# Ein Übersetzer-erzeugendes System auf der Basis attributierter Grammatiken

Zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
von der Fakultät für Informatik
der Universität (TH) Karlsruhe
genehmigte
**DISSERTATION**

von

**Dipl.-Inform. Uwe Kastens**

aus

**Bremen**

Tag der mündlichen Prüfung: 29. Juni 1976
Referent: Professor Dr. G. Goos
Korreferent: Professor Dr. P. Lockemann

# Inhalt

# 1 Einführung

In den letzten Jahren hat die Entwicklung von Übersetzer-erzeugenden Systemen aufgrund der Vielfalt von Programmiersprachen und Rechenanlagen immer mehr an Bedeutung gewonnen. Solche Systeme sollen die Entwicklung von Übersetzern vereinfachen und möglichst weitgehend automatisieren. Dieses Ziel ist für die Symbolentschlüsselung und die syntaktische Analyse von Programmiersprachen heute schon erreicht: Es sind eine Reihe von Verfahren bekannt und implementiert, mit denen aus lexikalischen und syntaktischen Definitionen praktikable Zerteilungsalgorithmen für Übersetzer erzeugt werden können. Um in ähnlich systematischer Weise auch komplexere semantische Eigenschaften heute gebräuchlicher Programmiersprachen zu behandeln, werden darüber hinaus Verfahren benötigt, mit denen aus semantischen Definitionen Übersetzer-Algorithmen zur semantischen Analyse und zur Code-Erzeugung generiert werden.

In dieser Arbeit stellen wir ein Übersetzer-erzeugendes System vor, das mit dem Ziel entwickelt wurde, aus Definitionen allgemein anwendbarer, höherer Programmiersprachen praktisch einsetzbare Übersetzer zu erzeugen, die mit relativ geringem Aufwand auf gebräuchlichen Rechenanlagen implementiert werden können.

Schwerpunkte der Arbeit sind die Entwicklung

- von Konzepten zur Beschreibung semantischer Spracheigenschaften,
- von Verfahren zur systematischen Transformation semantischer Definitionen in Übersetzer-Algorithmen und
- einer maschinenunabhängigen Grundlage für die Beschreibung dynamischer Semantik und für die Implementierung der erzeugten Übersetzer.

Wir haben eine Beschreibungssprache zur Formulierung vollständiger, maschinenunabhängiger Sprachdefinitionen entwickelt, die zugleich Eingabesprache für unser Übersetzer-erzeugendes System ist. Sie basiert auf der Beschreibungsmethode der attributierten Grammatiken ([Kn68]) und enthält Konzepte, die eine übersichtliche Formulierung von Spracheigenschaften erlauben. Attribute beschreiben die Eigenschaften der statischen und dynamischen Semantik und

sind den syntaktischen Sprachelementen zugeordnet. Die Definitionen aller Eigenschaften eines Sprachelements (Syntax, statische und dynamische Semantik) können zu einer Definitionseinheit zusammengefasst werden. Eine solche natürliche Strukturierung unterstützt die Lesbarkeit der Sprachdefinition. Da die Übersetzer-Algorithmen von unserem System selbständig aus der Sprachdefinition generiert werden, haben wir die Beschreibungssprache streng statisch und funktional entworfen. (Darin unterscheidet sie sich wesentlich von der in [Ga74] benutzten Sprache.)

Im Gegensatz zu Systemen, die Übersetzer aus algorithmischen Definitionen von Übersetzern generieren (sog. Compiler-Compiler) zeigen wir mit unserem System, wie statische Beschreibungen von Spracheigenschaften automatisch in Übersetzer-Algorithmen transformiert werden können. Der Entwurf einer Sprachdefinition wird dadurch erheblich vereinfacht. Es brauchen nur die Eigenschaften der Sprache definiert zu werden und nicht die Übersetzer-Algorithmen, mit denen sie überprüft und ausgewertet werden. Die Eingabe, aus der das System den Übersetzer generiert, ist deshalb zugleich eine formale Beschreibung der Sprache, die auch für den menschlichen Leser verständlich ist.

In unser Übersetzer-erzeugendes System sind Verfahren integriert, die die semantischen Definitionen auf Vollständigkeit und Konsistenz überprüfen, die semantischen Kontextabhängigkeiten analysieren und daraus systematisch die Übersetzer-Algorithmen generieren. Die Sprachdefinition wird nach den Kriterien überprüft, die auch bei der Übersetzung von Programmen höherer Programmiersprachen angewendet werden. Darüber hinaus wird festgestellt, ob die semantischen Eigenschaften vollständig und zyklenfrei definiert sind, und ob die dynamische Semantik im Sinne der Zielsprache strukturell korrekt formuliert ist.

Wir gehen davon aus, dass von einem Zerteilungsalgorithmus die syntaktische Struktur des zu übersetzenden Programms erkannt und durch einen Strukturbaum dargestellt wird. Die semantische Analyse ergänzt den Strukturbaum durch Zufügen von Attributen. Zur Bestimmung der Attribute ist es im allgemeinen erforderlich, den Strukturbaum (ganz oder teilweise) mehrfach zu durchlaufen: Das Übersetzer-erzeugende System ermittelt aus den Definitionen der semantischen Kontextabhängigkeiten automatisch die Reihenfolge, in der die Knoten des Baumes besucht werden, und die Vorschriften

zur Auswertung der Attribute ("Besuchssequenzen"). Daraus werden Übersetzer-Tabellen generiert, die einen einfachen Algorithmus zur semantischen Analyse steuern. Dieses Verfahren ist allgemeiner und breiter anwendbar als die z.B. in [Ga74], [Le75] und [Bo76] beschriebenen. Der erzeugte Übersetzer ist besser an die Erfordernisse der definierten Sprache angepasst.

Die dynamische Semantik der Programmiersprache wird mit den Begriffen und Eigenschaften einer abstrakten Maschine definiert, die ebenfalls als Attribute in das Beschreibungsmodell integriert sind. Wir betrachten die Code-Erzeugung als eine weitere Ergänzung des Strukturbaumes durch spezielle Attribute. Das Verfahren zur Bestimmung der Attribute ist deshalb auch für die Code-Generierung und quellsprachbezogene Optimierung anwendbar.

Wir haben beim Entwurf der abstrakten Maschine folgende Ziele verfolgt:

- Die dynamischen Eigenschaften allgemein anwendbarer, höherer Programmiersprachen zur strukturierten Programmierung sollen einfach auf die Eigenschaften der abstrakten Maschine abgebildet werden können.

- Die abstrakte Maschine soll mit relativ geringem Aufwand auf den heute gebräuchlichen Rechenanlagen implementierbar sein. Die von dem System generierten Übersetzer erzeugen Code für die abstrakte Maschine und sind selbst Programme, die auf der abstrakten Maschine laufen. Sie können deshalb leicht auf verschiedenen Maschinen verfügbar gemacht werden.

Für unser Übersetzer-erzeugendes System ergeben sich verschiedene Anwendungsgebiete: Zu vorhandenen Sprachdefinitionen können mit relativ geringem Aufwand Übersetzer für verschiedene Rechenanlagen erzeugt werden. Der Entwurf und die Weiterentwicklung von Programmiersprachen werden durch das System vereinfacht; denn es prüft die Vollständigkeit und Konsistenz der Sprachdefinition und macht den Zusammenhang zwischen Spracheigenschaften und Übersetzer-Algorithmen schon zum Zeitpunkt des Sprachentwurfs deutlich.

Die Komplexität der Sprachen, die von dem System verarbeitet werden können, ist weder durch die Beschreibungsmethode noch durch die Verfahren zur Erzeugung der Übersetzer

prinzipiell beschränkt. Das System ist so konzipiert, dass man praktisch einsetzbare Übersetzer für Sprachen der Komplexität von ALGOL 60 oder PASCAL mit vertretbarem Speicher- und Zeit-Aufwand generieren kann.

Wesentliche Ergebnisse unserer Arbeit sind

- Verfahren, mit denen aus attributierten Grammatiken systematisch tabellengesteuerte Übersetzer-Algorithmen zur semantischen Analyse und Synthese gewonnen werden können.

- eine Beschreibungssprache zur Formulierung von Sprachdefinitionen, die automatisch verarbeitet werden können und auch für den menschlichen Leser verständlich sind.

- eine abstrakte Maschine, auf deren Konzepte die dynamische Semantik einer grossen Klasse von Programmiersprachen abgebildet werden kann, und die mit relativ geringem Aufwand auf gebräuchlichen Rechenanlagen implementierbar ist.

Unser Übersetzer-erzeugendes System ist auf einer Rechenanlage vom Typ Burroughs B6700 zum grossen Teil implementiert. Einige Ergebnisse sind im Anhang B zusammengestellt.

Im Kapitel 2 dieser Arbeit geben wir einen Überblick über Methoden zur Definition von Programmiersprachen, über die Verwendung abstrakter Maschinen in Übersetzern und über Verfahren zur Transformation attributierter Grammatiken in Übersetzer. Kapitel 3 enthält eine Definition der Eingabesprache für das Übersetzer-erzeugende System, in der die Sprachdefinitionen formuliert werden. In Kapitel 4 werden die wesentlichen Verfahren beschrieben, mit denen aus solchen Sprachdefinitionen Übersetzer generiert werden. Die abstrakte Maschine wird in Kapitel 5 definiert.

Herr Prof. Dr. Gerhard Goos hat durch wertvolle Anregungen und fruchtbare Diskussionen wesentlich zum Zustandekommen dieser Arbeit beigetragen. Ihm gilt mein besonderer Dank.

Herrn Prof. Dr. Peter Lockemann danke ich für die freundliche Übernahme des Korreferats und die kritische Durchsicht der Arbeit.

# 2 Überblick

Die Techniken, mit denen in Übersetzern Programme höherer Programmiersprachen analysiert und in Programme einer Zielsprache transformiert werden, sind heute bekannt und weitgehend systematisiert - sieht man von der maschinenabhängigen Code-Erzeugung ab. Es liegt deshalb nahe, mit Hilfe Übersetzer-erzeugender Systeme die Implementierung von Übersetzern so zu vereinfachen, dass aus Beschreibungen der Eigenschaften von Programmiersprachen automatisch die zu ihrer Übersetzung notwendigen Algorithmen generiert oder Standard-Algorithmen durch Parametrisierung (Tabellensteuerung) an die Erfordernisse der jeweiligen Programmiersprache angepasst werden.

Solche Systeme können eingesetzt werden, um Programmiersprachen, deren Entwicklung abgeschlossen ist, auf verschiedenen Rechenanlagen verfügbar zu machen oder um die Entwicklung bzw. die Weiterentwicklung von Sprachen zu vereinfachen. Im letzteren Fall wird durch den Einsatz eines Übersetzer-erzeugenden Systems eine schnelle Überprüfung der Konsistenz, Vollständigkeit und Übersetzbarkeit der definierten Spracheigenschaften möglich; der für die Übersetzung nötige Aufwand wird deutlich gemacht, und die Sprache kann schon praktisch erprobt werden, bevor ihre Eigenschaften endgültig festgeschrieben sind.

Die Eingabesprache eines Übersetzer-erzeugenden Systems ist zugleich auch eine Beschreibungssprache, in der Sprachdefinitionen formuliert werden können. In Abschnitt 2.1 zeigen wir die Anforderungen an eine solche Beschreibungssprache auf, diskutieren die Eignung einiger bekannter Definitionsmethoden unter diesem Gesichtspunkt und begründen, weshalb wir der Eingabesprache unseres Systems (siehe Kapitel 3) die Methode der attributierten Grammatiken zugrunde gelegt haben.

Die Transformation von Programmen einer höheren Programmiersprache in Programme einer Maschinensprache kann in mehrere Schritte zerlegt werden. Wir gehen davon aus, dass im ersten Schritt die Begriffe der Quellsprache auf einfachere Funktionen und Eigenschaften einer abstrakten Maschine abgebildet werden. Ihre Konzepte sind die Grundbegriffe für die Beschreibung der dynamischen Semantik der Sprache in der Sprachdefinition und definieren die Zielsprache der gene-

rierten Übersetzer. Die weiteren Abbildungsschritte werden bei der Implementierung der abstrakten Maschine auf einer Rechenanlage durchgeführt. Dies ist für alle so erzeugten Übersetzer nur einmal für eine Rechenanlage nötig (UNCOL-Konzept, [St61]). In Abschnitt 2.2 diskutieren wir die Ziele für den Entwurf der abstrakten Maschine AMICO, die wir unserem System zugrunde legen. Sie wird in Kapitel 5 definiert.

In Abschnitt 2.3 geben wir einen Überblick über die wesentlichen Aufgaben, die ein Übersetzer-erzeugendes System zur Transformation von attributierten Grammatiken in Übersetzer-Algorithmen erfüllen muss, und stellen unsere Methoden anderen gegenüber. (Die Entwicklung Übersetzer-erzeugender Systeme bis zum Stand von 1967 beschreiben Feldman und Gries in [FG68] umfassend.)

## 2.1 Sprachdefinitionen

In diesem Abschnitt stellen wir Anforderungen auf, denen Sprachdefinitionen genügen müssen, die von einem Übersetzer-erzeugenden System verarbeitet werden, und diskutieren verschiedene Methoden zur Definition von Sprachen (2-Ebenen Grammatiken, attributierte Grammatiken, Transformations-Grammatiken, die "Wiener Methode" und die Methode der axiomatischen Definition).

### 2.1.1 Inhalt von Sprachdefinitionen

Eine Sprachdefinition gibt die Menge der Sätze, die der Sprache angehören, präzise an. Häufig wird eine Obermenge der Sprache durch eine kontext-freie Grammatik definiert. Kontextbedingungen (z.B. Aussagen über Gültigkeitsbereiche oder Typ-Abhängigkeiten) schränken dann die Menge der Sätze weiter ein (z.B. in verbaler Form in der Definition von ALGOL 68 [Na68]). Dies führt zu einer Unterscheidung von syntaktischen Eigenschaften einer Sprache (Struktur und Notation der Sätze) und den Eigenschaften der statischen Semantik. Es hängt jedoch weitgehend von der Beschreibungsmethode und von der Sprachdefinition selbst ab, wie gross

die Differenz zwischen der syntaktisch definierten Obermenge und der Sprachmenge ist.

Die Bedeutung der Sätze der Sprache wird in einem abstrakten Modell beschrieben. Die Definitionen der dynamischen Semantik bilden Sprachelemente auf Eigenschaften und Funktionen des Modells ab, deren Bedeutung dem Beschreibungsmodell als Pragmatik zugrunde liegt. Darüber hinaus sind im allgemeinen auch einige Kontextabhängigkeiten der dynamischen Semantik zuzurechnen. Sie definieren dann eine Untermenge der Sprache, die Menge der ausführbaren Programme. Es ist eine Frage des Stils der Sprachdefinition, ob auch nicht-ausführbare Programme zur Sprachmenge gehören oder nicht, und ob ihre Wirkung beschrieben wird oder undefiniert bleibt. In einigen Fällen wird es nicht durch die Sprache, sondern durch die Sprachdefinition (oder den Übersetzer) bestimmt, ob gewisse Kontextabhängigkeiten der statischen oder der dynamischen Semantik zuzurechnen sind.

Wir unterscheiden drei prinzipiell verschiedene Methoden zur Definition der dynamischen Semantik: operationale, denotationale und axiomatische Definitionen. Operationale Definitionen bilden die Elemente der definierten Sprache auf Zustandsübergänge im Beschreibungsmodell ab. Passt man das Modell als eine abstrakte Maschine auf, so wird auf diese Weise ein Interpretierer für die Sprache definiert. Denotationale Definitionen bilden die Elemente der Sprache (Quellsprache) auf Funktionen ab, die Zustandsübergänge im Beschreibungsmodell bewirken (Zielsprache einer abstrakten Maschine). Auf diese Weise wird die Transformation beschrieben, die ein Übersetzer für die abstrakte Maschine durchführt. Axiomatische Definitionen beruhen auf einem logischen Kalkül. Die dynamische Semantik wird durch Aussagen über die Zustände des Beschreibungsmodells vor und nach der Ausführung eines Sprachelements (relationale Methode) oder über Eigenschaften dieser Zustände (deduktive Methode) definiert (siehe [HL74]).

### 2.1.2 Anforderungen an Definitionsmethoden

Sollen Sprachdefinitionen von einem Übersetzer-erzeugenden System automatisch verarbeitet werden, so müssen sie in einer Beschreibungssprache formuliert werden, die folgende Kriterien erfüllt:

Die Beschreibungssprache muss formal definiert sein, sie soll möglichst weitgehend automatische Prüfungen der Konsistenz und Vollständigkeit der Sprachdefinitionen erlauben, und es müssen Übersetzer-Algorithmen generierbar sein, die die syntaktische Analyse (Zerteilung) und die semantische Analyse und Synthese gemäss der Sprachdefinition durchführen.

Die Beschreibungssprache ist die Schnittstelle des Übersetzer-erzeugenden Systems zu seinen Benutzern. Sie muss deshalb die Formulierung gut verständlicher und strukturierter Sprachdefinitionen unterstützen. Wird das System zur (Weiter-) Entwicklung von Sprachen eingesetzt, so steht zugleich mit einem einsetzbaren Übersetzer auch eine vollständige, formalisierte Definition der Sprache zur Verfügung. Sie kann zur Beurteilung der Sprache, zur Erstellung von Lehrunterlagen und als Sprachbeschreibung für den fortgeschrittenen Benutzer dienen, falls sie gut lesbar formuliert ist.

Für die gute Lesbarkeit und Verständlichkeit einer Sprachdefinition ist es wichtig, dass inhaltlich zusammengehörige Definitionen auch als strukturelle Einheiten formuliert sind, dass man die syntaktische Struktur der Sprache auch ohne Betrachten der Kontextabhängigkeiten erkennen kann und dass die Regeln der Beschreibungssprache leicht einsehbar sind. Im allgemeinen sind algorithmische Definitionen (z.B. in Form eines Übersetzers oder Interpretierers) schwerer zu verstehen als statische.

### 2.1.3 Attributierte Grammatiken

In dem Modell der attributierten Grammatiken, das Knuth in [Kn68] vorstellt, wird die syntaktische Struktur der Sätze der Sprache durch eine kontext-freie Grammatik definiert. Die Eigenschaften der Sprachelemente werden durch Attribute beschrieben, die den Nichtterminal-Symbolen zugeordnet sind. Sie werden abhängig von dem Kontext, in dem das Sprachelement steht, durch semantische Regeln bestimmt, die den syntaktischen Regeln zugeordnet sind. Diese Methode erlaubt die Beschreibung beliebig komplexer Kontextabhängigkeiten.

Knuth gibt in [Kn68] Bedingungen an, unter denen eine attributierte Grammatik "semantisch wohldefiniert" ist. Sie sind algorithmisch prüfbar und schränken die Mächtigkeit des Beschreibungsmodells nicht ein.

Die dynamische Semantik kann operational oder denotational formuliert werden. Bestimmte Attribute von Nichtterminal-Symbolen beschreiben entweder die Wirkung des jeweiligen Sprachelements oder eine Sequenz von Funktionen einer abstrakten Maschine, die diese Wirkung erbringt. Knuth gibt in [Kn68] und [Kn69] Beispiele für operationale Definitionen.

Attributierte Grammatiken eignen sich gut zur automatischen Verarbeitung. Mit Hilfe graphentheoretischer Verfahren können die semantischen Definitionen auf ihre Konsistenz und Vollständigkeit überprüft (siehe [Kn68] und Abschnitt 4.2) und in Übersetzer-Algorithmen transformiert werden. Diese Verfahren sind unabhängig von den Methoden, die zur Generierung der Zerteilungs-Algorithmen angewendet werden.

In unserem System ist das Modell der attributierten Grammatiken in eine funktionale, statische Beschreibungssprache integriert. Die Attribute werden als typisierte Objekte aufgefasst, deren Werte durch Ausdrücke über Attributen bestimmt werden (semantische Regeln). Die Typisierung der Attribute ermöglicht weitere Konsistenz-Prüfungen. Durch die Verwendung zusammengesetzter Attribute können Sprachdefinitionen einfach und übersichtlich formuliert werden.

### 2.1.4 2-Ebenen-Grammatiken

Mit Hilfe von 2-Ebenen-Grammatiken (z.B. van Wijngaarden- oder Affix-Grammatiken) ist es möglich, auch Kontextabhängigkeiten syntaktisch präzise zu definieren. Van Wijngaarden-Grammatiken enthalten zwei Regelmengen: die Meta-Regeln einer kontext-freien Grammatik und sogenannte Hyper-Regeln. Durch konsistentes Einsetzen von terminalen Zeichenreihen der Meta-Grammatik in die Hyper-Regeln werden die Regeln einer im allgemeinen nicht endlichen, kontext-freien Grammatik gebildet, die die Sprachmenge definiert [vW68].

Im ersten Bericht zu ALGOL 68 [vW68] wurden die Kontextabhängigkeiten der Sprache zum grossen Teil (z.B. Arten, Artanpassung), im revidierten Bericht [vW75] komplett (auch z.B. Gültigkeitsbereiche) syntaktisch definiert. Zu diesem Zweck enthält die Grammatik Regeln, die die Auswertung von "Prädikaten" dadurch beschreiben, dass sie abhängig vom jeweiligen Kontext entweder in eine Sackgasse führen oder auf das leere Wort abgeleitet werden.

Van Wijngaarden-Grammatiken genügen den in 2.1.2 genannten Kriterien der Transformierbarkeit nicht. Das allgemeine Wortproblem ist für sie nicht entscheidbar. Man kann also im allgemeinen algorithmisch nicht entscheiden, ob eine gegebene Zeichenreihe einer Sprache angehört, die durch eine van Wijngaarden-Grammatik definiert ist. Es sind noch keine nicht-trivialen Einschränkungen dieses Beschreibungsmittels bekannt, die eine Transformation in praktikable Analyse-Algorithmen erlauben würden (siehe auch [De75]).

In Affix-Grammatiken [Ko71a] werden den Nichtterminal-Symbolen einer kontext-freien Grammatik "Affixe" zugeordnet, die durch eine im allgemeinen kontext-freie Grammatik definiert werden. Diese entspricht der Meta-Grammatik eines van Wijngaarden-Systems. Anstelle der Vorschrift des konsistenten Einsetzens werden in wohldefinierten Affix-Grammatiken strengere Ableitungsvorschriften für die Affixe gefordert, um zu erreichen, dass die Affixe bei einer top-down-Zerteilung nach der Methode des rekursiven Abstiegs bestimmt werden können. Dadurch wird jedoch die mögliche Komplexität der Kontextabhängigkeiten soweit eingeschränkt, dass z.B. die Gültigkeitsbereichsregeln vieler Programmiersprachen (wie ALGOL 68) nicht durch eine einzige Grammatik definiert werden können. Ausserdem sind die Bedingungen für die Wohl-

definiertheit im allgemeinen nicht entscheidbar und deshalb auch nicht automatisch prüfbar.

Die Affixe bzw. die Nichtterminal-Symbole der Meta-Grammatik sind vergleichbar mit den Attributen der attributierten Grammatiken. Sie beschreiben Eigenschaften und Kontextabhängigkeiten von Sprachelementen. Sie werden jedoch nach unterschiedlichen Methoden bestimmt: in 2-Ebenen-Grammatiken durch Regeln der Meta-Ebene, in attributierten Grammatiken durch Ausdrücke in der Beschreibungssprache über Attributen von Nichtterminal-Symbolen, die in einem gemeinsamen Kontext stehen.

2-Ebenen-Grammatiken sind im allgemeinen schwer verständlich, da das Zusammenspiel der beiden Regelmengen relativ komplex ist. Eine Strukturierung der Definitionen nach ihrer inhaltlichen Zusammengehörigkeit ist zwar möglich (durch geeignetes Zusammenfassen von Hyper- und Meta-Regeln bzw. Syntax- und Affix-Regeln); sie wird durch die Beschreibungsmethode jedoch nicht unterstützt.

### 2.1.5 Wiener Methode

Anders als die oben diskutierten Methoden baut die Wiener Methode ("Vienna Definition Language", VDL [We72]) auf der formalen Definition von Sprachen durch einen Interpretierer auf. Die Struktur der Sätze einer Sprache wird durch eine "abstrakte Syntax" definiert. Die Notation der Sätze und die Reihenfolge von Unterstrukturen eines Sprachelements ("konkrete Syntax") ist in einer Sprachbeschreibung in VDL nicht definiert. Die Kontextabhängigkeiten von Sprachen werden zusammen mit den Wirkungen der Sprachelemente operational definiert. Eine Sprachdefinition in VDL besteht aus Regeln zum Aufbau einer Baumstruktur, die als Komponente den abstrakten Programmbaum zu einem Satz der Sprache enthält. Die Wirkung der Sätze wird durch algorithmische Transformationen des Baumes beschrieben.

Obwohl die Wiener Methode nicht mit dem Ziel der automatischen Verarbeitung definiert wurde, ist eine solche Anwendung durchaus denkbar. Aus Sprachdefinitionen in VDL würde man jedoch keine Übersetzer, sondern Interpretierer generieren.

Sprachdefinitionen sind in dieser Form relativ schwer zu verstehen, da es nötig ist, die Beschreibung algorithmisch auf einen Satz der Sprache anzuwenden, um Aussagen über Spracheigenschaften zu erhalten.

### 2.1.6 Transformations-Grammatiken

Transformations-Grammatiken, wie sie z.B. in [DR73] und [Sch76] beschrieben werden, definieren die syntaktisch korrekte Obermenge LO einer Sprache L durch eine kontextfreie Grammatik. Mit mehreren, nacheinander angewendeten Transformations-Grammatiken werden die Läufe eines Übersetzers beschrieben, der die Sätze der Sprache L in die Sätze einer Zielsprache LZ transformiert, und dabei die Kontextabhängigkeiten berücksichtigt, die LO auf L einschränken. Die Differenz zwischen den Sprachen LO und L wird hier indirekt definiert: Ein Satz aus LO gehört dann zu L, wenn er durch die angegebenen Regeln in einen Satz von LZ transformiert wird. Dies ist für den Leser einer solchen Sprachdefinition relativ schwer nachzuprüfen. Das Prinzip schliesst eine denotationale Definition der dynamischen Semantik ein.

Transformations-Grammatiken können als formale Beschreibung der Übersetzung verstanden werden. Sie sind deshalb systematisch in Algorithmen zur Manipulation von Bäumen übertragbar. Die formale Korrektheit einer solchen Grammatik ist auf das Halteproblem der Transformations-Algorithmen zurückzuführen, das im allgemeinen jedoch nicht entscheidbar ist.

### 2.1.7 Axiomatische Definition

Die Methode der axiomatischen Definition von Programmiersprachen wurde von Hoare in [Ho69] vorgestellt und in [HW73] für die Definition von PASCAL angewendet. Die semantischen Eigenschaften - Kontextabhängigkeiten und Wirkungen von Sprachelementen - der in diesem Modell definierten Sprachen werden in einem logischen Kalkül formuliert. Die Wirkung von Sprachelementen wird nicht direkt beschrieben,

sondern durch Aussagen über Zustände des Beschreibungsmodells vor und nach Ausführung eines Sprachelements. Diese Definitionsmethode eignet sich gut zum formalen Beweisen der Korrektheit von Programmen. Da die Axiomatik nur die Semantik sinnvoller Programme beschreibt, kann sie keine vollständige Grundlage für die Erzeugung von Übersetzern sein.

Es ist prinzipiell möglich, die Methode so zu erweitern, dass auch die syntaktische Struktur der Sprache definiert werden kann. In [Po76] wird die höhere Programmiersprache LEX [Go75a] axiomatisch definiert und damit demonstriert, dass die Methode auch für komplexere Spracheigenschaften, als sie PASCAL enthält, anwendbar ist.

Es sind heute noch keine Methoden bekannt, mit denen aus axiomatische Sprachdefinitionen Übersetzer generiert werden können.

## 2.2 Abstrakte Maschinen

Algorithmen können auf verschiedenen Abstraktions-Ebenen beschrieben werden. Eine abstrakte Maschine definiert Begriffe (Funktionen und Datenobjekte), die auf einer Abstraktions-Ebene als elementar anzusehen sind. Die Funktionen und Datenobjekte einer Abstraktions-Ebene werden auf die Begriffe der darunter liegenden Ebene zurückgeführt. Jede abstrakte Maschine definiert eine Sprache, in der die Begriffe der auf ihr aufbauenden Maschine formuliert werden.

Einer Hierarchie von abstrakten Maschinen kann man deshalb eine Hierarchie von Sprachen zuordnen. Übersetzer bilden die in einer Sprache formulierten Algorithmen in die Sprache der darunter liegenden Schicht ab. (Im Fall geschichteter Programmsysteme sind die Sprachen zu den einzelnen abstrakten Maschinen im allgemeinen in eine gemeinsame Implementierungssprache eingebettet, deren Definitions- und Aufruf-Konzepte die Funktionen der abstrakten Maschinen aufeinander abbilden.)

Die Übersetzung höherer Programmiersprachen ist eine Abbildung komplexer Eigenschaften der zu der Sprache gehörigen abstrakten Maschine auf die (bzw. eine Teilmenge der) Funktionen einer Rechenanlage und des auf ihr laufenden

Betriebssystems. Die Abbildung erfolgt im allgemeinen in mehreren Schritten (Zwischensprachen), die jeweils durch eine abstrakte Maschine definiert sind.

höhere Programmiersprache

.
.
.

Zwischensprache

.
.
.

Maschinen- und Betriebssystem-Funktionen

Die Übertragbarkeit des Übersetzers wird verbessert, wenn man die Zwischenschritte dieser Transformation so wählt, dass mit jeder Abbildung die Eigenschaften der zugrunde liegenden Rechenanlage stärker berücksichtigt werden.

Wird die dynamische Semantik in der Sprachdefinition mit den Begriffen einer maschinenunabhängigen abstrakten Maschine beschrieben, so ist damit zugleich der erste Transformationsschritt der Übersetzung definiert: die Erzeugung maschinenunabhängigen Codes, der durch weitere Transformationen auf verschiedenen Rechenanlagen implementiert werden kann. (Dieses Konzept liegt auch der Sprache UNCOL zugrunde [St61], die jedoch nicht zum Einsatz gebracht wurde.)

Im folgenden geben wir einen Überblick über unsere abstrakte Maschine AMICO. Wir stellen ihr die abstrakte Maschine JANUS gegenüber, die zeitlich parallel mit ähnlicher Zielsetzung entwickelt wurde. Die Übertragbarkeit von Übersetzern für die Sprachen PASCAL [No75] und BCPL [Ri69] wurde ebenfalls durch abstrakte Maschinen erzielt. Sie sind für eine Anwendung in einem Übersetzer-erzeugenden System weniger geeignet, da ihre Eigenschaften speziell auf diese Sprachen ausgerichtet sind.

### 2.2.1 AMICO

Die abstrakte Maschine AMICO (Abstract Machine for Implementation of COmpilers) definiert Grundfunktionen, auf die die dynamische Semantik einer grossen Klasse höherer Programmiersprachen einfach abgebildet werden kann. Die durch AMICO definierte Sprache ist in die Eingabesprache unseres Übersetzer-erzeugenden Systems integriert. Sie definiert eine maschinenunabhängige Zwischensprache, für die die generierten Übersetzer Code erzeugen. Darüber hinaus eignet sie sich als Implementierungssprache für das Übersetzer-erzeugende System selbst und die von ihm generierten Übersetzer.

Die Konzepte von AMICO entsprechen den elementaren Konzepten zur Strukturierung von Programmen und Daten in höheren Programmiersprachen. AMICO definiert pragmatisch Operationen auf Werten zu Datentypen, soweit sie in allgemein anwendbaren Programmiersprachen gebräuchlich sind. Diese Auswahl fördert die Übersichtlichkeit von Beschreibungen der Sprachen aus dieser Klasse. Es wird in Kauf genommen, dass Sprachen für spezielle Anwendungsgebiete wie Text- oder Listenverarbeitung nur mit grösserem Aufwand beschrieben werden können. AMICO ist nicht geeignet zur Definition von Sprachen, die in grösserem Umfang von den Funktionen eines zugrunde liegenden Betriebssystems Gebrauch machen (z.B. Sprachen zur Betriebssteuerung, spezielle E/A-Sprachen oder Dialogsprachen).

Die Speicherorganisation von AMICO baut auf einem allgemeinen Verbundkonzept auf. Sie ist an die unterschiedlichen Anforderungen verschiedener Programmiersprachen anpassbar und schliesst die Beschreibung des Umgebungsmodells ein. Einfache und zusammengesetzte Datentypen werden als Vorschriften (Schablonen) zur Interpretation von Speicherbereichen verstanden, mit denen die Zugriffsfunktionen spezifiziert werden. Die Grundfunktionen zur Ablaufsteuerung in höheren Programmiersprachen (Verzweigung, Fallunterscheidung, Schleife, Aufruf, Ausgang) sind auch in AMICO elementar. Dateien und ihre Operationen werden nur für einige Standard-Anwendungen definiert, soweit sie in den Sprachen der oben angegebenen Klasse gebräuchlich sind.

Die Implementierung von AMICO auf Rechenanlagen unterschiedlichen Typs wird dadurch erleichtert, dass zur Codie-

rung von Datenobjekten keine und zur Abbildung der Speicherstrukturen nur wenige Annahmen gemacht werden. Informationen, die bei der Übersetzung anfallen und für die maschinenabhängige Code-Erzeugung und -Optimierung ausgenutzt werden können, bleiben im AMICO-Programm erhalten. Dies wird z.B. durch weitgehende Differenzierung von Datentypen und Zugriffsfunktionen erreicht. In welchem Masse diese Informationen ausgenutzt werden, hängt von dem Aufwand ab, der für die Transformation der AMICO-Programme investiert wird.

Die Operationenfolgen der AMICO-Programme werden in Postfix-Form notiert. Sie können einfach aus dem Strukturbaum des übersetzten Programmes generiert werden. Zu den Operatoren werden die Typen aller ihrer Operanden und Ergebnisse explizit angegeben, so dass sie weitgehend unabhängig vom Kontext transformiert werden können.

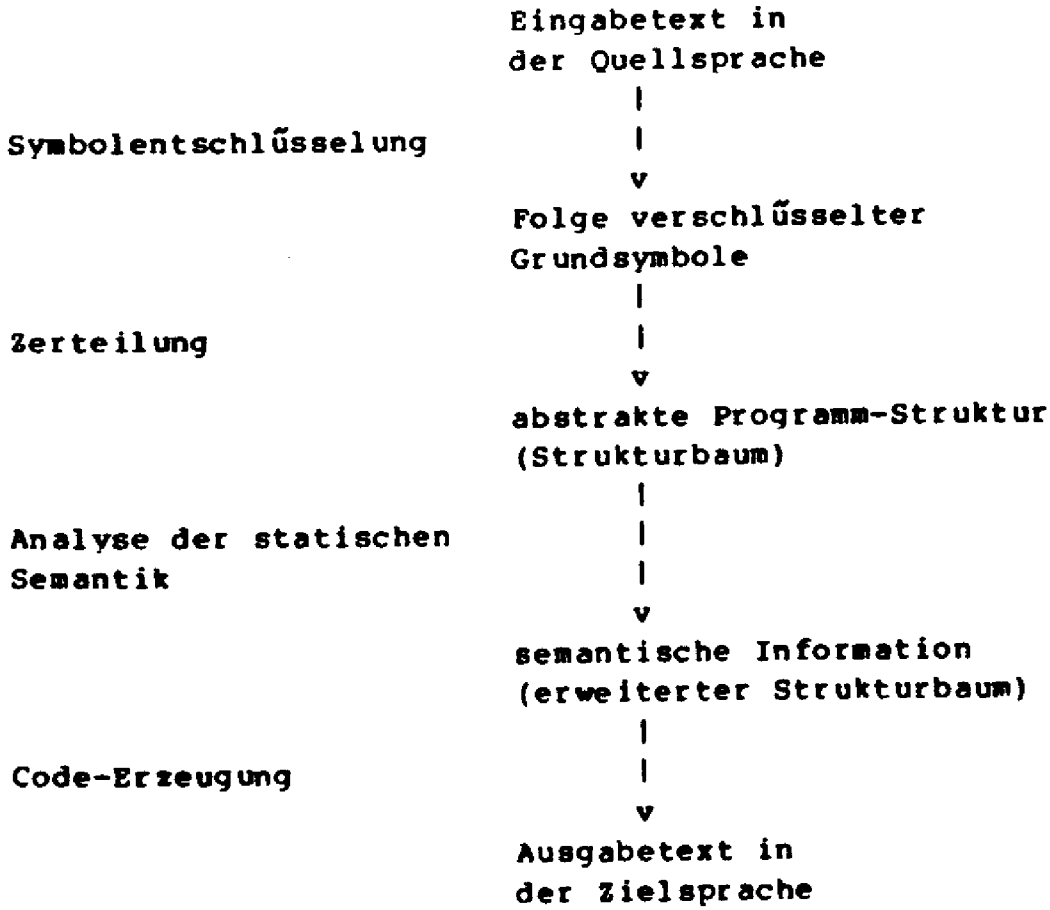
### 2.2.2 JANUS

JANUS [WH75] wurde als maschinenunabhängige Zwischensprache entwickelt, um die Übertragung von Übersetzern zu vereinfachen und das UNCOL-Problem zu lösen. Im Unterschied zu AMICO wurde nicht angestrebt, mit JANUS Begriffe bereitzustellen, mit denen die dynamische Semantik höherer Programmiersprachen einfach formuliert werden kann. Die Grundfunktionen von JANUS sind deshalb stärker auf die Eigenschaften konkreter Rechenanlagen als auf Konzepte höherer Programmiersprachen ausgerichtet. Der Aufwand für die Implementierung von JANUS ist gegenüber AMICO dadurch zwar geringer, für die Abbildung der Spracheigenschaften auf die Zwischensprache jedoch grösser.

JANUS ist eine erweiterbare Zwischensprache, die eine Familie von abstrakten Maschinen mit gleicher Struktur und gleichem Befehlsformat definiert. Die JANUS-Operationen können von einem Makro-Prozessor (z.B. STAGE2 [Wa70]) einfach in Maschinenbefehle transformiert werden. Sie wirken auf einen Akkumulator, ein Indexregister, einen Keller für Zwischenergebnisse und einen linearen Speicher. Konzepte zur dynamischen Speicherorganisation und zur Behandlung von Verbunden, die mit denen von AMICO vergleichbar wären, enthält die Kernsprache von JANUS nicht.

## 2.3 Transformation von Sprachdefinitionen in Übersetzer

Die Übersetzung eines Programms einer höherer Programmiersprachen ist im allgemeinen in folgende Schritte gegliedert:




Es können noch Algorithmen zur quell- und zielsprachenbezogenen Optimierung hinzukommen. Die Teilaufgaben werden entweder ineinander verzahnt oder nacheinander in verschiedenen Läufen erledigt. Die abstrakte Programmstruktur wird durch einen Strukturbaum dargestellt, der durch die semantische Information (häufig in Tabellen-Form) erweitert und dann in ein Programm der Zielsprache transformiert wird.

Im folgenden beschreiben wir Methoden, die man in einem Übersetzer-erzeugenden System anwenden kann, um systematisch aus Sprachdefinitionen auf der Basis attributierter Grammatiken Algorithmen zur Lösung der oben angegebenen Übersetzer-Aufgaben zu erzeugen. Verfahren zur Generierung von Symbolentschlüsselern aus regulären Ausdrücken sind hinrei-

chend bekannt (siehe [Gr71]) und werden hier nicht weiter diskutiert. Es gibt ebenfalls ausreichend Methoden, um systematisch Zerteiler zu erzeugen. Wir beschränken uns hier auf ihre Aufzählung, um sie gegenüber den Methoden zur Behandlung semantischer Spracheigenschaften abzugrenzen. In den Abschnitten 2.3.2 bis 2.3.5 geben wir einen Überblick über unsere Verfahren zur Transformation semantischer Definitionen und stellen sie anderen Verfahren zur Verarbeitung attributierter Grammatiken gegenüber.

### 2.3.1 Zerteiler-Generatoren

Die Algorithmen zur syntaktischen Analyse unterscheiden sich in ihrer Struktur und der Zerteilungsrichtung: Der Ablauf kann durch die Programmstruktur bestimmt werden (rekursiver Abstieg) oder tabellengesteuert sein (Präzedenz-, LL- und LR-Verfahren); die Zerteilung kann zielbezogen (rekursiver Abstieg, LL-Verfahren) oder quellbezogen (Präzedenz-, LR-Verfahren) sein. ([AU72] enthält umfassende Beschreibungen von Zerteilungsverfahren.) Die tabellengesteuerten Methoden eignen sich besonders gut für die automatische Generierung, da jeweils nur die Tabellen erzeugt werden müssen; die Algorithmen, die sie interpretieren, sind universell verwendbar.

Ältere Übersetzer-erzeugende Systeme generieren im wesentlichen Zerteilungs-Algorithmen, die entweder direkt die Erzeugung von Maschinen- oder Zwischen-Code steuern (z.B. PSL [Fe66]) oder komplexere Funktionen zur kontextabhängigen Code-Erzeugung anstossen (z.B. SMG [Go69], XPL [KHW70], Compiler-Compiler von Brooker und Morris [Ro64]). Komplexere semantische Eigenschaften der verarbeiteten Sprachen können von Systemen dieser Art nur mit Hilfe angeschlossener Funktionen behandelt werden, die jedoch nicht Teil der automatisch verarbeiteten Sprachdefinition sind, sondern als zusätzliche Algorithmen den Übersetzer vervollständigen. Diese Möglichkeiten bieten auch die oben genannten Verfahren zur Generierung von Zerteilern.

Der auf Affix-Grammatiken basierende Compiler-Compiler CDL (Compiler Definition Language [Ko71b]) bezieht die Analyse der Kontextabhängigkeiten in die Zerteilung mit ein. Dieses Vorgehen führt zu einem recht aufwendigen zertei-

lungsverfahren (rekursiver Abstieg mit parametrisierten Prozeduren) und zu Einschränkungen hinsichtlich der Komplexität semantischer Kontextabhängigkeiten.

Für den praktischen Einsatz wurden die tabellengesteuerten Verfahren weiterentwickelt und optimiert (z.B. LL(1), LALR(1), SLR(1)-Verfahren), so dass sie für hinreichend grosse Teilmengen der kontext-freien Grammatiken praktikable Zerteiler erzeugen.

Wir haben in unser System einen LALR(1)-Zerteiler-Generator integriert. Dieses Verfahren wurde von LaLonde in [La71] beschrieben und von Ulukut [Ul74] an der Universität Karlsruhe fortentwickelt und implementiert. Die generierten Zerteiler arbeiten tabellengesteuert, quellbezogen und deterministisch mit einem einzigen Vorausschau-Symbol im Rechtskontext. Sie wurden schon bei verschiedenen Übersetzer-Projekten erfolgreich eingesetzt (z.B. für die in [Go75a] und [Go75b] beschriebenen Sprachen).

In unserem Übersetzer-erzeugenden System kann man den Zerteiler-Generator leicht durch einen anderen ersetzen, der z.B. zielbezogen arbeitende Zerteiler erzeugt. Es wird nur gefordert, dass das Ergebnis der Zerteilung ein Strukturbaum ist, der die abstrakte Programmstruktur repräsentiert.

### 2.3.2 Transformation semantischer Definitionen

Wir fassen die Analyse der statischen Semantik als Vervollständigung der abstrakten Programmstruktur auf, die im Übersetzer durch den Strukturbaum dargestellt wird. Dabei werden den Knoten des Baumes die Attribute zugeordnet, die in der attributierten Grammatik für die entsprechenden Nichtterminal-Symbole definiert sind. Der Strukturbaum wird so in einen attributierten Strukturbaum transformiert. Die Trennung der Verfahren zur syntaktischen und semantischen Analyse erlaubt den Einsatz effizienter Zerteilungsverfahren, ohne dass dadurch die Komplexität der semantischen Kontextabhängigkeiten eingeschränkt würde.

Der Hauptteil unseres Systems umfasst Verfahren, mit denen die semantischen Definitionen überprüft und transformiert werden. Es wird automatisch festgestellt, ob die zu

verarbeitende Sprachdefinition der Beschreibungssprache angehört, alle Attribute gemäss der definierten Typen korrekt angewendet sind und ob die Sprache im Sinne von [Kn68] semantisch wohldefiniert ist. Darüber hinaus wird festgestellt, ob die dynamische Semantik durch syntaktisch korrekte Sätze der Zielsprache beschrieben wird. Der Entwurf korrekter und vollständiger Sprachdefinitionen wird dadurch weitgehend von dem System unterstützt.

Die statischen Definitionen der Semantik werden automatisch in Übersetzer-Algorithmen transformiert. Wir stellen dazu die Kontextabhängigkeiten wie in [Kn68] durch Graphen dar. Nach dem in Abschnitt 4.2 beschriebenen Verfahren werden daraus "Besuchssequenzen" generiert, die zu jedem Knotentyp des Strukturbaumes angeben, in welcher Reihenfolge die Knoten in der Umgebung eines solchen Knotens zu besuchen sind und wie zwischen den Besuchen die einzelnen Attribute bestimmt werden. Das Verfahren kann so variiert werden, dass Besuchssequenzen für eine der folgenden Ablaufstrategien erzeugt werden:

Bestimmung der Attribute

- nach vollständigem Aufbau des Strukturbaumes,
- während des Aufbaus des Strukturbaumes von unten nach oben oder
- während des Aufbaus des Strukturbaumes von oben nach unten.

Zu Übersetzer-Tabellen zusammengefasst steuern diese Besuchssequenzen einen Algorithmus, mit dem jeder durch die Zerteilung aufgestellte Strukturbaum so durchlaufen wird, dass alle Attribute bestimmt werden können. Die Ordnung, in der der Strukturbaum während der semantischen Analyse und Synthese durchlaufen wird, ist deshalb allein durch die semantischen Kontextabhängigkeiten der Sprache bestimmt. Einzelne Teilbäume können auch mehrfach durchlaufen werden, ohne dass ein weiterer Durchgang durch den gesamten Baum erfolgt. Der Analyse-Algorithmus ist ein Keller-Automat zur Attributierung des Strukturbaumes. Er ist allgemeiner als z.B. die in [Sch76] beschriebenen Baum-Automaten, die Bäume in fest vorgegebener Ordnung durchlaufen.

### 2.3.3 Darstellung der Attribute

Die in der Sprachdefinition verwendeten Attribute werden in Datenobjekte und die semantischen Funktionen zu ihrer Bestimmung in Funktionen des Übersetzers transformiert. Im einfachsten Fall wird ein Attribut eines Nichtterminal-Symbols durch Datenobjekte dargestellt, die im Strukturbaum jedem Knoten zugeordnet werden, der ein solches Symbol repräsentiert. In Kapitel 4 beschreiben wir Verfahren, mit denen zusammengesetzte Attribute auf Datenobjekte des Übersetzers abgebildet werden, die nicht im Strukturbaum enthalten sind. Dabei wird ausgenutzt, dass alle semantischen Definitionen in der verarbeiteten Sprachdefinition statisch - und deshalb seiteneffektfrei - sind. Die Anwendung dieser Verfahren führt z.B. automatisch zur Implementierung von Definitions- oder Arttabellen (falls die Eigenschaften der definierten Sprache dies erfordern) nach Techniken, die im Übersetzerbau gebräuchlich sind. Auf diese Weise kann unser System praktikable Übersetzer auch für Sprachen mit komplexen semantischen Eigenschaften generieren.

Die Implementierung von Attributen durch Datenobjekte ausserhalb des Strukturbaumes wurde auch in einer Arbeit von Ganzinger [Ga74] behandelt. Er setzt jedoch voraus, dass die Ablaufstrukturen für die semantische Analyse-Algorithmen explizit in der Sprachdefinition vorgegeben werden. Aus der Notwendigkeit, Attribute global darzustellen, leitet er weitere Einschränkungen für den Durchlauf durch den Strukturbaum und dadurch auch für die Komplexität der semantischen Definitionen ab.

### 2.3.4 Code-Erzeugung

Da die Definitionen der dynamischen Semantik vollständig in die Beschreibungssprache integriert sind, können die generierten Übersetzer den Code für die abstrakte Maschine mit den gleichen tabellengesteuerten Algorithmen erzeugen, mit denen sie auch die semantische Analyse durchführen. Darüber hinaus kann man auch Massnahmen zur quellsprachenbezogenen Optimierung in der Beschreibungssprache durch Attribute formulieren, die das Übersetzer-erzeugende System mit dem oben beschriebenen Methoden behandelt. In [NA74] und [NA75] werden Beispiele für die Beschreibung globaler Optimierungsmassnahmen durch semantische Attribute gegeben. Die Lösung spezieller Probleme maschinennaher Code-Erzeugung und

-Optimierung ist nicht Ziel dieses Verfahrens und unserer Arbeit.

### 2.3.5 Überlegungen zum Aufwand

Unsere Verfahren zur Transformation attributierter Grammatiken sind prinzipiell auf beliebig komplexe Sprachdefinitionen anwendbar. Der Preis für diese Allgemeinheit ist ein nicht unerheblicher Aufwand an Rechenzeit und Speicher für die Analyse der Sprachdefinition und die Generierung der Übersetzer. In [JOR75] wird gezeigt, dass der Zeitaufwand zur Feststellung der semantischen Wohldefiniertheit exponentiell vom Umfang der Sprachdefinition (Anzahl der syntaktischen und semantischen Regeln, der Nichtterminal-Symbole und ihrer Attribute) abhängen kann.

Wir haben deshalb unsere Beschreibungssprache so konzipiert, dass die semantischen Eigenschaften mit einer relativ geringen Anzahl von Attributen definiert werden können. Dies kann durch die Verwendung zusammengesetzter Attribute erreicht werden. Ausserdem kann die Komplexität der Sprachdefinition durch bedeutungserhaltende Transformationen, wie Eliminierung syntaktischer und semantischer Kettenproduktionen, weiter verringert werden. Unter diesen Voraussetzungen wird unser Verfahren für Programmiersprachen wie PASCAL [HW73] oder LEX [Go75a] mit akzeptablem Aufwand einsetzbar sein.

Für die im Anhang B definierten Beispiel-Sprachen überprüfte unser System die semantische Wohldefiniertheit und generierte die Analyse-Tabellen des Übersetzers. Es brauchte ca. 20 Sekunden Prozessorzeit auf der Burroughs B6700 für Beispiel 1 und ca. 8 Sekunden für Beispiel 2.

Die Übersetzer-Algorithmen zur semantischen Analyse und Synthese sind gut an die Erfordernisse der jeweiligen Sprache angepasst, da die Tabellen, die den Baum-Durchlauf steuern, aus den Kontextabhängigkeiten der Sprache erzeugt werden. Dieses Verfahren setzt allerdings eine Implementierung des Strukturbaumes als leicht zugängliche Datenstruktur (Geflecht im Kernspeicher) voraus. In Kapitel 4 zeigen wir, wie die Information über die angewendete Durchlaufordnung zur Verringerung des Speicheraufwands für den Strukturbaum herangezogen werden kann.

Bochmann gibt in [Bo76] einen Algorithmus an, mit dem aus der Analyse der semantischen Abhängigkeiten festgestellt wird, wie viele Durchgänge durch jeden Strukturbaum in fest vorgegebener Ordnung (von links nach rechts) nötig sind, um alle Attribute zu bestimmen. Dieses Verfahren erfordert zwar einen geringeren Aufwand, der nicht exponentiell vom Sprachumfang abhängt; es ist jedoch nur auf eine Untermenge der semantisch wohldefinierten Sprachen anwendbar. Definitionen von rekursiven Sprachelementen, in denen semantische Information von rechts nach links "fliesst" (z.B. ALGOL 68 Verbundselektionen: a of b of c oder Vereinbarungen von Artidentifikatoren), können so nicht oder erst nach einer Transformation behandelt werden.

Die nach dieser Methode konstruierten Übersetzer-Algorithmen sind wegen ihrer starren Durchlaufordnung schlechter an die jeweiligen semantischen Kontextabhängigkeiten angepasst und deshalb weniger effizient. So erfordert ein lokaler Informationsfluss von rechts nach links einen weiteren Lauf durch den gesamten Strukturbaum, während nach unserer Methode die Teilbäume in geeigneter Reihenfolge durchlaufen werden. Eine Linearisierung des Strukturbaumes ist auch bei Bochmanns Verfahren nicht möglich.

Das System SLS/1 [Le75] generiert Übersetzer für Sprachen, deren semantische Komplexität soweit eingeschränkt ist, dass alle Attribute in einem einzigen Durchgang durch den Strukturbaum in vorgegebener Ordnung von links nach rechts bestimmt werden können.

# 3 Beschreibungssprache für semantische Definitionen

In diesem Kapitel definieren wir eine Sprache zur Beschreibung der syntaktischen Struktur, der statischen und der dynamischen Semantik höherer Programmiersprachen. Mit dieser Beschreibungssprache soll das Prinzip der attributierten Grammatiken in eine Sprache integriert werden, die den in 2.1.2 diskutierten Anforderungen genügt (Überprüfbarkeit, Transformierbarkeit, Verständlichkeit). Die Beschreibungssprache ist zugleich Eingabesprache für unser Übersetzer-erzeugendes System. Im Anhang B wird ihre Anwendung durch die Beschreibung einer einfachen Programmiersprache demonstriert. Die Vollständigkeit der gewählten Beschreibungsmittel und deren Eignung zur Formulierung gut lesbarer Definitionen komplexer Sprachen kann erst durch eine intensive praktische Erprobung gezeigt und gegebenenfalls verbessert werden.

In [Kn68], [Kn69] und [Wi71] werden Beispiele für die Formulierung semantischer Definitionen mit Hilfe attributierter Grammatiken gegeben. Die dabei angewendeten Beschreibungsmittel sind jedoch an die jeweils definierte Sprache angepasst und nicht allgemein anwendbar. Die in [Ga74] vorgestellte Beschreibungssprache stimmt zwar in einigen Konzepten mit Elementen unserer Beschreibungssprache überein; sie enthält jedoch als wesentliche Sprachelemente Konzepte zur Formulierung von Ablaufstrukturen und genügt deshalb nicht den Anforderungen an eine statische Beschreibungssprache.

## 3.1 Konzeption

Die Eigenschaften eines Sprachelements werden durch den Kontext, in dem es auftritt, und durch Eigenschaften von Teilelementen, aus denen es zusammengesetzt ist, bestimmt. Diese Zusammenhänge sind festgelegt durch die syntaktischen Regeln, die bei der Ableitung des Sprachelements angewendet werden. Die Datenobjekte der Beschreibungssprache sind Nichtterminal-Symbole und Attribute. Jedem Nichtterminal-Symbol der Grammatik werden Attribute zugeordnet, die durch ihren Typ jeweils eine Menge von möglichen Eigenschaften beschreiben, die das Sprachelement in verschiedenem Kontext

annehmen kann. Den syntaktischen Regeln werden semantische Regeln zugeordnet. Sie definieren Funktionen zur Berechnung von Werten zu den Attributen im Kontext der syntaktische Regel. Die Werte müssen im Wertebereich des Attribut-Typs liegen (d.h. eine der möglichen Eigenschaften spezifizieren). Die semantischen Regeln werden im allgemeinen abhängig von Attributen der in der syntaktischen Regel auftretenden Nichtterminal-Symbole angegeben. Darüber hinaus können semantische Regeln zur Beschreibung von Spracheigenschaften formuliert werden, die durch einen grösseren als den durch eine syntaktische Regel gegebenen Kontext oder völlig unabhängig vom Kontext bestimmt werden. So können z.B. auch Massnahmen zur globalen Code-Optimierung in einfacher Weise beschrieben werden.

Die Zuordnung eines Wertes zu einem Attribut eines Nichtterminal-Symbols in einem bestimmten Kontext ist nicht veränderbar. Da die Beschreibungssprache keine dynamischen Konzepte enthält, ist die Reihenfolge, in der die semantischen Regeln angegeben werden, bedeutungslos. (In Kapitel 4.2 wird gezeigt, wie das Übersetzer-erzeugende System ermittelt, in welcher Reihenfolge die semantischen Regeln abzuarbeiten sind.)

Die semantischen Regeln sind aus einigen Grundoperationen und Fallunterscheidungen zusammengesetzte, streng funktionale Ausdrücke. Mit Hilfe eines Funktionskonzeptes können auch rekursiv zu berechnende Ausdrücke formuliert werden. Die Verständlichkeit von Sprachdefinitionen wird dadurch unterstützt, dass die Beschreibungssprache kein Variablenkonzept enthält und die Definitionen deshalb frei von Seiteneffekten sind.

Die grundlegenden Attribut-Typen (ganze Zahlen, logische Werte, Zeichenreihen), die Konstruktionsprinzipien zur Bildung neuer Typen (Ausschnitte, symbolische Typen, Mengen, Verbunde und Typ-Vereinigungen) und die zu den Typen definierten Operationen entsprechen im wesentlichen den in höheren Programmiersprachen gebräuchlichen Konzepten. Diese Begriffe eignen sich auch zur Beschreibung von Programmiersprachen. Die Beschreibungssprache enthält weitere Konzepte, die sich für die Definition komplexer Spracheigenschaften (z.B. Gültigkeitsbereiche) als zweckmässig erwiesen haben: einen speziellen Attribut-Typ für Bezeichner; Konstruktoren zur Bildung von Typen für Mengen, deren Elemente linear geordnet sind, und auf die mit Schlüsseln zugegriffen werden

kann; und Operationen auf solchen Objekten.

Unser Beschreibungsmodell schliesst die Beschreibung der dynamischen Semantik ein. Sie wird denotational durch Attribute definiert, deren Wertebereiche die Sprachelemente der abstrakten Maschine AMICO sind. Aus diesen Attributen wird direkt die Code-Erzeugung des Übersetzers generiert.

Diese Beschreibungssprache erlaubt eine weitgehende Überprüfung von Sprachdefinitionen auf ihre Vollständigkeit und Konsistenz durch das Übersetzer-erzeugende System. Es wird automatisch festgestellt, ob die Sprachdefinition die folgenden Bedingungen erfüllt:

a) Alle Eigenschaften jedes Sprachelements sind in jedem Kontext eindeutig definiert.

b) Es gibt keinen Satz der Sprache, in dem eine zyklische Abhängigkeit zwischen semantischen Eigenschaften besteht.

c) Die in der Sprachdefinition festgelegten Einschränkungen möglicher semantischer Eigenschaften werden in jedem Fall eingehalten.

Die Bedingung a) wird folgendermassen überprüft: Die Attribute jedes Nichtterminal-Symbols werden in zwei Klassen eingeteilt. Es wird unterschieden, ob der zugehörige Attribut-Wert durch den "äusseren Kontext" (erworbene Attribute, engl.: inherited attributes) oder durch den "inneren Kontext" (abgeleitete Attribute, engl.: derived attributes) bestimmt wird. Kein Attribut darf zugleich erworben und abgeleitet sein, und in jedem Kontext (d.h. zu jeder syntaktischen Regel) darf es höchstens eine semantische Regel für jedes Attribut geben. Diese Forderung ist sinnvoll, denn sie ist Voraussetzung für die semantische Eindeutigkeit.

Zur Vollständigkeit der semantischen Definitionen wird gefordert, dass es zu allen erworbenen Attributen eines Nichtterminal-Symbols in jedem Kontext, in dem es auftritt, eine semantische Regel gibt und dass es zu allen abgeleiteten Attributen eines Nichtterminal-Symbols zu jeder seiner Ableitungen (innerer Kontext) eine semantische Regel gibt.

Die Bedingung b) nennt Knuth in [KN68] "semantische Wohldefiniertheit". Das Verfahren, das er dort zu ihrer Überprüfung angibt, haben wir in modifizierter Form in unser

Verfahren zur systematischen Analyse semantischer Abhängigkeiten übernommen (siehe Abschnitt 4.2).

Die Bedingung c) ist erfüllt, wenn der Wert jedes Attributes dem spezifizierten Attribut-Typ angehört. Für die Attribute, mit denen die dynamische Semantik beschrieben wird, bedeutet diese Forderung, dass die Spezifikationen im Sinne der Zielsprache syntaktisch korrekt sind. Eine weitergehende inhaltliche Prüfung ist hier nicht möglich.

## 3.2 Definition der Beschreibungssprache

### 3.2.1 Terminologie

Zur Definition der Beschreibungssprache verwenden wir die folgenden Begriffe:

Eine Sprachdefinition ist ein Tupel (G,EA,AT,A,FT,SF,SR). Es bedeuten:

G=(T,V,P,Z) kontext-freie Grammatik

| | |
|---|---|
| V | Vokabular |
| T c V | Terminal-Symbole |
| N = V \ T | Nichtterminal-Symbole |
| P | syntaktische Regeln |
| Z∈N | Ziel-Symbol |
| EA | Menge elementarer Attribut-Werte |
| AT | Menge von Attribut-Typen. Für jedes at∈AT gilt at c EA oder at=(M,F), wobei M Teilmenge der Potenzmenge von AT und F eine Menge von Funktionen über M ist, die durch Typkonstruktoren definiert wird. Nur solche Typen aus AT sind sinnvoll, zu denen endliche Attribut-Werte angegeben werden können. (Dies ist wegen der sprachlichen Formulierung von Typ-Definitionen überprüfbar, siehe Abschnitt 4.5.) |

A — Abbildung $A : N \rightarrow 2^I$
A bildet die Nichtterminal-Symbole in die Potenzmenge einer Indexmenge ab. $A_X$ heisst die Menge der Attribute zu $X \in N$.

FT — Abbildung $FT : N \rightarrow \{ A_X \rightarrow AT \}$
FT ordnet den Attributen der Nichtterminal-Symbole Attribut-Typen zu.

SF — Menge von semantischen Funktionen
$SF_{p,a} : at_1 X \ldots X at_m \rightarrow at$, $m \geq 0$.
$SF_{p,a}$ ist einer Regel p und einem Attribut a vom Typ at zugeordnet. Es gilt:

$p = X_0 : t_0 X_1 t_1 X_2 \ldots X_n t_n$
$a \in A_{X_i}$, $t_i \in T^*$, $X_i \in N$, $0 \leq i \leq n$.

$SF_{p,a}$ definiert eine Vorschrift zur Bestimmung eines Wertes vom Typ at zu a. $SF_{p,a}$ kann in Abhängigkeit von anderen Attributen $a_k \in A_{X_j}$ vom Typ $at_k$, $1 \leq k \leq m$, $0 \leq j \leq n$, angegeben werden. (Ist a ein Attribut eines zusammengesetzten Attribut-Typs, und u eine Komponente von a vom Typ at, so bestimmt $SF_{p,a.u}$ einen Wert zu dieser Komponente.)

SR — Menge von Relationen $SR_p : at_1 X \ldots X at_m$ über Attributen $a_k \in A_{X_j}$ vom Typ $at_k$, $1 \leq k \leq m$, $0 \leq j \leq n$ zu einer Regel
$p = X_0 : t_0 X_1 t_2 X_2 \ldots X_n t_n$

Wir nennen die semantischen Funktionen und Relationen auch zusammenfassend semantische Regeln. Zu jedem Nichtterminal-Symbol werden die Mengen der abgeleiteten Attribute ($AD_X$) und der erworbenen Attribute ($AI_X$) unterschieden. Es gilt:

$AD_X = \{ a \mid a \in A_X$ und es gibt eine semantische Funktion $SF_{p,a}$ mit $p = X : v \in P, X \in N, v \in V^* \}$

$AI_X = \{ a \mid a \in A_X$ und es gibt eine semantische Funktion $SF_{p,a}$ mit $p = Y : uXv \in P, X,Y \in N, u,v \in V^* \}$

Wir fordern, dass die semantischen Funktionen $SF_{p,a}$ und die Attributmengen $AD_X$ und $AI_X$ folgenden Konsistenz-Bedingungen genügen:

a) Für alle $X \in N$ muss gelten
$AI_X \setminus/ AD_X = A_X$ und $AI_X /\setminus AD_X = \emptyset$

b) Zu jeder Regel $p = X:v \in P$ und jedem $a \in AD_X$ gibt es genau ein $SF_{p,a}$.

c) Zu jeder Regel $p = Y:uXv \in P$ und jedem $a \in AI_X$ gibt es genau ein $SF_{p,a}$.

d) Für das Ziel-Symbol Z der Grammatik gilt $AI_Z = \emptyset$.

Die syntaktische und semantische Analyse, die ein Übersetzer durchführt, kann mit diesen Begriffen folgendermassen beschrieben werden:

Ein Eingabesatz wird gemäss der Grammatik G zerteilt und seine abstrakte Struktur durch einen attributierten Strukturbaum (ASB) dargestellt. Jeder Knoten des ASB gehört einer Klasse an, die durch das Symbol $Y \in N$ bestimmt ist, das der Knoten repräsentiert. Zu jedem Knoten einer Klasse Y wird eine Menge von Attribut-Werten bestimmt, deren Anzahl und Typ-Zugehörigkeit durch die Y zugeordneten Attribute festgelegt ist.

Jeder Knoten k einer Klasse Y repräsentiert zusammen mit seinen direkten Nachfolge-Knoten $k_i$, $1 \leq i \leq n$, die Anwendung einer syntaktischen Regel

$$p = Y : t_0 X_1 t_1 X_2 \ldots X_n t_n \in P.$$

Wir nennen k dann vom Knoten-Typ p. Die Menge der Puntionen $SP_{p,a}$, mit $a \in A_{X_i} \setminus/ A_Y$, gibt die semantischen Regeln an, nach denen die abgeleiteten Attribute des Knotens k und die erworbenen Attribute der Knoten $k_i$ ermittelt werden. Die Werte der Attribute der Knoten k und $k_i$ müssen den Relationen $SR_p$ genügen.

Ein syntaktisch korrekter Satz ist semantisch korrekt, wenn alle Attribut-Werte aller Knoten des ASB den durch die Attribute vorgegebenen Attribut-Typen angehören und den jeweiligen Relationen aus $SR_p$ genügen.

Die Übersetzung eines Satzes der Sprache können wir nun wie folgt definieren: Wir zeichnen einen Attribut-Typ zat∈AT als "Ziel-Attribut-Typ" aus und verlangen, dass dem Zielsymbol Z genau ein Attribut fz vom Typ zat zugeordnet ist. Das Ergebnis der Übersetzung des Satzes ist dann das der Wert von fz.

Im folgenden definieren wir die konkrete Notation der Beschreibungssprache. Die Bedeutung der einzelnen Konstruktionen wird mit den hier eingeführten Begriffen erklärt. Die Grammatik wird in einer erweiterten BNF-Notation angegeben. Alle in < > eingeschlossenen Symbole und alle Bezeichner, die nur aus grossen Buchstaben bestehen, sind Terminal-Symbole der Beschreibungssprache. Regeln der Form

X_Bezeichner : Bezeichner

sind in der Grammatik nicht angegeben.

Eine Sprachdefinition hat die Form

```
Sprachdefinition: ( Definition || <;> ) .
Definition:
     Typ_Definition |
     Nichtterminal_Definition |
     Regel .
```

### 3.2.2 Attribut-Typen

```
elementarer_Typ:
     INT | BOOL | IDENT | STRING .
```

Die Menge EA der elementaren Attribut-Werte und die Menge AT der Attribut-Typen einer Sprachdefinition werden durch Typ_Definitionen bestimmt. Darüber hinaus werden durch das Übersetzer-erzeugende System einige Grundtypen und die zugehörigen elementaren Attribut-Werte standardmässig vorgegeben. Dies sind die Typen INT, BOOL, IDENT, STRING ∈ AT mit

INT Menge der ganzen Zahlen

BOOL {TRUE, FALSE}

IDENT Menge aller in Sätzen der definierten Sprache auftretende Bezeichner

STRING Menge aller in Sätzen der definierten Sprache auftretenden Texte zu den vordefinierten Grundsymbolen (siehe Abschnitt 3.2.3)

Attribut-Typ-Definitionen haben folgende syntaktische Struktur:

```
Typ_Definition:
     TYPE Bezeichner <:> Typ_Beschreibung .

Typ_Beschreibung:
     skalare_Liste | Ausschnitt | Vereinigungs_Typ |
     Verbund_Typ | Mengen_Typ .
```

### 3.2.2.1 Skalare Listen

Mit einer skalaren Liste der Form

```
skalare_Liste:
     <(> ( Bezeichner || <,> ) <)>.
```

werden zugleich ein Attribut-Typ aus AT und die angegebenen Bezeichner als elementare Attribute aus EA definiert. Die Reihenfolge, in der die Bezeichner angegeben werden, definiert eine Ordnung über diesen Attributen.

### 3.2.2.2 Ausschnitte

Durch Typ_Beschreibungen der Form

```
Ausschnitt:
     <[> ganze_Zahl <:> ganze_Zahl <]> |
     <[> Bezeichner <:> Bezeichner <]> .
```

werden geordnete Teilmengen des Typs INT oder einer skalaren Liste als Attribut-Typ definiert.

### 3.2.2.3 Vereinigungs-Typen

```
Vereinigungs_Typ:
     UNION <(> ( Typ_Angabe || <,> ) <)> .

Typ_Angabe:
     elementarer_Typ | Typ_Bezeichner .
```

Ein Vereinigungs_Typ beschreibt eine Menge von Werte-Paaren:

$$V = \{ (is\_at_i, v) \mid v \in at_i,\ is\_at_i \in s,\ \text{die } at_i \text{ sind durch die angegebenen Typen bestimmt, } s \text{ ist ein skalarer Typ} \}$$

Die elementaren Attribut-Werte der Form is_at werden implizit für alle in Vereinigungs_Typ-Beschreibungen auftretenden Attribut-Typen definiert. Attribute von Vereinigungs_Typen sind als Verbund-Attribute (siehe 3.2.2.4) mit zwei Komponenten aufzufassen, für die implizit die Komponenten-Bezeichner "unitedtype" und "value" eingeführt werden. Die Werte zu dieser Komponenten können (wie bei Verbund-Attributen) durch verschiedene semantische Regeln bestimmt werden. Die oben angegebene Konsistenz-Bedingung wird automatisch überprüft.

### 3.2.2.4 Verbund-Typen

```
Verbund_Typ:
     STRUCT <(> ( Komponenten_Spezifikation || <,> ) <)> .

Komponenten_Spezifikation:
     ( Bezeichner || <,> ) <:> Typ_Angabe .
```

Ein Verbund_Typ ist definiert durch eine Menge M von Tupeln von Attribut-Typen und Selektions-Funktionen $s_i$,

$1 \leq i \leq n$, die durch die angegebenen Bezeichner benannt werden. $s_i$ angewendet auf ein solches Tupel liefert dessen i-tes Glied. Die Glieder der Tupel nennen wir Komponenten. Ihre Typen sind durch die Verbund_Typ-Beschreibung definiert. Zu einem Verbund_Typ muss es mindestens einen endlichen Wert geben.

Beispiele:

(1) TYPE a : STRUCT( x:v, y:INT);
    TYPE v : UNION ( a, INT )

Zu a gibt es endliche Werte.

(2) TYPE b : STRUCT( x:b, y:INT)

Zu b gibt es keine endlichen Werte.

Der Wert eines Attributes vom Verbund_Typ wird entweder durch eine einzige semantische Regel oder komponentenweise durch mehrere semantische Regeln bestimmt. Insbesondere brauchen nicht alle Komponenten eines solchen Attributes abgeleitet oder alle erworben zu sein.

### 3.2.2.5 Mengen-Typen

Mengen_Typ:
    SETOF Typ_Angabe [ KEY Bezeichner ] [ LINEAR ] .

Ein Mengen_Typ beschreibt die Potenzmenge des zugrunde liegenden Typs. Es werden vier Formen von Mengen_Typen unterschieden:

a) SETOF Typ_Angabe

Der zugrunde liegende Typ muss ein Ausschnitt oder ein skalarer Typ sein. Über solchen Mengen sind die üblichen Mengen-Operationen (Vereinigung, Durchschnitt, Differenz) und Relationen (Mengen- und Element-Inklusion) definiert.

b) SETOF Typ_Angabe KEY Bezeichner

Der zugrunde liegende Typ muss ein Verbund_Typ sein, in dem eine Komponente durch den angegebenen Bezeichner als Schlüssel ausgezeichnet wird. Diese Komponente muss vom Typ IDENT, INT oder eines Ausschnitts von INT sein. Über solchen Mengen und dem Schlüsseltyp ist die Relation IN definiert:

Es gilt s IN m, falls m ein Element mit dem Schlüssel s enthält.

Eine Funktion zur Schlüssel-Identifikation s ID m liefert das Element von m mit dem Schlüssel s, falls s IN m gilt.

Bei der Bildung von Attribut-Werten eines solchen Mengen_Typs durch Vereinigung (siehe 3.2.5.1) wird die Eindeutigkeit des Schlüssels sichergestellt.

c) SETOP Typ_Angabe LINEAR

Attribute dieses Typs können als linear geordnete Listen von Elementen des zugrunde liegenden Typs aufgefasst werden. Ein Attribut vom Typ l = SETOP t ist ein geordnetes Paar (et,el) mit et∈t und el∈l. Die Anordnung der Elemente wird bei der Bildung eines solchen Attributes (siehe 3.2.5.1) bestimmt durch die Reihenfolge, in der die Elemente oder Teilmengen angegeben werden (nicht durch den Wert der Elemente; insbesondere können gleichwertige Elemente mehrfach auftreten). Die Funktionen HEAD und TAIL sind mit der üblichen Bedeutung definiert.

d) SETOP Typ_Angabe KEY Bezeichner LINEAR

Auf Attribute dieses Typs sind die Operationen von b) und c) anwendbar. Wie in b) müssen alle Elemente hinsichtlich des Schlüssels eindeutig sein.

In den Fällen b), c) und d) sind ausser der Vereinigung (in der Schreibweise von 3.2.5.1) keine weiteren Mengen-Operationen und -Relationen definiert.

Der Wert eines Attributes vom Mengen_Typ wird im allgemeinen durch eine einzige semantische Regel bestimmt. Ist der zugrunde liegende Typ ein Verbund_Typ, so wird durch diese Regel zumindest die Anzahl der Elemente und ggf. ihre

Ordnung und der Wert der Schlüssel-Komponente aller Elemente festgelegt. Die Werte der übrigen Komponenten können in weiteren Regeln bestimmt werden.

### 3.2.3 Definition der Attribute

Die Menge $A_X$ der Attribute eines Symbols X∈N wird durch folgende Definitionen angegeben:

```
Nichtterminal_Definition:
   NONTERM ( Nichtterminal_Bezeichner || <,> ) <:>
   [ Attribut_Definition ||<,> ] .

Attribut_Definition:
   ( Attribut_Bezeichner [ <:=> Ausdruck ] || <,> ) <:>
   Typ_Angabe .
```

Allen angegebenen Nichtterminal-Symbolen werden die gleichen Attribute der jeweiligen Attribut-Typen zugeordnet. Jedem Attribut wird ein Bezeichner zugeordnet, durch den das Attribut in Attribut_Benennungen (siehe 3.2.5.2) identifiziert werden kann. Ist die Vorschrift zur Bestimmung eines Attribut-Wertes unabhängig vom Kontext einer Regel, so kann sie als semantischer Ausdruck in der Attribut_Definition angegeben werden. Das Attribut wird in diesem Fall den abgeleiteten Attributen des Nichtterminal-Symbols zugerechnet.

Zur Beschreibung einiger Grundsymbol-Klassen werden folgende Nichtterminal-Symbole mit ihren Attributen durch das Übersetzer-erzeugende System standardmässig definiert (siehe auch 3.2.4.1)

```
NONTERM
   identsymbol: iden :={eindeutige Abbildung
                        des Grundsymbols in die
                        Attributmenge IDENT}  : IDENT ,
                text :={Text des Grundsymbols}: STRING ;
NONTERM
   intsymbol, realsymbol, stringsymbol :
                text :={Text des Grundsymbols} : STRING;
```

Mit Hilfe der Attribute text können die Grundsymbole (in der standardmässig festgelegten Schreibweise, siehe 3.2.4.1) in den erzeugten Zwischencode übernommen werden.

### 3.2.4 Regeln

```
Regel:
     RULE syntaktische_Regel
     [ SEMANTIC ( semantische_Regel || <;> ) ] END .
```

Die Regeln bilden den Kern jeder Sprachdefinition. Sie beschreiben die syntaktische Struktur und die semantischen Eigenschaften jeweils eines Sprachelements durch eine syntaktische Regel $p \in P$ und der ihr zugeordneten semantischen Regeln $SF_{p,a}$ und $SR_p$.

#### 3.2.4.1 Syntaktische Regeln

```
Syntaktische_Regel:
   Nichtterminal_Bezeichner <:> (syntaktische_Einheit)*.

Syntaktische_Einheit:
   syntaktisches_Element | Wiederholungs_Einheit |
   optionale_Einheit .

Syntaktisches_Element:
   Nichtterminal_Bezeichner | Terminal .

Terminal:
   <"> ( Zeichen )* <"> .

Zeichen:
   Zeichen_ausser_Apostroph | <""> .

Wiederholungs_Einheit:
   <(> ( syntaktisches_Element )* Trennung <)> |
   <(> ( syntaktisches_Element )* <)*> |
   <[> ( syntaktisches_Element )* Trennung <]> |
   <[> ( syntaktisches_Element )* <]*> .
```

```
Trennung:
   <//> Terminal .

optionale_Einheit:
   <[> ( syntaktisches_Element )* <]> .
```

Die syntaktischen Regeln werden in einer der erweiterten BNF ähnlichen Schreibweise angegeben. Durch die obligatorische Kennzeichnung aller Terminal-Symbole wird eine eindeutige Unterscheidung von den Wortsymbolen der Beschreibungssprache und Nichtterminal_Bezeichnern gewährleistet. Alle Terminal-Symbole werden aus Folgen von Zeichen aus dem Zeichensatz gebildet, der mit der Implementierung des Übersetzer-erzeugenden Systems festgelegt wird ("" ist das Bild der Terminal-Symbol-Klammer). Um eine eindeutige Zuordnung der semantischen Regeln zu gewährleisten, dürfen Wiederholungs_Einheiten und optionale_Einheiten höchstens ein Nichtterminal-Symbol enthalten und nicht geschachtelt auftreten.

Für bestimmte Grundsymbol-Klassen sind die Nichtterminal_Bezeichner intsymbol, realsymbol, stringsymbol und identsymbol standardmässig vordefiniert. Ihre Schreibweise ist durch das Übersetzer-erzeugende System vorgegeben:

```
intsymbol:
   ( Ziffer )* .

realsymbol:
   [ Ziffer ]* <.> ( Ziffer )* [ Exponent ] |
   [ Ziffer ]* Exponent .

Exponent:
   <E> [ <+> ] ( Ziffer )* | <E> <-> ( Ziffer )* .

stringsymbol:
   <"> Zeichen <"> .

identsymbol:
   Buchstabe ( (Ziffer | Buchstabe)* || <_> ) .
```

Diesen Nichtterminal-Symbolen sind standardmässig bestimmte Attribute zugeordnet, deren Werte abhängig von dem jeweiligen Grundsymbol bestimmt werden (siehe 3.2.3).

### 3.2.4.2 Semantische Regeln

```
semantische_Regel:
   Attribut_Bestimmung | Ausdruck .

Attribut_Bestimmung:
   Attribut_Benennung <:=> Ausdruck .
```

Eine Attribut_Bestimmung ist die Formulierung einer semantischen Funktion $SF_{p,a}$. Das Attribut a (bzw. die Attribut-Komponente) wird durch die Attribut_Benennung angegeben. Der Ausdruck beschreibt die Vorschrift zur Bestimmung des Attribut-Wertes. Das Attribut kann jedem Nichtterminal-Symbol der zugehörigen syntaktischen Regel zugeordnet sein - auch solchen, die als optional gekennzeichnet sind oder in einer Wiederholungsklammer stehen. In diesem Fall wird der durch die semantische_Regel bestimmte Wert den Attributen a aller bei der jeweiligen Anwendung der Regel tatsächlich abgeleiteten Nichtterminal-Symbole zugeordnet.

Relationen $SR_p$ über Attributen werden durch semantische Regeln formuliert, die aus einem Ausdruck vom Attribut_Typ BOOL bestehen. Sie beschreiben Aussagen über Attributen. (Sie werden in semantische Prüfungen des erzeugten Übersetzers transformiert.)

## 3.2.5 Semantische Ausdrücke

```
Ausdruck:
   einfacher_Ausdruck |
   Ausdruck dyadischer_Operator einfacher_Ausdruck .

einfacher_Ausdruck:
   Standard_Attribut_Notation |
   <(> Ausdruck <)> |
   Fallunterscheidung |
   Attribut_Benennung |
   monadischer_Operator einfacher_Ausdruck .
```

Ein semantischer Ausdruck ist eine Funktion zur Bestimmung eines Wertes vom Typ ATTRIBUT, der mit dem Typ übereinstimmen muss, der durch den Kontext vorgegeben wird, in dem

der Ausdruck steht. Es gibt keine implizite Typanpassung.

Die Ausdrücke werden mit Hilfe von vordefinierten Grundfunktionen (Operatoren, siehe 3.2.5.4), Attribut_Benennungen und Fallunterscheidungen formuliert.

### 3.2.5.1 Standardschreibweisen für Attribut-Werte

```
Standard_Attribut_Notation:
   ganze_Zahl | TRUE | FALSE | Bezeichner | Zeichenreihe |
   Typ_Bezeichner [ KEYS ] <(> ( Ausdruck || <,> ) <)>.
```

Die Standard-Notationen für Werte vom Typ INT (BOOL, IDENT, STRING) sind ganze Zahlen (TRUE und FALSE, Bezeichner, Zeichenreihen), für Werte von skalaren_Listen die in der Typ_Definition eingeführten Bezeichner und für die implizit eingeführten Werte, welche die Typ-Zugehörigkeit von vereinigten Typen angeben, sind es Typ_Bezeichner mit vorangestellten "is", z.B. is_atyp.

Die Standard-Notationen für Werte von Ausschnitt_Typen entsprechen den Notationen für Attribute des Typs, aus dem der Ausschnitt gebildet wurde.

Für alle übrigen Typen (Mengen, Verbunde und Vereinigungen) besteht die Standard-Notation der Werte aus dem Typ_-Bezeichner und einer Liste von Ausdrücken. Bei Verbund-Attributen müssen die Ausdrücke vom Typ der jeweiligen Komponente sein, bei Vereinigungs-Attributen von einem der vereinigten Typen, bei Mengen-Attributen vom Mengentyp oder vom zugrunde liegenden Typ. Die Vereinigung von Mengen mit Schlüsselzugriff und/oder linearer Ordnung wird ausschliesslich durch diese Form der Standard_Attribut_Notation angegeben. Die Reihenfolge der Ausdrücke bestimmt gegebenenfalls die lineare Anordnung der Mengen-Elemente.

Bezeichnet der Typ_Bezeichner einen Mengentyp mit Schlüsselzugriff und ist das Wortsymbol KEYS angegeben und beschreiben verschiedene Ausdrücke Attribute mit gleichem Schlüssel, so wird nur jeweils das Element des Ausdrucks, der am weitesten rechts steht, in die Ergebnismenge aufgenommen. Ist das Wortsymbol KEYS nicht angegeben, müssen alle Schlüssel verschieden sein.

### 3.2.5.2 Attribut-Benennungen

```
Attribut_Benennung:
   Nichtterminal_Benennung <.> Attribut_Bezeichner |
   Nichtterminal_Benennung <.> Komponenten_Benennung |
   THIS |
   THIS <.> Komponenten_Benennung .

Nichtterminal_Benennung:
   Nichtterminal_Bezeichner |
   Nichtterminal_Bezeichner <[> ganze_Zahl <]> .

Komponenten_Benennung:
   Attribut_Bezeichner <.> Komponenten_Bezeichner |
   Komponenten_Benennung <.> Komponenten_Bezeichner |
   HEAD | TAIL .
```

Eine Attribut_Benennung identifiziert ein Attribut eines Nichtterminal-Symbols oder eine Komponente eines Attributes vom Verbund_Typ. Die Nichtterminal_Benennung identifiziert ein Symbol der zugeordneten syntaktischen Regel. Falls diese mehrere gleich bezeichnete Nichtterminal-Symbole enthält, werden sie durch die angegebene ganze Zahl unterschieden.

Beispiel:

```
RULE bedingter_Ausdruck:
   IF Ausdruck THEN Ausdruck ELSE Ausdruck FI
   SEMANTIC Ausdruck[1].Art:= bool;
            Ausdruck[2].Art:= bedingter_Ausdruck.Art;
            Ausdruck[3].Art:= bedingter_Ausdruck.Art
END
```

Attribut_Benennungen, die sich auf Attribute eines Nichtterminal-Symbols beziehen, das in einer Wiederholungsklammer der syntaktischen Regel steht, beschreiben bei der Verwendung in Ausdrücken eine Menge von Werten des Typs des Attributes. Sie können deshalb nur in Standard-Notationen für Mengen stehen. (Zu Benennungen von Attributen optionaler Nichtterminal-Symbole siehe 3.2.6.2.) Die Bedeutung von THIS wird im Zusammenhang mit der Fallunterscheidung in 3.2.5.3 erklärt.

### 3.2.5.3 Fallunterscheidung

```
Fallunterscheidung:
   CASE Ausdruck OF ( Fall || <;> ) [ OUT Ausdruck ] ESAC .

Fall:
   ( Fallmarke <:> )* Ausdruck .

Fallmarke:
   Ausdruck .
```

Durch eine Fallunterscheidung können Attribut-Werte mit Hilfe verschiedener Funktionen bestimmt werden, abhängig von dem Wert des Auswahl-Ausdrucks. Fallunterscheidungen sind so definiert, dass sie möglichst vielseitig einsetzbar sind, da sie bei der Beschreibung semantischer Eigenschaften erfahrungsgemäss sehr häufig verwendet werden. Die Unterscheidungskriterien werden durch den Typ des Auswahl-Ausdrucks bestimmt.

Ist der Auswahl-Ausdruck vom Typ BOOL, INT, skalare Liste oder Ausschnitt, so müssen die Fallmarken Standardnotationen der entsprechenden Typen sein.

Ist der Auswahl-Ausdruck vom Vereinigungs_Typ, so sind die implizit für die vereinigten Typen eingeführten Bezeichner der Form is_typbez die Fallmarken.

Ist der Auswahl-Ausdruck vom Verbund_Typ, so sind die Fallmarken logische Ausdrücke der folgenden Form:

```
(Komponenten_Bezeichner <=>
     Standard_Attribut_Notation || AND )
```

Diese logischen Ausdrucke über den Komponenten des Auswahl-Ausdrucks bestimmen, welcher Fall der Fallunterscheidung anzuwenden ist. Sind die Bedingungen nicht disjunkt, so bestimmt die im Sinne der Aufschreibung erste erfüllte Bedingung das Ergebnis der Fallunterscheidung.

Ist der Auswahl-Ausdruck vom Typ einer linear geordneten Menge, so ist als Fallmarke nur der vordefinierte Bezeichner **EMPTY** zulässig, der für die leere Menge steht.

In den Ausdrücken nach den Fallmarken kann anstelle einer Attribut_Benennung das Symbol THIS stehen. Dadurch wird das Ergebnis der Auswertung des Fallausdruckes benannt.

Sind nicht alle Werte des Typs des Auswahl-Ausdrucks als Fallmarken angegeben, so muss die Fallunterscheidung einen OUT-Teil enthalten.

Beispiele:

1.
```
CASE e OF
integer: i;
real   : r
OUT z ESAC
```

2.
```
CASE u OF
is_einfache_Art: THIS;
is_proz_Art    : THIS.ergebnisart
OUT unart ESAC
```

3.
```
CASE v OF
x_komp = 1 AND y_komp = TRUE: THIS.komp;
x_komp = 2 : k
OUT 1 ESAC
```

4.
```
CASE list OF
EMPTY: z
OUT THIS.HEAD ESAC
```

### 3.2.5.4 Operatoren-Tabelle

In der folgenden Tabelle werden zu den Operatoren die Attribut-Typen der Operanden und des Ergebnisses angegeben.

| | | | | |
|---|---|---|---|---|
| OR,XOR<br>AND | BOOL | BOOL | BOOL | log. Disjunktionen<br>log. Konjunktion |
| =<br>≠ | C,S,U,V,H(SA) | | BOOL | Vergleiche |
| ≤,≥<br><,> | INT | INT | BOOL | Ordnungs-Relationen |

| | | | | |
|---|---|---|---|---|
| ≤,≥ | M(ISA) | M(ISA) | BOOL | Mengen-Inklusion |
| IN | SA | M(SA) | BOOL | Element-Inklusion |
| IN | K(Mk) | Mk | BOOL | Schlüssel-Inklusion |
| ID | K(Mk(T)) | Mk(T) | T | Schlüssel-Identifikation |
| + | M(SA) | M(SA) | M(SA) | Mengen-Vereinigung |
| * | | | | Mengen-Durchschnitt |
| +,- | INT | INT | INT | Addition, Subtraktion |
| * | | | | Multiplikation |
| // | | | | Ganzzahl-Division |
| MOD | | | | Modulo-Bildung |
| NOT | | BOOL | BOOL | Negation |

Abkürzungen:

| | |
|---|---|
| G | Grundtypen (INT, BOOL, IDENT) |
| A | Ausschnitt |
| U | Vereinigungs_Typ |
| V | Verbund_Typ |
| S | skalarer_Typ |
| ISA | INT, S, A |
| T | beliebiger Attribut_Typ |
| M(X) | Mengen_Typ über dem Typ X (ungeordnet, ohne Schlüssel) |
| Mk(X) | Mengen_Typ über dem Typ X (mit Schlüssel) |
| K(Mk(X)) | Schlüssel-Typ von Mk(X) |

### 3.2.6 Notationelle Erweiterungen

Im folgenden werden einige notationelle Erweiterungen der in 3.2.5 definierten Beschreibungssprache eingeführt. Sie erlauben eine vereinfachte Angabe der semantischen Regeln in vielen in der Praxis auftretenden Fällen. Die Erweiterungen sind prinzipiell auf die Notationen der strikten Beschreibungssprache zurückführbar. Sie können ausserdem zu einer Vereinfachung der Implementierung semantischer Funktionen im erzeugten Übersetzer ausgenutzt werden.

### 3.2.6.1 Attribut-Übernahme

```
Semantische_Regel:
   TRANSFER [ Attribut_Bezeichner || <,> ]
   [ WITH ( Nichtterminal_Benennung || <,> ) ] .
```

Dies ist eine Kurzschreibweise für eine oder mehrere semantische Identitäts-Funktionen:

Sei p = X :..U..V..W.. eine Regel aus P,
a sei ein abgeleitetes Attribut von X und U,
b sei ein erworbenes Attribut von X, V und W,
c sei ein erworbenes Attribut von X und U
und gleich bezeichnete Attribute seien vom gleichen Typ.
Dann ersetzt

(*) TRANSFER a,b,c WITH U,V,W

die semantischen Regeln

X.a:=U.a; V.b:=X.b; W.b:=X.b; U.c:=X.c

Sind U, V und W die einzigen Nichtterminal-Symbole der rechten Seite der Regel p, so hat

TRANSFER a,b,c

die gleiche Bedeutung wie (*).

Haben U,V,W und X ausser a, b und c keine weiteren gleich bezeichneten Attribute, so hat

TRANSFER WITH U,V,W

die gleiche Bedeutung wie (*).

### 3.2.6.2 Attribut-Benennungen in erweitertem Kontext

Mit den folgenden Konstruktionen kann in Ausdrücken auf Attribute von Nichtterminal-Symbolen Bezug genommen werden, die nicht in der zugeordneten syntaktischen Regel stehen. Sie erlauben die Beschreibung semantischer Abhängigkeiten in

einem grösseren Kontext, ohne dass eine Weitergabe der Attribute über mehrere Regeln explizit formuliert werden muss.

Wir unterscheiden zwischen einem Konstituenten_Attribut, das sich auf Attribute von nicht notwendig direkten Nachfolge-Knoten bezieht, und einem äusseren_Attribut, das sich auf ein Attribut eines nicht notwendig direkten Vorgänger-Knotens bezieht.

```
Attribut_Benennung:
   Konstituenten_Attribut | äusseres_Attribut .

Konstituenten_Attribut:
   Nichtterminal_Benennung
   CONSTITUENT Attribut_Benennung |
   [ Nichtterminal_Benennung ]
   CONSTITUENTS Attribut_Benennung .

äusseres_Attribut:
   INCLUDING Attribut_Benennung |
   INCLUDING <(> ( Attribut_Benennung || <,> ) <)>.
```

Der syntaktischen Regel

$$p = Y : X_1 \ldots X_n$$

sei ein semantischer Ausdruck zugeordnet, der ein Konstituenten_Attribut

$$X_i \text{ CONSTITUENT } z.a$$

enthält. Dann muss jede mögliche Ableitungssequenz

$$X_i \Rightarrow s,\ s \in T^* \text{ in}$$
$$X_i \Rightarrow uzv \Rightarrow s,\ u,v \in V^*$$

zerlegbar sein, wobei im ersten Schritt ($X_i \Rightarrow uzv$) keine Regel z:w angewendet wird. z darf in u und v nicht enthalten sein und daraus nicht oder nur unter Anwendung der Regel p ableitbar sein. So ist in jedem Fall der Nachfolge-Knoten der Klasse z, dessen Attribut a das Konstituenten_Attribut benennt, eindeutig bestimmt.

Beispiel:

```
RULE prozedurvereinbarung :
        "proc" prozedurspezifikation "=" anweisung
SEMANTIC
  ...
  anweisung.art:=
    prozedurspezifikation CONSTITUENT ergebnisart.art
  ...
END;

RULE prozedurspezifikation :
        [ formale_parameter ] ergebnisart
SEMANTIC ...
END;

RULE ergebnisart : artangabe
SEMANTIC TRANSFER art
END
```

Hat das Konstituenten_Attribut die Form

$X_i$ CONSTITUENTS Z.a
bzw. CONSTITUENTS Z.a

und sei

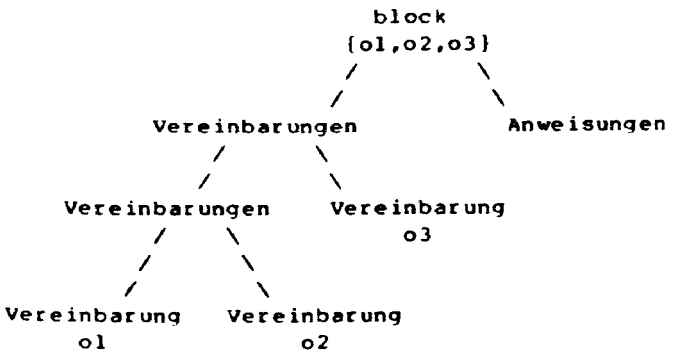
$$X_i \Rightarrow u_1\ Z_1\ u_2 \ldots u_n\ Z_n\ v \Rightarrow s$$
$$\text{bzw. } Y \Rightarrow u_1\ Z_1\ u_2 \ldots u_n\ Z_n\ v \Rightarrow s$$

eine Ableitungssequenz, für die gilt: Z ist in $u_k$ und v nicht enthalten und daraus nicht oder nur unter Anwendung der Regel p ableitbar. Dann benennt das Konstituenten_Attribut die Menge der Werte der Attribute $Z_j$.a. (Diese Form kann nur innerhalb von Standard-Notationen für Mengen-Attributen auftreten.) Die Reihenfolge der Mengenelemente wird ggf. durch die Reihenfolge der $Z_j$ bestimmt.

Beispiel:

```
RULE block: "begin" Vereinbarungen ";" Anweisungen "end"
SEMANTIC
   block.definitionen:=
   defliste(Vereinbarungen
            CONSTITUENTS Vereinbarung.Objekt)
END
```

Ein möglicher Teilbaum des attributierten Strukturbaumes kann dann folgende Form haben (o1, o2, o3 bezeichnen die Attribut-Werte):




Mit einem äusseren_Attribut wird auf ein Attribut eines nicht notwendig direkten Vorgänger-Knotens Bezug genommen. Sei der syntaktischen Regel p = Y:w ein semantischer Ausdruck zugeordnet, der ein äusseres_Attribut der Form

INCLUDING Z.a

enthält. Dann muss jede Ableitungssequenz S => uYv zerlegt werden können in

$$S \Rightarrow u_1\ Z\ v_1 \Rightarrow u_2\ W_2\ v_2 \Rightarrow \ldots \Rightarrow u_n\ W_n\ v_n \Rightarrow u\ Y\ v$$

mit $W_i \neq Y$, $W_i \neq Z$ und $r = W_i : x\ W_{i+1}\ y \in P$. (Z=Y ist zulässig.) Das äussere Attribut bezeichnet das Attribut a des nächsten Oberknotens der Knotenklasse Z zu einem Knoten der Klasse Y. In der Form

INCLUDING $(z_1.a_1,\ldots,z_n.a_n)$

bezeichnet es das Attribut $a_i$ des nächsten Oberknotens einer der angegebenen Knotenklassen $z_i$. Alle $z_i$ müssen verschieden sein.

Beispiel:

```
RULE Fallmarke: Konstante
SEMANTIC
   Konstante.Art =
            INCLUDING Fallunterscheidung.Auswahlart
END
```

Semantische Regeln, die Konstituenten_Attribute oder äussere_Attribute enthalten, können automatisch durch Einführung geeigneter zusätzlicher Attribute auf die strikte Form zurückgeführt werden. Wir demonstrieren diese Transformation an dem oben angegebenen Beispiel zum äusseren Attribut:

```
RULE Fallunterscheidung: "case" Auswahlformel "of"
        ( Fall // ";" ) [ "out" Formel ] "esac"
SEMANTIC
   Fallunterscheidung.Auswahlart := Auswahlformel.Art;
   Fall.impl_Auswahlart := Auswahlformel.Art
END

RULE Fall: ( Fallmarke ":" )* Formel
SEMANTIC
   Fallmarke.impl_Auswahlart := Fall.impl_Auswahlart
END

RULE Fallmarke: Konstante
SEMANTIC
   Konstante.Art = Fallmarke.impl_Auswahlart
END
```

Die Attribute impl_Auswahlart und die semantischen Regeln, in denen sie auftreten, wurden durch die Transformation eingeführt. Die letzte Regel ist der des obigen Beispiels gleichwertig.

Die Konstituenten Attribute können durch entsprechende Transformationen zurückgeführt werden. (Diese Transformationen werden bei der Analyse der semantischen Abhängigkei-

ten durchgeführt. Für die Realisierung der Attribute im Übersetzer werden jedoch die Abhängigkeiten zwischen Attributen eines weiteren Kontexts direkt berücksichtigt.)

### 3.2.6.2 Attribut-Benennungen zu optionalen Nichtterminal-Symbolen

Auf Attribute von Nichtterminal-Symbolen, die in der syntaktischen Regel als optional gekennzeichnet sind, wird in semantischen Ausdrücken durch ein optionales Attribut Bezug genommen:

```
einfacher_Ausdruck:
   optionales_Attribut.

optionales_Attribut:
   <[> Attribut_Benennung <,> Ausdruck <]> .
```

Wird das Nichtterminal-Symbol bei einer Anwendung der syntaktischen Regel berücksichtigt, so bestimmt die Attribut_Benennung (andernfalls der angegebene Ausdruck) den Wert des optionalen Attributes.

### 3.2.6.3 Funktions-Definitionen und -Aufrufe

```
Definition:
   Funktions_Definition .

Funktions_Definition:
   FUNCTION Funktions_Bezeichner
   [ <(> ( Parameter_Spezifikation || <,> ) <)> ]
   Typ_Angabe <:> Ausdruck .

Parameter_Spezifikation:
   ( Parameter_Bezeichner || <,> ) <:> Typ_Angabe .

einfacher_Ausdruck:
   Funktionsaufruf .
```

```
Funktionsaufruf:
   Funktions_Bezeichner [ <(> ( Ausdruck || <,> ) <)> ] .

Attribut_Benennung:
   Parameter_Bezeichner |
   Parameter_Bezeichner <.> Komponenten_Benennung .
```

Funktionen werden unabhängig von den Regeln der Sprachdefinition definiert. Ihr Gültigkeitsbereich ist die gesamte Sprachdefinition. Sie sind Abbildungen $at_1 X...X at_n \rightarrow at$, wobei $at_i$ die Parameter-Typen und at der Ergebnis-Typ sind. Der Ausdruck enthält nur Attribut_Benennungen der oben angegebenen Form. Die Parameter_Bezeichner können nur innerhalb des Ausdrucks auftreten.

Die Bedeutung eines Funktionsaufrufs wird durch textuelle Substitution des definierten Funktions-Rumpf beschrieben bei gleichzeitiger Ersetzung der Parameter_Bezeichner durch die entsprechenden im Aufruf angegebenen Ausdrücke. (Die von Programmiersprachen bekannten Probleme bei der textuellen Ersetzung bestehen hier nicht, da alle Ausdrücke statischer Natur sind und keine Seiteneffekte auftreten können.) Funktionen können auch rekursiv definiert werden. Die Bedeutung von Aufrufen rekursiver Funktionen wird wie oben durch eine textuelle Substitution beschrieben, die jedoch von den Werten einiger Parameter abhängt.

### 3.2.7 Formulierung der dynamischen Semantik

Die dynamische Semantik einer Sprachdefinition wird durch Abbildung der Sprachelemente auf die Eigenschaften der abstrakten Maschine AMICO (siehe Kapitel 5) beschrieben.

Die Sprache zu AMICO ist durch vorgegebene Attribut-Typen und Funktionen so in die Beschreibungssprache integriert, dass überprüft werden kann, ob jeder korrekte Satz der definierten Sprache auf einen syntaktisch korrekten Satz der Zielsprache abgebildet wird. Man nutzt dazu die strukturelle Übereinstimmung der Attribut-Typ-Definitionen mit den Definitionen der abstrakten Syntax der Zielsprache aus. Zu syntaktischen Regeln können nach folgendem Muster Attribut-Typen definiert werden:

```
A : B C D          TYPE A : STRUCT( b:B, c:C, d:D)
A : B | C | D      TYPE A : UNION( B, C, D)
A : ( B )*         TYPE A : SETOF B LINEAR
```

Beispiel:

```
Programm: ( Schablonendefinition || <;> ) <//>
          ( Segmentdefinition || <;> ) .
```

entspricht

```
TYPE C_Code : STRUCT( C_Sch: C_Schablonen,
                      C_Seg: C_Segmente ) ;
TYPE C_Segmente : SETOF C_Segment LINEAR ;
TYPE C_Schablonen : SETOF C_Schablone LINEAR
```

Der Anhang A enthält die vollständige Liste der vordefinierten Typen. Die Werte dieser Typen sind die Zeichenreihen, die aus den Nichtterminal-Symbolen der Regeln der Zielsprache ableitbar sind. Werden die Attribute hinsichtlich ihrer Typen korrekt angewendet, so führt ihre Zusammensetzung auch zu korrekten Sätzen der Zielsprache.

Die Wirkung der Elemente der definierten Sprache wird durch die Zuordnung von Attributen dieser Typen und durch die zugehörigen semantischen Regeln beschrieben. Wir fordern, dass dem Zielsymbol der Grammatik genau ein Attribut des Attribut-Typs C_Code zugeordnet ist, dessen Wert die Wirkung des gesamten Programmes beschreibt.

Die Einbeziehung der definierten Zwischensprache in das Beschreibungsverfahren erlaubt die Formulierung der statischen und dynamischen Semantik in geschlossener Form mit den gleichen Beschreibungsmitteln.

Bei der Analyse der semantischen Abhängigkeiten können für die Attribute zur Beschreibung der statischen und der dynamischen Semantik die gleichen Verfahren angewendet werden. Die einheitliche Formulierung ermöglicht ausserdem die Beschreibung von Optimierungen des Zwischencodes durch semantische Regeln.

Diese Art der Integration der dynamischen Semantik gewährleistet ausserdem die Anpassbarkeit des Übersetzer-erzeugenden Systems an andere Zielsetzungen. Sollen zum Beispiel Sprachen spezieller Anwendungsgebiete definiert wer-

den, die nicht der hier betrachteten Sprachklasse angehören, so sind nur die Attribut-Typdefinitionen der Beschreibung der Zwischensprache geeignet zu ersetzen. Die Analyse- und Generierungsverfahren des Übersetzer-erzeugenden Systems werden davon nicht betroffen. Auf die gleiche Weise kann auch die dynamische Semantik direkt durch die Eigenschaften einer konkreten Maschine beschrieben werden - mit dem Ziel effizientere Übersetzer zu generieren. Dadurch verliert natürlich die Sprachdefinition ihre Allgemeingültigkeit und der generierte Übersetzer seine Portabilitäts-Eigenschaft.

# 4 Erzeugung des Übersetzers

In diesem Kapitel stellen wir Verfahren vor, mit denen Sprachdefinitionen in Übersetzer transformiert werden können. Die Verfahren sind allgemein auf attributierte Grammatiken anwendbar. Bei ihrer Beschreibung beziehen wir uns auf die in Kapitel 3 definierte Eingabesprache für unser Übersetzer-erzeugendes System.

## 4.1 Konzeption

Ein Übersetzer-erzeugendes System, das aus Sprachdefinitionen in der Form attributierter Grammatiken Übersetzer generiert, muss folgende Aufgaben lösen:

a) Es muss prüfen, ob

   a1) die Sprachdefinition gemäss der Beschreibungssprache korrekt ist,

   a2) die syntaktischen Definitionen die Voraussetzungen des anzuwendenden Zerteilungs-Verfahrens erfüllen,

   a3) die semantischen Definitionen vollständig, konsistent und zyklenfrei sind und die Voraussetzungen des anzuwendenden Analyse-Verfahrens erfüllen,

   a4) die dynamische Semantik durch syntaktisch korrekte Programme der Zielsprache beschrieben wird.

b) Es muss Übersetzer-Algorithmen für die

   b1) Symbolentschlüsselung,

   b2) Zerteilung,

   b3) semantische Analyse und

   b4) Synthese der Programme in der Zielsprache

   erzeugen.

c) Es muss Datenobjekte des Übersetzers zur Implementierung

c1) des Strukturbaumes und

c2) der Attribute

definieren.

Für einige dieser Aufgaben sind systematische Lösungsmethoden hinreichend bekannt; sie werden hier nicht weiter diskutiert.

zu a) Man kann die Korrektheit einer Sprachdefinition im Sinne der Beschreibungssprache (a1) mit den gleichen Verfahren feststellen, mit denen die erzeugten Übersetzer prüfen, ob ein Programm der definierten Sprache angehört. a2 und a3 werden im wesentlichen als Eingangsvoraussetzungen bei der Erzeugung der Übersetzer-Algorithmen (b) geprüft. In Abschnitt 3.2.7 haben wir gezeigt, wie die Prüfung der strukturellen Korrektheit der Sätze in der Zielsprache (a4) auf die Typ-Prüfung für Attribute zurückgeführt und mit den für a1 angewendeten Verfahren gelöst wird.

zu b) Wir gehen davon aus, dass aus den syntaktischen Definitionen der Sprachdefinition mit Hilfe von Generatoren geeignete Übersetzer-Algorithmen für die Symbolentschlüsselung und Zerteilung erzeugt werden (b1,b2). Die Schnittstelle zu den Analyse- und Synthese-Algorithmen ist bestimmt durch die angewendeten Ableitungs-Regeln, die im übersetzten Programm enthaltenen semantisch relevanten Grundsymbole (Bezeichner, Zahlen, Zeichenreihen, usw.) und der Zuordnung ihrer Verschlüsselung zu ihrer textuellen Darstellung (Symbol-Tabelle).

In Abschnitt 4.2 beschreiben wir ein graphentheoretisches Verfahren, das aus den semantischen Regeln Tabellen (Besuchssequenzen) erzeugt, die den Ablauf der Algorithmen zur semantischen Analyse und Synthese steuern (b3,b4). Das Verfahren haben wir aus der in [Knu68] beschriebenen Methode zur Prüfung der semantischen Auswertbarkeit (Zyklenfreiheit) weiterentwickelt.

In Abschnitt 4.3 geben wir einen Keller-Automaten an, der den Strukturbaum durch Zufügen semantischer Informationen zu einem attributierten Strukturbaum vervollständigt. Die Übergangstabelle des Automaten ist die Menge der Besuchssequenzen. Er ist mit den in [Sch76] beschriebenen "kletternden Baum-Transduktoren" vergleichbar. Sie sind darauf beschränkt, Bäume nach einer fest vorgegebenen Strategie (z.B. links-abwärts) zu durchlaufen, während unser Automat Bäume auf Wegen durchläuft, die durch die Besuchssequenzen bestimmt sind.

In den Abschnitten 4.2 und 4.3 zeigen wir ausserdem, wie der Automat an verschiedene Zerteilungs-Algorithmen unterschiedlich angeschlossen werden kann (Beginn der Analyse nach vollständiger Zerteilung oder verzahnt mit zielbezogener oder quellbezogener Zerteilung).

Mit diesem Algorithmus werden auch die Attribute bestimmt, die die dynamische Semantik beschreiben. Zusätzliche Algorithmen für die Code-Erzeugung entfallen deshalb (b4). (Die Möglichkeiten zur technischen Verbesserung des Analyse-Algorithmus für diese spezielle Anwendung untersuchen wir hier nicht.)

zu c) Der Algorithmus zur semantischen Analyse setzt voraus, dass der Strukturbaum als leicht zugängliche Datenstruktur (Geflecht im Kernspeicher) implementiert wird. In Abschnitt 4.4 zeigen wir, wie wir die Struktur der Baumknoten aus der Sprachdefinition ermitteln, und geben an, wie der Speicheraufwand für den Strukturbaum verringert werden kann (c1).

Die Attribute der Nichtterminal-Symbole werden durch Datenobjekte implementiert, die den Knoten des Strukturbaumes zugeordnet sind. In Abschnitt 4.5 zeigen wir, wie alle Attribut-Typen, die in einer Sprachdefinition definiert werden können, auf im Übersetzer benutzte Datentypen abgebildet werden (c2). Zusammengesetzte Attribute werden durch Zugriffswege auf Datenobjekte ausserhalb des Strukturbaumes implementiert. Dabei nutzen wir aus, dass die Beschreibungssprache statisch (variablen- und seiteneffektfrei) ist. Auf diese Weise sind auch umfangreiche Attribute (z.B. Definitions-Tabellen) praktikabel implementierbar.

In Abschnitt 4.6 zeigen wir, wie der "Transport" von Attributen zwischen nicht benachbarten Baumknoten unter Ausnutzung der Besuchssequenzen systematisch vereinfacht werden kann. (Dieses Problem wird in [Ga74] weniger flexibel durch explizite Spezifikationen von Ablaufstrukturen in der Sprachdefinition gelöst.)

## 4.2 Systematische Analyse semantischer Abhängigkeiten

Das hier beschriebene Verfahren basiert auf dem Ansatz von Knuth in [Kn68], der Abhängigkeiten zwischen den Attributen durch Graphen darstellt. Knuth zeigt, wie aus der Untersuchung dieser Graphen festgestellt werden kann, ob die Semantik der Sprache zyklenfrei definiert ist, so dass zu jedem Satz der Sprache alle Attribute bestimmt werden können. Eine Sprache mit dieser Eigenschaft nennt er dann "semantisch wohldefiniert". Dieses Verfahren ist hier in modifizierter Form enthalten.

Die abstrakte Struktur eines Programmes zu einer attributierten Grammatik wird durch einen Strukturbaum dargestellt, und durch Zufügen von Attributen zu einem attributierten Strukturbaum (ASB) ergänzt. Um alle Attribute eines ASB zu bestimmen, muss man ihn oder einige seiner Unterbäume mehrfach in unterschiedlicher Richtung durchlaufen. Unser Verfahren ordnet jedem Knoten-Typ p (zu einer Regel p) eine "Besuchssequenz" zu, die beschreibt, in welcher Reihenfolge beim Durchlauf durch den ASB von einem Knoten des Typs p zu den Knoten in seiner direkten Nachbarschaft übergegangen wird. (Dabei können Knoten mehrfach besucht werden.) Ausserdem wird ermittelt, welche Attribute zwischen zwei solchen Besuchen bestimmt werden können.

Das Übersetzer-erzeugende System bestimmt diese Besuchssequenzen aus der Grammatik und den Abhängigkeiten zwischen den Attributen. Sie legen eindeutig fest, wie jeder zulässige ASB zu einer bestimmten Sprache durchlaufen werden kann, so dass alle Attribute berechnet werden können. Aus den Besuchssequenzen werden Übersetzer-Tabellen erzeugt, die den Ablauf der semantischen Analyse steuern. Sie könnten ebenso in Kontrollstrukturen eines Algorithmus transformiert

werden (z.B. rekursive Prozeduren oder Koroutinen).

Dieses Verfahren wurde in [Ka75] und [Ka76] veröffentlicht. Der Anhang B enthält ein Beispiel für die Graphen und Besuchssequenzen zu den Regeln einer einfachen Sprache.

### 4.2.1 Notation und Voraussetzungen

Die hier angegebenen Voraussetzungen, Schreibweisen und Abkürzungen entsprechen im wesentlichen denen aus dem 3. Kapitel:

| | |
|---|---|
| $G=(V,T,P,Z)$ | (kontext-freie) Grammatik |
| $V$ | Vokabular |
| $T \subset V$ | Terminal-Symbole |
| $P$ | Menge der syntaktischen Regeln |
| $Z \in V \setminus T$ | Zielsymbol |
| $U,W,X,Y,Z \in V \setminus T$ | beliebige Nichtterminal-Symbole |
| $u,v,w,x,y \in V^*$ | beliebige Zeichenfolgen |
| $p,q,r,s,t \in P$ | syntaktische Regeln |
| $p = Y:X_1....X_n$ | Notation einer Regel p. Dabei werden Terminal-Symbole auf der rechten Seite nicht angegeben, da sie in diesem Zusammenhang keine Bedeutung haben. |
| $M,N$ | Mengen von Attributen |
| $A_X$ | Menge der Attribute von X |
| $AD_X$ | Menge der abgeleiteten (derived) Attribute von X |
| $AI_X$ | Menge der erworbenen (inherited) Attribute von X |
| $a=q(a_1,...,a_n)$ | semantische Vorschrift zur Berechnung des Wertes von a (q entspricht einer semantischen Funktion $SF_{p,a}$ in Kapitel 3. Hier sind jedoch die Typen der Attribute unbedeutend; es werden nur die Abhängigkeiten betrachtet.) |
| $C_p$ | gerichteter Graph zu einer Regel p, der die durch alle q gegebenen Abhängigkeiten beschreibt |
| $D_p$ | durch Hüllenbildung aus $C_p$ gewonnener Graph |
| $E_p$ | durch Hüllenbildung aus $D_p$ gewonnener Graph |

| | |
|---|---|
| $k=(a_1,a_2)$ | Kante in einem Graphen von $a_1$ nach $a_2$ |
| $w(a)=(up,down)$ | wertepaar, das den Attribut a in einer $E_p$ zugeordnet ist |
| $b_{i,j}$ | j-ter Besuch eines Knotens $X_i$ |
| $F_p$ | Besuchssequenz (Folge von Attributen und Besuchen) |

(Wenn die Zuordnung zu einer Regel p unbedeutend oder aus dem Zusammenhang klar ist, wird der Index p weggelassen.)

Für die Grammatik und die Attribute gelten die Voraussetzungen aus Abschnitt 3.2.1: G ist eine reduzierte kontext-freie Grammatik, die $AI_X$ und $AD_X$ sind disjunkt und überdecken $A_X$, und $AI_Z$ ist leer. In den Abschnitten 4.2.5 bis 4.2.8 wird die "semantische Wohldefiniertheit" der Sprache vorausgesetzt. In Abschnitt 4.2.4 definieren wir diese Eigenschaft und geben ein Verfahren zu ihrer Prüfung an.

## 4.2.2 Strategie des Baum-Durchlaufes

Nimmt man an, dass der ASB vollständig aufgebaut ist, bevor die Attribute berechnet werden, so kann man zu jeder Regel $p = Y:X_1...X_n$ eine "Besuchssequenz" angeben, die die Wege durch jeden Knoten der Klasse Y und des Typs p und die bei jedem Besuch des Knotens berechenbaren Attribute bestimmt. Hängt die Berechnung von Attributen ab, deren Berechnungsvorschriften nicht p zugeordnet sind ($a \in AI_Y \setminus AD_{X_i}$), so wird der entsprechende, benachbarte Knoten besucht ($X_i$ oder der Vorgänger von Y).

Unser Verfahren erlaubt es darüber hinaus, die Besuchssequenzen so zu bestimmen, dass schon beim Aufbau des ASB Attribute berechnet werden können. In diesem Fall bestimmt die Ordnung, in der der Baum aufgebaut wird, die Reihenfolge der Besuche im ersten Durchlauf. Derartige Annahmen wirken sich nur auf den in Abschnitt 4.2.6 (Bewertung der Abhängigkeits-Graphen) beschriebenen Teil unseres Verfahrens aus. Wir nehmen dort an, dass der ASB von den Endknoten zur Wurzel hin aufgebaut wird (bottom-up) und dass dabei schon Attribute bestimmt werden, soweit es die Abhängigkeiten zulassen. Im Abschnitt 4.2.8 wird angegeben, wie das Verfahren zu modifizieren ist, falls eine andere Durchlauf-

strategie vorgegeben wird (z.B. top-down-Aufbau des ASB, oder Ermittlung der Attribute nach dem vollständigen Aufbau des ASB).

### 4.2.3 Abhängigkeits-Graphen

Zu jeder Regel $p = Y:X_1...X_n$ sind durch die Berechnungsvorschriften für die Attribute Abhängigkeiten zwischen den Attributen von $Y,X_1,...,X_n$ definiert. Es wird nun zu jeder Regel p ein gerichteter Graph $C_p$ konstruiert, der diese Abhängigkeiten beschreibt. Die Knoten von $C_p$ sind alle Attribute von $Y, X_1...X_n$. In $C_p$ führt genau dann eine Kante $k=(a_i,a)$ von $a_i$ nach a, wenn es zu p eine Berechnungsvorschrift $SF_{p,a}$ für a gibt, die von $a_i$ abhängt: $a := g(a_1, ...,a_i,...a_m)$. Die semantischen Relationen $SR_p$ können in dieses Verfahren einbezogen werden: Für jede $SR_p$ ordnet man Y ein neues abgeleitetes Attribut zu, dessen Berechnung von den Attributen in $SR_p$ abhängt. Diese Attribute haben keinen Einfluss auf die Abhängigkeiten zwischen den übrigen Attributen - sie stellen nur sicher, dass die Auswertung der $SR_p$ in den Besuchssequenzen berücksichtigt wird.

Da vorausgesetzt wird, dass alle Attribute entweder abgeleitet oder erworben sind, hat jeder Graph $C_p$ folgende Eigenschaften:

a) Nur für $a \in AD_Y \backslash / AI_{X_1} \backslash / ... \backslash / AI_{X_n}$ gibt es zu der Regel p Berechnungsvorschriften. In $C_p$ münden Kanten deshalb nur in diese Knoten.

b) Ist in einer Berechnungsvorschrift a:=g g eine konstante Funktion (d.h. a wird unabhängig von anderen Attribut-Werten berechnet), so münden in $C_p$ keine Kanten in a.

In den folgenden Abschnitten werden wir die Graphen C so vervollständigen, dass sie auch alle Abhängigkeiten zwischen Attributen beschreiben, die durch jede mögliche Zusammensetzung solcher Graphen zu einem ASB verursacht werden können. Dieser Vorgang ist als Hüllenbildung über den Graphen bezüglich der Kontext-Abhängigkeiten zu verstehen.

### 4.2.4 Semantische Wohldefiniertheit

Wir nennen nach [Kn68] eine Grammatik "semantisch wohldefiniert", wenn zu ihr kein ASB konstruiert werden kann, in dem einige Attribute zyklisch voneinander abhängen, und deshalb nicht berechnet werden können. Diese Eigenschaft der Grammatik kann mit Hilfe der Graphen C überprüft werden. Dazu vervollständigen wir jeden Graphen $C_p$ zu einer Regel $p = Y:X_1 \ldots X_n$, so dass er auch alle Abhängigkeiten zwischen den Attributen von $Y, X_1, \ldots, X_n$ beschreibt, die durch jede mögliche Ableitung der $X_i$ gegeben sind (Regeln $q = X_i:u$).

Zu jedem Graphen $C_p$ wird ein Graph $D_p$ konstruiert. Die Knotenmengen von $C_p$ und $D_p$ sind gleich. $D_p$ enthält eine Kante $k=(a_1,a_2)$ von $a_1$ nach $a_2$ genau dann, wenn k in $C_p$ enthalten ist oder wenn gilt $a_1 \in A_{X_i}$ und $a_2 \in AD_{X_i}$, $1 \leq i \leq n$ und es gibt einen Graphen $D_q$ zu einer Regel $q = X_i:u$, in dem ein Weg von $a_1$ nach $a_2$ führt. Die in $D_p$ aber nicht in $C_p$ enthaltenen Kanten nennen wir "durch die Regeln q in $D_p$ induziert".

Sind alle Graphen $D_p$ zyklenfrei, so kann man keinen ASB konstruieren, der einen Zyklus enthält - die Grammatik ist semantisch wohldefiniert. Enthält ein Graph $D_p$ zu einer Regel $p = Y:X_1 \ldots X_n$ einen Zyklus, so sind folgende Fälle zu unterscheiden:

a) Auf dem Zyklus liegen mindestens zwei Kanten, die durch verschiedene Regeln $q = X_i:u$ und $r = X_i:v$ in $D_p$ induziert sind. In diesem Fall führt keine der Ableitungssequenzen

$$Y \Rightarrow X_1 \ldots X_i \ldots X_n \Rightarrow X_1 \ldots u \ldots X_n$$
$$Y \Rightarrow X_1 \ldots X_i \ldots X_n \Rightarrow X_1 \ldots v \ldots X_n$$

zu einem Zyklus im ASB. Falls keine weiteren Zyklen auftreten, ist die Grammatik semantisch wohldefiniert. Der Zyklus in $D_p$ kann durch eine Transformation der Grammatik beseitigt werden: Man ersetzt die Regel p durch zwei Regeln

$p1 = Y:X_1 \ldots v \ldots X_n$ und $p2 = Y:X_1 \ldots v \ldots X_n$.

b) In allen anderen Fällen läßt sich ein ASB konstruieren, der einen Zyklus enthält. Die Grammatik ist dann nicht semantisch wohldefiniert.

Alle weiteren Untersuchungen setzen voraus, dass alle Graphen $D_p$ zyklenfrei sind.

### 4.2.5 Erweiterung der Abhängigkeits-Graphen

Die in Abschnitt 4.2.4 konstruierten Graphen D werden durch weitere Hüllenbildung zu Graphen E vervollständigt. Die Knotenmengen jedes $D_p$ und $E_p$ sind gleich. Ein Graph $E_p$ zu einer Regel $p = Y:X_1 \ldots X_n$ enthält genau dann eine Kante $k=(a_1,a_2)$, wenn eine der folgenden Bedingungen erfüllt ist:

a) k ist in $D_p$ enthalten.

b) $a_1 \in A_Y$ und $a_2 \in AI_Y$
und es gibt einen Graphen $E_q$ zu einer Regel $q = Z:uYv$, in dem ein Weg von $a_1$ nach $a_2$ führt. Diese Kanten beschreiben die Abhängigkeiten zwischen Attributen von Y, die durch jede mögliche Einbettung von Y in einen Kontext verursacht werden.

c) $a_1 \in A_Y$ und $a_2 \in AD_Y$
und es gibt einen Graphen $D_\alpha$ zu einer Regel $\alpha = Y:w$, der einen Weg von $a_1$ nach $a_2$ enthält.

d) $a_1 \in A_{X_i}$ und $a_2 \in AI_{X_i}$
und es gibt einen Graphen $E_q$ zu einer Regel $q = U : x\ X_i\ y$, der einen Weg von $a_1$ nach $a_2$ enthält.

Die nach c) und d) eingefügten Kanten beschreiben Abhängigkeiten zwischen Attributen, die bestehen, falls Y bzw. $X_i$ in einem anderen als durch die Regel p gegebenen Kontext stehen. (Diese Kanten gewährleisten, dass die Bewertung der Graphen und die Ermittlung der Besuchssequenzen zunächst für jeden Graphen $E_p$ unabhängig durchgeführt werden kann.) Wir nennen die in $E_p$ aber nicht in $D_p$ enthaltenen Kanten "durch die Regeln q in $E_p$ induziert".

Es ist möglich, dass die so konstruierten Graphen E Zyklen enthalten. In diesem Fall kann man jedoch die Grammatik so transformieren, dass ein solcher Graph durch mehrere zyklenfreie Graphen ersetzt wird. Da alle Graphen D zyklenfrei sind, muss ein Zyklus in einem Graphen $E_p$ minde-

stens zwei Kanten enthalten, die durch verschiedene Regeln q in $E_p$ induziert sind. Setzt man nun die Regel q in p ein und/oder führt neue Nichtterminal-Symbole für ein $X_i$ mit entsprechenden neuen Regeln ein, so enthalten die Graphen E den ursprünglichen Zyklus nicht mehr.

Bei den folgenden Untersuchungen setzen wir voraus, dass alle Graphen E zyklenfrei sind.

### 4.2.6 Bewertung der Abhängigkeits-Graphen

In diesem Abschnitt werden den Attributen a in den Graphen $E_p$ Wertepaare (up,down) = w(a) zugeordnet. (Wir unterscheiden hier die Formulierungen: ein Attribut (Knoten im Abhängigkeits-Graph) wird durch ein Wertepaar bewertet; ein Attribut im ASB wird berechnet.)

Sei $p = Y:X_1 \ldots X_n$ und $a \in A_Y \backslash / A_{X_1} \backslash / \ldots \backslash / A_{X_n}$
dann bedeutet $w(a) = (m,n)$:

a kann berechnet werden, bevor im ASB von Y aus zum m+1-ten mal der Vorgänger-Knoten besucht wird und bevor zwischen zwei Besuchen des Vorgänger-Knotens (bzw. vor dem 1. Besuch, falls m=0) zum n+1-ten mal ein Knoten $X_i$ besucht wird.

Es gilt

$w(a_1)=(m_1,n_1) = w(a_2)=(m_2,n_2)$
genau dann, wenn

und $m_1=m_2$ und $n_1=n_2$,

$w(a_1)=(m_1,n_1) < w(a_2)=(m_2,n_2)$
genau dann,wenn

$m_1<m_2$ oder ($m_1=m_2$ und $n_1<n_2$).

Die Attribute werden so bewertet, dass $a_1$ vor $a_2$ berechnet werden kann, wenn $w(a_1)<w(a_2)$, und $a_1$ und $a_2$ beim gleichen Besuch berechnet werden können, wenn $w(a_1)=w(a_2)$.

Zunächst wird in einem Graphen $E_p$ zu einer Regel $p = Y:X_1 \ldots X_n$ denjenigen Attributen, die beim ersten Besuch des Knotens Y berechnet werden können, das Wertepaar (0,0) zugeordnet. Dies sind alle Attribute $a \in AD_Y \backslash / AI_{X_i}$, $1 \le i \le n$,

zu denen es eine konstante Berechnungsvorschrift a:=α gibt.

Aufgrund der gewählten Strategie für den Durchlauf durch den ASB kann in $E_p$ auch allen Attributen von $X_i$, die ohne einen Besuch des Vorgänger-Knotens berechnet werden können, das Wertepaar (0,0) zugeordnet werden. Dies sind die Attribute $a \in AD_{X_i}$, in die in $E_p$ keine Kanten münden. (Zu anderen Durchlaufstrategien siehe 4.2.9.)

Wir nehmen zunächst an, dass alle Attribute $a \in AI_Y$, in die keine Kanten münden, nach dem ersten Besuch des Vorgänger-Knotens berechnet sind, und ordnen ihnen das Wertepaar (1,0) zu. Nach der Bewertung aller Graphen wird diese Annahme in Abschnitt 4.2.8 überprüft und - falls erforderlich - korrigiert.

Alle übrigen Attribute in $E_p$ werden nun sukzessive wie folgt bewertet. Sei a ein noch nicht bewertetes Attribut in $E_p$, seien $k_j=(a_j,a)$ alle Kanten, die in a münden, und alle $a_j$ seien bewertet. Dann kann a ein Wertepaar zugeordnet werden. Es sind dabei folgende Fälle zu unterscheiden:

a) Für $a \in AD_Y \setminus AI_{X_i}$, $1 \leq i \leq n$ (es gibt dann eine Berechnungsvorschrift a:=g(....) ), ist $w(a)=\max(w(a_j))$. Die Maximum-Bildung erfolgt gemäss der Relation <.

b) Für $a \in AD_{X_i}$, $1 \leq i \leq n$, sind alle Kanten $k_j$ durch Regeln $q = X_i:v$ induziert, und es ist ein Besuch des Knotens $X_i$ erforderlich, um die zu q angegebene Berechnungsvorschrift für a auszuwerten. Sei

$(m,n') = \max ( ( up(a_j) , down(a_j)+1 ) )$,

und sei n die kleinste Zahl mit $n \geq n'$, und für kein $a' \in AD_{X_r}$ gilt $w(a') = (m,n)$ und $i \neq r$, dann ist $w(a)=(m,n)$.

Hierdurch wird sichergestellt, dass die Besuche verschiedener Knoten $X_i$ und $X_r$ zu verschiedenen down-werten der dabei berechneten Attribute führen, und dass bei einem Besuch eines Knotens $X_i$ möglicherweise mehrere Attribute aus $AD_{X_i}$ berechnet werden können. (Um die Zahl der Besuche einzelner Knoten $X_i$ klein zu halten, ist es zweckmässig, ein Attribut nach dieser Regel erst dann zu bewerten, wenn möglichst viele Attribute $a \in AI_{X_i}$ bewertet sind.)

c) Für $a \in AI_Y$ ist zur Berechnung von a ein Besuch des Vorgänger-Knotens erforderlich. Es ist dann $w(a)=(m,\emptyset)$, wobei m der kleinste Wert ist, für den gilt:

$w(a_j) \leq (m,\emptyset)$, falls $a_j \in AI_Y$, und
$w(a_j) < (m,\emptyset)$ andernfalls.

### 4.2.7 Aufstellung der Besuchssequenzen

Sind in einem Graphen $E_p$ zu einer Regel $p = Y:X_1 \ldots X_n$ alle Attribute bewertet, so kann festgestellt werden, in welcher Reihenfolge im ASB die Knoten in der Umgebung eines Knotens des Typs p besucht werden müssen, und welche Attribute zwischen je zwei solchen Besuchen berechnet werden können.

Wir stellen zu $E_p$ eine Besuchssequenz $F_p$ auf, die eine lineare Folge von Attributen $a \in AD_Y \backslash / AI_{X_i}$, $1 \leq i \leq n$, (nur für diese gibt es Berechnungs-Vorschriften zu p) und Besuchen $b_{i,r}$ ist. Dabei bezeichnet $b_{i,r}$ den r-ten Besuch des Knoten $X_i$, falls $1 \leq i \leq n$, bzw. des Vorgänger-Knotens von Y, falls $i=\emptyset$. $F_p$ hat die folgenden Eigenschaften:

a) Alle Attribute $a \in AD_Y \backslash / AI_{X_i}$, $1 \leq i \leq n$, treten genau einmal in $F_p$ auf.

b) Für je zwei Attribute $a_1$, $a_2$ gilt: $a_1$ steht in $F_p$ vor $a_2$, falls $w(a_1) < w(a_2)$, oder $w(a_1) = w(a_2)$ und in $C_p$ gibt es einen Weg von $a_1$ nach $a_2$.

c) Zwischen je zwei Attributen $a_1$ und $a_2$ mit $w(a_1) < w(a_2)$ steht ein Besuch $b_{i,r}$. Es gilt $i=\emptyset$, falls es ein Attribut $a_k$ in $E_p$ gibt mit $a_k \in AI_Y$ und $w(a_k) = w(a_1)$.
Anderenfalls gibt es in $E_p$ mindestens ein $a_k \in AD_{X_i}$ mit $w(a_k) = w(a_1)$.
Mit r werden alle Besuche des gleichen Knotens (bei 1 beginnend) in der Reihenfolge ihres Auftretens in $F_p$ durchnumeriert.

d) Gilt für das erste Attribut a in $F_p$ $w(a)=(1,\emptyset)$, so geht ihm ein Besuch $b_{0,1}$ voraus.

e) Auf jedes Attribut $a \in AI_{X_i}$ muss in $F_p$ ein Besuch $b_{i,r}$ (nicht notwendig direkt) folgen.

Die $F_p$ stellen Vorschriften dar zur Berechnung der Attribute von Knoten des Typs p im ASB.

### 4.2.8 Anpassung der Bewertung von Graphen

Sind alle Graphen $E_p$ bewertet, so muss überprüft werden, ob die Annahme aus Abschnitt 4.2.6 gilt, dass alle Attribute $a \in AI_Y$, in die in $E_p$ keine Kanten münden, beim ersten Besuch des Vorgänger-Knotens berechnet werden können. Trifft dies für ein oder mehrere Nichtterminal-Symbole nicht zu, so sind die Bewertungen der Graphen in geeigneter Weise anzupassen, oder die Grammatik ist zu transformieren.

Sei M die Menge der Attribute $a \in AI_Y$, in die in den Graphen $E_p$ zu Regeln $p = Y:u$ keine Kanten münden. Zu jeder Regel $q = Z:vYw$ wird nun eine Menge $N_q$ gebildet mit

$N_q = \{a \mid a \in M$ und a steht in $F_q$ nach dem ersten Besuch von $Y\}$.

Es sind dann folgende Fälle zu unterscheiden:

a) $N_\alpha = \emptyset$ für alle $\alpha$.
Dann gilt die obige Annahme und $E_p$ und $F_p$ bleiben unverändert.

b) Alle $N_\alpha$ sind gleich und nicht leer.
In diesem Fall müssen die Graphen $E_p$ neu bewertet werden. Dabei erhalten die Attribute $a \in M \setminus N_\alpha$ das Wertepaar $(1,\emptyset)$ und $a \in N_\alpha$ das Wertepaar $(2,\emptyset)$. Die Bewertung der übrigen Attribute und die Bestimmung der Besuchssequenz erfolgt wie in den Abschnitten 4.2.6 und 4.2.7 beschrieben.

c) Es gibt mehrere paarweise verschiedene $N_{\alpha_1}, \ldots, N_{\alpha_m}$.
In diesem Fall ist die folgende Transformation der Grammatik erforderlich:

Zu jedem $N_{\alpha_j} \neq \emptyset$ wird die Regel

$q_j = z_j : v\ Y\ w$ ersetzt durch

$q_j' = z_j : v\ Y_j\ w$,

wobei $Y_j$ ein neues Nichtterminal-Symbol der Grammatik ist. Zu jeder Regel $p = Y:u$ wird eine neue Regel

$p_j = Y_j:u$

eingeführt. Die Graphen und Besuchssequenzen zu den Regeln $q_j$ sind gleich denen zu $q$. Die $E_{p_j}$ und $F_{p_j}$ werden wie im Fall b neu bestimmt.

### 4.2.9 Modifikationen des Algorithmus

In Abschnitt 4.2.2 wurden Annahmen zur Strategie gemacht, die beim Aufbau und Durchlaufen des ASB angewendet wird. Wir zeigen hier, wie der Algorithmus modifiziert werden muss, falls stattdessen die folgenden Annahmen gelten:

a) Der ASB wird von der Wurzel zu den Endknoten hin aufgebaut. Dabei werden die Knoten zu einer Regel $p = Y:X_1 \ldots X_n$ in der Reihenfolge $Y, T(X_1), \ldots, T(X_n)$ generiert; $T(X_i)$ bezeichnet den Teilbaum, dessen Wurzel $X_i$ ist. Damit ist die Reihenfolge der ersten Besuche jedes Knoten festgelegt. Sollen beim ersten Besuch eines Knotens der Klasse p alle Berechnungsvorschriften g ausgewertet werden, die keinen erneuten Besuch des Vorgänger-Knotens erfordern, so ist nur der Abschnitt 4.2.6 so zu verändern, dass die Anfangsbewertung der Attribute $aGAD_{X_i}$, in die keine Kanten münden, nicht (0,0) sondern (0,i) ist.

b) Die Auswertung der Berechnungsvorschriften beginnt erst, wenn der ASB vollständig aufgebaut ist. Der Durchlauf durch den ASB beginnt bei seiner Wurzel. In diesem Fall kann auch die Reihenfolge der ersten Besuche der Knoten $X_i$ (zu einer Regel $p = Y:X_1 \ldots X_n$) aus den semantischen Abhängigkeiten bestimmt werden. Dies wird durch folgende Änderungen der Regeln zur Bewertung der Graphen (Abschnitt 4.2.6) berücksichtigt:

Die Attribute $a \in AD_{X_i}$, in die keine Kanten münden, werden zunächst nicht bewertet. Stattdessen wird ein Fall d) wie folgt ergänzt:

d) Ist schon ein Attribut $a_1 \in AD_{X_i}$ bewertet, so wird allen $a \in AD_{X_i}$, in die keine Kanten münden, das Wertepaar $w(a_1)$ zugeordnet; andernfalls das Wertepaar $(u,d)$, wobei $(u,d-1)=\max(w(a_r))$ gilt und $a_r$ sind alle schon bewerteten Attribute in $E_p$. (Um die Anzahl der Besuche von $X_i$ gering zu halten, sollte diese Regel so spät wie möglich angewendet werden.)

## 4.3 Ablaufsteuerung der semantischen Analyse

Die Untersuchung der semantischen Abhängigkeiten liefert als Ergebnis zu jedem Knoten-Typ des Strukturbaumes eine Besuchssequenz, die den zur semantischen Analyse nötigen Baumdurchlauf und die dabei auszuführenden semantischen Aktionen eindeutig beschreibt (Abschnitt 4.2). Aus der Menge der Besuchssequenzen werden die Ablaufstrukturen für den Algorithmus zur semantischen Analyse des Übersetzers ermittelt. Für die Implementierung dieser Ablaufstrukturen kommen grundsätzlich alle Programmstrukturen in Frage, die auf einer Kellerorganisation basieren: z.B. rekursive Prozeduren, rekursive Koroutinen nach dem SIMULA-Klassen-Konzept oder ein Keller-Automat. Für die Anwendung in Übersetzer-erzeugenden Systemen eignet sich das Prinzip des Keller-Automaten am besten: Der Algorithmus selbst ist für jeden Übersetzer gleich, nur die Übergangstabelle (Menge der Besuchssequenzen), die seinen Ablauf steuert, wird für jeden Übersetzer erzeugt. Ausserdem stellt der Keller-Automat die geringsten Anforderungen an die benutzte Implementierungssprache und erbringt im allgemeinen die besten Ergebnisse hinsichtlich der Speicher- und Zeiteffizienz, denn der notwendige Aufwand zur Organisation des Kellers und der Übergangstabelle ist gering und kontrollierbar.

Die Arbeitsweise des Automaten wird beschrieben durch die Menge der Besuchssequenzen, einen Keller für Knoten des Strukturbaumes und einen Steuer-Keller, der Elemente der Besuchssequenzen aufnimmt. Der Zustand des Automaten wird beschrieben durch den gerade besuchten Knoten k, dessen

Knoten-Typ p und das gerade bearbeitete Element der Besuchssequenz $F_p$, das wir im folgenden mit s=(p,i), $1 \leq i \leq |F_p|$, bezeichnen.

Jede Besuchssequenz $F_p$ zu einer Regel p ist eine Folge von Elementen $e \in \{a_j\} \cup \{b_{0,n}\} \cup \{b_{m,n}\}$ mit der Bedeutung (vergl. Abschnitt 4.2):

$a_j$ Die semantische Regel zur Brechnung des Attributes $a_j$ wird ausgewertet.

$b_{0,n}$ Der Vorgänger-Knoten wird zum n-ten mal besucht.

$b_{m,n}$ Der m-te Teilbaum wird zum n-ten mal besucht.

Im Anfangszustand des Automaten sind beide Keller leer, $k=k_0$ bezeichnet die Wurzel des zu bearbeitenden Baumes (Teilbaumes) und s=(p,i)=(Knotentyp(k),1). Abhängig von s werden folgende Übergänge durchgeführt:

U1: $s=a_j$ s:=(p,i+1)

U2: $s=b_{m,n}$ $(m\neq 0)$
(p,i+1) -> Steuer-Keller;
k -> Knoten-Keller;
k:= m-ter Nachfolge-Knoten von k;
s:=(q,j), mit q=Knotentyp(k) und j, so dass $(q,j-1)=b_{0,n}$.

U3: $s=b_{0,n}$
k <- Knoten-Keller;
s <- Steuer-Keller.

Der Automat hält an, wenn beim Übergang U3 die Keller leer sind. Aufgrund der Konstruktion der Besuchssequenzen ist das genau dann der Fall, wenn der Vorgänger-Knoten von $k_0$ besucht werden soll.

Der Automat kann für die verschiedenen in Abschnitt 4.2 beschriebenen Durchlauf-Strategien eingesetzt werden (falls die Besuchssequenzen nach den entsprechenden Vorschriften aus 4.2 konstruiert sind).

a) Der Strukturbaum wird von unten nach oben aufgebaut, und gleichzeitig - soweit möglich - semantisch analysiert. Dann wird bei der Generierung jedes neuen Teilbaumes der Automat auf diesen Wurzel $k_0$ angesetzt. Ist $k_0$ vom Typ p, so [illegible] ist, wenn

$s=(p,i)=b_{0,1}$. Man überlegt sich leicht, dass gilt:

Entweder ist $b_{0,1}$ das letzte Element in $F_p$, dann wird der Teilbaum nicht wieder besucht, oder $k_0$ wird zu einem späteren Zeitpunkt von dem dann existierenden Vorgänger-Knoten aus besucht und die Bearbeitung von $F_p$ wird mit dem auf $b_{0,1}$ folgenden Element fortgesetzt. Die semantische Analyse ist beendet, wenn $k_0$ die Wurzel des Strukturbaumes ist, und der Automat anhält.

b) Die semantische Analyse wird erst begonnen, wenn der Strukturbaum vollständig aufgebaut ist. Der Automat wird auf die Wurzel des Strukturbaumes angesetzt und verhält sich dann so wie im Fall a) nach der Generierung der Wurzel des Strukturbaumes.

c) Der Strukturbaum wird von der Wurzel zu den Endknoten hin aufgebaut und gleichzeitig - soweit möglich - semantisch analysiert. Im Unterschied zu b) läuft der Automat verzahnt mit dem Baumaufbau ab. Dazu ist der oben beschriebene Übergang U2 in zwei verschiedene Übergänge aufzuspalten:

| | | |
|---|---|---|
| U2a: | $s=b_{m,n}$ $(m\neq 0, n\neq 1)$ | wie U2 |
| U2b: | $s=b_{m,1}$ $(m\neq 0)$ | (p,i+1) -> Steuer-Keller;<br>k -> Knoten-Keller;<br>Fortsetzen des Strukturbaum-Aufbaus mit der Generierung des m-ten Nachfolge-Knotens von k;<br>weiter wie U2. |

## 4.4 Strukturbaum-Aufbau

Der Strukturbaum ist die Darstellung der abstrakten Struktur des übersetzten Programmes in Baumform. Er wird eindeutig durch die bei der Zerteilung angewendeten Regeln beschrieben. Sein Aufbau wird systematisch durch die Zerteilungsinformation gesteuert: Die Anwendung einer syntaktischen Regel p führt zur Bildung eines Baum-Knotens des Typs p. Die Zerteilungsstrategie bestimmt die Richtung, in der der Strukturbaum aufgebaut wird:

Bei einer "bottom-up"-Zerteilung wird der Baum von den Endknoten zur Wurzel hin aufgebaut; bei einer "top-down"-Zerteilung von der Wurzel zu den Endknoten hin. Wird die Zerteilung z.B. nach der LALR(1)-Methode (bottom-up) durchgeführt, so sind bei der Anwendung einer Regel $p = Y:X_1...X_n$ die Reduktionen für $X_1$ bis $X_n$ schon durchgeführt und entsprechende Teilbäume dafür generiert. Bezüge auf deren Wurzeln befinden sich in einem Keller. Sie werden zur Bildung des Knotens vom Typ p entnommen und durch einen Bezug auf diesen ersetzt. Ausserdem wird die semantische Analyse für diesen Teilbaum - soweit wie möglich - durchgeführt (siehe 4.3).

Unser Verfahren zur semantischen Analyse erfordert, dass der gesamte Strukturbaum - oder wenigstens der gerade bearbeitete Teilbaum - als Geflecht im Übersetzer dargestellt wird, da die Besuchsreihenfolge nur durch die semantischen Abhängigkeiten, nicht aber durch die Anordnung der Baumknoten bestimmt wird. Aus den Besuchssequenzen zu den Knotentypen können Strategien zur Reduzierung des Speicheraufwands für den Strukturbaum abgeleitet werden:

a) Sei p eine Regel $Y:X_1...X_n$.
Dann wird nach der "Abarbeitung" der Besuchssequenz $F_p$ für einen Knoten dieses Typs weder dieser Knoten erneut besucht noch auf Attribute aus den durch $X_1,...,X_n$ beschriebenen Unterbäumen zugegriffen. Sie können deshalb aus dem Strukturbaum entfernt werden. (Die Attribute, die den Code beschreiben, der für die $X_i$ erzeugt wird, sind schon zur Bildung von entsprechenden Attributen von Y oder dessen Vorgänger-Knoten benutzt worden.)

b) Sei p eine Regel $Y:X_1...X_n$ und $F_p = ...e_i...e_j...$ die zugehörige Besuchssequenz und zwischen $e_i$ und $e_j$ stehe weder ein Besuch eines Knotens $X_k$ noch eine semantische Regel, in der ein Attribut von $X_k$ verwendet wird. Dann kann der Teilbaum, dessen Wurzel $X_k$ bezeichnet, nach Ausführung von $e_i$ auf Hintergrundspeicher ausgelagert und vor der Ausführung von $e_j$ wieder in den Strukturbaum integriert werden. Dabei sollte jedoch die "Entfernung" von $e_i$ nach $e_j$ hinreichend gross sein. Als Mass eignet sich die Anzahl von Besuchen $b_{r,m}, r,k$, zwischen $e_i$ und $e_j$, wobei Besuche des Vorgänger-Knotens stärker zu gewichten sind als Besuche anderer [illegible] Nachfolgeknoten $X_r$.

Im folgenden geben wir das Schema an, nach dem zu jeder Regel p die Datenstruktur der Strukturbaum-Knoten vom Typ p bestimmt wird. Wir verwenden dabei folgende Notation:

| | |
|---|---|
| rep(s) | Darstellung von Objekten s der Beschreibungssprache im Übersetzer, |
| ref(t) | Bezug auf ein Datenobjekt des Übersetzers mit der Darstellung t, |
| $(t_1,...,t_n)$ | Verbund von Datenobjekten des Übersetzers mit den Darstellungen $t_i$, |
| $[t_1]\ t_2$ | Reihung von Objekten mit der Darstellung $t_2$, die mit Objekten $t_1$ indiziert wird. |

Sei p eine Regel $Y : X_1...X_i...X_n$, und die Attribute zu Y seien $A_Y = \{a_1,...,a_m\}$. Dann werden die Knoten vom Typ p durch die folgende Datenstruktur repräsentiert:

$$rep(p) = (Knotentyp(p), s_1,...,s_n, rep(a_1),...,rep(a_m)).$$

Die $s_i$ werden dargestellt durch

| | | |
|---|---|---|
| $ref(rep(X_i)) \setminus/ nil$ | | $[\ X_i\ ]$ |
| $ref(list(X_i))$ | falls $X_i$ | $(\ X_i\ )^*$ |
| $ref(list(X_r)) \setminus/ nil$ | in der Form | $[\ X_i\ ]$ |
| $ref(rep(X_i))$ | | $X_i$ |

in der syntaktischen Regel auftritt. Es gilt

$$rep(X_i) = \setminus/_\alpha\ rep(\alpha) \quad \text{mit } \alpha = X_i{:}u$$

$$list(X_i) = (ref(rep(X_i)), ref(list(X_i))) \setminus/ (ref(rep(X_i)), nil)$$

$rep(a_i)$ = Repräsentation des Attributes $a_i$ (siehe Abschnitt 4.5)

Falls die folgenden syntaktischen und semantischen Bedingungen erfüllt sind, kann auf die Darstellung von Knoten zu bestimmten Regeln verzichtet werden (d.h. rep(p)=()):

a) Gilt für alle Regeln $p = Y{:}u$, $u \in T^*$ und $A_Y = \emptyset$, dann ist rep(p)=().

b) Gilt für alle Regeln $p = Y:u$, $u \in V^*$ und $A_Y = \emptyset$, und ist $rep(X_i) = ()$ für alle $X_i$ mit $u = u1\ X_i\ u2$, dann ist auch $rep(p) = ()$.

c) Gilt für alle Regeln $p = Y:uXv$, $u,v \in T^*$ und $A_Y = A_X$, und gibt es zu jedem Attribut $a \in AD_Y \setminus / AI_X$ eine semantische Regel $Y.a:=X.a$ bzw. $X.a:=Y.a$, dann ist $rep(p) = rep(X)$.

Gilt zu einer Regel $q = Z:X_1 \ldots X_m$ $rep(X_i) = ()$, so entfällt in rep(q) die Komponente $s_i$. Gilt zu einer Regel p $rep(p) = ()$, so kann die Besuchssequenz $F_p$ aus der Übergangsfunktion des Automaten, der die semantische Analyse steuert, gestrichen werden.

## 4.5 Darstellung der Attribute im Übersetzer

In der Implementierung des Übersetzers werden Attribute durch Variable eines geeigneten Typs dargestellt, die den jeweiligen Knoten des Strukturbaumes zugeordnet sind (siehe 4.4).

Bei der Festlegung der Datenstrukturen, mit denen zusammengesetzte Attribut-Typen dargestellt werden, wird die Eigenschaft der Beschreibungssprache ausgenutzt, dass die Werte dieser Variablen nach ihrer Bestimmung nicht wieder verändert werden. Im folgenden beschreiben wir die Regeln, nach denen alle in einer Sprachdefinition möglichen Attribut-Typen auf Datentypen des Übersetzers abgebildet werden.

Die Darstellung einfacher Attribut-Typen ist klar und soll hier nicht weiter diskutiert werden. Sie kann den Beschreibungen am Ende dieses Abschnittes entnommen werden. Vereinigungs-Typen werden, wie schon in der Beschreibungssprache festgelegt, als zweigliedrige Verbunde implementiert.

Für die Implementierung der Verbund-Typen und der Mengen-Typen mit linearer Ordnung und/oder Schlüsselzugriff sind die folgenden Überlegungen entscheidend: Attribute vom Verbund-Typ werden auch im Übersetzer durch eine entsprechende zusammengesetzte Datenstruktur (in einem auch physikalisch zusammenhängenden Speicherbereich) dargestellt. Das konzeptionelle Enthaltensein solcher Datenstrukturen in

anderen (z.B. Verbunde als Komponenten von Verbunden oder Verbund-Attribute in der Datenstruktur des Strukturbaum-Knotens) kann entweder durch physikalisches Enthaltensein oder durch Bezüge implementiert werden.

Betrachtet man nur die Darstellung einzelner Verbunde, so erscheint die Implementierung durch Bezüge für die Ausnutzung des Speichers und für die Zugriffzeit aufwendiger. Dieser Nachteil wird jedoch im allgemeinen durch die besondere Verwendung von Verbunden in der Beschreibungssprache ausgeglichen: Verbund-Attribute werden oft aus Attributen anderer Baum-Knoten zusammengesetzt und möglicherweise über mehrere Knoten unverändert "transportiert". In diesen Fällen brauchen nur die Bezüge und nicht die gesamten Verbund-Objekte kopiert und mehrfach dargestellt zu werden. (Nur wenn der für einen Verbund benötigte Speicherumfang nicht wesentlich grösser ist als der eines Bezuges wird diese Darstellung zu schlechteren Ergebnissen führen.)

Für linear geordnete Mengen und Mengen mit Schlüssel-Zugriff ist der Umfang einer einzelnen Menge nicht vorherbestimmbar. Deshalb ist eine Listen-Darstellung zweckmässig. Auch muss die Inklusion von Mengen geeignet implementiert werden. Teillisten können nicht durch einen Bezug auf ihr erstes Element dargestellt werden, denn die Ordnung der Mengen ist über den Elementen und nicht den Teilmengen definiert. Eine Anwendung der Funktion TAIL auf eine geordnete Menge, die häufig zum Durchlaufen geordneter Mengen in rekursiven Funktionen verwendet wird, würde zur wiederholten, mehrfachen Generierung von Listenköpfen für Teillisten führen, die jedoch nur als Zwischenergebnisse von Ausdrücken benötigt werden. Wir implementieren deshalb die linear geordneten Mengen durch strikte lineare Listen, deren Köpfe auf das erste und letzte Listenelement verweisen. Solche Mengen werden durch die Konkatenation der Listen vereinigt. Dabei muss man in ungünstigen Fällen, die allerdings selten sind, eine Teilliste kopieren.

Mit der gleichen Technik werden Mengen implementiert, deren Elemente durch einen Schlüssel identifiziert werden. Die Abbildung der Schlüssel auf die Mengenelemente realisiert ein Index zu jedem Mengen-Typ. Diese Zugriffswege werden während der semantischen Analyse so verändert, dass zum Zeitpunkt der Identifikation eines Schlüssels in einer Menge genau die Elemente dieser Menge über den Index erreichbar sind. Die dafür notwendige Information enthalten

die semantischen Regeln. Ausserdem kann festgestellt werden, ob nur Teilmengen in den Index aufgenommen oder aus ihr entfernt werden müssen (typischer Fall bei den Mengen vereinbarter Grössen geschachtelter Abschnitte). Diese Implementierungsmethode ist besonders vorteilhaft, wenn in grösseren Teilen des Strukturbaumes nur in jeweils einer Menge eines Typs identifiziert wird.

Wir geben nun in formalisierter Notation an, wie die Attribut-Typen auf Daten-Typen des Übersetzers abgebildet werden. Die Menge der Attribut-Typen, die in einer Sprachdefinition auftreten, definiert einen gerichteten Graphen TG=(TK,TA). Jeder Knoten aus TK repräsentiert einen Attribut-Typ $t \in TK \subset AT$. (Typ-Bezeichner können als Markierung der Knoten aufgefasst werden.) Es existiert genau dann eine Kante $k=s(t_1,t_2) \in TA$ von $t_1$ nach $t_2$, wenn der Typ $t_1$ aus dem Typ $t_2$ konstruiert ist. Die Kanten werden nach dem jeweils angewendeten Typ-Konstruktor unterschiedlich benannt:

| | |
|---|---|
| $str(t_1,t_2)$ | Der Verbundtyp $t_1$ enthält einen Komponententyp $t_2$. |
| $un(t_1,t_2)$ | Der Vereinigungstyp $t_1$ umfasst den Typ $t_2$. |
| $set(t_1,t_2)$ | $t_2$ ist der Elementtyp des Mengentyps $t_1$. |
| $setl(t_1,t_2)$ | $t_2$ ist der Elementtyp des Mengentyps $t_1$, für den lineare Ordnung definiert ist. |
| $setk(t_1,t_2)$ | $t_2$ ist der Elementtyp des Mengentyps $t_1$, für den Schlüsselzugriff definiert ist. |
| $setlk(t_1,t_2)$ | $t_2$ ist der Elementtyp des Mengentyps $t_1$, für den lineare Ordnung und Schlüsselzugriff definiert sind. |
| $aus(t_1,t_2)$ | $t_1$ ist ein Ausschnitt von $t_2$. |

Der Graph muss den folgenden Bedingungen genügen:

a) Es gibt keine Kante $s(t_1,t_2)$ mit $t_1 \in$ {INT,BOOL,IDENT} $\vee$ {skalare Typen}.

b) Für alle Kanten $aus(t_1,t_2)$ gilt $t_1 \in$ {INT, skalare Typen}

c) Für alle Kanten $set(t_1,t_2)$ gilt $t_2 \in$ {skalare Typen} oder es gibt eine Kante $aus(t_2,t_3)$.

d) Für alle Kanten $setk(t_1,t_2)$ und $setlk(t_1,t_2)$ gilt: Von $t_2$ gehen nur Kanten $str(t_2,t_3)$ aus.

e) Aus $s(t_1,t_2)$ und $s'(t_1,t_3)$ und $s \in$ {set,setl,setk,setlk, aus} folgt $s=s'$ und $t_2=t_3$.

f) Aus $str(t_1,t_2)$ und $s(t_1,t_3)$ folgt $s=str$.

g) Aus $un(t_1,t_2)$ und $s(t_1,t_3)$ folgt $s=un$.

h) Der Graph enthält nur zulässige Zyklen. Ein Zyklus

$Z=s1(t_1,t_2),s2(t_2,t_3),\ldots,sn(t_n,t_1)$

ist zulässig, wenn es mindestens eine Kante $un(t_i,t)$ mit $1 \leq i \leq n$ gibt und $t$ entweder auf einem zulässigen Zyklus $Z' \neq Z$ oder auf keinem Zyklus liegt.

Jedem Knoten $t$ eines solchen Graphen wird durch die Abbildung $rep(t)$ ein Datentyp des Übersetzers zugeordnet:

| Charakterisierung von t | rep(t) |
|---|---|
| t=INT | Menge der ganzen Zahlen |
| t=BOOL | {TRUE,FALSE} |
| t=IDENT | Teilmenge der natürlichen Zahlen |
| t=STRING | Teilmenge der natürlichen Zahlen |
| t=skalarer Typ | $\{e \in N \mid 0 \leq e \leq n-1, \lvert t \rvert = n\}$ |
| $aus(t,t_1)$ | {e∈rep($t_1$) \| untere Grenze von $t_1 \leq e \leq$ obere Grenze von $t_1$ } |
| $str(t,t_i)$ | $ref(rep(t_1),\ldots,rep(t_n))$, $1 \leq i \leq n$ |
| $un(t_1,t_i)$ | $(\{i \mid 1 \leq i \leq n\},rep(t_i))$ |
| $set(t,t_1)$ | $[rep(t_1)]$ rep(BOOL) |

setl(t,$t_1$) (listenanfang,listenende) mit listenanfang=listenende=listenbezug= ref(rep($t_1$),listenbezug)

setk(t,$t_1$) (listenanfang,listenende) mit listenanfang=listenende=listenbezug =indexbezug =ref(rep($t_1$),listenbezug,indexbezug). Ausserdem wird zu t ein Index vom Typ [Schlüsseltyp von t] indexbezug implementiert.

setlk(t,$t_1$) wie setk(t,$t_1$)

(Bezeichner und Zeichenreihen werden durch die Symbolentschlüsselung eineindeutig auf Teilmengen der natürlichen Zahlen abgebildet.)

## 4.6 Attribute im erweiterten Kontext

Im Abschnitt 3.2.6 wurde gezeigt, wie semantische Regeln, die auf Attribute nicht benachbarter Knoten Bezug nehmen (Konstituenten-Attribut und äusseres Attribut), durch implizite Einführung neuer Attribute auf Regeln der strikten Beschreibungsform (Abschnitte 3.2.4, 3.2.5) zurückgeführt werden können. Für die Implementierung ist dieses Verfahren jedoch recht aufwendig, denn die Darstellung der Strukturbaum-Knoten würde durch die implizit eingeführten Attribute erheblich ausgeweitet. Ausserdem wäre eine Reihe zusätzlicher semantischer Aktionen zum "Transport" dieser Attribute von der (den) Definitionsstelle(n) zu der (den) Anwendungsstelle(n) erforderlich.

Wir zeigen im folgenden, wie solche Funktionen über Attributen nicht benachbarter Knoten mit Hilfe von Datenobjekten, die global zum Strukturbaum sind, günstiger implementiert werden können. Dabei wird folgendes Prinzip angewendet: An der (den) Definitionsstelle(n) der Attribute werden die ermittelten Attribut-Werte in dem globalen Datenobjekt gespeichert und an der (den) Anwendungsstelle(n) entnommen. Die Gültigkeit der Besuchssequenzen wird durch dieses Verfahren nicht berührt.

### 4.6.1 Äusseres Attribut

Wird in einer semantischen Regel zu einer syntaktischen Regel p = Z:u ein äusseres Attribut

INCLUDING Y.a

angewendet, so ist jede Ableitungssequenz S => vZw zerlegbar in die Ableitungssequenzen

S => v1 Y w1 => v1 v2 Z w2 w1
mit v = v1 v2 und w = w1 w2.

Zu jedem Knoten der Klasse Z gibt es also im Strukturbaum einen Oberknoten der Klasse Y (siehe auch Abschnitt 3.2.6. 2). Abhängig von den angegebenen Voraussetzungen wird eine der folgenden Implementierungsmethoden angewendet:

a) Ist Y nicht rekursiv definiert, so wird eine zum Strukturbaum globale Variable vom Typ des Attributes Y.a eingerichtet, in die nach der Berechnung von Y.a der ermittelte Wert gespeichert wird. Bei der Auswertung des äusseren Attributes wird der Wert dieser Variablen benutzt, ohne erneut auf Y.a zuzugreifen. (Erfolgen zwischen der Bestimmung von Y.a und der Auswertung eines solchen äusseren Attributes Besuche des Vorgänger-Knotens von Y, so muss nach jedem solchen Besuch der globalen Variablen erneut der Wert von Y.a zugewiesen werden.)

b) Ist Y rekursiv definiert, so wird statt einer globalen Variablen ein globaler Keller für Attribut-Werte des entsprechenden Typs eingerichtet. Bei der Bestimmung von Y.a wird der ermittelte Wert gekellert, vor jedem darauffolgenden Besuch des Vorgänger-Knotens von Y entkellert und danach erneut gekellert, falls noch Auswertungen des äusseren Attributes folgen. Zur Auswertung des äusseren Attributes wird das oberste Kellerelement verwendet.

c) Ist das Attribut Y.a vom Typ einer Menge mit Schlüsselzugriff und wird das äussere Attribut nur in identifizierenden Zugriffen oder wiederum zur Bildung eines Attributes Y.a verwendet, so wird zusätzlich zum Index kein weiteres globales Objekt benötigt. Analog zu a)

und b) werden die Elemente von Y.a in den Index aufgenommen und wieder daraus entfernt. (Falls im Unterbaum zu einem Knoten von der Klasse Y in mehreren Mengen vom gleichen Typ identifizierend zugegriffen wird, ist dieses Verfahren nicht anwendbar. Die Behandlung des äusseren Attributes wird dann wie in a) bzw. b) implementiert.)

### 4.6.2 Konstituenten-Attribut

Wird in einer semantischen Regel zu einer syntaktischen Regel $p = Y:X_1 \ldots X_n$ ein Konstituenten-Attribut der Form

(1) $X_i$ CONSTITUENT Z.a bzw.
(2) CONSTITUENTS Z.a oder $X_i$ CONSTITUENTS Z.a

angewendet, so gibt es nur Ableitungssequenzen der Form

(1) $X_i \Rightarrow u\ Z\ v \Rightarrow s$ bzw.
(2) $X_i \Rightarrow u_1\ Z_1\ u_2\ Z_2 \ldots Z_n\ v$ oder
$Y \Rightarrow u_1\ Z_1\ u_2\ Z_2 \ldots Z_n\ v$,

für die dieBedingungen aus 3.2.6.2 gelten. Im Fall (2) beschreibt das Konstituenten-Attribut die Menge der Attribute $Z_j.a$ der Nachfolge-Knoten von $X_i$ bzw. von Y. Abhängig von den angegebenen Voraussetzungen wird eine der folgenden Implementierungsmethoden angewendet:

a) Falls kein u oder v der möglichen Ableitungssequenzen ein Y enthält und Y nicht aus Z ableitbar ist, wird eine zum Strukturbaum globale Variable vom Typ des Attributes Z.a im Fall (1) bzw. von einem entsprechenden Mengen-Typ im Fall (2) eingerichtet, die den Wert von Z.a aufnimmt. Bei der Auswertung des Konstituenten-Attributes wird der Wert dieser Variablen verwendet, ohne erneut auf Z.a zuzugreifen.

b) Ist Y rekursiv definiert, und liegt der Fall (1) des Konstituenten-Attributes vor, so wird anstelle der Variablen ein Keller für Elemente des entsprechenden Typs angelegt.

c) Ist Y rekursiv definiert und liegt der Fall (2) des Konstituenten-Attributes vor, und wird der Knoten vom Typ p vor der Bestimmung der Attribute Z.a mindestens einmal besucht, so wird ein Keller für Mengen von Attributen vom Typ von Z.a angelegt. Beim letzten Besuch des Knotens vom Typ p vor der Bestimmung von Z.a wird jeweils eine leere Menge auf den Keller gebracht. Bei der Bestimmung von Z.a wird der Wert in die Menge des obersten Kellerelementes eingefügt. Beim nächsten Besuch des Knotens vom Typ p enthält das oberste Kellerelement den Wert des Konstituenten-Attributes und wird entkellert.

d) Ist Y rekursiv definiert und liegt der Fall (2) des Konstituenten-Attributes vor, und wird der Knoten vom Typ p vor der Bestimmung von Z.a nicht besucht, so wird ein Keller für Paare (x,n) angelegt. x ist jeweils der Wert von Z.a und n gibt die Ordnung des Knotens Z an (bezüglich der Post-Ordnung des Strukturbaumes). Beim ersten Besuch eines Knotens vom Typ p bilden die obersten Kellerelemente, deren Ordnungsangabe nicht kleiner als die dieses Knotens ist den Wert des Konstituenten-Attributes, und werden entkellert. Falls ein Konstituenten-Attribut der Form (2) zur Bildung einer linear geordneten Menge angewendet wird, ist auch in den Fällen a) und c) zusätzlich eine Angabe der Ordnung des Knotens im Strukturbaum vorzusehen, die zur Bestimmung der Anordnung der Mengenelemente herangezogen wird.

# 5 Definition einer abstrakten Maschine

In diesem Kapitel definieren wir die abstrakte Maschine AMICO (Abstract Machine for Implementation of COmpilers). Sie beschreibt die Schnittstelle für die Implementierung unseres Übersetzer-erzeugenden Systems und definiert pragmatisch die Begriffe, die der Formulierung der dynamischen Semantik in den vom System verarbeiteten Sprachdefinitionen zugrunde liegen.

Wir stellen zunächst die Ziele der Entwicklung von AMICO dar und geben einen Überblick über die Struktur der Maschine. In den Abschnitten 5.2 bis 5.8 beschreiben wir die Datentypen und Operationen und geben ihre sprachliche Formulierung an. Abschnitt 5.9 enthält eine Liste der AMICO-Operationen.

## 5.1 Konzeption

Als maschinenunabhängige Schnittstelle des Übersetzer-erzeugenden Systems wird AMICO für folgende Bereiche verwendet:

a) Die dynamische Semantik von Programmiersprachen wird mit den Eigenschaften von AMICO definiert.

b) Die vom Übersetzer-erzeugenden System erzeugten Übersetzer produzieren AMICO-Programme.

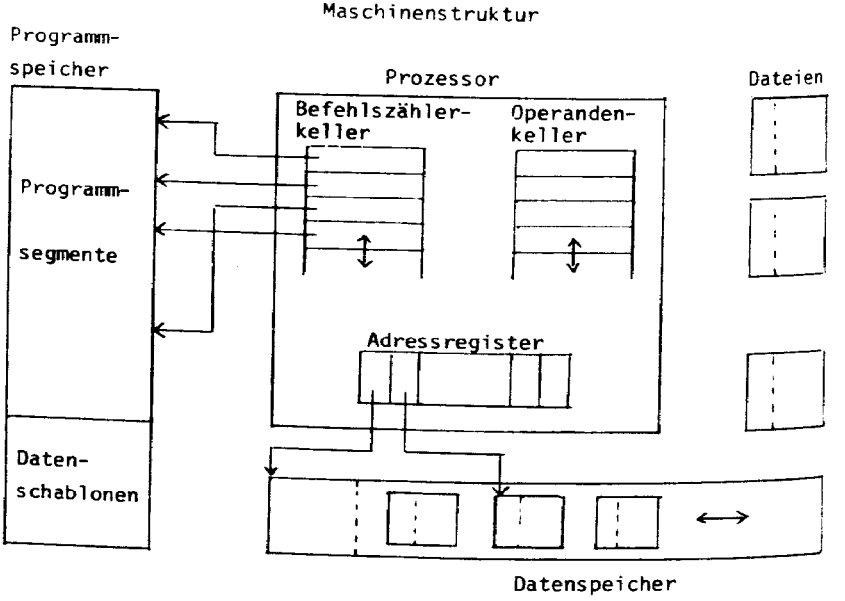
c) Die erzeugten Übersetzer sind AMICO-Programme.

d) Das Übersetzer-erzeugende System ist ein AMICO-Programm.

Die Verwendungsart a) legt die Definition der abstrakten Maschine fest, denn durch sie wird die Klasse der Sprachen, für die das Übersetzer-erzeugende System anwendbar ist, und die Qualität der Sprachdefinitionen bestimmt. Die Anforderungen an die abstrakte Maschine, die sich aus den Anwendungen a) und b) ergeben, stimmen weitgehend überein. Nimmt man einige weitere Eigenschaften hinzu, so lässt sich AMICO auch für die Aufgaben c) und d) verwenden.

Der Aufwand für die Übertragung des Übersetzer-erzeugenden Systems und aller von ihm erzeugten Übersetzer ist begrenzt auf den Aufwand zur Implementierung von AMICO auf einer Rechenanlage. Wird AMICO auf verschiedenen Rechenanlagen implementiert, so brauchen das Übersetzer-erzeugende System, die erzeugten Übersetzer und die von ihnen generierten Programme nicht auf derselben Anlage zu laufen. Komplexere Funktionen, die für bestimmte Rechenanlagen elementar sind, sind auch in AMICO nicht in einfachere zerlegt (z.B. Kellerorganisation für Umgebungen).

Die dynamische Semantik allgemein anwendbarer, höherer Programmiersprachen, die zur strukturierten Programmierung geeignet sind, kann einfach auf die Eigenschaften von AMICO abgebildet werden. Charakteristische Merkmale dieser Sprachklasse sind das Umgebungskonzept, das Blockstrukturen und Prozeduren, aber auch allgemeinere Programmoduln einschliesst, und Konzepte zur Strukturierung von Daten und Operationen. Eigenschaften von Sprachen mit speziellen Anwendungsgebieten (z.B. Textmanipulation, Listenverarbeitung, extensive E/A-Verarbeitung, Betriebs-Steuerung) können nicht oder nur mit grösserem Aufwand auf die abstrakte Maschine abgebildet werden. Die Eignung für solche Sprachen kann durch Erweiterung der abstrakten Maschine um entsprechende Funktionen erzielt werden.




Die abstrakte Maschine AMICO besteht aus folgenden Komponenten:

- Programmspeicher

  Er enthält Datenschablonen und Programmsegmente.

- Datenspeicher

  Seine Struktur wird durch eine Schablone beschrieben. Sie definiert einen Bereich mit statischen Komponenten und einen variablen Bereich, in dem nach dem Kellerprinzip dynamisch Komponenten eingefügt oder beseitigt werden können (z.B. Schachtelverbunde, mit denen das Umgebungsmodell realisiert wird).

- Dateien

Sie werden als Datenstrukturen ausserhalb des Adressraumes des Programmes aufgefasst.

- Prozessor

Er besteht aus einem Befehlszählerkeller, dessen Elemente Bezüge in den Programmspeicher sind, einem Operandenkeller, der die Operanden und Ergebnisse von Operationen aufnimmt, und Adressregistern, die Bezüge auf Datenobjekte aufnehmen.

Die Typen der Datenobjekte eines AMICO-Programmes werden durch Schablonen beschrieben. Sie sind als Vorschriften zur Interpretation von Speicherbereichen zu verstehen, mit denen Zugriffsfunktionen spezifiziert werden. Annahmen über den Speicherumfang von Objekten und die Codierung ihrer Inhalte werden nicht gemacht. Eine Uminterpretation von Speicherinhalten ist deshalb nicht sinnvoll (siehe Abschnitte 5.4, 5.6). Auf der Ebene der abstrakten Maschine besteht ebenso wie auf den meisten realen Maschinen keine Sicherheit gegen Fehlinterpretationen von Daten. Sie muss bei der Erzeugung der AMICO-Programme im gewünschten Umfang hergestellt werden.

Sowohl Datentypen von Programmiersprachen wie auch die Speicherorganisation der abstrakten Maschine selbst können mit den Begriffen der zusammengesetzten Datenobjekte beschrieben werden. Sie beruhen auf dem Konzept allgemeiner Verbunde, die neben statisch definierten Komponenten eine Speicherzone umfassen können, in der nach dem Kellerprinzip dynamisch Datenobjekte einfügbar sind. Der gesamte Adressraum der abstrakten Maschine wird durch einen solchen Verbund beschrieben. Seine Struktur kann auf die Anforderungen der jeweiligen Programmiersprache angepasst werden. Die dynamischen Veränderungen der Speicherstruktur sind so definiert, dass der Speicher der abstrakten Maschine durch eine Kellerstruktur in einem linearen oder stückweise linearen Speicherbereich einer realen Maschine implementiert werden kann (siehe Abschnitt 5.5).

Die Implementierung des Umgebungskonzeptes höherer Programmiersprachen durch einen Schachtelkeller wird als Spezialisierung dieser allgemeinen Speicherstruktur aufgefasst: Die variablen Komponenten des äussersten Verbundes sind die Schachtelverbunde. Für die Generierung und Beseitigung von Schachtelverbunden und für Zugriffe auf deren Komponenten

werden zusätzliche Funktionen eingeführt, deren Bedeutung durch Operationen mit Verbunden erklärt ist (siehe Abschnitt 5.7).

Das Dateikonzept von AMICO wird ebenfalls auf den Verbundbegriff zurückgeführt: Dateien werden als Verbunde aufgefasst, deren Komponenten ausserhalb des Adressraumes des Programmes liegen. Da die Implementierung von Dateien auf verschiedenen Rechenanlagen sehr stark differiert, ist dieses Modell auf einige Standard-Eigenschaften und -Operationen beschränkt, die den Benutzungsanforderungen der abstrakten Maschine genügen und auf den gebräuchlichen Rechenanlagen realisierbar sind (siehe Abschnitt 5.8).

Der ausführbare Teil eines AMICO-Programmes ist in Abschnitte (Programmsegmente) gegliedert, die den strukturellen Einheiten von Programmen höherer Programmiersprachen entsprechen. Ihre Ausführung wird durch die elementaren Funktionen zur Ablaufsteuerung strukturierter Programme gesteuert (Aufruf, Ausgang, Schleife und Verzweigung).

Die Programmstruktur, die Funktionen zur Ablaufsteuerung und die Beschreibung ihrer Wirkung durch einen Befehlszähler-Keller (Abschnitt 5.3) sind mit den Konzepten vergleichbar, die in Rechenanlagen vom Typ Burroughs 1700 realisiert sind. (Wilner stellt in [Wi74] die bei der Entwicklung dieser Maschine verfolgten Ziele dar.) Im Gegensatz zu dem dort angewendeten Prinzip der strikten Baum-Strukturierung von Programmen bilden wir jedoch einfache Programmverzweigungen nicht auf den Aufruf-Mechanismus ab, denn es kann nicht ausgeschlossen werden, dass Sprachen, die in AMICO übersetzt werden, solche Konzepte enthalten. Eine solche vom Übersetzer durchzuführende Abbildung müsste im allgemeinen bei der Implementierung von AMICO auf gebräuchlichen Rechenanlagen wieder rückgängig gemacht werden.

Die Operationssequenzen der Programmsegmente werden in Postfix-Notation angegeben, die leicht aus der Programmstruktur des übersetzten Programmes erzeugbar ist. Die Typen der Operanden und der Ergebnisse jedes Operators werden vollständig spezifiziert, so dass die Verwaltung von Zwischenergebnissen für Kellermaschinen und für Registermaschinen einfach implementierbar ist (siehe Abschnitte 5.2, 5.4).

Wir setzen voraus, dass jedes AMICO-Programm vollständig ist. Für das Zusammenbinden getrennt übersetzter Programmteile müssen die von unserem System erzeugten Übersetzer durch entsprechende Funktionen ergänzt werden. Moduln anderer Programmiersprachen oder Funktionen eines Laufzeitsystems können bei der Abbildung der AMICO-Programme auf Maschinenprogramme eingebunden werden.

## 5.2 Programmstruktur

Ein AMICO-Programm besteht aus einer beliebig geordneten Menge benannter Programmabschnitte (im folgenden Segmente genannt) und einer beliebig geordneten Menge benannter Schablonen für zusammengesetzte Datenobjekte (siehe Abschnitt 5.4.2).

```
Programm: (Schablonendefinition || <;> ) <||>
          (Segmentdefinition || <;> ) <#> .
Segmentdefinition:
          SEGMENT Typangaben Segmentbezeichner <;>
          (Operation || <;> )
          END Segmentbezeichner .
Operation:
          Operator Typangaben (Spezifikation)* .
Typangaben:
          <(> (Operandentyp)* </> (Ergebnistyp)* <)> .
Operandentypen:
          einfacher_Typ | Schablonenbezeichner .
Ergebnistyp:
          einfacher_Typ | Schablonenbezeichner .
Spezifikation:
          einfacher_Typ | Bezeichner | Konstante .
```

Ein Segment definiert eine Funktion über den angegebenen Typen, die Seiteneffekte verursachen kann (z.B. Veränderung des Inhalts von Objekten). Die Wirkung des Segments wird beschrieben durch eine Folge von Aufrufen von AMICO-Grundfunktionen, die selbst wiederum Aufrufe von Segmenten sein können. Die Operationenfolgen sind in einer erweiterten Postfix-Form notiert: Die Operanden eines Operators werden durch die zuletzt gebildeten Ergebnisse vorausgehender Operatoren bestimmt. Ein Operator kann zugleich mehrere Ergebnisse liefern. Die Typen der Operanden und der Ergeb-

nisse werden zu jedem Operator explizit angegeben. Die Verwaltung der Zwischenergebnisse ist auf Kellermaschinen ohne weitere Transformation implementierbar. Im Falle von Registermaschinen können die Zwischenergebnisse den Registern leicht mit Hilfe der vollständigen Typangaben in den Operationen zugeordnet werden.

Die Ausführung eines Segments wird durch einen Aufruf (siehe Abschnitt 5.3) veranlasst, dessen Operanden vom Typ der angegebenen Segment-Operanden sein müssen. Sie bilden die Operanden des (der) ersten Operator(en) der Operationenfolge. Die Ausführung wird beendet durch einen Ausgang (siehe Abschnitt 5.3), dessen Operanden die Ergebnisse des Segmentes sind. Die gebräuchlichen Verfahren zur Übergabe von Parametern sind auf dieses Konzept abbildbar, das wiederum unter Anwendung unterschiedlicher Techniken auf verschiedenen Rechenanlagen implementiert werden kann.

Die Ausführung eines AMICO-Programmes beginnt mit der Ausführung des mit 'main' bezeichneten Segmentes (siehe Abschnitt 5.3). Zugleich wird der Adressraum des Programmes als ein Verbund vom Typ 'main' erzeugt (siehe Abschnitt 5.5). Das Programm muss jeweils genau ein Segment und eine Schablone mit diesem Bezeichner enthalten.

Abschnitt 5.9 enthält eine vollständige Liste der Operatoren und ihrer möglichen Operanden- und Ergebnistypen und der weiteren Spezifikationen. Einige Operatoren beschreiben eine Menge von Funktionen, deren Elemente durch Einsetzen wählbarer Typangaben oder Spezifikationen gewonnen werden.

## 5.3 Ablaufsteuerung

Die AMICO-Operationen, mit denen der Programmablauf gesteuert wird, entsprechen den Grundfunktionen zur Ablaufsteuerung in Sprachen zur strukturierten Programmierung: Alternative, Fallunterscheidung, Schleife, Aufruf und Ausgang. Nur die einfachen Verzweigungen (Alternative und Fallunterscheidung) sind auf elementare Funktionen (bedingter Sprung und Verteiler-Sprung) zurückgeführt, da sie auf den gebräuchlichen Rechenanlagen weitgehend einheitlich implementiert sind.

Wir beschreiben im folgenden die Wirkung der Funktionen mit Hilfe eines Kellers von Befehlszählern. Die Elemente des Kellers identifizieren Operatoren in Segmenten. Zu Beginn der Programmausführung zeigt das einzige Kellerelement auf den ersten Operator des Segmentes 'main'. Es wird jeweils die Operation zu dem Operator, auf den das oberste Kellerelement zeigt, ausgeführt. Mit dem Beginn der Ausführung der Operation zeigt das oberste Kellerelement auf den in der Aufschreibung nächsten Operator. Der letzte Operator der Operationenfolge hat in diesem Sinn keinen Nachfolger. Er muss ein Steuerungsoperator sein, der mindestens das oberste Kellerelement neu bestimmt. Die Programmausführung ist beendet, wenn der Befehlszählerkeller leer ist.

Die Aufruf-Operatoren CALL und DYNCALL ändern das oberste Element, so dass es auf den nächsten Operator nach dem Aufruf zeigt, und bringen ein neues Element auf den Keller, das auf den ersten Operator des als Spezifikation bzw. als Operand angegebenen Segmentes zeigt. Die Operanden- und Ergebnistypen des Aufrufs müssen mit denen des Segments übereinstimmen.

Zum Rücksprung aus Segmenten dienen die Operatoren RETURN, MULTIPLERETURN, RETURNMAIN. Es werden das oberste Element des Befehlszählerkellers (RETURN) bzw. die in der Spezifikation angegeben Anzahl von Elementen (MULTIPLERETURN) bzw. alle Elemente bis auf das letzte (RETURNMAIN) entfernt. Die Operandentypen müssen mit den Ergebnistypen des (letzten) verlassenen Segments übereinstimmen. Im Falle von MULTIPLERETURN ist der Bezeichner des Segments explizit anzugeben.

Die Operationen EXITCALL und DYNEXITCALL verknüpfen die Wirkung von Rücksprung und Aufruf zur Funktion eines allgemeinen Ausgangs. Das oberste Element des Befehlszählerkellers zeigt nach der Operation auf den ersten Operator des als Spezifikation bzw. als Operand angegebenen Segments.

Schleifen werden durch den impliziten wiederholten Aufruf eines Segments formuliert. Die Ausführung der Schleifenoperatoren setzt das oberste Element des Befehlszählerkellers auf den Schleifenoperator zurück und bringt ein neues Element, das auf den ersten Operator des angegebenen Segments zeigt, auf den Keller, falls der Operand den Wert TRUE liefert. Andernfalls bleibt der Befehlszählerkeller unverändert. (Das oberste Kellerelement zeigt dann auf den

Operator, der auf den Schleifenoperator folgt.) Das angegebene Segment muss ein Ergebnis von Typ BOOL liefern. Das zum Operator LOOP (LOOPCOUNT) angegebene Segment hat keine Operanden (einen Operanden vom Typ COUNT). Auf diese Operatoren können alle Schleifenkonstruktionen höherer Programmiersprachen (mit und ohne Zählung) einfach zurückgeführt werden.

Verzweigungen in der Reihenfolge der Ausführung der Operatoren eines Segmentes werden durch Marken und Sprünge beschrieben. (Sprünge, bei denen ein Segment verlassen wird, sind nicht definiert. Sie müssen auf Ausgänge zurückgeführt werden.) Der Operator DEFLABEL ordnet dem angegebenen Bezeichner den Inhalt des obersten Elementes des Befehlszählerkellers zu. Es ist durch geeignete Operationen sicherzustellen, dass die Typen der Operanden und ihre Reihenfolge an allen Stellen, von denen aus diese Markendefinition erreichbar ist, gleich sind. Die Sprungoperatoren JUMP (JUMPFALSE) ersetzen den Inhalt des obersten Kellerelements durch den Wert, der dem angegebenen Bezeichner zugeordnet ist (falls der letzte Operand den Wert FALSE liefert).

Der Verteiler-Operator CASE ersetzt den Inhalt des obersten Kellerelements durch den Wert, der einem der angegebenen Bezeichner zugeordnet ist. Sind n Bezeichner angegeben und ist der Wert des Operanden (vom Typ COUNT) die Zahl i, so wird das oberste Kellerelement durch den Wert, der dem i+1-ten Bezeichner zugeordnet ist, ersetzt, falls $0<i<n-1$. Andernfalls bleibt der Befehlszählerkeller unverändert.

Ist ein Operator in dem gegebenen Kontext nicht ausführbar, so wird stattdessen ein Segment mit vordefinierter Bezeichnung aufgerufen. Dies sind z.B. die Segmente:

a) ARITHFAULT, falls arithmetische Operationen in dem jeweiligen Wertebereich nicht ausführbar sind,

b) INVALIDADDR, falls in einer Zugriffsoperation ein Wert nicht als Stelle innerhalb des Adressraumes des AMICO-Programms interpretiert werden kann, oder

c) STOREOVERFLOW, falls der durch einen Generator zuzuordnende Speicher nicht zur Verfügung steht.

Falls in dem Programm zu diesen Bezeichnern keine Segmente definiert sind, werden sie implizit eingefügt. Sie

bewirken dann die Ausgabe einer Fehlermeldung und einen Rücksprung in das Segment 'main'.

## 5.4 Datentypen

Die Datentypen von AMICO schaffen eine Basis zur Definition von Objekten und Zugriffswegen, die unabhängig ist von Codierungen und Umfang der Objekte der realen Maschine und deren Zugriffsmechanismen. Die Grundtypen von AMICO stimmen weitgehend mit den in Programmiersprachen der betrachteten Klasse und den auf den meisten Rechenanlagen vorhandenen Typen überein oder können einfach aufeinander abgebildet werden. Mit den in Abschnitt 5.4.2 definierten Strukturierungskonzepten werden daraus Schablonen gebildet, die angewendet auf einen Speicherbereich die Zugriffswege für die Teilobjekte definieren.

### 5.4.1 Einfache Datentypen

Die einfachen Datentypen INT, REAL, LONGINT, LONGREAL, BOOL und die auf ihnen definierten Operationen (siehe Abschnitt 5.9) entsprechen weitgehend den in Programmiersprachen angewendeten und auf realen Maschinen implementierten Typen und Operationen. Hinzu kommen zwei Typen für ganzzahlige Objekte mit kleinerem Zahlbereich, die im allgemeinen nur bei der Übersetzung von Programmen Bedeutung haben:

Der Typ SHORT umfasst nur nicht negative ganzzahlige Werte. Er wird z.B. zur Codierung von skalaren Listen oder Kennungen verwendet. Er kann je nach Rechenanlage auf ein Byte oder ein entsprechendes Teilwort abgebildet werden. Es sind nur Vergleichsoperationen, die Addition und Subtraktion von 1 mit Objekten dieses Typs definiert.

Der Typ COUNT dient speziell für zählungen und Indizierungen. Es sind nur Addition, Subtraktion und Multiplikation definiert. Auf einigen Rechenanlagen können dafür Indexregister ausgenutzt werden, deren Wertebereich häufig kleiner ist als der für den Typ INT.

Sei A<B eine Relation über den Typen A und B, die bedeutet, dass jedes Objekt des Typs A ohne Genauigkeitsverlust als ein Objekt vom Typ B dargestellt werden kann, dann gilt

SHORT < COUNT < INT < LONGINT und
REAL < LONGREAL.

Unter Einhaltung dieser Bedingung können Zahlwert-Typen, für die es auf einer Rechenanlage kein Äquivalent gibt, durch andere implementiert werden. Die durch die Implementierung festgelegten Wertebereiche der einzelnen Typen sind als Implementierungskonstante in AMICO-Programmen zugänglich.

Für die Verarbeitung von Zeichen sind der Typ CHAR und Vergleichsoperationen definiert. Alle Objekte dieses Typs können eindeutig auf Objekte des Typs SHORT abgebildet werden. Es wird vorausgesetzt, dass die Codierungen der Ziffern lückenlos von 0 bis 9 aufsteigen und dass die Buchstaben in alphabetischer Reihenfolge aufsteigend, möglicherweise lückenhaft, codiert sind.

Bezüge auf Datenobjekte bzw. Programmstellen werden durch die einfachen Typen ADDR und SEG beschrieben. Die Operationen mit diesen Objekten werden in den Abschnitten 5.3 und 5.6 erklärt.

Mit dem Operator CONST wird ein in Standardscheibweise angegebener Wert als Operand beschafft. Die Standardschreibweisen für Werte einfacher Typen (und der Typen SET n STRING n siehe Abschnitt 5.4.2.1) sind wie folgt definiert:

```
Konstante: ganze_Zahl | reelle_Zahl | logischer_Wert |
           Zeichen_Wert | Zeichenreihe | Mengen_Wert |
           Implementierungskonstante .
ganze_Zahl:
           ( Ziffer )* .
reelle_Zahl:
           [ Ziffer ]* <.> ( Ziffer )* [ Exponent ] |
           [ Ziffer ]* Exponent .
Exponent:  <E> [ <+> ] ( Ziffer )* |
           <E> <-> ( Ziffer )* .
logischer_Wert:
           TRUE | FALSE .
Zeichen_Wert:
```

```
            <"> Zeichen <"> .
Zeichenreihe:
            <"> [ Zeichen ]* <"> .
Zeichen:    <alle Zeichen des Zeichensatzes ausser "> |
            <""> .
Mengen_Wert:
            ( Dualziffer )* .
Ziffer:     <0> | <1> | <2> | <3> | <4> |
            <5> | <6> | <7> | <8> | <9> .
Dualziffer:
            <0> | <1> .
Implementierungskonstante:
            MAXSHORT | MINCOUNT | MAXCOUNT | MININT |
            MAXINT | MINLONGINT | MAXLONGINT | MINREAL |
            MAXREAL | MINLONGREAL | MAXLONGREAL |
            CARDINALSET | CARDINALCHAR .
```

Die Umcodierung des Wertes eines Operanden in einen Wert eines anderen Typs wird durch den Operator CONV (und die Operatoren GET, PUTINT, PUTREAL für die Konvertierung von Zeichenreihen in Zahlwerte und umgekehrt) bewirkt.

### 5.4.2 Zusammengesetzte Datentypen

Zusammengesetzte Datentypen werden durch Schablonen beschrieben, die den Umfang solcher Objekte und die Zugriffswege auf ihre Teilobjekte definieren. Schablonen für zusammengesetzte Objekte werden nach Konstruktionsprinzipien für Reihen, Reihungen und Verbunde aus einfachen Datentypen und Schablonen gebildet.

Wir unterscheiden Schablonen, die zusammengesetzte Objekte vollständig statisch definieren, und solche, die Objekte definieren, deren Struktur zur Ausführungszeit bestimmt und möglicherweise verändert wird. Schablonendefinitionen haben folgende Form:

```
Schablonendefinition:
    TYPE (Schablonenbezeichner <=> Schablone || <,> ) .
Schablone:
    statische_Schablone | variable_Schablone .
statische_Schablone:
    Reihe | statischer_Verbund .
```

```
Variable_Schablone:
    Reihung | variabler_Verbund .
```

### 5.4.2.1 Reihen und Reihungen

Reihen und Reihungen werden durch Zusammenfassen mehrerer Objekte gleichen Typs (den Elementen) gebildet. Für Reihen ist die Anzahl der Elemente statisch bestimmt und wird in der Schablone als ganzzahlige Konstante angegeben:

```
Reihe: ROW ganze_Zahl Elementtyp Packung.
```

Die Anzahl der Elemente einer Reihung wird erst bei der Ausführung des Programmes bestimmt. Reihungen können deshalb nur in den Speicherzonen für dynamisch generierte Komponenten von Verbunden untergebracht werden.

```
Reihung: ARRAY Elementtyp Packung.
Packung: PACKED | .
Elementtyp:
      einfacher Typ | Schablonenbezeichner.
```

Als Elementtyp darf nur ein einfacher Typ oder der Schablonenbezeichner eines statischen Verbundes angegeben werden.

Gepackte Reihen oder Reihungen sollten unter bestmöglicher Ausnutzung des Speichers implementiert werden. Dafür werden Einschränkungen bei der Bildung von Zugriffswegen und höhere Ausführungszeiten beim Zugriff auf die Elemente in Kauf genommen.

Das Schablonen-Konzept umfasst keine mehrstufigen Reihen und Reihungen. Die Funktionen zur Bildung von Zugriffswegen für die Elemente sind deshalb einfach zu implementieren. Sie können von dem Übersetzer, der die AMICO-Programme erzeugt, zur Bildung von Deskriptoren angewendet werden, die mehrstufige auf einstufige Reihen oder Reihungen abbilden.

Zeichenreihen können durch Objekte des Typs

ROW ganze_Zahl CHAR PACKED abgekürzt: STRING ganze_Zahl

oder

ARRAY CHAR PACKED abgekürzt: STRING

dargestellt werden. Für diesen Typ sind spezielle Funktionen definiert, mit denen Ausschnitte aus Zeichenreihen kopiert werden können.

Der Typ

ROW ganze_Zahl BOOL PACKED abgekürzt: SET ganze_Zahl

kann zur Darstellung von Mengen durch ihre charakteristische Funktion verwendet werden. Falls die ganze Zahl nicht grösser als die Implementierungskonstante MAXSET ist, (die nicht kleiner als die Mächtigkeit des Typs CHAR sein soll) wird dafür abkürzend SET ganze_Zahl geschrieben. Für diesen Typ sind über die Zugrifffunktionen für Reihen hinaus die üblichen Mengenoperationen definiert.

### 5.4.2.2 Verbund-Typen

Das Verbundkonzept hat in AMICO eine zentrale Bedeutung für die Beschreibung der Speicherorganisation. Es ist deshalb weitergehend als ähnliche Konzepte in Programmiersprachen. Eine Verbund-Schablone beschreibt Objekte, die aus im allgemeinen mehreren Objekten (Komponenten) verschiedener Typen zusammengesetzt sind. Sind alle Komponenten durch die Schablone definiert, so sprechen wir von einem statischen Verbund. Variable Verbunde enthalten darüber hinaus einen Speicherbereich, der zur Ausführungszeit wechselnde Komponenten aufnimmt. Die Generierung von Objekten wird auf das dynamische Zufügen solcher Komponenten zurückgeführt. Die Zugriffswege für die statischen und variablen Komponenten sind unterschiedlich.

Die Schablonen haben folgende Form:

```
statischer_Verbund:
        RECORD statischer_Teil END .
variabler_Verbund:
        RECORD statischer_Teil variabler_Teil END .
statischer_Teil:
        [ Packung ] allgemeine_Komponenten <,>
        alternative_Komponenten .
```

```
allgemeine_Komponenten:
        [ Komponenten ] .
alternative_Komponenten:
        [ VARIANT Komponenten ]* .
Komponenten:
        ( Komponentenbezeichner
        <=> statischer_Typ || <,> ) .
statischer_Typ:
        Schablonenbezeichner | einfacher_Typ | Reihe .
variabler_Teil:
        DYNAMIC [ Umfang ] Typspezifikationen .
Typspezifikationen:
        ( Schablonenbezeichner || <,> ) .
Umfang:
        ganze_Zahl .
```

Für einen Teil des statisch definierten Bereiches können mehrere Interpretationsvorschriften als Varianten angegeben werden. Die Anzahl und die Typen der Komponenten der Varianten brauchen nicht übereinzustimmen. (Hier gilt wie für die Anwendung von Schablonen allgemein, dass zwar Speicherbereiche aber nicht deren Inhalte uminterpretiert werden können. Dies ist bei der Erzeugung des AMICO-Programmes sicherzustellen.) Zur Bildung von Zugriffswegen werden den statisch definierten Komponenten symbolische Bezeichner zugeordnet.

Mit der Spezifikation des Umfangs des variablen Teiles eines Verbundes wird der maximale Speicherumfang aller zugleich zu dem Verbund gehörigen variablen Komponenten beschränkt. Als Mass für den Umfang wird eine Anzahl von Speichereinheiten angegeben. An dieser Stelle nehmen wir eine Abhängigkeit von der Rechenanlage und der Abbildung der AMICO-Datenobjekte in Kauf. Ist der Umfang nicht spezifiziert, so wird der Speicher für die variablen Komponenten bei ihrer Generierung aus dem umfassenden Verbund bezogen. In diesem Fall können nur in bestimmten Speichersituationen Komponenten zugefügt oder entfernt werden (siehe Abschnitt 5.5).

Die Typen der dynamisch zu generierenden Komponenten können als weitere Spezifikation des variablen Teiles angegeben werden. Es wird damit jedoch keine Aussage über die Anzahl und Reihenfolge solcher Komponenten gemacht.

## 5.5 Speicherorganisation

Die Speicherorganisation der abstrakten Maschine wird mit dem in Abschnitt 5.4.2.2 definierten Verbundbegriff beschrieben. Sie ist so flexibel, dass die unterschiedlichen Anforderungen verschiedener Programmiersprachen berücksichtigt werden können.

### 5.5.1 Adressraum eines AMICO-Programmes

Der gesamte Adressraum eines AMICO-Programmes wird durch eine Verbund-Schablone beschrieben, die von den übrigen Schablonen durch den vorgeschriebenen Bezeichner 'main' unterschieden ist. Zu dieser Schablone existiert während der Programmausführung genau ein Objekt. Die Komponenten dieses Verbundes können nach den Regeln aus 5.4.2.2 spezifiziert werden. Alle Schablonen, die Programm-Umgebungen (Schachteln) beschreiben (siehe Abschnitt 5.7), müssen als statische oder mögliche variable Komponenten des Verbundes 'main' definiert sein. Dadurch ist es möglich, die Zugriffsfunktionen für Umgebungsobjekte mit speziellen, der Rechenanlage angepassten Techniken zu implementieren.

Die statischen Komponenten können z.B. den Speicherbereich für globale Grössen, Programm-Konstante und die Schachteln nicht rekursiver Programmoduln beschreiben. Der Zugriff auf alle diese Objekte kann mit absoluten Adressen implementiert werden. Für einige Programmiersprachen reicht eine solche statisch definierte Speicherstruktur aus. Eine FORTRAN-ähnliche Speicherorganisation könnte wie folgt definiert werden:

```
TYPE main =
  RECORD common = c,
         subr1 = rec1,
         subr2 = rec2,

  END;
TYPE c = ..., rec1 = ..., ...
```

Eine dynamische Speicherorganisation wird durch den variablen Teil des Verbundes 'main' beschrieben. Die Spezifikation des Umfangs der variablen Speicherzone ist in [illegible] obligatorisch. Die Schablonen aller Umgebungs-

Schachteln werden als mögliche Typen der variablen Komponenten angegeben. Nach dem Kellerprinzip können Schachtel-Verbunde als variable Komponenten in den Verbund 'main' eingefügt oder aus ihm entfernt werden. Auf diese Weise kann z.B. eine Speicherorganisation für ALGOL-60 definiert werden:

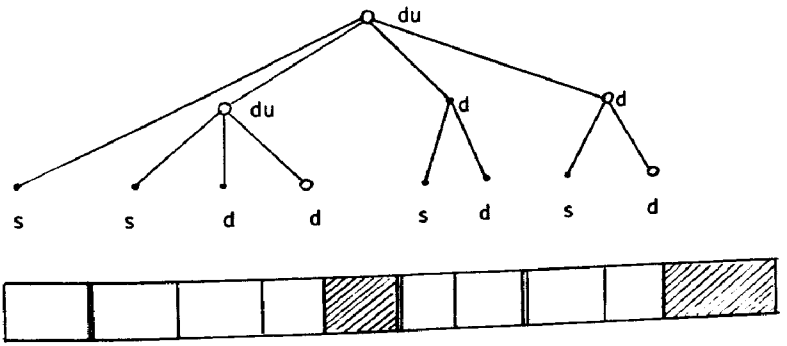
```
TYPE main =
  RECORD global = rec0,
         nichtrekursive_Prozedur1 = rec1,
         nichtrekursive_Prozedur2 = rec2,
         DYNAMIC 1000  Block1, Block2, ..., Blockn
  END;
TYPE rec0= ..., Block1 = ..., ...
```

Die Generierung und Beseitigung von variablen Komponenten der Verbunde erfolgt nach dem Kellerprinzip. Darüber hinaus gelten folgende Einschränkungen der dynamischen Veränderbarkeit von Verbunden:

Ein Verbund ist nur dann veränderbar (d. h. es kann eine Komponente zugefügt oder die letzte Komponente entfernt werden), wenn der Umfang seines variablen Speicherbereiches spezifiziert ist oder er selbst die letzte variable Komponente eines veränderbaren Verbundes ist.

Der Speicher des AMICO-Programmes bildet zu jedem Zeitpunkt eine Baumstruktur, in die nach der obigen Regel an bestimmmten Knoten weitere Unterbäume eingefügt werden können.

Als Beispiel geben wir einen Baum an, in dem die Knoten die Verbunde im Speicher repräsentieren. Sie sind mit s (=statischer Verbund), d (=variabler Verbund ohne Umfangs-Spezifikation) und du (=variabler Verbund mit Umfangs-Spezifikation) markiert. Die Verbunde, die in dieser Speichersituation veränderbar sind, werden durch o, die übrigen durch • gekennzeichnet. Darunter wird eine Projektion des Baumes angegeben, die zeigt, wie eine solche Speicherstruktur auf einen linearen Speicher abgebildet werden kann. Die Speicherbereiche, in die weitere Komponenten dynamisch eingefügt werden können, sind schraffiert gezeichnet.




Mit diesem Modell lassen sich auch Speicherorganisationen für komplexere Programmiersprachen (wie z.B. BALG) beschreiben:

```
TYPE main =
  RECORD global = rec0
         DYNAMIC 10000  Modul1, Modul2, ..., Modul n
  END;

TYPE Modul1 =
  RECORD >> lokale Komponenten <<
         DYNAMIC  1000
  END, ...
```

### 5.5.2 Annahmen zur Implementierung

Der Adressraum eines AMICO-Programmes kann als lineare oder mindestens stückweise lineare Folge von Stellen aufgefasst werden. Stellen sind Werte des Typs ADDR. Sie werden in Zugriffsfunktionen und zur Bildung von Zugriffswegen auf Objekte verwendet (siehe Abschnitt 5.4). Die Implementierung des beschriebenen Speichermodells auf einer Rechenanlage muss die folgenden Annahmen erfüllen:

a) Jedem Objekt, das nicht Teil eines gepackten zusammengesetzten Objektes ist, kann eine während seiner Existenz unveränderliche Stelle zugeordnet werden.

b) Die statischen Komponenten eines Verbundes werden in einem zusammenhängenden Speicherbereich untergebracht. Die Reihenfolge, die durch die Schablone vorgegeben ist, wird nicht verändert.

c) Die Elemente von Reihen und Reihungen werden in einem zusammenhängenden Speicherbereich untergebracht.

d) Die Differenz der Stellen zweier Objekte, die nach b) oder c) aufeinander folgen, wird allein durch den Typ des ersten bestimmt.

e) Ist ot ein Teilobjekt des Objektes o, so hängt es nur vom Typ von ot ab, ob es durch direktes Enthaltensein oder durch einen Bezug in o integriert ist.

### 5.5.3 Generierung und Beseitigung von Objekten

Ein Objekt zu einer Schablone s wird dadurch generiert, dass in einem variablen Verbund v eine neue Komponente eingefügt wird. Das Ergebnis der Generierung ist die Stelle des Objektes. Die Operanden der Generatoren sind die Stelle des Verbundes v, der vom angegebenen Verbundtyp ist, und im Falle des Reihungsgenerators zusätzlich die Anzahl der Elemente der Reihung. Das Objekt existiert so lange wie v existiert oder bis das Objekt aus v entfernt wird. Danach ist das Ergebnis von Zugriffen auf das Objekt undefiniert.

Mit den Operationen REMOVE, REMOVEARRAY, REMOVETO und REMOVEALL werden variable Komponenten aus einem Verbund v entfernt. Der erste Operand gibt jeweils die Stelle von v an, der zweite ggf. die Stelle des zu entfernenden Objektes. In den Operationen REMOVE und REMOVEARRAY muss dies die Stelle der letzten variablen Komponente von v sein. Die Operation REMOVETO entfernt das Objekt mit der angegebenen Stelle und alle später generierten Objekte. Die Operation REMOVEALL entfernt alle variablen Komponenten von v.

Die Operation NEXTADDR liefert als Ergebnis die gleiche Stelle, die ein stattdessen ausgeführter Generator liefern würde. Sie kann zur Bestimmung des Operanden der Funktion REMOVETO verwendet werden, nicht aber in Zugriffsfunktionen.

## 5.6 Zugriffswege und -funktionen

Bei der Beschreibung der Zugriffsfunktionen und der Funktionen zur Bildung von Zugriffswegen unterscheiden wir folgende Begriffe:

- Der absolute Zugriffsweg zu einem Objekt ist seine Stelle im Adressraum des Programmes, falls sie existiert. Sie ist ein Wert vom Typ ADDR.

- Ein relativer Zugriffsweg ist die Differenz zwischen zwei Stellen in einem linearen, zusammenhängenden Teil des Adressraumes. Er ist ein Wert vom Typ COUNT.

- Ein Index ist die Nummer eines Elementes einer Reihe oder Reihung. (Der Index des ersten Elementes jeder Reihe oder Reihung ist 0.) Er ist ebenfalls ein Wert vom Typ COUNT.

- Der Inhalt eines Objektes wird algorithmisch definiert: Wird eine Funktion zur Inhaltsbeschaffung (z.B. LOAD) auf die Stelle eine Objektes eines einfachen Typs angewendet und ist sie mit dem Typ t spezifiziert, so ist das Ergebnis ein Wert w vom Typ t, falls das Objekt existiert, mit der Typspezifikation t generiert wurde, w der Operand der zuletzt ausgeführten Funktion zur Inhaltsbestimmung (z.B. STORE) ist, und diese ebenfalls durch t spezifiziert ist. Für Objekte zusammengesetzter Typen gilt diese Definition komponenten- bzw. elementweise. (Eine Trennung dieser Definitionen ist nicht möglich, ohne Annahmen über die Codierung von Werten zu machen.)

### 5.6.1 Zugriffe auf Inhalte

Die elementaren Zugriffsfunktionen LOAD und STORE sind anwendbar auf die Stellen von Objekten eines elementaren Typs oder des Typs SET n. Sie beschaffen den Inhalt eines Objektes als Operand bzw. ersetzen den Inhalt eines Objektes durch ihren Operanden. Die Funktion STOREKEEP hat die gleiche Wirkung wie das Hintereinanderausführen von STORE und LOAD für dasselbe Objekt.

Mit den Funktionen EXCHANGEVAL, ROTATE und DELETE werden Operanden permutiert bzw. entfernt.

Zu Objekten, die Komponenten bzw. Elemente von gepackten Verbunden bzw. Reihen oder Reihungen sind, sind keine Stellen definiert. Ihr Inhalt wird durch spezielle Funktionen (LOADPACKEDFIELD, STOREPACKEDFIELD usw.) beschafft bzw. bestimmt. Die Zugriffswege auf die Objekte werden dabei implizit aus der Stelle des gepackten, zusammengesetzten Objektes und dem Elementindex bzw. der Verbundschablone und dem Komponentenbezeichner bestimmt.

Mit TRANSFER-Funktionen wird der Inhalt von Objekten durch den Inhalt anderer Objekte bestimmt. Sie sind anwendbar auf Objekte zu statischen Verbunden und aufeinanderfolgende Elemente nicht gepackter Reihen oder Reihungen. Die Operanden sind die Stellen des Ziel- und des Ursprungs-Objektes (TRANSFER) bzw. des ersten betroffenen Reihen- oder Reihungselementes und die Anzahl der zu übertragenden Elemente (TRANSFERELEMS). Für die Übertragung von aufeinanderfolgenden Elementen von Zeichenreihen existiert eine spezielle Funktion, von der wie oben beschrieben die Zugriffswege auf die Elemente aus den angegebenen Indizes implizit bestimmt werden. (Entsprechende Funktionen für andere gepackte Reihen oder Reihungen sind nicht definiert.)

### 5.6.2 Berechnung von Zugriffswegen

Zur Berechnung von Zugriffswegen für nicht gepackte Teilobjekte zusammengesetzter Objekte unterscheiden wir vier Funktions-Klassen. Während mit den Funktionen der ersten Klasse direkt absolute Zugriffswege auf beliebige Teilob-

jekte bestimmt werden, dienen die Funktionen der übrigen Klassen zum Aufbau von Deskriptoren und dem Zugriff mit Hilfe von Deskriptoren.

Mit den Funktionen SELECTFIELD, INDEXROW und INDEXARRAY wird aus der Stelle eines Verbundes bzw. einer Reihe oder Reihung und dem Bezeichner einer (statischen Komponente) bzw. einem Index die Stelle des Teilobjektes bestimmt.

Die Funktionen FIRSTRECFIELD, FIRSTROWELEM und FIRSTARRAYELEM bilden die Stelle eines zusammengesetzten Objektes auf die Stelle des ersten Teilobjektes ab. Sie dienen z.B. beim Aufbau von Deskriptoren zur Bestimmung der "Basis-Adresse". (Falls die Schachtelung zusammengesetzter Objekte durch physikalisches Enthaltensein (statt durch Bezüge) implementiert wird, sind dies Identitätsfunktionen.)

Relative Zugriffswege zwischen einer als Operand angegebenen Anzahl von Objekten des spezifizierten, statischen Typs werden mit der Funktion RELADDR bestimmt.

Die Funktion INCRABSADDR verknüpft einen absoluten mit einem relativen zu einem neuen absoluten Zugriffsweg. Sie kann zur Berechnung von Stellen mit Hilfe von Deskriptoren und zur "linearen Adressfortschaltung" eingesetzt werden.

### 5.6.3 Bezeichnete Stellen

Zur Vereinfachung der Formulierung der Zugriffsfunktionen innerhalb eines Segments kann einer Stelle des Adressraumes, die als Operand erarbeitet wird, mit Hilfe der Funktion DEF ein Bezeichner zugeordnet werden. Die Zuordnung besteht unverändert von der Ausführung dieses Operators bis zur Beendigung der Ausführung des Segments. Der Gültigkeitsbereich des Bezeichners ist das Segment, das seine Definition enthält. Er kann in der Spezifikation des Operators USE zur Bildung eines Operanden vom Typ ADDR verwendet werden. In der Operationenfolge darf diesem Operator kein Operator zur Ablaufsteuerung vorausgehen. Dieses Hilfsmittel ist geeignet zur Festlegung eines oder mehrerer Teil-Adressräume, auf deren Objekte in dem Segment zugegriffen wird. Im Modell der Maschine in Abschnitt 5.1 entsprechen die Adressregister den bezeichneten Stellen.

## 5.7 Umgebungen

Der Umgebungsbegriff ist charakteristisch für Sprachen zur strukturierten Programmierung. Eine Umgebung eines Programmabschnitts ist die Menge der Datenobjekte, die während der Ausführung des Abschnittes existieren und für die es in dem Abschnitt gültige Zugriffswege gibt. Durch textuelle (oder mit speziellen Sprachkonstruktionen erzielte) Schachtelung von Abschnitten werden Umgebungen erweitert um die Objekte und Zugriffswege der inneren Abschnitte.

Dieses Umgebungsmodell wird im allgemeinen durch eine Kellerorganisation für Schachtelverbunde (activation records) implementiert, die jeweils die Objekte mit gleicher Lebensdauer zusammenfassen. Durch Bezüge zwischen solchen Verbunden wird die Relation "U1 ist statischer Vorgänger von U2" beschrieben, die bedeutet, dass die Umgebung U2 durch Erweiterung direkt aus U1 hervorgegangen ist. Die Zugriffswege auf die Objekte einer Umgebung können mit Hilfe von Bezügen auf die Schachtelverbunde, die zu der Umgebung gehören, gebildet werden. Die Bezüge werden häufig durch Blockindexregister implementiert. (Diese Technik wurde in [Di63] vorgestellt.) Auf einigen Rechenanlagen sind die dafür nötigen Funktionen als Grundoperationen vorhanden.

Das Umgebungsmodell ist in AMICO durch einige zusätzliche Funktionen integriert. Sie enthalten alle für die Implementierung mit Blockindexregistern erforderlichen Informationen. Ihre Wirkung kann auf die in den vorangehenden Abschnitten definierten Funktionen zurückgeführt werden. Im Abschnitt 5.5 wurde gezeigt, dass sich die Speicherorganisation von AMICO zur Implementierung eines Schachtelkellers eignet.

Die Schachtelverbunde – und nur diese – werden als Komponenten des variablen Speicherbereiches des Verbundes 'main' generiert. Alle ihre Schablonen werden als mögliche Typen der variablen Komponenten aufgezählt. Jeder dieser Schablonen ist implizit eine zusätzliche Komponente zugeordnet, auf die nur mit den hier angegebenen Funktionen zugegriffen werden kann. Sie nimmt den Bezug auf den statischen Vorgänger auf.

Im folgenden beschreiben wir die Funktionen, mit denen die Schachtelverbunde zu Segmenten generiert, die Zugriffs-

wege auf sie bestimmt und Bezeichner zugeordnet werden. Diese Funktionen dürfen in einem Segment nur am Anfang der Operationenfolge stehen. Die unten angegebene Reihenfolge ist bindend.

Wir unterscheiden vier grundsätzlich verschiedene Fälle bei der Bestimmung der Umgebung eines Segments:

a) Es wird ein neuer Schachtelverbund generiert und in eine Umgebung eingebettet, die nicht notwendig die des aufrufenden Segmentes ist. (In höheren Programmiersprachen liegt diese Situation z.B. bei Prozeduren vor, die als aktuelle Parameter übergeben werden.) Die erste Operation (ENVDEPTH) gibt die Schachtelungstiefe der innersten Schachtel als ganze Zahl an. Sie hat keine Wirkung, sondern liefert notwendige Information für die Register-Zuordnung bei einer Implementierung durch Blockindexregister. Mit dem Operator NEWENV wird der Verbund als neue variable Komponente des Verbundes 'main' erzeugt. Die Operation SETSTATPRED bestimmt den statischen Vorgänger dieses Verbundes. Seine Stelle muss der letzte Operand des aufgerufenen Segmentes sein. Mit den darauf folgenden Operationen ENV werden nacheinander lückenlos die weiteren statischen Vorgänger ermittelt. Die Folge kann beendet werden, wenn auf weiter aussen liegende Schachteln nicht zugegriffen wird.

| Die Operationenfolge | hat die Wirkung |
|---|---|
| ENVDEPTH (/) n ; | |
| NEWENV (/) t1 b1; | USE (/ADDR) main;<br>GEN (ADDR/ADDR) t1;<br>DEF (ADDR/) b1; |
| SETSTATPRED (ADDR/)t2 b2; | DEF (ADDR/) b2;<br>USE (/ADDR) b1;<br>SELECTFIELD(ADDR/ADDR)t1 sV;<br>USE (/ADDR) b2;<br>STOREKEEP (ADDR ADDR/ADDR); |
| ENV (/) t3 b3; | SELECTFIELD(ADDR/ADDR)t2 sV;<br>LOAD (ADDR/ADDR);<br>DEF (ADDR/) b3; |
| ENV (/) t4 b4; | USE (/ADDR) b3;<br>SELECTFIELD(ADDR/ADDR)t3 sV;<br>LOAD (ADDR/ADDR);<br>DEF (ADDR/) b4; |

```
                                    USE (/ADDR) b4;
  ...                               ...
```

b) Im Unterschied zu a) wird kein neuer Schachtelverbund generiert. Es sind nur die Zugriffswege auf die durch den letzten Operanden des Segmentaufrufs definierte, schon existierende Umgebung zu bestimmen. Die Operationenfolge beginnt dann mit den Operationen

```
ENVDEPTH (/) n;
FIRSTENV (ADDR/) t1 b1;     DEFADDR (ADDR/) b1;
                            USEADDR (/ADDR) b1;
ENV (/) t2 b2;              s.o.
 ...                        ...
```

c) Es wird ein neuer Schachtelverbund in derselben Umgebung generiert, aus der das Segment aufgerufen wird. Die explizite Spezifikation der Umgebung wird ersetzt durch den Operator SAMEENV, der ein anderes Segment spezifiziert, in dem dieselbe Umgebung definiert ist. Die Gültigkeit der dort definierten Bezeichner für Schachtelverbunde wird damit auf dieses Segment ausgedehnt. Die Stellen der Schachtelverbunde werden aus der beim Aufruf bestehenden Umgebung entnommen. Ein Segment-Operand wird dafür nicht benötigt.

```
ENVDEPTH (/) n;
NEWENV (/) t1 b1;
SAMEENV (/) s
```

d) Im Unterschied zu c) wird kein neuer Schachtelverbund generiert:

```
ENDEPTH (/) n;
SAMEENV (/) s
```

Die so generierten Schachtelverbunde werden durch explizit anzugebende Operationen aus dem Schachtelkeller entfernt.

Die Operation

```
REMOVEENV (/) t  b;
```

hat die Wirkung

```
                    USE (/ADDR) main;
                    USE (/ADDR) b;
und                 REMOVE (ADDR ADDR/) main t
```

```
REMOVETOENV (/) b;          USE (/ADDR) main;
                            USE (/ADDR) b;
                            REMOVETO (ADDR ADDR/) main
```

Mit Segmenten, in denen zwar ein Schachtelverbund generiert, jedoch nicht entfernt wird, können allgemeinere Programmoduln beschrieben werden, deren Adressraum nach der Ausführung des Programmabschnitts erhalten bleibt.

Für Komponenten von Schachtelverbunden wird eine abkürzende Schreibweise der Zugriffsfunktionen definiert:

| Die Operation | hat die Wirkung |
|---|---|
| SELECTENVFIELD(/ADDR)t b k | USE (/ADDR) b;<br>SELECTFIELD(ADDR/ADDR) t k |

## 5.8 Dateien

Wir definieren hier ein einfaches Datei-Konzept mit Hilfe des in Abschnitt 5.2 eingeführten Verbundbegriffs. Dateien werden als zusammengesetzte Datenobjekte aufgefasst, die aus einigen Komponenten zur Beschreibung der Datei-Eigenschaften und einer linear geordneten Menge von Datei-Sätzen bestehen, deren Struktur durch eine Verbundschablone beschrieben ist. Die statischen Komponenten enthalten alle Informationen, die die Datei charakterisieren und für Zugriffe auf die übrigen Komponenten nötig sind (z.B. Dateibezeichner, Umfang, Zugriffsart, Speichermedium usw.). Die Datei-Sätze sind die variablen Komponenten des Verbundes. Der Adressraum des Dateiverbundes liegt ausserhalb des Adressraumes des AMICO-Programmes. Zu einer Datei, auf die von einem AMICO-Programm zugegriffen wird, muss im Adressraum des Programmes ein Verbund existieren, in den der Inhalt der statischen Komponenten des Dateiverbundes kopiert wird.

Dateibeschreibungen sind in ihrem Aufbau und Inhalt für verschiedene Rechenanlagen sehr unterschiedlich. Die statischen Komponenten der Dateiverbunde werden deshalb implizit durch Angaben zur Verwendungsart der Datei spezifiziert. Dabei werden nur die wichtigsten Standardfälle berücksichtigt, die den Benutzeranforderungen an diese abstrakte

Maschine genügen und auf gebräuchlichen Rechenanlagen realisiert werden können.

Die Dateischablonen haben folgende Struktur:

```
Dateischablone:
    FILE Dateispezifikationen variabler_Teil END .
Dateispezifikationen:
    Zugriffsart Zugriffsrichtung Speichermedium .
Zugriffsart:
    SEQUENTIAL / DIRECT .
Zugriffsrichtung:
    IN / OUT / INOUT .
Speichermedium:
    PRINTER / READER / TAPE / RANDOM .
```

Die Spezifikationen von Zugriffsart und Zugriffsrichtung bestimmen die Anwendbarkeit der verschiedenen Dateioperationen. Wir unterscheiden, ob auf die Dateikomponenten lesend und/oder schreibend, sequentiell oder direkt zugegriffen wird.

Als Speichermedium können die Standardfälle für die Erzeugung lesbarer Druckausgabe und die Verarbeitung lesbarer Eingabe spezifiziert werden, sowie Hintergrund-Speichermedien für sequentiellen und direkten Zugriff. Die für die Datei spezifizierten Zugriffsarten und -richtungen müssen mit den Speichermedien verträglich sein.

Die Spezifikation des variablen Teiles der Dateischablone hat die gleiche Form und Bedeutung wie sie für allgemeine Verbunde definiert ist: Die Umfangsangabe spezifiziert grob die Grösse des Adressraumes (Anzahl der Sätze) der Datei; die Typangaben bestimmen die möglichen Satztypen. Beide Spezifikationen können fehlen. Es sind nur Sätze zu statischen Schablonen möglich.

Beispiele für Dateischablonen-Definitionen:

```
TYPE Standardeingabe =
  FILE SEQUENTIAL IN READER
       DYNAMIC Karte
  END.
  Karte = STRING 80;
```

```
TYPE Standardausgabe =
  FILE SEQUENTIAL OUT PRINTER
       DYNAMIC Zeile
  END,
  Zeile = STRING 132;

TYPE direktfile =
  FILE DIRECT INOUT OTHER
       DYNAMIC 1000  t
  END,
  t = RECORD ... END
```

Im folgenden beschreiben wir die auf Dateien anwendbaren Operationen. Die Datei, auf die eine Operation anzuwenden ist, wird jeweils durch die Stelle eines Verbundes spezifiziert, der die Dateibeschreibung enthält. Alle Operationen liefern als Ergebnis einen logischen Wert, der anzeigt, ob die beschriebene Wirkung erzielt wurde.

Mit der Operation FILEOPEN wird der Adressraum für die Datei beschafft bzw. für Zugriffsoperationen zugänglich gemacht, und der Inhalt der statischen Komponenten in den Verbund kopiert, dessen Stelle als Operand angegeben ist. Die Operanden geben die Stelle und Länge einer Zeichenreihe an, deren Inhalt als Dateibezeichnung interpretiert wird. Sie dient zur Identifikation der Datei in der Umgebung, in der das Programm abläuft. Falls die Zugriffsrichtung der Datei mit IN spezifiziert ist und keine geeignete, existierende Datei zugeordnet werden kann, liefert diese Operation als Ergebnis den Wert FALSE.

Nach Ausführen der Operation FILECLOSE sind keine weiteren Zugriffe auf die Datei-Sätze möglich. Sie veranlasst, dass die Datei in der Umgebung des Programmes sichergestellt wird. Falls für die Datei das Speichermedium PRINTER spezifiziert ist, wird dann die Ausgabe auf einem Druckgerät veranlasst.

Durch die Operation FILEREMOVE wird die Datei beseitigt.

Nach Ausführen der Operation FILEOPEN und vor der Ausführung von FILECLOSE oder FILEREMOVE kann mit den folgenden Operationen auf die Datei-Sätze zugegriffen werden. Abhängig von der Spezifikation der Zugriffsart und -richtung sind verschiedene Funktionen anwendbar:

a) lesende Zugriffe bei den Zugriffsrichtungen IN oder INOUT (READSEQ, READDIRECT),

b) schreibende Zugriffe bei den Zugriffsrichtungen OUT oder INOUT (WRITESEQ, WRITEDIRECT),

c) sequentielle Zugriffe bei der Zugriffsart SEQUENTIAL (READSEQ, WRITESEQ) und

d) indizierte Zugriffe bei der Zugriffsart DIRECT (READDIRECT, WRITEDIRECT).

Diese Funktionen sind vergleichbar mit den in Abschnitt 5.6 definierten Funktionen, die den Inhalt eines Objektes ersetzen durch den Inhalt eines anderen vom gleichen Typ (TRANSPER). In diesem Fall liegt eines der Objekte nicht im Adressraum des Programmes, sondern in dem der Datei. Das Objekt im Adressraum des Programmes wird durch die als Operand angegebene Stelle spezifiziert. Der Satz der Datei, auf den zugegriffen wird, ist bei sequentiellem Zugriff implizit durch die Anordnung der Sätze bestimmt.

Im Falle eines direkten Zugriffs wird er durch den Wert eines weiteren Operanden vom Typ COUNT bestimmt. Dazu sind den Datei-Sätzen eindeutig nicht negative, ganze Zahlen zugeordnet. Beim schreibenden, sequentiellen Zugriff wird implizit ein weiterer Satz als Datei-Komponente generiert, beim schreibenden, direkten Zugriff nur dann, wenn noch kein Satz mit der angegebenen Identifikation existiert. Falls dabei der Umfang der Datei überschritten wird, ist das Ergebnis der Operation der Wert FALSE.

Das Ergebnis der n+1-ten Ausführung der Funktion READSEQ ist der Wert FALSE, wenn die Datei n Sätze enthält. (Die Wirkung weiterer Ausführungen dieser Funktion ist nicht definiert.) Das Ergebnis der Funktion READDIRECT ist der Wert FALSE, falls kein Datei-Satz mit der angegebenen Identifikation existiert.

## 5.9 Liste der Grundfunktionen

In der folgenden Liste werden die AMICO-Operatoren mit den jeweils definierten Operanden- und Ergebnistypen und ihren Spezifikationen in dem in Abschnitt 5.2 festgelegten Befehlsformat angegeben. Für Bezeichner von einfachen Typen, Schablonen, Segmenten, Stellen und für Folgen solcher Bezeichner werden (indizierte) Abkürzungen verwendet. Die Schreibweise einer zulässigen AMICO-Operation ergibt sich durch konsistentes Einsetzen entsprechender Bezeichner für die Abkürzungen:

```
IL   : INT | LONGINT.
I    : SHORT | COUNT | IL.
RL   : REAL | LONGREAL.
IR   : I | RL.
M    : Bezeichner einer Schablone SET n.
ET   : IR | S | BOOL | CHAR | ADDR | SEG.
ET-F: [ET]*.
S    : Schablonenbezeichner.
D    : Schablonenbezeichner für Datei.
V    : Schablonenbezeichner für Verbund.
SV   : Schablonenbezeichner für statischen Verbund.
VV   : Schablonenbezeichner für variablen Verbund.
R    : Schablonenbezeichner für Reihe.
A    : Schablonenbezeichner für Reihung.
AR   : A | R.
K    : Komponentenbezeichner.
SG   : Segmentbezeichner.
ST   : Bezeichner für eine Stelle.
L    : Markenbezeichner.
L-F  : Folge von Markenbezeichnern.
ESV  : ET | SV.
EV   : ET | V.
```

**Ablaufsteuerung (5.3)**

```
CALL(ET-F1/ET-F2)                    SG
DYNCALL(ET-F1 SEG/ET-F2)
RETURN(ET-F/)                        SG ganze Zahl
MULTIPLERETURN(ET-F/)                SG
RETURNMAIN(ET-F/)                    SG
EXITCALL(ET-F/)
```

```
DYNEXITCALL(ET-F SEG/)
LOOP(BOOL/)                          SG
LOOPCOUNT(COUNT BOOL/)               SG
DEFLABEL(/)                          L
JUMP(/)                              L
JUMPFALSE(BOOL/)                     L
CASE(COUNT/)                         L-F
```

Operationen auf Werten einfacher Typen (5.4.1)

```
CONST(/I) ganze Zahl
CONST(/RL) reelle Zahl
CONST(/BOOL) logischer Wert
CONST(/M) Mengen-Wert
CHANGE(ET1 ET2/ET2 ET1)
ROTATE(ET1 ET2 ET3/ET2 ET3 ET1)
DELETE(ET-F/)
DUPLICATE(ET/ET ET)
INCR(I/I)
DECR(I/I)
ADD(IR IR/IR)
SUB(IR IR/IR)
MULT(IR IR/IR)
DIV(IL IL/IL)
DIV(RL RL/RL)
EXP(IR I/IR)
EXP(IR IR/RL)
EQ(ET ET/BOOL)
NEQ(ET ET/BOOL)
LT(IR IR/BOOL)
GT(IR IR/BOOL)
LTE(IR IR/BOOL)
GTE(IR IR/BOOL)
CONV(IR1/IR2)
TRUNC(RL/IL)
ROUND(RL/IL)
CONV(CHAR/SHORT)
CONV(SHORT/CHAR)
GET(ADDR COUNT COUNT/COUNT IR)
PUTINT(ADDR COUNT COUNT IL/)
PUTREAL(ADDR COUNT COUNT COUNT RL/)
SINGLESET(SHORT/M)
SETKONJ(M M/M)
SETDISJ(M M/M)
```

```
SETDIF(M M/M)
SETCOMPL(M/M)
SETINCL(M M/BOOL)
SETELINC(SHORT M/BOOL)
AND(BOOL BOOL/BOOL)
OR(BOOL BOOL/BOOL)
XOR(BOOL BOOL/BOOL)
NOT(BOOL/BOOL)
```

Zugriffe auf Inhalte (5.6.1)

```
LOAD(ADDR/ET)
STORE(ADDR ET/)
STOREKEEP(ADDR ET/ET)
STORESTRING(ADDR COUNT/)          Zeichenreihe
EXCHANGEVAL(ADDR ET/ET)
LOADPACKEDFIELD(ADDR/ET)          V K
LOADPACKEDELEM(ADDR COUNT/ET)     AR
STOREPACKEDFIELD(ADDR ET/)        V K
STOREPACKEDELEM(ADDR COUNT ET/)   AR
TRANSFER(ADDR ADDR/) SV
TRANSFERELEMS(ADDR ADDR COUNT/) ESV
TRANSFERSTRING(ADDR COUNT ADDR COUNT COUNT/)
```

Berechnung von Zugriffswegen (5.4.3, 5.4.4)

```
DEFADDR(ADDR/)                          ST
USE(/ADDR)                              ST
USE(/SEG)                               SG
SELECTFIELD(ADDR/ADDR)                  V K
INDEXROW(ADDR COUNT/ADDR)               R
INDEXARRAY(ADDR COUNT/ADDR)             A
FIRSTRECFIELD(ADDR/ADDR)                V
FIRSTROWELEM(ADDR/ADDR)                 R
FIRSTARRAYELEM(ADDR/ADDR)               A
RELADDR(COUNT/COUNT)                    ESV
INCRABSADDR(ADDR COUNT/ADDR)
```

Generierung und Beseitigung von Objekten (5.3.3)

| | | |
|---|---|---|
| GEN(ADDR/ADDR) | VV | EV |
| GENARRAY(ADDR COUNT/ADDR) | VV | A |
| NEXTADDR(ADDR/ADDR) | VV | |
| REMOVE(ADDR ADDR/) | VV | EV |
| REMOVEARRAY(ADDR ADDR COUNT/) | VV | A |
| REMOVETO(ADDR ADDR/) | VV | |
| REMOVEALL(ADDR/) | VV | |

Umgebungen (5.7)

| | | | |
|---|---|---|---|
| ENVDEPTH(/) | ganze Zahl | | |
| NEWENV(/) | V | ST | |
| SETSTATPRED(ADDR/) | V | ST | |
| ENV(/) | V | ST | |
| FIRSTENV(ADDR/) | V | ST | |
| SAMEENV(/) | SG | | |
| REMOVENV(/) | V | ST | |
| REMOVETOENV(/) | ST | | |
| SELECTENVFIELD(/ADDR) | V | ST | K |

Dateien (5.8)

| | | |
|---|---|---|
| FILEOPEN(ADDR ADDR COUNT/) | D | |
| FILECLOSE(ADDR/) | D | |
| FILEREMOVE(ADDR/) | D | |
| READSEQ(ADDR ADDR/) | D | ESV |
| WRITESEQ(ADDR ADDR/) | D | ESV |
| READDIRECT(ADDR COUNT ADDR/) | D | ESV |
| WRITEDIRECT(ADDR COUNT ADDR/) | D | ESV |

## Anhang A

## Vordefinierte Attribut-Typen

Die im folgenden definierten Attribut-Typen dienen zur Beschreibung der dynamischen Semantik mit den Begriffen der abstrakten Maschine AMICO. Ihre Definition kann in den Sprachdefinitionen, die von unserem Übersetzer-erzeugenden System verarbeitet werden, vorausgesetzt werden. Die Definitionen entsprechen den syntaktischen Regeln für die AMICO-Programme (siehe Kapitel 5).

```
TYPE C_Code : STRUCT( C_Sch: C_Schablonendefinitionen,
                      C_Seg: C_Segmente );

TYPE C_Operationen : SETOF C_Operation LINEAR;

TYPE C_Segmente: SETOF C_Segment LINEAR;

TYPE C_Segment : STRUCT(C_Segopd, C_Segerg: C_Typen,
                        C_Segbez: IDENT,
                        C_Segopr: C_Operationen);

TYPE C_Operation: STRUCT(C_Opr: IDENT,
                         C_Opd, C_Erg: C_Typen,
                         C_Spez: C_Spezifikationen);

TYPE C_Spezifikationen: SETOF C_Spezifikation LINEAR;

TYPE C_Spezifikation: UNION(IDENT, INT, STRING);

TYPE C_Typen: SETOF IDENT LINEAR;

TYPE C_Schablonendefinitionen:
            SETOF C_Schablonendefinition LINEAR;

TYPE C_Schablonendefinition: STRUCT(C_Bez: IDENT,
                                    C_Schab: C_Schablone);

TYPE C_Schablone: UNION(C_Reihe, C_Reihung,
                        C_statischer_Verbund,
                        C_variabler_Verbund, C_Datei);
```

```
TYPE C_Reihe: STRUCT(C_Anz: INT, C_Elem: IDENT,
                     C_Packung: IDENT);

TYPE C_Reihung: STRUCT(C_Elem: IDENT, C_Packung: IDENT);

TYPE C_statischer_Verbund:
               STRUCT(C_Packung: IDENT,
                      C_allg_K: C_Komponenten,
                      C_alt_K: C_alternative_Komponenten);

TYPE C_alternative_Komponenten:
              SETOF C_Komponenten LINEAR;

TYPE C_Komponenten: SETOF C_Komponente LINEAR;

TYPE C_Komponente: STRUCT(C_Kompbez: IDENT,
                          C_Komptyp: IDENT);

TYPE C_variabler_Verbund:
              STRUCT(C_Packung:IDENT, C_allg_K: C_Komponenten,
                     C_alt_K: C_alternative_Komponenten,
                     C_var_Teil: C_variabler_Teil);

TYPE C_variabler_Teil:
              STRUCT(C_Umfang: INT, C_Komptypen: C_Typen);

TYPE C_Datei: STRUCT(C_Zugriffsart, C_Zugriffsrichtung,
                     C_Speichermedium: IDENT,
                     C_var_Teil: C_variabler_Teil);

TYPE C_Text:SETOF STRING LINEAR;
```

Der Typ C_Text dient zur Beschreibung grösserer, nicht explizit strukturierter Code-Abschnitte.

Neben diesen Attribut-Typen sind für alle AMICO-Operatoren Funktionen definiert, deren Ergebnis ein Objekt vom Typ C_Operation ist. Die frei wählbaren Typangaben und Spezifikationen werden jeweils als Parameter angegeben.

Beispiel:

C_STORE("int") steht für STORE(ADDR INT /)

## Anhang B

## Beispiel 1

Im folgenden definieren wir eine einfache, blockstrukturierte Programmiersprache mit den Begriffen unserer Beschreibungssprache (siehe Kapitel 3).

```
TYPE Objekte:
        SETOF Objekt KEY Objektbez LINEAR;
TYPE Objekt:
        STRUCT(Objektbez, Schachtelbez, Schachteltyp,
               Segmentbez: IDENT, Klasse: Var_Proz, Art:Arten);
TYPE Klauselbeschreibung:
        STRUCT(Segmentbez, Schachtelbez, Schachteltyp,
               Ergebnistyp: IDENT, BST: INT);
TYPE Arten: (int, real, void);
TYPE Art_Paar: STRUCT(Ausgangsart, Zielart: Arten);
TYPE Anpassungen: (weiten, entwerten, leer, fehler);
TYPE Zugriffsklasse: (Stelle, Inhalt);
TYPE Var_Proz: (Variable, Prozedur);


NONTERM Programm:
           gueltige_Objekte: Objekte,
           Code: C_Programm,
           BST:=0: INT,
           Bez:= "main": STRING;

NONTERM Klausel:
           Art: Arten,
           Anpassung: Anpassungen,
           Inhalte_Objekte: Objekte,
           gueltige_Objekte: Objekte,
           Beschreibung: Klauselbeschreibung,
           Schachteltypen: C_Typen,
           Schablonen: C_Schablonen,
           Code: C_Segmente;
```

```
NONTERM Vereinbarung:
            Definition: Objekt;

NONTERM Artangabe:
            Art: Arten;

NONTERM Anweisungen, Anweisung, Ausdruck, einf_Ausdruck:
            Art: Arten,
            Anpassung: Anpassungen,
            Code: C_Operationen;

NONTERM Benennung:
            Definition: Objekt,
            Zugriff: Zugriffsklasse,
            Anpassung: Anpassungen,
            Code: C_Operationen;


RULE Programm: Klausel "$"
SEMANTIC
  Programm.gueltige_Objekte:= Objekte();
  Klausel.Anpassung:= Anpassen(Klausel.Art, void);

  Programm.Code:=
    C_Text(Adressraum(Klausel.Schachteltypen),
           Klausel.Schablonen,
           Rahmenprogramm(Klausel.Beschreibung.Segmentbez),
           Klausel.Code)
END;

RULE Klausel: "begin" [ Vereinbarung // ";" ]* ";"
                      Anweisungen "end"
SEMANTIC
  TRANSFER Art, Anpassung WITH Anweisungen;
  Klausel.lokale_Objekte:=
    Objekte(Vereinbarung.Definition);
  Klausel.gueltige_Objekte:=
    Objekte SUPPRESSKEYS ( INCLUDING( Klausel.gueltige_Objekte,
                                      Programm.gueltige_Objekte),
                           Klausel.lokale_Objekte);
  Klausel.leere_Schachtel:=
    keine_Variablen(Klausel.lokale_Objekte);
```

```
Klausel.Beschreibung:=
  Klauselbeschreibung( newident, newident, newident,
                       CASE Klausel.Anpassung OF
                         leer: Art_in_Typ(Klausel.Art);
                         weiten: "real"
                       OUT C_leer
                       ESAC,
                       INCLUDING(Klausel.Beschreibung.BST,
                                 Programm.BST) +
                         CASE Klausel.leere_Schachtel OF
                           TRUE: 0; FALSE: 1
                         ESAC );
Klausel.Schablonen:=
  C_Schablonen( CONSTITUENTS Klausel.Schablonen,
                CASE Klausel.leere_Schachtel OF
                  TRUE: C_leer;
                  FALSE: C_statischer_Verbund
                           (Klausel.Beschreibung.Schachteltyp,
                            "unpacked",
                            Schachtelkomponenten
                              (Klausel.lokale_Objekte) )
                ESAC);
Klausel.Code:=
  C_Segmente
    (CONSTITUENTS Klausel.Code,
     C_Segment
       (C_leer, C_Typen(Klausel.Beschreibung.Ergebnistyp),
        Klausel.Beschreibung.Segmentbez,
        C_Operationen
          (C_ENVDEPTH(Klausel.Beschreibung.BST),
           CASE Klausel.leere_Schachtel OF
             TRUE: C_leer;
             Falso: C_NEWENV(Klausel.Beschreibung.Schachteltyp,
                             Klausel.Beschreibung.Schachtelbez)
           ESAC,
```

```
            C_SAMEENV(INCLUDING(Klausel.Beschreibung.Schachtelbez,
                                Programm.Bez) ),
            Anweisungen.Code,
            CASE Klausel.leere_Schachtel OF
              TRUE: C_leer;
              FALSE: C_REMOVEENV(Klausel.Beschreibung.Schachteltyp,
                                 Klausel.Beschreibung.Schachtelbez)
            ESAC,
            C_RETURN(Klausel.Beschreibung.Ergebnistyp)
           )
        )
      );
  Klausel.Schachteltypen:=
    C_Typen( CASE Klausel.leere_Schachtel OF
               TRUE: C_leer; FALSE: Klausel.Beschreibung.Schachteltyp
             ESAC,
             CONSTITUENTS Klausel.Schachteltypen)
END;


RULE Vereinbarung: Artangabe identsymbol
SEMANTIC
  Vereinbarung.Definition:=
    Objekt( identsymbol.iden,
            INCLUDING Klausel.Beschreibung.Schachtelbez,
            INCLUDING Klausel.Beschreibung.Schachteltyp,
            "", Variable, Artangabe.Art )
END;


RULE Artangabe: "integer"
SEMANTIC
  Artangabe.Art:=int
END;


RULE Artangabe: "real"
SEMANTIC
  Artangabe.Art:=real
END;
```

```
RULE Vereinbarung:
       "proc" [ Artangabe ] identsymbol "=" Klausel
SEMANTIC
  Vereinbarung.Definition:=
    Objekt( Identsymbol.ident, "", "",
            Klausel.Beschreibung.Segmentbez,
            Prozedur, [ Artangabe.Art, Void ] );
  Klausel.Anpassung:=
    Anpassen(Klausel.Art, [ Artangabe.Art, void ]);
  Klausel.anpassung =/ fehler
END;


RULE Anweisungen: Anweisung ";" Anweisungen
SEMANTIC
  TRANSFER Art, Anpassung WITH Anweisungen[2];
  Anweisung.Anpassung:=Anpassen(Anweisung.Art, void);

  Anweisungen[1].Code:=
    C_Operationen( Anweisung.Code, Anweisungen[2].Code )
END;


RULE Anweisungen: Anweisung
SEMANTIC TRANSFER
END;


RULE Anweisung: Ausdruck
SEMANTIC TRANSFER
END;


RULE Anweisung: Benennung ":=" Ausdruck
SEMANTIC
  Anweisung.Art:=Benennung.Definition.Art;
  Benennung.Definition.Klasse =/ Prozedur;
  Benennung.Zugriff:=Stelle;
  Benennung.Anpassung:=leer;
  Ausdruck.Anpassung:=Anpassen(Ausdruck.Art,Anweisung.Art);
  Ausdruck.Anpassung =/ fehler;
```

```
  Anweisung.Code:=
    C_Operation
      ( Benennung.Code, Ausdruck.Code,
        CASE Anweisung.Anpassung OF
          leer: C_STOREKEEP(Art_in_Typ(Anweisung.Art));
          entwerten: C_STORE( Art_in_Typ(Anweisung.Art));
          weiten: C_Operationen
                    ( C_STOREKEEP("int"),
                      C_CONV("int","real") )
        OUT C_leer
        ESAC )
END;


RULE Ausdruck: einf_Ausdruck
SEMANTIC TRANSFER
END;


RULE Anweisung: "print" Ausdruck
SEMANTIC
  Anweisung.Art:=void;
  Ausdruck.Anpassung:=Anpassen(Ausdruck.Art,real);
  Ausdruck.Anpassung =/ fehler;

  Anweisung.Code:=
    C_Operation( Ausdruck.Code,
                 C_CALL( C_Typen("real"),C_leer,"outreal"))
END;


RULE Ausdruck: Ausdruck "+" einf_Ausdruck
SEMANTIC
  Ausdruck[2].Art =/ void;
  einf_Ausdruck.Art =/ void;
  Ausdruck[1].Art:=
    CASE Ausdruck[2].Art = einf_Ausdruck.Art OF
      TRUE: Ausdruck[2].Art; FALSE: real
    ESAC;
  Ausdruck[2].Anpassung:=
    Anpassen(Ausdruck[2].Art,Ausdruck[1].Art);
  einf_Ausdruck.Anpassung:=
    Anpassen(einf_Ausdruck.Art,Ausdruck[1].Art);

  Ausdruck.Code:=
```

```
        ( Ausdruck[2].Code, einf_Ausdruck.Code,
          CASE Ausdruck[1].Art OF
            int: C_ADD("int","int","int");
            real: C_ADD("real","real","real")
          OUT C_leer
          ESAC,
          CASE Ausdruck[1].Anpassung OF
            weiten: C_CONV("int","real");
            entwerten: C_DELETE(Art_in_Typ(Ausdruck[1].Art))
          OUT C_leer
          ESAC)
END;


RULE einf_Ausdruck: Klausel
SEMANTIC
  TRANSFER Art, Anpassung;

  einf_Ausdruck.Code:=
    C_Operationen
      ( C_CALL( C_Typen(),
                Klausel.Beschreibung.Ergebnistyp,
                Klausel.Beschreibung.Segmentbez))
END;


RULE einf_Ausdruck: intsymbol
SEMANTIC
  einf_Ausdruck.Art:=int;

  einf_Ausdruck.Code:=
    C_Operationen( CASE einf_Ausdruck.Anpassung OF
                     leer: C_CONST("int",intsymbol.text);
                     weiten: C_CONST("REAL", intsymbol.text)
                   OUT C_leer
                   ESAC )
END;
```

```
RULE einf_Ausdruck: realsymbol
SEMANTIC
  einf_Ausdruck.Art:=real;

  einf_Ausdruck.Code:=
    C_Operationen( C_CONST("real",realsymbol.text))
END;


RULE einf_Ausdruck: Benennung
SEMANTIC
  einf_Ausdruck.Art:=Benennung.Definition.Art;
  Benennung.Zugriff:=Inhalt;
  TRANSFER Anpassung, Code
END;


RULE Benennung: identsymbol
SEMANTIC
  Benennung.Definition:=
    Identsymbol.text ID INCLUDING Klausel.gueltige_Objekte;

  Benennung.Code:=
    C_Operationen
      ( CASE Benennung.Definition.Klasse OF
          Prozedur: C_CALL( C_Typen(),
                            Art_in_Typ(Benennung.Definition.Art),
                            Benennung.Defintion.Segmentbez);
          Variable: C_Operationen
                      ( C_SELECTENVFIELD
                          (Benennung.Definition.Objektbez),
                        CASE Benennung.Zugriff OF
                          Inhalt: C_LOAD
                              (Art_in_Typ(Benennung.Definition.Art))
                          Stelle: C_leer
                        ESAC )
        ESAC,
        CASE Benennung.Anpassung OF
          weiten: C_CONV("int","real");
          entwerten: C_DELETE(Art_in_Typ(Benennung.Definition.Art))
        OUT C_leer
        ESAC )
END;
```

```
FUNCTION Anpassen (pri_Art,post_Art : Arten) Anpassungen :
  CASE pri_Art = post_Art OF
    TRUE: leer;
    FALSE: CASE Art_Paar(pri_Art,post_Art) OF
             Zielart=void : entwerten;
             Ausgangsart=int AND Zielart=real : weiten
           OUT fehler
           ESAC
  ESAC;


FUNCTION Art_in_Typ (Art : Arten) IDENT :
  CASE Art OF
    int: "int"; real: "real"; void: C_leer
  ESAC;


FUNCTION keine_Variablen (liste : Objekte) BOOL :
  CASE Liste OF
    EMPTY: TRUE
    OUT CASE Liste.HEAD.Klasse OF
        Variable: TRUE;
          Prozedur: keine_Variablen(Liste.TAIL )
      ESAC
  ESAC;


FUNCTION Schachtelkomponenten (Liste : Objekte) C_Komponenten :
  CASE Liste OF
    EMPTY: C_leer
    OUT CASE Liste.HEAD.Klasse OF
          Prozedur: Schachtelkomponenten(Liste.TAIL);
        Variable: C_Komponenten
                    ( C_Komponente(Liste.HEAD.Objektbez,
                                   Art_in_Typ(Liste.HEAD.Art)),
                      Schachtelkomponenten(Liste.TAIL ) )

      ESAC
  ESAC;
```

```
FUNCTION Rahmenprogramm ( Startsegment : IDENT ) C_Text :
  C_Text(" SEGMENT (/) main;
           SELECTENVFIELD(/ADDR) main main Dateibez;
           CONST (/COUNT) 0;
           STORESTRING (ADDR COUNT/) "aus";
           SELECTENVFIELD (/ADDR) main main Ausgabe_Datei;
           SELECTENVFIELD (/ADDR) main main Dateibez;
           CONST(/COUNT) 3;
           FILEOPEN (ADDR ADDR COUNT/BOOL) Dateityp;
           CALL (/) ", Startsegment,"
           SELECTENVFIELD (/ADDR) main main Ausgabe_Datei;
           FILECLOSE (ADDR / BOOL) Dateityp;
           DELELTE( BOOL /);
           RETURN (/);
           END main; ");


FUNCTION Adressraum ( Schachteltypen : C_Typen ) C_Text:
  C_Text(" TYPE main =
             RECORD Ausgabedatei = Dateityp,
                    Dateibez = String 3,
                    Text = Zeile
             DYNAMIC 10000 ", Schachteltypen, "
             END;
           TYPE zeile = STRING 132;
           TYPE Dateityp =
             FILE SEQUENTIAL OUT PRINTER
             DYNAMIC Zeile
             END; ");
```

```
FUNCTION Ausgabe C_Text :
  C_Text(" SEGMENT ( REAL /) outreal;
           ENVDEPTH (/) 0;
           SAMEENV (/) main;
           SELECTENVFIELD (/ ADDR ) main main text;
           CONST (/ COUNT ) 0;
           ROTATE ( REAL ADDR COUNT / ADDR COUNT REAL );
           CONST (/ COUNT ) 20;
           CONST (/ COUNT ) 10;
           ROTATE ( REAL COUNT COUNT / COUNT COUNT REAL );
           PUTREAL ( ADDR COUNT COUNT COUNT REAL /);
           CONST (/ COUNT ) 21;
           CONST (/ BOOL ) TRUE;
           LOOP ( COUNT BOOL /) fill;
           SELECTENVFIELD (/ ADDR ) main main Ausgabe_Datei;
           SELECTENVFIELD (/ ADDR ) main main Text;
           WRITESEQ ( ADDR ADDR / BOOL ) Zeile;
           DELELTE ( BOOL /);
           RETURN (/);
           END outreal;

           SEGMENT ( COUNT / COUNT BOOL ) fill;
           ENVDEPTH (/) 0;
           SAMEENV (/) main;
           SELECTENVFIELD (/ ADDR ) main main Text;
           CHANGE (COUNT ADDR / ADDR COUNT );
           STORESTRING ( ADDR COUNT /) " ";
           INCR ( COUNT / COUNT );
           DUPLICATE ( COUNT COUNT / COUNT );
           CONST (/ COUNT ) 133;
           EQUAL(CONT COUNT / BOOL );
           RETURN ( COUNT BOOL /);
           END FILL; ");
```

## Beispiel 2

Mit diesem Beispiel wird die Anwendung des graphentheoretischen Verfahrens zur Bestimmung der Besuchssequenzen (siehe Abschnitt 4.2) anhand einer einfachen Programmiersprache demonstriert. Die statische Semantik wurde hier auf die Identifikation vereinbarter Bezeichner und auf eine einfache Bestimmung und Anpassung von Arten beschränkt. Die Attribute haben folgende Bedeutung:

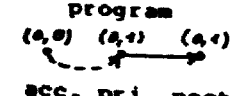
access = Menge der jeweils gültigen Bezeichner,
decl = Vereinbartes Objekt,
primode = Art vor der Artanpassung,
postmode = Art nach der Artanpassung.

Hier sind nur die syntaktischen und semantischen Regeln angegeben. Zu jeder syntaktischen Regel der Grammatik sind die Berechnungsvorschriften für die semantischen Attribute, die C-, D- und E-Graphen und die ermittelte Besuchs-Sequenz angegeben. Die C-,D- und E-Graphen sind jeweils für eine Regel zu einem Graphen zusammen gefasst und werden durch die Kennzeichnung der Kanten unterschieden:

---> Kanten im C-,D- und E-Graph
- -> Kanten im D- und E-Graph
...> Kanten im E-Graph

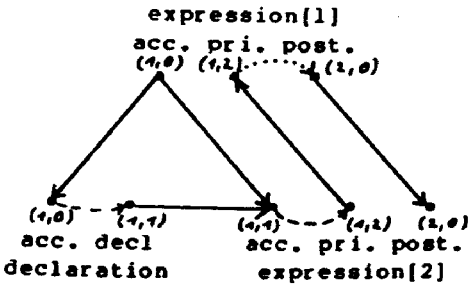
Zu jedem Knoten ist die Bewertung (up,down) des Attributes angegeben.

```
RULE program : expression
SEMANTIC
  expression.access := none,
  expression.postmode := expression.primode
END
```



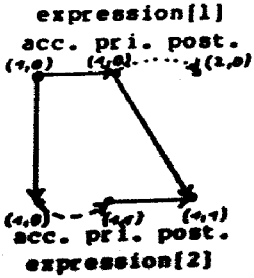

```
P1 = expression.access,B[1,1],expression.postmode,B[1,2]
```

```
RULE expression : "BEGIN" declaration ";" expression "END"
SEMANTIC
  declaration.access := expression[1].access;
  expression[2].access :=
     objects(declaration.decl,expression[1].access);
  expression[1].primode := expression[2].primode;
  expression[2].postmode := expression[1].postmode
END
```



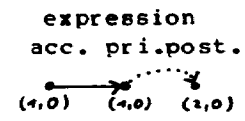

```
F2 = B[0,1],declaration.access,B[1,1],expression[2].access,
     B[2,1],expression[1].primode,B[0,2],expression[2].postmode
```

```
RULE expression : identsymbol ":=" expression
SEMANTIC
  expression[2].access := expression[1].access;
  expression[2].postmode :=
    coerce(expression[2].primode,expression[1].primode);
  expression[1].primode :=
    identsymbol.iden ID expression[1].access
END
```



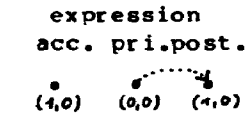

```
F3 = B[0,1],expression[2].access,expression[1].primode,B[1,1],
     expression[2].postmode,B[0,2],B[1,2]
```

```
RULE expression : identsymbol
SEMANTIC
  expression.primode := identsymbol.iden ID expression.access
END
```



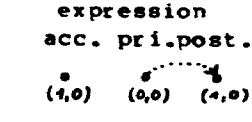

```
P4 = B[0,1],expression.primode,B[0,2]
```

```
RULE expression : intsymbol
SEMANTIC
  expression.primode := integer
END
```



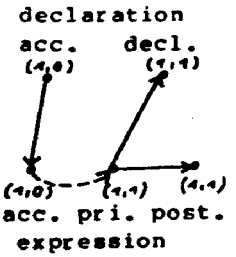

```
P5 = expression.primode,B[0,1],B[0,2]
```

```
RULE expression : realsymbol
SEMANTIC
  expression.primode := real
END
```




```
P6 = expression.primode,B[0,1],B[0,2]
```

```
RULE declaration : "NEW" identsymbol "=" expression
SEMANTIC
  declaration.decl :=
    declare(identsymbol.iden,expression.primode);
  expression.postmode := expression.primode;
     expression.access := declaration.access
END
```




```
P7 = B[0,1],expression.access,B[1,1],declaration.decl,
     expression.postmode,B[1,2]
```

## Literatur


AU72 Aho, A.V., Ullmann, J.D., The theory of parsing, translation, and compiling, Prentice-Hall, Englewood Cliffs, 1972

Bo76 Bochmann, G. V., Semantic evaluation from left to rigth, CACM 19 (1976), Nr.2, S. 55-62

De75 Deussen, P., A decidability criterion for van Wijngaarden grammars, Acta Informatica 5 (1975), S. 353-375

Di63 Dijkstra, E.W., An ALGOL 60 Translator for the X1, in Annual Review in Automatic Programming 3, Pergamon Press, Oxford, 1963, S. 329-345

DR73 DeRemer, F., Transformational grammars for languages and compilers, Technical Report 50, University of Newcastle upon Tyne, 1973

Fe66 Feldman, J., A formal semantics for computer languages and its application in a compiler-compiler CACM 9 (1966), Nr.1, S. 3-9

FG68 Feldman, J., Gries, D., Translator writing systems, CACM 11 (1968), Nr.2, S. 77-113

Ga74 Ganzinger, H., Modifizierte attributierte Grammatiken, Bericht 7420, TU München, 1974

Go69 Goos, G, SMG, Ein Compiler-erzeugendes Programm, Bericht 6905, Rechenzentrum der TH München, 1969

Go75a Goos, G., Die Programmiersprache LEX, Bericht 1/75, Fak. f. Informatik, Universität Karlsruhe, 1975

Go75b Goos, G., Die Programmiersprache BALG, vorläufige Fassung, Bericht 6/75, Fak. f. Informatik, Universität Karlsruhe, 1975

Gr71b Gries, D., Compiler construction for digital computers, Wiley, New York, 1971

HL74 Hoare, C.A.R., Lauer, P.E., Consistent and complementary formal theories of the semantics of programming languages, Acta Inforatica 3 (1974), S. 135-153

Ho69 Hoare, C.A.R., An aximatic basis for computer programming, CACM 12 (1969), Nr. 10, S. 567-581

HW73 Hoare, C.A.R., Wirth, N., An axiomatic definition of the programming language PASCAL, Acta Informatica 2 (1973), S. 335-355

JOR75 Jazayeri, M., Ogden, W.F., Rounds, W.C., The intrinsically exponential complexity of the circularity problem for attribute grammars, CACM 18 (1975), Nr.12, S. 697-706

Ka75 Kastens, U., Systematische Analyse semantischer Abhängigkeiten, Bericht 12/75, Fak. f. Informatik, Universität Karlsruhe, 1975

Ka76 Kastens, U., Systematische Analyse semantischer Abhängigkeiten, in Informatik Fachberichte 1, Springer-Verlag, 1976, S. 19-32

Kn68 Knuth, D.E., Semantics of context-free languages, in Math. Syst. Th. 2 (1968), S. 127-145, Korrekturen in Math. Syst. Th. 5 (1971), S. 95

Kn71 Knuth, D.E., Examples of formal semantics, in Lecture Notes in Mathematics 168, Springer-Verlag, 1971

Ko71a Koster, C.H.A., Affix grammars, in Peck, J.E.L. (Ed.), ALGOL 68 Implementation, North-Holland Publ. Comp., 1971, S. 95-109

Ko71b Koster, C.H.A., A compiler-compiler, Report MR127, Mathematisch Centrum Amsterdam, 1971

KHW70 McKeeman, W.M., Horning, J.J., Worthman, D.B., A compiler generator, Prentice-Hall Inc., Englewood Cliffs, N. J., 1970

La71 LaLonde, W.R., An efficient LALR parser generator, Technical Report 2, Computer Systems Research Group, University of Toronto, 1971

Le75 Lewi, J., DeVlaminck, K., Huens, J., Mertens, P., SLS/1: A translator writing system, in Lecture Notes in Computer Science 5, Springer-Verlag, 1975, S.627-653

Na60 Naur, P. (Ed.), Report on the algorithmic language ALGOL 60, Numer. Math. 2 (1960), S. 106-136

No75 Nori, K.V. et al., The PASCAL-P-Code compiler: Implementation notes, ETH Zürich, Berichte des Inst. f. Informatik, Nr.10, 1975

NA74 Neel, D., Armirchahy, M., Semantic attributes and improvement of generated code, in Proc. ACM Nat. Comp. Conf., San Diego, 1974, S. 1-10

NA75 Neel, D., Armirchahy, M., Removal of invariants from nested-loops in a single effective compiler pass, in SIGPLAN Notices 10 (1975), Nr.3, S. 87-96

Po76 Polak, W., Axiomatische Definition von LEX, Diplomarbeit, Fak. F. Informatik, Universität Karlsruhe, 1976

Ri69 Richards, M., BCPL - A tool for compiler writing and system programming, Proc. Spring Joint Comp. Conf., 1969

Ro64 Rosen, S., A compiler-building system developed by Brooker and Morris, CACM 7 (1964), Nr.7, S. 403-414

Sch76 Schreiber, P.P., Baum-Transduktoren, Dissertation, Fachbereich 20, TU Berlin, 1976

St61 Steel, T.B., A first version of UNCOL, Proc. Western. Joint Comp. Conf., 1961, S.371-378

Ul74 Ulukut, S., Implementierung eines LALR(1)-Analysators, Diplomarbeit, Fak. f. Informatik, Universität Karlsruhe, 1974

vW68 v. Wijngaarden, A. (Ed.), Report on the algorithmic language ALGOL 68, Numer. Math. 14 (1969), S. 79-218

vW75 v. Wijngaarden, A. (Ed.), Revised report on the algorithmic language ALGOL 68, Acta Informatica 5 (1975), S. 1-236

Wa70 Waite, W.M., The mobile programming system: STAGE2, CACM 13 (1970), Nr.7, S 415-421

We72 Wegner, P., The Vienna Definition Language, Computing Surveys 4 (1972), Nr.1, S. 5-63

Wi71 Wilner, W.T., Declarative semantic definition as illustrated by a definition of SIMULA 67, Ph.D. Thesis, Stanford University, Computer Science Dep., 1971

Wi74 Wilner, W.T., Structured programs, arcadian machines, and the Burroughs B1700, in Lecture Notes in Computer Science 7, Springer-Verlag, 1974, S. 133-148

WH75 Waite, W.M., Haddon, B.K., A preliminary definition of JANUS, Report SEG-75-1,nc., Dep. of Electr. Engeneering, University of Colorado, Boulder, 1975

## Lebenslauf

| | |
|---|---|
| 21.12.1946 | geboren in Bremen als Sohn des Bundesbahnbeamten Willfried Kastens und der Sekretärin Gerda Kastens, geb. Hünerberg |
| 1953 - 1957 | Besuch der Volksschulen in Hude/Oldenburg und Bremen |
| 1957 - 1966 | Besuch des Gymnasiums an der Hermann-Böse-Str. in Bremen |
| 19.2.1966 | Reifeprüfung |
| 1966 - 1968 | zweijährige Dienstzeit bei der Bundeswehr |
| 1968 | halbjähriges Hochschulpraktikum für Nachrichtentechnik bei der Firma Siemens in Bremen |
| 1968 - 1970 | Studium der Elektrotechnik an der Technischen Hochschule Darmstadt |
| 18.12.1970 | Diplom-Vorprüfung in der Fachrichtung Informatik |
| 1970 - 1973 | Studium der Informatik an der Universität (TH) Karlsruhe |
| 29.3.1973 | Diplom-Hauptprüfung |
| 1973 - 1975 | Graduiertenstipendiat am Institut II der Fakultät für Informatik der Universität (TH) Karlsruhe |
| 1975 - heute | wissenschaftlicher Angestellter am gleichen Institut |

SIGNATUR: UA145968

<14+>OS61454559598