktisch betrachtet bestehen sie aus dem Schlüsselwort `TYPE` und einem einfachen Bezeichner.

Die Definition der Typdefinitionsanweisung ist in Abbildung 3.8 angegeben.

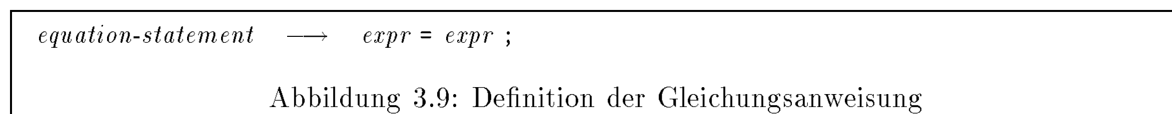
type-statement ⟶ `TYPE` *simple-id* ;

Abbildung 3.8: Definition der Typdefinitionsanweisung

3.3.2 Gleichungsanweisungen

Die Anweisungsart *equation-statement* dient zur Formalisierung mathematischer oder symbolischer Gleichungen. Auch ihr Aufbau ist syntaktisch sehr einfach: Jede Gleichung hat eine linke und eine rechte Seite, die mit dem Gleichheitszeichen verbunden werden. Die Seiten bestehen aus Ausdrücken (*expr*), die in Abschnitt 3.4 beschrieben werden.

Die formale Definition von *equation-statement* erfolgt in Abbildung 3.9.



equation-statement ⟶ *expr* `=` *expr* ;

Abbildung 3.9: Definition der Gleichungsanweisung

3.3.3 Relationsanweisungen

Relationsanweisungen (*relation-statement*) ermöglichen die Formulierung mathematischer oder symbolischer Relationen, die vom Constraintsystem in $^{\text{art}}$*deco* verarbeitet werden. Eine Relation wird durch das Schlüsselwort `RELATION`, einem einfachen Bezeichner (*simple-id*) als Name und einer in runden Klammern eingeschlossenen Folge von Bezeichnern eingeleitet. Zur

Klammerung der optionalen Relationstupel *relation-list* (beliebig lange Folge von Relationszeilen) dienen die Schlüsselwörter `BEGIN` und `END`. Ein Relationstupel (*relation*) besteht aus einer eventuell leeren Folge von Ausdrücken (vgl. Abschnitt 3.4). Um die Analogie zur Anweisungsfolge zu wahren, wird jedes Relationstupel durch ein Semikolon als Terminatorzeichen abgeschlossen, obwohl die schließende Klammer bereits als Terminatorzeichen interpretiert werden könnte.

Die Syntax für Relationen ist den Produktionen der Abbildung 3.10 zu entnehmen.

relation-statement	⟶	`RELATION` *simple-id* (*identifier-list*) `BEGIN` *relation-list* `END`
relation-list	⟶	*relation-list relation*
	|	*relation*
	|	ε
relation	⟶	(*expr-list*) ;
expr-list	⟶	*expr-list* , *expr*
	|	*expr*
	|	ε
identifier-list	⟶	*identifier-list identifier*
	|	*identifier*
	|	ε

Abbildung 3.10: Definition der Relationsanweisung

3.3.4 Selektionsanweisungen

Die Syntax des Anweisungstyps *selection-statement* lehnt sich an vergleichbare Konstrukte in allgemeinen Programmiersprachen an. Mit Hilfe einer Selektionsanweisung kann abhängig vom Wert des Bedingungsausdrucks zu bestimmten Anweisungen verzweigt werden. Grundsätzlich lassen sich in der $^{\text{art}}$*deco*-Wissensrepräsentationssprache zwei Arten von Selektionsanweisungen unterscheiden. Zum besseren Verständnis der Syntax wird ihre Semantik im folgenden kurz angedeutet (eine ausführliche Beschreibung der Semantik von Selektionsanweisungen befindet sich in Abschnitt 4.3.4 auf Seite 40):

1. IF-THEN-ELSE-Anweisungen:
   Abhängig von Wahrheitswert des Bedingungsausdrucks wird zu der Anweisung nach dem Schlüsselwort `THEN` (Bedingung ist wahr) oder zu der optionalen ELSE-Anweisung (Bedingung ist falsch) verzweigt.

2. CASE-Anweisungen:
   Hier können mehr als zwei Anweisungsalternativen angegeben werden. Die Anweisungen sind in einer eventuell leeren, durch die Schlüsselwörter `BEGIN` und `END` geklammerten Folge definiert. Der Bedingungsausdruck setzt sich aus zwei Teilen zusammen: Der erste Teil ist immer identisch und wird daher nur einmal hinter dem einleitenden Schlüsselwort `CASE` angegeben. Jeder Anweisung in der Anweisungsfolge ist ein Operator sowie der zweite Teil des Bedingungsausdrucks vorangestellt. Ist der kombinierte Bedingungsausdruck, der sich aus den mit dem Operator verbundenen beiden Teilen

zusammensetzt, wahr, wird zu der entsprechenden Anweisung verzweigt. Eine Ausnahme stellt die optionale Standard-Anweisung dar: Sie wird durch das Schlüsselwort `DEFAULT` eingeleitet und ausgeführt, falls keiner der kombinierten Bedingungsausdrücke zutrifft.

In Abbildung 3.11 werden die beiden Unterfälle der IF-THEN-ELSE-Anweisung (mit und ohne ELSE-Zweig) in zwei Produktionen definiert. Die dritte Produktion von *selection-statement* definiert die CASE-Anweisung. Die Produktionen für *selection* geben die Syntax der hierbei verwendbaren Konstruktionen der Anweisungsfolge an. Die aufgeführten Operatoren werden in Abschnitt 3.4.2 näher erläutert.

selection-statement	⟶	`IF` *expr* `THEN` *statement*
	|	`IF` *expr* `THEN` *statement* `ELSE` *statement*
	|	`CASE` *expr* `BEGIN` *selection-list* `END`
selection-list	⟶	*selection-list selection*
	|	*selection*
	|	ε
selection	⟶	*relation-op expr* `:` *statement*
	|	`=?` *expr* `:` *statement*
	|	`<>` *expr* `:` *statement*
	|	`IN [` *number* `..` *number* `]` `:` *statement*
	|	`IN [` *symbol-list* `]` `:` *statement*
	|	`DEFAULT` `:` *statement*
symbol-list	⟶	*symbol-list* `,` *symbol*
	|	*symbol*
	|	ε

Abbildung 3.11: Definition der Selektionsanweisung

3.3.5 Boolsche Anweisungen

Eine boolsche Anweisung besteht syntaktisch lediglich aus einem Ausdruck und einem terminalen Semikolon. Hier wird der ausdrucksorientierte Charakter der Verhaltensbeschreibungssprache deutlich, denn die boolsche Anweisung liefert den Wert des Ausdrucks zurück.

Abbildung 3.12 enthält die Produktion zur kontextfreien Definition der boolschen Anweisung. Der syntaktische Aufbau des Ausdrucks (*expr*) wird in Abschnitt 3.4 erläutert.

boolean-statement	⟶	*expr* `;`

Abbildung 3.12: Definition der boolschen Anweisung

3.4 Ausdrücke

Ein Ausdruck ist eine Folge von Operatoren und Operanden, die eine Berechnung spezifizieren. Beim Entwurf der hier vorgestellten Verhaltensbeschreibungssprache wurde auf eine

syntaktische Unterscheidung der verschiedenen Ausdruckstypen (z. B. boolscher, numerischer oder symbolischer Ausdruck) verzichtet. Die Definition eines Ausdrucks besteht daher aus Produktionen, die alle Ableitungen für *expr* darstellen.

Der Vorteil dieser Entscheidung liegt darin, daß so die Grammatik einfacher zu formulieren ist und der Blick auf die wesentlichen syntaktischen Strukturen nicht verschleiert wird. Ein Beispiel soll diese Aussage verdeutlichen: Hätte man numerische und symbolische Ausdrücke durch Produktionen für *num-expr* und *sym-expr* unterschieden, so hätte sich die Zahl der Produktionen für Selektionsanweisungen (siehe Abbildung 3.11) erhöht, weil für jedes Vorkommen von *expr* sowohl ein numerischer Ausdruck (*num-expr*) als auch ein symbolischer Ausdruck (*sym-expr*) zulässig ist. Ein weiterer Vorteil ist, daß eine klare Trennung zwischen Syntax und Semantik besteht, denn die Unterscheidung der Ausdruckstypen kann der semantischen Definition zugeordnet werden.

Der Nachteil dagegen ist der zwangsläufig größere Umfang der Semantikbeschreibung. Bei der Realisierung des Übersetzers müssen die semantischen Regeln ausprogrammiert werden und können nicht von einem Parsergenerator automatisch erzeugt werden (siehe Abschnitt 5.4.2.1, Seite 61).

Nach Abwägung der Vor- und Nachteile war für die Entscheidung das Argument der klaren Trennung von syntaktischen und semantischen Aspekten der Sprache ausschlaggebend. Die in den folgenden Abschnitten vorgenommene Einteilung der Produktionen für *expr* in arithmetische, boolsche und sonstige Ausdrücke dient daher nur einer besseren Strukturierung der Beschreibung und beinhaltet keine syntaktischen Informationen.

3.4.1 Arithmetische Ausdrücke

In der Grammatik der Verhaltensbeschreibungssprache existieren sieben verschiedene Operatoren: Für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division, die Potenzierung und die zwei Vorzeichen-Operatoren. Aus Gründen der Übersichtlichkeit sind die Summen-Operatoren `+` und `-` sowie die Produkt-Operatoren `*` und `/` jeweils zu *sum-op* bzw. *prod-op* zusammengefaßt worden. So reduziert sich die Anzahl der *expr*-Produktionen für die arithmetischen Ausdrücke auf vier:

1. Summe: Die Operatoren `+` und `-` sind zweistellig.
2. Produkt: Die Operatoren `*` und `/` sind zweistellig.
3. Potenzierung: Der Operator `^` ist zweistellig.
4. Vorzeichen: Die Operatoren `+` und `-` sind einstellig.

Wie der Abbildung 3.13 zu entnehmen ist, werden die zweistelligen Operatoren in Infix-Notation verwendet und die Vorzeichen-Operatoren werden vorangestellt.

expr	⟶	*expr sum-op expr*
	|	*expr prod-op expr*
	|	*expr pow-op expr*
	|	*sum-op expr*
sum-op	⟶	`+` | `-`
prod-op	⟶	`*` | `/`
pow-op	⟶	`^`

Abbildung 3.13: Definition der arithmetischen Ausdrücke

3.4.2 Boolsche Ausdrücke

Boolsche Ausdrücke dienen dazu, Wahrheitswerte zu berechnen. Bis auf den einstelligen Operator `NOT` sind alle Operatoren zweistellig und werden in Infix-Notation verwendet. Eine Unterteilung der Operatoren läßt sich wie folgt vornehmen:

- logische Operatoren (`NOT`, `AND`, `OR`)
  Diese Operatoren entsprechen den in der Logik verwendeten Operatoren $\neg$, $\wedge$ und $\vee$.

- Vergleichsoperatoren (`=?`, `<>`)
  Da der Gleichheitsoperator in *equation-statement* durch das Gleichheitszeichen (`=`) repräsentiert wird (siehe Abbildung 3.9), kann dieses Symbol nicht mehr als Vergleichsoperator eingesetzt werden. Der Übersetzer muß zur Erzeugung der korrekten internen Darstellung in der Lage sein, syntaktisch zwischen einer Zuweisung und einem Test auf Gleichheit zu unterscheiden. Auch für den Anwender sollte dieser Unterschied explizit sein. Daher wird der Vergleichsoperator durch ein Gleichheitszeichen mit anschließendem Fragezeichen (Hinweis auf den Testcharakter dieses Operators) symbolisiert. Der Test auf Ungleichheit hingegen kann ohne Konflikte in der gewohnten Schreibweise mit `<>` dargestellt werden. Das gleiche gilt für alle folgenden Operatoren.

- relationale Operatoren (`<`, `<=`, `>=`, `>`)
  Durch die Hintereinandersetzung der zwei Einzelzeichen bei `<=` und `>=` soll die in der Mathematik übliche Schreibweise zumindest angenähert werden, wie es auch in herkömmlichen Programmiersprachen wie C oder PASCAL üblich ist.

- Mengenoperator (`IN`)
  Dieser Operator kann als Element-Operator für zwei verschiedene Mengentypen benutzt werden:

  1. numerische Menge:
     Innerhalb eines eckigen Klammerpaares geben zwei durch zwei Punkte voneinander getrennte Zahlen die Unter- bzw. Obergrenze der in der Menge enthaltenen Werte an.
  2. Symbol-Menge:
     Innerhalb eines eckigen Klammerpaares steht die Folge derjenigen Symbole, die in der Menge enthalten sind.

In Abbildung 3.14 ist die Definition der boolschen Ausdrücke angegeben. Die syntaktischen Variablen *symbol-list* und *number* wurden in Abbildung 3.11 bzw. Abbildung 3.6 definiert.

expr	⟶	NOT *expr*
	|	*expr logical-op expr*
	|	*expr* =? *expr*
	|	*expr* <> *expr*
	|	*expr relation-op expr*
	|	*expr* IN [*number* .. *number*]
	|	*expr* IN [*symbol-list*]
logical-op	⟶	AND | OR
relation-op	⟶	< | <= | >= | >

Abbildung 3.14: Definition der boolschen Ausdrücke

3.4.3 Sonstige Ausdrücke

Die sonstigen Ausdrücke stellen bis auf zwei Ausnahmen (Funktionen und Klammerungen) die rekursiven Abschlüsse der Ableitungen für *expr* dar. Ein rekursiver Abschluß ist entweder eines der in Abschnitt 3.2.4 definierten literalen Konstanten oder ein Differential. Ein Differential besteht aus einem Bezeichner und einer beliebigen Anzahl von Apostrophen ('). Die Syntax des Differentials lehnt sich somit an die mathematische Schreibweise an.

Die beiden rekursiven Produktionen repräsentieren Funktionsaufrufe bzw. Klammerungen von Ausdrücken. Funktionsaufrufe werden durch einen einfachen Bezeichner eingeleitet, dem eine mit runden Klammern umschlossene Folge von Ausdrücken folgt. Diese Ausdrucksfolge stellt eine indirekte Rekursion dar. Auch Ausdrucksklammerungen werden mit runden Klammern verwendet.

Abbildung 3.15 zeigt die Definition der sonstigen Ausdrücke. Die literalen Konstanten *symbol*, *string* und *number* sind in Abbildung 3.6, die Bezeichner *identifier* und *simple-id* in den Abbildungen 3.2 bzw. 3.3 und die Ausdrucksfolge *expr-list* in Abbildung 3.10 definiert worden.

expr	⟶	*identifier*
	|	*symbol*
	|	*string*
	|	*number*
	|	*differential*
	|	*simple-id* (*expr*)
	|	(*expr*)
differential	⟶	*differential* '
	|	*identifier* '

Abbildung 3.15: Definition der sonstigen Ausdrücke

Kapitel 4

Semantik der Verhaltensbeschreibungssprache

Die in Kapitel 3 erläuterte kontextfreie Grammatik repräsentiert eine Obermenge der zulässigen Verhaltensbeschreibungen für hydraulische Komponenten in $^{\text{art}}$*deco*. Um eine korrekte Verhaltensbeschreibung zu definieren, sind weitere Einschränkungen notwendig: Den Sprachelementen werden Eigenschaften zugordnet, die zum Teil erst durch ihren Kontext bestimmt werden, wie z. B. der Typ eines Ausdrucks. Die folgende Semantikdefinition beschreibt also die Inhalte der Sprachelemente unter Berücksichtigung von Kontextabhängigkeiten.

4.1 Notation der Semantik

Im Gegensatz zur Syntax gibt es für die Semantik keine allgemein akzeptierte und standardmäßig verwendete Notation (vgl. Abschnitt 5.1.2.3 auf Seite 50). Viele semantische Zusammenhänge lassen sich nur schwer oder umständlich formalisieren und werden daher umgangssprachlich definiert. Aus diesem Grund wird im folgenden eine eigene, auf ihren speziellen Verwendungszweck zielende Notation[1] verwendet; es ist die sogenannte „syntaxgerichtete Definition“ in Anlehnung an [ASU88a].

Zu einer syntaxgerichteten Definition gehören zwei Elemente:

1. Kontextfreie Grammatik:
   Sie beschreibt die syntaktische Struktur der Übersetzereingabe und ist in Kapitel 3 definiert worden.

2. Semantische Regeln:
   Zu jedem Symbol der kontextfreien Grammatik gehört eine Menge von Attributen und zu jeder Produktion eine Menge von semantischen Regeln. Mit den semantischen Regeln werden die Attribute für die Symbole berechnet, die in der jeweiligen Produktion vorkommen.

[1] Da der Verwendungszweck des Übersetzers eine textuelle Transformation ist, besteht die Semantikbeschreibung im wesentlichen aus Operationen mit Zeichenketten, vgl. auch Abschnitt 4.1.1.

Während des Übersetzungsvorgangs baut der Übersetzer einen sogenannten Parse-Baum[2] auf, dessen Knoten Grammatiksymbole sind. Die Blätter sind Terminalsymbole; bei einem vollständig aufgebauten Parse-Baum repräsentieren sie die Eingabe des Übersetzers. Anhand der Baumstruktur läßt sich nur der syntaktische Aufbau des Quelltextes erkennen. Zur Beschreibung von semantischen Informationen werden die oben erwähnten Attribute verwendet und in den Knoten des Parse-Baums gespeichert.

Die Attribute in der Semantikdefinition des ^{art}*deco*-Übersetzers lassen sich in einem Durchlauf durch den Parse-Baum (von unten nach oben) mit Hilfe der semantischen Regeln berechnen. Jedes Attribut ist also entweder uninitialisiert oder trägt einen bereits ermittelten Wert, der sich aus dem Teilbaum ergab, dessen Wurzel der entsprechende Knoten ist. Die Attribute der Wurzel beinhalten dann die für die gesamte Verhaltensbeschreibung relevanten Werte.

4.1.1 Attribute und ihre Notation

Der Übersetzer für die Verhaltensbeschreibungssprache transformiert eine Eingabe in Infix-Notation in eine äquivalente Ausgabe in Präfix-Notation. Also besteht sowohl die Eingabe als auch die Ausgabe aus Zeichenketten. Beim Übersetzungsprozeß werden daher hauptsächlich String-Operationen durchgeführt. Zeichenketten, die in Teilbäumen gebildet werden, um dann an einem Knoten des Parse-Baums (bei einem fehlerfreien Programm spätestens an der Wurzel) zusammengefügt zu werden, benötigen bis zu ihrer Verwendung eine Struktur zur Zwischenspeicherung. Hierfür wird das Attribut s verwendet. Es beinhaltet die Zeichenkette, die der darunter befindliche Teilbaum als Ergebnis der Umwandlung der Infix-Eingabe in die Präfix-Ausgabe liefert.

Das zweite Attribut t gibt den Typ des aktuellen Knotens an. In einigen Situationen – insbesondere bei der Auswertung von Ausdrücken – ist der Typ der jeweiligen Eingabe relevant für die Gestalt oder Korrektheit der Ausgabe.

Für die Notation gilt: Sei X ein Grammatiksymbol und a ein Attribut von X. Dann bezeichnet $X.a$ den Wert, der aufgrund der semantischen Regel für das Attribut a in der benutzen X-Produktion berechnet wird. So ist beispielsweise *description.s* der Wert des Attributs s an der Wurzel des Parse-Baums und beinhaltet nach Beendigung des Übersetzungsvorgangs die Präfix-Ausgabe.

4.1.2 Semantische Regeln und ihre Notation

Für jede Produktion der kontextfreien Grammatik und jedes Attribut wird eine semantische Regel benötigt. Der Wert eines Attributs wird durch den Wert des Ausdrucks auf der rechten Seite der Zuweisung determiniert. Die Notation ist dabei für ein Attribut a eines Grammatiksymbols X wie folgt:

$X.a$:= *semantische Regel*

[2] Die formale Definition befindet sich in Abschnitt 5.1.2.2 auf Seite 49.

Um die semantischen Regeln formalisieren zu können, werden zur Beschreibung der Berechnungsvorschrift für die Attributwerte Kontrollstrukturen verwendet, wie sie in herkömmlichen Programmiersprachen zu finden sind. Die Schlüsselwörter und Operatoren dieser ausdrucksorientierten Metasprache sowie ihre Bedeutung werden hier nicht formal beschrieben, da sie in üblicher Weise benutzt werden:

- `if ( … ) then … else …`
  IF-THEN-ELSE-Ausdruck; sein Wert ist der Wert des Ausdrucks im THEN-Zweig, falls die in runde Klammern gesetzte Bedingung wahr ist, der Wert des Ausdrucks im ELSE-Zweig sonst.

- `case 1 : ( … ) then …`
  …
  `case` n `: ( … ) then …`
  n Alternativen; der Wert ist der Wert des Ausdrucks im THEN-Zweig der (einzigen) wahren Bedingung. Auch hier sind die Bedingungen mit runden Klammern umschlossen.

- `and`, `=`, `<>`
  Schlüsselwörter in Bedingungen; wie üblich können mehrere Bedingungsteile mit dem logischen Operator `and` verknüpft sowie Ausdrücke auf Gleichheit bzw. Ungleichheit getestet werden. Weiter sind Klammerungen mit runden Klammern erlaubt.

- `nil`
  Schlüsselwort für die leere Zeichenkette. Eine leere Zeichenkette ist dabei nicht ein uninitialisierter Wert für ein Attribut, sondern wird explizit zur Erzeugung einer leeren Ausgabe zugewiesen.

- `"`, ||
  Zeichenkettenkonstanten sind in Anführungszeichen (`"`) eingeschlossen. Mit dem zweistelligen Operator || können Zeichenketten konkateniert werden.

- `RISC`
  Dieser Ausdruck nimmt den Wert „wahr“ an, falls eine Operatorenreduzierung auf die Operatoren `+ * ^` vorgenommen werden soll, den Wert „falsch“ sonst. Zur Operatorenreduzierung siehe die Abschnitte 2.3.2 und 4.4.1.

Da in einer Produktion das gleiche Grammatiksymbol mehrfach auftreten kann, werden gleiche Nichtterminale durch eine tiefergestellte Zahl gemäß der Reihenfolge ihres Auftretens in der Produktion durchnumeriert. Dieses ist notwendig, weil gleiche Grammatiksymbole zwar die gleichen Attribute haben, sie aber in den verschiedenen Knoten des Parse-Baums nicht gleiche Attribut*werte* besitzen müssen.

Durch die Tatsache, daß die Ausgabe des art*deco*-Übersetzers eine Zeichenkette ist, kommt dem Attribut s als Träger der Präfix-Ausgabe eine besondere Rolle zu. Dieses Attribut wird daher bei jedem Grammatiksymbol verwendet und berechnet. Hingegen wird das Attribut t nur für die Nichtterminale *identifier* und *expr* verwendet. Um die semantischen Regeln übersichtlich zu halten und eine genaue Zuordnung zu den Produktionen der kontextfreien

Grammatik zu ermöglichen, wird daher nur die Berechnung des Zeichenketten-Attributs formal beschrieben, während die Berechnung des Typ-Attributs informal erläutert wird. Auf diese Weise gehört also zur n-ten Produktion einer Abbildung mit der Syntaxdefinition die n-te semantische Regel zur Berechnung von s in der korrespondierenden Abbildung mit der Semantik-Beschreibung.

4.2 Bezeichner

Jede Variable in der Verhaltensbeschreibung für eine hydraulische Komponente korrespondiert mit einer physikalischen Größe, die entweder an einer definierten Stelle (Gate) dieser Komponente beobachtet werden kann oder eine Eigenschaft der gesamten Komponente beschreibt. Die Referenzierung dieser Variablen mit Bezeichnern kann der Experte auf zwei verschiedene Arten vornehmen (vgl. dazu den Aufbau der Komponentenbeschreibungen in Abschnitt 6.1.3.1):

1. Referenzierung durch Namen:
   Der Experte referenziert die Variable mit einem einfachen Bezeichner *simple-id* und spezifiziert diesen – falls es sich um eine physikalische Größe an einem Gate handelt – mit der entsprechenden Gatenummer.
   Der Übersetzer erzeugt zuerst die interne Darstellung dieser Variablen und prüft dann, ob sie in der Komponentenbeschreibung vorhanden ist.

2. Referenzierung durch Alias-Namen:
   Da die Definition der Alias-Namen in der Komponentenbeschreibung eindeutig ist, reicht die alleinige Angabe eines einfachen Bezeichners *simple-id*, um die entsprechende Variable zu referenzieren.
   In diesem Fall prüft der Übersetzer direkt, ob der angegebene Bezeichner als Alias-Name in der Komponentenbeschreibung definiert ist und somit die entsprechende interne Darstellung ermittelt werden kann.

Für den ersten Fall gibt es nach Abbildung 3.3 drei verschiedene syntaktische Möglichkeiten, mit einem Bezeichner eine physikalische Größe zu referenzieren. Die zur Verdeutlichung erneut aufgeführte Syntax und die jeweilige Bedeutung ist dabei:

a) *identifier* $\longrightarrow$ *simple-id*$_1$ `[` *simple-id*$_2$ `]`
   Referenzierung der durch *simple-id*$_1$ bezeichneten physikalischen Größe der Komponente *simple-id*$_2$.

b) *identifier* $\longrightarrow$ *simple-id* `[` *number* `]`
   Referenzierung der durch *simple-id* bezeichneten physikalischen Größe an Gate *number* der Komponente, für die die Verhaltensbeschreibung definiert werden soll.

c) *identifier* $\longrightarrow$ *simple-id*
   Referenzierung der durch *simple-id* bezeichneten physikalischen Größe der Komponente, für die die Verhaltensbeschreibung definiert werden soll.

Da im zweiten Fall vom Experten nur ein einfacher Bezeichner angegeben wird, unterscheidet er sich syntaktisch nicht von der Möglichkeit c) des ersten Falls. Im Gegensatz hierzu kann aber jede der drei Bedeutungen a) bis c) intendiert sein. Welche Bedeutung konkret vorliegt, ermittelt der Übersetzer aus der Komponentenbeschreibung in zwei Schritten: Zuerst wird der zweite Fall angenommen und in der Komponentenbeschreibung die interne Darstellung des Bezeichners gesucht. Wird sie gefunden, so wird diese Darstellung als Ergebnis angesehen. Ansonsten wird der erste Fall angenommen und die Übersetzung entsprechend den semantischen Regeln in Abbildung 4.1 vorgenommen.

Diese Abbildung enthält die semantischen Regeln für das Attribut *identifier.s*. Jede Zeile korrespondiert mit der entsprechenden Produktion in Abbildung 3.3. Weiterhin existiert für das Grammatiksymbol *identifier* das Attribut *t*, das den Typ des Bezeichners angibt. Als Typ kommen die Werte *number*, *differential*, *symbol*, *string*, *name* oder *boolean* in Betracht. Er wird aus der Komponentenbeschreibung entnommen und als Attributwert von *identifier.t* direkt nach der Anwendung einer semantischen Regel für *identifier.s* gesetzt.

$$\begin{array}{lcl} \mathit{identifier.s} & := & \mathit{simple\text{-}id}_2.s \;\|\; \texttt{"@"} \;\|\; \mathit{simple\text{-}id}_1.s \\ \mathit{identifier.s} & := & \texttt{"GATE"} \;\|\; \mathit{number.s} \;\|\; \texttt{"@"} \;\|\; \mathit{simple\text{-}id.s} \\ \mathit{identifier.s} & := & \texttt{"SELF@"} \;\|\; \mathit{simple\text{-}id.s} \end{array}$$

Abbildung 4.1: Semantische Regeln für Bezeichner

4.3 Anweisungen

Die semantischen Regeln für die Anweisungen sind sehr einfach und beschränken sich im wesentlichen auf Zeichenkettenzuweisungen, ohne eine neue Zeichenkette aufzubauen. Dieses ist der Fall, weil die kontextfreie Grammatik hier sogenannte Kettenproduktionen enthält. Kettenproduktionen sind z. B. Produktionssequenzen der Form

$$\begin{array}{lcl} X & \longrightarrow & Y \\ Y & \longrightarrow & a \end{array}$$

für Nichtterminale X und Y sowie ein Terminal a. Sie dienen einer besseren Strukturierung der Grammatik, sind aber inhaltlich nicht notwendig. Die obigen zwei Produktionen könnten nämlich durch die eine Produktion

$$X \longrightarrow a$$

subsumiert werden, ohne die Sprache zu verändern. Bei der Implementierung eines Übersetzers wird man Kettenproduktionen ganz eliminieren, um die Effizienz zu steigern und die Zielcodegröße zu verringern.

Abbildung 4.2 enthält die semantischen Regeln, die zu den Produktionen in Abbildung 3.7 gehören. Den Klammerungen in der externen Darstellung mit `BEGIN` und `END` entsprechen gemäß der Lisp-ähnlichen internen Darstellung Klammerungen mit runden Klammern (vgl. siebte Regel). Leere Klammerungen können eliminiert werden, d. h. das entsprechende Zeichenketten-Attribut erhält den Wert `nil`. Anweisungssequenzen werden einfach konkateniert (vgl. vorletzte Regel).

description.s	:=	*statement.s*
statement.s	:=	*type-statement.s*
statement.s	:=	*equation-statement.s*
statement.s	:=	*relation-statement.s*
statement.s	:=	*condition-statement.s*
statement.s	:=	`"( TEST"` || *boolean-statement.s* || `")"`
statement.s	:=	`"("` || *statement-list.s* || `")"`
statement.s	:=	`nil`
statement.s	:=	`nil`
statement-list$_1$*.s*	:=	*statement-list*$_2$*.s* || *statement.s*
statement-list.s	:=	*statement.s*

Abbildung 4.2: Semantische Regeln für Anweisungen

4.3.1 Typdefinitionsanweisungen

Die Typdefinitionsanweisung dient dem Experten zur Kategorisierung der Verhaltensbeschreibungen für eine Komponente. Auf diese Weise kann z. B. die Modelltiefe auch textuell bezeichnet werden. Die derzeitige Übersetzerimplementation sieht jedoch eine Umsetzung in die Präfix-Ausgabe nicht vor, d. h. dem Zeichenketten-Attribut wird `nil` zugewiesen.

Der Bezeichner in der Typdefinitionsanweisung ist vom Typ *name* und darf in der Verhaltensbeschreibung nicht mit einem anderen Typ (z. B. *number* oder *symbol*) auftreten, sonst erfolgt eine Fehlermeldung.

In Abbildung 4.3 ist die semantische Regel zur Produktion aus Abbildung 3.8 aufgeführt.

type-statement.s	:=	`nil`

Abbildung 4.3: Semantische Regel für Typdefinitionsanweisungen

4.3.2 Gleichungsanweisungen

Gleichungsanweisungen können numerische oder symbolische Gleichungen definieren, die in ihrer Lisp-Darstellung vom $^{\text{art}}$*deco*-Constraintsystem verarbeitet werden. Für die Präfix-Notation ist dabei der Typ des ersten Ausdrucks bestimmend. Er entscheidet, ob die Lisp-Darstellung den Operator `IS` (bei symbolischen Gleichungen), den Operator `=` (bei numerischen Gleichungen) oder das Schlüsselwort `DIFF` (bei Vorliegen eines Differentials) erhält. Der zweite Ausdruck muß den gleichen Typ wie der erste besitzen, sonst erfolgt eine Fehlermeldung.

Abbildung 4.4 enthält die semantische Regel zur entsprechenden Produktion in Abbildung 3.9.

```
equation-statement.s  :=  if ( expr1.t = differential )
                             then "( DIFF" || expr1.s || expr2.s || ")"
                             else if ( expr1.t = symbol )
                                     then "( IS" || expr1.s || expr2.s || ")"
                                     else "( =" || expr1.s || expr2.s || ")"
```

Abbildung 4.4: Semantische Regel für Gleichungsanweisungen

4.3.3 Relationsanweisungen

Mit einer Relationsanweisung können beliebig viele n-stellige Tupel, deren Elemente zueinander in Relation stehen, definiert werden. Die Stelligkeit der Tupel wird beim Übersetzungsprozeß überprüft und es erfolgt eine Warnmeldung, falls die Stelligkeit eines Tupels aus *relation-list* nicht mit der des Tupels *identifier-list* übereinstimmt.

Die *identifier-list* legt fest, welche physikalischen Größen der betrachteten Komponente zueinander in Relation stehen. Die Tupel *relation* enthalten Ausdrücke für die jeweiligen Werte der korrespondierenden physikalischen Größen. Jede Relation trägt einen Namen (*simple-id*), der aber analog zum Namen in einer Typdefinitionsanweisung nur in der externen Wissensrepräsentation verwendet wird. Er soll dem besseren Verständnis für den Anwender dienen und taucht in der internen Darstellung nicht mehr auf. Trotzdem muß der Bezeichner *simple-id* vom Typ *name* sein und darf nicht als Bezeichner eines anderen Typs verwendet werden.

Die Abbildung 4.5 enthält die semantischen Regeln für die Relationsanweisung sowie für die Tupel. Die einzelnen Regeln beziehen sich auf die Produktionen in Abbildung 3.10.

```
relation-statement.s  :=  if ( identifier-list.s <> nil ) and
                             ( relation-list.s <> nil )
                             then "( (" || identifier-list.s || ")" ||
                                     relation-list.s || ")"
                             else nil
     relation-list1.s  :=  relation-list2.s || relation.s
      relation-list.s  :=  relation.s
      relation-list.s  :=  nil
           relation.s  :=  if ( expr-list.s <> nil )
                             then "(" || expr-list.s || ")"
                             else nil
         expr-list1.s  :=  expr-list2.s || "␣" || expr.s
          expr-list.s  :=  expr.s
          expr-list.s  :=  nil
   identifier-list1.s  :=  identifier-list2.s || "␣" || identifier.s
    identifier-list.s  :=  identifier.s
    identifier-list.s  :=  nil
```

Abbildung 4.5: Semantische Regeln für Relationsanweisungen

4.3.4 Selektionsanweisungen

Mit Hilfe von Selektionsanweisungen kann bei der Abarbeitung einer Verhaltensbeschreibung während der Konfigurationsprüfung abhängig vom Wert eines Ausdrucks die Gültigkeit von Anweisungen gesteuert werden. Grundsätzlich wird zwischen einer IF-THEN-ELSE-Anweisung und einer CASE-Anweisung unterschieden.

Bei einer IF-THEN-ELSE-Anweisung muß der Bedingungsausdruck (*expr*, siehe Abbildung 3.11) vom Typ *boolean* sein und kann somit die Werte „wahr" oder „falsch" annehmen. Ist dieses nicht der Fall, erfolgt eine Fehlermeldung. Bei einem wahren Bedingungsausdruck werden die Anweisungen im THEN-Zweig ausgeführt. Der ELSE-Zweig einer IF-THEN-ELSE-Anweisung wird nur bei einem negativen Bedingungsausdruck ausgeführt, ist aber optional. Unter Umständen sind hier weitere Selektionsanweisungen vorzufinden, die kaskadenförmig angeordnet sind. Um für bestimmte IF-THEN-Kaskaden eine übersichtlichere Notation anwenden zu können, steht die CASE-Anweisung zur Verfügung (s. u.).

Die semantische Regel für die einfache IF-THEN-Anweisung muß berücksichtigen, daß eine leere Anweisung im THEN-Zweig vorliegen kann. Dann aber hat diese Selektionsanweisung unabhängig vom Wahrheitswert des Bedingungsausdrucks keine Auswirkung. Sie ist also obsolet und wird nicht übersetzt (Zuweisung von `nil` zum Zeichenketten-Attribut *selection-statement.s*). Entsprechend können bei vollständigen IF-THEN-ELSE-Anweisungen vier verschiedene Fälle unterschieden werden, in denen die zwei Anweisungszweige leer sind oder nicht. Ist in nur einem Zweig die leere Anweisung, so erhält die Ausgabe das Lisp-Schlüsselwort `NIL` an der entsprechenden Stelle; sind beide Zweige leer, so ist die gesamte Selektionsanweisung überflüssig (siehe Abbildung 4.6).

```
selection-statement.s  :=  if ( statement.s <> nil )
                               then "( COND (" || expr.s || statement.s || ") )"
                               else nil
selection-statement.s  :=
       case 1 : ( statement_1.s <> nil ) and ( statement_2.s <> nil )
          then "( COND (" || expr.s || statement_1.s || ")" ||
                         "( ( NOT" || expr.s || ")" || statement_2.s || ") )"
       case 2 : ( statement_1.s <> nil ) and ( statement_2.s = nil )
          then "( COND (" || expr.s || statement_1.s || ")" ||
                         "( ( NOT" || expr.s || ") NIL ) )"
       case 3 : ( statement_1.s = nil ) and ( statement_2.s <> nil )
          then "( COND (" || expr.s || NIL )" ||
                         "( ( NOT" || expr.s || ")" || statement_2.s || ") )"
       case 4 : ( statement_1.s = nil ) and ( statement_2.s = nil )
          then nil
selection-statement.s  :=  if ( selection-list.s <> nil )
                               then "( COND" || selection-list.s || ")"
                               else nil
      selection-list_1.s  :=  selection-list_2.s || selection.s
        selection-list.s  :=  selection.s
```

Abbildung 4.6: Semantische Regeln für Selektionsanweisungen - Teil 1

Bei der CASE-Anweisung muß der Ausdruck *expr* den Typ *number* oder *symbol* besitzen. Er wird erst durch die Kombination mit dem Operator und dem zweiten Ausdruck vor jeder Anweisungszeile zu einem Bedingungsausdruck vom Typ *boolean*. Die Typen der Ausdrücke vor jeder Zeile müssen daher mit dem Typ des Ausdrucks hinter dem Schlüsselwort `CASE` übereinstimmen, sonst erfolgt eine Fehlermeldung. Die Anweisungen nach jeder der so gebildeten Bedingungsausdrücke, die den Wert „wahr“ annehmen, werden ausgeführt. Ist keine Bedingung wahr, wird die DEFAULT-Anweisung ausgeführt, falls eine definiert ist, sonst keine.

IF-THEN-ELSE-Anweisungen lassen sich in äquivalente CASE-Anweisungen umformen[3]. Die Wahl der anzuwendenden Selektionsanweisung hängt daher nur von der Übersichtlichkeit und Kompliziertheit der jeweiligen Konstrukte ab; die internen Darstellungen werden in Abbildung 4.6 gezeigt. In der Abbildung 3.11 (erste bis sechste Produktion) sind die den semantischen Regeln entsprechenden Produktionen der Grammatik zu finden.

Die Abbildungen 4.7 und 4.8 enthalten die semantischen Regeln für die Zeilen einer CASE-Anweisung. Der Ausdruck *expr** in den semantischen Regeln ist dabei der erste Teil des zu bildenden Bedingungsausdrucks und entspricht dem Ausdruck *expr* hinter dem Schlüsselwort `CASE` der Selektionsanweisung. Die entsprechenden Produktionen sind in Abbildung 3.11 ab der siebten Produktion zu finden.

```
selection.s  :=
      if ( statement.s <> nil )
         then "( (" || relation-op.s || expr*.s || expr.s || ")" || statement.s || ")"
         else "( (" || relation-op.s || expr*.s || expr.s || ") NIL )"
selection.s  :=
      case 1 : ( statement.s <> nil ) and ( expr.t = symbol )
         then "( ( EQUAL" || expr*.s || expr.s || ")" || statement.s || ")"
      case 2 : ( statement.s = nil ) and ( expr.t = symbol )
         then "( ( EQUAL" || expr*.s || expr.s || ") NIL )"
      case 3 : ( statement.s <> nil ) and ( expr.t <> symbol )
         then "( ( =" || expr*.s || expr.s || ")" || statement.s || ")"
      case 4 : ( statement.s = nil ) and ( expr.t <> symbol )
         then "( ( =" || expr*.s || expr.s || ") NIL )"
selection.s  :=
      case 1 : ( statement.s <> nil ) and ( expr.t = symbol )
         then "( ( NOT ( EQUAL" || expr*.s || expr.s || ") )" || statement.s || ")"
      case 2 : ( statement.s = nil ) and ( expr.t = symbol )
         then "( ( NOT ( EQUAL" || expr*.s || expr.s || ") ) NIL )"
      case 3 : ( statement.s <> nil ) and ( expr.t <> symbol )
         then "( ( NOT ( =" || expr*.s || expr.s || ") )" || statement.s || ")"
      case 4 : ( statement.s = nil ) and ( expr.t <> symbol )
         then "( ( NOT ( =" || expr*.s || expr.s || ") ) NIL )"
```

Abbildung 4.7: Semantische Regeln für Selektionsanweisungen - Teil 2a

[3] Die Schritte einer solchen Umformung werden hier nicht erläutert, da die Analogie zu allgemeinen Programmiersprachen groß ist (z. B. zu den if-else- und switch-Anweisungen in C) und damit die bekannten Techniken angewandt werden können.

```
selection.s  :=
    if ( statement.s <> nil )
       then "( ( AND ( >=" || expr*.s || number_1.s || ")" ||
                "( <=" || expr*.s || number_2.s || ") )" || statement.s || ")"
       else "( ( AND ( >=" || expr*.s || number_1.s || ")" ||
                "( <=" || expr*.s || number_2.s || ") ) NIL )"
selection.s  :=
    case 1 : ( statement.s <> nil ) and ( symbol-list.s <> nil )
       then "( ( MEMBER" || expr*.s || " '(" || symbol-list.s || ") )" ||
                statement.s || ")"
    case 2 : ( statement.s = nil ) and ( symbol-list.s <> nil )
       then "( ( MEMBER" || expr*.s || " '(" || symbol-list.s || ") ) NIL )"
    case 3 : ( statement.s <> nil ) and ( symbol-list.s = nil )
       then "( ( MEMBER" || expr*.s || NIL )" || statement.s || ")"
    case 4 : ( statement.s = nil ) and ( symbol-list.s = nil )
       then "( ( MEMBER" || expr*.s || NIL ) NIL )"
selection.s  :=  if ( statement.s <> nil )
                    then "( T" || statement.s || ")"
                    else "( T NIL )"
symbol-list_1.s  :=  symbol-list_2.s || "␣" || symbol-list.s
symbol-list.s    :=  symbol.s
symbol-list.s    :=  nil
```

Abbildung 4.8: Semantische Regeln für Selektionsanweisungen - Teil 2b

Da auf eine syntaktische Unterteilung der Ausdrücke nach Typen (also z. B. *num-expr* und *sym-expr*) verzichtet wurde, jedoch die interne Darstellung vom Typ des Ausdrucks abhängt, vergrößert sich der Formulierungsaufwand für die semantischen Regeln von CASE-Anweisungszeilen: Liegt ein symbolischer Ausdruck vor, so erfolgt ein Test auf Gleichheit mit dem Lisp-Schlüsselwort `EQUAL`, bei einem numerischen Typ mit `=`. Da jeweils noch eine leere Anweisung eliminiert werden kann, müssen in der zweiten und dritten semantischen Regel in Abbildung 4.7 vier Fälle unterschieden werden.

4.3.5 Boolsche Anweisungen

Boolsche Anweisungen haben i. d. R. kein Äquivalent in den allgemeinen Programmiersprachen. Sie bestehen nur aus einem Ausdruck, der vom Typ *boolean* sein muß (andernfalls erfolgt eine Fehlermeldung). Das Constraint-System von ^{art}*deco* überprüft die boolschen Anweisungen auf ihren Wahrheitsgehalt.

In Abbildung 4.9 ist die semantische Regel zur Berechnung der Zeichenkette für die interne Darstellung angegeben und bezieht sich auf die Produktion in Abbildung 3.12.

```
boolean-statement.s  :=  expr.s
```

Abbildung 4.9: Semantische Regel für boolsche Anweisungen

4.4 Ausdrücke

Der Ausdruck (*expr*) ist ein Grammatiksymbol, das neben dem Zeichenketten-Attribut s ein Typ-Attribut t besitzt. Die semantischen Regeln zur Typbestimmung werden im folgenden informal angegeben:

- arithmetische Ausdrücke: $expr.t := number$
- boolsche Ausdrücke: $expr.t := boolean$
- sonstige Ausdrücke: Der Typ hängt davon ab, mit welcher Produktion *expr* abgeleitet wird. Die möglichen Produktionen sind nach Abbildung 3.15:
  1. *identifier*: $expr.t := identifier.t$ (dieser Typ wurde der Komponentenbeschreibung entnommen, siehe Abschnitt 4.2)
  2. *symbol*: $expr.t := symbol$
  3. *string*: $expr.t := string$
  4. *number*: $expr.t := number$
  5. *differential*: $expr.t := number$
  6. Funktion: $expr.t := number$ (es wird angenommen, daß Funktionen keinen anderen Typ für den Funktionswert haben)
  7. Klammerung: $expr.t := expr.t$

In den folgenden Abschnitten werden die semantischen Regeln zur Berechnung der Präfix-Ausgabe für Ausdrücke erläutert.

4.4.1 Arithmetische Ausdrücke

Neben der Transformation in die interne Präfix-Notation ist in den semantischen Regeln für arithmetische Ausdrücke die evtl. erwünschte Reduzierung der Operatoren auf `+ * ^` zu berücksichtigen (vgl. Abschnitt 2.3.2).

Bei der Operatorenreduzierung müssen vom Übersetzer keine arithmetischen Operationen durchgeführt werden; sie ist auf rein textueller Basis möglich. Angewandt werden die bekannten Regeln für reele Zahlen a und b:

- Elimination der Subtraktion durch Addition:
  $a - b = a + -1 * b$
- Elimination der Division durch Multiplikation und Potenzierung:
  $\frac{a}{b} = a * b^{-1}$

Abbildung 4.10 enthält die entsprechenden semantischen Regeln und bezieht sich auf die ersten vier Produktionen in Abbildung 3.13. Der Ausdruck `RISC` gibt dabei an, ob eine Operatorenreduzierung vorgenommen werden soll (Wert von `RISC` ist „wahr“) oder nicht (Wert ist „falsch“).

```
expr1.s  :=  if ( RISC ) and ( sum-op.s = "-" )
                then "( +" || expr1.s || "( * -1" || expr2.s || ") )"
                else "(" || sum-op.s || expr2.s || expr3.s || ")"
expr1.s  :=  if ( RISC ) and ( prod-op.s = "/" )
                then "( *" || expr1.s || "( EXPT" || expr2.s || "-1 ) )"
                else "(" || prod-op.s || expr2.s || expr3.s || ")"
expr1.s  :=  "( EXPT" || expr2.s || expr3.s || ")"
expr1.s  :=  if ( sum-op.s = "-" )
                then "-" || expr2.s
                else expr2.s
```

Abbildung 4.10: Semantische Regeln für arithmetische Ausdrücke

4.4.2 Boolsche Ausdrücke

Wie auch in Abschnitt 4.3.4 muß bei der Transformation von boolschen Ausdrücken auf den Typ der durch einen Operator verbundenen Teilausdrücke geachtet werden, damit unterschiedliche interne Darstellungen erzeugt werden können. Weitere Einzelheiten sind der Abbildung 4.11 zu entnehmen, die die semantischen Regeln zu den ersten sieben Produktionen in Abbildung 3.14 auflistet.

```
expr1.s  :=  "( NOT" || expr2.s || ")"
expr1.s  :=  "(" || logical-op.s || expr2.s || expr3.s || ")"
expr1.s  :=  if ( expr2.t = symbol )
                then "( EQUAL" || expr2.s || expr3.s || ")"
                else "( =" || expr2.s || expr3.s || ")"
expr1.s  :=  if ( expr2.t = symbol )
                then "( NOT ( EQUAL" || expr2.s || expr3.s || ") )"
                else "( NOT ( =" || expr2.s || expr3.s || ") )"
expr1.s  :=  "(" || relation-op.s || expr2.s || expr3.s || ")"
expr1.s  :=  "( AND ( >=" || expr2.s || number1.s || ")" ||
                "( <=" || expr2.s || number2.s || ") )"
expr1.s  :=  if ( symbol-list.s <> nil )
                then "( MEMBER" || expr2.s || " '(" || symbol-list.s || ") )"
                else "( MEMBER" || expr2.s || "NIL )"
```

Abbildung 4.11: Semantische Regeln für boolsche Ausdrücke

4.4.3 Sonstige Ausdrücke

Die ersten fünf Produktionen der kontextfreien Grammatik für sonstige Ausdrücke in Abbildung 3.15 sind Kettenproduktionen, so daß die semantischen Regeln einfach aufgebaut sind.

Die sechste Produktion ermöglicht Funktionsaufrufe. Da dem Übersetzer keine Informationen über die Stelligkeit der Funktion vorliegen, können hier keine semantischen Prüfungen vorgenommen werden. An der letzten semantischen Regel ist zu erkennen, daß Klammern in der Präfix-Darstellung eliminiert werden, da hier keine Mehrdeutigkeiten auftreten können[4]. Die Abbildung 4.12 enthält die formale Darstellung der semantischen Regeln für die sonstigen Ausdrücke.

```
expr.s   :=  identifier.s
expr.s   :=  if ( expr-list.s <> nil )
                then "(" || simple-id.s || expr-list.s || ")"
                else simple-id.s
expr.s   :=  if ( symbol.s <> nil )
                then "'" || symbol.s
                else "NIL"
expr.s   :=  string.s
expr.s   :=  number.s
expr.s   :=  differential.s
expr1.s  :=  expr2.s
```

Abbildung 4.12: Semantische Regeln für sonstige Ausdrücke

[4]In der Infix-Darstellung werden die Mehrdeutigkeiten durch Regeln (z. B. „Punkt-vor-Strich“-Regel) oder Klammern beseitigt.

Kapitel 5

Implementierungsansätze

Dieses Kapitel befaßt sich mit den Überlegungen, die zur Realisierung eines Übersetzers anzustellen sind. Hierzu gehört die Auswahl der für das $^{\text{art}}$*deco*-Projekt relevanten Komponenten (Abschnitt 5.1) und die Diskussion der Implementierungart (Abschnitte 5.2 bis 5.4).

Neben den generellen Vor- und Nachteilen der beschriebenen Implementierungsalternativen wird die spezielle Domäne des Übersetzers dieser Diplomarbeit berücksichtigt. Der im Bereich technischer Expertensysteme erforderlichen Flexibilität der Sprachdefinition kann mit dem Einsatz von übersetzererzeugenden Werkzeugen Rechnung getragen werden; daher liegt hier der Schwerpunkt der Ausführungen.

5.1 Komponenten eines Übersetzers

Die Bestandteile eines typischen Übersetzers sowie ihre Funktionsweise werden in diesem Abschnitt nur angesprochen, aber nicht vertiefend erläutert. Verschiedene Werke der Standardliteratur (vor allem [ASU88a, ASU88b], aber auch [Kas90] und [Her92]) behandeln dieses Thema unterschiedlich ausführlich. Wichtig ist in diesem Abschnitt jedoch die Fragestellung, welche Komponenten zur Realisierung des Übersetzers im Rahmen des $^{\text{art}}$*deco*-Systems benötigt werden und welche nicht berücksichtigt werden müssen.

5.1.1 Überblick

Abbildung 5.1 enthält in Anlehnung an [ASU88a] schematisch die Phasen, die während der Übersetzung eines Quellprogramms nacheinander durchlaufen werden. In jeder Phase wird das Quellprogramm von einer Darstellung in eine andere überführt, bis am Ende das Zielprogramm erzeugt ist. Diese Unterteilung ist jedoch mehr logischer Natur. Bei der Implementierung von Übersetzern werden diese Phasen bzw. die zugehörigen Komponenten aus Effizienzgründen i. d. R. nicht so scharf getrennt. Auch die explizite Erzeugung aller Zwischendarstellungen ist eher die Ausnahme.

Phasenübergreifend sind die Komponenten zur Verwaltung einer Symboltabelle und zur Fehlerbehandlung. In der Symboltabelle werden z. B. Informationen gesammelt, die nur in einer bestimmten Übersetzungsphase ermittelt werden können, jedoch in einer anderen Phase benötigt werden. Auf Fehler kann in jeder Phase gestoßen werden, deshalb ist auch die Fehlerbehandlung allen Phasen zuzuordnen.

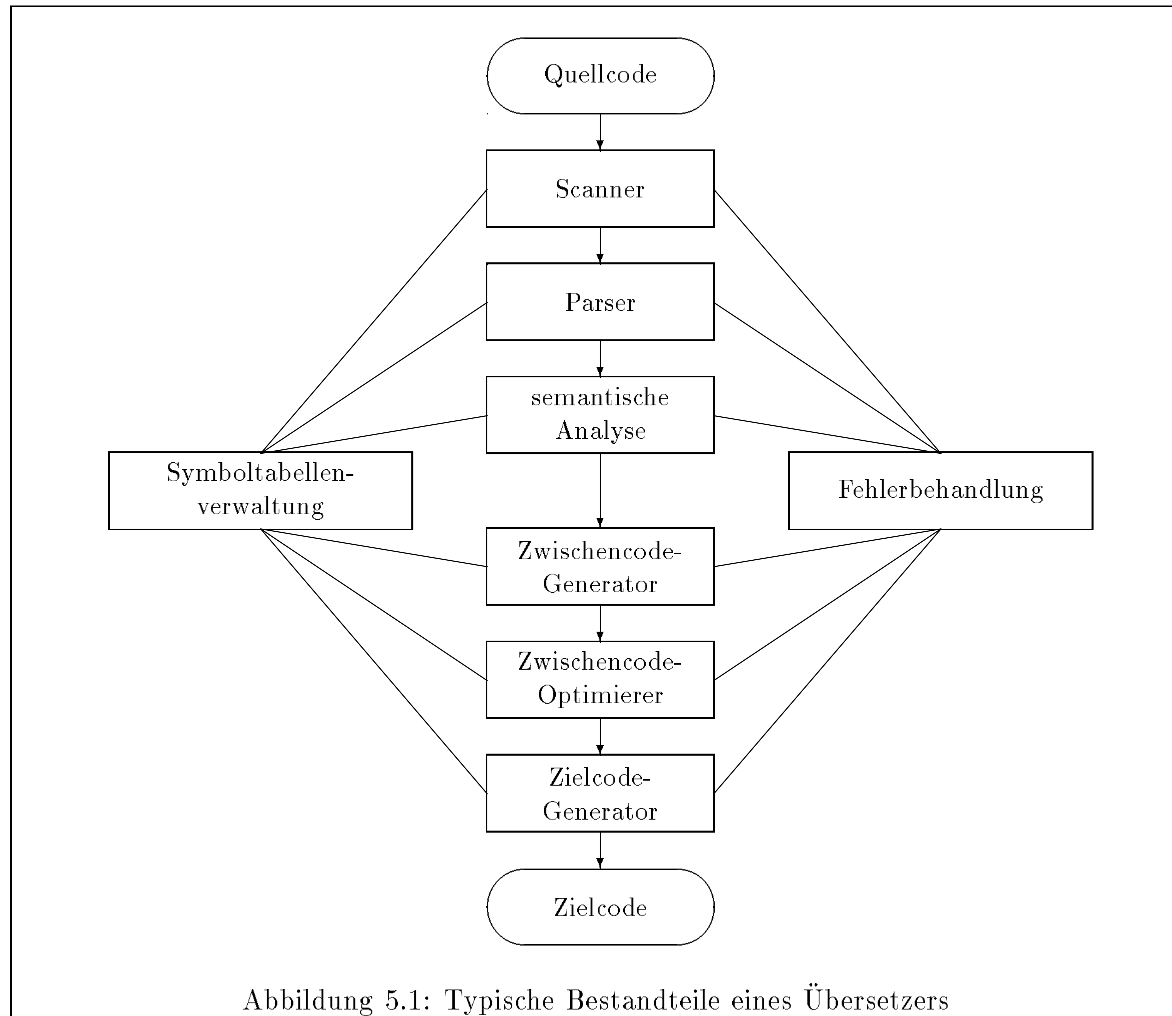


Abbildung 5.1: Typische Bestandteile eines Übersetzers

Eine grobe Unterteilung des Übersetzungsvorgangs kann in die Analysephase und die Synthesephase erfolgen. Die Analyse des Quellprogramms wird in den ersten drei Komponenten vorgenommen. Hier wird geprüft, ob es sich bei dem Quellprogramm um eine übersetzbare Eingabe handelt. Im positiven Fall ist das Ergebnis der Analysephase dann eine Zwischendarstellung. Diese Transformation ist nur eine strukturelle Umformung der Informationen aus dem Quellprogramm und kann unabhängig von der Zielmaschine durchgeführt werden.

Besonderer Implementierungstechniken bedarf es hingegen in der Synthesephase, der die letzten drei Übersetzerkomponenten zuzuordnen sind. Hier wird die Zwischendarstellung optimiert und dann in die Maschinensprache der Zielmaschine übersetzt. Die Implementierung ist also von der jeweiligen Zielmaschine abhängig und daher schwerer zu standardisieren (und damit zu automatisieren) als die am Quellprogramm orientierten Transformationsverfahren der Analysephase.

5.1.2 Komponenten der Analysephase

5.1.2.1 Scanner

Der Scanner führt die lexikalische Analyse durch. In dieser Phase wird geprüft, ob der Quelltext durch die Erkennung und Klassifizierung bestimmter Textteile zerlegt werden kann. Dazu liest der Scanner die Zeichen des Quellprogramms sequentiell ein und wandelt sie nach vorgegebenen Regeln in eine Folge von Symbolen um. Jedes Symbol stellt eine logisch zusammengehörige Folge von Zeichen dar und wird Token genannt. Typische Beispiele für Token sind Schlüsselwörter, Bezeichner oder Operatoren.

Die Token lassen sich wie folgt nach ihrem Informationsgehalt unterscheiden: Da die Menge der Schlüsselwörter in einer Programmiersprache endlich ist, kann zu den Schlüsselwörtern *je ein* unterschiedliches Token definiert werden. Ein solches Token besitzt dann in der vom Scanner erzeugten Tokenfolge den gleichen Informationsgehalt wie die textuelle Repräsentation des Schlüsselworts an seiner Quelltextposition. Dagegen ist die Ausprägungsmenge z. B. für numerische Konstanten theoretisch unendlich groß. Wird aber *nur ein* Token für das Auftreten einer beliebigen numerischen Konstante vereinbart, muß der Scanner zusätzliche Informationen wie den Wert der Konstanten bereitstellen. Solche Informationen werden in der Symboltabelle abgelegt und können in den folgenden Übersetzungsphasen abgerufen werden.

Neben der Transformation des Quelltextes in die Tokenfolge und Symboltabelleneinträge ist eine weitere Aufgabe des Scanners, den Quelltext um semantisch unbedeutende Informationen zu bereinigen, um sie nicht an die nächsten Übersetzungsphasen weiterzuleiten. Hierbei handelt es sich im wesentlichen um Kommentare und Füllzeichen, also um Sprachelemente, die nur der besseren Verständlichkeit des Quelltextes für menschliche Leser dienen.

5.1.2.2 Parser

Die syntaktische Analyse der vom Scanner gelieferten Tokenfolge führt der Parser durch. In dieser Übersetzungsphase wird geprüft, ob die Tokenfolge ein Wort darstellt, das von der Grammatik der Quellsprache erzeugt werden kann. Dazu versucht der Parser, zu der Tokenfolge mit Hilfe der kontextfreien Grammatik einen sogenannten Parse-Baum mit den unten genannten Eigenschaften aufzubauen. Im positiven Fall ist die Tokenfolge aus dem Startsymbol der Grammatik herleitbar und damit das Quellprogramm syntaktisch korrekt. Der Aufbau des Parse-Baums zeigt dann auch an, *wie* diese Herleitung möglich ist.

Der vollständige Parse-Baum repräsentiert also die syntaktische Struktur der Eingabe, wobei die Regeln dieser Strukturierung durch die zugehörige kontextfreie Grammatik spezifiziert werden. In Anlehnung an [ASU88a] besitzt ein Parse-Baum zu einer gegebenen kontextfreien Grammatik die folgenden Eigenschaften:

1. Die Wurzel ist mit dem Startsymbol der Grammatik markiert.
2. Jedes Blatt ist entweder mit einem Terminal oder mit ε markiert.
3. Jeder innere Knoten ist mit einem Nichtterminal der Grammatik markiert.

4. Sei ein innerer Knoten mit dem Nichtterminal A markiert, und seien die Nachfolger dieses Knotens von links nach rechts mit den Symbolen X_1, X_2, ..., X_n markiert, wobei jedes X_i entweder ein Terminal oder ein Nichtterminal ist. Dann ist $A \longrightarrow X_1 X_2 \cdots X_n$ eine Produktion der kontextfreien Grammatik.

Die Blätter des Parse-Baums stellen dann von links nach rechts gelesen das Wort dar, welches aus dem Nichtterminal an der Wurzel hergeleitet wurde. Also leitet der Parse-Baum genau das an seinen Blättern ablesbare Wort her. Zu einem gegebenen Wort einer kontextfreien Grammatik kann es jedoch mehrere Parse-Bäume und damit Ableitungen dieses Wortes geben. Die kontextfreie Grammatik ist dann mehrdeutig.

Verschiedene Parse-Bäume eines Wortes haben jedoch i. d. R. auch mehrere Bedeutungen, die z. B. die Assoziativität (Auswertungsreihenfolge von Terminalsymbolen gleicher Priorität) betreffen. Daher sollten Mehrdeutigkeiten in Grammatiken zur Beschreibung der Syntax einer Programmiersprache vermieden werden. Ist es nicht möglich, eine eindeutige Grammatik zu entwerfen, oder werden in der Grammatik Mehrdeutigkeiten bewußt zugelassen[1], so müssen diese mit Hilfe spezieller Regeln aufgelöst werden (siehe Abschnitt 6.2.1).

5.1.2.3 Semantische Analyse

Die semantische Analyse berücksichtigt, daß viele Programmkonstrukte i. d. R. nicht isoliert betrachtet werden können. Sie besitzen Eigenschaften, deren Ausprägungen sich erst durch die Einbeziehung ihres Umfeldes ermitteln lassen. Diese sogenannten kontextabhängigen Eigenschaften lassen sich in Attributen speichern und den Knoten des Parse-Baums zuordnen. Die Ausgabe der semantischen Analyse ist also ein solcher um berechnete Attribute erweiterter Parse-Baum („attributierter Parse-Baum“).

Ein typisches Beispiel für kontextabhängige Attribute ist der Typ eines Ausdrucks. Wenn die darin verwendeten Operatoren überladen (d. h. für unterschiedliche Operandentypen definiert) sind, dann hängt er von den Typen der beteiligten Variablen und Literale ab. Neue Attributwerte werden durch semantische Regeln auf der Grundlage bereits ermittelter und im Parse-Baum gespeicherter Attributwerte berechnet. Hierzu existieren verschiedene Baumdurchlaufverfahren (vgl. [Kas90]). In Abschnitt 6.2.2.1 wird die bei der Implementation des *artdeco*-Übersetzers angewandte Methode ausführlich beschrieben.

Ein formales Modell zur Beschreibung der semantischen Eigenschaften von Programmen unter Berücksichtigung der Eigenschaften ihrer Sprachkonstrukte sind attributierte Grammatiken. Sie enthalten präzise Regeln zur Berechnung der kontextabhängigen Attribute. Andere Ansätze sind z. B. die Denotationale Semantik, die Van Wijngaarden Grammatik, die Vienna Definition Language oder die Axiomatische Semantik (vgl. [Kas90]), von denen sich keiner zur allgemeinen Verwendung durchsetzen konnte. Die Einführung eines dieser Formalismen erschien jedoch im Zusammenhang mit der Erstellung des vergleichsweise einfachen *artdeco*-Übersetzers der Problematik nicht angemessen. Aus diesem Grund wurde zur Beschreibung der Semantik die in Abschnitt 4.1 erläuterte „syntaxgerichtete Definition“ verwendet.

[1] Mehrdeutigkeiten in der Grammatikdefinition können dazu beitragen, umständliche und für den menschlichen Leser schwer nachvollziehbare Konstruktionen zu vermeiden.

Relevanz für den ^{art}*deco*-Übersetzer

Für den Übersetzer der ^{art}*deco*-Verhaltensbeschreibungssprache sind die Phasen der lexikalischen und der syntaktischen Analyse von großer Wichtigkeit, denn ohne die hier vorgenommenen Prüfungen kann nicht entschieden werden, ob die jeweilige Eingabe formal korrekt ist. Da aber nur formal korrekte Verhaltensbeschreibungen in ihre äquivalente interne Darstellung übersetzt werden können, sind der Scanner und der Parser unverzichtbare Bestandteile des ^{art}*deco*-Übersetzers.

Auch die Semantikanalyse wird in diesem Übersetzer unbedingt benötigt. Semantisch fehlerhafte Eingaben müssen so früh wie möglich entdeckt und zurückgewiesen werden, damit sie nicht weiterverarbeitet werden und u. U. inkonsistente Wissensbasen zur Folge haben.

5.1.3 Komponenten der Synthesephase

5.1.3.1 Zwischencodegenerator

Der Zwischencodegenerator erzeugt aus den Informationen der Analysephase eine Zwischendarstellung des Quellprogramms. Im allgemeinen handelt es sich bei dieser Zwischendarstellung um ein Programm für eine abstrakte Maschine, das leicht in das Zielprogramm überführbar ist.

Als Beispiel für einen sinnvollen Zwischencode sei der sogenannte „Drei-Adreß-Code" angeführt. Er besteht aus einer Folge von Instruktionen, von denen jede höchstens drei Operanden enthält. Diese Darstellung weist damit bereits wesentliche Eigenschaften von Assemblersprachen für existierende Prozessoren auf, ist aber dennoch prozessorunabhängig formulierbar.

5.1.3.2 Zwischencodeoptimierer

In der Phase der Codeoptimierung wird versucht, den Zwischencode hinsichtlich seiner Laufzeit oder seines Umfangs semantikerhaltend zu verbessern, damit ein effizienterer Zielcode erzeugt werden kann. Hierzu wird ein bestimmter Aufwand benötigt, um Optimierungspotentiale zu entdecken und die Optimierungen durchzuführen. Dieser Aufwand kann in allgemeinen Übersetzern erheblich sein, ermöglicht jedoch i. d. R. eine deutliche Laufzeitverbesserung des erzeugten Maschinenprogramms.

Typische Arten der Optimierung sind die Eliminierung von nicht erreichbaren Programmteilen (z. B. der THEN-Zweig einer Selektionsanweisung, deren Bedingung immer falsch ist) und die sogenannte „Konstantenfaltung", bei der Formeln mit konstanten Operanden zur Übersetzungszeit ausgewertet werden. Bei der ersten Optimierungsart wird die spätere Zielcodegröße verringert; bei der zweiten wird die Ausführungszeit verringert, da die notwendigen numerischen Berechnungen nicht zur Laufzeit, sondern bereits zur Übersetzungszeit durchgeführt werden.

5.1.3.3 Zielcodeerzeuger

Der Zielcodeerzeuger transformiert den optimierten Zwischencode in den ausführbaren Maschinencode. Wichtige Aspekte sind hierbei die Speicherallokation für die Bezeichner des Quelltextes und die Erzeugung von verschiebbarem Zielcode. Von den behandelten Übersetzerkomponenten ist diese Komponente also am stärksten von der Zielmaschine abhängig.

Relevanz für den artdeco-Übersetzer

In Abschnitt 4.1.1 wurde dargelegt, daß sich die interne Darstellung als „Zielcode" einer Verhaltensbeschreibung für Komponenten im artdeco-System bereits mit Hilfe der semantischen Regeln in der Semantikanalysephase ermitteln läßt. Der Wurzelknoten des zugehörigen Parse-Baums enthält dann diesen Wert in seinem Zeichenketten-Attribut. Auch Optimierungen wie die Eliminierung überflüssiger Konstrukte (z. B. leere Relationsdefinitionen) und die algebraischen Umformungen lassen sich mit den semantischen Regeln vornehmen.

Das bedeutet, die vollständige, optimierte Zieldarstellung ist bereits nach der lexikalischen, syntaktischen und semantischen Analyse bekannt. Aus diesem Grund sind für den artdeco-Übersetzer keine weiteren Komponenten notwendig und die drei Synthesephasen können entfallen.

5.1.4 Phasenübergreifende Komponenten

5.1.4.1 Symboltabelle

Die Symboltabelle dient der Zwischenspeicherung von Informationen. Sie ist eine Datenstruktur, die am Ende des Übersetzungsvorgangs für jeden im Quelltext auftretenden Bezeichner einen Eintrag mit Feldern für die zugehörigen Informationen enthält. Diese Felder werden auch „Attribute" genannt, sind aber nicht mit den Attributen des Parse-Baums zu verwechseln.

Eine Symboltabelle muß verwaltet werden, wenn in einer Übersetzungsphase ein benötigter Attributwert nicht zur Verfügung steht, er aber in einer vorgelagerten Phase ermittelt werden kann. Dieser Attributwert wird dann in der vorgelagerten Phase in die Symboltabelle eingetragen und steht damit den nachfolgenden Phasen zur Verfügung. Die Voraussetzung für die effiziente Nutzung einer Symboltabelle ist daher der schnelle speichernde und lesende Zugriff auf die Attributwerte eines Eintrags.

5.1.4.2 Fehlerbehandlung

Die Fehlerbehandlung ist für das Erkennen und Melden von Fehlern zuständig. In jeder Phase der Übersetzung können Situationen auftreten, die die Transformation der Eingabe in eine äquivalente Ausgabe verhindern. Bei den zur Übersetzungszeit erkennbaren Fehlern sind vier verschiedene Fehlerarten zu unterscheiden:

1. Lexikalische Fehler:
   Der Quelltext besteht nicht ausschließlich aus den Terminalen der Sprachdefinition (erkennbar während der lexikalischen Analyse).

2. Syntaktische Fehler:
   Die Tokenfolge verletzt die Strukturregeln der Sprachdefinition (erkennbar während der syntaktischen Analyse).

3. Semantische Fehler:
   Semantische Fehler sind während der semantischen Analyse erkennbar. Es handelt sich um syntaktisch korrekte Konstrukte, die jedoch für die darin enthaltenen Operationen keinen Sinn ergeben. Der Semantikdefinitionsteil, der zur Übersetzungszeit überprüfbar ist, heißt *statische* Semantik; in Abgrenzung hierzu ist die sog. *dynamische* Semantik nur zur Laufzeit überprüfbar.

4. Rechnerabhängige Fehler:
   Die letzte Fehlerart kann in jeder Übersetzungsphase auftreten und ist abhängig vom verwendeten Rechner- bzw. Softwaresystem. Sie tritt auf, wenn z. B. aus Gründen der Speicherknappheit eine weitere Übersetzung unmöglich ist.

Relevanz für den ^art^*deco*-Übersetzer

Die beschriebenen phasenübergreifenden Komponenten werden selbstverständlich auch im ^art^*deco*-Übersetzer benötigt:

- Die Symboltabelle muß Attribute für Alias-Namen, Typ und Wertebereich der Bezeichner aufnehmen können, damit diese Informationen bei Ausführung der semantischen Regeln zur Vefügung stehen (vgl. Abschnitt 6.1.3.2).

- Die Fehlerbehandlungskomponente muß in der Lage sein, die während des Übersetzungsvorgangs auftretenden Fehler sofort dem Benutzer anzuzeigen. Um eine schnelle Fehlerkorrektur zu ermöglichen, ist die Angabe weiterer Hinweise (wie z. B. der Fehlerart) und die Kennzeichnung der fehlerhaften Quelltextstelle vorzusehen.

5.2 Grundsätzliche Implementierungsalternativen

In Anlehnung an [Wai94] lassen sich die Implementierungsansätze für Übersetzer nach der Art der Kodierung ihrer Sprachspezifikation als *operational* oder *deklarativ* unterscheiden.

Operationale Implementation

Die operationale Implementation wird durch eine manuelle Ausprogrammierung des Übersetzers in einer Programmiersprache der dritten Generation vorgenommen, vorzugsweise in PASCAL oder C. Hierbei gehen die Spezifikationen der Quellsprache und der Zielsprache zum großen Teil in programmiertechnischen Details unter, sie sind also nicht explizit sichtbar.

Aus diesem Grund wird die Wartung des Übersetzers bei einer manuellen Ausprogrammierung wesentlich erschwert; jede Änderung der Quell- oder Zielsyntax zieht i. a. umfangreiche Änderungen der Implementation nach sich (vgl. Abschnitt 5.3).

Überarbeitungen dieser Art können jedoch im Rahmen des *artdeco*-Expertensystems vorkommen, weil sich erfahrungsgemäß während der Test- und frühen Anwendungsphase eines Systems noch neue Anforderungen ergeben, die auch eine Überarbeitung der Wissensrepräsentation erfordern.

Deklarative Implementation

Ein anderer Weg ist die deklarative Implementation, bei der das Übersetzungsproblem beschrieben wird, nicht aber die dazu notwendige Lösung im Detail. Bei dieser Alternative erzeugen Generatoren aus der Quell- und Zielsprachenspezifikation die notwendigen Übersetzerkomponenten, so daß der Entwickler sich auf das Problem des Sprachentwurfs konzentrieren kann. Sind die Quell- und Zielsprachensyntax fertiggestellt, müssen sie in die (deklarative) Definition des Übersetzers überführt werden. Dieser Aufwand ist jedoch vergleichsweise gering; auch Syntaxänderungen können schnell abgebildet werden. Die Generierung des neuen Übersetzers erfolgt dann automatisch.

Der Nachteil dieser Vorgehensweise ist jedoch die im allgemeinen geringere Effizienz eines automatisch erzeugten Übersetzers gegenüber einem handimplementierten. Die zu erwartende Effizienz ist aber für den Einsatz im *artdeco*-System ausreichend, da jede Komponentenbeschreibung sofort nach ihrer Eingabe übersetzt wird; es handelt sich also nur um vergleichsweise kleine Datenmengen. In Abschnitt 5.4 werden die Werkzeuge zur automatischen Übersetzergenerierung näher erläutert. Insbesondere auf die im Rahmen dieser Diplomarbeit verwendeten Generatoren Lex und Yacc wird in Abschnitt 5.4.2 eingegangen.

5.3 Manuelle Ausprogrammierung

Wie in Abschnitt 5.1 gezeigt wurde, sind für den *artdeco*-Übersetzer die Phasen der lexikalischen, syntaktischen und semantischen Analyse die wichtigsten. Da es sich hierbei um einen relativ einfachen Übersetzer handelt, können viele Aufgaben der semantischen Analyse (z. B. die Typüberprüfung oder die Erzeugung der Präfix-Ausgabe) schon parallel zur Syntaxanalyse erfüllt werden.

Um den direkten Bezug der Semantik zum entsprechenden syntaktischen Konstrukt zu verdeutlichen, wurde in der Semantikdefinition jeder semantischen Regel eine Produktion der kontextfreien Grammatik zugeordnet (vgl. Abschnitt 4.1). Somit kristallisieren sich – abgesehen von der Symboltabellenverwaltung und der Fehlerbehandlung – der Scanner und der Parser als die wesentlichen Bestandteile des zu implementierenden Übersetzers heraus; sie werden daher im folgenden behandelt.

5.3.1 Der Scanner

Die Grundlage der Scannerimplementation bilden die als Notation für die lexikalische Analyse in Abschnitt 3.1.1 eingeführten regulären Ausdrücke. Jedem regulären Ausdruck entspricht eine Menge von Zeichenketten. Der Scanner hat die Aufgabe, nach dem Prinzip der maximalen Überdeckung (s. u.) zu einer eingelesenen Zeichenkette (also einem Ausschnitt des Quelltextes) den entsprechenden regulären Ausdruck zu ermitteln.

Der *entsprechende* reguläre Ausdruck ist dabei derjenige, der gerade die aktuelle Zeichenkette überdeckt. Der Scanner liefert dann das diesem regulären Ausdruck zugeordnete Token zur Weiterverarbeitung an den Parser. Dabei dürfen keine Mehrdeutigkeiten existieren; zu jeder möglichen Zeichenkette muß für den Fall einer Überdeckung eindeutig feststehen, welches Token geliefert wird.

Existieren zwei reguläre Ausdrücke für unterschiedlich lange Zeichenketten mit gleicher Anfangssequenz, so findet das Prinzip der maximalen Überdeckung seine Anwendung: Läßt sich sowohl einer Zeichenkette A der Länge n als auch einer Zeichenkette B der Länge $n + 1$, die aus A und dem konkatenierten nächsten Zeichen der Eingabe besteht, jeweils ein sie überdeckender regulärer Ausdruck zuordnen, so wird derjenige reguläre Ausdruck gewählt, der die längere Zeichenkette überdeckt.

Das Verfahren einer solchen Zuordnung von regulären Ausdrücken wird für manuell ausprogrammierte Scanner am Beispiel der Implementierung eines endlichen Automaten in den folgenden Abschnitten kurz erläutert. Die hierzu notwendige grundsätzliche Vorgehensweise wird in Anlehnung an [ASU88a] beschrieben.

Endliche Automaten

Endliche Automaten sind aus der theoretischen Informatik bekannt (siehe z. B. [HU90]) und werden hier nicht formal definiert. Sie lassen sich als sog. Übergangsgraphen darstellen, in denen festgelegt wird, welche Aktionen ausgeführt werden müssen, um eine Zeichenkette zeichenweise zu lesen und ihr einen regulären Ausdruck zuzuordnen. Übergangsgraphen sind markierte, gerichtete Graphen und bestehen aus einer Menge von Zuständen und Kanten, die die Übergangsfunktionen des endlichen Automaten symbolisieren. Zusätzlich sind der Startzustand und die Endzustände besonders ausgezeichnet und die Menge der Eingabesymbole definiert.

Die Zustände werden durch Kreise symbolisiert. Jeder Zustand repräsentiert die Folge der bereits gelesenen Eingabezeichen. Die Zustände sind durch als Pfeile dargestellte Kanten verbunden, die Marken besitzen. Die Marke an einer Kante von Zustand s nach Zustand t gibt dabei an, welches Eingabezeichen zulässig ist, nachdem im Übergangsgraph der Zustand s erreicht wurde. Stimmt das nächste Eingabezeichen mit dieser Marke überein, so wird zum Zustand t gewechselt. Ein Übergangsgraph heißt *nichtdeterministisch*, wenn er Zustände besitzt, von denen aus mehrere Kanten mit derselben Marke zu verschiedenen Zuständen führen, oder er Übergänge bei leerem Zeichen (sogenannte ε-Übergänge) besitzt. Sonst heißt er *deterministisch*.

Derjenige Zustand des Übergangsgraphen, in dem sich die Kontrolle am Anfang des Erkennungsprozesses für einen regulären Ausdruck befindet, wird als Startzustand bezeichnet. Der Übergang zu einem anderen Zustand geschieht nach dem Lesen des nächsten Eingabezeichens. Stimmt dieses Zeichen mit der Marke an einer aus dem aktuellen Zustand herausführenden Kante überein, so wird in den durch diese Kante festgelegten Zustand gewechselt. Existiert keine solche Marke, so ist der Erkennungsprozeß gescheitert und ein lexikalischer Fehler liegt vor. Endzustände sind Zustände, aus denen keine Kante herausführt. Ist ein solcher Zustand erreicht worden, muß die aktuelle Zeichenkette akzeptiert werden, und das Token für den sie überdeckenden regulären Ausdruck wird an den Parser geliefert.

Die Abbildung 5.2 enthält ein Beispiel für einen deterministischen Übergangsgraphen mit Startzustand 0. Zur besseren Übersichtlichkeit wurden teilweise mehrere Kanten zu einer zusammengefaßt sowie komplementäre Marken verwendet. Der dargestellte reguläre Ausdruck überdeckt eine Zeichenkette für eine Zahl (syntaktische Variable *number*, vgl. Abbildung 3.6 auf Seite 25).

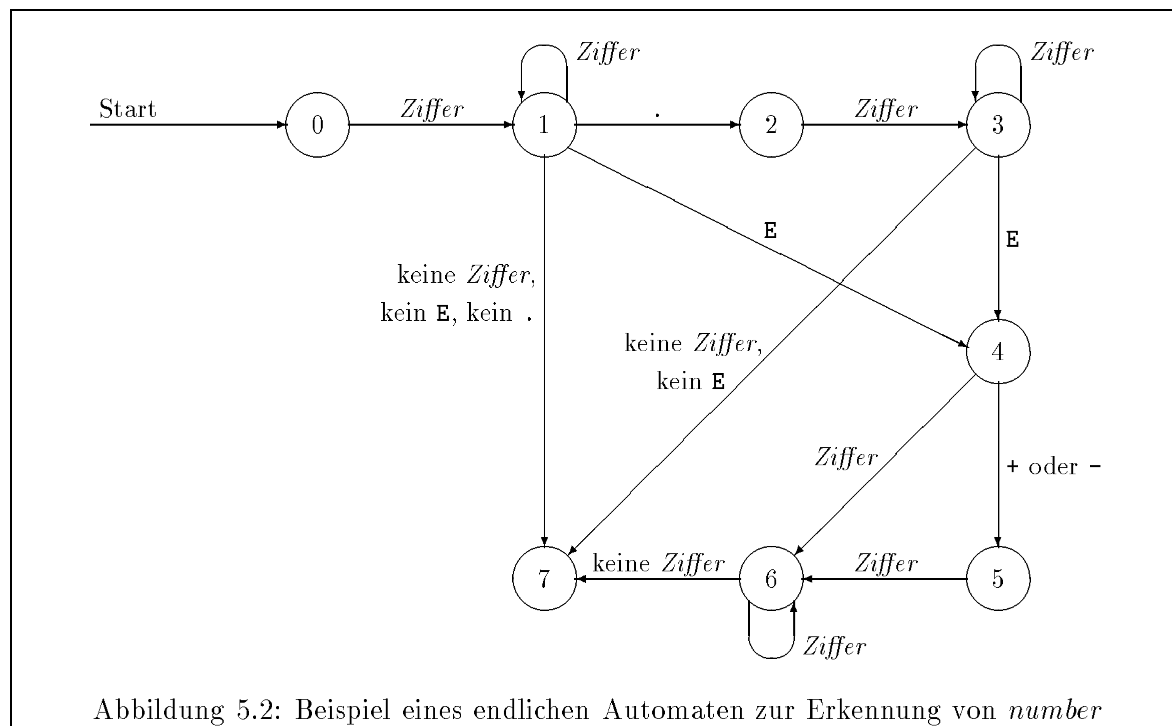


Abbildung 5.2: Beispiel eines endlichen Automaten zur Erkennung von *number*

Implementierung

Vor der Implementierung eines Scanners muß für jeden regulären Ausdruck ein Übergangsgraph erzeugt werden. Hierzu existiert eine Vielzahl von Strategien, die sich hauptsächlich darin unterscheiden, welchen Platz- und Zeitbedarf die jeweils erzeugten Automaten haben. Grundsätzlich gilt, daß deterministische endliche Automaten eine geringere Laufzeit benötigen, aber unter Umständen erheblich mehr Platz. Dagegen haben nichtdeterministische endliche Automaten einen geringeren Platzbedarf, aber schlechteres Laufzeitverhalten. Bekannt sind weiterhin Optimierungsmechanismen zur Minimierung der Zustandsmenge (vgl. [Kas90]).

Sind die optimierten Automaten erstellt, können zwei Implementierungstechniken angewandt werden. Beim ersten Verfahren wird ein sog. tabellengesteuerter Automat implementiert. Dazu wird die Übergangsfunktion des endlichen Automaten in einer geeigneten Matrix repräsentiert und in der Erkennungsphase interpretiert. Der Vorteil ist, daß bei Änderungen der lexikalischen Spezifikation nur diese Matrix neu erstellt werden muß, nicht aber der Matrixinterpretierer. Die Nachteile sind der hohe Platzbedarf der Matrix und der verhältnismäßig langsame Zugriff auf ihre Einträge. Um diese Nachteile zu vermeiden, findet das zweite Verfahren der Implementation – die sogenannte direkte Programmierung – Anwendung. Hierbei werden die Zustände und Kanten des Automaten systematisch in Programmfragmente umgesetzt, in denen die Zustände durch Programmstellen repräsentiert und die Übergänge durch bedingte Verzweigungen und Schleifen realisiert werden.

Im allgemeinen wird ein Endzustand erst erreicht, nachdem das erste Zeichen gelesen wurde, das nicht mehr zum erkannten Token gehört. Dieses sogenannte Lookahead-Zeichen muß in den Eingabepuffer zurückgeschrieben werden, damit es für den nächsten Erkennungsprozeß wieder zur Verfügung steht.

Zusammenfassung

Für kleinere Sprachdefinitionen kann die manuelle Ausprogrammierung des Scanners sinnvoll sein, weil nicht die Benutzung eines Werkzeugs erlernt werden muß und sich die Komplexität der Übergangsgraphen in Grenzen hält. Bei größeren Projekten ist diese Methode jedoch eher ungeeignet. Alle Übergangsgraphen müssen zu ihrer Implementierung deterministisch sein, denn aus Effizienzgründen sollten Backtracking-Verfahren vermieden werden. Je größer die zu scannenden Quelltexte sind, desto wichtiger wird eine zusätzliche, meist aufwendige Optimierung des Scanners, denn der Scanner ist die einzige Komponente des Übersetzers, die jedes Zeichen der Eingabe verarbeiten muß.

Erkennbar wird auch die schlechte Wartbarkeit einer manuellen Implementierung. Die bei einer Spezifikationsänderung erforderlichen Anpassungen bzw. Neuerstellungen der Übergangsgraphen und ihrer korrespondierenden Codefragmente sind i. a. umfangreich. Bereits vorgenommene Optimierungen sind oft hinfällig, wenn sie besondere Merkmale der alten Spezifikation ausnutzten. Am robustesten ist der Realisationsansatz über einen tabelleninterpretierenden Automaten; daher wird dieser Weg auch von einigen Scannergeneratoren beschritten.

Als Vorteil läßt sich feststellen, daß sorgfältig optimierte manuelle Implementationen des laufzeitkritischen Scanners i. a. am effizientesten sind. Da aber im ^{art}*deco*-Projekt gerade im Hinblick auf die externe Wissensrepräsentation eine gute Wartbarkeit erwünscht ist, wird in diesem Kontext der Vorteil der besseren Effizienz als geringer bewertet. Weiterhin sind manuell ausprogrammierte Scanner schlechter auf andere Rechnersysteme portierbar. Selbst Implementationen in der Hochsprache C müssen in der Regel angepaßt werden. Dagegen verwenden die Generatorwerkzeuge auf allen Systemen die gleichen Spezifikationen, die Anpassungen sind also auf die vom Entwickler kodierten Aktionen (vgl. Abschnitt 6.1.2 ab Seite 68) begrenzt.

5.3.2 Der Parser

Die Eingabe des Parsers wird – wie bekannt – durch eine kontextfreie Grammtik spezifiziert. Während der Syntaxanalyse wird geprüft, ob ein bestehendes Wort von dieser Grammatik erzeugt werden kann. Dazu wird ein Parse-Baum (siehe Abschnitt 5.1.2.2) aufgebaut. Ist seine Erzeugung erfolgreich, so liegt kein syntaktischer Fehler vor, und die semantische Analyse kann erfolgen.

Grundsätzlich existieren zwei Strategien zum Aufbau des Parse-Baums: die Top-Down- und die Bottom-Up-Methode. Nach der Top-Down-Methode wird der Baum von der Wurzel in Richtung Blätter konstruiert, für die Bottom-Up-Methode gilt die Umkehrung. Da jedoch i. d. R. in einem optimierten Übersetzer der Parse-Baum nicht explizit aufgebaut wird, spricht man zur Widerspiegelung der prinzipiellen Vorgehensrichtung von *ziel*bezogenen (Richtung Blätter) bzw. *quell*bezogenen Parsern (Richtung Wurzel).

Nach [ASU88a] existiert zu jeder kontextfreien Grammatik ein Parser mit höchstens kubischem Zeitaufwand im Verhältnis zur Anzahl der Token. Dagegen können für bestimmte Grammatikklassen (LL- und LR-Klasse[2]) sogar Parser mit linearem Aufwand erzeugt werden; sie arbeiten die Tokenfolge mit einem Symbol Vorschau („Lookahead-Symbol") linear ab. Gerade die Grammatiken zur Spezifikation der in der Praxis vorkommenden programmiersprachlichen Konstrukte lassen sich i. d. R. recht einfach durch bestimmte Transformationsregeln spracherhaltend auf eine der erwähnten Grammatikklassen abbilden.

5.3.2.1 Zielbezogene Parser

Zielbezogene Parser lassen sich für LL-Grammatiken zum Beispiel nach der Methode des „rekursiven Abstiegs" manuell entwickeln. Dazu wird für jedes Nichtterminal X eine Funktion erstellt, die alle aus X erzeugbaren Symbolfolgen akzeptiert. Jede Funktion erhält für jede Produktion

$$X \longrightarrow x_1 \cdots x_n$$

der kontextfreien Grammatik eine Operationssequenz, in der für alle nichtterminalen x_i die entsprechenden Funktionen direkt oder indirekt rekursiv aufgerufen werden und alle terminalen x_j mit den aktuellen Vorschau-Token verglichen werden. Stimmt ein Terminal nicht mit dem aktuellen Lookahead überein, so kann es nicht akzeptiert werden und ein syntaktischer Fehler liegt vor.

Die Folge der während der Abarbeitung rekursiv aufgerufenen Funktionen definiert dabei implizit einen Parse-Baum für die Eingabe. Ist die Grammatik so konstruiert, daß es aufgrund eines einzigen Lookahead-Tokens möglich ist, eine der gegebenenfalls mehreren Produktionen für das Nichtterminal X eindeutig zu wählen, kann ineffizientes Backtracking vermieden werden. In diesem Fall handelt es sich um einen sogenannten *prädiktiven* Parser.

[2]Eine ausführliche Behandlung dieser Klassen ist in [ASU88a] und [Kas90] zu finden; die hier verwendete LALR(1)-Grammatik als Einschränkung der LR(1)-Grammatik wird in Abschnitt 5.4.2.2 näher erläutert.

5.3.2.2 Quellbezogene Parser

Für die Klasse der LR-Grammatiken werden quellbezogene Parser implementiert. Diese Parser basieren auf dem formalen Modell der sog. Kellerautomaten, die genau die kontextfreien Sprachen erkennen. Auch hier sei auf die Standardliteratur zur theoretischen Informatik verwiesen (z. B. [HU90]).

In Analogie zu den Ausführungen zur Implementation von endlichen Automaten in Scannern beruht die Realisierung von Kellerautomaten auf der Ermittlung seiner Zustände und Übergänge, die in einer Übergangstabelle gespeichert werden. Diese Übergangstabelle enthält i. a. redundante Informationen; ihr Umfang kann daher durch verschiedene Verfahren reduziert werden. Die eigentliche Aufgabe des Parsers ist nun die Interpretation der Übergangstabelle. Die Klasse der LR-Grammatiken ist mächtiger als die in Abschnitt 5.3.2.1 erwähnte Klasse der LL-Grammatiken; sie ist deshalb in den meisten Fällen besser zur Syntax-Spezifikation geeignet und vereinfacht notwendige Anpassungen der Ausgangsgrammatik. Die manuelle Implementierung der zugehörigen Kellerautomaten ist jedoch kaum praktikabel und wird in der Praxis mit Werkzeugen zur automatischen Erzeugung vorgenommen.

Zusammenfassung

Die bei der Diskussion der Scannerimplementierung angeführten Argumente besitzen auch für den Parser Gültigkeit: Eine manuelle Implementation kann mit hohem Programmieraufwand effizienter sein, ist aber sehr viel schlechter zu warten und i. d. R. schlechter portierbar. Sie birgt ein beträchtliches Fehlerrisiko bei der korrekten Umsetzung einer kontextfreien Grammatik in die Programmfragmente des Parsers; dagegen darf bei Parsergeneratoren davon ausgegangen werden, daß die automatische Umsetzung von ihren Entwicklern ausreichend getestet wurde und damit fehlerfrei ist.

Hinzu kommt, daß sowohl der Scanner als auch der Parser nach derselben Implementierungsart erstellt werden sollten, um Probleme im Zusammenspiel dieser Komponenten zu vermeiden. Aus diesen Gründen wurde auch bei der Implementierung des Parsers die Benutzung eines Generators vorgezogen.

5.4 Automatische Generierung

Die Entwicklung von Werkzeugen zur Unterstützung und Automatisierung der Implementierung einzelner Übersetzerkomponenten ist nach [Kas90] vor allem aufgrund von zwei Punkten gut möglich:

1. Die Verfahren zur Herstellung der Übersetzerkomponenten sind weitgehend systematisiert (vgl. Abschnitt 5.3).

2. Die Funktion der Komponenten kann auf der Grundlage formaler Methoden präzise spezifiziert werden. Für jede der benötigten Komponenten existieren geeignete formale Funktionsbeschreibungen.

Die automatische Erzeugung der einzelnen Übersetzerkomponenten ist allerdings nur *ein* Schritt auf dem Weg zu einem funktionsfähigen Programm. Zusätzlich müssen die Einzelteile mit passenden Schnittstellen zusammengesetzt und durch ergänzende Module zum Übersetzer vervollständigt und konfiguriert werden. Da die letztgenannten Aufgaben nur von den verwendeten Werkzeugen und der Übersetzerstruktur abhängen, lassen sie sich ebenfalls automatisieren. Hierbei handelt es sich um sog. Übersetzerentwicklungsumgebungen.

Die Entwicklungsumgebungen werden am Beispiel „Eli" in Abschnitt 5.4.1 kurz erläutert und stützen sich auf die Ausführungen von [Wai94] und [GHL+92]. Die für diese Diplomarbeit wichtigen Einzelwerkzeuge Lex und Yacc zur Implementieung von Scanner und Parser werden in Abschnitt 5.4.2 behandelt.

5.4.1 Übersetzerentwicklungsumgebungen

Das in Kooperation der Arbeitsgruppen von Prof. Dr. U. Kastens an der Universität-Gesamthochschule Paderborn und Prof. Dr. W. M. Waite an der University of Colorado entwickelte integrierte Werkzeugsystem „Eli" ist ein Vertreter der Entwicklungsumgebungen. Es enthält alle zur Erzeugung eines Übersetzers notwendigen Werkzeuge, dessen Schnittstellen aufeinander abgestimmt sind. Als Ergebnis generiert „Eli" einen vollständigen Satz von C-Modulen einschließlich Makefile und einem ausführbaren Programm.

Als Eingabe benötigt „Eli" die Spezifikationen für die Einzelwerkzeuge. Hierzu existieren mehrere Beschreibungssprachen, mit denen z. B. die strukturelle Analyse des Quelltextes, der Parse-Baum (d. h. sein Aufbau und seine Abarbeitung) oder die Erzeugung des Zielcodes festgelegt werden können. Ein integriertes Expertensystem verwaltet diese Spezifikationen und regelt während des Entwicklungsprozesses die Aufrufe der entsprechenden Werkzeuge sowie ihre Abhängigkeiten und Datenflüsse.

Um die Benutzereingaben zu reduzieren, sind vorgefertigte Lösungen und Bibliotheken mit wiederverwendbaren Spezifikationen für häufig auftretende Teilaufgaben der Sprachimplementierung im System vorhanden. In anderen Fällen können vorhandene Spezifikationen und sogar die vom System verwendeten Werkzeuge an die eigenen Anforderungen angepaßt oder ausgetauscht werden.

Die Vorteile einer Entwicklungsumgebung liegen besonders in der guten Abstimmung der einzelnen Module und ihrer dadurch relativ hohen Effizienz. Spezialisierte Werkzeuge oder manuelle Implementierungen können zwar noch effizienter sein, benötigen aber einen wesentlich höheren Programmierungsaufwand. Ein weiterer Gesichtspunkt ist das breite Anwendungsfeld einer integrierten Umgebung, das nicht auf die Erzeugung von Übersetzern beschränkt sein muß.

Das letzte Argument ist jedoch in bezug auf das ^{art}*deco*-Projekt bedeutungslos. [Wai94] führt aus, daß bei einem „typischen Compiler" der Anteil des Scannens und Parsens nur etwa 15 Prozent des gesamten Übersetzungsprozesses betrage. Die restlichen 85 Prozent seien daher nicht mit Standard-Werkzeugen wie Lex und Yacc, sondern nur mit integrierten Entwicklungsumgebungen zu bewältigen. Doch der Übersetzer für die Komponentenbeschreibungssprache ist kein „typischer Compiler", denn sein Schwerpunkt liegt in der lexikalischen und syntaktischen Analyse (vgl. Abschnitt 5.1.3.3).

Trotz eines Online-Hilfesystems und der Bereitstellung universeller Prototypen wirkt sich nachteilig aus, daß „Eli“-spezifische Sprachen zur Funktionsbeschreibung der Übersetzerkomponenten beherrscht und angewandt werden müssen. Dagegen erfordert der Einsatz von Lex und Yacc die Kenntnis von zwei Formalismen, die in zahlreichen Veröffentlichungen beschrieben werden. Ausschlaggebend gegen die Verwendung einer Übersetzerentwicklungsumgebung wie „Eli“ ist jedoch ihre geringere Verbreitung. Im Gegensatz zu „Eli“ sind die Standardwerkzeuge Lex und Yacc bzw. ihre GNU-Äquivalente Flex und Bison praktisch auf jedem UNIX-System und sogar unter MS-DOS verfügbar.

5.4.2 Lex und Yacc

Die beiden Werkzeuge Lex und Yacc sind aufeinander abgestimmt und verwenden eine einheitliche Schnittstelle. Lex erzeugt für gegebene reguläre Ausdrücke einen Erkenner auf der Basis eines endlichen Automaten. Die Eingabe für Yacc besteht aus einer kontextfreien Grammatik der LALR(1)-Klasse[3], die Ausgabe ist ein tabellengesteuerter Kellerautomat.

Für beide Werkzeuge muß die Eingabe in ihrer jeweils spezifischen Syntax formuliert werden. Neben diesen erforderlichen Spezifikationen kann der Entwickler in C programmierte Aktionen definieren. Eine typische Aktion im Scanner wäre z. B. die Eintragung des aktuellen Bezeichners in die Symboltabelle; innerhalb des Parsers können z. B. die semantischen Regeln als Aktionen implementiert werden.

Die Ausgabe besteht aus C-Quelltexten, die mit entsprechenden Compilern in ein ablauffähiges Programm übersetzt werden können. Auf diese Weise ist ein höheres Maß an Portierbarkeit erreicht, denn vorhandene Lex- bzw. Yacc-Spezifikationen müssen für andere Rechner oder Betriebssysteme nicht angepaßt werden. Damit auch die vom Entwickler definierten Aktionen weitestgehend portierbar sind, werden sie in derselben Zielsprache – also C – kodiert. Die Werkzeuge kopieren sie dann direkt in die C-Ausgabe hinein und vermeiden auf diese Weise langsamere Funktionsaufrufe.

Da bei der Implementation des ^art^*deco*-Übersetzers Lex und Yacc verwendet wurden, wird auf ihre Funktionsweise im folgenden näher eingegangen.

5.4.2.1 Zusammenwirken der Funktionen

Innerhalb eines mit Lex und Yacc generierten Übersetzers sind einige Funktionen besonders hervorzuheben, weil sie sein eigentliches Gerüst darstellen. Hierbei handelt es sich sowohl um automatisch als auch um manuell erstellte C-Funktionen. Diese Komponenten sind:

1. `main()`: Jedes C-Programm beginnt seine Ausführung in der Funktion `main()`. Sie wird vom Entwickler manuell erstellt.

2. `yylex()`: Diese Funktion wird mit dem Scannergenerator Lex automatisch erzeugt und enthält die Implementation des endlichen Automaten für die lexikalische Analyse.

[3] LALR(1) bedeutet Look-Ahead, Left-to-right scanning of the input, Rightmost derivation, 1 look ahead token; eine Definition erfolgt in Abschnitt 5.4.2.2.

3. `yyparse()`: Der Parsergenerator Yacc erzeugt diese Funktion automatisch; sie enthält einen tabellengesteuerten Kellerautomaten und nimmt die syntaktische Analyse vor. Die Aspekte der semantischen Analyse muß der Entwickler manuell programmieren, werden aber von Yacc automatisch integriert.

4. `yyerror()`: Diese Funktion dient der Fehlerbehandlung von syntaktischen Fehlern. Sie kann automatisch generiert werden, gibt dann aber nur die Meldung `Syntax error` aus. Eine manuelle Implementation ist möglich und sinnvoll, um dezidiertere Ausgaben zu erzeugen, die dem Anwender des Übersetzers die Auffindung und Beseitigung der Fehlerquelle erleichtern.

Das Zusammenwirken dieser Programmteile ist bei jedem mit Lex und Yacc erzeugten Übersetzer gleich. Am Anfang wird in der Funktion `main()` direkt oder indirekt die Parserfunktion `yyparse()` aufgerufen. Der Aufbau des Parse-Baums erfordert als Eingabe die vom Scanner gelieferte Tokenfolge. Hierzu wird die Scannerfunktion `yylex()` beauftragt, das nächste Token (zunächst also das erste) zu ermitteln. `yylex()` liest nun Zeichen aus der Standardeingabequelle `stdin` (bzw. nach erfolgter Umlenkung aus einer Datei), bis die aktuelle Zeichenkette von einem regulären Ausdruck überdeckt wird und gibt das zugehörige Token an die Parserfunktion zurück. Nach der Verarbeitung dieses Tokens durch die Parserfunktion fordert sie erneut das nächste Token bei der Scannerfunktion an, u. s. w.

Der geschilderte Kreislauf wird unterbrochen, falls von `yylex()` das Ende der Eingabe gelesen wird, oder wenn ein Fehler auftritt. Danach kehrt die Kontrolle zu `main()` zurück, die den Rückgabewert von `yyparse()` (0 bei erfolgreichem Parsen der Eingabe, 1 im Fehlerfall) verarbeiten kann. Tritt ein Fehler auf, dann ruft die Parserfunktion vor seiner Beendigung die Funktion `yyerror()` auf. So kann eine geeignete Fehlermeldung erzeugt werden, ehe die Verarbeitung terminiert. Die Abbildung 5.3 verdeutlicht in Anlehnung an [Her92] diese Zusammenhänge graphisch.

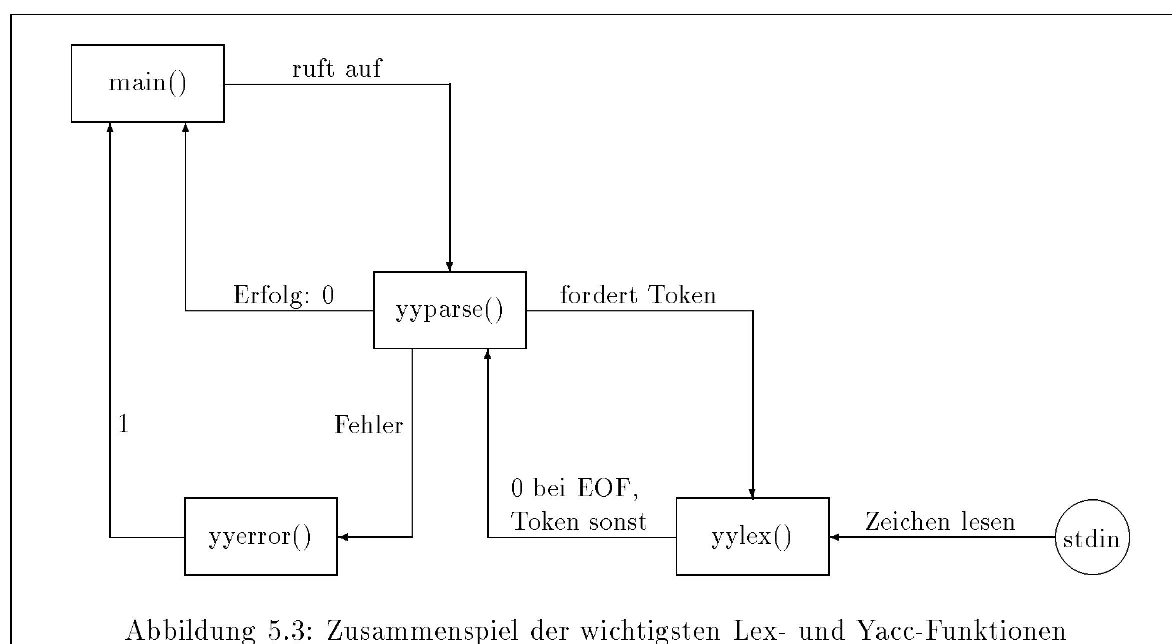


Abbildung 5.3: Zusammenspiel der wichtigsten Lex- und Yacc-Funktionen

5.4.2.2 Arbeitsweise eines LR-Kellerautomaten

Die Konstruktion einer kontextfreien LALR-Grammatik erfordert das Verständnis der Arbeitsweise eines sie erkennenden Kellerautomaten. Zwar ist die LALR-Bedingung weniger restriktiv[4] als z. B. die LL-Bedingung, dennoch müssen i. d. R. einige Konstrukte der Ausgangsgrammatik so transformiert werden, daß keine Konflikte auftreten. Aus diesem Grund wird im folgenden in Anlehnung an [Kas90] die Klasse der LALR(1)-Grammatiken definiert und die generelle Vorgehensweise eines entsprechenden Kellerautomaten beschrieben.

Zunächst erfolgt die Definition der Klasse der LR(k)-Grammatiken als Grundlage quellbezogener Parser:

> Eine kontextfreie Grammatik $G = (T, N, P, S)$ ist eine LR(k)-Grammatik, wenn für je zwei Rechtsableitungen
>
> $$S \stackrel{*}{\Longrightarrow} uAw \Longrightarrow uxw$$
> $$S \stackrel{*}{\Longrightarrow} u'Bw' \Longrightarrow u'x'w' = uxv$$
>
> gilt:
>
> $$\text{aus} \quad \text{Anf}_k(w) = \text{Anf}_k(v) \quad \text{folgt} \quad u = u',\ A = B \text{ und } x = x'$$
>
> wobei $u, u', x, x' \in \mathcal{P}(N \cup T)$ und $w, w', v \in \mathcal{P}(T)$.

Erklärung der Symbole:

- $\Longrightarrow$: Ein Rechtsableitungsschritt, d. h. das am weitesten rechts stehende Nichtterminal wird ersetzt. Der Stern ($\stackrel{*}{\Longrightarrow}$) bezeichnet eine beliebige Zahl von wiederholten Ableitungsschritten.
- $\text{Anf}_k(w)$ mit $w \in \mathcal{P}(T)$: Die ersten k Terminale von w, falls $|w| > k$, bzw. w sonst (d. h. die ersten k Symbole der Folge w werden bestimmt).
- $\mathcal{P}(M)$ mit M als Menge: Die Potenzmenge von M.

Die obige Definition besagt also, daß in einer Situation mit akzeptiertem ux und k Symbolen als Vorschau („Lookahead") von w der nächste Schritt der Rechtsableitung eindeutig bestimmt ist, falls die Grammatik der LR(k)-Bedingung genügt. Im folgenden werden nur noch Lookaheads mit einem Symbol Vorschau betrachtet, da Grammatiken mit $k > 1$ in der Praxis des Übersetzerbaus nicht verwendet werden.

Jeder Zustand des endlichen Kellerautomaten zur Erkennung von LR-Grammatiken ist charakterisiert durch eine Menge von Situationen (s. u.), deren Produktionen parallel verfolgt werden. Dieses ist notwendig, weil bei der quellbezogenen Analyse über die Anwendung einer Produktion erst dann entschieden werden kann, wenn ihre rechte Seite vollständig akzeptiert wurde.

Eine Situation kann formal als

$$[A \longrightarrow u \bullet v,\ R] \quad \text{mit } u, v \in \mathcal{P}(N \cup T) \text{ und } R \in \mathcal{P}(T)$$

[4] Weniger Restriktionen implizieren eine größere Anzahl von mit der Grammatik erzeugbaren Wörtern, die Grammatik ist somit mächtiger.

beschrieben werden, wobei $A \longrightarrow uv \in P$ eine Produktion der Grammatik G ist. Der Punkt ($\bullet$) gibt an, daß in dieser Produktion u bereits vollständig abgeleitet ist, v aber noch nicht. R ist der erwartete Rechtskontext und enthält eine Menge von Terminalen höchstens der Länge k (= 1 in diesem Fall, s. o.). Nach der vollständigen Abarbeitung der Produktion – der Punkt steht also am Ende – muß der Rest der Eingabe mit einem Element aus R beginnen, sonst kann der bereits gelesene, aber noch nicht reduzierte Teil der Eingabe nicht auf das Nichtterminal A reduziert werden.

Der durch einen geeigneten Algorithmus (siehe z. B. [ASU88a]) erstellte LR-Kellerautomat hat die folgenden Eigenschaften:

- Zustände:
  1. Anfangszustand:
     Er enthält u. a. die Situation $[S \longrightarrow \bullet w,\ \text{EOF}]$, wobei S das Startsymbol und $\text{EOF} \notin N \cup T$ ein sonst nicht vorkommendes Schlußzeichen ist.
  2. beliebiger Zustand q:
     Sei in q eine Situation $[A \longrightarrow u \bullet Xv,\ R]$ mit $X \in N \cup T$ und seien $X \longrightarrow w_1$, $\ldots$, $X \longrightarrow w_n \in P$ alle Produktionen für X, dann enthält q auch die Situationen $[X \longrightarrow \bullet w_1,\ R'], \ldots, [X \longrightarrow \bullet w_n,\ R']$ mit $R' = \text{Anf}_k(vR)$. Die Erzeugung dieser Situationen heißt „Hüllenbildung“.

- Übergänge:
  Für jedes Symbol $x \in N \cup T$, zu dem es in einem Zustand q eine Situation $[A \longrightarrow u \bullet xv,\ R]$ gibt, existiert ein Übergang in einen Zustand q' unter x, der die zu diesem Übergang korrespondierende Situation $[A \longrightarrow ux \bullet v,\ R]$ enthält.

- Konfliktfreiheit:
  Bei der Erstellung des Automaten können zwei Arten von Konflikten auftreten. Ein Konflikt zeigt an, daß die betrachtete Grammatik nicht die erforderliche LR-Bedingung erfüllt. Sie muß dann solange transformiert werden, bis Konfliktfreiheit hergestellt ist.
  1. Reduce-reduce-Konflikt:
     Enthalte ein Zustand zwei Situationen der Form
     $$[A \longrightarrow u\bullet,\ R_1] \text{ und } [B \longrightarrow v\bullet,\ R_2] \text{ mit } A \neq B \text{ oder } u \neq v.$$
     Ist das nächste Symbol der Eingabe $t \in R_1 \cap R_2$, dann kann nicht entschieden werden, nach welcher Produktion zu reduzieren ist.
  2. Shift-reduce-Konflikt:
     Enthalte ein Zustand zwei Situationen der Form
     $$[A \longrightarrow u\bullet,\ R_1] \text{ und } [B \longrightarrow v \bullet t,\ R_2] \text{ mit } t \in T$$
     und gelte $t \in \text{Anf}_k(R_1)$, dann kann nicht entschieden werden, ob ein Zeichen zu lesen oder nach der erstgenannten Produktion zu reduzieren ist.

Bei praktischen Anwendungen ist die Anzahl der Zustände i. d. R. bereits im LR(1)-Fall sehr groß. Effizienter sind daher weniger mächtige Grammatikklassen mit geringeren Zustandszahlen. Häufig wird die engere LALR(1)-Klasse verwendet, deren Mächtigkeit dennoch zur Beschreibung der üblichen programmierspachlichen Konstrukte ausreicht.

LALR(1)-Automaten basieren auf der Zustandsmenge, die bei der Konstruktion eines LR(0)-Automaten entsteht. Durch die fehlenden Rechtskontexte in den Situationen werden diejenigen Zustände eines LR(1)-Automaten, deren Situationen sich nur in den Rechtskontexten unterscheiden, auf einen einzigen Zustand abgebildet. Rechtskontexte werden jedoch wie oben beschrieben zur Vermeidung von Konflikten bei der Entscheidung über Reduzierungen herangezogen. Ein LR(0)-Automat ist daher nur dann konfliktfrei, wenn seine Zustände entweder *keine* Reduktionssituation oder ausschließlich *eine* Reduktionssituation enthalten. Hierdurch wird erkennbar, daß die Menge der LR(0)-Sprachen im Vergleich zur Menge der LR(1)-Sprachen kleiner ist. Um ihre Mächtigkeit nicht zu sehr einzuschränken, ordnet man den LR(0)-Situationen die Rechtskontexte zu, wie sie sich im LR(1)-Fall ergeben würden. Auf diese Weise stehen dem so konstruierten LALR(1)-Automaten bei Reduktionen präzisere Entscheidungsinformationen zur Verfügung.

Während der Analysephase kann ein Yacc-generierter LALR(1)-Kellerautomat vier verschiedene Operationen vornehmen: Shift, Reduce, Accept und Error. Seine generelle Arbeitsweise besteht dabei in der wiederholten Ausführung der folgenden Punkte:

1. Entscheidung über die nächste Operation:
   Abhängig vom aktuellen Zustand entscheidet der Automat zunächst, ob ein neues Lookahead-Token benötigt wird. Wenn ja, wird zur Ermittlung des nächsten Symbols die Scannerfunktion `yylex()` aufgerufen. Der aktuelle Zustand und das aktuelle Lookahead-Symbol sind bestimmend für die auszuführende Operation.

2. Ausführung der nächsten Operation:
   Die in 1. ermittelte Operation wird ausgeführt. Handelt es sich um Accept oder Error, stoppt die Verarbeitung. Sonst wird ein Zustandswechsel ausgeführt. Bei Shift wird ein neuer Zustand auf dem Stack abgelegt, bei Reduce werden Zustände vom Stack entfernt.
   Sei q der aktuelle Zustand. Dann bewirken die Operationen im einzelnen:

   - Shift eines Symbols $v \in N \cup T$:
     Voraussetzung: q enthält eine Situation der Form $[A \longrightarrow u \bullet v,\ R]$.
     Dann wird q gekellert, der neue aktuelle Zustand ist der Nachfolger von q unter v. Ist v ein Terminal, wird das in 1. gelesene Lookahead akzeptiert und im nächsten Schritt muß ein neues gelesen werden. Ist v ein Nichtterminal, bleibt das aktuelle Lookahead-Symbol weiter gültig.

   - Reduce einer Produktion $p \in P$:
     Voraussetzung: q enthält eine Situation der Form $[A \longrightarrow uv\bullet,\ R]$ und das aktuelle Lookahead-Symbol ist Element von R.
     Dann werden soviele Zustände entkellert, wie die rechte Produktionsseite Zeichen enthält. Vom dann obersten Kellerzustand wird ein Übergang mit A zum neuen aktuellen Zustand ausgeführt (Shift mit Nichtterminal).

   - Accept:
     Voraussetzung: q enthält eine Situation der Form $[S \longrightarrow uv\bullet,\ \text{EOF}]$ und das aktuelle Lookahead-Symbol ist das Eingabeende-Zeichen EOF.
     Dann wird ein Reduce mit dem Startssymbol S ausgeführt und die (fehlerfreie) Verarbeitung stoppt.

- Error:
  Voraussetzung: Keine der oben angeführten Operationen kann ausgeführt werden. Dann wird die Fehlerfunktion `yyerror()` aufgerufen und die Syntaxanalyse beendet.

Kapitel 6

Implementation des Übersetzers

Dieses Kapitel behandelt die wichtigsten Aspekte der Implementierung des $^{\text{art}}$*deco*-Übersetzers. Die lexikalische Analyse wurde mit dem Scannergenerator Lex realisiert, die syntaktische und die semantische Analyse mit dem Parsergenerator Yacc. Die Ausführungen dieses Kapitels beziehen sich dabei auf die prinzipielle Vorgehensweise und nicht auf jede Detaillösung. Daher sind weitergehende Informationen direkt dem Quelltext zu entnehmen. Als Literatur wurde im wesentlichen [Her92] und [DS92] herangezogen.

6.1 Implementation des $^{\text{art}}$*deco*-Scanners

Jede Scannerspezifikation als Eingabe für Lex besteht aus drei Teilen, von denen jedoch nur der zweite obligatorisch ist. Die drei Teile werden syntaktisch durch zwei aufeinanderfolgende Prozentzeichen getrennt und in den folgenden Abschnitten erläutert.

1. Deklarationen und Definitionen, optional
%%
2. Lex-spezifische Übersetzungsregeln
%%
3. benutzerdefinierte Hilfsfunktionen, optional

6.1.1 Deklarations- und Definitionsteil

In diesem Teil der Scannerspezifikation werden die verwendeten C-Variablen, -Funktionen und -Strukturen deklariert sowie C-Konstanten und die sogenannten *regulären Definitionen* definiert. Reguläre Defintionen sind mit einem eindeutigen Namen versehene reguläre Ausdrücke. Sie werden im Regelteil wie Makros für mehrfach auftretende reguläre Ausdrücke verwendet und vermeiden unnötige Wiederholungen. Hierdurch verbessert sich die Lesbarkeit und Wartbarkeit der Spezifikation.

Die wichtigsten Konstanten sind die ganzzahligen Repräsentationen der Token. Sie können manuell definiert oder vom Parsergenerator Yacc aus der kontextfreien Grammatik ermittelt

werden. In jedem Fall ist jedoch zu beachten, daß diese Definitionen ein Teil der Schnittstelle zwischen Scanner und Parser sind, da sie ihre Übergabewerte festlegen. Die im $^{\text{art}}$*deco*-Übersetzer verwendeten Token sind in Anhang A.1 aufgelistet. Die Funktionen und Strukturen beziehen sich im wesentlichen auf die vom Entwickler definierten Hilfsfunktionen. Durch diese Prototypen kann innerhalb der Aktionen im Regelteil auf die Hilfsfunktionen zugegriffen werden. Reguläre Definitionen werden im $^{\text{art}}$*deco*-Scanner für die literalen Konstanten *digit*, *number*, *symbol* und *string*, sowie den einfachen Bezeichner *simple-id* und das Differential *differential* definiert. Ihr Aufbau in Lex-Syntax ist dem Quelltext der Implementation zu entnehmen.

6.1.2 Übersetzungsregeln

Die Übersetzungsregeln definieren das Verhalten des Scanners und werden von Lex als endlicher Automat in der Funktion `yylex()` kodiert. Im folgenden wird zunächst ihr genereller Aufbau beschrieben, bevor auf die im $^{\text{art}}$*deco*-Scanner verwendeten Regeln eingegangen wird.

6.1.2.1 Aufbau der Regeln

Im Regelteil wird die Zuordnung der Token zu den regulären Ausdrücken mit einer Folge von Regeln beschrieben. Jede Übersetzungsregel hat dabei die Form

$r \qquad \{\ a\ \}$

r ist ein regulärer Ausdruck in einer Lex-spezifischen Notation, deren Erläuterung z. B. in [Her92] zu finden ist. Die in geschweiften Klammern angegebene Aktion a ist ein C-Programmstück, das ausgeführt wird, wenn sich im Eingabetext eine Zeichenkette befindet, die durch den regulären Ausdruck r überdeckt wird.

Um Mehrdeutigkeiten bezüglich der Auswahl einer Aktion zu vermeiden, geht der Lex-Scanner nach der folgenden Hierarchie vor:

1. Wird in der Eingabe eine Zeichenkette gefunden, die durch keinen regulären Ausdruck r_i überdeckt wird, so wird diese unverändert an die Standardausgabe weitergegeben.

2. Anwendung des Prinzips der maximalen Überdeckung, also Ermittlung der längsten durch einen regulären Ausdruck überdeckten Zeichenkette und Ausführung der zugehörigen Aktion.

3. Wird eine Zeichenkette durch zwei reguläre Ausdrücke maximal überdeckt, wird die textuell zuerst angegebene Aktion ausgeführt.

Im allgemeinen liefert die ausgewählte Aktion als Abschluß ein Token an den aufrufenden Parser zurück (z. B. mit `return` *Token*) und die Kontrolle geht dadurch wieder an den Parser über. Ist dieses jedoch nicht der Fall, liest der Scanner die Eingabe weiter und sucht nach der nächsten durch einen regulären Ausdruck überdeckten Zeichenkette. Auf diese Weise ist es möglich, zum Beispiel Füllzeichen herauszufiltern.

6.1.2.2 Regeln im ${}^{\text{art}}$deco-Scanner

Die zur Spezifikation des ${}^{\text{art}}$deco-Scanners verwendeten Regeln lassen sich in drei Kategorien unterteilen, die in den nachfolgenden Abschnitten genauer beschrieben werden. Die Regeln unterscheiden sich dabei in der Art ihres Aktionsteils:

1. Aktionsteil ohne Rückgabewert,
2. Aktionsteil mit Token als Rückgabewert,
3. Aktionsteil mit Token als Rückgabewert und Attributwertbestimmung.

Kein Rückgabewert

Mit einem Aktionsteil ohne Rückgabewert ist es möglich, bestimmte lexikalische Elemente herauszufiltern und somit die weiteren Übersetzungsphasen von bedeutungslosen Informationen zu entlasten. Bei dem hier beschriebenen Scanner handelt es sich um die Unterdrückung der Füllzeichen und Kommentare.

Die Füllzeichen sind das Leerzeichen, das Tabulatorzeichen (Lex-Notation `\t`) und das Zeilenendezeichen (`\n`), die beliebig oft aufeinander folgen dürfen. Um eine ständige Orientierung über die aktuell zu analysierende Quelltextposition zu haben (anwendbar z. B. bei Fehlerausgaben) wird nach jedem Zeilenwechsel die globale Ganzzahl-Variable `lineno` inkrementiert. Desweiteren werden Kommentare herausgefiltert. Sie werden durch ein Prozentzeichen eingeleitet und enden mit dem ersten folgenden Zeilenwechsel.

Die Abbildung 6.1 gibt die Regeln mit Aktionen ohne Rückgabewert im ${}^{\text{art}}$deco-Scanner an. Die zwei aufeinander folgenden geschweiften Klammern (`{}`) stellen leere Aktionen dar. Die eckigen Klammern im Teil der regulären Ausdrücke sind die Lex-spezifische Notation für Zeichenklassen; die in den Klammern angegebenen Zeichen decken genau ein Zeichen ab, wenn dieses hier enthalten ist. Der letzte reguläre Ausdruck definiert einen Kommentar. Damit das Prozentzeichen nicht als Metazeichen interpretiert wird, muß ein Backslash vorangestellt werden. Der Punkt deckt ein beliebiges Zeichen außer dem Zeilenendezeichen ab, der Stern bzw. das Pluszeichen haben dieselbe Bedeutung wie in Abschnitt 3.1.1 definiert.

```
[ \t]+  {}
\n      { lineno++; }
\%.*    {}
```

Abbildung 6.1: Regeln mit Aktionen ohne Rückgabewert

Token als Rückgabewert

Die in diesem Abschnitt besprochenen Regelaktionen bestehen nur aus der Übergabe eines Tokens an den aufrufenden Parser. Bei den hierfür definierten Token handelt es sich um ganz-

zahlige Werte[1], die für jedes Schlüsselwort und für einige aus mehreren Zeichen bestehenden Operatoren eindeutig festgelegt sind.

Die Eingabedatei des Scanners enthält neben der Verhaltensbeschreibung auch die Strukturbeschreibung einer Komponente in $^{\text{art}}$*deco* (siehe Abschnitt 6.1.3.1). Die Strukturinformationen werden vom Übersetzer für den Aufbau einer Symboltabelle benötigt und müssen vor der eigentlichen Übersetzungsphase zur Verfügung stehen. Sie sind daher den Verhaltensbeschreibungen vorangestellt. Weil ein Lex-generierter Scanner jedoch die Eingabedatei vom Anfang an liest, muß der Einstiegspunkt für die Verhaltensbeschreibungen ermittelt werden. Die vom Scanner nicht verwertbaren Strukturinformationen können dann überlesen werden. Hierzu wird in der Funktion zum Aufbau der Symboltabelle (`InitSymbolTab()`, siehe ebenfalls Abschnitt 6.1.3.1) die erste Zeile der Verhaltensbeschreibung in der globalen Ganzzahl-Variable `start_lineno` gespeichert. Der Scanner prüft nun in jeder Aktion, die ein Token liefert, ob die aktuelle Zeilennummer `lineno` größer oder gleich `start_lineno` ist. Wenn ja, befindet er sich im richtigen Teil der Eingabedatei und kann die erforderliche Aktion ausführen; wenn nein, wird die Eingabe weiter gelesen.

Einige Beispiele in der Abbildung 6.2 zeigen die Implementation des oben geschilderten Sachverhalts. Wichtig ist die textuelle Anordnung der abgebildeten Regeln. Die Aktion der letzten Regel wird nur dann ausgeführt, wenn kein vorheriger regulärer Ausdruck die aktuelle Zeichenkette überdeckt. Diese „Defaultregel“ muß daher am Ende der Regelfolge stehen. Hier kann es sich bei der aktuellen Zeichenkette nur um ein einzelnes Zeichen handeln; dessen ASCII-Wert wird als Token an den Parser geliefert. Im allgemeinen ist dieses Zeichen ein Interpunktionszeichen wie : oder ; und wird in der Lex-eigenen Variable `yytext[0]` gespeichert. Handelt es sich jedoch um ein vom Parser nicht erwartetes Zeichen, so gibt er eine Syntax-Fehlermeldung aus. Durch diese Scanner-Konstruktion können also lexikalische Fehler überhaupt nicht auftreten bzw. werden nicht als solche erkannt.

```
TYPE   { if ( lineno >= start_lineno ) return TYPE; }
\.\.+  { if ( lineno >= start_lineno ) return DOTS; }
...    ...
.      { if ( lineno >= start_lineno ) return yytext[0]; }
```

Abbildung 6.2: Beispiele für Regelaktionen, die ein Token liefern

Token als Rückgabewert und Attributwertermittlung

Der $^{\text{art}}$*deco*-Scanner verwendet nur Zeichenketten-Attribute; sie enthalten die Darstellung der zugehörigen Quelltextelemente. Diejenigen lexikalischen Elemente, für die der Parser zusätzlich zu ihrem Token die Angabe ihrer textuellen Repräsentation benötigt, lassen sich in drei Gruppen unterteilen:

- Literale Konstanten:
  Das Zeichenketten-Attribut ist notwendig, damit der Parser die Präfix-Ausgabe erzeugen kann. Der Scanner erzeugt bereits die interne Repräsentation aus der Zeichenkette,

[1] Diese Werte sind größer als 255, damit auch einzelne ASCII-Zeichen (Wert kleiner als 256) als Token interpretiert werden können.

die aus der Eingabedatei gelesen wurde. In der Regel unterscheiden sich die externe und die interne Darstellung nicht und es genügt daher eine Speicherallokation sowie das Kopieren der Zeichenkette. Nur die Symbole müssen so transformiert werden, daß das Zeichenketten-Attribut keine Kleinbuchstaben und Leerzeichen enthält. Dazu werden alle Kleinbuchstaben in Großbuchstaben und alle Leerzeichen in Unterstriche (_) umgewandelt. Weiterhin werden die umschließenden Apostrophe entfernt.

- Bezeichner:
  Hier wird das Zeichenketten-Attribut zur Bezeichneridentifikation in der Symboltabelle und zur Erzeugung der Präfix-Ausgabe benötigt.

- Operatoren:
  Die Zuordnung eines Tokens zu einer Gruppe von Operatoren (z. B. das Token `RELOP` für die relationalen Operatoren `<`, `<=`, `>=` und `>`) vereinfacht die Formulierung der kontextfreien Grammatik. Da die Information, welcher Operator aktuell vorliegt, zur Erzeugung der Präfix-Ausgabe nicht verloren gehen darf, wird seine Quelltext-Darstellung im Zeichenketten-Attribut gespeichert.

In der Implementierung wird die von Lex zur Verfügung gestellte globale Variable `yylval` zur Attributwertspeicherung verwendet. Das Datenfeld `yylval.lisp` ist vom Typ `char*` und zeigt auf den Speicherbereich des Attributwerts. Die globale Variable `yytext` zeigt auf die vom Scanner eingelesene Quelltext-Zeichenkette.

In der Abbildung 6.3 wird am Beispiel des lexikalischen Elements *number* eine Regelaktion mit Attributwertermittlung gezeigt. Dabei ist `{number}` die Verwendung einer regulären Definition, die analog zu Abbildung 3.6 in Lex-Notation den Aufbau einer Zahl definiert. Die im Aktionsteil verwendete Funktion `strdup()` ist eine C-Standardfunktion zum Duplizieren von Zeichenketten.

```
{number}  { if ( lineno >= start_lineno )
               { yylval.lisp = strdup ( yytext ); return NUMBER; } }
```

Abbildung 6.3: Beispiel für Regelaktion mit Attributwertermittlung

6.1.3 Hilfsfunktionen

Die einzige dem Scanner zugeordnete Hilfsfunktion wird nicht zur eigentlichen lexikalischen Analyse und damit nicht als Lex-Spezifikation benötigt, sondern dient dem zeitlich vorgelagerten Aufbau einer Symboltabelle. Da hierzu die Komponentenbeschreibung aus der Quelldatei eingelesen werden muß, kann diese Funktion als erweiterte Scanneraufgabe angesehen werden und wird deshalb an dieser Stelle erläutert.

Die folgenden Abschnitte behandeln die Syntax der Komponentenbeschreibung als Eingabe der Funktion `InitSymbolTab()`, die Ausgabe dieser Funktion (Repräsentation der Symboltabelle im Speicher) sowie ihre Arbeitsweise.

6.1.3.1 Aufbau der Komponentenbeschreibung

Die Komponentenbeschreibung gliedert sich in vier Abschnittsarten, die jeweils mit einem in eckigen Klammern eingeschlossenen Schlüsselwort[2] beginnen. Auch hier gilt, daß Füllzeichen überlesen werden. Kommentare können analog zur Verhaltensbeschreibungssprache mit einem Prozentzeichen eingeleitet werden und gelten bis zum Zeilenende. Der Aufbau der Abschnitte ist im einzelnen:

- `[OBJECT]`
  Hier wird ein Name für die zu beschreibende Komponente definiert.

- `[GLOBALS]`
  Dieser Abschnitt enthält die Deklarationen für Variablen, deren korrespondierende physikalische Größen sich auf die Komponente beziehen.
  Jede Variablendeklaration kann bis zu vier durch Kommata getrennte Angaben umfassen, von denen nur der erste obligatorisch ist. Jede Deklaration muß in einer eigenen Zeile vorgenommen werden und darf sich nicht über mehrere Zeilen erstrecken. Die Angaben sind:

  1. *Name*: Eine eindeutige Zeichenkette gemäß der regulären Definition für *simple-id* in Abbildung 3.2. Dieser Name wird zur Erzeugung der internen Darstellung der Variable verwendet.
  2. *Alias-Name*: Dito. Der Alias-Name gibt die externe Darstellung der Variable an.
  3. *Typ*: Als Typ ist eine der folgenden Angaben zulässig: `number`, `string`, `symbol`, `boolean`. Wird keine Angabe gemacht, wird der Typ `number` angenommen.
  4. *Wertebereich*: Die Notation des Wertebereichs hängt vom Typ der Variable ab. Bei `number`, `string` und `symbol` ist sie analog zur in Abschnitt 3.4.2 dargestellten Schreibweise für Mengen anzugeben. Der Wertebereich für `boolean` ist immer gleich und kann entfallen.

  Auch bei ausgelassenen Angaben muß ein Komma gesetzt werden, es sei denn, die Kommata stehen am Zeilenende.

- `[GATE` x`]`, wobei x eine Ganzzahl ist
  Für jeden Anschluß (Gate) der Komponente ist ein solcher Abschnitt anzugeben. Die Notation dieser Abschnitte ist die gleiche wie unter `[GLOBALS]`. Die interne Darstellung für jede Variable muß nur innerhalb eines Abschnitts eindeutig sein, da automatisch die Gatenummer zur weiteren Spezifikation herangezogen wird (vgl. Abschnitt 6.1.3.2); der Alias-Name hingegen muß für die gesamte Komponentenbeschreibung eindeutig sein. Die Anzahl der Abschnitte dieser Art ist nur durch den vorhandenen Speicherplatz begrenzt.

- `[TRANSITIONS]`
  In diesem Abschnitt sind die Komponentenverhaltensbeschreibungen gemäß der Syntax aus Kapitel 3 abgelegt. Sie dienen dem eigentlichen Übersetzer als Eingabe und werden von der Funktion zum Symboltabellenaufbau überlesen.

[2] Diese Schlüsselwörter beziehen sich nur auf die Funktion `InitSymbolTab()` zur Unterscheidung der Komponentenbeschreibungsabschnitte, sie haben also nichts mit den Schlüsselwörtern der Verhaltensbeschreibungssprache zu tun.

Die Funktion `InitSymbolTab()` liest jede Zeile der Eingabedatei und speichert die enthaltenen Informationen in der aufzubauenden Symboltabelle ab. Tritt ein Fehler auf, so erfolgt keine Meldung und die Verarbeitung wird auch nicht unterbrochen, sondern mit der nächsten Quelltextzeile fortgesetzt. Kommen zwei Variablendeklarationen mit dem gleichen Namen oder Alias-Namen an Stellen vor, an denen Eindeutigkeit vorausgesetzt wird, so wird die zweite Deklaration ignoriert, auch wenn sie sich z. B. im Variablentyp unterscheiden sollte.

Die Abbildung 6.4 enthält Fragmente einer Komponentenbeschreibung, die von der Funktion `InitSymbolTab()` akzeptiert wird; ein vollständiges Beispiel ist in Anhang B zu finden.

```
% Komponentenbeschreibung

[OBJECT]
4/3-Wege-Proportionalventil

[GLOBALS]
position,     ,    symbol,   ['open', 'blocked', 'crossed']
Rh % hydraulischer Widerstand
...

[GATE 1]
acceleration

...

[GATE 3]
flow,          Q3,  number,   [-5 .. 5]
pressure,      p3,

...

[TRANSITIONS]
BEGIN
 Q3 = 0;
 ...
END
```

Abbildung 6.4: Beispiel für eine Komponentenbeschreibung in $^{\text{art}}$*deco*

6.1.3.2 Repräsentation der Symboltabelle

Die Symboltabelle besteht aus einer verketteten Liste von Einträgen, die die eingelesenen Informationen enthalten. Beim Aufbau der Symboltabelle wird für jede eindeutige Variablendeklaration der Abschnitte `[GLOBALS]` und `[GATE` x`]` ein neuer Eintrag angelegt. Die Konvertierung der Quellangaben in die interne Darstellung geschieht dabei wie folgt:

- *Name*: Um die globale Eindeutigkeit der Variablennamen zu gewährleisten, wird für Namen innerhalb der Gate-Abschnitte die Gatenummer in die interne Präfix-Darstellung integriert. So wird z. B. der Name `flow` an Gate 3 aus Abbildung 6.4 in `GATE3@flow`

umgewandelt und gespeichert. Die Namen innerhalb von `[GLOBALS]` werden hingegen unverändert im Symboltabelleneintrag abgelegt.

- *Alias-Name*: Hier ist keine Konvertierung notwendig, weil die Alias-Namen bereits in der Eingabedatei eindeutig definiert sein müssen. Fehlt diese Angabe, so kann die Variable nur über ihren Namen referenziert werden.

- *Typ*: Der Typ wird intern als Ganzzahl kodiert; eine fehlende Angabe entspricht `number`.

- *Wertebereich*: Fehlende Wertebereiche implizieren eine Zulässigkeit aller möglichen Werte. Sonst wird für numerische Variablen die Unter- und die Obergrenze gespeichert; die zulässigen Werte für Symbole werden in einer linearen Liste abgelegt. Für Strings und boolsche Variablen sind keine Wertebereichseinschränkungen vorgesehen.

Nachdem die Symboltabelle vollständig aufgebaut ist, sind alle in den Verhaltensbeschreibungen zulässigen Variablen bekannt. Der Parser kann innerhalb der semantischen Regeln auf diese Informationen zugreifen und diverse Prüfungen vornehmen (siehe z. B. Abschnitt 6.2.3.4). Eine Erweiterung der Symboltabelle um zusätzliche Variablendeklarationen ist nun nicht mehr möglich. Das Auftreten nicht gespeicherter Variablennamen in den Verhaltensbeschreibungen führt also zu einer Fehlermeldung.

Eine Ausnahme stellen Bezeichner für Relationen und in Typdefinitionsanweisungen dar, denn sie müssen nicht in der Komponentenbeschreibung deklariert werden. Damit aber auch diesbezügliche Informationen ausgewertet werden können, nimmt der Parser ihre Zwischenspeicherung in der Symboltabelle zum Zeitpunkt ihres ersten Auftretens vor.

6.2 Implementation des artdeco-Parsers

Analog zur Scannerspezifikation besteht die Parserspezifikation als Eingabe für Yacc aus drei Teilen, von denen der erste und dritte Teil optional sind. Auch hier werden die drei Teile syntaktisch durch zwei aufeinanderfolgende Prozentzeichen getrennt; in den folgenden Abschnitten wird näher auf sie eingegangen.

1. Deklarationen und Definitionen, optional
`%%`
2. Yacc-spezifische Übersetzungsregeln
`%%`
3. benutzerdefinierte Hilfsfunktionen, optional

6.2.1 Deklarationen und Definitionen

In diesem Teil der Yacc-Eingabe werden C-Variablen, -Strukturen und -Funktionsprototypen festgelegt, die im Regelteil oder in den Hilfsfunktionen benötigt werden.

In bezug auf die Grammatikdefinition enthält der Deklarations- und Definitionsteil der Yacc-Eingabe nur die Definitionen der terminalen und nichtterminalen Grammatiksymbole sowie des Startsymbols; die Produktionen der Grammatik werden im Regelteil (siehe Abschnitt 6.2.2) definiert. Mit Hilfe eines Yacc-Schlüsselwortes wird jeweils die Art der darauffolgenden Symbole festgelegt. Definitionen von Nichtterminalen beginnen mit `%type`, Definitionen von Terminalen mit `%token`. Terminale, die aus nur einem Zeichen bestehen, können im Regelteil mit Apostrophen eingeschlossen referenziert werden; eine Tokendefiniton entfällt dann. Die Definition des Startsymbols wird mit dem Schlüsselwort `%start` eingeleitet.

Die Angabe der Symbole hat zwei wichtige Aufgaben: Erstens kann jedem Token von Yacc eine eindeutige Ganzzahl zugeordnet werden. Diese Festlegungen sind ein wesentlicher Bestandteil der Schnittstellendefinition zwischen Scanner und Parser. Zweitens können Prioritäts- und Assoziativitätsbedingungen für mehrdeutige Grammatiken formuliert werden. Dieses wird im folgenden näher ausgeführt.

Konflikauflösungen

Erhält Yacc eine mehrdeutige kontextfreie Grammatik als Eingabe, so treten bei der Konstruktion des Kellerautomaten Konflikte auf (vgl. Abschnitt 5.4.2.2). Da die Formulierung einer konfliktfreien – d. h. einer der LALR(1)-Bedingung genügenden – Grammatik für den Entwickler aufwendig sein kann bzw. schwerer verständliche Produktionen bedingt, besitzt Yacc vorgegebene Verhaltensregeln zur Beseitigung der Konflikte. Diese Standardregeln sind:

- bei Shift-reduce-Konflikt:
  Die Shift-Operation wird durchgeführt. Dieses Vorgehen impliziert eine Rechtsassoziativität.

- bei Reduce-reduce-Konflikt:
  Es wird diejenige Regel reduziert, die im Regelteil textuell zuerst definiert wurde.

Da diese Standardregeln nicht immer erwünscht sind, hat der Entwickler die Möglichkeit, im Deklarations- und Definitionsteil andere Konfliktauflösungen anzugeben. Zur Festlegung von Assoziativitäten wird das Schlüsselwort `%token` durch eines der Schlüsselwörter `%left` (Linksassoziativität), `%right` (Rechtsassoziativität) oder `%nonassoc` (keine Assoziativität, d. h. das Symbol darf nicht mehrmals hintereinander angegeben werden) ersetzt. Die Priorität der Symbole wird durch die textuelle Reihenfolge ihrer Definition in aufsteigender Form bestimmt. Werden in einer `%token`-, `%left`-, `%right`- oder `%nonassoc`-Definition mehrere Symbole aufgeführt, so besitzen diese untereinander die gleiche Priorität.

Ein kurzes Beispiel soll die Konfliktauflösung anhand einer konkreten Situation verdeutlichen. In der kontextfreien Grammatik dieses Projekts existiert eine Produktion (vgl. Abbildung 3.13)

$$expr \quad \longrightarrow \quad expr\ sum\text{-}op\ expr$$

Während der Parsergenerierung erzeugt Yacc für den Kellerautomaten einen Zustand, der u. a. die folgenden Situationen enthält:

$[expr \longrightarrow\ expr \bullet \texttt{SUMOP}\ expr,\ \ R]$
$[expr \longrightarrow\ expr\ \texttt{SUMOP}\ expr \bullet,\ \ \{\texttt{SUMOP}\}]$,

wobei R ein nicht näher spezifizierter Rechtskontext und `SUMOP` das Token für die Terminale `+` und `-` ist. Man erkennt, daß in diesem Zustand ein Shift-reduce-Konflikt bei einem Lookahead-Token `SUMOP` vorliegt. Die Standard-Konfliktauflösung von Yacc würde eine Shift-Operation vorsehen; es läge also eine rechtsassoziative Abarbeitung vor. Da die Summen-Operatoren `+` und `-` in der Mathematik und somit sinnvollerweise auch in der Verhaltensbeschreibungssprache linksassoziativ sind, kann mit der Definition

```
%left SUMOP
```

im oben beschriebenen Zustand eine Reduce-Operation erzwungen werden, so daß die Abarbeitung linksassoziativ erfolgt.

Eine textuell nachgeordnete Definition

```
%left PRODOP
```

legt dann neben der Assoziativität fest, daß die Produkt-Operatoren `*` und `/` eine höhere Priorität als `SUMOP` besitzen. Auf diese Weise wird die „Punkt-vor-Strich“-Regel formuliert, die allein mit der vorliegenden kontextfreien Grammatik nicht definiert wäre.

Die vollständige Definition der Assoziativitäts- und Prioritätsregeln der Verhaltensbeschreibungssprache von $^{\text{art}}$*deco* ist Anhang A.1 zu entnehmen.

Semantische Informationen

Zu fast jedem Symbol der kontextfreien Grammatik wird ein Attribut benötigt, das semantische Informationen aufnehmen kann. Eine Ausnahme bilden nur diejenigen Terminalsymbole, deren Semantik eindeutig ist, wie z. B. die Zeichenkette `IN?`, die durch das Token `ISIN` vollständig beschrieben wird. Alle anderen Symbole benötigen mindestens eins von zwei möglichen Attributen (vgl. Abschnitt 4.1.1).

Die beiden Nichtterminale *identifier* und *expr* besitzen das Attribut `info`, das einen Zeiger auf einen Speicherbereich mit Informationen über die textuelle Repräsentation, den Typ und evtl. den korrespondierenden Symboltabelleneintrag (bei *identifier*) enthält. Die anderen terminalen und nichtterminalen Symbole besitzen das Attribut `lisp`, also einen Zeiger auf die Zeichenkette mit der Präfix-Darstellung, die für den aktuellen Knoten im Parse-Baum erzeugt wurde.

Die Deklaration der Attribute wird mit dem Yacc-Schlüsselwort `%union` eingeleitet; die Attribute können dann als Datenfelder der globalen Kommunikationsvariablen `yylval` referenziert werden.

Die Zuordnung der Attribute zu den Grammatiksymbolen erfolgt für Terminale in den Token-Definitionen `%token`, `%left`, `%right` und `%nonassoc`. Das entsprechende Attribut wird hinter dem Schlüsselwort in spitzen Klammern angegeben. Das folgende Beispiel weist also dem Token `SUMOP` das Zeichenketten-Attribut zu

```
%left <lisp> SUMOP
```

und kann dann die Werte `"+"` und `"-"` annehmen. Der Unterschied bei der Attributzuordnung für Nichtterminale besteht im einleitenden Schlüsselwort `%type`. Sie lautet z. B. für *expr*

```
%type <info> expr
```

Da jedem Yacc-Grammatiksymbol maximal ein Attribut zugeordnet werden kann, enthält das Attribut `info` – wie oben beschrieben – einen Zeiger auf eine Zeichenkette für die Präfix-Darstellung. Hierdurch ist auch das Attribut `lisp` in den semantischen Informationen `info` enthalten.

Auch an dieser Stelle sei auf den Anhang A.1 verwiesen, der die vollständige Angabe der Attributzuordnungen enthält.

6.2.2 Übersetzungsregeln

Der zweite Teil der Yacc-Spezifiktion enthält die Produktionen der kontextfreien Grammatik und die zugehörigen semantischen Regeln. Im folgenden wird zunächst die Regelsyntax für Yacc beschrieben, bevor auf ausgewählte Beispiele aus der Implementation der vorliegenden Diplomarbeit eingegangen wird.

6.2.2.1 Aufbau der Regeln

Regelsyntax

Der syntaktische Aufbau einer Übersetzungsregel ist wie folgt:

$$
\begin{array}{lll}
l & : & r_1 \; \{ \; s_1 \; \} \\
 & | & r_2 \; \{ \; s_2 \; \} \\
 & \ldots & \\
 & | & r_n \; \{ \; s_n \; \} \\
 & ; &
\end{array}
$$

Hierbei ist l die linke Seite der Produktion (also ein Nichtterminal) und die r_i sind die alternativen rechten Seiten der Produktion (bestehend aus Nichtterminalen und Terminalen). Mit geschweiften Klammern umschlossene semantische Regeln s_i sind als eine Folge von C-Anweisungen kodiert. Zur Referenzierung der Attributwerte existieren Variablen in Yacc-Syntax: Das Symbol `$$` verweist auf den Attributwert, der mit dem Nichtterminal auf der linken Seite assoziiert ist, während die Symbole `$`i den mit dem i-ten Grammatiksymbol der rechten Seite assoziierten Attributwert referenzieren. Die semantischen Regeln werden immer dann ausgeführt, wenn mit der korrespondierenden Produktion reduziert wird.

Eine semantische Regel für den $^{\text{art}}deco$-Parser besteht im wesentlichen aus drei Teilen:

1. Überprüfung der statischen Semantik (Typüberprüfungen), d. h. Auswertung und Erzeugung der Typ-Attribute, falls vorhanden,

2. Erzeugung der Präfix-Ausgabe, d. h. Auswertung und Erzeugung der Zeichenketten-Attribute,
3. Fehlerbehandlung.

Typüberprüfung

Typüberprüfungen sind sinnvoll für die Grammatiksymbole *identifier* und *expr*, denn nur zu ihnen sind semantische Informationen vorhanden, die u. a. den Typ angeben. Als Typkodierung können die Konstanten `NUMB`, `BOOL`, `SYMB`, `STRG` und `NAME` verwendet werden.

Die Makros `IsType()` und `EqType()` dienen der Typüberprüfung. Beide Makros liefern als Ergebnis einen Wahrheitswert. Die Parameter von `IsType()` sind ein Attribut `info`, also ein Zeiger auf semantische Informationen und eine der oben aufgeführten Typkonstanten. Der Rückgabewert ist genau dann „wahr", wenn die Typkonstante mit dem in `info` gespeicherten Wert übereinstimmt. `EqType()` hingegen bekommt zwei Zeiger auf semantische Informationen übergeben und liefert genau dann „wahr", wenn die Typinformationen gleich sind. Das Ergebnis der Makroaufrufe kann der Erzeugung der Präfix-Ausgabe oder der Fehlerbehandlung dienen.

Erzeugung der Präfix-Darstellung

Auch die Ermittlung des Zeichenketten-Attributs wird durch einige Makros unterstützt. `IsLisp()` liefert genau dann „wahr", wenn die übergebene Zeichenkette nicht leer ist. Das Makro `InLisp()` sucht den als zweiten Parameter angegebenen Buchstaben in der Zeichenkette (erster Parameter). Im Erfolgsfall wird „wahr" zurückgegeben, „falsch" sonst. Das letzte Makro, `DelLisp()`, löscht den Zeichenkettenparameter aus dem Speicher. Das zeichenkettenerzeugende Gegenstück, die Funktion `NewLisp()`, wird im Abschnitt 6.2.3 beschrieben.

Die Abbildungen 6.5 (Seite 79), 6.6 (Seite 80) und 6.7 (Seite 81) enthalten Beispiele für den Einsatz der erwähnten Makros und Funktionen.

Die Strategie für die Erzeugung des Zeichenketten-Attributs ist dabei immer dieselbe: Abhängig von den Attributwerten der Grammatiksymbole der rechten Produktionsseite wird das Attribut des linken Symbols mit `NewLisp()` erzeugt. Die Informationen der Attributwerte der rechten Seite sind nun im Zeichenketten-Attribut der linken Seite (im Parse-Baum also im Vaterknoten) enthalten und können mit `DelLisp()` gelöscht werden.

Fehlerbehandlung

Zur Fehlerbehandlung dient die in Abschnitt 6.2.3 erläuterte Funktion `yyerror()`, die eine Fehlerausgabe ermöglicht. Weil sich die auszugebenden Fehlermeldungen in der Yacc-Spezifikation häufig wiederholen, sind hierfür einige konstante Zeichenketten definiert (siehe Quelltext der Implementation). Da der Speicherplatz für die Attributwerte in jedem Fall (also auch im Fehlerfall) freigegeben werden sollte, erfolgt der Aufruf der Funktion `yyerror()` am Ende einer semantischen Regel. Vorher evtl. aufgetretene Fehlerzustände können in den beiden boolschen Variablen `ok` und `ok2` zwischengespeichert werden.

6.2.2.2 Ausgewählte Beispiele

Die in diesem Abschnitt erläuterten Beispiele für die Implementation der Übersetzungsregeln im ^{art}*deco*-Parser sollen die grundlegenden Aspekte sowie die Benutzung der Hilfsfunktionen vorführen; sie stellen daher keine vollständige Referenz dar. Nicht erwähnte Einzelheiten sind dem Quelltext der Implementation zu entnehmen.

Erzeugung der Präfix-Darstellung

Die Startsymbol *description* befindet sich in der Wurzel des Parse-Baums. Kann nach diesem Nichtterminal abgeleitet werden, so ist der Parse-Vorgang beendet. Das bedeutet, daß die Präfix-Ausgabe vollständig erzeugt und in einem Zeichenketten-Attribut gespeichert ist. Dieser Attributwert wird an die Standardausgabe `stdout` übergeben und kann mit dem Makro `DelLisp()` gelöscht werden.

Abbildung 6.5 zeigt die beschriebene Übersetzungsregel. Die Funktion `fprintf()` ist in den C-Standardbibliotheken vorhanden und schreibt Daten in die angegebene Ausgabeeinheit. Ihre Verwendung ist der Dokumentation für die entsprechende C-Bibliothek zu entnehmen.

```
description  :  statement
                {
                  fprintf ( stdout, "%s\n", $1 );
                  DelLisp ( $1 );
                }
             ;
```

Abbildung 6.5: Übersetzungsregel für *description*

Typüberprüfung

Das zweite Beispiel soll die Vorgehensweise der Typüberprüfung im Zusammenspiel mit der Fehlerbehandlung sowie die Behandlung des Attributs zur Speicherung von semantischen Informationen (`info`) erläutern. Es handelt sich um die Implementation eines Ausdrucks, der aus zwei untergeordneten Ausdrücken und dem dazwischen stehenden Potenzierungszeichen (`^`, Token `POWER`) besteht (vgl. Abbildung 3.13).

Als erstes wird mit dem Makro `IsType()` überprüft, ob es sich bei beiden Ausdrücken der rechten Seite um numerische Ausdrücke handelt. Ist dieses der Fall, kann mit `NewLisp()` die Präfix-Ausgabe als Zeichenketten-Attribut sowie mit `NewInfo()` die semantischen Informationen für das Attribut `info` der linken Seite aufgebaut werden. Die genaue Beschreibung dieser Funktionen erfolgt in Abschnitt 6.2.3. Danach werden die semantischen Informationen der rechten Audrücke mit `DelInfo()` aus dem Speicher entfernt; sie sind nun vollständig in den Attributwerten für *expr* der linken Produktionsseite enthalten. Liegen falsche Ausdruckstypen vor, erfolgt eine Fehlerausgabe.

Die Abbildung 6.6 gibt die Implementation der obigen Ausführungen wieder. Dabei erzwingt das Yacc-Makro `YYERROR` einen Programmabbruch und die Konstante `ERRBOOL` steht für die Zeichenkette

`"Type mismatch: expression with type ‘boolean’ expected"`,

die von der Funktion `yyerror()` an die Standardausgabe `stdout` ausgegeben wird.

```
expr   :   expr POWER expr
           {
             if ( ( ok = IsType ( $1, NUMB ) && IsType ( $3, NUMB ) ) )
               $$ = NewInfo ( NewLisp ( "(EXPT %s %s)", $1->lisp, $3->lisp ),
                              NUMB );
             DelInfo ( $1 );
             DelInfo ( $3 );
             if ( !ok )
             {
               yyerror ( ERRNUMB );
               YYERROR;
             }
           }
       ;
```

Abbildung 6.6: Übersetzungsregel für einen numerischen Ausdruck

Verwendung der Symboltabelle

Ein wichtiges Element innerhalb der Übersetzungsregeln ist die Interaktion mit der Symboltabelle. Wie in Abschnitt 6.1.3 beschrieben wurde, wird die Symboltabelle vor dem Start des eigentlichen Übersetzers aufgebaut, damit die entsprechende Komponentenbeschreibung dem Übersetzer zur Verfügung steht. Hierbei geht es im wesentlichen um die Alias-Namen, Typen und Wertebereiche der Variablen. Neben diesem festen Sockel von Symboltabelleneinträgen kann der Parser weitere Eintragungen vornehmen.

Eine Notwendigkeit des Zugriffs auf die Symboltabelle besteht in denjenigen Übersetzungsregeln, die zu Produktionen der kontextfreien Grammatik mit den Nichtterminalen *simple-id* und *identifier* definiert werden. In einigen Regeln werden nur Symboltabelleneinträge gelesen, in anderen müssen auch neue Einträge erzeugt werden können. Neueinträge sind notwendig, wenn Bezeichner vom Typ `NAME` erwartet werden, wie z. B. in der Typdefinitionsanweisung, die im folgenden näher erläutert wird.

Für den in der Typdefinitionsanweisung auftretenden Bezeichner *simple-id* (bzw. `SIMPLEID` als Token) muß sichergestellt werden, daß er erstens im bisherigen Übersetzungsvorgang noch nicht mit einem anderen Typ als `NAME` vorkam und daß zweitens ein späteres Vorkommen dieses Bezeichners mit einem Typ ungleich `NAME` zu einer Fehlermeldung führt. Daher wird zunächst mit der Funktion `GetSymb()` überprüft, ob der aktuelle Bezeichner bereits in der Symboltabelle gespeichert ist. Wenn nicht, wird er durch die Funktion `NewSymb()` mit der Typinformation `NAME` eingefügt. An dieser Stelle der Aktion ist der Bezeichner `SIMPLEID` also auf jeden Fall in der Symboltabelle gespeichert. Nun wird getestet, ob dieser (eventuell

schon länger bestehende) Symboltabelleneintrag den Typ `NAME` hat: Wenn ja, wird eine leere Zeichenkette erzeugt (denn die Typdefinitionsanweisung bewirkt keine Präfix-Ausgabe), wenn nein, erfolgt eine Fehlermeldung.

Die Implementation dieser Aktionen ist der Abbildung 6.7 zu entnehmen. Die Konstante `ERRNAME` steht analog zu oben für die Fehlerzeichenkette

`"Type mismatch: expression with type 'number' expected"`.

Die Funktionen zur Symboltabellenmanipulation werden in Abschnitt 6.2.3 erläutert; die globale Variable `symb` stellt einen Zeiger auf einen Symboltabelleneintrag dar und dient zu seiner Zwischenspeicherung.

```
type_statement   :  TYPE SIMPLEID ';'
                    {
                      if ( !( symb = GetSymb ( $2, GS_STR ) ) )
                        symb = NewSymb ( NewLisp ( $2 ), NAME );
                      if ( ( ok = IsType ( symb, NAME ) ) )
                        $$ = NewLisp ( "" );
                      DelLisp ( $2 );
                      if ( !ok )
                      {
                        yyerror ( ERRNAME );
                        YYERROR;
                      }
                    }
                 ;
```

Abbildung 6.7: Übersetzungsregel für *type-statement*

6.2.3 Hilfsfunktionen

Bei der Implementation des Übersetzers der artdeco-Verhaltensbeschreibungssprache wurden Hilfsfunktionen verwendet, die sich in vier Kategorien einteilen lassen:

- Verwaltung der Symboltabelle,
- Verwaltung des Attributs für die semantischen Informationen (`info`),
- Verwaltung des Attributs für die Präfix-Ausgabe (`lisp`),
- sonstige Hilfsfunktionen.

Diese Hilfsfunktionen werden in den folgenden Abschnitten kurz erläutert.

6.2.3.1 Funktionen für die Symboltabellenverwaltung

Die Funktionen zur Implementation der Symboltabelle arbeiten auf der im Abschnitt 6.1.3.2 angedeuteten Datenstruktur, also einer verketteten Liste von Symboltabelleneinträgen des Typs `SymbolTab`.

Insgesamt stehen vier verschiedene Funktionen zur Verfügung, die für die Speicherplatzverwaltung und die Informationsrückgewinnung sorgen:

- `void InitSymbolTab (void)`
  Diese Funktion wird vor dem eigentlichen Übersetzungsvorgang aufgerufen; sie liest die Komponentenbeschreibung und speichert jede Variablenvereinbarung in der Symboltabelle als neuen Eintrag. Weitere Angaben sind dem Abschnitt 6.1.3.2 ab Seite 73 zu entnehmen.

- `void KillSymbolTab (void)`
  Nach der Beendigung des Übersetzungsvorgangs wird `KillSymbolTab()` aufgerufen, um den von der Symboltabelle dynamisch belegten Speicherplatz wieder freizugeben. Dieses geschieht unabhängig vom Erfolg der Übersetzung, also auch im Fehlerfall. Es werden immer *alle* Einträge gelöscht.

- `SymbolTab *NewSymb (char *name, int type)`
  Mit dieser Funktion wird ein neuer Symboltabelleneintrag an die bestehende Symboltabelle angefügt. Der Parameter `name` zeigt dabei auf die Zeichenkette, die im Namensfeld gespeichert wird, der Parameter `type` wird im Typfeld des neuen Eintrags abgelegt. Der Rückgabewert ist der Zeiger auf den neuen Symboltabelleneintrag.

- `SymbolTab *GetSymb (char *str, int GS_type)`
  Die Abfrage, ob ein bestimmter Symboltabelleneintrag existiert, wird mit der Funktion `GetSymb()` vorgenommen. Der Parameter `str` zeigt auf die zu suchende Zeichenkette. Diese Suche geschieht linear und wird mit dem Parameter `GS_type` eingeschränkt. Für `GS_type` stehen die Konstantendefinitionen `GS_NAME` und `GS_ALIAS` zur Verfügung; sie bewirken die ausschließliche Suche der angegebenen Zeichenkette bei den Variablennamen bzw. den Alias-Namen. Als Funktionsergebnis wird ein Zeiger auf den gefundenen Eintrag im Erfolgsfall (bzw. ein `NULL`-Zeiger sonst) zurückgeliefert.

6.2.3.2 Funktionen für das Attribut `info`

Das Attribut `info` speichert semantische Infomationen des assoziierten Grammatiksymbols in einer Struktur `SemanInfo`. Hierbei handelt es sich um das in jedem Fall benötigte Zeichenketten-Attribut `lisp` für die entsprechende Präfix-Ausgabe, um die Typinformation `type` und um einen Zeiger auf den zum Grammatiksymbol korrespondierenden Symboltabelleneintrag. Zusätzlich können Informationen abgelegt werden, aus welchem untergeordneten Grammatiksymbol das aktuelle Grammatiksymbol abgeleitet wurde. Die Verwendung dieses Details ist jedoch dem Quelltext der Implementation zu entnehmen.

Zur Manipulation der semantischen Informationen können zwei Funktionen, die die Speicherorganisation übernehmen, verwendet werden:

- `SemanInfo *NewInfo (char *lisp, int type)`
  Die Funktion `NewInfo()` erzeugt eine Instanz der Struktur `SemanInfo` und belegt ihre Datenfelder mit den angegebenen Parametern bzw. mit Standardwerten. Der Parameter `lisp` ist ein Zeiger auf das Zeichenketten-Attribut gleichen Namens und `type` eine

Typangabe (mit einem der Werte `NUMB`, `BOOL`, `SYMB`, `STRG` oder `NAME`). Der Rückgabewert ist ein Zeiger auf den neuen Datenbereich. Sollen weitere Informationen abgelegt werden (z. B. der Zeiger auf einen Symboltabelleneintrag), so ist das außerhalb dieser Funktion vorzunehmen.

- `void DelInfo (SemanInfo *info)`
  Das Gegenstück zur obigen Funktion ist `DelInfo()`. Sie gibt den Speicherbereich frei, auf den der Parameter `info` zeigt. Dieser Zeiger besitzt danach keine Gültigkeit mehr.

6.2.3.3 Funktionen für das Attribut `lisp`

Einige Funktionen dieser Kategorie wurden bereits im Abschnitt über den generellen Regelaufbau (6.2.2.1) erläutert. Es sind die hier nicht näher beschriebenen Makros

- `int IsLisp (char *lisp)`
- `int InLisp (char *Lisp, char ch)`
- `void DelLisp (char *lisp)`

Ebenfalls in diese Kategorie fällt die Spezifikation der folgenden zeichenkettenerzeugenden Funktionen:

- `char *NewLisp (char *format, ...)`
  Diese Funktion erzeugt eine neue Präfix-Ausgabe, indem sie den benötigten Speicherplatz belegt und mit den Informationen der Parameter initialisiert. Der Aufbau der Parameterliste ist analog zur Funktion `printf()` aus den C-Standardbibliotheken, d. h. der obligatorische erste Parameter enthält die Formatangabe der zu erzeugenden Zeichenkette, jeder weitere Parameter die Angaben der einzufügenden Zeichen. Von `NewLisp()` werden nur Parameter vom Typ `char*` erwartet, in der Formatangabe sind also nur Konvertierungsanweisungen mit `%s` erlaubt. Andere Parameterarten werden nicht benötigt, weil der Übersetzer allein mit Zeichenkettenoperationen die Präfix-Darstellung erzeugen kann. Die Anzahl der Konvertierungsanweisungen und der optionalen Parameter muß genau übereinstimmen. Weitere Einzelheiten sind den entsprechenden Dokumentationen zu entnehmen. Der Rückgabewert besteht aus einem Zeiger auf die erzeugte Zeichenkette.

- `char *ChangeLisp (char *old_lisp, char *new_lisp)`
  Mit `ChangeLisp()` kann eine bestehende Zeichenkette (Parameter `old_lisp`) in eine neue (Parameter `new_lisp`) umgewandelt werden. Der alte Speicherplatz wird dazu freigegeben und neuer allokiert. Der Rückgabewert der Funktion zeigt auf den neuen Speicherbereich der Zeichenkette. Der Hauptverwendungszweck dieser Funktion besteht in der Umwandlung von leeren Zeichenketten in eine geeignete Präfix-Ausgabe, zum Beispiel in `NIL`.

6.2.3.4 Sonstige Funktionen

Abschließend werden die sonstigen Funtionen beschrieben; sie lassen sich in keine der obigen Kategorien einordnen:

- `int IsDom (SymbolTab *symb, char *str)`
  Mit der Funktion `IsDom()` kann geprüft werden, ob ein bestimmter Wert, der als Parameter `str` übergeben wird, im Wertebereich des mit dem Zeiger `symb` spezifizierten Symboltabelleneintrag enthalten ist. Sinnvolle Wertebereichsanalysen beschränken sich auf die Variablentypen `NUMB` und `SYMB`. Hat der Eintrag, auf den `symb` zeigt, den Typ `NUMB`, so wird mit `str` eine Zeichenkette erwartet, die sich in einen numerischen Wert transformieren läßt. Liegt der Typ `SYMB` vor, so wird eine Zeichenkette erwartet, die der Syntax für Symbole entspricht. Der Rückgabewert ist `NULL`, falls ein falscher Typ vorliegt oder `str` nicht im Wertebereich des Symboltabelleneintrags liegt; er ist ungleich `NULL` sonst.

- `int yyerror (char *str)`
  Diese Funktion dient der Fehlerausgabe. Sie schreibt die Zeichenkette `str` in die Standardausgabe `stdout` und liefert den Wert Eins zurück. Der Rückgabewert veranlaßt die Parserfunktion `yylex()` zur Beendigung des Übersetzungsprozesses (siehe auch Abschnitt 5.4.2.1).

Kapitel 7

Schlußbemerkungen

Der in dieser Diplomarbeit vorgestellte Übersetzer für das ^{art}*deco*-Projekt befindet sich derzeit im Prototypenstatus und muß bis zur Anwendungsreife noch eine umfangreiche Testphase durchlaufen.

Die Beurteilung, ob der Sprachumfang der Verhaltensbeschreibungssprache den Erfordernissen im physikalisch-technischen Bereich genügt, kann nur im Rahmen der bisherigen Anwendungsbeispiele erfolgen. Hier traten keine Formulierungsprobleme auf; gerade die Verwendung von Alias-Namen hat sich als besonders komfortabel herausgestellt. Dennoch blieb die Zahl der Produktionen der kontextfreien Grammatik und damit die Zahl der semantischen Regeln in den syntaxgerichteten Definitionen auf ein Minimum begrenzt und daher überschaubar. Sollten trotzdem Erweiterungen bzw. Änderungen der Definition für die externe oder die interne Wissensrepräsentation erforderlich sein, kann davon ausgegangen werden, daß diese mit Hilfe der vorgestellten Hilfsfunktionen in vertretbarem Aufwand möglich sind.

Für einen Einsatz des ^{art}*deco*-Übersetzers in realen Problemstellungen sollten jedoch u. a. die folgenden Punkte noch berücksichtigt und implementiert werden:

- Makros:
  Die Verwendung von Makros für häufig benötigte Formulierungen innerhalb einer Verhaltensbeschreibung kann sinnvoll sein. Hierzu ist der Einsatz eines Makroexpanders als Präprozessor vorzusehen. Eventuell kann hierfür ein existierender Präprozessor benutzt werden, oder es wird ein neu entwickelter (manuell ausprogrammiert oder mit einem Generator erstellt) in den Übersetzer integriert.

- Fehlerbehandlung:
  Die Fehlerbehandlung muß über semantische Typprüfungen hinaus erweitert werden. Bisher wird dem Anwender ein syntaktischer Fehler nicht näher spezifiziert. In der kontextfreien Grammatik definierte Fehlerproduktionen können dagegen genauere Fehlermeldungen ermöglichen. Der Parsergenerator Yacc bietet zur Erzeugung von geeigneten Diagnosemeldungen spezielle Verarbeitungsmechanismen derartiger Fehlerproduktionen an. (siehe z. B. [ASU88a]).
  Auch die Fehlererkennung bezüglich der Referenzierung von Zustandsgrößen *anderer*

Komponenten kann verbessert werden. Die Voraussetzung ist dazu allerdings eine Zugriffsmöglichkeit des Übersetzers auf das interne Objektsystem von ^art^*deco*, um verschiedene Plausibilitätskontrollen (Wertebereichsprüfungen etc.) vornehmen zu können.

- Kommunikation mit dem Anwender:
  Ein Dialog mit dem Anwender ist in der aktuellen Implementierung nicht vorgesehen; es wird lediglich eine Eingabedatei gelesen und eine Ausgabedatei (evtl. mit Fehlermeldungen) erzeugt. Innerhalb der ^art^*deco*-Wissensakquisitionskomponente sollten dem Anwender jedoch zusätzlich geeignete Editier- und Kommunikationsmöglichkeiten (z. B. Hilfetexte) angeboten werden.

Anhang A

Definition der Verhaltensbeschreibungssprache

In Anhang A wird die Definition der Verhaltensbeschreibungssprache für die Wissensakquisitionskomponente des $^{\text{art}}$*deco*-Expertensystems zusammengefaßt.

A.1 Grammatiksymbole

Die folgende Aufstellung definiert in Yacc-Syntax (siehe Abschnitt 6.2.1) die verwendeten Terminale, ihre Assoziativität und Priorität sowie die zugehörigen Attribute. Dabei repräsentieren die Token `DOTS`, `ISEQ`, `ISNOTEQ` und `ISIN` die Zeichenketten `..`, `=?`, `<>` und `IN?`.

```
%token           TYPE BEGIN END RELATION CASE DEFAULT DOTS IF THEN
%right           ELSE
%token    <lisp> SIMPLEID NUMBER DIFFERENTIAL SYMBOL STRING
%nonassoc        ISIN
%left     <lisp> LOGOP
%nonassoc        ISEQ ISNOTEQ
%nonassoc <lisp> RELOP
%left     <lisp> SUMOP
%left     <lisp> PRODOP
%right           POWER
%token           NOT
%left            SIGN

%type     <lisp> statement_list statement
%type     <lisp> type_statement equation_statement boolean_statement
%type     <lisp> relation_statement relation_list relation
%type     <lisp> selection_statement selection_list selection
%type     <lisp> expr_list identifier_list symbol_list
%type     <info> expr identifier

%start description
```

A.2 Syntax

Die Notation der kontextfreien Grammatik wird in Abschnitt 3.1 definiert. Die Nichtterminale sind kursiv und die Terminale in Schreibmaschinenschriftart dargestellt.

Die Produktionen *62)* bis *79)* werden vom Scanner in der lexikalischen Analyse erkannt. Ihnen stehen also keine semantischen Regeln (siehe Abschnitt A.3) gegenüber.

1) *description* ⟶ *statement*

2) *statement* ⟶ *type-statement*
3) | *equation-statement*
4) | *relation-statement*
5) | *selection-statement*
6) | *boolean-statement*
7) | `BEGIN` *statement-list* `END`
8) | `BEGIN END`
9) | `;`

10) *statement-list* ⟶ *statement-list statement*
11) | *statement*

12) *type-statement* ⟶ `TYPE` *simple-id* `;`

13) *equation-statement* ⟶ *expr* `=` *expr* `;`

14) *relation-statement* ⟶ `RELATION` *simple-id* `(` *identifier-list* `)` `BEGIN` *relation-list* `END`

15) *selection-statement* ⟶ `IF` *expr* `THEN` *statement*
16) | `IF` *expr* `THEN` *statement* `ELSE` *statement*
17) | `CASE` *expr* `BEGIN` *selection-list* `END`

18) *boolean-statement* ⟶ *expr* `;`

19) *expr* ⟶ *expr sum-op expr*
20) | *expr prod-op expr*
21) | *expr pow-op expr*
22) | *sum-op expr*
23) | `NOT` *expr*
24) | *expr logical-op expr*
25) | *expr* `=?` *expr*
26) | *expr* `<>` *expr*
27) | *expr relation-op expr*
28) | *expr* `IN [` *number* `..` *number* `]`
29) | *expr* `IN [` *symbol-list* `]`
30) | *identifier*
31) | *symbol*
32) | *string*
33) | *number*
34) | *differential*
35) | *simple-id* `(` *expr-list* `)`
36) | `(` *expr* `)`

37) *identifier-list* ⟶ *identifier-list identifier*
38) | *identifier*
39) | ε

40) *relation-list* ⟶ *relation-list relation*
41) | *relation*
42) | ε

43) *selection-list* ⟶ *selection-list selection*
44) | *selection*
45) | ε

46) *expr-list* ⟶ *expr-list* `,` *expr*
47) | *expr*
48) | ε

49) *symbol-list* ⟶ *symbol-list* `,` *symbol*
50) | *symbol*
51) | ε

52) *relation* ⟶ `(` *expr-list* `)` `;`

53) *selection* ⟶ *relation-op expr* `:` *statement*
54) | `=?` *expr* `:` *statement*
55) | `<>` *expr* `:` *statement*
56) | `IN` `[` *number* `..` *number* `]` `:` *statement*
57) | `IN` `[` *symbol-list* `]` `:` *statement*
58) | `DEFAULT` `:` *statement*

59) *identifier* ⟶ *simple-id* `[` *simple-id* `]`
60) | *simple-id* `[` *number* `]`
61) | *simple-id*

62) *simple-id* ⟶ *first-char following-char*

63) *first-char* ⟶ `!` | `#` | `$` | `&` | `/` | `?` | `@` | `A` | ··· | `Z` | `\` | `_` | `a` | ··· | `z` | `{` | `|` | `}` | `~`

64) *following-char* ⟶ *first-char following-char*
65) | *digit following-char*
66) | `.` *following-char*
67) | ε

68) *differential* ⟶ *differential* `'`
69) | *identifier* `'`

70) *number* ⟶ *digit*$^+$ `(` `.` *digit*$^+$ `)`$^?$ `(` `E` `(` `+` | `-` `)`$^?$ *digit*$^+$ `)`$^?$

71) *digit* ⟶ `0` | ··· | `9`

72) *symbol* ⟶ `'` `(` ␣ | ··· | `&` | `(` | ··· | `~` `)`* `'`

73) *string* ⟶ `"` `(` ␣ | `!` | `#` | ··· | `~` `)`* `"`

74) *logical-op* ⟶ `AND` | `OR`

75) *relation-op* ⟶ `<` | `<=` | `>=` | `>`

76) *sum-op* ⟶ `+` | `-`

78) *prod-op* ⟶ `*` | `/`

79) *pow-op* ⟶ `^`

A.3 Semantik

Die Notation der syntaxgerichteten Definition wird in Abschnitt 4.1 näher beschrieben. Jede semantische Regel zur Berechnung des Zeichenketten-Attributs korrespondiert mit der entsprechend numerierten Produktion der kontextfreien Grammatik aus Abschnitt A.2.

```
 1)           description.s  :=  statement.s
 2)             statement.s  :=  type-statement.s
 3)             statement.s  :=  equation-statement.s
 4)             statement.s  :=  relation-statement.s
 5)             statement.s  :=  condition-statement.s
 6)             statement.s  :=  "( TEST" || boolean-statement.s || ")"
 7)             statement.s  :=  "(" || statement-list.s || ")"
 8)             statement.s  :=  nil
 9)             statement.s  :=  nil
10)      statement-list_1.s  :=  statement-list_2.s || statement.s
11)        statement-list.s  :=  statement.s
12)        type-statement.s  :=  nil
13)    equation-statement.s  :=  if ( expr_1.t = differential )
                                    then "( DIFF" || expr_1.s || expr_2.s || ")"
                                    else if ( expr_1.t = symbol )
                                            then "( IS" || expr_1.s || expr_2.s || ")"
                                            else "( =" || expr_1.s || expr_2.s || ")"
14)    relation-statement.s  :=  if ( identifier-list.s <> nil ) and
                                    ( relation-list.s <> nil )
                                    then "( (" || identifier-list.s || ")" ||
                                            relation-list.s || ")"
                                    else nil
15)   selection-statement.s  :=  if ( statement.s <> nil )
                                    then "( COND (" || expr.s || statement.s || ") )"
                                    else nil
16)   selection-statement.s  :=
            case 1 : ( statement_1.s <> nil ) and ( statement_2.s <> nil )
               then "( COND (" || expr.s || statement_1.s || ")" ||
                          "( ( NOT" || expr.s || ")" || statement_2.s || ") )"
            case 2 : ( statement_1.s <> nil ) and ( statement_2.s = nil )
               then "( COND (" || expr.s || statement_1.s || ")" ||
                          "( ( NOT" || expr.s || ") NIL ) )"
            case 3 : ( statement_1.s = nil ) and ( statement_2.s <> nil )
               then "( COND (" || expr.s || NIL )" ||
                          "( ( NOT" || expr.s || ")" || statement_2.s || ") )"
            case 4 : ( statement_1.s = nil ) and ( statement_2.s = nil )
               then nil
17)   selection-statement.s  :=  if ( selection-list.s <> nil )
                                    then "( COND" || selection-list.s || ")"
                                    else nil
18)     boolean-statement.s  :=  expr.s
```

19)	$expr_1.s$	:=	`if ( RISC ) and (` $sum\text{-}op.s$ `= " )`
			`then "( +"` $\|$ $expr_1.s$ $\|$ `"( * -1"` $\|$ $expr_2.s$ $\|$ `") )"`
			`else "("` $\|$ $sum\text{-}op.s$ $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`

20)	$expr_1.s$	:=	`if ( RISC ) and (` $prod\text{-}op.s$ `= "/" )`
			`then "( *"` $\|$ $expr_1.s$ $\|$ `"( EXPT"` $\|$ $expr_2.s$ $\|$ `"-1 ) )"`
			`else "("` $\|$ $prod\text{-}op.s$ $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`

21)	$expr_1.s$	:=	`"( EXPT"` $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`
22)	$expr_1.s$	:=	`if (` $sum\text{-}op.s$ `= "-" )` `then "-"` $\|$ $expr_2.s$ `else` $expr_2.s$
23)	$expr_1.s$	:=	`"( NOT"` $\|$ $expr_2.s$ $\|$ `")"`
24)	$expr_1.s$	:=	`"("` $\|$ $logical\text{-}op.s$ $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`

25)	$expr_1.s$	:=	`if (` $expr_2.t = symbol$ `)`
			`then "( EQUAL"` $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`
			`else "( ="` $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`

26)	$expr_1.s$	:=	`if (` $expr_2.t = symbol$ `)`
			`then "( NOT ( EQUAL"` $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `") )"`
			`else "( NOT ( ="` $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `") )"`

27)	$expr_1.s$	:=	`"("` $\|$ $relation\text{-}op.s$ $\|$ $expr_2.s$ $\|$ $expr_3.s$ $\|$ `")"`
28)	$expr_1.s$	:=	`"( AND ( >="` $\|$ $expr_2.s$ $\|$ $number_1.s$ $\|$ `")"` $\|$ `"( <="` $\|$ $expr_2.s$ $\|$ $number_2.s$ $\|$ `") )"`

29)	$expr_1.s$	:=	`if (` $symbol\text{-}list.s$ `<> nil )`
			`then "( MEMBER"` $\|$ $expr_2.s$ $\|$ `" '("` $\|$ $symbol\text{-}list.s$ $\|$ `") )"`
			`else "( MEMBER"` $\|$ $expr_2.s$ $\|$ `"NIL )"`

30)	$expr.s$	:=	$identifier.s$

31)	$expr.s$	:=	`if (` $expr\text{-}list.s$ `<> nil )`
			`then "("` $\|$ $simple\text{-}id.s$ $\|$ $expr\text{-}list.s$ $\|$ `")"`
			`else` $simple\text{-}id.s$

32)	$expr.s$	:=	`if (` $symbol.s$ `<> nil )`
			`then "'"` $\|$ $symbol.s$
			`else "NIL"`

33)	$expr.s$	:=	$string.s$
34)	$expr.s$	:=	$number.s$
35)	$expr.s$	:=	$differential.s$
36)	$expr_1.s$	:=	$expr_2.s$
37)	$identifier\text{-}list_1.s$	:=	$identifier\text{-}list_2.s$ $\|$ `"␣"` $\|$ $identifier.s$
38)	$identifier\text{-}list.s$	:=	$identifier.s$
39)	$identifier\text{-}list.s$	:=	`nil`
40)	$relation\text{-}list_1.s$	:=	$relation\text{-}list_2.s$ $\|$ $relation.s$
41)	$relation\text{-}list.s$	:=	$relation.s$
42)	$relation\text{-}list.s$	:=	`nil`
43)	$selection\text{-}list_1.s$	:=	$selection\text{-}list_2.s$ $\|$ $selection.s$
44)	$selection\text{-}list.s$	:=	$selection.s$
45)	$selection\text{-}list.s$	:=	`nil`

```
46)     expr-list1.s   :=  expr-list2.s || "␣" || expr.s
47)      expr-list.s   :=  expr.s
48)      expr-list.s   :=  nil
49)   symbol-list1.s   :=  symbol-list2.s || "␣" || symbol-list.s
50)    symbol-list.s   :=  symbol.s
51)    symbol-list.s   :=  nil
52)       relation.s   :=  if ( expr-list.s <> nil )
                              then "(" || expr-list.s || ")"
                              else nil
53)      selection.s   :=
              if ( statement.s <> nil )
                 then "( (" || relation-op.s || expr*.s || expr.s || ")" || statement.s || ")"
                 else "( (" || relation-op.s || expr*.s || expr.s || ") NIL )"
54)      selection.s   :=
              case 1 : ( statement.s <> nil ) and ( expr.t = symbol )
                 then "( ( EQUAL" || expr*.s || expr.s || ")" || statement.s || ")"
              case 2 : ( statement.s = nil ) and ( expr.t = symbol )
                 then "( ( EQUAL" || expr*.s || expr.s || ") NIL )"
              case 3 : ( statement.s <> nil ) and ( expr.t <> symbol )
                 then "( ( =" || expr*.s || expr.s || ")" || statement.s || ")"
              case 4 : ( statement.s = nil ) and ( expr.t <> symbol )
                 then "( ( =" || expr*.s || expr.s || ") NIL )"
55)      selection.s   :=
              case 1 : ( statement.s <> nil ) and ( expr.t = symbol )
                 then "( ( NOT ( EQUAL" || expr*.s || expr.s || ") )" || statement.s || ")"
              case 2 : ( statement.s = nil ) and ( expr.t = symbol )
                 then "( ( NOT ( EQUAL" || expr*.s || expr.s || ") ) NIL )"
              case 3 : ( statement.s <> nil ) and ( expr.t <> symbol )
                 then "( ( NOT ( =" || expr*.s || expr.s || ") )" || statement.s || ")"
              case 4 : ( statement.s = nil ) and ( expr.t <> symbol )
                 then "( ( NOT ( =" || expr*.s || expr.s || ") ) NIL )"
56)      selection.s   :=
              if ( statement.s <> nil )
                 then "( ( AND ( >=" || expr*.s || number1.s || ")" ||
                            "( <=" || expr*.s || number2.s || ") )" || statement.s || ")"
                 else "( ( AND ( >=" || expr*.s || number1.s || ")" ||
                            "( <=" || expr*.s || number2.s || ") ) NIL )"
57)      selection.s   :=
              case 1 : ( statement.s <> nil ) and ( symbol-list.s <> nil )
                 then "( ( MEMBER" || expr*.s || " '(" || symbol-list.s || ") )" ||
                            statement.s || ")"
              case 2 : ( statement.s = nil ) and ( symbol-list.s <> nil )
                 then "( ( MEMBER" || expr*.s || " '(" || symbol-list.s || ") ) NIL )"
              case 3 : ( statement.s <> nil ) and ( symbol-list.s = nil )
                 then "( ( MEMBER" || expr*.s || NIL )" || statement.s || ")"
              case 4 : ( statement.s = nil ) and ( symbol-list.s = nil )
                 then "( ( MEMBER" || expr*.s || NIL ) NIL )"
```

58) *selection.s* := if (*statement.s* <> nil)
then "(T" || *statement.s* || ")"
else "(T NIL)"

59) *identifier.s* := $simple\text{-}id_2.s$ || "@" || $simple\text{-}id_1.s$

60) *identifier.s* := "GATE" || *number.s* || "@" || *simple-id.s*

61) *identifier.s* := "SELF@" || *simple-id.s*

Anhang B

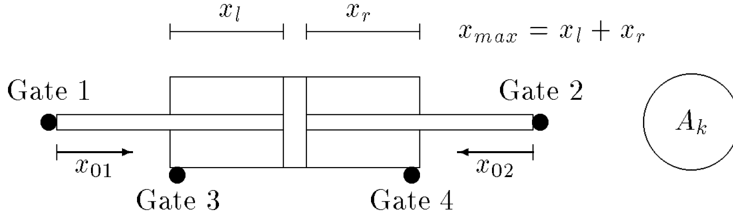
Beispiel einer Verhaltensbeschreibung

Dieses Beispiel stellt die Verhaltensbeschreibung eines Gleichgangzylinders in den Modelltiefen 0 und 1 dar. Die Modellierung für das $^{\text{art}}$*deco*-System sowie die mathematische Beschreibung ist [Sue94] (Seiten 67 bis 71) entnommen. Der Schwerpunkt der folgenden Ausführungen soll dabei nicht in der Darstellung der Modellbildung oder der physikalischen Gesetzmäßigkeiten liegen, sondern es soll an einem nichttrivialen Beispiel die vom Experten durchzuführende (ingenieursmäßige) Vorgehensweise bei der Wissensformulierung verdeutlicht werden.

B.1 Parameter

Parameter	Einheit	Beschreibung
A_k	$[\mathrm{m}^2]$	Kolbenfläche
d	$\left[\frac{\mathrm{N\ s}}{\mathrm{m}}\right]$	viskose Reibung
F_i	$[\mathrm{N}]$	Kraft an Gate i, wobei $i = 3, 4$
$h_1(\ldots)$	$[\mathrm{m}]$	Hilfsfunktion 1
$h_2(\ldots)$	$\left[\frac{\mathrm{m}}{\mathrm{s}^2}\right]$	Hilfsfunktion 2
m	$[\mathrm{kg}]$	bewegte Masse
p_i	$\left[\frac{\mathrm{N}}{\mathrm{m}^2}\right]$	Druck an Gate i, wobei $i = 3, 4$
Q_i	$\left[\frac{1}{\mathrm{s}}\right]$	Fluß an Gate i, wobei $i = 3, 4$
t	$[\mathrm{s}]$	Simulationszeit
v	$\left[\frac{\mathrm{m}}{\mathrm{s}}\right]$	Geschwindigkeit (Parameter der Hilfsfunktionen)
x_{0i}	$[\mathrm{m}]$	Position an Gate i, wobei $i = 1, 2$
x_{1i}	$\left[\frac{\mathrm{m}}{\mathrm{s}}\right]$	Geschwindigkeit an Gate i, wobei $i = 1, 2$
x_{2i}	$\left[\frac{\mathrm{m}}{\mathrm{s}^2}\right]$	Beschleunigung an Gate i, wobei $i = 1, 2$
x_{max}	$[\mathrm{m}]$	Hub

B.2 Graphische Darstellung



B.3 Mathematische Darstellung

B.3.1 Modelltiefe 0 (Statische Verhaltensbeschreibung)

$$
\begin{aligned}
x_{01}(t) &= -x_{02}(t) \\
x_{02}(t) &= \begin{cases} \frac{x_{max}}{2} & \text{für} \quad h_1(t, x_{12}(t)) \geq \frac{x_{max}}{2} \\ -\frac{x_{max}}{2} & \text{für} \quad h_1(t, x_{12}(t)) \leq -\frac{x_{max}}{2} \\ h_1(t, x_{12}(t)) & \text{sonst} \end{cases} \\
x_{11}(t) &= -x_{12}(t) \\
x_{21}(t) &= x_{22}(t) \;=\; 0 \\
Q_3(t) &= 0.06\; A_k\; x_{12}(t) \\
Q_4(t) &= 0.06\; A_k\; x_{11}(t) \\
F_2(t) - F_1(t) &= \frac{A_k}{10}\,(p_3(t) \;-\; p_4(t)) \\
h_1(t, v(t)) &= x_0 + t\; v(t)
\end{aligned}
$$

B.3.2 Modelltiefe 1 (Berücksichtigung der Masse)

$$
\begin{aligned}
\dot{x}_{02}(t) &= x_{12}(t) \\
\dot{x}_{12}(t) &= x_{22}(t) \\
x_{01}(t) &= -x_{02}(t) \\
x_{11}(t) &= -x_{12}(t) \\
x_{21}(t) &= -x_{22}(t) \\
x_{22}(t) &= \begin{cases} 0 \quad \text{für} & (x_{02}(t) \leq \frac{x_{max}}{-2} \wedge h_2(p_3(t), p_4(t), F_2(t), F_1(t), x_{12}(t)) \leq 0) \\ \qquad \vee & (x_{02}(t) \geq \frac{x_{max}}{2} \wedge h_2(p_3(t), p_4(t), F_2(t), F_1(t), x_{12}(t)) \geq 0) \\ -h_2(p_3(t), p_4(t), F_2(t), F_1(t), x_{12}(t)) & \text{sonst} \end{cases} \\
Q_3(t) &= 0.06\; A_k\; x_{12}(t) \\
Q_4(t) &= 0.06\; A_k\; x_{11}(t) \\
F_2(t) - F_1(t) &= \frac{A_k}{10}\,(p_3(t) \;-\; p_4(t)) \\
h_2(p_3(t), p_4(t), F_2(t), F_1(t), v(t)) &= \frac{A_k/10\,(p_3(t) - p_4(t)) - F_2(t) + F_1(t) + d\; v(t)}{m}
\end{aligned}
$$

B.4 Externe Darstellung in artdeco

Das hier in der Syntax der artdeco-Verhaltensbeschreibungssprache aufgeführte Beispiel enthält die Formulierung der mathematischen Gleichungen aus Abschnitt B.3.

Auf dieser Seite wird der physikalische Aufbau der Komponente „Gleichgangzylinder" beschrieben, wobei die mathematischen Bezeichner als Alias-Namen für die interne Darstellung definiert werden. Alle Variablen sind numerisch. Auf der folgenden Seite befindet sich die Darstellung des Komponentenverhaltens gemäß der Modelltiefen 0 und 1.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                          %
% Beschreibung der Komponentenstruktur                     %
%                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[OBJECT]
Gleichgangzylinder

[GLOBALS]
Ak
d
m
position,       x0
t
v
xmax

[GATE 1]
acceleration,   x21
force,          F1
position,       x01
velocity,       x11

[GATE 2]
acceleration,   x22
force,          F2
position,       x02
velocity,       x12

[CONNECTION 3]
flow,           Q3
pressure,       p3

[CONNECTION 4]
flow,           Q4
pressure,       p4
```

```
[TRANSITIONS] % Verhaltensbeschreibungen
BEGIN
  TYPE MD0;   % Modelltiefe 0 (statische Verhaltensbeschreibung)

  x01 = -x02;
  x02 = CASE h1 (t, x12)
        BEGIN
          >=  xmax / 2 :  xmax / 2;
          <= -xmax / 2 : -xmax / 2;
          DEFAULT      :  h1 (t, x12);
        END;

  x11 = -x12;
  x12 = -x11;

  x21 = 0;
  x22 = 0;

  Q3 = 0.06 * Ak * x12;
  Q4 = 0.06 * Ak * x11;

  F2 - F1 = Ak / 10 * (p3 - p4);

  h1 (t, v) = x0 + t * v;


  TYPE MD1;  % Modelltiefe 1 (Beruecksichtigung der Massentraegheit)

  x02' = x12;
  x12' = x22;

  x01 = -x02;
  x11 = -x12;

  x21 = -x22;
  x22 = IF ( ((x02 <= xmax / -2) AND (h2 (p3, p4, F2, F1, x12) <= 0)) OR
             ((x02 >= xmax /  2) AND (h2 (p3, p4, F2, F1, x12) >= 0)) )
          THEN 0;
          ELSE -h2 (p3, p4, F2, F1, x12);

  Q3 = 0.06 * Ak * x12;
  Q4 = 0.06 * Ak * x11;

  F2 - F1 = Ak / 10 * (p3 - p4);

  h2 (p3, p4, F2, F1, v) = (Ak / 10 * (p3 - p4) - F2 + F1 + d * v) / m;
END
```

B.5 Interne Darstellung in $^{\text{art}}$*deco*

Dieser Abschnitt enthält den Ausdruck der internen Präfix-Darstellung. Zur besseren Übersichtlichkeit erfolgte eine Nachbearbeitung der textuellen Formatierung durch den Autor dieser Diplomarbeit. Die Übersetzung wurde mit Einsatz der Operatorenreduzierung vorgenommen (siehe Abschnitt 4.1.2).

Es sei dem geneigten Leser freigestellt, ob er sich von der Äquivalenz der internen zur externen Darstellung überzeugen möchte.

```
% Lisp code generated by compiler version 0.9 on 03/27/95, 20:55:26

(
(= GATE1@position -GATE2@position)
(COND
  ( (>= (h1 t GATE2@velocity) (* xmax (EXPT 2 -1)))
    (= GATE2@position (* xmax (EXPT 2 -1))) )
  ( (<= (h1 t GATE2@velocity) (* -xmax (EXPT 2 -1)))
    (= GATE2@position (* -xmax (EXPT 2 -1))) )
  ( T
    (= GATE2@position (h1 t GATE2@velocity)) ))
(= GATE1@velocity -GATE2@velocity)
(= GATE2@velocity -GATE1@velocity)
(= GATE1@acceleration 0)
(= GATE2@acceleration 0)
(= GATE3@flow (* (* 0.06 Ak) GATE2@velocity))
(= GATE4@flow (* (* 0.06 Ak) GATE1@velocity))
(= (+ GATE2@force (* -1 GATE1@force))
   (* (* Ak (EXPT 10 -1)) (+ GATE3@pressure (* -1 GATE4@pressure))))
(= (h1 t v) (+ position (* t v)))
(DIFF x02 GATE2@velocity)
(DIFF x12 GATE2@acceleration)
(= GATE1@position -GATE2@position)
(= GATE1@velocity -GATE2@velocity)
(= GATE1@acceleration -GATE2@acceleration)
(COND
  ( (OR (AND (<= GATE2@position (* xmax (EXPT -2 -1)))
             (<= (h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force GATE2@velocity) 0))
        (AND (>= GATE2@position (* xmax (EXPT 2 -1)))
             (>= (h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force GATE2@velocity) 0)))
    (= GATE2@acceleration 0) )
  ( (NOT (OR (AND (<= GATE2@position (* xmax (EXPT -2 -1)))
                  (<= (h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force GATE2@velocity) 0))
             (AND (>= GATE2@position (* xmax (EXPT 2 -1)))
                  (>= (h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force GATE2@velocity) 0))))
    (= GATE2@acceleration -(h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force GATE2@velocity)) ))
(= GATE3@flow (* (* 0.06 Ak) GATE2@velocity))
(= GATE4@flow (* (* 0.06 Ak) GATE1@velocity))
(= (+ GATE2@force (* -1 GATE1@force))
   (* (* Ak (EXPT 10 -1)) (+ GATE3@pressure (* -1 GATE4@pressure))) )
(= (h2 GATE3@pressure GATE4@pressure GATE2@force GATE1@force v)
   (* (+ (+ (+ (* (* Ak (EXPT 10 -1))
                  (+ GATE3@pressure (* -1 GATE4@pressure)))
               (* -1 GATE2@force))
            GATE1@force)
         (* d v))
      (EXPT m -1)))
)
```

Abbildungsverzeichnis

2.1 Schematischer Aufbau von $^{\text{art}}$*deco* 7

2.2 Aufbau einer Komponente in $^{\text{art}}$*deco* (Gleichgangzylinder) 10

3.1 Beispiele des Kommentareinsatzes 22

3.2 Definition des einfachen Bezeichners 23

3.3 Definition des komplexen Bezeichners 23

3.4 Liste der Schlüsselwörter 24

3.5 Liste der Operatoren und Interpunktionszeichen 24

3.6 Definition der literalen Konstanten 25

3.7 Struktur der Verhaltensbeschreibung 26

3.8 Definition der Typdefinitionsanweisung 26

3.9 Definition der Gleichungsanweisung 26

3.10 Definition der Relationsanweisung 27

3.11 Definition der Selektionsanweisung 28

3.12 Definition der boolschen Anweisung 28

3.13 Definition der arithmetischen Ausdrücke 30

3.14 Definition der boolschen Ausdrücke 31

3.15 Definition der sonstigen Ausdrücke 31

4.1 Semantische Regeln für Bezeichner 37

4.2 Semantische Regeln für Anweisungen 38

4.3 Semantische Regel für Typdefinitionsanweisungen 38

4.4 Semantische Regel für Gleichungsanweisungen 39

4.5 Semantische Regeln für Relationsanweisungen 39

4.6 Semantische Regeln für Selektionsanweisungen - Teil 1 40

4.7 Semantische Regeln für Selektionsanweisungen - Teil 2a 41
4.8 Semantische Regeln für Selektionsanweisungen - Teil 2b 42
4.9 Semantische Regel für boolsche Anweisungen 42
4.10 Semantische Regeln für arithmetische Ausdrücke 44
4.11 Semantische Regeln für boolsche Ausdrücke 44
4.12 Semantische Regeln für sonstige Ausdrücke 45

5.1 Typische Bestandteile eines Übersetzers 48
5.2 Beispiel eines endlichen Automaten zur Erkennung von *number* 56
5.3 Zusammenspiel der wichtigsten Lex- und Yacc-Funktionen 62

6.1 Regeln mit Aktionen ohne Rückgabewert 69
6.2 Beispiele für Regelaktionen, die ein Token liefern 70
6.3 Beispiel für Regelaktion mit Attributwertermittlung 71
6.4 Beispiel für eine Komponentenbeschreibung in ^art^*deco* 73
6.5 Übersetzungsregel für *description* 79
6.6 Übersetzungsregel für einen numerischen Ausdruck 80
6.7 Übersetzungsregel für *type-statement* 81

Literaturverzeichnis

[ASU88a] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilerbau*, Teil 1. Addison-Wesley (Deutschland) GmbH, Bonn, 1988.

[ASU88b] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilerbau*, Teil 2. Addison-Wesley (Deutschland) GmbH, Bonn, 1988.

[DS92] Charles Donnelly und Richard Stallman. *Bison, the YACC-compatible Parser Generator*, Dezember 1992.

[Frü88] Hans W. Früchtenicht. *Technische Expertensysteme: Wissensrepräsentation und Schlußfolgerungsverfahren*, Kapitel *Zusammenfassende Darstellung*, Seiten 1–21. R. Oldenbourg Verlag GmbH, München Wien, 1988.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane und Willian M. Waite. *Eli: A Complete, Flexible Computer Construction System. Communications of the ACM*, 2: Seiten 121–132, Februar 1992.

[Her92] Helmut Herold. *Lex und Yacc: lexikalische und syntaktische Analyse.* Addison-Wesley (Deutschland) GmbH, Bonn, erste Auflage, 1992.

[HU90] John E. Hopcroft und Jeffry D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Addison-Wesley (Deutschland) GmbH, Bonn, zweite Auflage, 1990.

[Kas90] Uwe Kastens. *Übersetzerbau.* R. Oldenbourg Verlag GmbH, München Wien, 1990.

[Kip88] Jörg Kippe. *Komponentenorientierte Repräsentation technischer Systeme.* In Hans W. Früchtenicht (Hrsg.), *Technische Expertensysteme: Wissensrepräsentation und Schlußfolgerungsverfahren*, Seiten 155–226. R. Oldenbourg Verlag GmbH, München Wien, 1988.

[KS94] Hans Kleine Büning und Benno Stein. *Knowledge Based Support Within Configuration and Design Tasks.* Internes Papier, Universität-GH Paderborn, FB 17 – Praktische Informatik, 1994.

[Lü92] Joachim Lückel. *CAMeL Tools – Computer Aided Mechatronic Laboratory.* Universität-GH Paderborn, FB 10 – Automatisierungstechnik, 1992.

[Pup89] Frank Puppe. *Wissensrepräsentationen und Problemlösungsstrategien in Expertensystemen.* In Kai von Luck (Hrsg.), *Künstliche Intelligenz (7. Frühjahrsschule, KIFS-89)*, Seiten 167–186. Springer-Verlag, Berlin Heidelberg New York London Paris Tokyo, 1989.

[Sch91] Gabriele Schmiedel. *DIWA – Diagnosewerkzeug mit graphischer Wissensakquisition.* In H.-J. Bullinger (Hrsg.), *Expertensysteme in Produktion und Engineering (IAO-Forum)*, Seiten 121–132. Springer-Verlag, Berlin Heidelberg New York London Paris Tokyo Hong Kong Barcelona, April 1991.

[Ste95] Benno Stein. *Functional Models in Configuration Systems.* Dissertation, Universität-GH Paderborn, FB 17 – Praktische Informatik, 1995.

[Sue94] Michael Suermann. *Wissensbasierte Modellbildung und Simulation von hydraulischen Schaltkreisen.* Diplomarbeit, Universität-GH Paderborn, FB 17 – Praktische Informatik, Mai 1994.

[Wai94] William M. Waite. *Beyond LEX and YACC: How to Generate the Whole Compiler.* Internes Papier, University of Colorado, Department of Electrical and Computer Engineering, 1994.