Handbuch der Informatik

herausgegeben von
Prof. Dr. Albert Endres
Prof. Dr. Hermann Krallmann
Dr. Peter Schnupp

Band 3.3

Übersetzerbau

von
Prof. Dr. Uwe Kastens
Universität-Gesamthochschule Paderborn

R. Oldenbourg Verlag München Wien 1990

Prof. Dr. Uwe Kastens
Studium der Informatik an der TH Darmstadt und der Universität Karlsruhe. 1976 Promotion in Karlsruhe. Seit 1982 Professor für Praktische Informatik an der Universität-GH Paderborn. Forschungsgebiet Programmiersprachen und Übersetzer, Entwicklung von Übersetzungsmethoden und -werkzeugen.

Anschrift:
Universität-Gesamthochschule Paderborn
Fachbereich 17
Postfach 1621
4790 Paderborn

CIP-Titelaufnahme der Deutschen Bibliothek

Handbuch der Informatik : [die umfassende Darstellung der Informatik in Einzelbänden] / hrsg. von Albert Endres ... – München ; Wien : Oldenbourg.

NE: Endres, Albert [Hrsg.]

Bd. 3.3. Kastens, Uwe: Übersetzerbau. – 1990

Kastens, Uwe:
Übersetzerbau / von Uwe Kastens. – München ; Wien : Oldenbourg, 1990
(Handbuch der Informatik ; Bd. 3.3)
ISBN 3-486-20780-6

Lasersatz: Jens Peter Oldenbourg, München
Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-20780-6

Inhalt

Vorwort der Herausgeber

Das *Handbuch der Informatik* versucht, das Gesamtgebiet der Informatik mit seinen Grundlagen, seinen Teilgebieten und seinen wichtigsten Anwendungen zusammenhängend darzustellen. Es informiert Lehrende und Lernende, DV-Praktiker und DV-Nutzer über Konzepte, Methoden und Techniken, deren grundlegende Bedeutung anerkannt ist und deren Nützlichkeit sich in der Praxis gezeigt hat. Grenzen und Entwicklungstendenzen eines Gebiets werden angesprochen.

Jeder Band des *Handbuchs der Informatik* behandelt ein in sich abgeschlossenes Thema. Der Leser kann sich – auch unabhängig von den anderen Bänden des Handbuchs – in das betreffende Gebiet neu einarbeiten, vorhandenes Wissen auffrischen oder im Sinne eines Nachschlagewerks einzelne Themen aufspüren und vertiefen. Hierbei helfen die Strukturierung und Typographie des Textes und die Hinweise auf weiterführende Literatur.

Der vorliegende Band behandelt die Implementierung von Programmiersprachen, auch Übersetzerbau genannt. Er sollte in Zusammenhang gesehen werden mit mehreren anderen Bänden, in denen bestimmte Klassen von Programmiersprachen aus der Sicht des Benutzers vorgestellt werden.

Der Übersetzerbau ist eines der klassischen Wissensgebiete der Informatik und kann auf eine relativ lange Entwicklung zurückblicken. Deshalb sind die hier zur Anwendung kommenden Methoden stärker theoretisch untermauert, als dies in anderen Gebieten der Fall ist. Dennoch sind Übersetzer komplexe Software-Produkte und Übersetzerbauer sind gut beraten, sich an Software-Engineering-Grundsätze zu halten. Die lange gehegte Hoffnung, daß man Übersetzer eines Tages vollautomatisch erzeugen könnte, ist bescheideneren Zielen gewichen. Andererseits ist zu bemerken, daß viele der für den Übersetzerbau entwickelten Techniken in anderen Bereichen der Informatik Eingang gefunden haben.

In einem einzelnen Band wird hier in sehr prägnanter und systematischer Weise das gesamte Gebiet des Übersetzerbaus dargestellt, angefangen von der syntaktischen Analyse, über die semantische Analyse bis zur Code-Generierung und Optimierung. Ein besonderer Schwerpunkt liegt auf Werkzeugen zur automatischen Erstellung von Teilen eines Übersetzers, ein Gebiet, auf dem der Autor eigene wissenschaftliche Beiträge geleistet hat. Die Auswertung attributierter Grammatiken hat sich dabei als eine Technik erwiesen, die für viele verschiedene Aufgaben zur Anwendung kommt.

Durch den klaren Aufbau, die sorgfältige Begriffsbildung und die gute Abwägung zwischen theoretischen und praktischen Aspekten ist hier im wahrsten Sinne des Wortes ein Handbuch entstanden, das dem Leser als verläßlicher und leicht verwendbarer Ratgeber dienen kann.

A. Endres H. Krallmann P. Schnupp

Vorwort des Autors

Programmiersprachen sind die Ausdrucksmittel der Software-Entwicklung. Um Programme auf Rechnern auszuführen, werden sie in deren Maschinensprache übersetzt. Dies leisten Übersetzer, die selbst komplexe Programmsysteme sind. Zusammen mit ergänzenden Werkzeugen zur Programmentwicklung macht ein Übersetzer eine Programmiersprache auf einem Rechner praktisch einsetzbar. Übersetzer gehören deshalb zur Grundsoftware von Rechnern.

Zur Konstruktion von Übersetzern werden Verfahren und Algorithmen angewandt, die ihre Grundlage in wohlverstandenen theoretischen Modellen haben. Hierauf basieren systematische Techniken und der Einsatz von Werkzeugen im Übersetzerbau. Übersetzer sind komplex strukturierte Programmsysteme, an die hohe Qualitäts- und Leistungsanforderungen gestellt werden. Zu ihrer Herstellung müssen deshalb Methoden des Software-Engineering konsequent angewandt werden.

Jeder Informatiker benutzt Übersetzer in der Programmentwicklung. Kenntnisse im Übersetzerbau sind nützlich für den Umgang mit Programmiersprachen und vertiefen das Verständnis für Spracheigenschaften. Nur wenige Informatiker sind jemals tatsächlich an der Entwicklung eines Übersetzers beteiligt. Trotzdem liefert der Übersetzerbau wertvolles Wissen und Methoden von allgemeiner Bedeutung: In vielen Bereichen außerhalb des Übersetzerbaus können Probleme mit Übersetzerverfahren systematisch gelöst werden, wenn man sie als Übersetzungsprobleme erkennt. Beispiele dafür reichen von der Umsetzung spezieller Eingabesprachen für Anwendungssysteme über die Aufbereitung strukturierter Daten bis zur Verarbeitung von Protokollen oder Signalfolgen.

Das Studium des Übersetzerbaus liefert wichtige methodische Erkenntnisse: Hier wird an einem gut verstandenen Problembereich auf der Basis langjähriger Informatikforschung demonstriert, wie man praktikable Techniken konsequent aus theoretisch fundierten Modellen ableitet und in komplexen Software-Systemen systematisch anwendet. Der Übersetzerbau liefert zahlreiche Beispiele für die Übertragung von Lösungsverfahren aus Gebieten der Theorie oder anderen Anwendungsgebieten, wie Graphfärbung oder Seitentauschverfahren zur Registerzuteilung, Mustererkennung zur Code-Erzeugung oder Anordnungsverfahren zur Code-Optimierung.

Prinzipien der Software-Entwicklung werden anhand der Übersetzerkonstruktion exemplarisch gezeigt: Um die Komplexität des Problems zu bewältigen, muß es konsequent in Teilaufgaben zerlegt werden. Es reicht nicht aus, diese durch geeignete Algorithmen zu lösen. Erst durch eine saubere modulare Gliederung mit klaren Schnitt-

stellen entsteht ein brauchbares Produkt aus wartbaren und wiederverwendbaren Komponenten. Auch der Einsatz von Werkzeugen wird in kaum einem anderen Bereich so konsequent und vielfältig demonstriert.

Der Übersetzerbau ist eines der ältesten Forschungsgebiete der Informatik, das sich sehr lebendig fortentwickelt. Insbesondere neue Prozessorarchitekturen, wie RISC-Prozessoren und verschiedene Parallelisierungstechniken, stellen Anforderungen an Übersetzer, die mit neuartigen Verfahren gelöst werden müssen. Ebenso ist die weitere Entwicklung generierender Werkzeuge Gegenstand aktueller Forschung.

Mit diesem Buch soll sich der Leser in das Gebiet des Übersetzerbaus einarbeiten können. Er lernt die Aufgabenstellung, Probleme und systematische Lösungsverfahren kennen, um sie beurteilen und anwenden zu können. Dabei werden Grundkenntnisse der Informatik, insbesondere zu Programmiersprachen, formalen Methoden und zur Software-Entwicklung vorausgesetzt. Das Buch wendet sich an Lernende, Lehrende und Praktizierende in der Informatik und in Gebieten, die Informatikmethoden anwenden. Es kann als Begleittext zu Vorlesungen eingesetzt, zum Nachschlagen von Verfahren, Algorithmen und Werkzeugen verwendet, oder als Anleitung zur Konstruktion von Übersetzern oder Übersetzermodulen in anderen Programmsystemen benutzt werden.

In dieses Buch ist die Erfahrung aus der Entwicklung von Übersetzern und Werkzeugen und aus Vorlesungen und Praktika an der Universität-GH Paderborn eingeflossen. Die Mitarbeiter meiner Arbeitsgruppe haben dazu wesentlich beigetragen. Die Kooperation mit Prof. W. M. Waite zu Übersetzerwerkzeugen und Diskussionen mit Prof. G. Goos haben wichtige Anregungen zu diesem Buch gegeben. Frau Birgit Farr hat mit perfektem Einsatz von Werkzeugen die Texte und Abbildungen erstellt und unermüdlich überarbeitet. Schließlich wäre die Arbeit an diesem Buch ohne die Geduld und das Verständnis meiner Familie nicht möglich gewesen. Ihnen allen danke ich für ihre Beiträge.

Paderborn, im April 1990

Uwe Kastens

1. Einführung

Übersetzung

Die Aufgabe der Übersetzung von Texten einer Sprache in eine andere ist für natürliche, gesprochene und geschriebene Sprachen allgemein bekannt. In der Informatik wird der Begriff Übersetzer im gleichen Sinne für Programmsysteme verwendet, die Sprachen für die Programmierung von Rechnern übersetzen. Auch die Aufgaben sind grundsätzlich vergleichbar: Struktur und Bedeutung der Sätze werden ermittelt und ohne Veränderung der Bedeutung in die Zielsprache transformiert. Die Verfahren zur Lösung der Aufgaben werden jedoch durch Eigenschaften geprägt, die Programmiersprachen von natürlichen Sprachen unterscheiden: Sie sind durch strikte, formale Regeln definiert, aus denen systematische Übersetzungsalgorithmen abgeleitet werden können. Da insbesondere bei der Übersetzung in Maschinensprachen ein großer Unterschied im Sprachniveau überbrückt wird, ist die Herstellung des Zielprogramms eine besonders komplexe Aufgabe. Programme werden übersetzt, um sie auszuführen. Sie müssen deshalb nicht nur korrekt übersetzt werden, sondern sollen auch möglichst schnell und speichergünstig ausführbar sein. Der Übersetzer muß dazu Optimierungsaufgaben lösen.

Historie

Der Übersetzerbau ist einer der ältesten Wissenschaftsbereiche der Informatik. Erste Publikationen zu systematischen Übersetzungsverfahren stammen schon aus den frühen 50er Jahren. Die Entwicklung des ersten Fortran-Übersetzers hat lange gültige Maßstäbe insbesondere in der Code-Optimierung gesetzt. Weitere frühe Meilensteine sind die Implementierungen von Algol 60 und Algol 68. Einen ausführlichen historischen Überblick findet man in dem Artikel von F. L. Bauer in BAUE76. Als Ergebnis dieser für die Informatik langen Entwicklungszeit des Übersetzerbaus sind heute für viele klassische Übersetzeraufgaben systematische Lösungen und Algorithmen bekannt; für einige sind Werkzeuge zur automatischen Generierung von Übersetzerteilen im Einsatz. Neue Konzepte in Programmiersprachen und Eigenschaften neuer Rechnerarchitekturen erfordern, daß Übersetzertechniken entwickelt werden.

Warum ist es heute für Informatiker wichtig, Kenntnisse im Übersetzerbau zu beherrschen?

Anwendungen

- **Übersetzer für allgemein anwendbare Programmiersprachen gehören (zusammen mit Betriebssystemen und Datenbanken) zur Grundsoftware von Rechnern, deren Aufbau und Arbeitsweise jeder Informatiker verstehen sollte.**
- **Für die Implementierung von Anwendungssystemen mit sprachlicher Benutzeroberfläche (z. B. Kommandosprachen, Datenbanksprachen) sind Übersetzertechniken notwendig.**

- Häufig können Lösungen gleichartiger Probleme (z. B. Entwicklung von Programmen aus parametrisierten Grundbausteinen) durch Definition von Spezifikationssprachen und deren Implementierung mit Übersetzermethoden systematisiert und vereinfacht werden.
- Übersetzungsmethoden werden auch außerhalb des Übersetzerbaus zweckmäßig eingesetzt, z. B. Automaten zur Verarbeitung von Informationsströmen, Codierung von Bezeichnern, Zuordnung von Eigenschaften und Transformation von Datenstrukturen.

Methodik

Auch dort, wo Kenntnisse und Techniken des Übersetzerbaus nicht unmittelbar angewandt werden, kann man aus dem exemplarischen Studium der Übersetzerkonstruktion wichtige methodische Erkenntnisse gewinnen. Es werden hier allgemeine Prinzipien zur Lösung einer sehr komplexen Aufgabe systematisch angewandt:

- konsequente, an den Aufgaben orientierte Strukturierung,
- algorithmische Lösungen auf der Basis theoretischer Modelle,
- Einsatz von Werkzeugen und
- Adaptierung von standardisierten Verfahren.

Grob vereinfacht gehört zur Entwicklung eines Übersetzers das Lösen von Teilaufgaben mit formalen Methoden und systematischen Algorithmen sowie das Zusammensetzen der Teillösungen nach Methoden des Software-Engineerings zur Modularisierung und Schnittstellendefinition. Bei geeigneter Wahl der meist bekannten Algorithmen werden die Qualitäten des Produktes "Übersetzer" wesentlich durch seine Strukturierung und Schnittstellen bestimmt. Deshalb kommt den Engineering-Aspekten im Übersetzerbau ein hoher Stellenwert zu.

Berührte Wissensgebiete

Im Übersetzerbau wird ein breites Spektrum von Wissensgebieten der Informatik berührt:

- *Programmiersprachen*: Die Sprachdefinition ist wesentlicher Teil der Aufgabenspezifikation eines Übersetzers. Sprachkonzepte und -elemente müssen vollständig verstanden sein, um sie korrekt zu implementieren.
- *Rechnerarchitektur*: Für die Abbildung auf die Zielmaschine sind Kenntnisse ihrer Eigenschaften (Instruktionssatz, Speicherstrukturen, Registerstrukturen) erforderlich.
- *Theoretische Grundlagen*: Lösungen von Übersetzeraufgaben basieren auf Methoden und Algorithmen aus der Theorie formaler Sprachen, Automaten und Graphen.
- *Software-Engineering*: Ein Übersetzer ist ein komplexes Software-Produkt, dessen Qualitäten durch systematische Konstruktionsmethoden bestimmt werden.

Wir geben in diesem Buch eine Übersicht zu Methoden und Techniken für die Konstruktion von Übersetzern, die höhere Programmiersprachen in Maschinenprogramme übersetzen. Dies ist die umfassendste Aufgabenstellung. Bei der Konstruktion von Übersetzern für niedere Sprachen, Anwendersprachen oder für abstrakte Zielmaschinen vereinfacht sich die Lösung einiger Teilaufgaben oder sie entfallen ganz.

Übersicht

In den folgenden Abschnitten führen wir in die Aufgabenstellung für Übersetzer ein. Nach einer allgemeinen Betrachtung der Übersetzungsaufgabe in Abschnitt 1.1 stellen wir Eigenschaften von Quellsprachen (Abschnitt 1.2) und von Zielmaschinen (Abschnitt 1.3) aus der Sicht der Übersetzerkonstruktion vor.

Das Kapitel 2 ist der Übersetzerstruktur gewidmet. Wir leiten sie aus den Übersetzeraufgaben her. Hier sind die Software-Engineering-Aspekte der Übersetzerkonstruktion zusammengefaßt. In den Kapiteln 3 bis 6 behandeln wir Verfahren zur Lösung der Analyseaufgaben im Übersetzer. Zur lexikalischen, syntaktischen und semantischen Analyse stellen wir jeweils die wichtigsten systematischen Verfahren mit ihrer theoretischen Fundierung und praktischen Umsetzung vor. Wegen der Komplexität der Aufgabe der semantischen Analyse haben wir sie auf die Darstellung der Attributierungsmethode in Kapitel 5 und ihrer Anwendung in Kapitel 6 aufgeteilt. Die Aufgaben zur Erzeugung der Zielprogramme werden in Kapitel 7 behandelt. Hier nehmen im Vergleich zu den Analysekapiteln Aspekte des Entwurfs einen größeren Raum ein als die Algorithmen zur Transformation. Kapitel 8 stellt Techniken und Algorithmen zur Verbesserung der erzeugten Zielprogramme vor, die insbesondere für die Konstruktion leistungsfähiger, optimierender Übersetzer wichtig und notwendig sind.

In den Kapiteln 2 bis 7 ist jeweils ein Abschnitt dem Einsatz von Werkzeugen gewidmet, die nach den vorher beschriebenen Verfahren Übersetzermodule generieren. In den Einleitungen aller Kapitel weisen wir kurz darauf hin, wie die besprochenen Verfahren auch für Aufgaben am Rande oder außerhalb der Konstruktion von Übersetzern angewandt werden können.

1.1 Von der Quellsprache zur Zielmaschine

Übersetzung

Die zentrale Aufgabe eines Übersetzers ist es, alle Sätze einer *Quellsprache*, die Quellprogramme, in *gleichbedeutende* Sätze einer *Zielsprache*, die Ziel- oder Maschinenprogramme, zu transformieren. Diese Aufgabe wird durch die Definition der Quellsprache einerseits und der Zielsprache bzw. Zielmaschine andererseits spezifiziert.

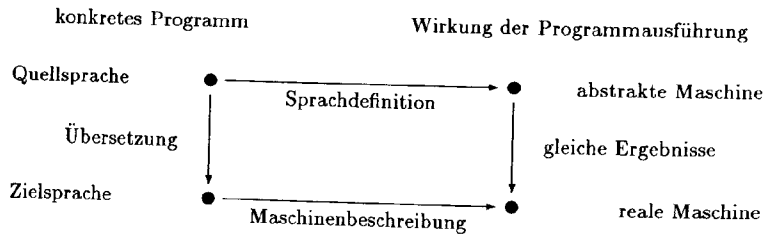


Abb. 1.1-1: Übersetzung von Quellprogrammen in gleichbedeutende Zielprogramme.

Eine Sprachdefinition legt fest, unter welchen Bedingungen eine Zeichenfolge ein korrektes Programm ist und welche Ergebnisse die Ausführung eines Programms mit bestimmten Daten liefert. Dies gilt im Prinzip für Quell- und Zielsprache gleichermaßen. In dem für den Übersetzerbau wichtigsten Fall haben wir eine Quellsprache auf dem Niveau einer höheren Programmiersprache und eine Maschinensprache als Zielsprache. Der Übersetzer muß dann zu jedem korrekten Programm der Quellsprache ein Maschinenprogramm erzeugen, das bei Ausführung auf der Zielmaschine die Ergebnisse produziert, die in der Quellsprachdefinition spezifiziert sind (Abb. 1.1-1). Eine Sprachdefinition besteht aus Regeln, die die Menge der korrekten Programme festlegen, und einer Beschreibung der Bedeutung von Sprachelementen in Termen einer *abstrakten Sprachmaschine*.

Korrektes Quellprogramm

Der Übersetzer muß feststellen, ob das Quellprogramm korrekt im Sinne der Sprachdefinition ist. Wir unterscheiden Bedingungen, die anhand des Programmtextes zur *Übersetzungszeit* prüfbar sind, und solche, die zur *Laufzeit* bei der Ausführung des Programms mit bestimmten Eingabedaten geprüft werden. Erfüllt ein Programm die ersteren Bedingungen, so ist es *übersetzbar*; sind auch die Laufzeitbedingungen erfüllt, dann ist es außerdem *ausführbar*. Unter dem Begriff *Fehlerbehandlung* fassen wir alle Aktionen zusammen, mit denen der Übersetzer die Übersetzbarkeit überprüft, und die notwendig sind, um fehlerhafte Programme weiterzuverarbeiten. Fehler müssen erkannt und dem Benutzer in geeigneter Form gemeldet werden. Soll der Übersetzer nicht nach Erkennen des ersten Fehlers anhalten, so müssen seine internen Datenstrukturen so instandgesetzt werden, daß keine systembedingten Fehler auftreten. Die Minimalanforderungen an die Fehlerbehandlung sind:

- Kein korrektes Programm darf eine Fehlermeldung verursachen,
- zu jedem fehlerhaften Programm muß mindestens eine Fehlermeldung produziert werden.

Weitergehende Kriterien bestimmen die Qualität der Fehlerbehandlung. Verletzungen der Sprachdefinition, die erst bei der Ausführung des Programms zur Laufzeit erkannt werden können (z. B. Indizierung von Reihungen außerhalb der definierten Grenzen) werden durch Erzeugen von *Laufzeitprüfungen* im Zielprogramm behandelt.

Übersetzerentwurf

Aus mehreren Gründen der Übersetzerstrukturierung (s. Kap. 2) ist es zweckmäßig, die Übersetzung in mindestens zwei Schritte aufzuteilen: einen Analyseteil, der die Übersetzbarkeit des Quellprogramms prüft und das Programm in eine *Zwischensprache* übersetzt, und einen Syntheseteil, der ein Zwischensprachprogramm in die Zielsprache übersetzt. Die Zwischensprache wird meist für einen Übersetzer oder eine Übersetzerfamilie entworfen. Sie sollte so beschaffen sein, daß ihre Erzeugung im ersten Schritt einfach ist, weil sie nahe an der Quellsprache liegt, und keine Analyse im zweiten Schritt nötig ist.

Die Eigenschaften der Quellsprache und der Zielmaschine spezifizieren die vom Übersetzer zu leistende Transformation. Wichtigste Vorarbeit für den Entwurf eines Übersetzers ist deshalb die sorgfältige Analyse der Sprachdefinition und der Maschinenbeschreibung. Dabei müssen auch Detailfragen vollständig und präzise geklärt werden. In den beiden folgenden Abschnitten zeigen wir, nach welchen Kriterien die Eigenschaften analysiert werden, um daraus die Übersetzeraufgaben zu entwickeln.

Im Sinne des Software-Engineering sind Quellsprache und Zielmaschine der Kern der Spezifikation für den zu erstellenden Übersetzer. Hinzu kommen Anforderungen, die sich durch seinen intendierten Einsatz ergeben: Die Benutzung der Sprache in der Ausbildung stellt andere Anforderungen als die Übersetzung von Programmen, die langfristig und häufig benutzt werden. Außerdem darf der Übersetzer nicht isoliert entwickelt werden. Er ist in eine Umgebung eingebettet, zu der weitere Werkzeuge zur Programmentwicklung, wie Editoren, Testhilfen, und das Betriebssystem, gehören. Diese zusätzlichen Randbedingungen beeinflussen Struktur und Implementierung des Übersetzers. Wir gehen darauf am Ende von Kapitel 2 näher ein.

1.2 Eigenschaften der Quellsprache

Definition der Quellsprache

Die Definition der Quellsprache ist die Grundlage der Aufgaben des Analyseteils im Übersetzer. Wir gehen hier davon aus, daß dem Übersetzerbauer eine solche Definition vorliegt. Dies ist für verbreitete Programmiersprachen meist der Fall. Wird die Quellsprache erst im Zuge der Übersetzerentwicklung entworfen (z. B. spezielle Anwendersprache), so ist es dringend angeraten, frühzeitig ein definierendes Dokument zu erstellen. Es wird ohnehin benötigt, denn die Auffassung

"der Übersetzer definiert die Sprache" ist für seine Benutzung und spätere Wartung nicht akzeptabel.

Sprachdefinition

Häufig gibt es neben einem definierenden Dokument (Reference Manual) auch eine Benutzungsbeschreibung (User Manual) oder Erläuterungen zur zweckmäßigen Verwendung von Sprachelementen (Rationale). Für den Übersetzerbauer ist die Sprachdefinition entscheidend. Weitere Dokumente werden herangezogen, um die Intension des Sprachentwerfers zu erhellen.

Die Spracheigenschaften sollten unter folgenden Aspekten sorgfältig analysiert werden:

- Zusammenfassung von Eigenschaften, die zu einer Teilaufgabe des Übersetzers gehören, und Abgrenzung von anderen (z. B. Typprüfung),
- Feststellen von Abhängigkeiten zwischen den Teilaufgaben,
- Erkennen von Standard-Sprachkonzepten, für die es Standard-Lösungen gibt (z. B. Gültigkeitsregeln in blockstrukturierten Sprachen),
- Erkennen von lückenhaften, unklaren oder widersprüchlichen Regeln,
- Erkennen von Spracheigenschaften, deren Implementierung schwierig oder so nicht möglich ist (z. B. Programmeigenschaften, die erst zur Laufzeit ermittelt werden können, aber zur Übersetzungszeit benötigt werden).

Die genannten Probleme treten regelmäßig bei neuen Sprachen auf, die noch nicht oder nur vom Sprachentwerfer selbst implementiert sind.

Sprachdefinitionen sind im allgemeinen in zwei zueinander orthogonalen Richtungen gegliedert: nach Teilbereichen der Sprache, wie Definitionen, Anweisungen, Ausdrücken, und nach Ebenen der Beschreibung von Sprachelementen: Grundsymbole, syntaktische Struktur, Kontextabhängigkeiten und Bedeutung. In der Gliederung des Textes ist meist das zweite Kriterium dem ersten untergeordnet. Für den Übersetzerbauer ist die Einordnung nach den Beschreibungsebenen ausschlaggebend, da sie unmittelbar zu einer an den Aufgaben orientierten Strukturierung des Übersetzers führt.

1.2.1 Notation der Grundsymbole

Grundsymbole

Der Programmtext ist eine Folge von Zeichen eines Zeichensatzes (z. B. ASCII). Darin werden Teilfolgen zu Grundsymbolen zusammengefaßt: Bezeichner, Wortsymbole, Literale für numerische Werte, für Zeichenreihen und Einzelzeichen sowie Spezialzeichen (wie ; und :=). Die Schreibweise von Bezeichnern und Literalen wird meist durch ein Regelwerk formal definiert, z. B. durch Syntaxdiagramme oder reguläre Ausdrücke. Neben der Notation der Grundsymbole selbst wird ihre Abgrenzung durch Zwischenräume, Zeilenwechsel, Tabulatoren oder Kommentare sowie deren Schreibweise festgelegt.

Nur in wenigen Sprachen, wie Algol 60, werden Wortsymbole durch ein Fluchtsymbol oder durch Berandung mit ' gekennzeichnet. Meist sind sie als Buchstabenfolgen definiert, die nicht als Bezeichner in Programmen verwendet werden dürfen. Die lexikalische Analyse der Symbole wird erheblich erschwert, wenn (wie in PL/1) die gleiche Zeichenfolge abhängig vom Kontext als Bezeichner oder Wortsymbol zu interpretieren ist. Es ist weiter wichtig, herauszufinden, ob bestimmte Symbole für vordefinierte Größen (wie integer, real, read, write) als Wortsymbole oder Bezeichner definiert sind. Nur im zweiten Fall könnten sie in einem Programm redefiniert werden.

Wortsymbole

Die Definition der Grundsymbole spezifiziert die Übersetzeraufgabe der lexikalischen Analyse (Kapitel 3).

1.2.2 Syntaktische Struktur

Die syntaktische Struktur von Programmen wird seit der Definition von Algol 60 in NAUR63 durch kontextfreie Grammatiken definiert. Die Regeln der Grammatik führen Namen für strukturelle Einheiten ein (z. B. *Anweisung*) und geben an, wie diese aus Teilstrukturen und Grundsymbolen zusammengesetzt werden, z. B.:

Kontextfreie Grammatik

```
Block      ::=  'begin' {Deklaration} {Anweisung} 'end'
Anweisung  ::=  Zuweisung | Schleife | Block
Zuweisung  ::=  Variable ':=' Ausdruck ';'
Schleife   ::=  'while' Ausdruck 'do' Anweisung
```

Als Notation für die Grammatik wird meist eine *erweiterte Backus-Naur-Form* (*EBNF*) verwendet, die auch durch eine Darstellung in Form von Syntaxdiagrammen veranschaulicht werden kann.

Die *syntaktische Analyse* des Übersetzers (Kapitel 4) prüft, ob ein gegebenes Programm im Sinne der Grammatik syntaktisch korrekt ist. Die Lösung dieser Aufgabe wird systematisch aus einer kontextfreien Grammatik entwickelt, die man durch geeignete Umformungen aus der Grammatik der Sprachdefinition herstellt: Mehrdeutigkeiten, die nur durch Kontextanalyse entscheidbar sind, werden entfernt; weitergehende Restriktionen des Analyseverfahrens müssen erfüllt werden; Regeln, die nur zur Einführung von Benennungen für Strukturen (z. B. *Zuweisung*, *Schleife*) dienen, können gestrichen werden.

Die syntaktische Analyse ermittelt auch die abstrakte Struktur eines Programmes, damit den Strukturelementen (z. B. *Block*, *Ausdruck*) die unten besprochenen Eigenschaften (z. B. Typen) zugeordnet werden können. Für den Übersetzer spezifiziert man diese *abstrakte Syntax* ebenfalls durch eine Grammatik, die als Vergröberung aus der *konkreten Syntax* für die Notation der Programme entwickelt wird.

Abstrakte Struktur

1.2.3 Kontextabhängigkeiten (statische Semantik)

Kontextabhängige Eigenschaften

Höhere Programmiersprachen sind im formalen Sinne keine kontextfreien Sprachen. Die kontextfreie Grammatik definiert eine Obermenge der übersetzbaren Programme. Den Sprachelementen werden Eigenschaften zugeordnet, die durch den Kontext im Programm bestimmt werden, z. B. der Typ eines Ausdrucks. Korrekte Programme müssen Einschränkungen bezüglich solcher Eigenschaften genügen. Diese Regeln werden meist in Sprachdefinitionen verbal mit Bezug auf die Syntax beschrieben. Die für den Übersetzerbauer wünschenswerte Präzision wird durch Formalisierung solcher Beschreibungen erzielt, z. B. Definition und systematische Verwendung des Begriffs Typgleichheit.

Allerdings verliert die Sprachdefinition mit zunehmender Formalisierung an Verständlichkeit für den Programmierer. Ein Beispiel dafür ist die Definition von Algol 68 in WIJN75. (Dieser Mangel sollte dann durch ein zusätzliches erläuterndes Dokument ausgeglichen werden.)

Es gibt verschiedene Kalküle zur formalen Definition kontextabhängiger Spracheigenschaften:

- Attributierte Grammatik: Sie erweitert die kontextfreie Grammatik um Attribute und Attributregeln, siehe Kapitel 5.
- Denotationale Semantik: Sie ordnet den Konstrukten der abstrakten Syntax Funktionen zur Berechnung ihrer Eigenschaften zu (z. B. GORD79, TENN76).
- Van Wijngaarden Grammatik: ein formales System zur Erzeugung kontextfreier Grammatiken (WIJN75).
- Vienna Definition Language: Die Ausführung des Programmes wird durch Transformation seiner abstrakten Struktur beschrieben (LUCA69, WEGN72).
- Axiomatische Semantik: Axiome über Strukturelementen beschreiben deren Bedeutung (HOAR73).

Semantische Analyse

Attributierte Grammatiken

Für den Übersetzerbau haben Spezifikationen in Form attributierter Grammatiken die größte Bedeutung gewonnen, da sie der Aufgabenzerlegung sehr nahe kommen und Implementierungen daraus systematisch oder automatisch abgeleitet werden können. In Kapitel 5 führen wir attributierte Grammatiken und Attributauswerter ein und wenden sie in Kapitel 6 zur semantischen Analyse an.

Für die Analyse einer Sprachdefinition hinsichtlich der kontextabhängigen Eigenschaften hat sich folgendes Vorgehen bewährt: Zunächst untersuche man solche Eigenschaften, die sich auf große Bereiche der Sprache auswirken und spezifiziere sie. Dies sind bei höheren Programmiersprachen Typregeln und Regeln zur Gültigkeit von Definitionen. Dann ermittle man die verbleibenden, meist lokalen Kontextabhängigkeiten und stelle den Zusammenhang zu den globalen Eigenschaften her.

Gültigkeit von Definitionen (scope rules)

Gültigkeitsregeln

Eine Definition ordnet einem Bezeichner (*definierendes Auftreten*) bestimmte Eigenschaften zu. Beispiele sind Variablen- oder Konstantendefinitionen, die einen Bezeichner für eine Variable bestimmten Typs oder einen Bezeichner für einen bestimmten Wert einführen.

Eine solche Definition gilt in einem bestimmten Abschnitt des Programms. Das heißt, in diesem Abschnitt wird ein *angewandt auftretender Bezeichner* mit den ihm in der Definition zugeordneten Eigenschaften verwendet. Die Gültigkeitsregeln legen fest, wie angewandtes und definierendes Auftreten zu identifizieren sind. In den meisten hierarchisch strukturierten Programmiersprachen (z. B. Algol 60, Algol 68, Pascal, Modula-2, Ada) basieren die Gültigkeitsregeln auf dem Grundschema der sogenannten *Verdeckungsregel*:

> Eine Definition für einen Bezeichner gilt in dem kleinsten sie umfassenden Abschnitt (Block, Prozedur, Modul) ausgenommen darin enthaltene Abschnitte, die eine Definition für den gleichen Bezeichner enthalten.

Diese Grundregel wird meist durch Zusatzregeln ergänzt, z. B. Import und Export für Module in Modula-2 und Ada, definierendes vor angewandtem Auftreten in Pascal, Modula-2 und Ada. In manchen Sprachen sind die Gültigkeitsregeln unterschiedlich für verschiedene Klassen von Definitionen. (Z. B. ist in Pascal die Anwendung von Typbezeichnern als Referenztyp vor ihrer Definition erlaubt.)

Für Sprachen mit separat übersetzbaren Modulen muß festgestellt werden, welche Definitionen über die Grenzen von Übersetzungseinheiten hinweg im- und exportiert werden und wie diese auf beiden Seiten formuliert werden. Diese Regeln bestimmen die Aufgaben der Bindung zwischen Übersetzungseinheiten.

Typisierung

Typregeln

In statisch typisierten höheren Programmiersprachen wird Objekten in ihrer Definition neben anderen Eigenschaften ein Typ zugeordnet. Zunächst ist festzustellen, welche *Typklassen* in der Sprache vorgesehen sind (vordefinierte Grundtypen, Verbunde, Reihungen, Referenzen usw.) und durch welche Eigenschaften jeder Typ spezifiziert wird, z. B. Index und Elementtyp für Reihungen. Für Typprüfungen und die Implementierung der Objekte ist es entscheidend, welche der Eigenschaften zur Übersetzungszeit bestimmt werden können. Sind z. B. die Grenzen von Reihungen wie in Pascal statisch bestimmbar, können die Zugriffsfunktionen für Elemente übersetzt werden, andernfalls müssen Deskriptoren angelegt und dynamisch ausgewertet werden.

Typprüfungen für Ausdrücke werden entweder als Aussagen über einen Typ (z. B. "Der Ausdruck in einer bedingten Anweisung muß vom Typ `boolean` sein") oder als Relation zwischen zwei Typen formuliert: Typen der linken und rechten Seite einer Zuweisung, des formalen und des aktuellen Parameters. Es wird entweder *Typäquivalenz* (in Pascal bei Referenzparametern) oder *Typverträglichkeit* (z. B. bei Zuweisungen) gefordert. Die Regeln für diese Relationen müssen anhand der Sprachdefinition präzise und umfassend formuliert werden. Im Falle der Typverträglichkeit wird ggf. ein Wert eines Typs implizit in einen Wert eines anderen Typs konvertiert (z. B. `integer` nach `real`).

Überladene Operatoren

Eine in den meisten Sprachen zu lösende Aufgabe bei der Typbestimmung für Ausdrücke ist die *Operatoridentifikation* (*overloading resolution*): Der Typ des Ergebnisses einer Operation wie + wird durch die Typen der Operanden nach Regeln der Sprache bestimmt (in Pascal z. B. `integer`, `real` oder ein Mengentyp). Wir sprechen dann von *überladenen Operatoren*. Diese Aufgabe wird schwieriger, wenn es außer den vorgegebenen auch im Programm definierte überladene Operatoren (Algol 68, Ada) oder Funktionen (Ada) gibt.

1.2.4 Bedeutung (dynamische Semantik)

Programmausführung

Die dynamische Semantik einer Sprache definiert, in welchen Schritten ein Programm ausgeführt und welche Wirkung dabei erzielt wird. Diese Beschreibung basiert auf Grundkonzepten wie

- Werte als Ergebnisse der Auswertung von Ausdrücken,
- Variablen als identifizierbare Objekte mit bestimmter Lebensdauer, Zugriffswegen und durch Zuweisungen zugeordnetem Wert,
- Abfolge der Ausführung von Anweisungen.

Zustand

Der *Zustand* bei der Abarbeitung eines Programms wird durch die gerade ausgeführte Anweisung, die Menge der zu diesem Zeitpunkt existierenden Variablen und deren Werte beschrieben. Die Definition der dynamischen Semantik mit solchen Begriffen kann man als Definition einer abstrakten Sprachmaschine auffassen. Es ist Aufgabe des Übersetzers, ein Zielprogramm zu erzeugen, das genau die beschriebene Wirkung erzielt - soweit sie (in der Aus- und Eingabe) beobachtbar ist. Dabei müssen nicht notwendig alle Einzelschritte wie beschrieben ausgeführt werden. Dies wird besonders bei der Optimierung ausgenutzt.

Die Beschreibung der dynamischen Semantik baut auf den Begriffen der statischen Semantik (Definition, Typen) auf, wird aber im Text der Sprachdefinition davon meist nicht scharf abgegrenzt. Bei der Analyse der dynamischen Eigenschaften strukturiert man zweckmäßig nach den oben genannten Grundkonzepten.

Die Ergebnisse der Analyse der dynamischen Semantik bilden die Grundlage für die Spezifikation des Syntheseteils des Übersetzers und können unmittelbar zur Festlegung einer Zwischensprache dienen.

Die dynamische Semantik wird in den meisten Sprachdefinitionen verbal in engem Zusammenhang mit den Kontextabhängigkeiten formuliert. Als formale Beschreibungsmittel kommen denotationale Semantik, axiomatische Semantik oder die Vienna Definition Language in Frage.

Ausdrücke

Auswertung von Ausdrücken

Die abstrakte Sprachmaschine benutzt eine bestimmte Menge von Operatoren. Dabei werden in der Quellsprache überladene Operatoren unterschieden (z. B. `integer`- und `real`-Addition, Ergebnis der Operatoridentifikation). Auch Zuweisungen und Vergleich werden nach ihrem Typ unterschieden. Die Bedeutung gebräuchlicher arithmetischer und logischer Operatoren wird im allgemeinen nicht weiter erläutert, sondern als *Pragmatik* vorausgesetzt. Dies gilt meist ebenso für die Darstellung von Werten.

Es ist für die Optimierung wichtig, festzustellen, ob und wieweit die Reihenfolge der Auswertung von Teilausdrücken und Operanden vorgeschrieben ist, ob Umformungen zulässig sind (Ausnutzung von Assoziativitäts- und Distributivitätsgesetz) und ob auch Teilausdrücke ausgewertet werden müssen, die nicht zum Ergebnis beitragen (`0 * x`, `false and b`). In diesem Zusammenhang muß auch festgestellt werden, ob die Auswertung von Ausdrücken *Seiteneffekte* verursachen, d. h. den Programmzustand verändern kann (z. B. durch Zuweisungen an nicht lokale Variablen bei Funktionsaufrufen). Nur dadurch können verschiedene Auswertungsreihenfolgen die Programmausführung unterschiedlich beeinflussen.

In manchen Fällen müssen Ausdrücke zur Übersetzungszeit ausgewertet werden (Konstantendeklarationen, Reihungsgrenzen); darüber hinaus kann dies eine Maßnahme der Optimierung sein. Grundsätzlich muß dabei sichergestellt sein, daß das gleiche Ergebnis wie bei Auswertung zur Laufzeit erzielt wird (auch bei Übersetzern mit unterschiedlicher Übersetzungs- und Laufzeitarithmetik). Es gibt Operationen der abstrakten Sprachmaschine, die im Quellprogramm nicht explizit angegeben werden. Dies sind insbesondere Inhaltsoperationen für Variablen und Typkonversionen.

Objekte, Umgebungen

Lebensdauer von Objekten

Objekte werden durch ihre Klasse (Variable, Prozedur usw.), ihren Typ und ihre *Lebensdauer* beschrieben. Wir unterscheiden

- statische oder globale Objekte, deren Lebensdauer die gesamte Programmausführung umfaßt,

- lokale Objekte, deren Lebensdauer auf die Abarbeitung einer Prozedur oder eines Blockes begrenzt ist, und
- dynamische Objekte, deren Lebensdauer mit ihrer Generierung (`new`-Operation in Pascal) beginnt und bei Programmende bzw. expliziter Vernichtung (`dispose`-Operation in Pascal) endet.

Es kann Gegenstand der Optimierung sein, die tatsächliche Lebensdauer eines Objektes weiter zu verkürzen. Die Eigenschaften von Objekten der beiden ersten Kategorien werden im Programm durch Definitionen bestimmt, die von dynamischen Objekten durch Parameter des Generators.

Umgebung

Die Menge der lokalen Objekte eines Prozeduraufrufes oder der Ausführung eines Blocks fassen wir als zusammengesetztes Objekt auf (*Schachtel, activation record*). Zusammen mit den Objekten der statisch umfassenden Aufrufe bilden sie die Umgebung der direkt zugreifbaren Objekte. Falls die Quellsprache rekursive Aufrufe nicht ausschließt, werden geschachtelte Aufrufe, die geschachtelte Lebensdauer der Objekte bedeuten, durch einen Keller von Schachteln modelliert (Laufzeitkeller). Ohne rekursive Aufrufe können alle lokalen Objekte auch wie statische Objekte behandelt werden (z. B. in Fortran). Falls die Sprache nebenläufige Prozesse oder Koroutinen erlaubt, muß jedem von ihnen ein eigener Keller zugeordnet werden.

Zugriffswege

Definitionen von Objekten führen zugleich auch *Zugriffswege* für sie ein, z. B. Indizierung für Reihungskomponenten, Selektion für Verbundkomponenten. Sie sind als Folge von Grundoperationen der abstrakten Sprachmaschine aufzufassen. Ein besonderes Augenmerk muß den Möglichkeiten gewidmet werden, mit denen dasselbe Objekt auf verschiedenen Zugriffswegen erreicht werden kann (z. B. Referenzparameter und Zugriff auf den aktuellen Parameter als globale Variable). Wir sprechen dann von *Aliasnamen*. Hierdurch werden insbesondere die Möglichkeiten der Optimierung eingeschränkt.

Ablaufstrukturen

Ablaufstrukturen

Üblicherweise kommen in imperativen Sprachen die Grundstrukturen Verzweigung (`if, case`), Schleifen (`for, while, repeat`), Sprünge auf markierte Programmstellen und Aufrufe vor. Eventuell kommen Ausnahmebehandlung (exceptions), Koroutinenumschaltung und Prozeßsynchronisation hinzu. Häufig werden komplexere Konstrukte durch einfachere definiert (Schleifen durch Sprünge). Folgende besonders problemträchtigen Eigenschaften verdienen erhöhte Aufmerksamkeit:

- Kann die Lebensdauer von Objekten durch Sprünge oder Anweisung zum Verlassen von Schleifen beendet werden?

- Wann werden in Zählschleifen die begrenzenden Ausdrücke ausgewertet; kann der Zähler explizit oder implizit verändert werden, welchen Wert hat er nach Abarbeitung der Schleife?
- Welche Formen der Parameterübergabe sind vorgeschrieben und welche sind als Implementierung erlaubt (call by value,by reference,by result,by value and result,by name)?

Laufzeitprüfungen

Laufzeitprüfungen

Ein Programm muß gewisse Bedingungen erfüllen, damit es ausführbar ist. Wenn sie nur bei der Ausführung entschieden werden können, da sie von den Eingabedaten abhängen, muß der Übersetzer dafür Laufzeitprüfungen in das Zielprogramm einsetzen. Dies sind z. B. Überschreitung von Wertebereichen, Indizierung außerhalb von Reihungsgrenzen, Zugriff mit undefinierten Referenzen.

In Sprachdefinitionen wird meist nicht explizit zwischen Bedingungen zur Übersetzbarkeit und zur Ausführbarkeit unterschieden. Es werden Formulierungen verwendet wie "Die Wirkung des Konstruktes ... ist nicht definiert, falls ...". Damit hätte der Implementierer jegliche Freiheit, Laufzeitprüfung zu erzeugen, wegzulassen oder zur Übersetzungszeit zu prüfen, falls möglich. Die beste, aber aufwendigste Lösung ist es, bei der Übersetzung jegliche Informationen auszunutzen, um Laufzeitprüfungen überflüssig zu machen (s. Kap. 8) und die verbleibenden auf Wunsch abschalten zu können. Vorsicht ist geboten, wenn die Verletzung solcher Bedingungen zur Übersetzungszeit erkannt wird: Das Programm darf nur dann als nicht übersetzbar klassifiziert werden, wenn die betreffende Programmstelle unter beliebigen Eingabedaten erreicht wird.

1.3 Eigenschaften der Zielmaschine

Maschineneigenschaften

Mit der Analyse der Zielmaschineneigenschaften wird die Grundlage für die Spezifikation des Syntheseteils des Übersetzers gelegt. Insbesondere müssen die Eigenschaften untersucht werden, mit denen die Konzepte der abstrakten Maschine (Zwischensprache) implementiert werden können. In diesem Abschnitt klassifizieren wir nur die in üblichen Fällen wichtigsten Eigenschaften. In Kapitel 7 gehen wir genauer auf den Entwurf der Abbildung von der abstrakten auf die reale Maschine ein.

Der Aufwand für Entwurf und Implementierung wird wesentlich durch die *Orthogonalität* (uneingeschränkte Kombinierbarkeit) der Maschineneigenschaften bestimmt, d. h. einheitlicher Registersatz, alle (sinnvollen) Adressierungsarten in jeder Instruktion erlaubt, einheitliche Instruktionsformate.

Speicher

Speicherstruktur

Der gesamte vom Programm nutzbare Hauptspeicher ist sein *Adreßraum*. Er kann durch den Umfang des realen Hauptspeichers beschränkt sein oder nur durch den Wertebereich von Adressen beim *virtuelle Speicher*. Zur Abbildung der Objekte der abstrakten Maschine wird er für verschiedene Verwendungsarten segmentiert (z. B. Programm, statische Daten, Laufzeitkeller, Halde). Maximaler Umfang von Segmenten und deren Verwendungsarten können durch die Maschine eingeschränkt sein. Für die Speicherabbildung und Adressierung von Objekten ist es von Bedeutung, welches die kleinste adressierbare Speichereinheit ist (Byte oder Wort) und welche Randbedingungen für die Adressierung von Objekten eingehalten werden müssen (Ausrichtung).

Register

Registerstruktur

Register nehmen Operanden und Ergebnisse von Operationen auf. Es werden allgemein verwendbare Register oder *Registerklassen* für bestimmte Verwendungsarten unterschieden: Datenregister, Adreßregister, Indexregister, Gleitpunktregister. Daneben gibt es Spezialregister, die von bestimmten Instruktionen implizit benutzt werden, z. B. Instruktionszeiger, Kellerpegel, Bedingungscode.

Zugriffe auf den Speicher werden langsamer ausgeführt als Registerzugriffe. Die Ausführungszeit des Programms kann deshalb wesentlich reduziert werden, wenn der Übersetzer möglichst viele Speicherzugriffe durch Registerzugriffe ersetzt. Steht ein großer, einheitlicher Registersatz zur Verfügung, so können solche Optimierungen sehr wirkungsvoll sein. Dies gilt besonders für die Klasse der RISC-Prozessoren (reduced instruction set computer). Hier sind Speicherzugriffe nur mit speziellen Instruktionen zum Laden und Speichern möglich. Alle übrigen Operationen verwenden ausschließlich Registeroperanden. Einige Prozessoren verwalten ihre Register als Keller, so daß bei Prozedureintritt gekellert und bei Prozedurrückkehr entkellert wird. Lokale Objekte von Prozeduren können dann unmittelbar Registern zugeordnet werden. Überlappen sich die Kellerausschnitte, die der aufgerufenen Umgebung zugeordnet sind (register windows), so können Parameter effizient in Registern übergeben werden.

Adressierungsarten

Operanden von Instruktionen enthalten entweder den zu verknüpfenden Wert unmittelbar, benennen ein Register, das den Wert enthält, oder sind eine Formel zur Berechnung der Speicheradresse des Wertes (z. B. Summe aus dem Inhalt zweier Register und einer Relativadresse). Es ist Aufgabe der Code-Erzeugung und -Optimierung, unter den vielfältigen Möglichkeiten die kostengünstigste auszuwählen.

Instruktionsformat

Das Instruktionsformat bestimmt zu den Instruktionen Anzahl und Adressierungsart der Operanden und ggf. Modifikatoren, die den Umfang der zu verknüpfenden Werte bestimmen. Instruktionen mit n expliziten Operanden nennt man n-Adreßinstruktionen. Einen Instruktionssatz charakterisiert man nach Zahl der in zweistelligen Operationen explizit angegebenen Operanden (2 Operanden, Ergebnis):

0-Adreß	beide Operanden und das Ergebnis in einem Keller,
1-Adreß	ein Operand und das Ergebnis in einem vorgegebenen Register (Akkumulator),
2-Adreß	ein Operand nimmt nach der Operation das Ergebnis auf,
3-Adreß	beide Operanden und das Ergebnis werden unabhängig angegeben.

Der häufigste Fall sind 2-Adreß-Maschinen. Kann das Ergebnis auch in einem Speicheroperanden anfallen, so sind besondere Optimierungsmaßnahmen nötig, um diese Eigenschaft vorteilhaft auszunutzen.

Instruktionssatz

Instruktionssatz

Schließlich ist der Instruktionssatz der Maschine zu untersuchen, um festzustellen, welche Operationen der abstrakten Maschine unmittelbar auf eine Instruktion abgebildet werden können. Für die verbleibenden Operationen müssen geeignete Instruktionsfolgen (z. B. für Zuweisung von Verbunden) oder Aufrufe von Laufzeitroutinen (z. B. für fehlende Gleitpunktinstruktionen) entworfen werden. Spezialinstruktionen für Prozeduraufrufe müssen besonders sorgfältig analysiert werden, da ihre Wirkung die Struktur der Schachteln mitbestimmt.

Parallelverarbeitung

Prozessoren sind zur Steigerung ihrer Leistung aus parallel arbeitenden Komponenten zusammengesetzt. Für einige Prozessorklassen kann die Ausführungszeit der Programme durch eine günstige Anordnung der Instruktionen (*instruction scheduling*) wesentlich reduziert werden. Prozessorinterne Parallelität kann nach grundsätzlich verschiedenen Prinzipien erzielt werden:

Parallele Funktionseinheiten

Funktionale Parallelität liegt vor, wenn einzelne Instruktionen von verschiedenen Funktionseinheiten parallel ausgeführt werden. Im einfachsten Fall sind dies die zentrale Prozessoreinheit mit Koprozessoren für spezielle Instruktionen, z. B. für Gleitpunkt- oder Graphikoperationen. Das Prinzip wird konsequent fortgesetzt in Prozessoren mit einer großen Anzahl von Funktionseinheiten, die in jedem Schritt eine individuelle Instruktion abarbeiten. Der Code für solch einen Prozessor ist zu einer Folge von Instruktionstupeln mit je einer Komponente für jede

Funktionseinheit angeordnet. Man spricht auch von *horizontalem* Code (oder *very long instruction word*). Es ist Aufgabe des Übersetzers, Möglichkeiten der Parallelisierung des Instruktionsstromes zu erkennen und auszunutzen, ohne die Bedeutung des Programmes zu verändern.

Vektoroperationen

Nach dem Prinzip der Datenparallelität arbeitende Prozessoren können jeweils eine Instruktion gleichzeitig auf die Elemente von Operandenvektoren anwenden. Sind solche mächtigen Vektoroperationen nicht schon im Quellprogramm enthalten, so muß sie der Übersetzer durch Analyse und Umstrukturierung von Schleifen erzeugen.

Fließband-verarbeitung

Ein drittes Parallelisierungsprinzip basiert auf der Untergliederung jeder Instruktion in mehrere, nacheinander auszuführende Abarbeitungsschritte, z. B. Decodieren der Operation, Operanden beschaffen, Operation ausführen, Ergebnis ablegen. Für jeden der Schritte ist eine spezielle Funktionseinheit zuständig. Die Funktionseinheiten sind so hintereinandergeschaltet, daß sie den Instruktionsstrom wie am *Fließband* abarbeiten (*pipelining*). Der Übersetzer versucht die Instruktionen so anzuordnen, daß keine Abhängigkeiten zwischen dicht aufeinanderfolgenden Instruktionen bestehen. Je nach Ausführung des Prozessors würden solche Konflikte von der Hardware erkannt und duch kurzfristiges Anhalten des Fließbandes vermieden, oder aber von der Hardware ignoriert und zu falschen Ergebnissen führen. Im ersten Fall tragen die Übersetzermaßnahmen zur Effizienzsteigerung bei, im zweiten sind sie für die korrekte Übersetzung notwendig.

Die Technik der Fließbandverarbeitung wird häufig mit einer der beiden anderen in einem Prozessor kombiniert. Auch die Implementierung von Programmen auf mehreren gekoppelten, verteilten Prozessoren kann mit Übersetzungsmethoden unterstützt werden. Hier muß jedoch in größeren Programmeinheiten und mit genauerer Kenntnis über das dynamische Verhalten des Algorithmus parallelisiert werden.

2. Übersetzerstruktur und Schnittstellen

Übersetzerqualitäten

Die Konstruktion von Übersetzern ist zu einem großen Anteil eine Software-Engineering-Aufgabe. Entscheidend für die Übersetzerqualitäten ist deshalb eine gute Strukturierung des Übersetzers. Wie jedes andere komplexe Software-Produkt soll auch ein Übersetzer validierbar, wartbar, portabel, adaptierbar, effizient und aus wiederverwendbaren Modulen aufgebaut sein. Diese allgemeinen Forderungen werden durch die besondere Aufgabenstellung eines Übersetzers noch verstärkt: Ein korrekter Übersetzer ist Voraussetzung für die korrekte Implementierung der mit ihm übersetzten Programme. Deshalb sind besonders hohe Anforderungen an seine Validierung zu stellen. Für Sprachen wie Pascal und Ada wurden spezielle Validierungsverfahren und Testsätze entwickelt. Im Übersetzerbau bietet sich besonders an, wiederverwendbare Module einzusetzen, da die Konzepte von Quellsprachen und Zielmaschinen häufig gleich sind und sich nur in ihrer Ausführung unterscheiden.

In diesem Kapitel demonstrieren wir an der Strukturierung von Übersetzern die Anwendung einer Reihe allgemeiner Prinzipien aus dem Software-Engineering. Da die Aufgaben von Übersetzern im Vergleich mit anderen Problemen recht präzise faßbar sind und für ihre Lösungen systematische Verfahren existieren, werden solche Prinzipien hier besonders deutlich.

Aufgabenorientierte Struktur

Die klare Zerlegung des Übersetzungsproblems führt zu einer aufgabenorientierten Struktur. Sie kann systematisch aus der Sprachdefinition und den Anforderungen der Transformation für die Zielmaschine hergeleitet werden (Abschnitt 2.1). Aus der Verfeinerung des Problems ergeben sich Teilaufgaben, die mit standardisierbaren Verfahren lösbar sind. Dies ist die Grundlage für systematische Implementierungen und für den Einsatz generierender Werkzeuge (Abschnitt 2.5). Eine aufgabenorientierte Zerlegung fördert auch die Effizienz des Übersetzers. Die verschiedenen, spezialisierten Teilaufgaben können mit jeweils passenden, effizienten Algorithmen gelöst werden. Diese innere Effizienz der Übersetzermodule bleibt für das Gesamtprodukt erhalten, wenn die Schnittstellen zwischen ihnen möglichst eng sind, also nur die tatsächlich notwendige Information übergeben wird (äußere Effizienz).

Schnittstellen

Die Übersetzung von der Quell- in die Zielsprache ist insgesamt eine sehr komplexe Transformation. Die Gliederung des Übersetzers zerlegt sie in eine Sequenz einfacherer Transformationen. Die Schnittstellen zwischen den Transformationen können wiederum als Sprachen angesehen werden. Solche Zwischensprachen sind eine besonders prä-

zise spezifizierbare Form von Schnittstellen. Die Übersetzerstruktur zeigt besonders deutlich die Verwendung von Modulen zweier grundsätzlich verschiedener Klassen. Der zentrale Modul jedes Transformationsschrittes ist ein Filter, der aus einem Informationsstrom einen anderen erzeugt. Die Filter kooperieren mit Datenmodulen, die jeweils eine abstrakte Datenstruktur mit ihren Operationen nach dem Geheimnisprinzip (information hiding) implementieren (Abschnitt 2.2).

Portierung

Die Portierbarkeit eines Übersetzers (Abschnitt 2.3) wird unterstützt, indem man Abhängigkeiten vom Grundsystem und vom Übersetzungsziel in Module isoliert und durch Zwischensprachen abtrennt. Ein Software-Produkt wie ein Übersetzer wird im allgemeinen nicht isoliert entwickelt. Er ist in eine Umgebung eingebettet, in der er mit anderen Programmen kooperiert, die ihn zu einem Sprachsystem vervollständigen (Abschnitt 2.4).

Die hier genannten Prinzipien sind nicht nur auf die Konstruktion von Übersetzern für Programmiersprachen anwendbar. Das Übersetzungsproblem läßt sich allgemeiner charakterisieren: Ein Strom von Informationen, der eine komplexe Struktur hat und zwischen dessen Elementen komplexe Beziehungen bestehen, wird analysiert und transformiert. Dies trifft z. B. auch auf Aufgaben zu, die eine Folge von Signalen in eine Folge von Reaktionen darauf umsetzen.

2.1 Aufgabenorientierte Übersetzerstruktur

Die Aufgabenstellung - bestimmt durch Quellsprache und Zielmaschine - legt eine Zweiteilung in *Analyseteil* (frontend) und *Syntheseteil* (backend) nahe. Randbedingungen, wie Portierung auf andere Zielmaschinen und Einsatz desselben Syntheseteils für die Übersetzer verschiedener Sprachen, verlangen, daß Analyse und Synthese durch eine klare Schnittstelle (Zwischensprache) getrennt sind (Abb. 2.1-1).

Fehlermeldungen und Informationen für den Benutzer müssen textuell aufbereitet und an geeigneten Stellen im Programmprotokoll eingefügt werden. Sie fallen nicht nur im Analyseteil an. Auch bei der Synthese können Übersetzerbeschränkungen oder Übersetzerfehler erkannt werden. Deshalb muß ein Modul, der Meldungen von jeder Übersetzerphase entgegennehmen kann, in beiden Teilen zugänglich sein. Im folgenden verfeinern wir diese Struktur weiter anhand der Übersetzeraufgaben.

2.1.1 Analyse

Teilaufgaben der Analyse

Es ist klar, daß der Analyseteil nicht nur ein Analysator im strengen Sinne ist: Außer der Aussage, ob die Eingabe ein übersetzbares Quellprogramm ist, muß er eine Transformation in die Zwischensprache

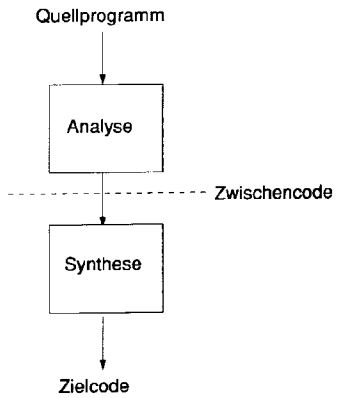


Abb. 2.1-1: Zweiteilung des Übersetzers.

liefern. Diese Übersetzung ist jedoch im wesentlichen eine strukturelle Umformung der Information aus dem Quellprogramm, die nicht die besonderen Techniken des Syntheseteils erfordert.

Die Teilaufgaben der Analyse können wir anhand der Eigenschaften höherer Programmiersprachen weiter verfeinern (Abb. 2.1-2). Das Lesen der Eingabe wird als einzelne Aufgabe abgetrennt, um spezielle effiziente Implementierungen oder auch den Anschluß von Präprozessoren zu vereinfachen. Bezeichner, Wortsymbole und Literale werden von den *Bezeichner-* und *Literalmodulen* gespeichert. Die Erkennung der Grundsymbole verbleibt als Aufgabe der *lexikalischen Analyse* (scanner). Die Berechnung des Ableitungsbaumes anhand der konkreten Syntax als Aufgabe der *syntaktischen Analyse* (*Zerteiler*, parser) wird nicht weiter zerlegt. Ihr Ergebnis ist die abstrakte Baumstruktur des Quellprogramms.

Zur *semantischen Analyse* höherer Programmiersprachen gehören vielfältige Aufgaben: Sammeln und Strukturieren von Objektdefinitionen in dem *Definitionsmodul*, Zuordnung zwischen angewandten Bezeichnern und ihren im Kontext gültigen Definitionen mit dem Modul zur *Bezeichneridentifikation*, Prüfung von *Typrelationen* (Typgleichheit, Typverträglichkeit) sowie Ermittlung der Operanden- und Ergebnistypen überladener Operatoren (*Operatoridentifikation*). Alle diese Teilaufgaben werden durch Ergänzung des abstrakten Programmbaums um Attribute (attributierter Programmbaum) gelöst. Die zentrale Steuerung der *Attributierung* sowie die Prüfung lokaler Kontextabhängigkeiten verbleibt als Aufgabe der zentralen semantischen Analyse. Damit ergibt sich die in Abbildung 2.1-3 weiter verfeinerte Struktur des Analyseteils. Für Sprachen

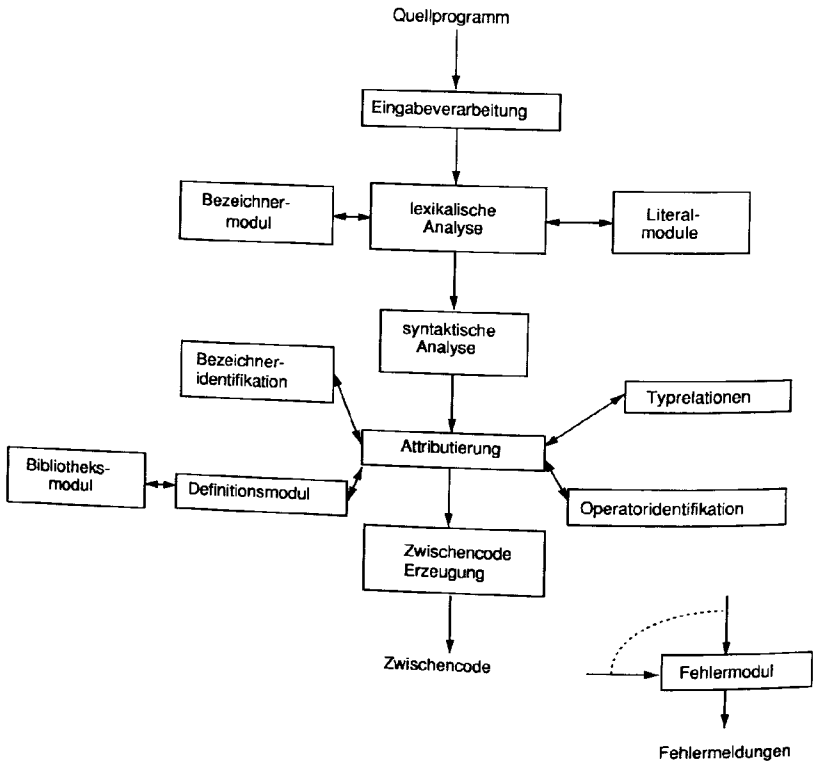


Abb. 2.1-2: Struktur und Abhängigkeiten des Analyseteils.

mit dem Konzept separat übersetzbarer Übersetzungseinheiten kommt noch ein Modul hinzu, der Definitionen aus einer *Bibliothek* vorübersetzter Einheiten liest und solche in diese schreibt.

Abhängig von den Eigenschaften der Quellsprache können selbstverständlich einige dieser Aufgaben sehr einfach zu lösen sein oder ganz entfallen, z. B. Typprüfungen bei nicht oder nicht statisch typisierten Sprachen.

2.1.2 Abstrakte Maschine, Zwischensprache

Durch eine klar ausgeprägte Schnittstelle zwischen Analyse und Synthese werden diese Übersetzeraufgaben deutlich gegeneinander abgegrenzt. Außerdem eröffnet sie die Möglichkeit, beide Übersetzerteile unabhängig voneinander wiederzuverwenden: den Analyseteil bei der Implementierung der Quellsprache für eine andere Zielmaschine; den

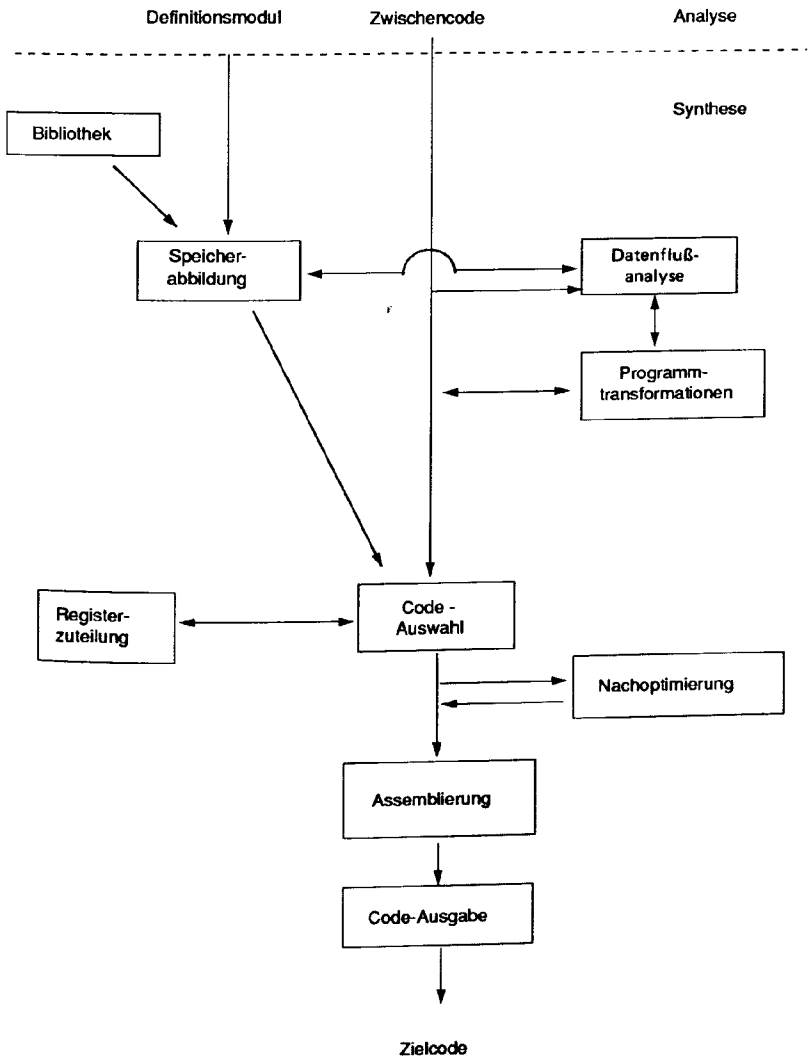


Abb. 2.1-3: Struktur und Abhängigkeiten des Syntheseteils.

Syntheseteil bei der Implementierung einer anderen Quellsprache auf derselben Maschine. Dies gilt selbstverständlich nur für solche Quellsprachen oder Zielmaschinen, deren Eigenschaften nicht zu weit von-

einander abweichen. Meist wird eine Zwischensprache für einen Übersetzer oder eine Familie von Übersetzern speziell entworfen.

Entwurf der Zwischensprache

Die Zwischensprache kann selbst als Sprache einer abstrakten Maschine aufgefaßt werden. Ihre Objekte und Operationen orientiert man zweckmäßig an den Konzepten der abstrakten Sprachmaschine. Strukturen und Operationen werden im Hinblick auf die Code-Erzeugung vereinfacht, ohne spezielle Eigenschaften der Zielmaschine in die maschinenunabhängige Zwischensprache einfließen zu lassen. Die Zwischensprache sollte so entworfen werden, daß auch der maschinenunabhängige Teil der Optimierung auf ihr aufbauen kann.

Im einzelnen sollten folgende Kriterien berücksichtigt werden:

- Strukturvereinfachung: Prozeduren werden entschachtelt. Ablaufstrukturen werden auf einige Grundoperationen zurückgeführt, aber für die Optimierung wichtige Strukturinformation sollte erhalten bleiben. Ausdrücke werden als Bäume repräsentiert.
- Explizite Operationen: Alle in der Quellsprache impliziten Operationen (auch Inhaltsoperation und Typkonversion) erscheinen explizit, überladene Operatoren sind identifiziert. Alle Zugriffsoperationen werden explizit als Folge einiger Grundoperationen (z. B. Selektion, Indizierung) angegeben.
- Keine weitreichenden Kontextabhängigkeiten: Jede zur Übersetzung einer Operation notwendige Information ist direkt mit dem Operator und seinen unmittelbaren Operanden verfügbar.
- Keine Überspezifikation: Freiheiten der Implementierung (z. B. Auswertungsreihenfolge in Ausdrücken) sollten nicht unnötig eingeschränkt werden.

Die Elemente der Zwischensprache sind Tupel (Operator, Attribute, Operanden), die zusammen eine Sequenz von Bäumen oder Graphen bilden. Sie können nach verschiedenen Techniken implementiert werden (s. Abschnitt 2.2).

2.1.3 Synthese

Der Syntheseteil leistet die Übersetzung aus der Zwischensprache in die Maschinensprache der Zielmaschine. Die Syntheseaufgaben werden im wesentlichen durch die Eigenschaften der Zielmaschine bestimmt. Dabei bilden die Auswahl und die Ausgabe der Maschinenbefehle die zentralen Strukturen. Die Erzeugung der Operanden wird in speziellen Modulen separiert. Die Notwendigkeit, Code unter Ausnutzung von Kontextinformation zu verbessern wird durch zusätzliche Optimierungsphasen berücksichtigt.

Teilaufgaben der Synthese

Im Syntheseteil lassen sich weitere Teilaufgaben modular abtrennen (s. Abb. 2.1-3). Die zentrale Phase der Synthese ist die *Code-Auswahl*. Sie bestimmt zu jedem Zwischensprachoperator eine Folge von Maschinen-

befehlen mit ihren Operanden. Mit Hilfe begrenzter Kontextinformation wird diese Auswahl verbessert. Für die Operanden wird hier ihre Form (Adressierungsart) festgelegt. Die darin verwendeten Register werden im Modul *Registerzuteilung* vergeben. Die *Speicherabbildung* ordnet Objekten der Quellsprache (repräsentiert im Definitionsmodul) Adressierungsinformation für ihre Unterbringung im Speicher der Zielmaschine zu (Speicherklasse, Relativadressen, Umfang, Ausrichtung). Die Abbildung kann bei einem Durchgang durch die Definitionen vorgenommen werden; ihre Ergebnisse stehen der Code-Auswahl zur Verfügung.

Das Ergebnis der Code-Auswahl ist eine Folge von Maschinenbefehlen als Tupel (Operation, Operanden). Sie werden durch die *Assemblierung* in das Instruktionsformat (Bitmuster) der Zielmaschine codiert. Außerdem werden hier die realen Code-Adressen zu den Marken eingesetzt.

Der vom Assemblierer erzeugte Code wird auf eine Datei in einem bestimmten Format ausgegeben. Sie enthält neben dem Maschinenprogramm Informationen über die Struktur der Code-Segmente, Referenzen auf Objekte in anderen Code-Dateien sowie Informationen für Laufzeittesthilfen. Der Aufbau der Code-Datei wird vom Binder, Lader und den Testhilfen bestimmt und kann bei gleichem Prozessor der Zielmaschine für verschiedene Betriebssysteme unterschiedlich sein. Deshalb trennt man die Code-Ausgabe zweckmäßig von der Assemblierung ab.

Optimierung

Die soweit beschriebenen Synthesemodule übersetzen Zwischensprachkonstrukte unter Berücksichtigung von nur sehr begrenztem Kontext (einzelne Zwischensprachoperatoren oder kurze Folgen davon). Um möglichst speicher- und laufzeiteffizienten Code zu generieren, muß größerer Kontext betrachtet werden. Dazu werden vielfältige Verfahren angewandt, die in *Optimierungsphasen* des Übersetzers zusammengefaßt werden. (Obwohl der erzeugte Code im allgemeinen nicht optimal ist, hat sich der Begriff Optimierung eingebürgert.) Wir unterscheiden zwischen Optimierungen auf der Ebene der Zwischensprache und solchen auf dem erzeugten Maschinencode.

Auf der Ebene der Zwischensprache wird der *Datenfluß* des Programms analysiert, um überflüssige Berechnungen zu eliminieren (z. B. Zuweisungen an nicht weiter benutzte Variablen, mehrfach verwendete oder zur Übersetzungszeit bestimmbare Ausdrücke, Laufzeitprüfungen) oder Speicherbedarf für Variablen und Zwischenergebnisse zu reduzieren. Solche Analyseverfahren werden auf nicht durch Sprünge oder Marken unterbrochene Anweisungsfolgen, auf Prozedurrümpfe oder gar über mehrere Prozeduren hinweg angewandt. Sie sind weitgehend maschinenunabhängig.

Auf der Ebene des erzeugten Codes wird mit der *Nachoptimierung* versucht, kurze Befehlsfolgen durch effizientere zu ersetzen, oder einzel-

ne, überflüssige Befehle wegzulassen. Solche Redundanz wird häufig durch Zusammensetzen unabhängig erzeugter Code-Stücke verursacht.

2.2 Schnittstellen

Die in Abschnitt 2.1 spezifizierten Übersetzermodule werden wegen ihrer prinzipiell unterschiedlichen Außenschnittstellen in zwei Modulklassen eingeteilt: *Filter* und *Datenmodule*.

2.2.1 Filter

Transformation durch Filter

Filter transformieren einen Eingabestrom in einen Ausgabestrom. Sie können zu Ketten von einzelnen Transformationsschritten zusammengesetzt werden (Filter und *pipes* in Unix). Die zentralen Übersetzermodule bilden eine solche Kette von Filtern: Eingabeverarbeitung, lexikalische und syntaktische Analyse, Attributierung, Code-Auswahl, Assemblierung, Code-Ausgabe.

Die Datenströme zwischen den Filtern repräsentieren das zu übersetzende Programm jeweils in einer Zwischenform: Zeichenfolge, Symbolfolge, Strukturbaum, Zwischencode, Folge abstrakter Instruktionen, assemblierte Instruktionen.

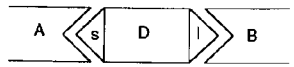
Solch ein Datenstrom ist eine lineare Folge von meist strukturierten Elementen, z. B. Tupel, die Grundsymbole der Symbolfolge oder Maschinenbefehle des Zielcodes beschreiben. Im Falle des Strukturbaums oder des Zwischencodes wird eine höhere Datenstruktur (Baum bzw. Graph) als lineare Folge von Elementen (Knoten, Operatoren) erzeugt und verarbeitet.

Zwei aufeinanderfolgende Filtermodule *A* und *B* kommunizieren über einen Datenstrom *D*, den *A* elementweise produziert und *B* elementweise konsumiert. Wir unterscheiden drei Techniken zur Implementierung solcher Schnittstellen, die in Abbildung 2.2-1 (a-c) schematisch dargestellt sind.

Implementierung der Datenströme

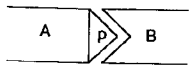
Im Fall (a) ist der Datenstrom als Datenmodul realisiert. Seine Operationen zum Lesen und Schreiben werden von den Filtern *A* bzw. *B* aufgerufen. *D* kann so durch einen Puffer im Speicher oder eine Datei implementiert werden. Bei dieser Technik sind *A* und *B* nicht direkt synchronisiert; produzierte, aber noch nicht verarbeitete Elemente werden zwischengespeichert. Im Falle des Strukturbaumes zwischen Zerteiler und Attributierung kann auch eine komplexe Datenstruktur ganz oder teilweise von dem Modul aufgebaut werden, bevor sie weiterverarbeitet wird.

In den Varianten (b) und (c) wird jedes produzierte Datenelement sofort weiterverarbeitet. Im Fall (b) ist der Konsument *B* der aktive Modul, der *A* durch Aufrufe treibt, während im Fall (c) der Produzent *A* den



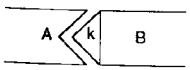
s: schreibe Element
l: lies Element

a) Datenstrom als Modul (Puffer, Datei)



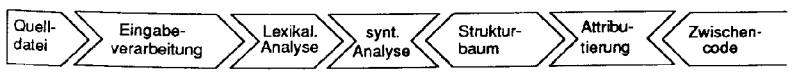
p: produziere Element

b) Konsument B ruft Produzent A auf



k: konsumiere Element

c) Produzent A ruft Konsument B auf



d) Typische Analyseschnittstellen

e) Typische Syntheseschnittstellen

Abb. 2.2-1: Schnittstellen zwischen den zentralen Filtermoduln des Übersetzers.

Konsumenten zum Verarbeiten eines Elementes aufruft. Die Unterschiede äußern sich somit auch in der Ablaufstruktur der Module *A* und *B*. Die Komponenten zusammengesetzter Datenstromelemente werden in den Fällen (b) und (c) als Parameter der Schnittstellenprozeduren übergeben.

Abbildung 2.2-1 (d) und (e) zeigen typische Ausprägungen der Filterschnittstellen im Übersetzer. Im Analyseteil sind häufig der Zerteiler und die Attributierung die treibenden Filter, zwischen denen der Strukturbaum als Datenstruktur aufgebaut wird. Unter bestimmten Voraussetzungen bei sehr einfacher Attributierung kann diese Schnittstelle auch nach Variante (c) implementiert werden. Analyse- und Syntheseteil sind hier durch den Zwischencode-Modul (häufig eine Datei) entkoppelt. An

dieser Schnittstelle kann der Übersetzer leicht zerlegt oder eine Optimierungsphase eingefügt werden. Wird die Code-Auswahl für jedes Zwischencodeelement einzeln ausgeführt, so kann auch diese Schnittstelle nach der Technik (c) implementiert werden.

Es sei noch darauf hingewiesen, daß die Filter nicht notwendig einzeln als geschlossene Programmstrukturen implementiert werden müssen. Zur Effizienzsteigerung kann man einige der Rümpfe von Funktionen zum Produzieren oder Konsumieren auch an der Aufrufstelle offen einbauen. Die Modulstruktur existiert dann nur noch konzeptionell und ist nicht mehr im Übersetzerprogramm sichtbar. Dies ist auch über mehrere Filterstufen hinweg möglich. So kann im Extremfall eines einfachen syntaxgesteuerten Übersetzers die übrige Analyse und Synthese in die Struktur der syntaktischen Analyse eingebettet werden.

2.2.2 Datenmodule

Aufgaben der Datenmodule

Datenmodule implementieren den veränderlichen Zustand einer Datenstruktur. Schnittstellenfunktionen bewirken Zustandsveränderungen oder liefern Informationen aus der Datenstruktur Die Funktionen werden von anderen Übersetzermodulen, hauptsächlich den Filtern, benutzt.

Die folgenden Module speichern die Eigenschaften von Objekten jeweils einer Klasse. Haben die Einträge zu den Objekten immer die gleiche Struktur, so werden sie z. B. als Tabelle organisiert, andernfalls als verzeigerte Datenstruktur. Jenseits der Modulschnittstelle wird auf die Einträge über die Identität bzw. Codierung des Objektes (Tabellenindex, Zeiger) zugegriffen.

Der *Bezeichnermodul* (Symboltabelle) speichert den Text von Bezeichnern (und ggf. Wortsymbolen) und codiert die Einträge bijektiv durch ganze Zahlen aus einem Indexbereich. Jeder Bezeichner wird nur einmal eingetragen und erhält einen Code verschieden von allen anderen.

Die *Literalmodule* speichern und codieren Repräsentationen der Werte von Literalen jeweils eines Typs. Sie implementieren außerdem Operationen auf solchen Werten, die zur Übersetzungszeit ausgeführt werden.

Der *Definitionsmodul* enthält je einen Eintrag zu jedem im Programm definierten Objekt, der dessen Eigenschaften beschreibt, z. B. definierter Bezeichner, Objektklasse (Variable, Konstante, Prozedur usw.) und Typ.

Die *Speicherabbildung* ordnet den definierten Objekten Adressierungsinformation zu, z. B. relative Adresse und Umfang für Datenobjekte, oder Adresse im Code für Marken und Prozeduren. Diese Information kann als Erweiterung des Definitionsmoduls aufgefaßt werden.

Den folgenden Modulen liegt jeweils ein spezielles Zustandsmodell zugrunde:

Die *Bezeichneridentifikation* implementiert die Sprachregeln für die Gültigkeit von Definitionen. Der Modulzustand repräsentiert eine Menge von gültigen Definitionen. Eine Zugriffsoperation liefert die zu einem Bezeichner gerade gültige Definition. Zustandsverändernde Operationen (Definitionen eines Abschnitts gültig oder ungültig setzen) werden von der Attributierung beim Navigieren durch den Strukturbaum veranlaßt.

Die *Bibliothek* verwaltet die Definitionen zu separat übersetzten Modulen.

Die Module für *Typrelationen* und *Operatoridentifikation* bestehen im allgemeinen nur aus einer Menge von Funktionen über Typen und besitzen keinen veränderlichen Zustand.

Die *Registerzuteilung* verwaltet die Register für die Erzeugung von Befehlsfolgen. Der Zustand dieses Moduls gibt die Menge der für Zwischenergebnisse und Adressierung verfügbaren Register an. Schnittstellenoperationen vergeben und belegen freie Register nach einer bestimmten Strategie und setzen sie nach ihrer letzten Verwendung wieder frei.

Der *Fehlermodul* sammelt Fehlermeldungen von den übrigen Übersetzermodulen und fügt sie sortiert in das Programmprotokoll ein.

2.2.3 Übersetzerpässe

Übersetzerpässe

Jeder der Filtermodule vollzieht einen vollständigen Durchgang durch eine Repräsentation des Quellprogramms. Sie bilden hintereinandergeschaltete Transformationen. Außer über die Datenströme können sie auch über gemeinsam benutzte Datenmodule Information austauschen.

Es kann notwendig oder zweckmäßig sein, die Filter zu mehreren Pässen zusammenzufassen. Ein *Übersetzerpaß* besteht aus einem oder mehreren Filtern, die ihre Transformation vollständig erledigen, bevor die Transformation des nächsten Passes begonnen wird. Daraus folgt, daß die Programmrepräsentation zwischen zwei Pässen vollständig in einem Datenmodul, z. B. einer Datei gespeichert wird.

Eine Aufteilung in Pässe kann entweder vorgenommen werden, um den Übersetzer in kleinere eigenständige Programme aufzugliedern (dies ist prinzipiell für jeden einzelnen Filter möglich), oder durch Eigenschaften der übersetzten Sprache erzwungen werden. Letzteres gilt immer dann, wenn ein späterer Filter *B* an einer Stelle *b* in der Programmrepräsentation Information benötigt, die ein früherer Filter *A* an einer Stelle *a* beschafft, und *a* beliebig weit entfernt im Programm auf *b* folgt (Vorwärtsreferenz).

In solch einem Fall muß der Filter *A* einem früheren Paß als *B* zugeordnet werden. Beispiele dafür sind Anwendungen von Bezeichnern beliebig weit vor ihrer Definition oder das Adressieren von Vorwärtssprüngen über beliebig große Distanzen. Man speichert dann die not-

wendige Information in einem von beiden Pässen benutzten Datenmodul (z. B. Definitionsmodul).

Die bisher vorgestellten Strukturierungsprinzipien basieren auf einem strikten linearen Informationsfluß, der zwar über Datenmodule verzweigen kann, aber keinen Informationsrückfluß benötigt. In manchen Situationen ist ein solcher Rückfluß notwendig: z. B. vom Zerteiler zur lexikalischen Analyse, wenn nur aus dem syntaktischen Kontext bestimmt werden kann, ob ein Symbol als Bezeichner oder Wortsymbol zu interpretieren ist. Solche Strukturen sollten nur wenn unbedingt erforderlich und nur mit größter Disziplin eingeführt werden. Sie stellen zusätzliche Randbedingungen für die Synchronisation der beteiligten Module.

2.3 Portierung von Übersetzern

Portabilität

Ein Übersetzer ist ein komplexes Software-Produkt, dessen Herstellung hohen Entwicklungsaufwand erfordert. Damit dieser in vernünftiger Relation zur Lebensdauer des Übersetzers steht, müssen für die Software-Konstruktion allgemein gültige Regeln angewandt werden, um Programmqualitäten wie Wartbarkeit, Adaptierbarkeit und Portabilität zu erreichen. Die Portierung von Übersetzern erfordert spezielle Maßnahmen, da nicht nur der Übersetzer selber, sondern auch die von ihm erzeugten Programme in der neuen Umgebung laufen sollen. Auch diese erweiterten Anforderungen der Portabilität können durch sorgfältige Strukturierung erfüllt werden, die zu verändernde Teile in möglichst kleinen Übersetzermodulen isoliert.

Bei der Portierung eines Übersetzers müssen wir einerseits zwischen Abhängigkeiten vom Betriebssystem und solchen vom Prozessor der Maschine unterscheiden, und andererseits differenzieren, ob sie das Übersetzerprogramm oder die generierten Programme betreffen.

Zum Erreichen der Portabilität des Übersetzerprogrammes selbst wendet man allgemeine Regeln des Software-Engineering an: Implementierung in einer weit verbreiteten Programmiersprache unter Einhaltung des Sprachstandards und modulare Isolierung nicht portabler Teile. Dies sind im Übersetzer im wesentlichen Systemabhängigkeiten wie

- die Eingabe des Quelltextes,
- die Ein- und Ausgabe von Zwischensprachen und Informationen über separat übersetzte Programmodule,
- die Ausgabe des Zielprogrammes und
- die Verwaltung dynamisch zugeteilten Speichers.

Diese Aufgaben sind auch in höheren Implementierungssprachen portabel lösbar. Da sie aber die Leistungsfähigkeit des Übersetzers wesentlich bestimmen, müssen meist systemabhängige effizientere Lösungen gewählt werden.

Abhängigkeiten des Übersetzerprogrammes vom Prozessor werden durch die Verwendung einer maschinenunabhängigen Implementierungssprache weitgehend vermieden. Es verbleiben die prozessorabhängige Repräsentation von Daten, z.B. numerische Wertebereiche und Genauigkeiten, die in den Literalmodulen isoliert wird, sowie die Codierung der Instruktionen im Befehlsformat des Prozessors, die der Assemblierer vornimmt.

Die Umstellung eines Übersetzers, so daß er Code für eine neue Zielmaschine erzeugt (*retargeting*), erfordert weitergehende Maßnahmen insbesondere im Syntheseteil. Wird bei gleichem Zielprozessor nur das Betriebssystem verändert, so sind davon die Code-Ausgabe, die das Format der Code-Datei bestimmt, und die Stellen der Code-Erzeugung betroffen, an denen Systemaufrufe generiert werden. Ein neuer Zielprozessor erfordert Anpassung oder Neuentwicklung

- der Speicherzuteilung,
- der Code-Auswahl,
- der Registerzuteilung,
- der Assemblierung,
- der Nachoptimierung und
- der Literalmodule.

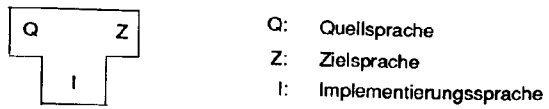
Auch diese Module können möglichst portabel realisiert werden, indem man ihre zentralen Algorithmen und Datenstrukturen systematisch und prozessorunabhängig entwickelt oder im Falle der Code-Auswahl und Nachoptimierung Generatoren einsetzt.

Die Zwischensprache zwischen Analyse- und Syntheseteil grenzt zwar die quellsprachabhängigen Aufgaben von den übrigen ab, ist aber wegen der prozessorabhängigen Literalmodule und der prozessorunabhängigen globalen Optimierung keine exakte Grenze für Prozessorabhängigkeiten.

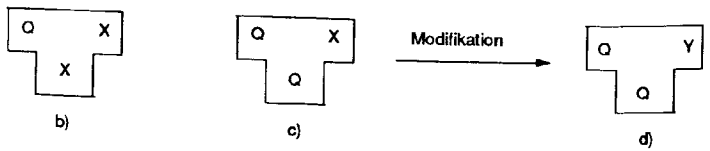
Selbstübersetzung

Eine spezielle, häufig angewandte Technik zur Portierung von Übersetzern ist die *Selbstübersetzung* (*bootstrapping*): Sie wird angewandt, wenn der Übersetzer als Programm in der Quellsprache selbst vorliegt. Dieses wird durch Modifikation der betroffenen Module an die neue Zielmaschine angepaßt und auf dieser in mehreren Übersetzungsschritten installiert.

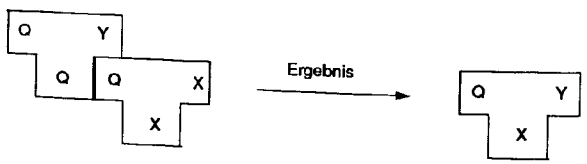
In Abbildung 2.3-1 wird eine Variante der Selbstübersetzung schematisch dargestellt. Wir repräsentieren dazu Übersetzer durch *T-Diagramme* (a) mit den drei beteiligten Sprachen (Quell-, Ziel- und Implementierungssprache). Für eine Portierung von einer Maschine *X* auf ei-



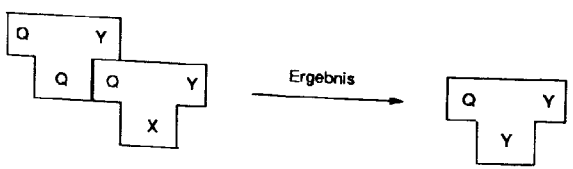
a) Schema eines Übersetzers



Voraussetzungen für die Selbstübersetzung



e) Erzeugung des Querübersetzers



f) Erzeugung des portierten Übersetzers

Abb. 2.3-1: Portierung von Maschine X auf Y durch Selbstübersetzung.

ne Maschine *Y* wird zunächst durch Modifikation der prozessorabhängigen Teile im Übersetzerprogramm (c) ein Übersetzer für *Y* hergestellt (d). Dieser wird mit dem auf *X* laufenden Übersetzer (b) zu einem *Querübersetzer* (*cross-compiler*) übersetzt, der auf *X* läuft und Code für *Y*

erzeugt (e). Mit ihm übersetzt man (d) erneut, um schließlich den auf *Y* zu installierenden Übersetzer zu erhalten. In der Praxis müssen diese Schritte weiter verfeinert werden: Da das Übersetzerprogramm selbst prozessorabhängig ist und mit dem Wechsel von *X* nach *Y* auch ein Betriebssystemwechsel verbunden sein kann, werden mehrere Varianten von (d) für die nachfolgenden Schritte benötigt.

Voraussetzung für das Verfahren ist selbstverständlich, daß sich die Quellsprache auch zur Implementierung von Übersetzern eignet. Ferner muß bei der ersten Implementierung einer neuen Sprache gleichzeitig ein Übersetzer in einer verfügbaren Implementierungssprache entwickelt werden. Dieser kann jedoch ein Prototyp sein, der z. B. auf die im endgültigen Übersetzer verwendeten Sprachelemente beschränkt ist und keine Maßnahmen zur Code-Verbesserung enthält. Durch Selbstübersetzung auf der Entwicklungsmaschine wird damit die erste Implementierung der angestrebten Qualität erzeugt. Jede weitere Portierung setzt voraus, daß eine Maschine *X* existiert, auf der der Übersetzer lauffähig ist.

Diese Technik eignet sich besonders, um neue Übersetzer zu schon weitverbreiteten Quellsprachen verfügbar zu machen. Die Verbreitung neuer Quellsprachen wird hierdurch unterstützt, falls man für die Erstimplementierung eine allgemein verfügbare Maschine *X* wählt, oder anstelle von *X* eine abstrakte Maschine setzt und dafür einen Interpretierer in einer weitverbreiteten Sprache realisiert. Dieses Vorgehen hat z. B. wesentlich zur schnellen Verbreitung von Pascal beigetragen.

2.4 Übersetzerumgebung

Der Übersetzer ist das zentrale Programm zur Implementierung einer Programmiersprache. Weitere Werkzeuge zur Programmentwicklung vervollständigen das Sprachsystem. Sie unterstützen den Benutzer bei der Herstellung von Programmen mit Hilfe von Editoren und Präprozessoren, verwalten Modulbibliotheken und erleichtern das Testen der Programme. Im folgenden wird die Funktionalität solcher Werkzeuge vorgestellt und ihr Einfluß auf die Struktur und die Schnittstellen des Übersetzers aufgezeigt.

Texteditor

Nach dem Entwurf eines Programms werden zu seiner Implementierung Übersetzer und Texteditor mehrmals abwechselnd aufgerufen, um Fehler aufzuzeigen und zu beheben. Insbesondere kleine Programme oder Programmodule entwickelt man häufig im interaktiven Dialog mit recht kurzen Denkpausen zwischen den Schritten. Diese Benutzungsart kann durch die Sprachimplementierung unterstützt werden. Die Aufrufe des Texteditors und des Übersetzers und ggf. weitere Systemkomponenten

werden in einer gemeinsamen interaktiven Steuerung zusammengefaßt. Ferner wird der Editor so eingestellt, daß er nach der Übersetzung selbständig im Quellprogramm auf Fehlerstellen positioniert und die Meldungen des Übersetzers dazu zeigt. Sehr einfache Sprachsysteme beschränken sich dabei auf den ersten vom Übersetzer festgestellten Fehler und brechen die weitere Übersetzung ab. Maßnahmen zur Instandsetzung nach Fehlern entfallen dann im Übersetzer. Allerdings kann dadurch die Zahl der notwendigen Entwicklungsschritte wesentlich erhöht werden.

Syntaxgesteuerter Editor
Das Quellprogramm wird dem Übersetzer nicht in Form einer vollständigen Eingabedatei präsentiert, sondern im interaktiven Dialog vom Benutzer entwickelt. Dabei wird der Benutzer anhand der Sprachsyntax so geführt, daß er die Programmstruktur schrittweise verfeinert. Das Ergebnis ist ein in jedem Fall syntaktisch korrektes Programm. Um die Interaktionsschritte, insbesondere bei Feinstrukturen wie Ausdrücken erträglich zu begrenzen, ist häufig zusätzlich die sequentielle Eingabe zusammenhängender Programmstücke möglich. Solche Editoren nehmen die Aufgaben der lexikalischen und syntaktischen Analyse wahr. Ihr Ergebnis ist die Repräsentation des Programms, wie sie nach der syntaktischen Analyse im üblichen Übersetzer vorliegt. Hierauf setzen die übrigen Übersetzerphasen auf. Um so konstruierte Programme auch weiterentwickeln zu können, werden solche Programmrepräsentationen gespeichert. Sie können mit speziellen Editorfunktionen verändert werden. *Inkrementelle Übersetzer* gehen mit der Lösung von Analyseaufgaben noch einen Schritt weiter: Sie berechnen auch kontextabhängige Eigenschaften schrittweise interaktiv. Dieses Vorgehen erfordert jedoch einen erhöhten Speicher- und Laufzeitaufwand, um unvollständige Strukturen zu verarbeiten und Änderungen durchzuführen.

Präprozessor
Ein Präprozessor transformiert eine Textdatei mit eingestreuten speziellen Kommandos in ein Quellprogramm, das dann vom Übersetzer weiterverarbeitet wird. Seine Transformationen sind meist unabhängig von der Quellsprache auf Texten definiert. Typische Operationen (wie sie auch der unter Unix eingesetzte *cpp* leistet) sind z. B.:
- Eine im Kommando benannte Textdatei wird an dessen Stelle eingefügt. Damit kann das Duplizieren von Programmstücken (z. B. für mehrere Programmodule gemeinsame Deklarationsteile) vermieden werden.
- Programmstücke werden abhängig vom Wert spezieller Präprozessorvariablen ausgeblendet. Damit können Varianten des Programmes (z. B. Testversionen) hergestellt werden, ohne die übereinstimmenden Programmteile zu duplizieren (bedingte Übersetzung).

- Textmakros, die auch parametrisiert sein können, werden an den Anwendungsstellen eingesetzt. Im einfachsten, nicht parametrisierten Fall werden damit Bezeichner für Literale zur Verbesserung der Lesbarkeit des Programms eingeführt. Im allgemeinen Fall ist dies ein Hilfsmittel zur Abstraktion auf textueller Ebene.

Die Präprozessorkommandos sind meist in speziell durch ein *Fluchtsymbol* gekennzeichneten Textzeilen enthalten, so daß sie ohne strukturelle Analyse des Textes erkannt und verarbeitet werden können. Die vom Präprozessor interpretierten Bezeichner für Variablen und Makros unterliegen deshalb nicht den Gültigkeitsregeln der Sprache.

Im Prinzip ist es für den Übersetzer gleichgültig, ob das Quellprogramm vollständig vom Benutzer hergestellt oder von einem Präprozessor aufbereitet wurde. Die bei der Übersetzung festgestellten Fehler sollten jedoch auf den ursprünglichen Text des Benutzers bezogen sein. Dieses Zuordnungsproblem kann z. B. dadurch gelöst werden, daß der Präprozessor im erzeugten Text spezielle Kommandos hinterläßt, welche von der Eingabeverarbeitung zur Bestimmung der Quellposition interpretiert werden. Damit kann auch die Herkunft von Textstücken aus verschiedenen Dateien kenntlich gemacht werden.

Separate Übersetzung
Separate Übersetzung

Moderne höhere Programmiersprachen, wie Ada und Modula-2, umfassen Sprachelemente zur modularen Gliederung von Programmen. Weiter werden die Implementierung eines Moduls und seine Schnittstellenbeschreibung mit den außerhalb des Moduls benutzbaren Objekten strukturell getrennt (in Modula-2: *implementation module* und *definition module*). Solche Programmstrukturen bilden *Übersetzungseinheiten*, die vom Übersetzer einzeln übersetzt werden. Die im folgenden besprochenen Maßnahmen sind dann notwendig, wenn die Programmiersprache verlangt, daß die konsistente Verwendung typisierter Objekte in den Schnittstellen zwischen Übersetzungseinheiten geprüft wird. Niedere Programmiersprachen wie Fortran und C erlauben zwar auch die separate Übersetzung von Modulen, fordern aber nicht solche Konsistenzprüfungen.

Das Ergebnis der Übersetzung des Implementierungsteils eines Moduls ist (wie im Falle der Übersetzung vollständiger Programme) der erzeugte Code, der noch Referenzen auf externe Objekte enthalten kann, die beim Binden des Programms aufgelöst werden. Die Schnittstellenbeschreibung eines Moduls wird in die Datenstruktur übersetzt. Sie enthält im wesentlichen die Beschreibungen der Schnittstellenobjekte, wie sie im Definitionsmodul des Übersetzers abgelegt und durch Informationen zur Speicherabbildung erweitert sind. Benutzt ein Modul *A* Schnittstellenobjekte eines Moduls *B*, so nimmt man deren Beschrei-

bungen aus dem Übersetzungsergebnis von *B* bei der Übersetzung von *A* zusätzlich in einem Definitionsmodul auf.

Diese Spracheigenschaft der separaten Übersetzbarkeit wird durch eine zusätzliche Komponente des Sprachsystems, die *Modulbibliothek*, realisiert. Sie nimmt die Übersetzungsergebnisse zu den einzelnen Modulen auf (z. B. mit einer geeigneten Dateiverwaltung) und stellt dem Übersetzer auf Anforderung übersetzte Schnittstellenbeschreibungen zur Verfügung. Aus den Abhängigkeiten zwischen den Modulen kann sie feststellen, welche Code-Dateien zum Gesamtprogramm gehören und diese z. B. durch Aufruf des Systembinders zum ausführbaren Programm binden.

Übersetzungsreihenfolge

Die Abhängigkeiten zwischen den Modulen erzwingen bestimmte Reihenfolgen zwischen den Übersetzungsschritten: Die Schnittstellenbeschreibung eines Moduls A muß vor ihrer Benutzung in anderen Modulen und vor der Übersetzung des Implementierungsteils von A übersetzt werden. Die Modulbibliothek kann den Programmentwickler passiv durch Prüfung oder aktiv durch Vorschlag einer Übersetzungsreihenfolge unterstützen. Wird in einem Programmsystem die Schnittstellenbeschreibung eines Moduls verändert, so können alle von dem Modul direkt oder indirekt abhängigen Module davon betroffen sein. Sie - und nur diese - müssen neu übersetzt werden. Die Modulbibliothek kann diese Übersetzungseinheiten anhand der Abhängigkeitsstruktur und z. B. Zeitstempeln der Übersetzungsergebnisse ermitteln. Das sofortige Initiieren aller notwendigen Nachübersetzungen ist nicht zweckmäßig, da in der Programmentwicklung Schnittstellenänderungen meist die Anpassung mehrerer Module erfordern. Der Benutzer sollte deshalb den Zeitpunkt der Aktualisierung explizit bestimmen. (Diese Aufgabe der Modulbibliothek ist vergleichbar mit der des Unix-Werkzeuges *make*.)

Testhilfen

Testhilfen

Die Fehlersuche in ablauffähigen Programmen kann durch interaktive Werkzeuge, sogenannte *debugger*, unterstützt werden. Das Programm wird unter der Kontrolle der Testhilfe ausgeführt. Der Benutzer kann die Ausführung an bestimmten Stellen anhalten, Variablenwerte inspizieren und ggf. ändern. Bei einem Programmabbruch werden Abbruchstelle, Aufrufhistorie und Variablenwerte der Umgebung angezeigt. Die Interaktion mit dem Werkzeug erfolgt auf dem Niveau der Quellsprache: Programmstellen werden im Quellprogramm benannt und ihre Werte gemäß ihres Typs dargestellt. Zu diesem Zweck muß der Übersetzer neben dem erzeugten Code zusätzliche Information bereitstellen, die den Variablen ihren Namen, den Typ und die Speicherstelle zuordnet und zu Stellen im Quellprogramm die zugehörigen Code-Adressen angibt. Für die Repräsentation dieser Information werden häufig bestimmte Formate in der Code-Datei festgelegt, die bei der Implementierung der Code-Ausgabe berücksichtigt werden müssen. Sind die Formate hinreichend standardi-

siert, so kann das Werkzeug auch für Programme verschiedener Sprachen angewandt werden. Datentypen und Angaben von Zugriffswegen müssen dann passend zugeordnet werden.

Interpretierer

Interpretierer

Ein Interpretierer ist ein Programm, das eine abstrakte Sprachmaschine implementiert. Seine Eingaben sind das Quellprogramm und dessen Eingabedaten. Die Operationen des Programms werden schrittweise ausgeführt (interpretiert) und Ergebnisse als Ausgabe produziert. Die Komponenten eines Interpretiererprogramms sind vergleichbar mit denen eines Hardware-Prozessors:

- Ein Programmspeicher nimmt die Repräsentation des Quellprogrammes auf.
- Ein Datenspeicher enthält die Werte der Variablen mit der für ihren Zugriff nötigen Information. Er ist als dynamische Datenstruktur organisiert, so daß Variablen bei Programmausführung generiert werden können.
- Ein Prozessormodul decodiert die Anweisungen und führt sie auf den Daten aus. Sein Verweis auf die aktuelle Anweisung und der Inhalt lokaler Variablen (Register) beschreiben zusammen mit Daten- und Programmspeicher den Zustand der abstrakten Maschine.

Die Programmrepräsentation wird nach ähnlichen Kriterien entworfen wie die Zwischensprache eines Übersetzers. Sie enthält alle für die Ausführung des Programms relevante Information, die durch statische Analyse des Programmtextes ermittelt werden kann. Dem eigentlichen Interpretierer wird ein Übersetzungsschritt vorangestellt, der das Quellprogramm entsprechend transformiert. Er umfaßt von den Übersetzeraufgaben mindestens lexikalische und syntaktische Analyse sowie die Erzeugung der Programmrepräsentation. In welchem Umfang kontextabhängige Eigenschaften in der Übersetzungsphase analysiert werden können, hängt von den Konzepten der Quellsprache ab. Typische interpretativ implementierte Sprachen (wie Lisp, APL, Snobol, Smalltalk) definieren die Typen von Objekten oder die Bindung von Eigenschaften an Bezeichner dynamisch, so daß sie erst zur Laufzeit des Programms ermittelt werden können. In diesem Fall werden die Aufgaben der Typprüfung, Operatoridentifikation und Bezeichneridentifikation vom Interpretierer übernommen. Sein Datenspeicher wird deshalb um Typeigenschaften und Zugriffsstrukturen erweitert.

2.5 Übersetzergeneratoren

Generierende Werkzeuge

Zur Herstellung von Übersetzern für Programmiersprachen werden sehr weitgehend systematisierte Verfahren angewandt. Zentrale Übersetzeraufgaben können auf der Grundlage formaler Methoden so präzise spezifiziert werden, daß daraus die Implementierung von Übersetzermodulen konstruiert werden kann. Deshalb werden seit langem Werkzeuge entwickelt, die die Übersetzerkonstruktion unterstützen und automatisieren. Die vorigen Abschnitte dieses Kapitels haben gezeigt, daß ein Übersetzer ein komplex strukturiertes Programmsystem ist. Soll er automatisch hergestellt werden, so reicht es nicht aus, die Implementierungen der zentralen Algorithmen zu generieren. Sie müssen mit passenden Schnittstellen zusammengesetzt und durch ergänzende Module zum Übersetzer vervollständigt werden. Im folgenden zeigen wir, wie die Konfigurierung des Übersetzers durch Werkzeuge unterstützt werden kann. Auf generierende Werkzeuge für spezielle Übersetzeraufgaben weisen wir in den entsprechenden Kapiteln hin.

Für zentrale Übersetzeraufgaben können mit generierenden Werkzeugen aus formalen Spezifikationen Übersetzermodule erzeugt werden. Abbildung 2.5-1 zeigt diesen Zusammenhang graphisch. Die Spezifikationen sind weitgehend deklarative Beschreibungen in Termen von Spracheigenschaften und der Übersetzeraufgabe auf der Basis eines formalen Systems. Das Werkzeug konstruiert daraus einen Algorithmus nach einem passenden Schema und implementiert diesen als Programmodul mit bestimmten Schnittstellen in einer geeigneten Programmiersprache. So werden die Grundsymbole durch reguläre Ausdrücke spezifiziert und daraus ein endlicher Automat zur lexikalischen Analyse generiert. Die syntaktische Struktur wird durch eine kontextfreie Grammatik spezifiziert und daraus ein Kellerautomat als Zerteiler generiert. Die semantische Analyse (und ggf. die Erzeugung und Transformation einer Zwischensprache) wird durch eine attributierte Grammatik spezifiziert und daraus ein Attributauswerter generiert. Die Abbildung von Zwischensprachkonstrukten auf Maschinencode-Sequenzen wird z. B. durch Baummuster spezifiziert und daraus ein Entscheidungsalgorithmus zur Code-Auswahl generiert. Für jede dieser Aufgaben existieren verschiedene Werkzeuge, die sich unter anderem in der Mächtigkeit der Spezifikationsform, dem Verfahren, das dem generierten Algorithmus zugrundeliegt, der Technik mit der er implementiert wird, seiner Implementierungssprache, seinen Schnittstellen und Leistungsdaten unterscheiden.

Zur Konstruktion eines Übersetzers mit Hilfe solcher Werkzeuge müssen die Spezifikationen entwickelt, die Schnittstellen der generierten Module einander angepaßt und die übrigen Übersetzermodule implementiert und angeschlossen werden. Auch die Spezifikationen müssen so entwickelt werden, daß die generierten Module zusammenpassen. So wird z. B. die kontextfreie Grammatik um Aktionen erweitert, die Bäume genau der Struktur erzeugen, auf der der Attributauswerter aufsetzt, und Codierungen (z. B. von Grundsymbolen) müssen zwischen den Modulen übereinstimmend festgelegt werden. Schließlich werden die Ergebnisse der Generierung im zusammengesetzten Übersetzer überprüft und daraufhin die Spezifikationen ggf. korrigiert und verbessert.

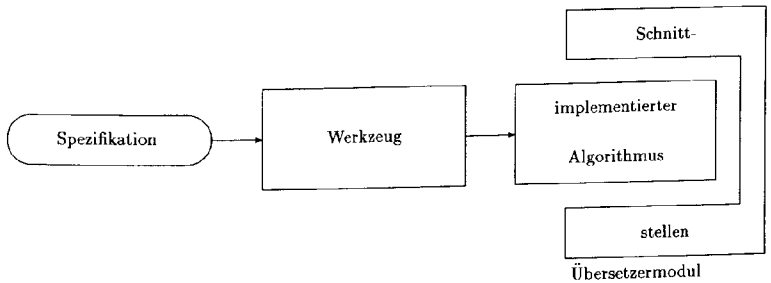


Abb. 2.5-1: Schema generierender Werkzeuge.

Solch ein Einsatz einzelner generierender Werkzeuge vereinfacht die Übersetzerkonstruktion erheblich gegenüber der manuellen Entwicklung aller Module. Es verbleibt jedoch ein beträchtlicher Aufwand zur Konfigurierung des Übersetzers. Da diese Tätigkeit nur von den verwendeten Werkzeugen und der Übersetzerstruktur abhängt, kann auch sie weitgehend automatisiert werden. Dazu werden zwei grundsätzlich verschiedene Wege beschritten. Die Konfigurierungsprobleme werden ein für alle mal gelöst, wenn alle Werkzeuge gemeinsam zusammenpassend entwickelt und in ein einziges Gesamtsystem integriert sind. Ein Beispiel dafür ist der Generator CoCo (RECH85) mit Werkzeugen zur Generierung des Analyseteils. In solchen monolithischen Systemen sind allerdings die Übersetzerstruktur, die Generierungsverfahren und alle Implementierungsentscheidungen von vornherein fixiert. Einzelne Werkzeuge können nicht durch besser geeignete ausgetauscht werden.

Übersetzerentwicklungsumgebungen

Die Verwendung sogenannter Übersetzerentwicklungsumgebungen bietet die in der Praxis meist notwendige Flexibilität. Das zugrundeliegende Prinzip sei hier an dem System Eli (WAIT88) beschrieben: Die Schritte des Entwicklungsprozesses mit den Aufrufen der Werkzeuge, der Abhängigkeiten und dem Datenfluß zwischen ihnen und den Operationen zum Zusammensetzen des Übersetzers werden als Herstellungsvorschrift formuliert (vergleichbar mit Beschreibungen für das Unix-Werkzeug *make*). Sie steuert ein allgemeines Werkzeugkontroll-

system, hier Odin (CLEM88), das für die jeweils gegebenen Spezifikationen die vorgeschriebenen Schritte durchführt. Dabei werden anfallende Zwischenprodukte verwaltet und das unnötige Wiederausführen einzelner Schritte nach Modifikation von Spezifikationen vermieden. Abbildung 2.5-2 zeigt die Struktur von Eli schematisch. Neben Werkzeugen zur Generierung zentraler Übersetzermodule sind vorgefertigte, allgemein anwendbare Implementierungen der meisten übrigen Module integriert. Solch ein System kann durch Ergänzung und Austausch von Werkzeugen verbessert und an spezielle Anforderungen angepaßt werden, ohne daß jede einzelne Übersetzergenerierung durch Konfigurierungsarbeit belastet wird.

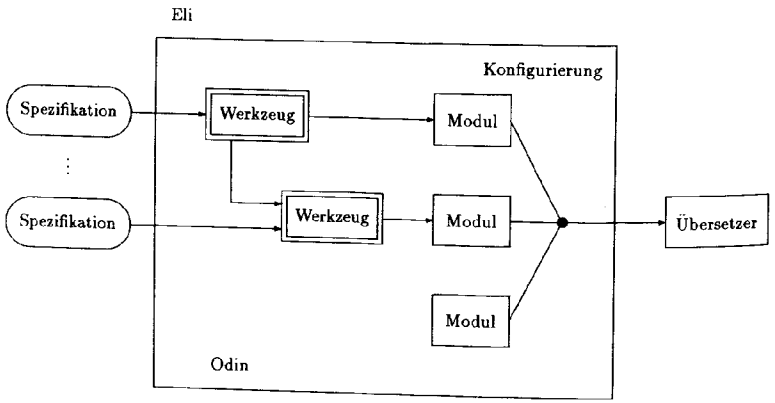


Abb. 2.5-2: Schema einer Übersetzerentwicklungsumgebung.

3. Lexikalische Analyse

Überblick

Auf lexikalischer Ebene betrachtet ist ein Programm eine Folge von Symbolen, wie Bezeichner für Objekte, Wortsymbole wie `begin` zur Strukturierung des Textes, oder Notationen für Werte wie die Gleitpunktzahl `2.57E-3`. Jedes Symbol besteht aus einer Folge von Zeichen, die nach bestimmten Regeln, z. B. für Zahlwerte, aufgebaut oder im Falle von Wortsymbolen fest vorgegeben ist. Zusammen mit Füllzeichen (Zwischenräume, Zeilenwechsel) ist das gesamte Programm eine Folge einzelner Zeichen, die der Übersetzer als Eingabe verarbeitet. Es ist Aufgabe der *lexikalischen Analyse*, in solch einer Zeichenfolge die Symbole zu erkennen und in eine Folge ihrer Codierungen zu transformieren.

Der lexikalischen Analyse liegt das formale Modell endlicher Automaten zugrunde (Abschnitt 3.2). Sie erkennen Symbole in Zeichenfolgen nach vorgegebenen Regeln. Diese werden durch reguläre Ausdrücke oder gleichwertige Methoden präzise spezifiziert (Abschnitt 3.1). Endliche Automaten können aus solchen Spezifikationen systematisch konstruiert und effizient implementiert werden (Abschnitt 3.3). Auf solchen Techniken basieren generierende Werkzeuge, von denen wir einige in Abschnitt 3.4 vorstellen.

Die auf die lexikalische Analyse folgenden Übersetzerphasen verarbeiten die Symbolfolge nicht in textueller, sondern in codierter Form. Während der zentrale Modul der lexikalischen Analyse die Art der Symbole codiert, wird ihr Informationsgehalt, z. B. der Wert von Zahlsymbolen, mit Hilfe von Datenmodulen gespeichert und codiert.

Auch außerhalb von Übersetzern finden die Techniken der lexikalischen Analyse vielfältige Anwendung. Jede Art der Analyse von Texten in formalen Sprachen (z. B. Kommando- oder Steuersprachen), aber auch von natürlichsprachlichen Texten beginnt mit einer solchen Phase. Für diese Anwendungen ist eine Gliederung in den zentralen Automaten und die Datenmodule ebenfalls nützlich. Die Techniken zur Implementierung endlicher Automaten können auch über die Verarbeitung von Texten hinaus angewandt werden, etwa zur Analyse von Signalströmen oder Operationsfolgen

Aufgaben der lexikalischen Analyse

Im folgenden geben wir zunächst einen kurzen Überblick über die Aufgaben der lexikalischen Analyse von Programmiersprachen. Ein oder mehrere im Quellprogramm unmittelbar aufeinanderfolgende Zeichen bilden ein *Grundsymbol* (*token*). Die Schreibweise ist durch Regeln für die Notation von Programmen spezifiziert. Der folgende Ausschnitt aus einem Pascal-Programm

```
if a < 0 then a := a + 0.5;
```

besteht aus der Folge der Grundsymbole `if`, `a`, `<`, `0`, `then`, `a`, `:=`, `a`, `+`, `0.5`, `;` . Nach ihrer Notation gruppieren wir Grundsymbole üblicher Programmiersprachen in

- *Wortsymbole* (*keywords*) wie `if` und `then` mit meist strukturierendem Charakter,
- *Spezialsymbole* (*delimiter*) wie `<`, `:=`, `+`, `;` für Operatoren und Begrenzer,
- *Bezeichner* (*identifier*) wie `a` als Namen für definierte Objekte, und
- *Literale* (*literals*) wie `0` und `0.5` als Notationen für Werte bestimmter Datentypen der Sprache (ganze Zahlen, Gleitpunktzahlen, Zeichen und Zeichenreihen).

Repräsentation der Grundsymbole

Die Repräsentation der Grundsymbole in der Ausgabe der lexikalischen Analyse wird durch ihre weitere Verarbeitung im Übersetzer bestimmt. Eine Darstellung durch die Zeichenreihen selbst ist zu aufwendig und für die weitere Analyse auch nicht erforderlich. Die lexikalische Analyse bildet die Grundsymbole deshalb auf eine Codierung ihres Informationsgehaltes ab. Die nachfolgende Übersetzerphase, der Zerteiler, bestimmt die syntaktische Struktur des Programmes. Die Grundsymbole spielen dabei die Rolle der Terminale in der dem Zerteiler zugrundeliegenden kontextfreien Grammatik. Wir klassifizieren deshalb die Grundsymbole nach den Terminalen, die sie repräsentieren. Dabei steht jedes Wortsymbol und Spezialsymbol jeweils für ein Terminal. Alle Bezeichner werden auf ein einziges Terminal abgebildet, da es für die strukturelle Analyse bedeutungslos ist, welcher Bezeichner an einer bestimmten Programmstelle auftritt. Dies gilt ebenso für die Literale eines jeden Typs. Für die Semantik des Programmes ist die Identität der Bezeichner und der Wert der Literale sehr wohl von Bedeutung. Sie werden deshalb als Attribut in die Codierung der Symbole aufgenommen. Schließlich ordnen wir jedem Grundsymbol seine Position im Quelltext zu, damit der Übersetzer in der Lage ist, Fehlermeldungen im Quelltext zu positionieren. Jedes Grundsymbol wird also durch ein Tripel

```
(Syntaxcode, Attribut, Quellposition)
```

z. B. `(Bezeichner, 127, (12, 24))` oder
`(Semikolon, 0, (14, 38))`

beschrieben, dessen Komponenten numerisch codiert werden. Die Komponente Attribut ist für Wortsymbole und Spezialsymbole redundant.

Neben den Grundsymbolen enthält das Quellprogramm Zeichen, die (außer zur Trennung von Symbolen) nicht zur Bedeutung des Programmes beitragen, wie Zwischenräume, Zeilenwechsel, Tabulatoren und Kommentare. Sie werden überlesen. Damit ergeben sich folgende Aufgaben für die lexikalische Analyse:

Erkennen	der Grundsymbole,
Ausblenden	bedeutungsloser Zeichen,
Codieren	der Grundsymbole.

Diese Aufgaben werden bis auf die Bestimmung der Symbolattribute von dem zentralen Modul der lexikalischen Analyse erledigt. Seine Schnittstelle zu den nachfolgenden Übersetzerphasen kann durch die Prozedur

```
procedure Lex (var gs: Grundsymbol)
```

beschrieben werden. Dabei steht `Grundsymbol` für den Typ obiger Tripel.

Die lexikalische Analyse wirkt als Filtermodul, der in den üblichen Übersetzerstrukturen durch den Zerteiler getrieben wird. Jeder Aufruf der Prozedur liefert die Codierung des nächsten, noch nicht verarbeiteten Symbols der Eingabe.

3.1 Spezifikation von Grundsymbolen

Die Schreibweise von Grundsymbolen wird meist in einem speziellen Abschnitt der Sprachdefinition beschrieben. Dazu wird im allgemeinen eine von drei formalen Beschreibungsmethoden verwendet: Syntaxdiagramme, reguläre Grammatiken oder reguläre Ausdrücke. Solche Spezifikationen dieser formal äquivalenten Methoden sind hinreichend präzise und vollständig. Aus ihnen können endliche Automaten zur Erkennung der Symbole konstruiert werden.

Grundsymbole wie Wortsymbole und Spezialsymbole sind durch Angabe ihrer Zeichenreihe eindeutig spezifiziert, z. B. `begin` und `:=`. Hierfür ist eine Beschreibung mit den obigen Methoden überflüssig. Dies gilt auch, wenn verschiedene Notationen das gleiche Symbol repräsentieren, z. B. `and` und `&` für den Konjunktionsoperator. Für Bezeichner und Literale gibt es jedoch im Prinzip beliebig viele (nur durch die Zeilenlänge begrenzte) Repräsentanten, deren Schreibweise bestimmten Regeln unterliegt. Auf ihre Spezifikation konzentrieren wir uns im folgenden.

Reguläre Ausdrücke

Die Regeln für die Zusammensetzung von Symbolen aus Zeichen können präzise durch *reguläre Ausdrücke* spezifiziert werden. So kann man z. B. eine verbale Beschreibung wie:

"Ein *Bezeichner* besteht aus einem *Buchstaben* gefolgt von beliebig vielen *Buchstaben* oder *Ziffern*."

durch die Formel

Bezeichner = Buchstabe (Buchstabe | Ziffer)*

ausdrücken. Zusammen mit den Regeln

Ziffer = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Buchstabe = a | ... | z | A | ... | Z

ist damit definiert, welche Zeichenfolgen korrekte Bezeichner sind. Jede dieser Formeln ordnet einem Namen (*Bezeichner*, *Ziffer*, *Buchstabe*) einen regulären Ausdruck zu. Er definiert eine Menge von Zeichenfolgen über einem bestimmten Zeichensatz. Diese Menge heißt *Sprache* des regulären Ausdrucks A, kurz $L(A)$. So ist

L (Ziffer) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Sprachen regulärer Ausdrücke

Wir geben nun die Bedeutung der oben informell verwendeten Konstrukte zum Aufbau regulärer Ausdrücke an. Sei Σ der zugrundeliegende Zeichensatz und seien B und C reguläre Ausdrücke mit den Sprachen $L(B)$ und $L(C)$, dann ist die Sprache $L(A)$ eines regulären Ausdrucks A wie folgt definiert:

$A = \varepsilon$	$L(A) = \{\varepsilon\}$, die Menge bestehend aus der leeren Zeichenfolge
$A = a$	$L(A) = \{a\}$, mit $a \in \Sigma$
$A = BC$	$L(A) = \{bc \mid b \in L(B), c \in L(C)\}$
$A = B\|C$	$L(A) = L(B) \cup L(C)$
$A = B^*$	$L(A) = \{\epsilon\} \cup \{ab \mid a \in L(A), b \in L(B)\}$
$A = B^+$	ist gleichwertig zu $A = BB^*$
$A = (B)$	ist gleichwertig zu $A = B$

Notation regulärer Ausdrücke

Komplexere reguläre Ausdrücke werden gebildet, indem man statt einen Namen für einen Teilausdruck einzuführen, den Teilausdruck selbst einsetzt. Dabei haben die Operatoren +, * höchste und | geringste Bindungsstärke: A | BC* ist gleichwertig zu A | (B(C)*). Damit können wir nun reguläre Ausdrücke für weitere Symbole angeben, z. B. für ganze Zahlen

Zahl = Ziffer Ziffer* oder
Zahl = Ziffer +

und Gleitpunktzahlen

Gleitpunktzahl = Zahl (Fraktion | (Fraktion | ε) Exponent)
Fraktion = . Zahl
Exponent = E (+ | - | ε) Zahl

oder in einem einzigen regulärem Ausdruck

Gleitpunktzahl =
Ziffer+ (. Ziffer+ | (. Ziffer+ | ε) E (+ | - | ε) Ziffer+).

Um längere Aufzählungen von Teilmengen des Zeichensatzes durch Alternativen, z. B. Buchstaben und Ziffern, zu vermeiden, verwendet man häufig die Notation

Ziffer = [0123456789].

Legt man eine vorgegebene Anordnung der Zeichen im Zeichensatz durch ihre Codierung zugrunde, so läßt sich weiter abkürzen

Ziffer = [0-9]
Buchstabe = [a-zA-Z].

Auch ist es häufig einfacher das Komplement einer Teilmenge des Zeichensatzes anzugeben, z. B. alle Zeichen außer Apostroph:

nicht_Apostroph = [^ ']

Soll in einem solchen Konstrukt eines der hier zum Aufbau regulärer Ausdrücke verwendeten Zeichen (wie [,], +, * usw.) als Zeichen auftreten, so kennzeichnet man es durch ein vorangestelltes Fluchtsymbol, z. B. bedeutet [^\+] alle Zeichen außer +. Damit können wir z. B. auch für Zeichenreihenliterale in Pascal einen einfachen regulären Ausdruck angeben:

Zeichenreihe = ' ([^ '] | ' ') * '

Diese Notation hat sich insbesondere für zahlreiche Software-Werkzeuge unter dem Betriebssystem Unix eingebürgert und wird auch für die in Abschnitt 3.4 vorgestellten Generatoren verwendet.

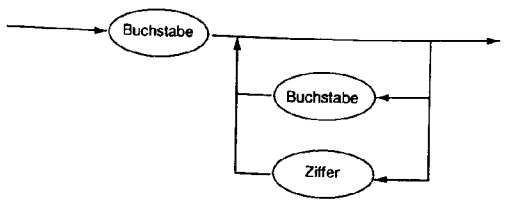
Reguläre Grammatik

Die Formeln, die einem Namen einen regulären Ausdruck zuordnen, können auch als Produktionen einer kontextfreien Grammatik (s. Kap. 4) aufgefaßt werden. Es ist dann möglich, sie durch systematische Zerlegung in benannte einfachere Teilausdrücke und Einführung rekursiver Produktionen für * und $^+$ so zu transformieren, daß alle Produktionen eine der Formen

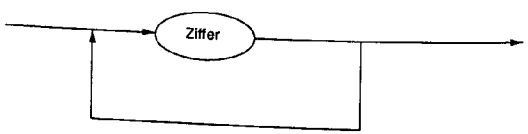
$$A = aB,\ A = a \text{ oder } A = \varepsilon$$

mit $a \in \Sigma$ haben. Eine solche Grammatik heißt *reguläre Grammatik.*

Bezeichner



Zahl



Gleitpunktzahl

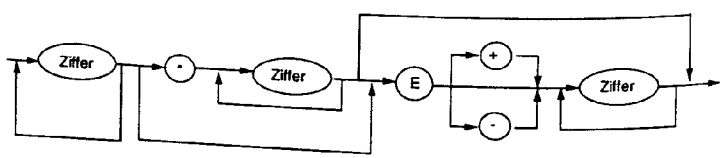


Abb. 3.1-1: Syntaxdiagramme.

Syntaxdiagramme

Eine sehr anschauliche, graphische Spezifikation von Symbolen wird durch *Syntaxdiagramme* gegeben. Diese Methode wird häufig in Sprachdefinitionen verwendet und ist ebenso präzise und mächtig wie die Spezifikation durch reguläre Ausdrücke.

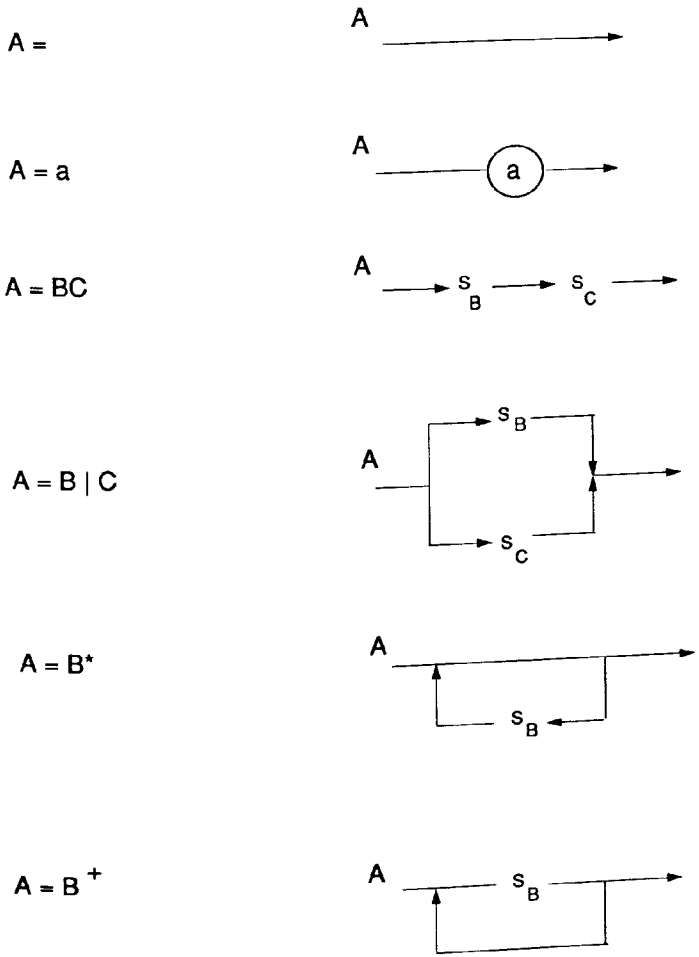


Abb. 3.1-2: Transformation von regulären Ausdrücken in Syntaxdiagramme.

Abbildung 3.1-1 zeigt Syntaxdiagramme für Bezeichner, ganze Zahlen und Gleitpunktzahlen nach den Regeln von Pascal. Die Knoten in einem Syntaxdiagramm repräsentieren einzelne Zeichen (oder eines aus einer Menge von Zeichen, z. B. die Knoten mit der Inschrift *Ziffer*). Eine Kante verbindet zwei Knoten, wenn ihre Zeichen in dem Symbol aufeinander folgen können. Jedes Diagramm hat je eine eindeutige Ein-

gangs- und Ausgangskante, deren Ursprungs- bzw. Zielknoten weggelassen ist. Damit beschreibt jeder Weg durch das Diagramm eine Zeichenreihe für das Symbol.

Aus einem regulären Ausdruck können wir systematisch ein gleichwertiges Syntaxdiagramm durch Zusammensetzen einfacherer Diagramme konstruieren. Seien S_B und S_C Syntaxdiagramme für die regulären Ausdrücke B und C, dann wendet man die in Abbildung 3.1-2 gezeigten Regeln zur Komposition an. Mit dieser Konstruktion erhält man die Syntaxdiagramme für *Bezeichner* und *Zahl* von Abbildung 3.1-1 aus den oben angegebenen regulären Ausdrücken. Im Diagramm für *Gleitpunktzahl* ist die bei schematischer Konstruktion notwendige Wiederholung des Teildiagramms für *Fraktion* durch geeignete Plazierung der Kante, die den Exponenten ausläßt, vermieden worden.

3.2 Konstruktion endlicher Automaten

Endlicher Automat

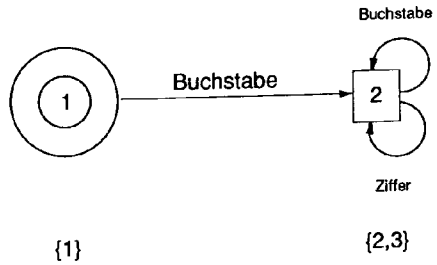
Die zentrale Prozedur der lexikalischen Analyse enthält als Kern einen Algorithmus, der eine Zeichenreihe daraufhin analysiert, ob sie zur Sprachmenge eines Symbols gehört. Ihm liegt das formale Konzept eines endlichen Automaten zugrunde. Wir zeigen hier, wie für jede der Symbolspezifikationen solch ein Automat systematisch konstruiert wird, die Automaten zusammengesetzt, implementiert und in die Prozedur eingebettet werden.

Ein *endlicher Automat* ist ein Tupel $A = (Q, \Sigma, \partial, q_1, F)$ mit einer Zustandsmenge Q, dem Zeichensatz Σ, einer Übergangsfunktion ∂, einem Anfangszustand $q_1 \in Q$ und einer Menge von Endzuständen F. Die Übergangsfunktion $\partial : Q \times \Sigma \rightarrow Q$ beschreibt das Verhalten des Automaten: Wenn gilt $\partial(q, a) = q'$ akzeptiert der Automat im Zustand q das Zeichen a und geht in dem Zustand q' über. Jede Zeichenfolge, die den Automaten aus dem Anfangszustand q_1 in einen Endzustand aus F mittels der Übergangsfunktion ∂ überführt, gehört zur Sprache $L(A)$, welche der Automat akzeptiert.

Ein endlicher Automat kann anschaulich durch einen Graphen dargestellt werden: Seine Knoten repräsentieren die Zustände. Den Anfangszustand kennzeichnen wir durch einen Doppelkreis, die Endzustände durch ein Rechteck, die übrigen durch einfache Kreise. Die mit Zeichen markierten Kanten repräsentieren die Übergangsfunktion. Abbildung 3.2-1 zeigt einen solchen Automaten für Bezeichner. Dort haben wir zur Vereinfachung für alle Buchstaben bzw. Ziffern jeweils nur eine Kante angegeben.

Automatenkonstruktion

Wir konstruieren nun einen solchen Automaten systematisch aus dem zugehörigen Syntaxdiagramm. Die Knoten im Syntaxdiagramm werden zu Übergängen des Automaten und die Verbindungen im Syntax-



Bezeichner:

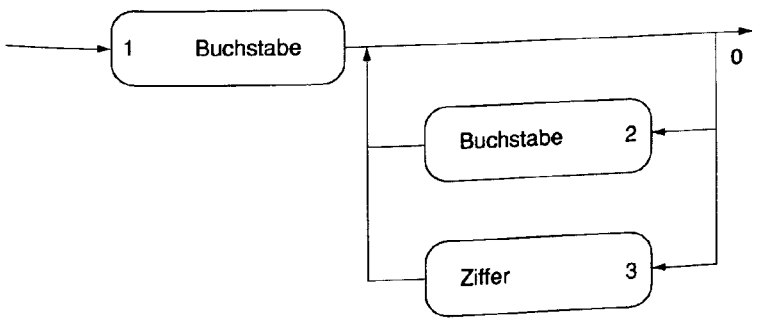
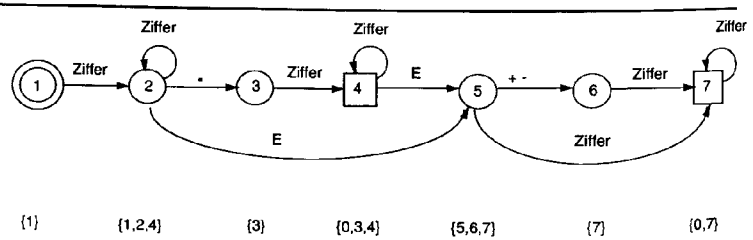


Abb. 3.2-1: Endlicher Automat und Syntaxdiagramm für Bezeichner.

diagramm werden Automatenzustände. Dazu markieren wir alle Knoten des Syntaxdiagramms eindeutig mit ganzen Zahlen und den Ausgang des Diagramms mit 0. Nach dem folgenden Verfahren berechnet man sukzessive Mengen solcher Marken, die jeweils einen Knoten des Automaten repräsentieren:

1) Die Menge der Marken von Knoten, die vom Eingang des Syntaxdiagramms direkt, ohne einen Knoten zu passieren, erreichbar sind, bildet den Anfangszustand q_1 des Automaten.
2) Wähle einen schon konstruierten Zustand q mit Markenmenge m und ein Zeichen a. Sei k die Menge von Knoten mit der Inschrift a, deren Marke in m ist. Dann bilde die Menge m' von Marken, die von einem Knoten aus k direkt erreichbar ist.
3) Gibt es noch keinen Zustand q' mit Markenmenge m', so füge q' dem Automaten zu.

4) Verbinde im Automaten q und q' durch eine Kante, die mit a markiert ist (Übergangsfunktion: $\partial(q, a) = q'$).
5) Wiederhole die Schritte 2 bis 4, bis alle Paare von konstruierten Zuständen und Zeichen betrachtet sind.
6) Jeder Zustand mit Markenmengen m ist ein Endzustand, falls $0 \in m$ gilt, d. h. der Ausgang des Diagramms direkt erreichbar ist.



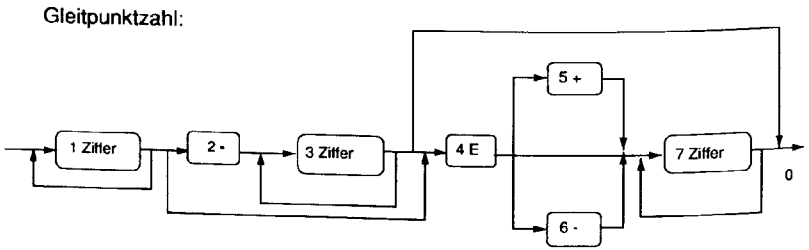


Abb. 3.2-2: Automat und Syntaxdiagramm für Gleitpunktzahlen.

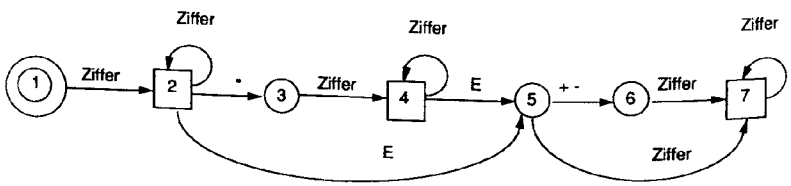


Abb. 3.2-3: Automat für ganze und Gleitpunktzahlen.

Abbildung 3.2-1 gibt auch die Markenmenge zu den Zuständen an. Abbildung 3.2-2 zeigt das Syntaxdiagramm und den so konstruierten Automaten für Gleitpunktzahlen.

Einen Automaten, der alle Grundsymbole einer Programmiersprache akzeptiert, konstruiert man nach dem gleichen Verfahren: Man fügt die Eingangskanten aller Syntaxdiagramme zu einem einzigen Eingang zusammen. Die Ausgänge werden mit unterschiedlichen Marken versehen. Damit können Endzustände, die eine solche Ausgangsmarke enthalten, der Erkennung des zugehörigen Symbols zugeordnet werden. Kombiniert man so z. B. die Diagramme für ganze und Gleitpunktzahlen aus Abbildung 3.1-1, entsteht der Automat in Abbildung 3.2-3, der mit dem Gleitpunktautomaten in Abbildung 3.2-2 fast übereinstimmt. Der Zustand 2 ist nun ein Endzustand für ganze Zahlen.

Regel des längsten Musters

In den bisher betrachteten Automaten sind auch aus Endzuständen weitere Übergänge möglich. (Dies ist nur bei Symbolen mit einem abschließenden Zeichen nicht der Fall, z. B. :=.) Es muß deshalb geklärt werden, wann der Automat tatsächlich anhält. Dies wird durch die *Regel des längsten Musters* natürlich und einfach festgelegt:

Der Automat hält an, wenn mit dem nächsten Zeichen kein weiterer Übergang möglich ist. Die bis zum zuletzt durchlaufenen Endzustand akzeptierte Zeichenfolge ist das erkannte Grundsymbol.

Falls kein Endzustand erreicht wurde, ist die Zeichenfolge fehlerhaft. Nach dieser Regel erkennt z. B. der Automat aus Abbildung 3.2-3 in den Anfängen der Zeichenfolgen

```
379 +       die ganze Zahl 379,
379.45 +    die Gleitpunktzahl 379.45,
10..20      die ganze Zahl 10,
10END       die ganze Zahl 10,
.53         einen Fehler.
```

Ein Automat für alle Pascal-Symbole würde im letzten Fall keinen Fehler, sondern das Symbol . und im nächsten Schritt eine ganze Zahl erkennen.

3.3 Implementierung der lexikalischen Analyse

Die zentrale Prozedur der lexikalischen Analyse enthält die Implementierung eines endlichen Automaten. Er erkennt jeweils ein Grundsymbol in der Eingabezeichenfolge und bestimmt dessen syntaktischen Code. Der Automat ist eingebettet in eine Umgebung, in der auf den

Anfang des anstehenden Symbols positioniert und nach seiner Erkennung das Symbolattribut bestimmt wird. Diese Aufgaben werden unter Benutzung des Eingabemoduls und der Module für Bezeichner und Literale erledigt. Im folgenden geben wir zunächst eine Implementierung des Rahmens mit den Schnittstellen zu diesen Modulen an und stellen dann zwei Implementierungstechniken für den Automaten vor.

3.3.1 Rahmen und Schnittstellen

Eingabeverarbeitung

Die Symbolerkennung und das Lesen der Eingabe gehört zu den im höchsten Maße laufzeitkritischen Aufgaben eines Übersetzers, da sie Operationen für jedes einzelne Zeichen des Quellprogramms erfordern. Die Implementierung muß deshalb besonders sorgfältig unter Effizienzkriterien entworfen werden. Insbesondere sollten unnötiges Kopieren von Zeichen, komplexe Operationen mit Zeichen oder gar Prozeduraufrufe zur Zeichenbeschaffung vermieden werden.

Das Lesen der Eingabe trennt man modular ab, um auch eine Implementierung durch systemnahe Funktionen zu ermöglichen, falls die Implementierungssprache (z. B. Pascal) eine hinreichend effiziente Lösung nicht erlaubt. Dafür ist z. B. folgende Schnittstelle zweckmäßig:

```
var       Zeile: array [Grenzen] of char;
          lrand, rrand, aktpos : integer;
          Zeilennr : integer;

function  NeueZeile : boolean;
```

Die Variable `Zeile` ist der Textspeicher des Eingabemoduls. Seine Grenzen werden außerhalb des Eingabemoduls nicht verwendet. Nach einem Aufruf von `NeueZeile` steht der Inhalt der nächsten Eingabezeile im Textspeicher in den Elementen `Zeile[lrand +1]` bis `Zeile[rrand -1]` zur Verfügung und der Eingabemodul bestimmt die Variablen, so daß gilt: `lrand < rrand` und `aktpos = lrand +1`. Ist das Eingabeende erreicht, liefert die Funktion das Ergebnis `false`. Der Analyseautomat kann innerhalb dieser Grenzen den Index des aktuellen Zeichens beliebig verschieben und Zeichen der Zeile lesen. (Mit diesen Festlegungen kann z. B. als Textspeicher auch unmittelbar ein Systempuffer zugeordnet werden, um Kopieren zu vermeiden. In einer Implementierungssprache wie C würde man auf die Reihung `Zeile` in der Schnittstelle verzichten und die drei Positionsvariablen als Zeichenreferenzen implementieren.)

Um explizite Grenzenprüfungen zu vermeiden, ist es zweckmäßig, daß der Eingabemodul ein nicht in Grundsymbolen verwendetes Zeichen an der Position `Zeile[lrand]` einsetzt (*Sentinel*-Technik). Im allgemeinen reicht das Bereitstellen jeweils einer Zeile aus, da in den

meisten Programmiersprachen ein Zeilenwechsel jedes Grundsymbol abschließt.

Die hier beschriebenen Variablen der Schnittstelle sind gleichzeitig auch das Gedächtnis der Analyseprozedur, das den Eingabezustand zwischen je zwei Aufrufen speichert.

Rahmen

Abbildung 3.3-1 zeigt den Rahmen einer Analyseprozedur ohne den Automaten. Ein Aufruf liefert den Deskriptor des nächsten Grundsymbols als Wert des Ergebnisparameters `gs`. Die äußere `repeat`-Schleife wird wiederholt bis die Eingabe auf den Anfang eines Symbols positioniert ist und der Automat von da aus ein Grundsymbol korrekt erkannt hat. Den Syntaxcode des Grundsymbols bestimmt der Automat aus dem erreichten Endzustand. Der Anfang des Symbols wird durch Überlesen von Zwischenräumen und Zeilenwechseln bestimmt. Falls der Automat von da aus keinen Endzustand erreicht, liegt ein Symbolfehler vor; die Schleife wird wiederholt.

Der Automat arbeitet nach der Regel des längsten Musters und speichert beim Verlassen eines Endzustandes die Position des letzten zum Symbol gehörenden Zeichens (`endepos`). Nach der `repeat`-Schleife wird deshalb `aktpos` für den nächsten Aufruf auf das nachfolgende Zeichen positioniert.

Symbolattribute

Schließlich wird das Symbolattribut bestimmt, sofern es nicht im Falle von Spezialsymbolen irrelevant ist. Hierfür werden Prozeduren von Modulen zur Speicherung von Bezeichnern und Literalen verwendet. Sie speichern diese Symbole und codieren sie für ihre weitere Verwendung im Übersetzer. Die Module werden von der lexikalischen Analyse abgetrennt, da die gespeicherten Symbole erst in späteren Übersetzerphasen wiederverwendet werden. Die Aufrufe dieser Prozeduren übergeben Anfangs- und Endposition des Symbols in der Eingabezeichenreihe und liefern als Ergebnis das Symbolattribut.

Bezeichnermodul

Bezeichnermodul

Der *Bezeichnermodul* implementiert eine bijektive Abbildung zwischen allen im Programm auftretenden Bezeichnern und einer ganzzahligen Codierung (Symbolattribut). Dazu legt man einen Textspeicher an, der die Bezeichnertexte aufnimmt und eine Symboltabelle mit einem Eintrag für jeden Bezeichner. Der Index des Tabelleneintrags ist gleichzeitig die Codierung des Bezeichners. Da jeder auftretende Bezeichner auf sein Vorkommen in der Tabelle geprüft werden muß, wendet man zur Reduktion der notwendigen Anzahl aufwendiger Textvergleiche meist *Hash-Verfahren* an. Aus dem Bezeichnertext wird mittels einer Hash-Funktion und eines Hash-Vektors ein Index der Symboltabelle berechnet. Dann vergleicht man mit dem neuen Bezeichner nur die schon eingetragenen Bezeichner, die den gleichen Hash-Wert geliefert haben. Sie sind in dem Bezeichnermodul z. B. in sogenannten Kollisi-

```
procedure Lex (var gs: Grundsymbol);
(* Vorbed.:
   zeile[aktpos]: erstes nicht akzeptiertes Zeichen
   Nachbed.:
   gs:              erkanntes Grundsymbol
   zeile[aktpos]: erstes nicht akzeptiertes Zeichen
*)
var endepos : integer;
begin
  gs.syntaxcode := leer;
                   (* nicht verwendeter Syntaxcode *)
  repeat
     while zeile [aktpos] = ' '
           do aktpos := aktpos+1;
     gs.pos.zeile := zeilennr;
     gs.pos.spalte := aktpos-lrand;
     if aktpos = rrand then
     begin if not NeueZeile then
           begin gs.syntaxcode := Eingabeende;
                 endepos := aktpos-1
           end
     end else begin

(* Automatenimplementierung wird hier eingesetzt *)

         if gs.syntaxcode = leer then
         begin (* falsches Symbol *)
               Fehlermeldung (Fehler, 12, gs.pos);
               if aktpos < rrand
               then aktpos := aktpos+1
         end
      end
  until gs.syntaxcode <> leer;
  aktpos := endepos+1;

  (* Bestimmung des Symbolattributs: *)
  case gs.syntaxcode of
  Bezeichner:  BzAttr(gs.pos.spalte+lrand, endepos,
                      gs.syntaxcode,gs.symbolattr);
  GanzeZahl:   GzAttr(gs.pos.spalte+lrand, endepos,
                      gs.symbolattr);
  GlpktZahl:   GlAttr(gs.pos.spalte+lrand, endepos,
                      gs.symbolattr);
  Zeichenreihe:ZrAttr(gs.pos.spalte+lrand, endepos,
                      gs.syntaxcode,gs.symbolattr);
  Semikolon, Punkt, Zuweisung, ...: ;
  end
end (* Lex *)
```

Abb. 3.3-1: Rahmen zur lexikalischen Analyse-Prozedur.

onslisten zusammengefaßt. (Man kann die Zahl der Textvergleiche weiter reduzieren, wenn man sich auf den Vergleich von Bezeichnern gleicher Länge beschränkt.)

Stimmt die Schreibweise von Wortsymbolen mit der von Bezeichnern überein, wie in den meisten heute gebräuchlichen Programmiersprachen, so ist es zweckmäßig, die Unterscheidung vom Bezeichnermodul treffen zu lassen. Eine Unterscheidung durch den endlichen Automaten würde diesen stark aufblähen. Dazu initialisiert man den Bezeichnermodul mit den Wortsymbolen und ordnet allen Einträgen in der Symboltabelle auch ihren Syntaxcode zu (den Code für Bezeichner oder die Codes der Wortsymbole). Ein neu eingetragenes Symbol ist dann in jedem Fall ein Bezeichner. Aus diesem Grunde wird beim Aufruf der Attributprozedur zusätzlich ein Ergebnisparameter für den Syntaxcode angegeben.

Literalmodule

Literalmodul

Die *Literalmodule* für verschiedene Typen (ganze Zahlen, Gleitpunktzahlen, Zeichen und Zeichenreihen) speichern Repräsentationen der Literalwerte. Da diese Werte letztendlich in dem erzeugten Zielprogramm wieder eingesetzt werden, sollten sie in der Zielrepräsentation gespeichert werden. Diese wird durch den Assemblermodul festgelegt. Je nach dessen Implementierung (s. Kap. 7) kann dies entweder eine textuelle oder eine in das Datenformat der Zielmaschine konvertierte Repräsentation sein. Die Implementierung dieser Module ist also von der Zielmaschine und der Assembler-Implementierung abhängig und sollte deshalb von der lexikalischen Analyse modular abgetrennt sein.

Weiter ist zu beachten, daß in optimierenden Übersetzern Operationen, die nur Literale als Operanden haben, zur Übersetzungszeit ausgeführt werden (siehe Konstantenfaltung in Kap. 8). Diese Operationen müssen in dem Datenformat der Zielmaschine ausgewertet werden. Falls der Übersetzer nicht auf der Zielmaschine läuft (Querübersetzer), müssen die Operationen durch spezielle Funktionen des Literalmoduls implementiert werden, der die Zielarithmetik simuliert. Diese Maßnahme ist auch erforderlich, wenn die Wertebereiche der Quellsprache von denen der Implementierungssprache des Übersetzers abweichen (z. B. Datentyp `Cardinal` in der Quellsprache Modula-2 und `integer` in der Implementierungssprache Pascal).

Für einige Literaltypen sind Besonderheiten zu beachten: Die Werte ganzer Zahlen werden nicht nur für die Konstantenfaltung, sondern auch zur Prüfung von Bedingungen der statischen Semantik benötigt (z. B. Prüfung der Grenzen von Ausschnittstypen). Eine Konvertierung ist deshalb auch notwendig, falls nicht optimiert wird.

Falls, wie z. B. in Pascal, die Notation von Zeichen- und Zeichenreihenliteralen übereinstimmt (`'X'` hat den Typ `char`, `'XY'` den Typ

`string`), wird erst durch die Länge des Literals im Zeichenreihenmodul der syntaktische Code bestimmt. Die zugehörige Attributprozedur hat deshalb ebenso wie die für Bezeichner den Syntaxcode als Ausgabeparameter.

Da mehrere der hier betrachteten Module die Zeichenreihen von Symbolen speichern (Bezeichnermodul, Zeichenreihenmodul und eventuell auch die Zahlwertmodule) ist es zweckmäßig, einen gemeinsam verwendeten Modul für diese Aufgabe zu implementieren.

3.3.2 Implementierung des endlichen Automaten

Zur Implementierung des endlichen Automaten stellen wir zwei unterschiedliche Techniken vor: tabellengesteuerte und direkt programmierte Automaten. Bei manueller Implementierung wählt man die laufzeiteffizientere Technik der direkten Programmierung. Generatoren existieren für beide Techniken (siehe Abschnitt 3.4).

Endlicher Automat

Die Übergangsfunktion ∂ eines endlichen Automaten kann man durch eine Matrix

```
var Nachf: array [Zustände, Zeichen] of Zustände
```

Tabellengesteuert

darstellen. Sie enthält an der Stelle `Nachf [q,c]` den Wert ∂ (q,c), also den Zustand, der von `q` mit dem Zeichen `c` erreicht wird. Soll der Automat nach der Regel des längsten Musters arbeiten, so muß er jeweils beim Verlassen eines Endzustandes eine Operation auslösen (Speichern des Syntaxcodes und der Zeichenposition). Hierfür wenden wir folgendes allgemeine Konzept für Automaten an: Bei jedem Zustandsübergang kann der Automat Ausgabe produzieren. Dies ist in unserem Fall ein Syntaxcode oder die leere Ausgabe. Dazu führt man eine weitere Matrix

```
var Ausgabe: array [Zustände, Zeichen] of Syntaxcode
```

ein.

Die beiden Tabellen können erheblichen Umfang annehmen: Bei etwa 40 Zuständen für die Pascal-Symbole und 128 Zeichen sind dies zweimal 5120 Einträge. Der Tabellenumfang wird drastisch reduziert, wenn man alle Zeichen, die in jedem Zustand den gleichen Übergang verursachen, auf eine *Zeichenklasse* und nur eine Tabellenspalte abbildet (z. B. Buchstaben, Ziffern usw.). Abbildung 3.3-2 zeigt die Inhalte beider Tabellen in der so komprimierten Form für den Zahlautomaten aus Abbildung 3.2-3. Einträge für die leere Ausgabe sind weggelassen. Da der Automat beim Übergang in den Zustand 0 anhält, gibt es dafür keine Zeile in den Tabellen.

Zustand	Ziffer	•	E	+-	sonstige
1	2	0	0	0	0
2	2	3, IZ	5, IZ	0, IZ	0, IZ
3	4	0	0	0	0
4	4	0, RZ	5, RZ	0, RZ	0, RZ
5	7	0	0	6	0
6	7	0	0	0	0
7	7	0, RZ	0, RZ	0, RZ	0, RZ

IZ = Ganze Zahl RZ = Gleitpunktzahl

Abb. 3.3-2: Übergangs- und Ausgabetabelle für Zahlenautomat.

```
(* lrand < aktpos < rrand, Zeile [aktpos] <> ' ' *)

    Zustand := 1; aktpos := aktpos-1;
    repeat
        aktpos := aktpos+1;
        k := Klasse [Zeile [aktpos]];
        s := Ausgabe [Zustand, k];
        if s <> leer then
        begin gs.syntaxcode := t;
              endepos := aktpos -1
        end;
        Zustand := Nachf [Zustand, k]
    until Zustand = 0
```

Abb. 3.3-3: Tabelleninterpretierender Automat.

Im Kern der Analyseprozedur setzen wir ein Programmstück ein, das den Automaten durch Interpretation solcher Tabellen implementiert, siehe Abbildung 3.3-3. Durch Austausch der Tabellen kann der Interpretierer an die jeweiligen Symbolspezifikationen angepaßt werden. Diese Technik ist deshalb zur Generierung besonders geeignet. Für diese Flexibilität muß neben dem Speicheraufwand der Tabellen auch eine Laufzeiteinbuße in Kauf genommen werden: Jedes Zeichen eines Symbols erfordert einen Schleifendurchlauf mit zwei Tabellenzugriffen. Diese Nachteile vermeidet man mit der folgenden Implementierungstechnik.

Direkt programmiert

Mit der Technik der *direkten Programmierung* setzt man den Automaten systematisch in ein Programmstück um, in dem die Zustände durch Programmstellen repräsentiert und die Übergänge durch bedingte Verzweigungen und Schleifen bewirkt werden. Der für alle Symbole konstruierte Gesamtautomat zerfällt in eine Reihe von Teilautomaten, die jeweils mit der gleichen Zeichenklasse aus dem Anfangszustand verzweigen. Jeden dieser Teilautomaten setzt man in ein Programm-

```
(* lrand < aktpos < rrand, Zeile [aktpos] <> ' ' *)

k := Klasse [Zeile [aktpos]]; aktpos := aktpos +1;

case k of
Ziffer : (* Zahlautomat *)
begin                                              (* 2 *)
while Klasse [Zeile [aktpos]] = Ziffer
      do aktpos := aktpos+1;
gs.syntaxcode := GanzeZahl; endepos := aktpos-1;
if Zeile [aktpos] = '.' then
begin aktpos := aktpos+1;                          (* 3 *)
      if Klasse [Zeile [aktpos]] = Ziffer then
      begin aktpos := aktpos +1;                   (* 4 *)
            while Klasse [Zeile [aktpos]] = Ziffer
                  do aktpos := aktpos+1;
            gs.syntaxcode := Gleitpunktzahl;
            endepos := aktpos-1;
      end else goto 99;
end;
if Zeile [aktpos] = 'E' then
begin aktpos := aktpos+1;                          (* 5 *)
     if Klasse [Zeile [aktpos]] = PlusMinus then
        aktpos := aktpos+1;                        (* 6 *)
     if Klasse [Zeile [aktpos]] = Ziffer then
     begin aktpos := aktpos+1;                     (* 7 *)
           while Klasse [Zeile [aktpos]] = Ziffer
                 do aktpos := aktpos+1;
           gs.syntaxcode := Gleitpunktzahl;
           endepos := aktpos-1
     end
end
end;  (* Zahlautomat *)
...   (* weitere Teilautomaten *)
end;  (* case *)
99:   (* kein weiterer Uebergang moeglich *)
```

Abb. 3.3-4: Direkt programmierter Automat.

stück um, das in eine Fallunterscheidung über die Zeichenklassen im Anfangszustand eingebettet wird.

Abbildung 3.3-4 zeigt einen so direkt programmierten Automaten, wobei nur der Zahlautomat aus den früheren Beispielen ausgeführt ist. Die den Zuständen entsprechenden Programmstellen sind durch Kommentare gekennzeichnet. Ein Übergang wird durch Prüfen des Zeichens, ggf. Verzweigung und Fortschalten von `aktpos` realisiert. Ist kein weiterer Übergang möglich, wird zum Ende des Automaten (Marke

99 in Abb. 3.3-4) gesprungen. Die Ausgabeoperationen werden an den entsprechenden Programmstellen unmittelbar eingesetzt. Um bei der Prüfung von Zeichen Mengenoperationen, die häufig zeitaufwendig sind, zu vermeiden, bildet man wie beim Tabelleninterpretierer die Zeichen auf Zeichenklassen ab. Mit dieser Technik reduziert man auch die Anzahl der Fallmarken und ihren Wertebereich in der äußeren Fallunterscheidung. Die in Abbildung 3.3-4 nicht gezeigten trivialen Teilautomaten für Spezialzeichen, wie ;, [,] usw. kann man mit Hilfe eines weiteren Abbildungsvektors (von Zeichen auf den Syntaxcode) zusammenfassen.

In beiden gezeigten Implementierungstechniken wird angenommen, daß ein Symbol immer ganz in einer Eingabezeile enthalten ist. Dies entspricht meist den Symbolspezifikationen von Programmiersprachen. Allerdings können sich geklammerte Kommentare (* wie in Pascal *) auch über mehrere Zeilen erstrecken. Bei der Automatenkonstruktion behandelt man sie wie Grundsymbole und konstruiert einen Teilautomaten dafür. Beim Erreichen ihres Endzustandes wird jedoch kein Syntaxcode als Ausgabe erzeugt. Ferner muß beim Übergang mit dem Zeilenendezeichen die nächste Zeile gelesen werden. Man führt dazu eine spezielle Ausgabeoperation ein. Mehrzeilige Zeichenreihenliterale können entsprechend behandelt werden.

3.4 Generatoren

Wie wir in den vorigen Abschnitten gezeigt haben, ist die zentrale Aufgabe eines Generators für die lexikalische Analyse - die Konstruktion und Implementierung eines endlichen Automaten - recht einfach. Über die Brauchbarkeit eines solchen Werkzeuges für den praktischen Einsatz in der Übersetzerkonstruktion entscheiden deshalb im wesentlichen Fragen der Einbettbarkeit des generierten Moduls in die Umgebung der lexikalischen Analyse, seine Effizienz, die Form der Symbolspezifikationen und die Handhabbarkeit des Generators. Sind diese Aspekte nicht befriedigend gelöst, ist eine systematische manuelle Implementierung vorzuziehen. In WAIT86 werden laufzeitkritische Aspekte diskutiert und Techniken zur effizienten Implementierung angegeben. Im folgenden stellen wir zu den wichtigsten Einsatzkriterien exemplarisch die Eigenschaften dreier Generatoren vor: Lex (LESK75), ein unter Unix weit verbreitetes Werkzeug, das auch außerhalb des Übersetzerbaus eingesetzt wird; GLA, eine verbesserte Version des in HEUR86 beschriebenen Generators, der nach Techniken aus WAIT86 speziell auf die Übersetzerkonstruktion und deren Leistungsanforderungen ausgerichtet ist; Rex (GROS89), eine leistungsfähigere Neuimplementierung von Lex.

Spezifikation

Spezifikationsform

In allen drei Systemen werden die Grundsymbole durch reguläre Ausdrücke spezifiziert. Sie werden in ähnlicher Form wie in Abschnitt 3.1 notiert. Lex und Rex erlauben die Einführung von Namen für mehrfach vorkommende Teilausdrücke, während für GLA jedes Grundsymbol durch einen geschlossenen Ausdruck spezifiziert wird. GLA bietet eine große, erweiterbare Menge vordefinierter Spezifikationen für Grundsymbole verschiedener Programmiersprachen an. Alle Generatoren lösen Mehrdeutigkeiten nach der Regel des längsten Musters, d. h. die längste akzeptierbare Zeichenfolge wird erkannt. Weiter können Analysekonflikte durch Kontextangaben, Beeinflussung der Automatenzustände (Lex, Rex) oder Zufügen selbstprogrammierter Analyseprozeduren (GLA) gelöst werden.

Schnittstellen

Einbettung

Die Generatoren erzeugen eine zentrale Analysefunktion, die den syntaktischen Code des erkannten Symbols als Ergebnis des Aufrufes liefert. Für GLA werden die Codes den Symbolen in einem speziellen Teil der Spezifikation zugeordnet. Sie können deshalb in Abbildungsvektoren der Implementierung effizient ausgenutzt werden. Für Lex und Rex ordnet man den regulären Ausdrücken beliebige Anweisungen in der Implementierungssprache zu, die das Funktionsergebnis bestimmen und ggf. Symbolattribute berechnen. Zur Berechnung von Symbolattributen bietet GLA statt dessen eine funktionale Schnittstelle an. Der Spezifikation eines attributierten Symbols (z. B. Bezeichner oder Literal) wird eine Funktion zugeordnet, deren Parameter auf den Symboltext verweisen und das berechnete Attribut übergeben. Damit können Bezeichner- und Literalmodule modular abgetrennt werden. Implementierungen von Standardlösungen für diese Module stehen passend mit dem Werkzeug zur Verfügung.

Implementierung

Implementierung

Alle drei Werkzeuge generieren Analysatoren in der Implementierungssprache C, Rex auch wahlweise in Modula-2. Lex und Rex erzeugen tabellengesteuerte Automaten. Die von Rex generierten Automaten benötigen nur etwa ein Viertel der Größe und gleichzeitig etwa ein Viertel der Laufzeit gegenüber denen von Lex. GLA generiert besonders effiziente Automaten nach der Technik der direkten Programmierung. Die Automatenstrukturen werden systematisch in Ablaufstrukturen umgesetzt und damit Laufzeit für eine Tabelleninterpretation erspart.

4. Syntaktische Analyse

Überblick

Programme sind in Strukturen wie Prozeduren, Anweisungen und Ausdrücke gegliedert, die selbst wieder geschachtelt sein können. Die Eigenschaften zur statischen und dynamischen Semantik von Programmelementen sind an diese syntaktischen Strukturen gebunden. Es ist Aufgabe der *syntaktischen Analyse*, aus der Symbolfolge eines Programms dessen Struktur zu ermitteln und als abstrakte Datenstruktur für die weitere Übersetzung zu repräsentieren. Die Regeln für die syntaktische Strukturierung von Programmen werden präzise durch das formale System kontextfreier Grammatiken spezifiziert. Aus einer solchen Spezifikation kann man systematisch einen Analysealgorithmus, *Zerteiler* (*parser*), konstruieren, der feststellt, ob die Symbolfolge im Sinne der Grammatik korrekt ist, und die abstrakte Struktur dazu als *Strukturbaum* ermittelt.

Zerteiler basieren auf dem formalen Modell von *Kellerautomaten*. Sie können nach verschiedenen Verfahren systematisch manuell oder mit Hilfe generierender Werkzeuge automatisch konstruiert und effizient implementiert werden. Die hohe Qualität der heute verfügbaren Werkzeuge macht die manuelle Zerteilerimplementierung weitgehend überflüssig. In diesem Kapitel stellen wir die Grundlagen der beiden wichtigsten Konstruktionsverfahren und Implementierungstechniken dazu vor. Auch bei der Verwendung von Werkzeugen sind solche Kenntnisse nützlich, um die Leistungsfähigkeit und Grenzen von Generatoren beurteilen zu können, sie angemessen einzusetzen und die Spezifikationen an die Randbedingungen des Generators anzupassen. Darüber hinaus muß die Abbildung der syntaktischen Regeln auf die abstrakte Programmstruktur sorgfältig entworfen werden, da auf dieser zentralen Datenstruktur die nachfolgende semantische Analyse und Transformation aufbaut.

Zerteiler werden auch außerhalb vollständiger Übersetzer angewandt. So reicht es bei manchen Problemstellungen aus, die Struktur von Programmen festzustellen und ggf. zu transformieren, z. B. um einfache Spracherweiterungen auf eine Kernsprache zurückzuführen, oder Programme auf strukturelle Eigenschaften hin statistisch zu untersuchen. In diesen Fällen verwendet man nur die ersten Übersetzerphasen bis zum Zerteiler und läßt ihn die gewünschte Ausgabe unmittelbar produzieren. Zerteilungsverfahren sind nicht auf die Analyse von Programmen begrenzt. Auch auf andere nach syntaktischen Regeln hierarchisch strukturierte Symbolfolgen sind sie anwendbar, z. B. Eingaben für Textformatierer oder große, komplex strukturierte Datensätze, wie Gleichungssysteme, als Eingabe für Anwendungsprogramme.

Für die Entwicklung eines Zerteilers als Übersetzermodul unterscheiden wir drei kontextfreie Grammatiken, welche die Programmstruktur nach verschiedenen Gesichtspunkten beschreiben: Die in der Sprachdefinition enthaltene Grammatik definiert den Aufbau syntaktisch korrekter Programme für den menschlichen Leser. Diese formt man so um, daß sie den Erfordernissen des Verfahrens zur Syntaxanalyse genügt, das im Zerteiler angewandt wird. Beide Grammatiken beschreiben die konkrete Syntax von Quellprogrammen. Der Zerteiler konstruiert für ein Quellprogramm eine Ableitung gemäß der Zerteilergrammatik. Dabei stößt er Aktionen an, die einen Strukturbaum zu dem Programm aufbauen. Dieser wird durch eine abstrakte Syntax beschrieben.

In Abschnitt 4.1 führen wir kontextfreie Grammatiken ein und stellen den Zusammenhang zwischen konkreter und abstrakter Syntax her. Durch kontextfreie Grammatiken definierte Sprachen werden von Kellerautomaten akzeptiert. Für bestimmte Grammatikklassen können Zerteiler als deterministische Kellerautomaten systematisch hergestellt werden - nur solche sind für die Übersetzerkonstruktion von Interesse. In den Abschnitten 4.2 und 4.3 betrachten wir zielbezogene Zerteiler zu den LL-Klassen und quellbezogene Zerteiler zu Grammatiken der LR-Klassen. Für erstere gibt es hinreichend einfache Konstruktionsverfahren, so daß sie auch für realistische Grammatiken ohne Werkzeuge anwendbar sind. Die LR-Verfahren sind mächtiger und vereinfachen deshalb etwa notwendige Anpassungen der Grammatik. Die Automatenkonstruktion ist für realistische Grammatiken jedoch nur mit Hilfe von Generatoren praktikabel. Mit den Verfahren dieser beiden Klassen können, manuell oder mit Werkzeugen, auf einfache Weise leistungsfähige Zerteiler sicher hergestellt werden. Deshalb haben andere Verfahren heute im Übersetzerbau keine praktische Bedeutung. In Abschnitt 4.5 stellen wir exemplarisch einige Zerteilergeneratoren vor. Abschnitt 4.4 ist der Behandlung syntaktischer Fehler gewidmet.

Ausführlichere Darstellungen der hier besprochenen Verfahren findet man z. B. in WAIT84. In AHOS86 werden daneben auch andere Verfahren beschrieben und AHOU72 vertieft insbesondere den theoretischen Hintergrund.

4.1 Kontextfreie Grammatiken, abstrakte und konkrete Syntax

Kontextfreie Grammatik

Eine *kontextfreie Grammatik* ist ein formales System von Ersetzungsregeln. Es beschreibt, wie bestimmte Folgen von Symbolen als Sätze der so definierten Sprache erzeugt werden können. Den Sätzen

wird durch die Grammatik außerdem eine syntaktische Struktur aufgeprägt.
Eine kontextfreie Grammatik ist ein Quadrupel $G = (T, N, P, S)$ mit

T	Menge der Terminale,
N	Menge der Nichtterminale,
$V=T \cup N$	Vokabular, T und N sind disjunkt,
$S \in N$	Zielsymbol (auch: Startsymbol),
$P \subset N \times V^*$	Menge der Produktionen.

Produktionen notieren wir in der Form $A ::= x$ (auch $A \rightarrow x$ oder A : x sind gebräuchlich).

Grundprinzip der kontextfreien Grammatiken ist die *Ableitung*: Sei $w = uAv$ eine Folge von Zeichen des Vokabulars V, und $A ::= x$ eine Produktion $p \in P$, dann wendet man p in w an, indem das Nichtterminal A durch x, die rechte Seite der Produktion, ersetzt wird. Ein solcher *Ableitungsschritt* wird durch $uAv \Rightarrow uxv$ angegeben. Für die in Abbildung 4.1-1 angegebene Ausdrucksgrammatik ist z. B. $(A) \Rightarrow (A + F)$ ein Ableitungsschritt unter Anwendung der Produktion p_2. Mehrere Ableitungsschritte nacheinander angewandt notieren wir durch $\Rightarrow^*$, z. B. in $(A) \Rightarrow^* (b + b)$. Wird in jedem Ableitungsschritt in $u \Rightarrow^* v$ jeweils das am weitesten links stehende Nichtterminal ersetzt, so sprechen wir von einer *Linksableitung*, $u \Rightarrow^L v$. Rechtsableitungen sind entsprechend definiert.

Ableitung

Terminale	T = {b, +, *, (,)}
Nichtterminale	N = {E, A, F, B}
Zielsymbol	S = E
Produktionen	P = {p_1, ..., p_7}

p_1: E ::= A
p_2: A ::= A + F
p_3: A ::= F
p_4: F ::= F * B
p_5: F ::= B
p_6: B ::= (A)
p_7: B ::= b

Abb. 4.1-1: Kontextfreie Grammatik für Ausdrücke.

Sprache

Ein Satz aus der Sprache $L(G)$ einer kontextfreien Grammatik G ist eine Folge terminaler Symbole, die aus dem Zielsymbol ableitbar ist:

$L(G) = \{ w \mid w \in T^* \text{ und } S =>^* w \}$

Im folgenden setzen wir voraus, daß G *reduziert* ist, d. h. jedes Zeichen $x \in V$ und jede Produktion $p \in P$ in der Ableitung mindestens eines Satzes aus $L(G)$ vorkommt.

Ableitungsbaum

Die Ableitung eines Satzes prägt diesem eine Struktur auf, die wir als *Ableitungsbaum* darstellen: Seine Wurzel ist das Startsymbol, seine Blätter sind die Terminale des Satzes. Ein innerer Knoten repräsentiert ein Nichtterminal und zusammen mit seinen unmittelbaren Nachfolgern die Anwendung einer Produktion. Abbildung 4.1-2 zeigt einen Ableitungsbaum für $b * (b + b)$ zu der Grammatik aus Abbildung 4.1-1. Die Nichtterminalknoten sind zusätzlich mit der jeweils angewandten Produktion markiert.

Gibt es zu einem Satz aus $L(G)$ verschiedene Ableitungsbäume (bzw. verschiedene Links- oder Rechtsableitungen), so ist die Grammatik *mehrdeutig*.

Für die Notation von kontextfreien Grammatiken sind eine Reihe von Konventionen und Vereinfachungen gebräuchlich: Meist gibt man nur die Produktionen an. Alle Zeichen, die auf der linken Seite einer Produktion vorkommen, sind Nichtterminale, alle übrigen Terminale. Das Zielsymbol kommt meist nicht auf der rechten Seite einer Produktion vor oder wird anderweitig aus dem Kontext bestimmt. Produktionen zum gleichen Nichtterminal faßt man zu einer Produktion mit alternativen rechten Seiten zusammen, z. B. p_2 und p_3 aus Abbildung 4.1-1 zu

A ::= A + F | F .

BNF

Diese Notation bezeichnet man auch als *Backus-Naur-Form (BNF)* nach den Autoren der ersten Anwendung einer kontextfreien Grammatik zur Definition einer Programmiersprache, Algol 60 in NAUR63.

EBNF

Weitere Vereinfachungen ergeben sich, wenn man für die rechte Seite von Produktionen beliebig geschachtelte Konstrukte regulärer Ausdrücke erlaubt:

x^+	nicht leere Folge von x,
x^*	Folge von x oder leer,
{x}	wie x^*,
x ‖ t	nicht leere Folge von x, getrennt durch t,
[x]	x oder leer,
x | y	Alternative,
(x)	Klammerung.

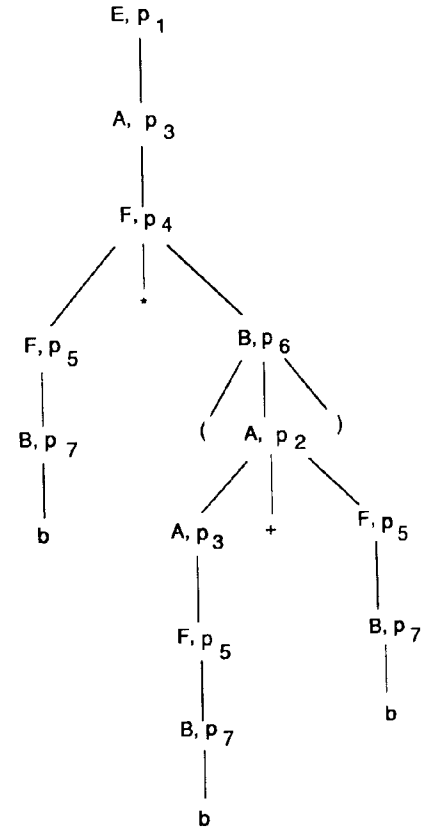


Abb. 4.1-2: Abteilungsbaum für b*(b+b).

Damit kann die Produktion .

A ::= A + F | F

auch als

A ::= F {+ F} oder A ::= F (+F)* oder A ::= F ‖ +

notiert werden. Bei Verwendung solcher Konstrukte sprechen wir von *Erweiterter BNF (EBNF)*. Um Terminale der Grammatik von den zur Notation von Produktionen verwendeten Zeichen zu unterscheiden, schließt man erstere häufig in Apostrophe ein, z. B.

A ::= F ('+' F) * .

Kontextfreie Grammatiken können auch durch Syntaxdiagramme (s. Kap. 3) anschaulich dargestellt werden, z. B. in der Sprachdefinition von Pascal JENS74.

Zerteilergrammatik

Ein *Zerteiler* für eine kontextfreie Grammatik *G* ist ein Algorithmus, der zu einer Symbolfolge *s* die Ableitung gemäß *G* berechnet oder feststellt, daß *s* nicht zur Sprache von *G* gehört. Die in der Definition einer Programmiersprache gegebene Grammatik kann meist nicht unmittelbar als Grammatik zur Zerteilerkonstruktion verwendet werden. Aus folgenden Gründen sind Transformationen notwendig oder zweckmäßig:

- Mehrdeutigkeiten werden entfernt. Die damit ausgedrückten verschiedenen Konstrukte werden in der semantischen Analyse unterschieden (z. B. Variablenbezeichner und parameterlose Funktionsaufrufe als Operanden). Weitergehende Transformationen können notwendig sein, um die Grammatik an die Bedingungen des angewandten Analyseverfahrens anzupassen.
- Syntaktische Restriktionen, die einfacher in der semantischen Analyse geprüft werden können, werden entfernt (z. B. Einschränkung der möglichen Typen von Reihungselementen).
- Kettenproduktionen, die nur der besseren Verbalisierung der Sprachdefinition dienen, werden entfernt, um die Analyse zu beschleunigen.

Diese Transformationen dürfen die Sprache gegenüber der ursprünglichen Grammatik nicht einschränken; eventuelle Erweiterungen müssen durch Prüfungen in der semantischen Analyse ausgeglichen werden.

Konkrete Syntax

Die so hergestellte Grammatik beschreibt die *konkrete Syntax* für den Zerteiler. Ein Ableitungsbaum zu einer solchen Grammatik ist für die nachfolgende semantische Analyse jedoch unnötig redundant: Er enthält Terminale als Blätter, die keine semantische Bedeutung tragen (z. B. `begin`, `end`, `;`), und Ableitungsschritte, die für die weitere Analyse nicht relevant sind. Wir konstruieren den Zerteiler deshalb so, daß er bei der Berechnung einer Ableitung zur konkreten Syntax nicht den zugehörigen Ableitungsbaum, sondern einen kompakteren *Strukturbaum* erzeugt. Er kann als Vergröberung des Ableitungsbaumes aufgefaßt werden.

Abstrakte Syntax

Strukturbäume werden ebenfalls durch eine kontextfreie Grammatik, die *abstrakte Syntax*, spezifiziert. Man entwickelt sie durch systematische Vergröberung der konkreten Syntax. Abbildung 4.1-3 zeigt eine abstrakte Syntax zur Grammatik aus Abbildung 4.1-1. Sie ist wie folgt aus der konkreten Syntax entstanden:

ap_1 : E ::= E Opr E
ap_2 : E ::= b
ap_3 : Opr ::=
ap_4 : Opr ::=

Abb. 4.1-3: Abstrakte Syntax für Ausdrücke.

- Die semantisch bedeutungslosen Klammern in p_6 sind weggelassen.
- Die Produktionen p_1 , p_3 , p_5 und p_6 mit nur einem Nichtterminal auf der rechten Seite (*Kettenproduktionen*) sind weggelassen und die beteiligten Nichtterminale *E, A, F, B* auf ein einziges, *E*, abgebildet.
- Die Produktionen p_2 und p_3, die sich nur in dem Operatorzeichen unterscheiden, sind zusammengefaßt und ein Nichtterminal *Opr* für die Operatoren eingeführt. In realistischen Grammatiken existiert dies meist schon, z. B. *AddOpr* ::= + | - , *MulOpr* ::= * | /.
- Die Terminale der Operatoren können in der abstrakten Syntax weggelassen werden. Sie werden durch die Produktionen ap_3 für + und ap_4 für * unterschieden.

Abbildung 4.1-4 zeigt den Strukturbaum für `b * (b + b)` zu dieser abstrakten Syntax. Die Terminale für `b` verbleiben in der Syntax und im Baum, da sie semantisch relevante Symbolattribute (Bezeichneridentität) tragen.

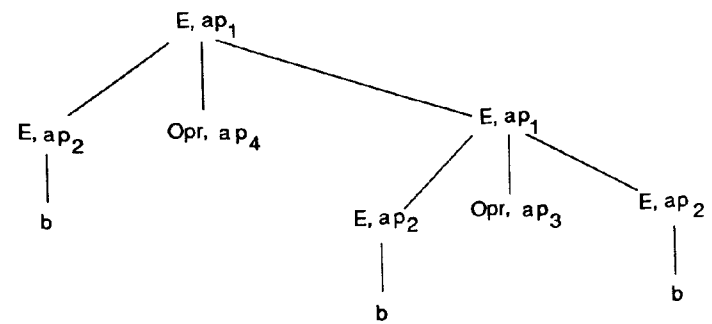


Abb. 4.1-4: Strukturbaum für b*(b+b).

Aufbau des Strukturbaums

Der Strukturbaum wird durch Aktionen aufgebaut, die der Zerteiler beim Herstellen der Ableitung anstößt:

- Akzeptiert er ein semantisch relevantes Terminal, so wird ein Blattknoten erzeugt (*Symbolanknüpfung*).

• Wendet er eine Produktion an, zu der es in der abstrakten Syntax einen Repräsentanten gibt, wird ein entsprechender innerer Knoten erzeugt (*Strukturanknüpfung*).

Die Reihenfolge, in der der Zerteiler durch Ausführen der Anknüpfungen die Knoten erzeugt, wird durch das Analyseverfahren und die Einbettung der Anknüpfungen bestimmt: Ein quellbezogener Zerteiler erzeugt den Baum in Postfixlinearisierung, ein zielbezogener Zerteiler erzeugt ihn je nach Integration der Anknüpfungen in Präfix- oder Postfixlinearisierung.

Die Implementierung der Strukturbaumschnittstelle kann (wie in Kapitel 2 allgemein diskutiert) nach verschiedenen Techniken erfolgen. Sie unterscheiden sich in der Implementierung der vom Zerteiler aufgerufenen Anknüpfungsprozeduren:

1) Es wird jeweils ein Knoten als Datenobjekt gebildet und mit Hilfe eines Kellers die Datenstruktur des Baumes aufgebaut.
2) Die einer Produktion zugeordneten Aktionen der semantischen Analyse (Attributierung) werden unmittelbar ausgeführt. (Attributwerte werden in Variablen, Kellern und speziellen abstrakten Datentypen gespeichert.)
3) Die Knoten werden in einen Puffer (oder eine Datei) ausgegeben. Die semantische Analyse liest sie und verfährt wie in (1) oder (2).

Methode (2) ist die effizienteste, da der Baum weder linearisiert noch als Datenstruktur explizit aufgebaut wird. Er existiert nur konzeptionell in der Aufruffolge für die Knoten. Sie ist selbstverständlich nur dann anwendbar, wenn die Attributierung in dieser durch den Zerteiler vorgegebenen Reihenfolge durchführbar ist.

4.2 Zielbezogene Zerteiler

Rekursive Prozeduren

Ein *zielbezogener Zerteiler* kann am einfachsten durch einen rekursiven Algorithmus zu einer kontextfreien Grammatik G beschrieben werden: Zu jedem Nichtterminal X enthält er eine Prozedur, die alle aus X ableitbaren Symbolfolgen akzeptiert. Ihr Rumpf enthält für jede Produktion mit X auf der linken Seite $X ::= x_1 \ldots x_n$ eine Operationsfolge. Bei Aufruf von X wird eine dieser Alternativen ausgewählt. Für jedes Zeichen x_i der rechten Seite der Produktion enthält die Operationsfolge eine der beiden folgenden Operationen:

$x_i \in$ T: Akzeptiere das nächste Zeichen der Eingabe falls es gleich x_i ist, sonst liegt ein Fehler vor.

$x_i \in$ N: Rufe die Prozedur für x_i auf.

Ein Aufruf der Prozedur für das Startsymbol startet den Zerteiler. Wir geben unten an, unter welchen Bedingungen ein so konstruierter Zerteiler die Entscheidungen über die anzuwendende Produktion immer eindeutig treffen kann.

Entscheidungssituation

Ein solcher Zerteiler konstruiert eine Linksableitung für jeden Satz aus der Sprache von G: Sei

$$S \Rightarrow^{L} u\,A\,v_m\,v_{m-1} \ldots v_o \Rightarrow u\,x\,v_m\,v_{m-1} \ldots v_o \Rightarrow^{*} uy$$

eine Ableitung des Satzes $uy \in L(G)$. Der Zerteiler habe schon aus S den Anfang einer Linksableitung konstruiert, in der A das am weitesten links stehende Nichtterminal ist. Die Terminalfolge u ist der Anfang der Eingabe, der schon akzeptiert wurde. Der Zerteiler hat dabei die Produktionen

$$S \rightarrow w_o\,A_o\,v_o,\ A_o \rightarrow w_1\,A_1\,v_1,\ \ldots,\ A_{m-1} \rightarrow w_m\,A\,v_m$$

angewandt und alle w_i auf Teile von u reduziert. Im weiteren Verlauf wird $A\,v_m \ldots v_o$ auf den Rest der Eingabe y abgeleitet. Der Rekursionskeller des Zerteilers enthält Positionen in den Operationsfolgen, welche die noch abzuarbeitenden Enden v_i der noch nicht vollständig abgearbeiteten Produktionen angeben. Im nächsten Schritt entscheidet der Zerteiler, welche Produktion für das am weitesten links stehende Nichtterminal A anzuwenden ist, hier $A ::= x$. Abbildung 4.2-1 zeigt die hier beschriebene Situation graphisch. Der linke, stark umrandete Teil des Ableitungsbaumes repräsentiert die soweit konstruierte Linksableitung.

Gibt es mehrere Produktionen für A, etwa $A ::= x$ und $A ::= x'$, so entscheidet der Zerteiler anhand der nächsten k Zeichen der Eingabe, welche anzuwenden ist.

Wir wollen hier zunächst die einfachste und am meisten angewandte Klasse für zielbezogene Zerteiler betrachten. Sie verwenden für die zu treffenden Entscheidungen nur das jeweils nächste Zeichen der Eingabe ($k = 1$). Für einen solchen Zerteiler bestimmen wir zu jeder Produktion $A ::= x$ eine Menge von Terminalsymbolen $E \subseteq T$ als *Entscheidungsmenge* (*director set*). Steht eine Ableitung von A an, so wird diejenige Produktion für A angewandt, in deren Entscheidungsmenge das nächste Eingabesymbol liegt.

Anfangs- und Folgemengen

Zur Berechnung der Entscheidungsmengen benötigen wir die beiden folgenden Funktionen. Zur späteren Verallgemeinerung geben wir sie für beliebige k an:

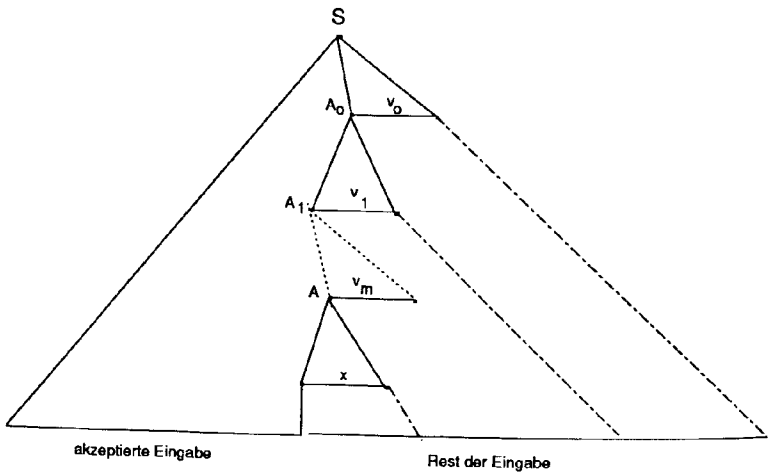


Abb. 4.2-1: Entscheidungssituation eines zielbezogenen Zerteilers.

Die Menge der terminalen Anfänge einer Zeichenfolge $w \in V^+$ ist

$$Anf_k(w) = \{ v \mid v \in T^+, |v| = k, w =>^* v u \} \cup \{ v \mid v \in T^*, |v| < k, w =>^* v \},$$

wobei |v| die Länge (Anzahl der Symbole) von v angibt. Man beachte, daß $Anf_k(w)$ auch Symbolfolgen enthalten kann, die kürzer als k sind, insbesondere kann $Anf_k(w)$ auch die leere Folge ε enthalten.
Die Menge der terminalen Folgen eines Zeichens $x \in V$ ist

$$Folge_k(x) = \{ v \mid v \in T^*, |v| \leq k, S =>^* u x w, v \in Anf_k(w) \}.$$

Sie gibt an, welche Symbolfolgen auf ein x in einer Ableitung folgen können. Für $k = 1$ schreiben wir abkürzend $Anf(w)$ bzw. $Folge(x)$.
Damit können wir für eine Produktion p: $A ::= x$ die Entscheidungsmenge zu $E_p = Anf(w)$ bestimmen, falls x nicht auf ε ableitbar ist.
Allgemein gilt

$$E_p = Anf(x\ Folge(A)).$$

Dabei steht $x\ Folge(A)$ für die Zusammensetzung von x mit jedem Element aus $Folge(A)$. Die Entscheidungen des Zerteilers können immer dann eindeutig getroffen werden, wenn die E_p für Produktionen mit

gleicher linker Seite disjunkt sind. Hieraus können wir eine Anforderung an die Grammatik formulieren:

Starke LL(k)-Bedingung: Eine kontextfreie Grammatik heißt *stark LL(k)*, wenn für je zwei Produktionen $A ::= x$ und $A ::= x'$ gilt Stark LL(k)

$$Anf_k(x\ Folge_k(A)) \cap Anf_k(x'\ Folge_k(A)) = \emptyset.$$

Falls die Grammatik keine ε-Produktionen ($B ::= \varepsilon$) enthält und $k = 1$ gilt, kann auf die Berechnung der Folgemengen verzichtet werden.

Diese Bedingung ist eine Verschärfung der allgemeinen LL(k)-Bedingungen, die in ROSE70 eingeführt wurden. Die Prüfung der starken LL(k)-Eigenschaft und die Konstruktion des Automaten anhand der Entscheidungsmengen zu den Produktionen ist wesentlich einfacher als im Fall allgemeiner LL(k)-Grammatiken. Für den Fall k=1 sind beide Bedingungen gleichwertig. In der Praxis beschränkt man sich grundsätzlich auf Zerteiler, die anhand des nächsten Symbols der Eingabe ihre Entscheidung treffen, und wendet obige Bedingung für starke LL(1)-Grammatiken an.

		Entscheidungsmengen E_p
p_1	Prog ::= Block '#'	begin
p_2	Block ::= 'begin' Decls Stmts 'end'	begin
p_3	Decls ::= Decl ';' Decls	new
p_4	Decls ::=	Id
p_5	Decl ::= 'new' Id	new
p_6	Stmts ::= Stmts ';' Stmt	begin Id
p_7	Stmts ::= Stmt	begin Id
p_8	Stmt ::= Block	begin
p_9	Stmt ::= Id ':=' Id	Id

	Anfangsmengen	Folgemengen
Prog:	begin	ε
Block:	begin	# ; end
Decls:	ε new	begin Id
Decl:	new	;
Stmts:	begin Id	; end
Stmt:	begin Id	; end

Abb. 4.2-2: Beispiel für Anfangs-, Folge- und Entscheidungsmengen.

Abbildung 4.2-2 zeigt ein Beispiel für eine Grammatik, die die starke LL(1)-Bedingung nicht erfüllt. Zu den Nichtterminalen sind Anfangs- und Folgemengen und zu den Produktionen die Entscheidungsmengen angegeben. Für die Produktionenpaare p_3 , p_4 sowie p_8 , p_9 sind die E_p disjunkt, für p_6 , p_7 jedoch nicht.

Linksrekursion

Grammatiken verletzen häufig die LL(k)-Bedingungen, weil bestimmte Produktionsmuster darin auftreten. Man kann solche Produktionen systematisch transformieren. Die Ursache für den Entscheidungskonflikt in obigem Beispiel ist die linksrekursive Produktion p_6.

Eine LL(k)-Grammatik enthält keine linksrekursiven Produktionen !

Dies gilt auch für indirekte Linksrekursion mit $A ::= B\ u$ und $B =>^* A\ v$.

Ersetzen wir die Produktionen p_6 und p_7 durch

p_{10}	Stmts	::= Stmt X
p_{11}	X	::= ';' Stmt X
p_{12}	X	::= ε

mit dem neuen Nichtterminal X, dann sind die Entscheidungsmengen für p_{11} und p_{12} disjunkt und der Konflikt ist beseitigt. Das allgemeine Schema zur Transformation direkter Linksrekursion lautet

A ::= Au | v wird transformiert in
A ::= vX, X ::= uX | ε.

Dabei werden alle Produktionen für A zunächst in der ersten Form, eventuell unter Verwendung von EBNF-Konstrukten für u und v zusammengefaßt. Nach der Transformation entfernt man die EBNF-Konstrukte wieder. Nach dem gleichen Schema transformiert man linksrekursive Produktionen in Ausdrucksgrammatiken wie A ::= A '+' F | F aus Abbildung 4.1-1.

Linksfaktorisieren

In obigem Beispiel hätte auch eine rechtsrekursive Formulierung von p_6 *Stmts* ::= *Stmt* ';' *Stmts* zum Konflikt geführt, da ihr Anfang mit dem von p_7 übereinstimmt. Produktionen der Form

$$A ::= v\ u \mid v\ w$$

führen ebenfalls zu LL(k)-Konflikten. Man behebt sie durch *Linksfaktorisieren* des gemeinsamen Anfangs der Alternativen .

$$A ::= v\ X \quad \textit{und}\ X ::= u \mid w$$

mit dem neuen Nichtterminal X.

Rekursiver Abstieg

Die meistverwendete Technik zur Implementierung für LL(1)-Zerteiler basiert auf dem anfangs beschriebenen Satz rekursiver Prozeduren. Sie wird als *rekursiver Abstieg* (*recursive descent*) bezeichnet. Abbildung 4.2-3 zeigt zwei solche Prozeduren für das Beispiel in Abbildung 4.2-2. In Prozeduren zu Nichtterminalen mit mehreren alternativen Produktionen (wie *Decls*) wird die Entscheidung durch eine Fallunterscheidung für das anstehende Symbol der Eingabe über die Elemente der Entscheidungsmengen getroffen. (Anstelle der in Abbildung 4.2-3 angegebenen terminalen Zeichenreihen werden die Syntaxcodes der Symbole eingesetzt.)

```
procedure Block;
begin                  (* Anfang p2 *)
      akzeptiere ('begin');
      Decls;
      Stmts;
      akzeptiere ('end');
                         (* Ende p2 *)
end;

procedure Decls;
begin
      if symbol = 'new'
      then begin  (* Anfang p3 *)
                Decl;
                akzeptiere (';')
                Decls;
                     (* Ende p3 *)
      end else ;     (* leer p4 *)
end;
```

Abb. 4.2-3: Prozeduren eines Zerteilers mit rekursivem Abstieg.

Strukturbaum

An jeder beliebigen Stelle der Operationenfolge zu einer Produktion können Aktionen eingefügt werden, die einen Strukturbaum aufbauen oder semantische Attribute berechnen. Gibt man z. B. die Produktionskennung am Anfang der Operationenfolge aus, so erscheinen die bei der Ableitung angewandten Produktionen in Postfixanordnung (die Produktionskennung eines Baumknotens steht vor denen seiner Unterbäume); bei Ausgabe am Ende wird Präfixanordnung erzeugt (die Produktionskennung eines Baumknotens steht nach denen seiner Unterbäume).

Diese Technik des rekursiven Abstiegs kann man nach Prüfung der LL(1)-Eigenschaft systematisch auch ohne Verwendung generierender Werkzeuge anwenden.

EBNF im rekursiven Abstieg

Das Schema läßt sich auch auf Produktionen mit EBNF-Konstrukten erweitern: So hätte in unserem Beispiel die Produktion p_2 auch

```
Block ::= 'begin' {Decl ';'} Stmt {';' Stmt} 'end'
```

lauten können. In der Prozedur setzt man dann für die Wiederholungskonstrukte Schleifen ein:

```
while Symbol = 'new' do begin Decl; akzeptiere (';') end;
```

Hier wird *Anf* (*Decl* ';') als Entscheidungsmenge in der Schleifenbedingung verwendet. Anhand solcher Mengen prüft man eine entsprechend erweiterte LL(1)-Bedingung, um die Eindeutigkeit der Entscheidungen sicherzustellen. Die Verwendung von EBNF-Konstrukten vermeidet häufig konfliktträchtige Rekursionen und deren Transformation (wie im Falle von *Stmts*).

Implementierungstechniken

Bei der Technik des rekursiven Abstiegs übernimmt der Laufzeitkeller mit den Rückkehradressen für Prozeduraufrufe die Rolle des Automatenkellers. Die Aufrufe dieser einfach strukturierten Prozeduren (ohne Parameter oder lokale Variablen) könnten auf das Kellern der Rückkehradresse und einen Unterprogrammsprung reduziert werden. Da diese Optimierung meist nicht vom Übersetzer der Implementierungssprache geleistet wird, kann man Laufzeit einsparen und den Kellerumfang verringern, wenn man in der Zerteilerimplementierung die Rekursion in eine Schleife auflöst und den Keller explizit verwaltet. Dazu werden alle Programmstücke zu Produktionen und die Stellen nach einem Aufruf mit einer Marke versehen. Anstelle eines Aufrufs wird die nachfolgende Marke gekellert und über die Entscheidungsmenge auf eine Produktion verzweigt. Am Ende einer Produktion erfolgt ein Sprung auf die entkellerte Marke. (Erlaubt die Implementierungssprache Marken nicht als Datenobjekte, so verwendet man statt dessen geeignete Codierungen und Sprungverteiler.)

Ein LL(1)-Zerteiler kann auch als Kellerautomat implementiert werden, der eine Übergangstabelle interpretiert. Die Tabelle beschreibt eine Übergangsfunktion, die Paare von Zuständen und Eingabesymbolen auf eine Operation abbildet. Die Zustände entsprechen den Positionen vor bzw. nach den Elementen der Operationenfolge der Produktionen. Die Operationen sind:

- Akzeptiere das nächste Symbol und gehe in den Nachfolgezustand über.
- Kellere den Nachfolgezustand und gehe in den Anfangszustand einer Produktion (entspricht dem Aufruf).
- Entkellere einen Zustand als aktuellen Zustand.
- Halte an; die Eingabe ist akzeptiert.
- Zeige einen Fehler an.

Durch aufeinanderfolgende Codierungen für Nachfolgezustände kann man den Umfang der Tabellenelemente reduzieren. Für die Herstellung der Tabellen benötigt man insbesondere bei größeren Grammatiken generierende Werkzeuge. Wegen der Interpretation der Tabelle ist die Geschwindigkeit so implementierter Zerteiler meist geringer als die der Implementierungen nach der Technik des rekursiven Abstiegs insbesondere mit direkt programmiertem Keller.

4.3 Quellbezogene Zerteiler

In diesem Abschnitt stellen wir die Grundlagen von quellbezogenen LR(k)-Zerteilern vor. Sie dienen im wesentlichen dazu, den Einsatz von Zerteilergeneratoren für diese Methoden verständlich zu machen. Im Gegensatz zu den LL(k)-Verfahren ist die manuelle Implementierung solcher Zerteiler für realistische Grammatiken nicht praktikabel. Da LR(k)-Zerteiler für größere Grammatikklassen konstruierbar sind, erspart diese Zerteilungsmethode häufig Grammatiktransformationen und hat deshalb im Übersetzerbau große Bedeutung (siehe Abschnitt 4.5). LR-Verfahren wurden ursprünglich von Knuth in KNUT65 vorgestellt, erlangten aber erst später mit Einführung der SLR- und LALR-Techniken (DERE69, DERE71) praktische Bedeutung.

Quellbezogene Zerteiler konstruieren den Ableitungsbaum von den Blättern (der Eingabe) zur Wurzel hin und bilden dabei die Rechtsableitung in umgekehrter Reihenfolge, d. h. der Ableitungsbaum wird in Postfixreihenfolge erzeugt.

Entscheidungssituation

Abbildung 4.3-1 zeigt die Entscheidungssituation eines quellbezogenen Kellerautomaten bei Anwendung der Produktion $A ::= x$. Die bisher akzeptierte Eingabe ist zu ux reduziert und wird durch den Kellerinhalt repräsentiert. Mit weiteren k Symbolen Vorschau aus der Eingabe entscheidet der LR(k)-Automat, ob

x zu A reduziert wird,
x' zu B reduziert wird, mit $ux = u'x'$ oder
weitergelesen und später x' zu B reduziert wird mit $u\,x\,w = u'\,x'\,w'$.

Die folgende Bedingung garantiert, daß diese Entscheidung immer eindeutig getroffen werden kann:

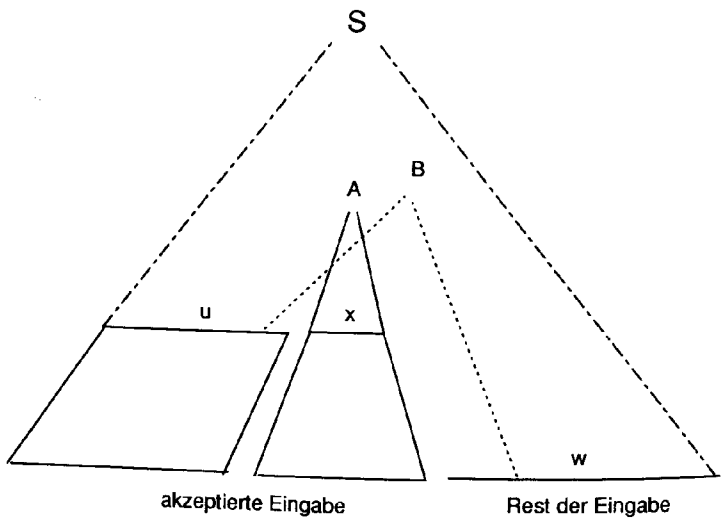


Abb. 4.3-1: Entscheidungssituation eines quellbezogenen Zerteilers.

LR(k)-Grammatik

LR(k)-Bedingung: Eine kontextfreie Grammatik ist eine *LR(k)-Grammatik*, wenn für je zwei Rechtsableitungen

$$Z \Rightarrow^* u\,A\,w \Rightarrow u\,x\,w$$
$$Z \Rightarrow^* u'\,B\,w' \Rightarrow u'\,x'\,w' = u\,x\,v$$

mit $u, u', x, x' \in V^*$ und $w, w', v \in T^*$

gilt: aus $Anf_k(w) = Anf_k(v)$ folgt $u = u'$, $A = B$, $x = x'$.

Mit akzeptiertem ux und k Symbolen als Vorschau von w ist also der nächste Schritt der Rechtsableitung eindeutig bestimmt. Zum Verständnis dieser Bedingung müssen wir beachten, daß die Rechtsableitung in umgekehrter Reihenfolge gebildet wird, also in den Schritten der LR(k)-Bedingung von rechts nach links. Der Kenntnisstand des Automaten ist also ux im Keller und k Symbole von w. Entweder reduziert er x zu A oder er gliedert $u\,x\,v$ in $u'\,x'\,w'$ auf und reduziert x' zu B. Falls dabei x' noch nicht vollständig gelesen ist, liest er weiter und reduziert später zu B.

Situationen

LR-Zerteiler sind Kellerautomaten, die wie LL-Zerteiler Zustände kellern. Die durch einen Zustand repräsentierte Information erläutern wir am Vergleich mit den LL-Automaten. Im LL(1)-Fall beschreibt ein Zustand die aktuelle Analyseposition innerhalb der Operationenfolge zu einer Produktion. Diese Information wird durch eine *Situation (item)*

$$[A ::= u \bullet v] \quad \text{mit } u, v \in V^*$$

abstrahiert. Der Punkt gibt an, daß u in der Produktion $A ::= uv$ vollständig abgeleitet und v noch nicht abgeleitet ist. Für den allgemeinen LL(k)-Automaten, den wir im vorigen Abschnitt nicht besprochen haben, kommt als weiteres unterscheidendes Merkmal der *erwartete Rechtskontext* hinzu: Rechtskontext

$$[A ::= uv \bullet R]$$

R ist eine Menge von Zeichenreihen höchstens der Länge k. Nach vollständigem Abarbeiten der Produktion muß der Rest der Eingabe mit einem Element aus R beginnen. R repräsentiert Klassen von Kellerinhalten des Automaten, also begonnenen, aber noch nicht abgeschlossenen Ableitungen von Produktionen.

Im Gegensatz zum LL-Automaten bildet ein LR-Automat den Ableitungsbaum von den Blättern zur Wurzel hin. Er entscheidet über die Anwendung einer Produktion erst, wenn ihre rechte Seite vollständig akzeptiert ist, wie Abbildung 4.3-1 verdeutlicht. Deshalb kann ein Zustand nicht einer einzigen Situation einer Produktion zugeordnet werden. Er repräsentiert vielmehr eine *Menge von Situationen,* deren Produktionen parallel verfolgt werden. In Abbildung 4.3-2 sind die Zustände mit der Übergangsfunktion als Graph eines LR(1)-Automaten für eine Beispielgrammatik angegeben. Die Rechtskontextmengen der Situationen sind in diesem Beispiel alle einelementig. LR-Zustand

Bevor wir die Konstruktion eines solchen Automaten beschreiben, erläutern wir seine Arbeitsweise. Sei q der aktuelle Zustand, dann führt er eine der folgenden Operationen gemäß der Übergangsfunktion aus: Übergänge

- *Lesen (shift)* eines Zeichens $v \in V$. q enthält dann eine Situation der Form $[A::=u \bullet x v\ R]$. q wird gekellert; der neue aktuelle Zustand ist der Nachfolger von q unter x. Ist $x \in T$, so wird das in der Eingabe anstehende Zeichen x akzeptiert. Bei $x \in N$ bleibt die Eingabe unverändert, siehe (2).
- *Reduzieren (reduce)* einer Produktion $p \in P$. q enthält dann eine Situation der Form $[A::=uv \bullet R]$ und das nächste Symbol der Eingabe ist aus R. Es werden so viele Zustände entkellert, wie die rechte Seite $u\,v$ Zeichen enthält. Der dann oberste Kellerzustand wird aktueller Zustand und von dort ein Übergang mit A ausgeführt, siehe (1). Falls p die Startproduktion ist, hält der Automat an.

Ist keiner der beiden Übergänge möglich, hat der Automat einen syntaktischen Fehler erkannt. Diese typischen Übergänge haben auch zu der Bezeichnung *Shift-Reduce-Zerteiler* geführt. Abbildung 4.3-3

Grammatik:
p_1: B ::= '(' D ';' S ')'
p_2: D ::= D ';' a
p_3: D ::= a
p_4: S ::= b ';' S
p_5: S ::= b

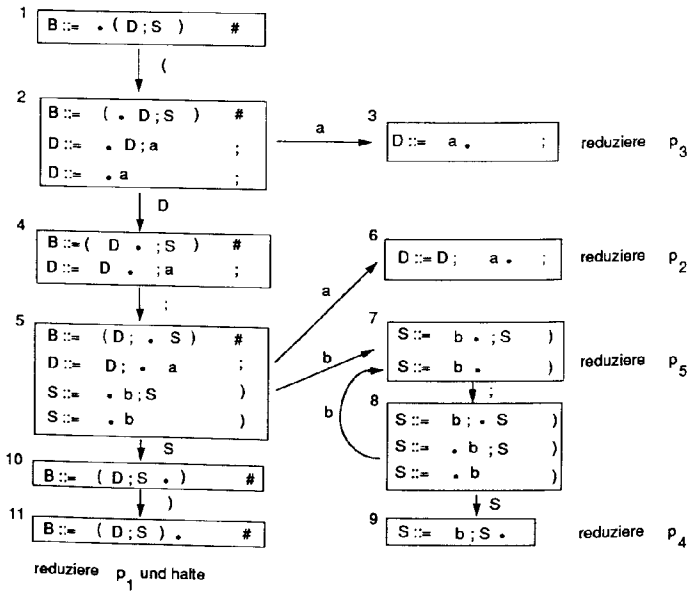


Abb. 4.3-2: Zustände und Übergangsfunktion eines LR(1)-, LALR(1) und SLR(1)-Automaten.

zeigt den Ablauf des Automaten aus Abbildung 4.3-2 für ein kleines Beispiel.

Automaten-konstruktion

Algorithmen zur Konstruktion von LR(k)-Automaten berechnen Zustände und Übergangsfunktion, so daß folgende Eigenschaften gelten:

1) Enthält ein Zustand q eine Situation $[A ::= u \bullet X v\ R]$ mit $X \in N$, und seien $X ::= w_1, \ldots, X ::= w_n$ alle Produktionen für X, so enthält q auch die Situationen $[X ::= \bullet w_1\ R'], \ldots, [X ::= \bullet w_n\ R']$ mit $R' = Anf_k(v\ R)$. Diese *Hüllenbildung* korrespondiert zu dem Verhalten des Automaten, das ihn zunächst eine Produktion für X abarbeiten läßt und ihn nach Reduktion von X wieder in q zurückführt (vgl. Zustände 2, 5, 8 in Abb. 4.3-2).

Keller	Eingabe	Reduktion
1	(a; a; b; b) #	
1 2	a; a; b; b) #	
1 2 3	; a; b; b) #	p_3
1 2	; a; b; b) #	
1 2 4	; a; b; b) #	
1 2 4 5	a; b; b) #	
1 2 4 5 6	; b; b) #	p_2
1 2	; b; b) #	
1 2 4	; b; b) #	
1 2 4 5	b; b) #	
1 2 4 5 7	; b) #	
1 2 4 5 7 8	b) #	
1 2 4 5 7 8 7	) #	p_5
1 2 4 5 7 8	) #	
1 2 4 5 7 8 9	) #	p_4
1 2 4 5	) #	
1 2 4 5 10	) #	
1 2 4 5 10 11	#	p_1
1	#	

Abb. 4.3-3: Ablauf des LR-Automaten aus Abbildung 4.3-2.

2) Für jedes Zeichen $x \in V$, zu dem es in q eine Situation $[A ::= u \bullet x v\ R]$ gibt, existiert ein Übergang in einen Zustand q' unter x. Der Zustand q' enthält die dazu korrespondierende Situation $[A ::= u \bullet x v\ R]$.

3) Der Anfangszustand enthält die Situation $[S ::= \bullet w\ \#]$, wobei $S ::= w$ die eindeutige Startproduktion der Grammatik und # ein sonst nicht vorkommendes Schlußzeichen ist.
4) Enthält ein Zustand zwei Situationen $[A ::= u\bullet R_1]$ *und* $[B ::= v\bullet R_2]$ mit $A \neq B$ oder $u \neq v$ und $t \in R_1 \cap R_2$, so kann bei einer anstehenden Eingabe t nicht entschieden werden, nach welcher Produktion zu reduzieren ist. Es liegt ein sogenannter *Reduziere-reduziere-Konflikt* vor. Die Grammatik erfüllt die LR(k)-Bedingung nicht.
5) Enthält ein Zustand eine Situation $[A ::= u\bullet R]$ und gibt es von q einen Übergang unter $t \in Anf(R)$, so kann bei einer anstehenden Eingabe t nicht entschieden werden, ob zu lesen oder zu reduzieren ist. Es liegt ein Lies-reduziere-Konflikt vor. Die Grammatik erfüllt die LR(k)-Bedingung nicht.

Zur Konstruktion des Automaten beginnt man mit dem Startzustand und bildet nach obigen Regeln Zustände und Übergänge, bis der Automat vollständig ist.

Die Automaten werden im allgemeinen durch einen Algorithmus implementiert, der eine Übergangstabelle interpretiert. Selbstverständlich dient die den Zuständen zugeordnete Information, die Menge der Situationen, nur der Automatenkonstruktion; im Automaten werden die Zustände numerisch codiert. Die Übergangsfunktion wird entweder durch eine Matrix (Zustände × Grammatiksymbole) repräsentiert, wobei Techniken zur Kompaktierung angewandt werden, oder man wählt eine Listenrepräsentation. Die Einträge in der Tabelle codieren die verschiedenen Übergänge:

lies und gehe in Zustand q
reduziere Produktion p
Fehler

Grammatikklassen SLR(1) und LALR(1)

Da für Grammatiken realistischer Programmiersprachen die Anzahl der Zustände eines LR(k)-Automaten schon bei k=1 sehr groß ist, wendet man Verfahren an, die zu engeren Grammatikklassen Automaten mit geringeren Zustandszahlen liefern. Die beiden wichtigsten Verfahren, SLR(1) und LALR(1), basieren auf der Zustandsmenge, die man bei der Konstruktion des LR(0)-Automaten erhält. In diesem Fall wird für die Situationen kein Rechtskontext betrachtet. Dadurch werden viele Zustände des LR(k)-Automaten, die sich nur im Rechtskontext unterscheiden, auf einen einzigen abgebildet. Aus den obigen Eigenschaften (4) und (5) (Konflikte) sieht man, daß der Rechtskontext nur zur Entscheidung bei Reduktionen herangezogen wird. Deshalb ist ein LR(0)-Automat genau dann konfliktfrei (und die Grammatik erfüllt die LR(0)-Bedingung), wenn Reduktionssituationen nur einzeln in einem Zustand auftreten. In dem Beispiel aus Abbildung 4.3-2 wären bei der LR(0)-

Konstruktion die gleichen Zustände entstanden. Die Zustände 3, 6, 9 und 11 sind solche LR(0)-Reduktionszustände. Zustand 7 verletzt jedoch die LR(0)-Bedingung. (Hätte man Produktion 4 links- statt rechtsrekursiv formuliert, so wäre ein LR(0)-Automat entstanden.)

Um mächtigere Automaten zu erhalten, berechnet man bei der LALR(1)-Methode die Zustände wie im LR(0)-Automaten, ordnet ihren Situationen aber die Rechtskontexte zu, wie sie sich im LR(1)-Fall ergeben würden. Damit steht bei Reduktionen präzisere Entscheidungsinformation zur Verfügung. Im Falle unseres Beispiels aus Abbildung 4.3-2 hätte die LALR(1)-Methode den gleichen, konfliktfreien Automaten geliefert. In der Praxis reicht diese Automatenklasse für Zerteiler von Programmiersprachen aus. Sie ist mächtiger als die LL(1)-Klasse.

Das recht komplexe Verfahren zur Berechnung des Rechtskontextes wird in der SLR(1)-Methode vereinfacht: Man berechnet den LR(0)-Automaten und fügt den Reduktionssituationen $[A ::= u\bullet R]$ als Entscheidungsinformation den Rechtskontext $R = Folge(A)$ hinzu. Da diese Mengen größer oder gleich denen aus der LALR(1)-Konstruktion sind, liegt die Klasse der SLR(1)-Grammatiken zwischen LALR(1)- und LR(0)-Grammatiken. Unser Beispiel hätte auch nach dieser Methode den gleichen Automaten ergeben. Die größere Mächtigkeit der LALR(1)-Konstruktion wird jedoch vorteilhaft für Grammatiken realistischer Programmiersprachen genutzt, die erst umgeformt werden müßten, um die SLR(1)-Bedingung zu erfüllen.

Selbstverständlich unterscheiden sich die Methoden nur in der Konstruktion der Übergangsfunktion. Die Implementierungen der Zerteiler und die Repräsentation der Übergangsfunktion sind gleich.

4.4 Behandlung syntaktischer Fehler

Fehlerbehandlung

Wir haben bisher offengelassen, wie ein Zerteiler auf das Erkennen eines syntaktischen Fehlers reagiert. Als radikale Maßnahme könnte er den Fehler anzeigen, seine Arbeit einstellen und dadurch die Übersetzung vorzeitig beenden. Der Benutzer wäre dann gezwungen, jeden syntaktischen Fehler einzeln zu beheben und die Übersetzung erneut zu starten. Gerade in der Entwicklungsphase von Programmen ist solch ein häufiger Wechsel zwischen Edieren und Übersetzen lästig und zeitaufwendig. Wir betrachten deshalb hier Verfahren zur *Fehlerbehandlung* (error recovery), mit denen der Zerteiler nach Erkennen und Melden eines Fehlers weiterarbeiten kann.

Damit auch nachfolgende Analysephasen fortgesetzt werden können, muß die Ausgabe des Zerteilers, die Folge der Anknüpfungen, auch im Fehlerfall konsistent sein. Klare Schnittstellen zwischen dem Zerteiler und benachbarten Übersetzerphasen stellen weitere Anforderungen an die

Fehlerbehandlung: Schon akzeptierte Eingabe soll nicht zurückgesetzt und einmal angestoßene Anknüpfungen nicht zurückgenommen werden. Dazu muß der Fehler an der frühest möglichen Stelle erkannt werden. Von dort muß eine Fortsetzung gefunden werden, so daß der Rest der Eingabe analysiert wird, möglichst ohne dabei Folgefehler anzuzeigen. Weiter ist anzustreben, daß die Analyse korrekter Programme möglichst wenig durch Zeit- und Speicheraufwand der Fehlerbehandlung belastet wird. Verfahren mit anderer Zielsetzung, z. B. möglichst umfassende Analyse der Fehlersymptome, lassen wir hier außer Betracht.

Fehlerstelle

Maßnahmen der Fehlerbehandlung werden häufig als Versuche zur Korrektur fehlerhafter Programme mißverstanden. Daß sie dies im allgemeinen nicht leisten können, wird insbesondere an Klammerungsfehlern deutlich. Der Programmausschnitt

```
... a := b * (x+d; ...
```

enthält offensichtlich einen Fehler. Ob eine schließende Klammer fehlt oder die öffnende nicht beabsichtigt war, kann nur der Programmierer entscheiden. Ein Zerteiler, der die Eingabe von links nach rechts verarbeitet, erkennt nur das Symptom eines Fehlers, hier am Semikolon. Wir definieren deshalb die Fehlerstelle in diesem Sinne als Stelle im Programmtext, an der das Symptom erkennbar ist.

Wir nennen $w \in T^*$ ein *korrektes Präfix* der Sprache $L(G)$, wenn es ein $u \in T^*$ gibt, mit $wu \in L(G)$. Dann ist in einer Symbolfolge $wtx \in T^*$ Das Symbol $t \in T$ die erste *Fehlerstelle*, falls w ein korrektes Präfix, wt aber kein korrektes Präfix ist.

Damit ist in obigem Beispiel das Semikolon die Fehlerstelle.

Die in den vorigen Abschnitten betrachteten LL- und LR-Verfahren haben die gewünschte Eigenschaft, den Fehler an der Fehlerstelle zu erkennen. Sie akzeptieren w, haben keinen zulässigen Übergang mit t, führen keine weiteren Reduktionen (mit etwaigen Anknüpfungen) aus und stoßen die Fehlerbehandlung an.

Aufsetzpunkt

Ziel der Fehlerbehandlung ist es, den internen Zustand des Zerteilers durch Ausführen von Übergängen und Reduktionen so zu verändern, daß die Eingabe mit einem der nächsten auf w folgenden Symbole weiter analysiert werden kann. Die Maßnahmen der Fehlerbehandlung können wir als Änderung der Eingabe beschreiben, durch Löschen und Einfügen von Symbolen:

Der *Aufsetzpunkt* ist das erste Symbol der Eingabe ab der Fehlerstelle, das vom Zerteiler nach der Fehlerbehandlung akzeptiert wird.

Sei in der Eingabe $wtu = wydz\ t$ die Fehlerstelle und d der Aufsetzpunkt, so löscht die Fehlerbehandlung y und fügt dafür $v \in T^*$ ein, wobei wvd ein korrektes Präfix ist. Diese Änderung ist nicht als Fehlerkorrektur im Sinne der Programmentwicklung zu verstehen. Sie wird auch nicht tatsächlich an der Eingabe durchgeführt, vielmehr arbeitet der Zerteiler, als ob die Eingabe $wvdz$ gelautet hätte. Im obigen Beispiel könnte d = ; gewählt werden, $y = \varepsilon$ und v =) würde eingefügt.

Die drei folgenden Verfahren unterscheiden sich im wesentlichen durch die Methode zur Bestimmung des Aufsetzpunktes und die Operationen des Zerteilers, den Aufsetzpunkt zu erreichen.

Feste Aufsetzpunkte

Feste Aufsetzpunkte

Erkennt der Zerteiler beim Akzeptieren eines Nichtterminals A einen Fehler, so überliest er die nächsten Symbole aus der Eingabe bis zu einem Aufsetzpunkt d aus einer Symbolmenge D_A. Er setzt die Analyse so fort, als ob A akzeptiert wäre. Dies geschieht bei LL-Zerteilern mit rekursivem Abstieg durch Beenden des Aufrufs von A; bei LR-Zerteilern durch Entkellern bis der Zustand $[B::=u\bullet Av]$ enthält. D_A wird zu *Folge* (A) bestimmt und ggf. durch weitere Symbole heuristisch ergänzt (z. B. *Anf* (A), falls A in Listen mit Separatoren wie ; vorkommt). Bei diesen Verfahren ist die Folge der ausgeführten Anknüpfungen für die abgebrochene Analyse von A nicht konsistent.

Fehlerproduktionen

Fehlerproduktionen

Für einige wichtige Nichtterminale (z. B. Ausdruck, Anweisung, Deklaration) führt man in der Grammatik zusätzliche Fehlerproduktionen der Form

Anweisung ::= error

ein, wobei *error* ein fiktives Terminal ist. Man konstruiert den Zerteiler so, daß er im Fehlerfall entkellert bis ein Zustand erreicht ist, in dem ein Nichtterminal mit einer Fehlerproduktion akzeptiert wird (bei rekursivem Abstieg durch Beenden von Aufrufen). Er überliest Symbole der Eingabe solange bis ein akzeptabler Aufsetzpunkt erreicht ist, und setzt die Zerteilung mit dem Akzeptieren des Nichtterminals fort. Auch hier bleibt die Folge der Anknüpfungen für die entkellerten Zustände unvollständig.

Simulierte Fortsetzung

Simulierte Fortsetzung

Dieses Verfahren ist systematisch anwendbar und garantiert die Konsistenz der Anknüpfungen. Wegen seines hohen Konstruktionsaufwands ist es vorwiegend für automatisch generierte Zerteiler geeignet. Es wird in RÖHR80 ausführlich beschrieben. Wie im Falle der Fehler-

produktionenmethode wird die Menge D der möglichen Aufsetzpunkte dynamisch nach dem Erkennen eines Fehlers bestimmt. Da hierzu der gesamte Kellerinhalt als Information herangezogen wird, kann meist verhindert werden, daß große Programmstücke und wichtige Strukturierungssymbole überlesen werden. Die Fehlerbehandlung läuft in folgenden Schritten ab:

1) Der Inhalt des Zerteilerkellers wird gespeichert.
2) Die Analyse wird unter Kontrolle der Fehlerbehandlung simuliert fortgesetzt mit dem Ziel, möglichst schnell den Endzustand zu erreichen (d. h. den Keller zu leeren). Anstatt Symbole der Eingabe zu lesen, werden die zu akzeptierenden Symbole in die Menge D der möglichen Aufsetzpunkte eingefügt. Damit die Simulation terminiert, wendet man nur ausgezeichnete, nicht rekursive Produktionen der Grammatik an.
3) Die Eingabe wird bis zum nächsten Symbol $d \in D$ überlesen.
4) Die Zerteilung wird mit dem gespeicherten Kellerinhalt unter Kontrolle der Fehlerbehandlung fortgesetzt, um einen Kellerzustand herzustellen, in dem der Aufsetzpunkt akzeptabel ist. Dazu werden wieder nur die für (2) ausgezeichneten Produktionen angewandt und zu akzeptierende Symbole generiert statt gelesen.
5) Die Zerteilung wird normal fortgesetzt.

Die in (4) eingeführten Symbole können als zusätzliche Fehlerinformation für den Benutzer dienen. Der hier sicher erhebliche Laufzeitaufwand für die Fehlerbehandlung ist vertretbar, da er nur bei fehlerhaften Programmen anfällt.

4.5 Zerteilergeneratoren

Die in den vorigen Abschnitten beschriebenen systematischen Verfahren zur Herstellung eines Zerteilers werden auch in Zerteilergeneratoren angewandt. Solche Programmsysteme generieren ein Übersetzermodul aus einer kontextfreien Grammatik als Spezifikation. Für den Einsatz der Systeme zur Übersetzerkonstruktion sind folgende Eigenschaften von besonderer Bedeutung: Spezifikationsform der kontextfreien Grammatik, Analysemethode (meist LL(1) oder LALR(1)), Fehlerbehandlung und Schnittstellen der generierten Zerteiler. Im folgenden werden wir diese Aspekte diskutieren und dabei auf einige der zahlreichen, zum Teil weit verbreiteten Zerteilergeneratoren hinweisen. Leistungsaspekte generierter Zerteiler werden in WAIT85 diskutiert.

Spezifikationsform

Kontextfreie Grammatik

Die Spezifikation für einen zu generierenden Zerteiler ist eine kontextfreie Grammatik, ergänzt um Angaben zur Zerteilerschnittstelle (z. B.

Symbolcodierungen, Anknüpfungen) und zur Beeinflussung der Automatengenerierung (z. B. Konfliktlösung, Fehlerbehandlung). Abgesehen von kleineren notationellen Unterschieden (z. B. Schreibweise der Metasymbole) verlangen einige Generatoren strikte BNF, wie der unter Unix weit verbreitete Yacc (JOHN75), andere erlauben EBNF-Konstrukte in den Produktionen. Diese werden entweder direkt in Strukturen des Zerteileralgorithmus umgesetzt (wie in Abschnitt 4.2 für LL-Zerteiler gezeigt) oder in einer der Generierung vorgeschalteten Transformationsphase auf die strikte BNF zurückgeführt, z. B. in den Generatoren PGS (DENC84) und Lalr (GROS89). Werden dabei gleiche EBNF-Konstrukte (z. B. mehrfaches Auftreten von Stmt*) nicht identifiziert, sondern durch verschiedene neue Nichtterminale repräsentiert, so können Konflikte auftreten, die bei sorgfältiger Transformation vermeidbar wären. In solch einem Fall verzichtet man besser auf die Verwendung der EBNF.

Die Ausnutzung der EBNF führt meist zu kompakteren und damit übersichtlicheren Grammatiken. Allerdings ist die Zuordnung von Anknüpfungen hier schwieriger. Diese Form ist deshalb insbesondere dann vorteilhaft auszunutzen, wenn die Aufgabe des Zerteilers im wesentlichen auf die Analyse beschränkt ist und nur an wenigen Stellen Aktionen angestoßen werden. Für Zerteiler, die innerhalb eines Übersetzers einen vollständigen abstrakten Strukturbaum (tatsächlich oder konzeptionell) aufbauen, verwendet man im allgemeinen keine EBNF-Konstrukte in der Spezifikation.

Anknüpfungen

Die Anknüpfungen werden meist als Anweisungssequenzen in der Implementierungssprache des generierten Zerteilers den Produktionen zugeordnet. Manche Generatoren sehen auch einfache Formen einer Attributberechnung während der Syntaxanalyse vor. So kann z. B. für Yacc, PGS und Lalr in der Anknüpfung zu einer Produktion ein Wert als Ergebnis der Reduktion dem Symbol der linken Seite zugeordnet und auf Werte der Symbole der rechten Seite zugegriffen werden. Damit lassen sich z. B. der Baumaufbau im Übersetzer oder eine Ausdrucksauswertung in einem Tischrechnerprogramm vollständig beschreiben. Für weitergehende Attributierungen während der Syntaxanalyse (*parse time attribution*) fügt man Anknüpfungen innerhalb der rechten Seite von Produktionen ein. Falls die Eingabeform dies nicht erlaubt (z. B. Yacc), verwendet man statt dessen ε-Produktionen mit Anknüpfungen.

Analysemethode

Analysemethode

Die kontextfreie Grammatik muß je nach der im Generator verwendeten Analysemethode der LL(1)- bzw. LALR(1)-Bedingung genügen und ggf. entsprechend transformiert werden. Die Anforderungen an LL(1)-Grammatiken sind insbesondere für realistische Programmiersprachen recht schwierig zu erfüllen, während LALR(1)-Grammatiken, die die

LL(1)-Klasse echt enthalten, meist mit wenigen Anpassungen auskommen. In jedem Fall überprüft der Generator die Grammatikeigenschaften und zeigt ggf. Konflikte an.

Im Falle eines LL(1)-Generators, z. B. Llgen (TANE83), Deer (GRAY87) oder Palo (KAST89a), wendet man die im Abschnitt 4.2 gezeigten Transformationen zur Konfliktbeseitigung an. Dabei ist das Vermeiden von Linksrekursionen besonders störend: So führt z. B. die rechtsrekursive Beschreibung zweistelliger, linksassoziativer Operatoren nicht auf die intendierte natürliche, abstrakte Struktur von Ausdrücken. Man könnte sie durch geeignet plazierte Anknüpfungen erzeugen. Als Konsequenz wird auch die Attributierung solcher Strukturen schwieriger und weniger natürlich (z. B. zur Operatoridentifikation und Zwischencode-Erzeugung). Auch etwa notwendige Linksfaktorisierungen fassen unter Umständen semantisch unterschiedliche Strukturen syntaktisch zusammen und erschweren dadurch die weitere Verarbeitung. Wird die Sprache erst beim Entwurf der LL(1)-Grammatik festgelegt, so können eindeutige Anfänge der syntaktischen Konstrukte z. B. durch Einfügen von Wortsymbolen erzielt werden.

Im Falle von LALR(1)-Generatoren (Yacc, PGS, Lalr) treten auch bei üblichen Programmiersprachen nur wenige Konflikte auf, deren Ursachen nicht Mehrdeutigkeiten, sondern zu kurze Vorschau oder die Differenz zwischen LALR(1) und LR(1) sind. Zur Behebung solcher Konflikte muß man die Situationsmengen von Zuständen untersuchen. Eine übersichtliche Ausgabe der Zustände durch den Generator ist dafür hilfreich. Daran verfolgt man vom Konfliktzustand aus die Kette der Vorgängerzustände, um herauszufinden, weshalb die am Konflikt beteiligten Situationen zusammentreffen. Man strukturiert die Produktionen dann so um, daß die Konfliktsituationen getrennt werden. Manche Generatoren (z. B. PGS) erlauben es, direkt auf die Übergangsfunktion Einfluß zu nehmen, um Konflikte zu beseitigen, ohne die Produktionen zu verändern. Damit löst man z. B. die Mehrdeutigkeit von optionalen `else`-Anweisungen indem eine Reduktion zur einseitigen bedingten Anweisung verhindert wird, falls ein `else` folgt. Generatoren wie Yacc lassen auch mehrdeutige Produktionen z. B. für Ausdrücke zu, die durch Angabe von Präzedenzen und Assoziativität der Operatoren eindeutig werden. Es werden sogar Zerteiler für konfliktbehaftete Grammatiken erzeugt, wobei dann Konflikte nach internen Regeln des Generators (Lesen vor Reduzieren, reduziere p_i vor reduziere p_j, falls p_i textuell vor p_j steht) aufgelöst werden. Es ist jedoch sehr fragwürdig, solche Eigenschaften auszunutzen, da der Grammatik dann kaum mehr zu entnehmen ist, welche Sprache akzeptiert wird.

Fehlerbehandlung Fehlerbehandlung

Einige Generatoren wie PGS , Lalr und Palo erzeugen automatisch Zerteiler mit einer Fehlerbehandlung nach der Methode der simulierten Fortsetzung. Hier erhält man ohne jegliche zusätzliche Spezifikationen einen Zerteiler, der sich im Fehlerfall meist akzeptabel verhält und immer eine konsistente Ausgabe produziert. Das Verhalten läßt sich verbessern, wenn man z. B. Symbole spezifiziert, die als Aufsetzpunkt ungeeignet sind (z. B. Bezeichner), oder die nicht überlesen werden sollen (z. B. wichtige Klammern). Dieses Verfahren erfordert keine Modifikation der Grammatik. Im Generator Yacc wird die Technik der Fehlerproduktionen angewandt. Die Qualität der Fehlerbehandlung ist dann abhängig von dem Aufwand, den man in die Modifikation der Grammatik investiert. Ohne Einfügen von Fehlerproduktionen hält der Zerteiler beim ersten Fehler an. Eine konsistente Ausgabe ist in jedem Fall auf diese Weise schwierig zu erzielen.

Die von einem generierten Zerteiler ausgelösten Fehlermeldungen können im Quellprogramm an der Fehlerstelle positioniert werden. Allerdings ist der Meldungstext im allgemeinen undifferenziert, da Informationen über den Zerteilerzustand kaum hilfreich sind. Dies gilt auch für die im erreichten Zustand akzeptablen Symbole. Nützliche Information kann durch Anzeigen des Aufsetzpunktes gegeben werden.

Implementierungstechnik Implementierung

Die im Generator angewandte Technik zur Implementierung des Zerteileralgorithmus bestimmt wesentlich dessen Leistungsdaten. LL(1)-Zerteiler werden meist nach der Technik des rekursiven Abstiegs (z. B. Llgen) oder mit direkt programmiertem Keller (z. B. Deer und Palo) erzeugt. Letztere erzielen kürzere Laufzeiten. Tabellengesteuerte LL(1)-Zerteiler können zwar mit weniger Speicher auskommen, sind aber im allgemeinen langsamer. LALR(1)-Zerteiler werden im allgemeinen als Tabelleninterpretierer erzeugt. Direkte Programmierung ist zwar möglich, aber noch ungebräuchlich. Da die Zahl der Zustände und der Symbole für realistische Grammatiken recht hoch, der Tabelleninhalt aber recht redundant ist, wenden Generatoren verschiedene Verfahren zur Speicherreduktion an (DENC84). Sie können allerdings durch komplexe Zugriffsfunktionen die Laufzeit erhöhen.

Schnittstellen Schnittstellen

Der von einem Generator erzeugte Zerteiler ist kein eigenständiges Programm, sondern wird mit seinen Schnittstellen zur lexikalischen und semantischen Analyse und zum Fehlermodul integriert. Dazu muß zum einen die Implementierungssprache des Zerteilers geeignet sein (z. B. C bei Yacc, Deer und Palo sowie C, Pascal, Modula-2 oder Ada bei PGS und Lalr), des weiteren müssen die Schnittstellen der Module passen

oder angepaßt werden. Solche Anpassungen bleiben dem Benutzer erspart, wenn mehrere Werkzeuge aufeinander abgestimmt sind (z. B. Lex und Yacc oder GAG und PGS) oder in einer Übersetzerentwicklungsumgebung integriert sind (WAIT88).

An der Schnittstelle zur lexikalischen Analyse wird wie in Kapitel 3 beschrieben, eine Prozedur vom Zerteiler aufgerufen, die den Syntaxcode des nächsten Symbols liefert. Die Codierung muß für den Zerteiler zusätzlich zur kontextfreien Grammatik festgelegt werden. Attribute semantisch relevanter Symbole müssen an der Schnittstelle zur semantischen Analyse verfügbar sein, sie brauchen jedoch nicht durch den Zerteiler hindurch gereicht zu werden. Falls der Zerteiler eine Fehlerbehandlung enthält, die zum Erreichen des Aufsetzpunktes auch Symbole einfügt, müssen für diese an der Schnittstelle geeignete Attributwerte erzeugt werden.

Fehlermeldungen sollten nicht im generierten Zerteiler direkt, sondern über eine im gesamten Übersetzer einheitlich verwendete Fehlerprozedur ausgegeben werden. Die Fehlerposition fällt an der Eingabeschnittstelle an. Die semantische Analyse durch Attributauswerter wird über die Anknüpfungen der Produktionen angeschlossen. Sie führen entweder Operationen zum Aufbau eines Strukturbaumes aus, oder werten Attribute schon während der Syntaxanalyse aus (parse time attribution). Im Falle eines LALR(1)-Zerteilers baut man den Baum von unten nach oben mit Anknüpfungen auf, die am Ende der Produktion stehen und bei der Reduktion angestoßen werden. Auf die gleiche Weise können Attributberechnungen von unten nach oben im Baum durchgeführt werden. Anknüpfungen an Positionen innerhalb der rechten Seiten von Produktionen (für weitergehende Attributierungen) können jedoch in bestimmten Fällen die LALR(1)-Eigenschaft der Grammatik zerstören (PURD80). Dies gilt nicht für LL(1)-Zerteiler: Anknüpfungen können während der zielorientierten Analyse an beliebigen Stellen angestoßen werden, um einen Strukturbaum aufzubauen oder Attribute während eines vollständigen Baumdurchlaufes zu berechnen.

5. Attributierte Grammatiken

Überblick

Die Übersetzung eines Programmes ist an seiner abstrakten Struktur orientiert, die im Strukturbaum repräsentiert wird. Die Programmkomponenten können jedoch nicht jeweils isoliert betrachtet werden, da sie Eigenschaften haben, die sich erst durch ihre Einbettung in einen größeren Kontext ergeben, z. B. der Typ eines Ausdrucks oder die durch Definitionen festgelegten Eigenschaften eines Objektes an dessen Anwendungsstellen. Solche *kontextabhängigen* Eigenschaften ordnen wir den Strukturelementen zu und repäsentieren sie als *Attribute* der Strukturbaumknoten. Sie werden durch die Regeln der statischen Semantik bestimmt, die auch einschränkende Bedingungen für die Übersetzbarkeit des Programms umfassen. Die Regeln zur Berechnung kontextabhängiger Eigenschaften können in dem formalen Modell *attributierter Grammatiken* präzise spezifiziert werden. Wir führen sie in Abschnitt 5.1 anhand eines Beispiels ein. Grundlage dieser Spezifikationsform sind Regeln, die angeben, wie im Kontext einer bestimmten Struktur ein Attribut aus anderen Attributen berechnet wird, z. B. der Typ eines Ausdrucks aus dem Operator und den Typen der Operanden.

Aus einer Spezifikation in Form einer attributierten Grammatik kann man systematisch oder mit Hilfe von Werkzeugen (Abschnitt 5.4) einen Attributauswerter als zentralen Modul der semantischen Analyse konstruieren. Er durchläuft den Strukturbaum und berechnet alle Attribute der Knoten unter Anwendung der spezifischen Regeln. Die Konstruktionsverfahren basieren auf unterschiedlichen Strategien für den Baumdurchlauf (Abschnitt 5.2). Sie wirken sich auf die Spezifikation durch mehr oder weniger starke Beschränkungen der Attributabhängigkeiten aus. Die Kenntnis der Verfahren und der Implementierungstechniken (Abschnitt 5.3) ist nicht nur für die manuelle Konstruktion von Attributauswertern nötig, sie ist auch nützlich für die Beurteilung und den Einsatz generierender Werkzeuge. Im nachfolgenden Kapitel 6 zeigen wir, wie die Methode der Attributierung für die Hauptaufgaben der semantischen Analyse und der Abbildung in eine Zwischensprache eingesetzt wird.

Die Methode der Attributauswertung läßt sich auch auf weiter gefaßte Aufgabenstellungen ausdehnen. So können z. B. anstelle von Eigenschaften der statischen Semantik - oder zusätzlich dazu - statistische Informationen über die Verwendung von Sprachelementen gesammelt, die Einhaltung von Regeln zum Programmierstil überprüft oder Programmteile symbolisch ausgeführt werden. Die Übersetzung der Quellsprache in eine andere, höhere Programmiersprache läßt sich meist vollständig durch einen Attributauswerter mit vorangehender lexikalischer und syntaktischer Analyse realisieren. Teilaufgaben der Code-Erzeu-

gung (z. B. Übersetzung logischer Ausdrücke in Abschnitt 7.3.3 oder Registerzuteilung für Ausdrücke in Abschnitt 7.5.3) und der Optimierung (z. B. hierarchische Datenflußanalyse in Abschnitt 8.2.6) können ebenfalls als Attributierungsproblem aufgefaßt werden. Auch Eigenschaften in anderen hierarchischen Strukturen, z. B. in Abschnitte gegliederte Dokumente, können mit der Attributierungsmethode analysiert und transformiert werden. Das Anwendungsmuster für die Attributierung wird durch einen regelbasierten Informationsfluß in Baumstrukturen allgemein charakterisiert.

Attributierte Grammatiken wurden von Knuth in KNUT68 eingeführt. Mit der Entwicklung effizienter Verfahren zur systematischen Konstruktion leistungsfähiger Attributauswerter fanden sie Eingang in die Übersetzerkonstruktion und die Entwicklung von Generatoren. Konstruktion und Anwendung von Attributauswertern nach der Methode aus KAST80 werden ausführlich in WAIT84 beschrieben. In LORH84 werden der theoretische Hintergrund dargestellt und einige Generatoren präsentiert. DERA88 enthält eine umfassende Bibliographie und Übersicht über zahlreiche Generatoren.

5.1 Definition und Beispiel

Typbestimmung

Eine attributierte Grammatik erweitert eine kontextfreie Grammatik um kontextabhängige Spezifikationen in Form von Regeln zur Berechnung von Attributen. Man nennt sie deshalb auch eine Zwei-Ebenenspezifikation. Ein zunächst intuitives Verständnis von attributierten Grammatiken mag das folgende Beispiel für die Typbestimmung in Ausdrücken geben. Der Produktion

pOpr: Ausdruck ::= lOpd Opr rOpd

wird die *Attributregel*

Ausdruck.Typ := opr_identif (Opr.Symbol, lOpd.Typ, rOpd.Typ)

zugeordnet. Sie sagt aus, daß im Kontext der Produktion *pOpr* das Typattribut des Ausdruckes durch Aufruf einer Funktion zur Operatoridentifikation bestimmt wird. Ihre Parameter sind das Operatorsymbol und die Typen der Operanden. In einem Strukturbaum wird diese Berechnung auf alle Knoten angewandt, die nach *pOpr* abgeleitet sind. Man kann sich zunächst vorstellen, daß die Knoten zusätzliche Komponenten für die Attributwerte enthalten. In Abschnitt 5.3 zeigen wir dazu Optimierungsmaßnahmen. Abbildung 5.1-1 zeigt einen Ausschnitt aus einem Strukturbaum, in dem die Operatoridentifikation als Ergebnistyp `real` geliefert hat.

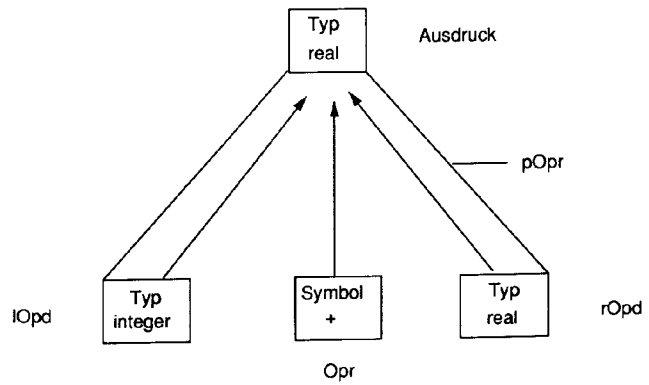


Abb. 5.1-1: Strukturbaumausschnitt mit Attributwerten.

An diesem Beispiel wird deutlich, daß ein Attributauswerter die für den jeweiligen syntaktischen Kontext (hier Produktion *pOpr*) spezifizierten Attributregeln anwenden muß. Da die Berechnungen im allgemeinen von anderen Attributen abhängen (wie die Parameter der Funktion *opr_identif*), muß er sicherstellen, daß diese Attributwerte vorher berechnet sind (hier die Typattribute der Operanden; das Operatorsymbol wird schon durch die lexikalische Analyse bestimmt). Dazu muß er den Strukturbaum in geeigneter Weise durchlaufen (hier würde z. B. ein Aufwärtsdurchgang ausreichen).

Es muß betont werden, daß die attributierte Grammatik keine Algorithmen zum Baumdurchlauf explizit enthält, sondern nur funktionale Attributabhängigkeiten spezifiziert. Sie ist deklarativ wie eine kontextfreie Syntax. Solche Algorithmen können aus der kontextfreien Struktur und den Attributabhängigkeiten konstruiert werden, falls die attributierte Grammatik gewisse Bedingungen der Wohldefiniertheit erfüllt. Für große Teilklassen attributierter Grammatiken existieren effiziente Auswertungsalgorithmen.

Kontextbedingungen

Die berechneten Attributwerte im Strukturbaum dienen zwei Zielen: Einerseits liefern sie weitere Informationen für die Code-Erzeugung. (Für `real`-Operationen werden andere Instruktionen erzeugt als für `integer`-Operationen.) Zum anderen dienen sie der Überprüfung von Bedingungen der statischen Semantik. Daß der Ausdruck einer bedingten Anweisung vom Typ `bool` sein muß, wird z. B. in einer attribu-

tierten Grammatik durch eine *Kontextbedingung* über Attribute zur entsprechenden Produktion ausgedrückt:

pIf: Anweisung ::= `if` Ausdruck `then` Anweisung `else` Anweisung

mit der Kontextbedingung

`condition` Ausdruck.Typ = `bool`.

Sprache

Sind in einem Strukturbaum alle Bedingungen über Attributwerte im jeweiligen Kontext erfüllt, so repräsentiert er ein übersetzbares Programm. In diesem Sinne schränkt die attributierte Grammatik die Menge der syntaktisch korrekten Programme weiter auf die Sprache *L(AG)* ein. Der Attributauswerter überprüft die spezifizierten Kontextbedingungen und stellt bei Verletzung semantische Fehler fest.

Abbildung 5.1-2 enthält als ein umfangreicheres Beispiel eine attributierte Grammatik für Funktionsaufrufe. Hieran wollen wir die Definition von attributierten Grammatiken im folgenden erläutern:

Attribute

Das Skelett einer attributierten Grammatik ist eine reduzierte kontextfreie Grammatik *G*. In der Übersetzerkonstruktion ist dies im allgemeinen eine abstrakte Syntax. Den Symbolen von *G* aus $T \cup N$ werden Attribute aus einer *Attributmenge A* zugeordnet. Wir notieren Attribute als *Symbolname.Attributname*. Gleichbenannte Attribute zu verschiedenen Symbolen (z. B. *Aufruf.Typ, Ausdr.Typ*) sind verschiedene Attribute!

Jedes Exemplar eines Attributs an einem Strukturbaumknoten nimmt bei der Attributauswertung einen Wert aus seinem Wertebereich an. Im Beispiel sind die Wertebereiche der Attribute

Typ	Beschreibung des Quellsprachtyps (z. B. `integer`, `real` ...)
gD	Menge der im Kontext gültigen Definitionen,
PDef	Beschreibung einer Prozedurdefinition bestehend aus dem Prozedurbezeichner, der Liste der formalen Parametertypen und dem Resultattyp,
fPl	Liste von Typen der formalen Parameter,
id	Symbol eines Bezeichners.

Attributregeln

Jeder Produktion *p* ist eine Menge von *Attributregeln* R_p zugeordnet, z. B.

Param.fP := head (Paraml1 .fPl)
Paraml2 .fPl := tail (Paraml1 .fPl)
Param.gD := Paraml1 .gD
Paraml2 .gD := Paraml1 .gD

Symbole mit Attributen:

Aufruf	{gD, Typ}	$= A_{Aufruf}$
ProzBez	{gD, PDef}	$= A_{ProzBez}$
Paraml	{gD, fPL}	$= A_{Paraml}$
Param	{gD, fP}	$= A_{param}$
Ausdr	{gD, Typ}	$= A_{Ausdr}$
Bez	{id}	$= A_{Bez}$

Produktionen mit Attributregeln und Kontextbedingungen:

p_1: Aufruf ::= ProzBez Paraml
  Aufruf.Typ := ProzBez.PDef.Resultattyp
  Paraml.fPl := ProzBez.PDef.formParm
  transfer gD
  condition Paraml.fPL ≠ leer

p_2: $Paraml_1$ ::= Param $Paraml_2$
  Param.fP := head ($Paraml_1$.fPl)
  $Paraml_2$.fPl := tail ($Paraml_2$.fPl)
  transfer gD
  condition $Paraml_2$.fPl ≠ leer

p_3: Paraml ::= Param
  Param.fP := head (Paraml.fPl)
  transfer gD
  condition tail (Paraml.fPl) = leer

p_4: Param ::= Ausdr
  transfer gD
  condition Param.fP = Ausdr.Typ

P_5: ProzBez ::= Bez
  ProzBez.PDef := identifiziere (Bez.id, ProzBez.gD)
  condition ProzBez.PDef.Klasse = ProzedurDef

Abb. 5.1-2: Attributierte Grammatik für Aufrufe.

zu *p2*. Die beiden letzten Attributregeln sind in Abbildung 5.1-2 durch *transfer gD* abgekürzt notiert. Jede Attributregel definiert die Berechnung eines Attributwertes abhängig von anderen Attributen im gleichen Kontext. Die Anordnung der Attributregeln ist bedeutungslos. Die Reihenfolge ihrer Auswertung wird bei der Konstruktion des Attributaus-

werters bestimmt. Die beiden obigen Attributregeln teilen die Liste der formalen Parameter (*Paraml₁.fPl*) auf den ersten Parameter (*head*) und die restlichen (*tail*) auf.

Attributierter Strukturbaum

Abbildung 5.1-3 zeigt einen attributierten Strukturbaum zu unserem Beispiel. Die Knoten repräsentieren jeweils ein Symbol mit seinen Attributexemplaren als Komponenten. Ein Knoten mit seinen unmittelbaren Söhnen repräsentiert die Anwendung einer Produktion. Die im jeweiligen Kontext anzuwendenden Attributregeln werden durch Pfeile abstrahiert. Sie münden in dem definierten Attributexemplar und entspringen aus den Attributexemplaren, von denen die Berechnung abhängt. Es werden hier nur die Abhängigkeiten, nicht die zur Berechnung angewandten Funktionen dargestellt. Zu überprüfende Kontextbedingungen sind durch Kreise angegeben, in die Pfeile von den Argumenten der Bedingung münden. Die Attributwerte, die sich aus den Berechnungen ergeben, sind hier nicht gezeigt. Zur besseren Übersicht sind die Pfeile für *gD* Attribute weggelassen.

In den Attributregeln zu einer Produktion p können alle Attribute aus der Menge A_p der in der Produktion vorkommenden Symbole auftreten. Im Beispiel ist Ap_2 = {*Paraml₁.gD, Paraml₁.fPl, Param.gD, Param.fP, Paraml₂.gD, Paraml₂.fPl* }. Einige davon werden durch Attributregeln zu der Produktion definiert; sie bilden die Menge Ad_p der in *p definierend auftretenden Attribute*. In p_2 sind die Ad_{p_2} = {*Param.fP, Paraml₂.fPl, Param.gD, Paraml₂.gD*}.

Erworbene und abgeleitete Attribute

Man beobachtet im Beispiel, daß einige Attributregeln ein Attribut eines Symbols auf der rechten Seite der Produktion definieren (alle zu p_2). Wir nennen solche Attribute *erworben* (inherited). Definiert eine Attributregel ein Attribut des Symbols auf der linken Seite der Produktion, so ist dies ein *abgeleitetes* (synthesized) Attribut, z. B. *ProzBez.PDef* in p5.

Die Klassifikation erworben oder abgeleitet gilt für jedes Auftreten eines Attributs gleichermaßen, d. h. ein erworbenes Attribut (z. B. *Param.fP*) wird in jeder Produktion, in der sein Symbol rechts auftritt, definiert (p_2, p_3); entsprechendes gilt für abgeleitete Attribute. Die Attributmengen A bzw. A_X werden deshalb in zwei disjunkte Teilmengen AI und AS bzw. AI_X und AS_X aufgeteilt. In unserem Beispiel sind alle mit *gD*, *fP*, *fPl* bezeichneten Attribute erworben, die übrigen abgeleitet. Weil die Wurzel der Grammatik nicht auf der rechten Seite von Produktionen auftritt, kann sie kein erworbenes Attribut haben. Ebenso können in der attributierten Grammatik keine Attributregeln für abgeleitete Attribute von terminalen Blättern der Grammatik existieren. Attribute wie *Bez.id* kann man als abgeleitete Attribute auffassen, die beim Baumaufbau vor Beginn der Attributauswertung durch die lexikalische Analyse bestimmt werden.

Definierend und angewandt auftretende Attribute

Mit dieser Klassifizierung der Attribute werden die Mengen der definierend und angewandt auftretenden Attribute Ad_p, Aa_p definiert:

$$Ad_p = AS_{X_0} \cup \bigcup_{i=1}^{n} AI_{X_i}$$

$$Aa_p = AI_{X_0} \cup \bigcup_{i=1}^{n} AS_{X_i}$$

mit p: $X_0 = X_1 \ldots X_n$.

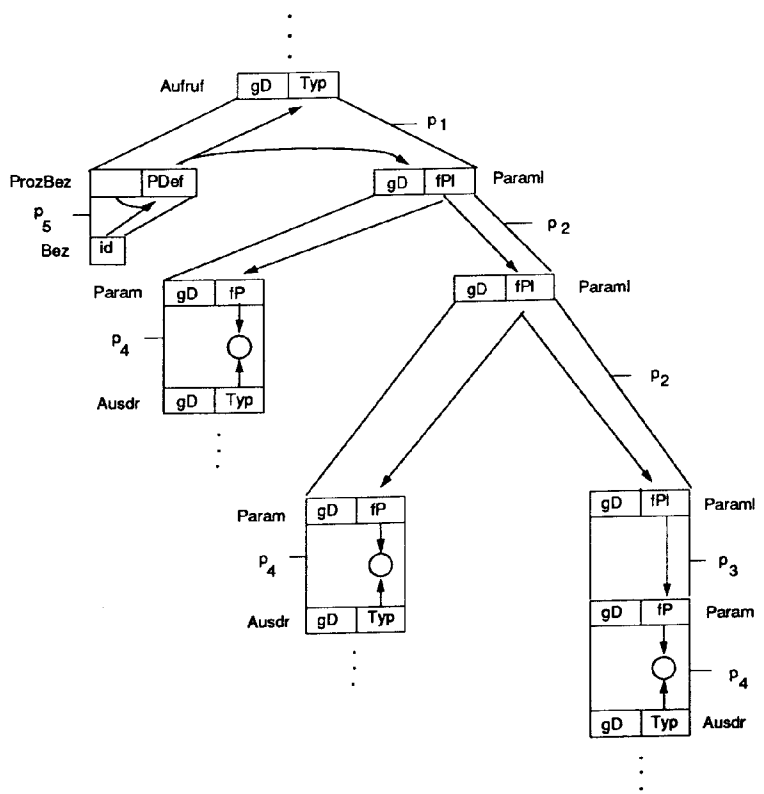


Abb. 5.1-3: Attributierter Strukturbaum für Aufrufe.

Die Abbildung 5.1-4 zeigt diesen Zusammenhang graphisch: Attribute in Aa_p werden nicht im Kontext von p definiert, sondern entweder in den Unterbäumen oder im umgebenden Kontext, aus dem Y_o abgeleitet ist.

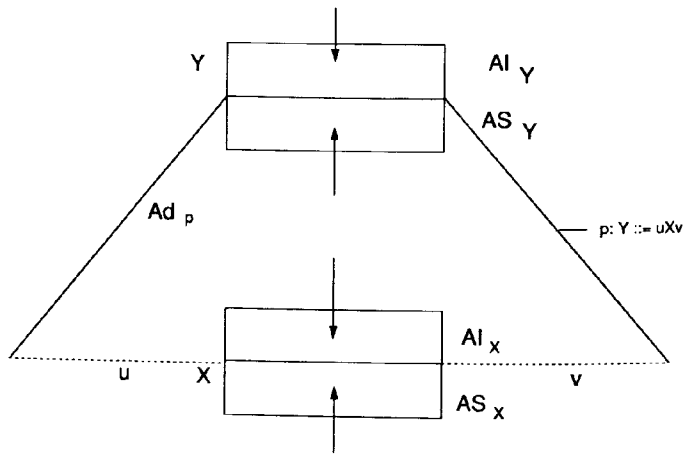


Abb. 5.1-4: Definierend und angewandt auftretende Attribute.

Bochmann-Normal-Form

Bei späteren Überlegungen werden wir zur Vereinfachung fordern, daß auf der rechten Seite von Attributregeln nur Attributauftreten aus Aa_p vorkommen. Wir sprechen dann von einer attributierten Grammatik in *Bochmann-Normal-Form* (*BnNF*). Dies ist keine wesentliche Einschränkung, denn sie kann durch einfache Substitution von Attributregeln erfüllt werden. Sei z. B. $X.d \in Ad_p$, dann verletzt die erste der beiden folgenden Attributregeln die BnNF

$Y.b := f(..., X.d,...)$
$X.d := g(....)$.

Hier ersetzt man sie durch

$Y.b := f(..., g(...), ...)$
$X.d := g(...)$.

Neben den Attributregeln werden einer Produktion p Kontextbedingungen C_p zugeordnet. Dies sind Prädikate über Attribute aus A_p. Sie schränken die Menge der syntaktisch korrekten Programme auf die auch

semantisch korrekten Sätze der Sprache L(AG) ein. In $p2$ wird z. B. gefordert, daß die Liste der formalen Parameter nicht leer ist.

5.2 Konstruktion von Attributauswertern

Eine attributierte Grammatik definiert, wie in einem beliebigen Strukturbaum der zugrundeliegenden Grammatik alle Attributwerte berechnet werden. Unter gewissen einschränkenden Voraussetzungen kann man aus der attributierten Grammatik einen Algorithmus als *Attributauswerter* konstruieren, der dieses leistet. Wir werden dafür mehrere systematische Verfahren betrachten. Einige davon sind in einfacheren Fällen manuell anwendbar, andere erfordern insbesondere für komplexere attributierte Grammatiken den Einsatz von Werkzeugen (Abschnitt 5.4).

Ein Attributauswerter ist ein Algorithmus, der den Strukturbaum auf bestimmten Wegen entlang der Kanten durchläuft und Attributwerte gemäß Attributregeln aus R_p berechnet bzw. Kontextbedingungen aus C_p prüft, wenn er sich an einem Knoten, der nach p abgeleitet ist, befindet. Die Konstruktion des Algorithmus muß garantieren, daß alle Attributwerte von denen eine Attributregel oder Kontextbedingung abhängt, berechnet sind, bevor die Attributregel angewandt wird. Deshalb spielen die in den Attributregeln ausgedrückten Abhängigkeiten zwischen Attributen eine zentrale Rolle bei der Konstruktion.

Insbesondere muß festgestellt werden, ob die Attributabhängigkeiten so definiert sind, daß es keinen Strukturbaum gibt, in dem einige Attributexemplare zyklisch voneinander abhängen. In solch einem Fall wären ihre Werte nicht effektiv berechenbar. Eine attributierte Grammatik, die diese Bedingung erfüllt, heißt *wohldefiniert*. Eine exakte Prüfung dieser Bedingung für eine beliebige attributierte Grammatik erfordert exponentiellen Aufwand und ist deshalb nicht praktikabel. Statt dessen werden bei den hier betrachteten Konstruktionsverfahren schärfere Bedingungen vorausgesetzt und bei der Konstruktion geprüft.

5.2.1 Abhängigkeitsgraphen und Zerlegungen

Wohldefinierte attributierte Grammatiken

Zu jedem Strukturbaum mit den Attributexemplaren an seinen Knoten können wir wie in Abbildung 5.1-3 einen Abhängigkeitsgraphen angeben: Die Pfeile zwischen den Attributexemplaren geben die in den Attributregeln ausgedrückten Abhängigkeiten an.

Definition 5.1: Eine attributierte Grammatik heißt *wohldefiniert*, wenn die Abhängigkeitsgraphen zu allen Strukturbäumen zyklenfrei sind.

Abhängigkeitsgraphen

Da die zu konstruierenden Attributauswerter in der Lage sein sollen, jeden beliebigen Strukturbaum zu attributieren, müssen wir unter-

suchen, wie diese Graphen aus der attributierten Grammatik entstehen. Aus der Konstruktion des Graphen in Abbildung 5.1-3 erkennt man leicht, daß er aus Teilgraphen zusammengesetzt ist, die jeweils zur Anwendung einer Produktion gehören. Diese sind für alle Anwendungen einer Produktion identisch, da sie die ihr zugeordneten Attributregeln repräsentieren. Abbildung 5.2-1 zeigt diese Graphen für die Produktionen p_1 , p_2 , p_4 unseres Beispiels.

Definition 5.2: Der Graph DDP_p der *direkten Abhängigkeiten* zu einer Produktion p enthält

- die Attributauftreten Ap als Knoten und
- gerichtete Kanten (a, b) von a nach b, falls in einer Attributregel aus $R_p b$ von a abhängt, also

  b := f(... a ...).

Notwendige Voraussetzung für die Wohldefiniertheit ist die lokale Zyklenfreiheit aller DDP_p. Aber durch die Zusammensetzung solcher Graphen könnten in einem Strukturbaum Zyklen entstehen. Die im folgenden vorgestellten Klassifikationen und Konstruktionsverfahren garantieren die Zyklenfreiheit und damit die Wohldefiniertheit der attributierten Grammatik.

Zerlegungen

Grundlage für eine Reihe wichtiger Konstruktionsverfahren für Attributauswerter und für die Klassifizierung von attributierten Grammatiken ist die *Zerlegung* der Attributmengen der Symbole in Teilmengen.

Betrachten wir in Abbildung 5.2-2 den Knoten zu einem Symbol X mit seinen Attributexemplaren im attributierten Strukturbaum. Er steht in den benachbarten Kontexten der Produktionen p und q. Seine erworbenen Attribute werden im Kontext p, seine abgeleiteten im Kontext q berechnet. Zwischen der Berechnung eines erworbenen und eines abgeleiteten Attributs (und umgekehrt) muß offensichtlich ein Kontextwechsel stattfinden. Wir fassen die Attribute, die zwischen zwei Kontextwechseln an X berechnet werden, zu einer Menge zusammen. Damit erhalten wir eine Folge von Mengen mit alternierend erworbenen und abgeleiteten Attributen.

Für die Konstruktion von Attributauswertern fordern wir nun, daß für jeden Knoten X dasselbe durch die Zerlegung beschriebene Auswertungsmuster gilt.

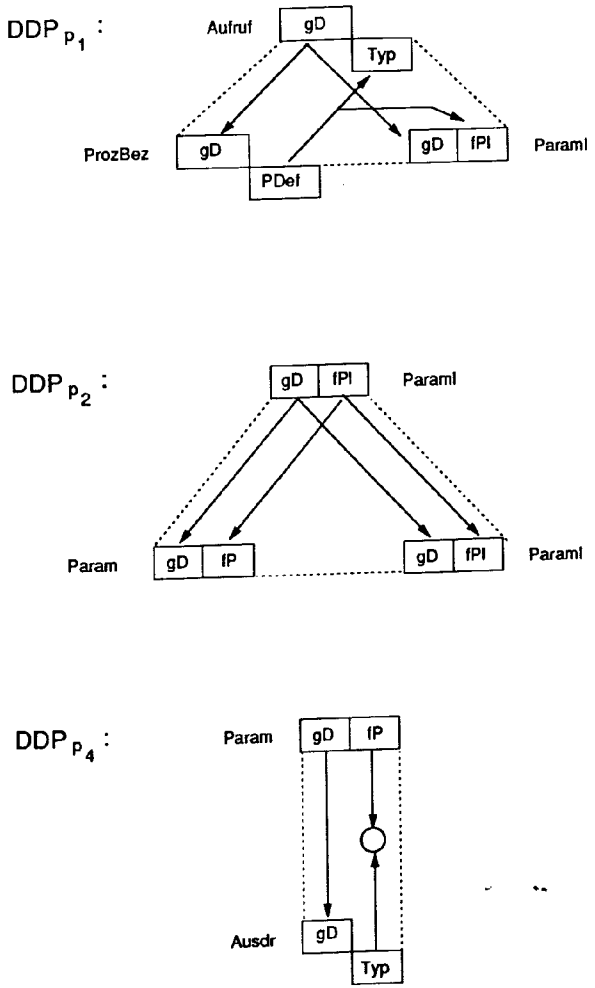


Abb. 5.2-1: Direkte Attributabhängigkeiten zu Produktionen.

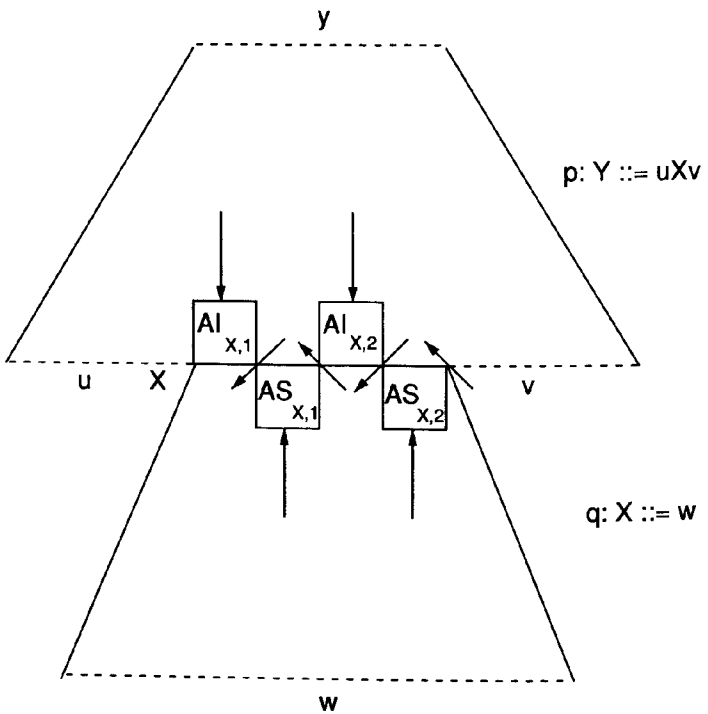


Abb. 5.2-2: Zerlegung der Attributmengen.

Definition 5.3: Eine *Zerlegung* Z zerlegt alle Attributmengen A_X in eine Folge disjunkter Teilmengen

$$AI_{X,1}, AS_{X,1}, \ldots, AI_{X,m_X}, AS_{X,m_X}$$

$$\text{mit } AI_X = \bigcup_{i=1}^{m_X} AI_{X,i},\ AS_X = \bigcup_{i=1}^{m_X} AS_{X,i}.$$

Klassen attributierter Grammatiken

Mit diesem Begriff lassen sich attributierte Grammatiken und zugehörige Attributauswerterverfahren klassifizieren.

Wollen wir einen Attributauswerter konstruieren, der den Baum mit einer bestimmen Strategie durchläuft, müssen wir eine dafür zulässige Zerlegung finden. Sie garantiert, daß die Attribute jedes Knotens

beim Baumdurchlauf in der durch die Zerlegung festgelegten Reihenfolge berechnet werden. Mögliche Baumdurchlaufstrategien sind z. B.

LAG:	Durchläufe von links nach rechts,
AAG:	alternierende Durchläufe,
SWEEP:	Durchläufe mit spezifischer Permutation der Unterbäume zu jeder Produktion,
VISIT:	spezifische Besuchssequenzen zu jeder Produktion *p*.

Eine attributierte Grammatik gehört der Klasse *K* an, wenn es für sie eine Zerlegung gibt, die für die Strategie *K* zulässig ist. In der obigem Aufzählung sind die Klassen nach aufsteigender Mächtigkeit angeordnet.

Im folgenden werden wir zunächst das Prinzip der Besuchssequenzen für die Steuerung von Attributauswertern aller obiger Klassen betrachten und dann ein Verfahren zur Konstruktion von LAG-Auswertern angeben.

5.2.2 Besuchssequenzen

Besuchssequenzen

Ein Attributauswerter durchläuft den Strukturbaum entlang den syntaktischen Kanten und wertet dabei Attributregeln und Kontextbedingungen aus. Befindet er sich an einem Knoten, der nach einer Produktion

$$p: Y ::= u\, X_i\, v$$

abgeleitet ist, so kann er Operationen der folgenden Arten ausführen:

eval a	berechne einen Attributwert zu $a \in Ad_p$ nach der Attributregel aus R_p,
cond c	werte Kontextbedingung $c \in C_p$ aus,
$\downarrow_j X_i$	besuche den Sohnknoten zu X_i zum j-ten Mal,
$\uparrow_j$	besuche den Vaterknoten von Y zum j-ten Mal.

Einen beliebigen Durchlauf durch den Strukturbaum mit Attributberechnungen in passender Reihenfolge kann man beschreiben, indem man jedem Knoten eine *Besuchssequenz* als Folge obiger Operationen zuordnet. Die Besuchssequenzen für Knoten, die nach der gleichen Produktion *p* abgeleitet sind, sind identisch. Wir konstruieren deshalb zu einer attributierten Grammatik eine Menge $VS = \bigcup_{p \in P} vs_p$ der Besuchsse-

quenzen zu den Produktionen aus P. Diese steuert den Attributauswerter für jeden Strukturbaum. In benachbarten Kontexten des Strukturbaumes werden Besuchssequenzen verzahnt abgearbeitet, siehe Abbildung 5.2-3. Der erste Nachfolgerbesuch zu X_i ($\downarrow_1 X_i$) in vs_p führt zum Anfang von vs_q ; $\downarrow j\, X_i$ führt zur Operation, die auf $\uparrow j$-1 in vs_q folgt.

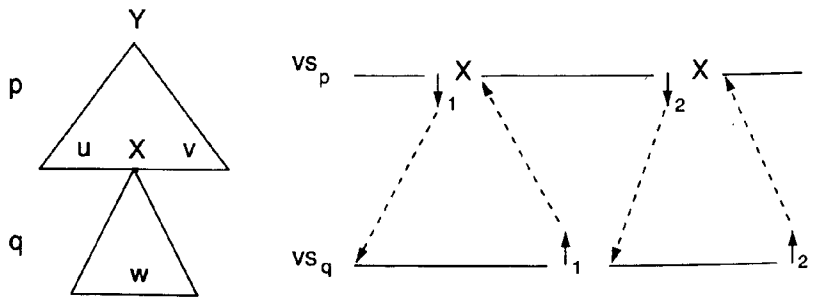


Abb. 5.2-3: Verzahnte Abarbeitung von Besuchssequenzen.

Die Besuchssequenzen müssen offensichtlich so konstruiert sein, daß

- sie mit einem Vaterbesuch enden,
- die Anzahl der Vaterbesuche und Sohnbesuche bezogen auf dasselbe Symbol gleich sind,
- die Attribute aus $AI_{X_i,j}$ vor $\downarrow j\, X_i$ und aus $AS_{X_0,j}$ vor $\uparrow j$ berechnet werden, und
- die Kontextbedingungen ausgewertet werden, wenn ihre Argumente berechnet sind.

Hat man eine für die Klasse *VISIT* zulässige Zerlegung bestimmt, so kann man leicht aus den Abhängigkeitsgraphen DDP_p unter Einhaltung obiger Regeln die Menge der Besuchssequenzen konstruieren. Abbildung 5.2-4 gibt einen Satz von Besuchssequenzen für unser Beispiel an. In diesem einfachen Fall wird jeder Knoten genau einmal besucht. Ferner werden hier in jeder vs_p die Söhne von links nach rechts besucht. Diese Menge VS beschreibt deshalb einen links-abwärts Durchgang durch den Baum. Wir werden auf solche VS im nächsten Abschnitt näher eingehen. vs_{p2} hätte man auch so angeben können, daß erst der rechte und dann der linke Sohn besucht wird. Dabei müßten die den Besuchen vorangehenden Berechnungen mit vertauscht werden.

Mit Besuchssequenzen können auch allgemeinere als paßorientierte Baumdurchläufe (wie in diesem Beispiel) beschrieben werden. Diese Eigenschaft wird in der semantischen Analyse realistischer Programmiersprachen ausgenutzt, z. B. wenn der Deklarationsteil eines Blockes einen weiteren Besuch erfordert, bevor der Anweisungsteil ausgewertet werden kann. Besuchssequenzgesteuerte Attributauswerter wurden in KAST80 eingeführt.

vs_{p_1}:
eval ProzBez.gD
$\downarrow_1$ ProzBez
eval Paraml.gD
eval Paraml.fPL
cond
$\downarrow_1$ Paraml
eval Aufruf.Typ
$\uparrow_1$

vs_{p_2}:
eval Param.gD
eval Param.fP
$\downarrow_1$ Param
eval $Paraml_2$.gD
eval $Paraml_2$.fPL
cond
$\downarrow_1 Paraml_2$
$\uparrow_1$

vs_{p_3}:
eval Param.fP
cond
$\downarrow_1$ Param
$\uparrow_1$

vs_{p_4}:
eval Ausdruck.gD
$\downarrow_1$ Ausdr
cond
$\uparrow_1$

vs_{p_5}:
eval ProzBez.PDef
cond
$\uparrow_1$

Abb. 5.2-4: Besuchssequenzen zur Aufrufattributierung.

5.2.3 LAG-Attributauswerter

LAG-Attributauswerter

Eine einfache Klasse von Attributauswertern durchläuft den Strukturbaum in einem oder mehreren links-abwärts Durchgängen. Im Fall eines einzigen solchen Durchgangs sind die Angabe der Zerlegung und die Konstruktion von Besuchssequenzen trivial: Die Attributmengen werden jeweils in $AI_{X,1} = AI_X$ und $AS_{X,1} = AS_X$ zerlegt. Die Besuchssequenzen enthalten genau einen Vorgängerbesuch und einen Nachfolgerbesuch der Söhne in der Reihenfolge von links nach rechts. Die erworbenen Attribute $AI_{X_i,1}$ von Symbolen der rechten Seite einer Produktion werden vor dem Besuch $\downarrow_1 X_i$ ausgewertet, die abgeleiteten Attribute $AS_{X_0,1}$ vor dem Vorgängerbesuch $\uparrow_1$ am Ende der Besuchssequenz. Abbildung 5.2-5 zeigt diese Auswertungsreihenfolge schematisch. Die Besuchsequenzen in Abbildung 5.2-4 folgen dieser Anordnung.

LAG(1)

Den Attributabhängigkeiten in den Graphen DDP_p kann man unmittelbar ansehen, ob die attributierte Grammatik für diese Auswertungsstrategie geeignet ist: Es darf kein Pfeil von rechts nach links führen oder zwischen Attributen desselben Symbols der rechten Seite be-

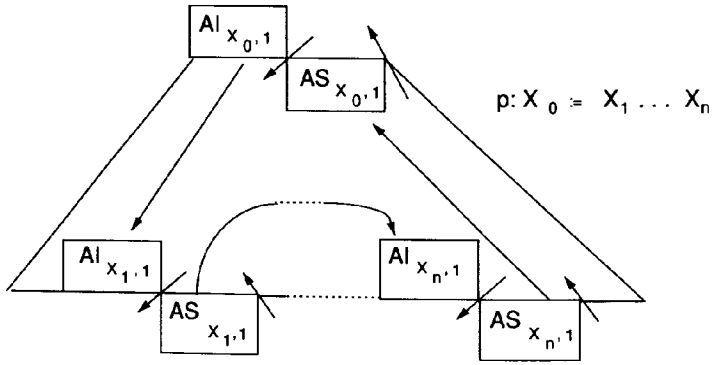


Abb. 5.2-5: LAG(1)-Auswertungsreihenfolge.

stehen (BnNF vorausgesetzt). Diese Eigenschaft beschreibt folgende Definition der zugehörigen AG-Klasse:

Definition 5.4: Eine attributierte Grammatik in BnNF ist eine *LAG(1)*, wenn es eine Zerlegung mit $m_X = 1$ für alle $X \in V$ gibt und für alle $p: X_0 ::= X_1 \ldots X_n \in P$ gilt: Aus $(X_i.a, X_j.b) \in DDP_p$ folgt $j = 0$ oder $i < j$.

LAG(k)

Solche Attributauswerter können wir leicht erweitern, so daß sie den Baum in k links-abwärts Durchgängen durchlaufen, und damit komplexere Attributabhängigkeiten lösen können: Im k-ten Durchlauf sind alle Attributwerte aus früheren Durchläufen verfügbar. Für die in diesem Durchlauf berechneten Attribute gelten dieselben Bedingungen wie im LAG(1)-Fall.

Definition 5.5: Eine attributierte Grammatik in BnNF ist eine *LAG(k)*, wenn es eine Zerlegung mit $m_X = k$ für alle $X \in V$ gibt und für alle $p \in P: X_0 ::= X_1 \ldots X_n$ gilt: Aus $(X_i.a, X_j.b) \in DDP_p$ und

$X_i.a \in AI_{X_{i,g}} \cup AS_{X_{i,g}}$ (d. h., $X_i.a$ wird im g-ten Durchlauf berechnet) und

$X_j.b \in AI_{X_{j,h}} \cup AS_{X_{j,h}}$ (d. h., $X_j.b$ wird im h-ten Durchlauf berechnet)

folgt: $g < h$ oder $g = h$ und ($j = 0$ oder $i < j$).

Besuchssequenzen für den *LAG(k)*-Fall konstruiert man durch Hintereinandersetzen von k Abschnitten, die jeweils wie im *LAG(1)*-Fall aufgebaut sind und mit $\uparrow_1, \ldots, \uparrow_k$ enden.


```
Eingabe: Abhaengigkeitsgraphen DDP[p] fuer p aus P,
         Attributmenge A

Ausgabe: disjunkte Attributmengen A[h], h = 1,..,k,
         so dass A[h] im h-ten Durchlauf berechnet
         wird, falls die LAG (k) fuer ein k
         erfuellt ist. k ist dann minimal.

B := A; (* noch nicht zugeordnete Attribute *)
h := 0;

repeat
  h := h+1; A[h] := B; B:= {};
    repeat
        changed := false;
        for all p aus P do
            for all (X[i].a, X[j].b) aus DDP[p] do
                if X[j].b in A[h] then
                   if X[i].a in B or
                     (X[i].a in A[h] and (0<j<=i))
                   then begin
                      A[h] := A[h] - {X[j].b};
                      B := B + {X[j].b};
                      changed := true
                   end
    until not changed
until B = {} or A[h] = {};


falls    B = {} ist LAG (k) erfuellt fuer k = h

falls A[h] = {} ist LAG (k) fuer kein k erfuellt
```

Abb. 5.2-6: Algorithmus zur Bestimmung der LAG(k)-Zerlegung.

Abbildung 5.2-6 gibt einen Algorithmus an, der entscheidet, ob eine attributierte Grammatik die *LAG(k)*-Bedingungen für ein k erfüllt, in diesem Fall das kleinste k bestimmt, die Attribute den k Durchläufen zuordnet und damit auch die Zerlegung bestimmt. Dem Algorithmus liegt folgendes Prinzip zugrunde: Es wird versucht, alle noch nicht einem Paß zugeordneten Attribute dem aktuell betrachteten h-ten Paß zuzuordnen. Anhand der DDP_p werden diejenigen Attribute gestrichen, welche die Bedingung verletzen. Brauchen keine Attribute mehr gestrichen zu werden, so ist der h-te Paß vollständig zugeordnet. Sind alle Attribute zugeordnet, so ist die LAG(k)-Bedingung erfüllt. Mußten alle betrachteten Attribute gestrichen werden, so kann die LAG(k)-Bedingung für kein k erfüllt werden.

5.3 Implementierung von Attributauswertern

Tabellengesteuerte Attributauswerter

Das Grundprinzip eines Attributauswerters ist ein Kellerautomat, der den Strukturbaum durchläuft und von den Besuchssequenzen gesteuert wird. Der Keller enthält Paare von Verweisen auf Strukturbaumknoten und die an diesem Knoten als nächste auszuführende Operation der Besuchssequenz. Bei einem Nachfolgerbesuch wird der aktuelle Knoten und die auf den Besuch folgende Position in der Besuchssequenz gekellert. Die Operation, die an dem besuchten Knoten als nächstes auszuführen ist, wird aus der Produktionsnummer im Knoten und der Besuchsnummer bestimmt. Ein Vorgängerbesuch entkellert ein solches Paar. Mit geeigneter Codierung der Besuchssequenz werden nach diesem Verfahren tabellengesteuerte Attributauswerter implementiert.

Direkt programmierte Attributauswerter

Ein LAG(k)-Attributauswerter führt k links-abwärts Durchläufe durch den Strukturbaum aus. Jeden der k Durchläufe können wir durch eine Menge von im allgemeinen rekursiven Prozeduren implementieren, je eine für jedes Nichtterminal (siehe Abb. 5.3-1). Im Rumpf der Prozedur wählt eine Fallunterscheidung die zur angewandten Produktion gehörige Besuchssequenz aus.

```
procedure eval_h_X (x: node);
(* h-ter Pass fuer Nichtterminal X *)
begin
  case x^.prod of
  p1:  begin
         (* p1 sei X ::= Y ... Z *)

         (* berechne alle erworbenen Attribute
            von x^.sohn[1] aus A[h] *)
         eval_h_Y (x^.sohn[1]);
         ...
         (* berechne alle erworbenen Attribute
            von x^.sohn[n] aus A[h] *)
         eval_h_Z (x^.sohn[n]);

         (* berechne alle abgeleiteten Attribute
            von x aus A[h] *)
       end;
  p2:  ...
  ...
  end
end
```

Abb. 5.3-1: Rekursive LAG(k)-Prozeduren.

Bei dieser Implementierung sind die Besuchssequenzen abschnittweise für jeden Durchlauf ausprogrammiert. Die Rolle des Zustandskellers wird von dem Parameter k und der Rückkehradresse zu jedem Prozeduraufruf übernommen. Die Prozeduren haben die gleiche Struktur wie die eines LL-Zerteilers, der nach der Methode des rekursiven Abstiegs implementiert ist. Deshalb können diejenigen für den ersten Attributierungsdurchlauf mit den Zerteilerprozeduren verschmolzen werden (*parse time attribution*). Die Fallunterscheidung über die Produktionen wird ersetzt durch die Fallunterscheidung über das nächste Symbol. In die Besuchssequenz wird das Akzeptieren von Terminalen und das Generieren der Sohnknoten vor der Berechnung seiner erworbenen Attribute eingefügt.

Bisher sind wir davon ausgegangen, daß die Attributwerte als Komponenten der Strukturbaumknoten gespeichert werden. Zusammen mit der Verkettung der Knoten im Baum gibt Abbildung 5.3-2 eine Möglichkeit der Repräsentation der Knoten an. Alternativ könnte statt n Sohnverweisen auch jeweils ein Sohn- und ein Bruderverweis gewählt und bei Terminalknoten auf die Produktionskennung verzichtet werden. Schematisch angewandt führt diese Implementierungstechnik zu einem funktionsfähigen, im allgemeinen aber zu speicheraufwendigen Attributauswerter. Wir wollen deshalb einige Techniken zur Reduktion des Speicheraufwands angeben.

```
type node    = ^ noderec;

     noderec = record
                 prod: production;
                 sohn: array [1..maxrhs] of node;
                 pos:  quellposition;
               case symb: grammatiksymbol of
               X:     (Attribute von X);
               Y:     (Attribute von Y);
               ...
               end;
```

Abb. 5.3-2: Attributierter Strukturbaumknoten.

Attribute mit komplexen Werten

Häufig beschreibt ein Attribut eine komplexe Eigenschaft durch einen aus mehreren Komponenten zusammengesetzten Wert (z. B. Eigenschaften eines definierten Objektes, Typangaben, Liste gültiger Definitionen). Solche Werte würden in unserem naiven Speichermodell häufig identisch kopiert werden (z. B. die Attribute *gD* in Abb. 5.1-2). Dieser Aufwand ist unvertretbar und überflüssig: Attribute werden nach ihrer Berechnung nicht mehr verändert. Zusammengesetzte Werte können

deshalb durch einen Verweis auf den Wert repräsentiert und das Kopieren durch die Kopie des Verweises implementiert werden.

Attributoptimierung

Der Speicherumfang der Strukturbaumknoten kann weiter drastisch reduziert werden, indem alle Exemplare eines Attributes $X.a$ nicht in jedem Knoten zu X, sondern in einer einzigen Variablen oder einem Keller außerhalb des Strukturbaumes gespeichert werden. Der Wert der Variablen bzw. des obersten Kellerelementes muß dann der Wert des gerade beim Baumdurchlauf benutzten Exemplares von $X.a$ sein. Anhand der Lebensdauer von $X.a$, die durch die Besuchssequenzen beschrieben wird, kann man feststellen, ob eine solche Speicheroptimierung möglich ist.

Für LAG(k)-Attributauswerter lassen sich einfach zu überprüfende Lebensdauerbedingungen angeben: Wird ein Attribut $X.a$ nur in einem Durchlauf h verwendet, also dort berechnet und zum letztenmal benutzt, so kann es durch einen globalen Keller implementiert werden. Bei der oben angegebenen Implementierung des Attributauswerters durch rekursive Prozeduren braucht man diesen Keller nicht explizit zu deklarieren. Erworbene Attribute $X.a$, die diese Bedingung erfüllen, werden durch einen Wertparameter `a` der Prozedur `eval_h_X` implementiert; abgeleitete Attribute $X.a$ durch einen Referenzparameter `a`, für den in den `eval_h_X` aufrufenden Prozeduren eine lokale Variable deklariert wird.

Gilt zusätzlich für ein Attribut $X.a$, das obige Bedingung erfüllt, daß zwischen seiner Berechnung und seiner letzten Benutzung kein anderer Knoten zu einem X besucht wird, so kann man statt des Kellers eine globale Variable für $X.a$ implementieren.

Mit diesen Optimierungsmaßnahmen verbleiben nur noch solche Attribute im Strukturbaum, die in einem Durchlauf berechnet und in einem späteren benutzt werden. Die oben genannten Bedingungen gelten selbstverständlich für alle paßorientierten Attributauswerter (LAG, AAG, SWEEP). Für allgemeine, durch Besuchssequenzen gesteuerte Auswerter können entsprechende Bedingungen angegeben werden, die aber schwieriger zu prüfen sind. Sie werden von Generatoren wie GAG (Abschnitt 5.4) angewandt.

5.4 Generatoren für Attributauswerter

Die in Abschnitt 5.2 besprochenen Techniken zur Konstruktion von Attributauswertern werden in generierenden Systemen automatisch angewandt. Sie analysieren die Attributabhängigkeiten der gegebenen attributierten Grammatik und generieren daraus eine Implementierung eines Attributauswerters, falls die attributierte Grammatik der dem gewählten Attributierungsverfahren zugrundeliegenden Klasse angehört.

Damit kann sich der Konstrukteur eines Übersetzers darauf konzentrieren, die in der semantischen Analyse zu bestimmenden Eigenschaften durch Attributregeln zu spezifizieren. Dies ist dann besonders vorteilhaft, wenn die Sprachregeln komplex sind und der verwendete Generator eine mächtige Klasse attributierter Grammatiken verarbeiten kann.

Im folgenden diskutieren wir Gesichtspunkte, die für die Auswahl und den Einsatz von Generatoren für Attributauswerter wesentlich sind: Eingabeform, Analysemethode, Implementierungstechnik und Schnittstellen. Die Verweise auf existierende Systeme sind als Beispiele zu verstehen. Generatoren für Attributauswerter werden derzeit insbesondere im Hinblick auf ihre praktische Einsetzbarkeit und Effizienz der Produkte weiter- und neuentwickelt. Einen guten Überblick über solche Systeme gibt (DERA88).

Spezifikationsform

Notation der attributierten Grammatiken

Attributwerte werden insbesondere zur Beschreibung realistischer Eigenschaften von Programmiersprachen durch recht komplexe Operationen und Funktionen über strukturierten Daten berechnet. Ein wesentliches Merkmal der Generatoren ist deshalb die sprachliche Form der Attributregeln. Die Systeme können hierzu in drei Klassen unterschieden werden:

- Die Attributierung wird in einer Programmiersprache als *Gastsprache* formuliert (z. B. HLP in RÄIH84). Dies ist dann meist die Sprache, in der der Attributauswerter generiert wird. Die Generierung wird dadurch auf die Strukturen des Attributauswerters und Textsubstitution für die Attributzugriffe beschränkt. Falls die Gastsprache jedoch nicht funktional oder ausdrucksorientiert ist (wie z. B. Pascal) hat der Generator keine Kontrolle über mögliche Seiteneffekte der Attributregeln. Wird dies nicht durch Programmierdisziplin ersetzt, verliert die attributierte Grammatik ihren deklarativen Charakter und der Generator die Möglichkeit zur Optimierung.
- Die Attributierung wird in einer für den Generator speziellen Eingabesprache formuliert (z. B. ALADIN für GAG in KAST82). In diesem Fall übersetzt der Generator die Attributregeln in die Implementierungssprache des Attributauswerters. Die Eingabesprache muß dann hinreichend mächtige Datentypen und Operationen enthalten und sollte funktionalen Charakter haben, um die deklarativen Eigenschaften zu erhalten. In diesem Fall hat der Generator vollständige Kontrolle über die Spezifikation und kann auch weitgehende Optimierungen zur Implementierung anwenden.
- Die Attributierung wird unter Verwendung von Datentypen und Funktionen, die außerhalb der attributierten Grammatik in der Implementierungssprache spezifiziert sind, streng funktional formuliert. In diesem Fall hat der Generator vollständige Kontrolle über die Verwendung der

Attribute, falls die externen Funktionen keine Seiteneffekte verursachen. Intendierte Seiteneffekte müssen durch Attributabhängigkeiten beschrieben werden. Damit wird der Übersetzungsaufwand im Generator reduziert und auch Flexibilität für die Implementierungssprache erzielt. Das System LIGA in KAST89 verfolgt z. B. diesen Ansatz.

In attributierten Grammatiken beschreibt eine große Anzahl von Attributregeln den Transport identischer Attributwerte. Eine attributierte Grammatik wird wesentlich übersichtlicher, wenn in der Eingabesprache für häufig vorkommende Attributierungsmuster *abkürzende Notationen* vorgesehen sind. Anwendungsbeispiele dafür zeigen die attributierte Grammatik in Abbildung 5.1-2 und Attributierungen in Kapitel 6.

Der Entwurf von attributierten Grammatiken wird vereinfacht und die Übersichtlichkeit des Ergebnisses verbessert, wenn eine attributierte Grammatik stärker strukturiert werden kann. Hier ist allerdings eine Strukturierung nach Auswertungsdurchläufen weniger hilfreich als eine solche nach inhaltlichen Gesichtspunkten (z. B. Zusammenfassung aller Attributregeln zur Typisierung und zur Bezeichneridentifikation). Solche Eigenschaften sind derzeit noch Gegenstand der Entwicklung neuer Systeme.

Ein Generator sollte attributierte Grammatiken zu abstrakter Syntax erlauben, da sie weniger redundant sind. Dies ist nur möglich, wenn Zerteilergenerierung und Strukturbaumaufbau zur konkreten Syntax von der Generierung des Attributauswerters entkoppelt sind. Andererseits ist die Auswertung von Attributen während der Syntaxanalyse eine Maßnahme zur Leistungsverbesserung von Attributauswertern. Automatische, für den Entwerfer transparente Lösungen hierzu sind zur Zeit noch Gegenstand der Forschung.

Auswertungsmethode

Klassen attributierter Grammatiken

Mit der im Generator festgelegten Methode zur Attributauswertung wird die Klasse der akzeptierten attributierten Grammatiken festgelegt. Je mächtiger die Klasse ist, desto weniger müssen beim Entwurf Einschränkungen zu den Attributabhängigkeiten berücksichtigt werden. Der Entwerfer kann sich stärker auf die Beschreibung von zu berechnenden Eigenschaften konzentrieren und braucht weniger ablauforientiert zu formulieren. Unter den paßorientierten Methoden sind die Klassen AAG(k), z. B. LINGUIST (FARR82), und SWEEP(k), z. B. MUG2 (GANZ82), in diesem Sinne auch für realistische Programmiersprachen hinreichend mächtig. Auf Besuchssequenzen und der Klasse OAG basierte Verfahren, z. B. GAG und SSAGS (PAYT82), sind noch flexibler. Diese Auswerter reduzieren außerdem den Aufwand für den Baumdurchlauf. Die mächtigsten Verfahren verarbeiten allgemeine attributierte Grammatiken durch Interpretation der Attributabhängigkeiten zur

Laufzeit, z. B. ERN (JOUR84). Dadurch wird der Laufzeitaufwand erhöht und Verletzungen der Wohldefiniertheit werden erst bei der Attributauswertung erkannt.

Implementierungstechnik

Implementierungstechnik

Für paßorientierte und durch Besuchssequenzen gesteuerte Attributauswerter können die gleichen Techniken der Ablaufsteuerung angewandt werden wie im zielorientierten Zerteiler: rekursive Prozeduren, Schleife mit explizitem Keller oder Tabelleninterpretation, wiederum mit Laufzeitvorteilen für die zweite Technik. Auf Besuchssequenzen basierende Verfahren reduzieren den Laufzeitaufwand gegenüber paßorientierten, da sie nicht alle Teilbäume mit der maximalen Besuchszahl durchlaufen. Allerdings ist bei der semantischen Analyse im allgemeinen der Laufzeitaufwand für die Berechnung von nicht-trivialen Attributwerten signifikant höher als der für die Ablaufsteuerung des Baumdurchlaufes.

Für die Leistungsdaten des Attributauswerters ist deshalb der Speicheraufwand prinzipiell kritischer. Er wird bestimmt durch die
1) Repräsentation der Attributwerte,
2) die Dauer ihrer Existenz im Speicher und durch
3) den Strukturbaum.

Zu (1) kann bei streng funktionaler Eingabesprache durch Verweisrepräsentation für komplexe Werte erhebliche Ersparnis erzielt werden. Je nach Eingabeform wird diese Typabbildung entweder vom Generator oder vom Benutzer vorgenommen.

Die Lebensdauer von Attributwerten ist häufig hinreichend kurz und kann anhand der attributierten Grammatik a priori festgestellt werden. Durch automatisch vom Generator angewandte Optimierungen von (2) werden viele Attribute statt im Baumknoten in globalen Variablen oder Kellern untergebracht (z. B. in GAG, LIGA, LINGUIST, MUG2). Dabei können gleichzeitig auch einige kopierende Attributregeln weggelassen werden. Zu (3) kann auf die Speicherung solcher Teile des Strukturbaums verzichtet werden, deren Attributierung vollständig während der Syntaxanalyse erledigt wird.

Schnittstellen

Schnittstellen

Die beiden Hauptschnittstellen des Attributauswerters sind der Strukturbaum zum Zerteiler und der Zwischencode zur Synthesephase. Darüber hinaus werden Informationen und Strukturen des Definitionsmoduls auch in der Synthese verwendet. Die entsprechenden Attributwerte müssen dann nach der semantischen Analyse erhalten bleiben.

Damit die Attributierung auf dem abstrakten Strukturbaum erfolgen kann, muß die Generierung des Attributauswerters von der des Zer-

teilers abgetrennt sein oder die abstrakte Syntax im Generator selbst aus der konkreten hergestellt werden. Die zu attributierende Baumstruktur wird vom Generator festgelegt und z. B. durch die aus der attributierten Grammatik extrahierten Syntax beschrieben. Anknüpfungen in der konkreten Syntax spezifizieren dann die Konstruktion des Baumes, die der Attributierung vorgeschaltet ist.

Als Ausgangsschnittstelle der semantischen Analyse ist der attributierte Strukturbaum zu komplex und zu groß. Er enthält viele Attribute, die so nur während der Analyse verwendet werden. Die für die Synthese notwendige Information wird besser auf eine einfachere Zwischensprache abgebildet. Eine solche Transformation kann auch durch Attributierung geleistet werden. Dazu berechnet man die Elemente der Zwischensprache konzeptionell als Attributwerte. Sie können tatsächlich durch Ausgabefunktionen erzeugt werden, deren Aufrufreihenfolge durch Attributabhängigkeiten spezifiziert wird. Diese Technik kann immer angewandt werden, wenn der Generator die Verwendung von (ggf. externen) Ausgabefunktionen in der attributierten Grammatik erlaubt (z. B. GAG, LIGA). Als Alternative zu dieser Technik kann die Herstellung der Zwischensprache auch durch Transformation des attributierten Baumes spezifiziert und von einem Werkzeug zur Baumtransformation (z. B. OPTRAN in MUG2) generiert werden.

6. Semantische Analyse

Überblick

Die semantische Analyse berechnet kontextabhängige Eigenschaften von Elementen des Programms in seiner abstrakten Repräsentation. Zusammen mit der Transformation in eine Zwischensprache wird diese Aufgabe durch eine attributierte Grammatik spezifiziert (s. Kap. 5). Der zentrale Übersetzermodul ist ein Attributauswerter, der mit speziellen Datenmodulen kooperiert. In diesem Kapitel konzentrieren wir uns auf zwei wichtige Klassen kontextabhängiger Eigenschaften: Typbestimmung und Bezeichneridentifikation. Die hierfür entwickelten Attributierungs- und Implementierungsschemata sind prinzipiell auf die Analyse jeder höheren Programmiersprache anwendbar.

Für Programmiersprachen mit statischer Typbindung werden die Typen von Ausdrücken und Operanden als kontextabhängige Attribute bestimmt und die Einhaltung von Sprachregeln zur Typisierung überpüft (Abschnitt 6.1). Programme enthalten Definitionen, die Bezeichner für Objekte mit bestimmten Eigenschaften einführen. Gültigkeitsregeln der Sprache beschreiben, wie einem angewandten Auftreten eines Bezeichners anhand der Programmstruktur die jeweils gültige Definition zugeordnet wird. In Abschnitt 6.2 zeigen wir die Attributierung dieser Regeln und Techniken zur Implementierung der dabei benutzten Datenmodule. Mit der Typprüfung und der Bezeichneridentifikation werden auch Verletzungen von Sprachregeln erkannt. Durch Maßnahmen zur Fehlerbehandlung (Abschnitt 6.3) wird erreicht, daß Fehler deutlich gemeldet werden und die Datenstrukturen des Übersetzers konsistent bleiben. In Abschnitt 6.4 zeigen wir, wie als Ergebnis der Attributierung und als Schnittstelle zur Synthesephase eine Programmrepräsentation in einer Zwischensprache erzeugt wird.

Das recht einfache Attributierungsschema zur Typbestimmung kann ebenso auf die Berechnung anderer Eigenschaften von Ausdrücken angewandt werden, z. B. zur symbolischen Auswertung. Die hier vorgestellte Lösung für die Bezeichneridentifikation demonstriert die Anwendung eines allgemeineren Entwurfsprinzips: Die strikt deklarative Spezifikation durch Regeln einer attributierten Grammatik wird mit einer effizienten Zugriffsstruktur in einem Datenmodul zu einer insgesamt klaren und leistungsfähigen Lösung kombiniert.

6.1 Typprüfung

Typregeln

In höheren, statisch typisierten Programmiersprachen gehört die Typprüfung zu den zentralen Aufgaben der semantischen Analyse. Die Sprachregeln dazu betreffen große Bereiche der abstrakten Syntax: De-

klarationen und Ausdrücke. Wir spezifizieren Typeigenschaften, indem wir solchen Sprachelementen jeweils ein Attribut zuordnen, das den Typ beschreibt und Berechnungsregeln dafür angibt.

Die Typregeln können wir anhand ihrer Anwendung in der syntaktischen Struktur in drei Bereiche gliedern:

- Typangaben und Typdefinitionen in Deklarationen,
- Typbestimmung für Ausdrücke,
- Relationen über Typen (wie Gleichheit, Verträglichkeit), die zur Typbestimmung und -prüfung angewandt werden.

Typangaben in Deklarationen

Im folgenden geben wir an, wie diese Aufgaben in attributierten Grammatiken spezifiziert und durch Attributauswerter gelöst werden. Eine Deklaration wie

```
var a: array [1..10] of integer;
```

bestimmt den Typ des Objektes *a*. Er gehört zu den Eigenschaften, die die Deklaration mit dem Bezeichner a verknüpft. Hinzu kommen Eigenschaften wie Variable, Blockzugehörigkeit usw. Der Typ ist deshalb Teil der in der Deklaration festgelegten Objektbeschreibung (s. Abschnitt 6.2).

Die abstrakte Syntax für eine solche Variablendeklaration könnte lauten

VarDekl ::= Bezeichner Typangabe.

Die Typeigenschaft der deklarierten Variable wird dann durch den aus *Typangabe* abgeleiteten Teilbaum bestimmt. Wir ordnen deshalb dem Nichtterminal Typangabe und seinen Unterstrukturen ein abgeleitetes Attribut *Typ* zu.

Der Wertebereich dieser (wie auch der später noch eingeführten) Typattribute umfaßt Abstraktionen aller in der Programmiersprache definierten und konstruierbaren Typen.

Wir stellen die Typabstraktionen durch einen Typdeskriptor dar mit Varianten für jede Typklasse, z. B. mit den Pascal-Vereinbarungen in Abbildung 6.1-1. Die Varianten einer Typklasse geben jeweils ihre abstrakten Eigenschaften an:

- Basistyp und Grenzen für Ausschnitte,
- Index- und Elementtyp für Reihungen,
- Definitionsliste der Komponenten für Verbunde usw.

Eine Variante (typ_bez) repräsentiert Typbezeichner, die in Typangaben angewandt auftreten. Sie werden durch einen Verweis des Bezeichnermoduls dargestellt, der erst bei der Bezeichneridentifikation


```
type

Typ_Ref    = ^ Typ_Deskr;

Typ_Klasse = (grund_typ, sub_typ, arr_typ, rec_typ, ..., typ_bez);

Typ_Deskr  = record case   tk: Typ_Klasse of
               grund_typ: ;
               sub_typ:   (basis : Typ_Ref; ug, og : integer);
               arr_typ:   (index, elem : Typ_Ref);
               rec_typ:   (komp: Def_Liste);
                  ...
               typ_bez:   (id: Symbol_Ref)
             end;
```

Abb. 6.1-1: Implementierung von Typattributen.

durch den dafür definierten Typ ersetzt wird. Die vordefinierten Grundtypen werden durch Verweise auf verschiedene Exemplare der entsprechenden Typdeskriptoren unterschieden.

Werte der Typattribute sind Verweise auf die Typdeskriptoren (Typ_Ref). Sie werden gemäß der Syntax für Typangaben im attributierten Strukturbaum von unten nach oben berechnet, z. B.

$Typangabe_1$ ::= 'array' $Typangabe_2$ 'of' $Typangabe_3$
$Typangabe_1$.typ := gen_arr_typ ($Typangabe_2$.typ, $Typangabe_3$.typ)

Die Funktion gen_arr_typ liefert einen Verweis auf einen neuen Reihungstypdeskriptor. Die in diesem Kontext notwendige Prüfung der Zulässigkeit des Indextyps kann erst erfolgen, wenn alle eventuell vorkommenden Typbezeichner durch ihre zugeordneten Typen ersetzt sind.

Zur Typbestimmung von Ausdrücken ordnen wir allen Nichtterminalen für Ausdrücke und Operanden ein abgeleitetes Attribut mit dem oben entworfenen Wertebereich zu. Die zentrale Aufgabe besteht hier in der *Identifikation überladener Operatoren*. In den meisten Programmiersprachen sind Operatorzeichen mit verschiedenen Bedeutungen überladen: So steht z. B. in Pascal der Operator + sowohl für die integer- und die real-Addition als auch für die Vereinigung von Mengen. Welche Bedeutung jeweils im Kontext angewandt wird, hängt von den Operandentypen ab. In Sprachen wie Ada und Algol 68 können Operatoren auch durch Definitionen im Programm zusätzlich überladen werden. In Ada bestimmt neben den Operanden auch der Kontext, in dem der Ausdruck steht, die Bedeutung des Operators.

Zeichen	linker Operand	rechter Operand	Ergebnis	Operatorbedeutung
+	integer	integer	integer	iadd
+	real	real	real	radd
+	set	set	set	sunion
/	real	real	real	rdiv
=	t	t	boolean	eq

Abb. 6.1-2: Operatortabelle.

Typen von Ausdrücken

Die verschiedenen Bedeutungen eines Operators können wir für die Typbestimmung anhand ihrer Signatur unterscheiden. Wir stellen für alle Operatoren und ihre Bedeutungen eine Operatortabelle auf, wie sie beispielhaft für einige zweistellige Operatoren in Abbildung 6.1-2 angegeben ist. Jede Zeile der Tabelle codiert eine Operatorbedeutung mit ihrem überladenen Operatorzeichen, den Operandentypen und dem Ergebnistyp. Die Tabelle wird durch eine Funktion

```
identifiziere_Operator (z, l, r)
```

implementiert. Sie bestimmt zu gegebenen Operatorzeichen `z` und Operandentypen `l` und `r` die codierte Operatorbedeutung. Dies ist der Eintrag der letzten Spalte der Tabelle zu einer Zeile mit dem Operatorzeichen `z`, so daß `l` und `r` jeweils an die Operandentypen anpaßbar sind, wie es Anpassungsregeln der Sprache vorschreiben. Aus der so codierten Operatorbedeutung können dann der Ergebnistyp und die ggf. notwendigen Anpassungen für die Operanden bestimmt werden. In der Tabelle steht `set` für die Klasse der Mengentypen und `t` für beliebige Typen. In diesen Fällen müssen außerdem die Operandentypen gleich sein. Die Attributierung für Ausdrücke der abstrakten Syntax könnte dann z. B. lauten:

Ausdruck$_1$ ::= Ausdruck$_2$ Opr Ausdruck$_3$
    Opr.bedeutung :=
        identifiziere_Operator (Opr.zeichen, Ausdruck$_2$.typ, Ausdruck$_3$.typ)
    Ausdruck$_1$.typ := Ergebnistyp (Opr.bedeutung, Ausdruck$_2$.typ).

Die Typattribute von Literalen werden durch ihre Notation bestimmt, von Variablen und Konstantenbezeichnern aus der für sie gültigen Definition (s. Abschnitt 6.2), von Zugriffsoperationen (Reihungsindizierung, Verbundselektion) durch den Element- bzw. Komponententyp.

Relationen über Typen

Die Typen von Ausdrücken müssen in speziellem Kontext bestimmte Bedingungen erfüllen. So muß z. B. in einer bedingten Anweisung der Typ des Ausdrucks `boolean` sein oder darauf angepaßt werden können; in Zuweisungen muß der Typ des Ausdrucks an den der Variablen anpaßbar sein. Dies wird in der attributierten Grammatik durch Kontextbedingungen formuliert, z. B.

Zuweisung ::= Variable ':=' Ausdruck
    `condition` anpaßbar (Ausdruck.typ, Variable.typ).

Ein Beispiel für eine weitere Relation über Typen sind aktuelle Parameter, die als Referenz übergeben werden. Ihr Typ muß z. B. mit dem des formalen Parameters übereinstimmen:

```
Akt_Param ::= Ausdruck
    condition
        if Akt_Param.übergabe = var_param
        then gleiche_Typen (Ausdruck.typ, Akt_Param.formaler_typ)
        else anpaßbar (Ausdruck.typ, Akt_Param.formaler_typ).
```

Solche Kontextbedingungen werden als Relationen über Typen definiert und durch Funktionen wie `gleiche_typen` und `anpaßbar` implementiert. Sie operationalisieren die Regeln der Sprache für Anpaßbarkeit (compatibility) und Äquivalenz von Typen. So gilt z. B. in Pascal, daß Ausschnittstypen zu ihrem Basistyp anpaßbar sind. Dies wird durch entsprechende Bedingungen über den Argumenten der Funktion ausgedrückt, die Typabstraktionen der oben eingeführten Wertebereiche sind.

Die Regeln für Typäquivalenz in Pascal (Namensäquivalenz) können auf Vergleiche der Verweise auf die Typabstraktionen zurückgeführt werden. Voraussetzung dafür ist, daß in den Typabstraktionen auftretende Typbezeichner vorher durch den für sie definierten Typ ersetzt wurden (s. Abschnitt 6.2).

Gilt wie in Algol 68 die strukturelle Äquivalenz für Typen, so muß die Funktion `gleiche_Typen` die Komponenten der beiden Argumente vergleichen und dabei rekursiv angewandt werden. Ein zusätzlicher Parameter, der als Liste von Typpaaren die auf jeder Rekursionsstufe verglichenen Typen akkumuliert, stellt den Abbruch der Rekursion auch bei rekursiv definierten Typen sicher.
So sind z. B. zwei Typen

```
type  t1 = record a: integer; b: ↑t1 end;
      t2 = record a: integer; b: ↑t2 end;
```

verschieden unter Namensäquivalenz, aber gleich unter struktureller Äquivalenz.

6.2 Bezeichneridentifikation

Gültigkeit von Definitionen

Die Übersetzeraufgabe der Bezeichneridentifikation bestimmt zu jedem *angewandten Auftreten* die gültige Definition des Bezeichners (*definierendes Auftreten*). Bezeichner treten angewandt z. B. in Ausdrücken und Typangaben auf. Aus der zugeordneten Definition entnimmt man Eigenschaften des definierten Objekts, z. B. den Typ einer Variablen zur Typprüfung.

Jede Definition eines Bezeichners hat einen *Gültigkeitsbereich*. Das ist der Abschnitt des Quellprogramms, in dem Anwendungen des Bezeichners dieser Definition zugeordnet werden. Die Gültigkeitsbereiche werden durch spezielle Regeln der Sprachdefinition (scope rules) bestimmt. Für die meisten blockstrukturierten Sprachen basieren sie auf der sogenannten *Verdeckungsregel*:

> Eine Definition eines Bezeichners *b* gilt im kleinsten sie umfassenden Block, ausgenommen darin enthaltene Blöcke mit einer Definition von b.

Blöcke im Sinne dieser Regel sind Syntaxkonstrukte (Teilbäume des abstrakten Strukturbaumes), denen Definitionen explizit oder implizit zugeordnet sind, in Pascal z. B. Prozeduren und `with`-Anweisungen.

Während z. B. Algol 60 mit obiger Regel auskommt, wird sie in anderen Sprachen durch zusätzliche Regeln ergänzt: Pascal verlangt z. B., daß kein Bezeichner vor seiner Definition angewandt werden darf (ausgenommen Typbezeichner in Referenztypangaben).

Beide Regeln zusammen begründen, daß folgendes Pascal-Programm fehlerhaft ist:

```
program t (output);
  procedure p ; begin writeln ('äußeres p') end;
  procedure q;
        procedure r; begin p end;
        procedure p; begin writeln ('inneres p') end;
  begin (*q*) r end;
begin (*t*) q end.
```

Gemäß der Verdeckungsregel identifiziert der Aufruf von `p` in `r` die innere Definition von `p`. Diese Anwendung ist jedoch unzulässig, da

sie der Definition vorausgeht. Viele, insbesondere 1-Paß-Übersetzer erkennen diesen Fehler nicht, sondern identifizieren die äußere Definition von `p`. Sprachen wie Modula-2 ergänzen die Verdeckungsregel durch Regeln zum Import und Export von Definitionen über Modulgrenzen.

Im folgenden zeigen wir zunächst die Spezifikation der Bezeichneridentifikation in attributierten Grammatiken und geben dann eine spezielle Zugriffsstruktur zur effizienten Implementierung an. Wir legen dabei nur die Verdeckungsregel zugrunde. Zusatzregeln können durch Verfeinerung der Attributierung berücksichtigt werden.

6.2.1 Attributierung von Gültigkeitsbereichen

Für die Attributierung der Bezeichneridentifikation setzen wir voraus, daß die Definitionen in einem Programm durch geeignete Attributierung beschrieben werden. Eine Definition ordnet einem Bezeichner Eigenschaften zu, z. B. "*b* ist eine Variable vom Typ *t*". Ihre Abstraktion ist deshalb ein Paar

(Bezeichner, Eigenschaften).

Attributierung der Bezeichneridentifikation

Die Eigenschaften umfassen alle an den Anwendungsstellen relevanten Informationen, wie Klasse des definierten Objektes (Variable, Konstante, Prozedur usw.), seinen Typ, die umfassende Struktur (Prozedurdefinition), die Blockschachtelungstiefe usw. Sie können je nach Sprache variieren. Solch ein Paar wird als abgeleitetes Attribut jedes Nichtterminals berechnet, das eine Definition syntaktisch beschreibt. Die Werte dieser Attribute werden im kleinsten sie umfassenden Block zu einer Liste lokaler Definitionen zusammengefaßt (Attribut `loc` in Abb. 6.2-1).

Hierauf aufbauend entwickeln wir die Attributierung der Bezeichneridentifikation. Ihr Ziel ist es, im Kontext jedes angewandten Bezeichners ein Attribut `env` (für environment) so zu bestimmen, daß es alle in dieser Umgebung gültigen Definitionen enthält. Eine Funktion identifiziert dann darin die gesuchte Definition:

```
Applied_ident ::= Identifier
  Applied_ident.def := identify_Env (Identifier.id, Applied_ident.env)
```

Nehmen wir zunächst einmal an, daß der Wertebereich der Attribute `env` ebenfalls als Liste von Definitionen gegeben ist und die Funktion `identify_Env` darin die Definition für den Bezeichner linear sucht. In der Implementierung werden wir den Suchaufwand reduzieren. Die Verdeckung einer äußeren Definition durch eine weiter innen liegende kann dann dadurch erreicht werden, daß die innere in der Liste vor der äußeren gefunden wird.

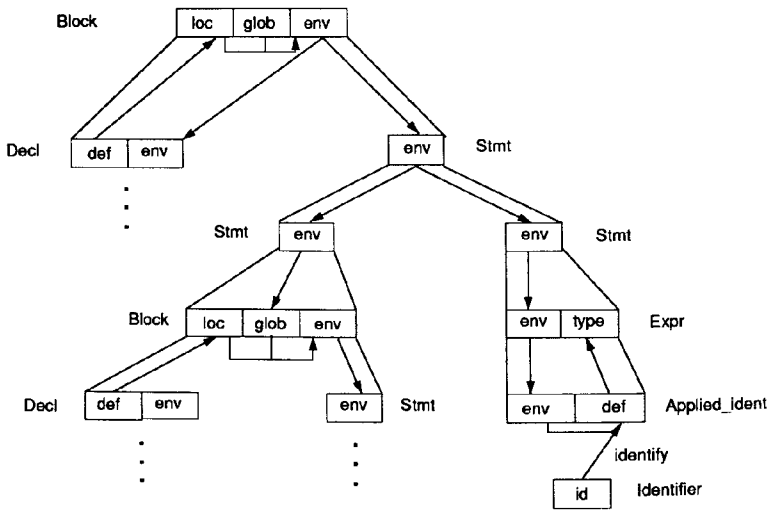


Abb. 6.2-1: Attributierung der Bezeichneridentifikation.

Wir können nun die Verdeckungsregel beschreiben, indem wir allen Nichtterminalen, aus denen angewandte Bezeichner abgeleitet werden können, ein solches erworbenes Attribut `env` zuordnen. In jedem Kontext, ausgenommen Blöcken, werden diese Attribute zu identischen Werten bestimmt. Im Falle von Blöcken werden die lokalen Definitionen den globalen (vorne in der Liste) zugefügt und zu dem abgeleiteten Attribut `Block.env` berechnet:

```
Block  ::= Decl Stmt
           Block.loc := Decl.def
           Block.env := new_Env (Block.loc, Block.glob)
           Decl.env := Block.env
           Stmt.env := Block.env
```

Damit ergibt sich für die Blockstruktur eines Programms die in Abbildung 6.2-1 dargestellte Attributierung.

Attributauswerter

Man sieht leicht, daß diese Attributierung die LAG(2)-Bedingung erfüllt: Im ersten Baumdurchlauf werden die lokalen Definitionen jedes Blockes (Attribute `def`) bestimmt, im zweiten Durchlauf werden alle Attribute `env` berechnet, die angewandten Bezeichner identifiziert und im Aufwärtsdurchgang die davon abhängende Typbestimmung durchgeführt.

Eine genauere Analyse des zweiten Baumdurchlaufs zeigt wichtige Optimierungsmöglichkeiten auf:

- Die Werte der `env`-Attribute werden im 2. Durchlauf berechnet und nach der Identifikation nicht mehr verwendet.
- Die Werte aller `env`-Attribute innerhalb eines Blockes (ohne darin eingeschachtelte Blöcke) sind gleich.

Hieraus folgt, daß ein Attributauswerter statt der vielen `env`-Attribute an den Strukturbaumknoten eine einzige globale Variable benutzen kann. Sie nimmt jeweils den Wert des `env`-Attributs am gerade besuchten Knoten an. Dazu muß jeweils beim Eintritt in einen Blockkontext die Liste der lokalen Definitionen der gerade gültigen Definitionsliste zugefügt werden; beim Verlassen des Blockkontextes wird sie wieder unzugänglich. Die Liste verhält sich wie ein Keller.

Außerdem können in speziellen Notationen der attributierten Grammatik alle `env`-Attribute außer `Block.env` und die zugehörigen Attributregeln weggelassen werden. Zur Identifikation nimmt man dann direkt Bezug auf `Block.env`. Die fehlenden Attribute und Regeln sind systematisch ergänzbar.

Angewandte Bezeichner in Definitionen

Besondere Beachtung müssen wir Bezeichnern widmen, die angewandt in Definitionen auftreten, z. B. Konstantenbezeichner in Ausschnittstypen oder Typbezeichner in Typangaben, wie `max`, `index` und `arr` in

```
const   max = 100;
type    index = 1 .. max;
        pt = ↑ arr;
        arr = array [index] of integer;
var     p : pt;
```

Den Bezeichnern `max`, `index` und `arr` sind Eigenschaften zugeordnet, die an ihrer Anwendungsstelle wiederum Eigenschaften von Definitionen und deren Abstraktion in der Attributierung bestimmen. Deshalb würde die oben skizzierte Attributierung im Definitionsteil zu zyklischen Abhängigkeiten führen. Wegen der Verdeckungsregel wird dieser Zyklus auch nicht durch die Zusatzregel "definierendes vor angewandtem Auftreten" in Sprachen wie Pascal behoben. In der Attributierung wendet man folgende Technik an, um den Zyklus zu vermeiden: Die Identifikation von Bezeichnern, die zur Abstraktion von Definitionen beitragen, wird verzögert. In den Attributwerten, die Definitionen beschreiben, werden die nichtidentifizierten Bezeichner (Codierung des Bezeichnermoduls) als Abstraktion verwendet. Damit wird eine vorläufige Definitionsliste im Blockkontext berechnet. In diesem Kontext wird daraus ein zusätzliches Attribut berechnet, das dieselben lokalen

Definitionen, aber mit den identifizierten Bezeichnern enthält. Dazu werden alle lokalen Definitionen durchlaufen und darin auftretende angewandte Bezeichner identifiziert.

6.2.2 Implementierung der Bezeichneridentifikation

Aus der in Abschnitt 6.2.1 vorgestellten Attributierung der Bezeichneridentifikation leiten wir eine weiter verbesserte Implementierung her. Wir führen zwei Datenmodule ein, welche die Attributwerte der beiden Phasen aufnehmen (Abb. 6.2-2). Der Attributauswerter steuert den Ablauf und berechnet bzw. benutzt Attribute durch Aufruf der Modulfunktionen.

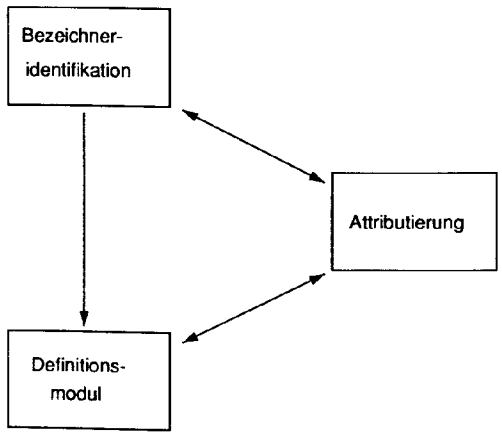


Abb. 6.2-2: Module zur Bezeichneridentifikation.

Definitionsmodul

Der Definitionsmodul speichert zu jeder Definition des Programms die Eigenschaften des definierten Objektes, wie dessen Klasse (Variable, Prozedur usw.), Typ und seine Zugehörigkeit zu umfassenden Strukturen. Hinzu kommen Eigenschaften, die im Zuge der weiteren Analyse bestimmt und solche, die später durch die Speicherabbildung (Abschnitt 7.2) berechnet werden. Der Modul wird mit Funktionen zum Bilden einer neuen Definition, zum schreibenden oder lesenden Zugriff auf bestimmte Eigenschaften einer Definition benutzt. Die von dem Modul verwalteten Daten werden auch nach Abschluß der Bezeichneridentifikation bei der Erzeugung des Zwischencodes und der Speicherzuteilung verwendet. Die Eigenschaften zur Speicherabbildung werden in der Synthesephase weiterverwendet.

Modul zur Bezeichneridentifikation

Der Modul zur Bezeichneridentifikation implementiert die eindeutige Zuordnung zwischen angewandt auftretenden Bezeichnern und ihrer Definition mit den im Definitionsmodul gespeicherten Eigenschaften. Wenn im Zuge der Attributierung jedem Bezeichner seine Definition zugeordnet wurde, ist die Aufgabe dieses Moduls erledigt. Seine Datenstrukturen werden nicht weiter benötigt. Auf die Eigenschaften zu dem Bezeichner wird über die identifizierte Definition im Definitionsmodul direkt zugegriffen.

Der Bezeichneridentifikation liegt das abstrakte Modell baumstrukturiert geschachtelter Umgebungen zugrunde. Eine Umgebung ordnet einer Menge von Bezeichnern jeweils genau eine Definition zu. Programmstrukturen, in denen neue Definitionen für Anwendungen gültig sind (z. B. Module, Prozeduren, Blöcke oder `with`-Anweisungen) bilden eine neue Umgebung: Die umfassende Umgebung wird um die lokalen Definitionen erweitert, wobei lokale Definitionen globale zum gleichen Bezeichner verdecken. Dazu enthält die Modulschnittstelle z. B. folgende in der Attributierung verwendete Funktionen

```
function    init_Env : Env; ...
function    new_Env (e : Env) : Env, ...
function    add_Def (id: Id_Ref; d : Def; e : Env) :
            Boolean; ...
function    identify_Env (id : Id_Ref; e : Env) : Def; ...
function    identify_Range (id : Id_Ref; e : Env) : Def; ...
```

Die Funktion `init_Env` initialisiert die Wurzel eines Umgebungsbaumes. `new_Env` bildet zu der umfassenden Umgebung `e` eine neue Umgebung. `add_Def` fügt in diese eine Definition `d` für den Bezeichner `id` ein und prüft, ob eine lokale Definition für diesen Bezeichner schon existiert. `identify_Env` (`identify_Range`) liefert die in `e` gültige (lokale) Definition zum Bezeichner `id`, der als Codierung des Bezeichnermoduls angegeben ist.

Implementierungstechniken

Zur Implementierung dieser Funktionen im Modul Bezeichneridentifikation können verschiedene Datenstrukturen gewählt werden, die sich in ihrer Sucheffizienz unterscheiden:

1) Die Umgebung ist eine lineare Liste von Referenzen auf Definitionen. `add_Def` fügt `d` vorne an die Liste `e` an. Zur Identifikation wird die Liste von Definitionen innerer Umgebungen zu äußeren hin durchsucht.

2) Umgebungen werden entsprechend ihrer Schachtelung als Baumstruktur aufgebaut. `new_Env` erzeugt einen Baumknoten mit einer Referenz auf die lokalen Definitionen `d` und auf den Knoten zur unmittelbar umfassenden Umgebung `e`. `identify_Env` durchsucht die Definitionslisten zu Knoten ausgehend von e bis hin zur Wurzel.

3) Wie in (2) wird durch *new_Env* ein Baum aufgebaut. Um die lineare Suche zu vermeiden, legt man einen Abbildungsvektor an, der mit den

Bezeichnercodierungen indiziert wird. Er enthält zu jedem Bezeichner eine Liste von Referenzen auf Definitionen dieses Bezeichners. Die Listen werden durch lokale Prozeduren des Moduls wie

```
procedure enter (e : Env); ...
procedure leave (e : Env); ...
```

jeweils als Keller verwaltet: `enter` fügt die lokalen Definitionen `d` einer Umgebung jeweils in den Keller zu ihrem Bezeichner ein, `leave` entkellert sie. Damit ist die jeweils gültige Definition eines Bezeichners immer als erstes Kellerelement über den Vektor unmittelbar zugreifbar. Ebenso wird ohne Suche festgestellt, daß ggf. keine gültige Definition existiert. Der Zustand der Suchstruktur wird durch einen Knoten *e* im Baum, der aktuellen Umgebung, repräsentiert. Alle Definitionslisten auf dem Wege von der Wurzel bis `e` (und sonst keine) sind in dieser Reihenfolge in die Keller eingefügt. Dieser Zustand muß bei Ausführen von `identify_Env (id, e)` bestehen oder falls nötig durch Kelleroperationen hergestellt werden. Der Auf- und Abbau von Umgebungen ist also durch die Reihenfolge der Identifikationen im Attributauswerter bestimmt. Die Attributierung von Gültigkeitsregeln wie in Pascal und Algol 60 führt zu Attributauswertern, die alle Identifikationen in einer Umgebung durchführen, bevor diese verlassen wird. Damit ist garantiert, daß keine Definitionsliste mehr als einmal gekellert und entkellert wird. (Man könnte dann auch den Umgebungsbaum durch einen Umgebungskeller ersetzen.) Für die Gültigkeitsregeln von Modula-2 mit importierten Definitionen in lokalen Modulen gilt diese vereinfachende Bedingung nicht.

Aufwand

Der Aufwand der Verfahren (1) und (2) ist proportional zum Produkt der Anzahl zu identifizierender Bezeichner und der Anzahl der jeweils gültigen Definitionen (Listenlänge). Der Aufwand von (3) ist proportional zur Summe der Zahl der Definitionen und Bezeichneranwendungen, falls, wie in (3) erläutert, in jedem Block geschlossen identifiziert wird. Insgesamt wachsen die Vorteile des Verfahrens (3) mit konstantem Aufwand für `identify_Env` gegenüber der linearen Suche in (1) und (2) mit dem Verhältnis von angewandten Bezeichnern zu Definitionen und mit der Zahl der nicht lokalen Anwendungen von Definitionen.

Schließlich sei noch auf ein grundsätzlich zu lösendes Problem hingewiesen: Wie schon bei der Attributierung der Bezeichneridentifikation dargelegt, können angewandte Bezeichner auch in den Abstraktionen von Definitionen (Definitionsmodul-Einträgen) auftreten. Unmittelbar nach Einfügen der lokalen Definitionen in die Menge der gültigen Definitionen müssen diese durchlaufen und die darin enthaltenen angewandten Bezeichner identifiziert werden.

6.3 Fehlerbehandlung

Für die semantische Analyse gibt es im Gegensatz zur Zerteilung kein geschlossenes Verfahren zur Fehlerbehandlung. Man kann jedoch einige Entwurfsprinzipien für attributierte Grammatiken allgemein und für die Typprüfung und die Bezeichneridentifikation insbesondere angeben, deren Berücksichtigung die Qualität der Fehlerbehandlung verbessert.

Voraussetzungen

Ein Attributauswerter muß robust gegenüber Eingabefehlern sein, d. h. er darf auch im Fehlerfall nicht in einen undefinierten Zustand geraten. Dazu sind folgende Bedingungen zu erfüllen:

- Auch für syntaktisch fehlerhafte Programme muß ein vollständiger Strukturbaum vorliegen, der der abstrakten Syntax genügt. Diese Forderung sollte die syntaktische Fehlerbehandlung erfüllen, siehe Abschnitt 4.4. Werden dabei terminale Blätter mit Symbolattributen eingefügt, so sind diese mit geeigneten Fehlerwerten vorzubesetzen.
- Jede Attributregel muß auch bei Vorliegen eines semantischen Fehlers einen Wert aus dem Wertebereich des Attributs berechnen, da nach festgestellter Verletzung einer Kontextbedingung der Auswerter nicht anhalten soll. Dies wird z. B. beim Generator GAG durch Regeln der Eingabesprache erzwungen und für die attributierte Grammatik überprüft. Ggf. müssen die Wertebereiche um Fehlerwerte erweitert werden (s. Abschnitt 6.3.2, 6.3.3).

Spezifische Meldungen

Fehlermeldungen werden in Attributauswertern ausgelöst, wenn die Auswertung einer Kontextbedingung an einem bestimmten Knoten das angegebene Prädikat nicht erfüllt. Der Kontextbedingung sollte ein spezifischer, die Fehlersituation beschreibender Text zugeordnet sein, der zusammen mit der Quellposition des Knotens ausgegeben wird. Unter diesen Randbedingungen führen folgende Maßnahmen zur Qualitätsverbesserung:

- Eine Kontextbedingung sollte nicht mehrere Prädikate verknüpfen und einer einzigen Meldung zuordnen. Statt dessen gibt man besser für jedes Prädikat eine separate Kontextbedingung mit einem spezifischen Fehlertext an.

Gute Positionierung

- Kontextbedingungen sollten dem betroffenen Syntaxkonstrukt zugeordnet sein, damit die Fehlermeldung eine möglichst zutreffende Quellposition erhält. Ggf. muß die zur Prüfung notwendige Information durch zusätzliche (meist erworbene) Attribute in den geeigneten Kontext transportiert werden. Ein gutes Beispiel dafür ist die Prüfung der Parametertypen in Abbildung 5.1-2; Typfehler werden am betroffenen Parameter festgestellt und gemeldet. Schlechte Fehlermeldungen würde man in diesem Beispiel erhalten, wenn man statt der erworbenen Attribute für die formalen Parameter abgeleitete Attribute für die Typen der

aktuellen Parameter berechnen und im Kontext des Aufrufs die Listen der formalen und aktuellen Parametertypen miteinander vergleichen würde. Die Fehlermeldung würde dann am Aufruf positioniert und ihr Text müßte dann etwa unspezifisch lauten "einer der Parameter hat unzulässigen Typ".

Die Spezifikation der semantischen Analyse durch eine attributierte Grammatik und die modulare Zerlegung in Teilaufgaben (Attributierung und Datenmodule) wirkt sich auch positiv für den Entwurf der Fehlerbehandlung aus: Sie kann mit relativ geringem Aufwand nachträglich verbessert werden, was für einen ohne diese Systematik entwickelten Übersetzer kaum praktikabel wäre.

Typfehler

Um die Konsistenz der Attributregeln zur Typbestimmung zu garantieren, wird ein spezieller Fehlertyp in den Wertebereich der Typattribute aufgenommen. Er wird als Attributwert berechnet, wenn für einen fehlerhaften Ausdruck oder eine Typangabe in einer Deklaration der Typ nicht bestimmt werden kann. Um Folgefehler zu vermeiden, müssen die Funktionen für Typvergleiche und Operatoridentifikation so erweitert werden, daß sie Fehlertypen als Argumente akzeptieren ohne weitere Kontextbedingungen zu verletzen. Umgekehrt kann man verhindern, daß der Fehlertyp sich unnötig weit verbreitet und damit die Erkennung weiterer Fehler unterdrückt. So kann die Operatoridentifikation für einen Vergleichsoperator durchaus den Ergebnistyp `bool` liefern, auch wenn seine Operanden den Fehlertyp haben.

Definitionsfehler

Im Zusammenhang mit der Zuordnung von Definitionen treten zwei charakteristische Fehlersituationen auf:

- unzulässige Mehrfachdefinition im gleichen Block und
- angewandte Bezeichner ohne gültige Definition.

Mehrfachdefinitionen können beim Aufbau der lokalen Definitionslisten des Definitionsmoduls erkannt werden. Enthält die Abstraktion von Definitionen auch ihre Quellposition, so kann man in der Fehlermeldung zusätzlich auf die vorangehende Definition zum gleichen Bezeichner verweisen. Hier erfordert die Erkennung für jede Definition das vollständige Durchlaufen der soweit aufgebauten Liste lokaler Definitionen. Man kann diesen Suchaufwand vermeiden, wenn man die Prüfung in die Bezeichneridentifikation integriert und den direkten Zugriff über die Definitionskeller ausnutzt.

Das Fehlen einer Definition für einen angewandten Bezeichner stellt der Bezeichneridentifikationsmodul ohne weitere Suche anhand der leeren Bezeichnerliste fest. Die Funktion `identify` muß trotzdem eine Definition als Ergebnis liefern. Sie sollte ein Fehlerobjekt mit geeigneten Eigenschaften beschreiben (z. B. eine Variable vom Fehlertyp). Durch weitere Maßnahmen der Fehlerbehandlung kann man steuern, ob und an welchen Anwendungen desselben Bezeichners das Fehlen der Definition erneut angezeigt wird:
1) Die Fehlerdefinition wird ohne weitere Maßnahme als Ergebnis geliefert. Konsequenz: Der Fehler wird an jeder Anwendungsstelle des Bezeichners gezeigt.
2) Eine Fehlerdefinition wird in die Liste der lokalen Definitionen des kleinsten umfassenden Blocks eingefügt. Konsequenz: Der Fehler wird für diesen Bezeichner im selben Block nicht wiederholt.
3) Eine Fehlerdefinition wird in die Liste der lokalen Definitionen des größten umfassenden Blocks eingefügt. Konsequenz: Der Fehler wird für keine Anwendung desselben Bezeichners wiederholt.

Alternative (1) erfordert den geringsten Implementierungsaufwand, (2) liefert dem Benutzer die am besten differenzierte Information, (3) produziert die wenigsten Meldungen. Die Entscheidung zwischen den Alternativen muß unter Berücksichtigung der Randbedingungen für die Übersetzerimplementierung getroffen werden.

6.4 Zwischencode-Erzeugung

Den Abschluß der Analysephase und die Schnittstelle zur Synthesephase bildet die *Zwischencode-Erzeugung*. Sie transformiert die Analyseergebnisse in eine für die Synthese geeignete Form. Solche übersetzerinternen Programmrepräsentationen werden Zwischensprachen genannt. Nur für sehr einfache Übersetzungen oder bei großer Ähnlichkeit zwischen Quell- und Zielsprache ist es zweckmäßig, diesen Transformationsschritt wegzulassen und direkt in die Zielsprache zu übersetzen.

Der attributierte Strukturbaum repräsentiert zusammen mit dem Definitionsmodul und den Typattributen alle für die Übersetzung notwendigen Eigenschaften des Quellprogrammes. Folgende Überlegungen begründen, weshalb es notwendig ist, ihn in eine Zwischensprache zu transformieren:

Struktur vereinfachen

Elimination von Analyseredundanz. Der Strukturbaum beschreibt die abstrakte Struktur des Quellprogramms. Sie enthält im allgemeinen Konstrukte, die nur für die Analyse, nicht aber für die Synthese relevant sind. So sind z. B. die Teilbäume für Definitionen und Typangaben überflüssig, da die in ihnen enthaltene Information im Definitionsmodul und in den Typattributen repräsentiert ist. Im Anweisungsteil wird z. B. die Teilstruktur für with-Anweisungen (in Pascal und Modula-2) nur für die Analyse benötigt. Viele Attribute, die nur zur Prüfung von Kontextbedingungen oder zur Berechnung anderer Attribute benutzt werden, sind für die Synthese nicht notwendig. Durch das Eliminieren sol-

cher Strukturen und Attribute in der Zwischensprache wird die Synthese vereinfacht.

Operatoren abbilden

Zielorientierte Repräsentation von Ausdrücken. Die Ergebnisse der Identifikation überladener Operatoren (Bedeutung der Operatoren) werden im Strukturbaum durch Attributwerte repräsentiert. Die Synthese kann jedoch wesentlich vereinfacht werden, wenn man in der Zwischensprache Klassen von Operatoren, die mit unterschiedlichen Techniken auf die Zielmaschine abgebildet werden, strukturell unterscheidet, d. h. spezielle Zwischensprachkonstrukte (Baumknoten der Zwischensprachrepräsentation) für jede Klasse einführt. Solche Klassen sind z. B. arithmetische Operationen, logische Operationen, die durch Kurzauswertung übersetzt werden, oder Mengenoperationen, die auf Instruktionsfolgen (statt einer Maschineninstruktion) abgebildet werden. Darüber hinaus enthält die Typattributierung des Strukturbaumes Informationen über alle notwendigen, im Quellprogramm impliziten Anpassungen (z. B. Inhaltsoperation). Diese müssen als Anpassungsoperationen der Zwischensprache explizit eingefügt werden. Weiter werden Operanden des Quellprogramms (Variable, Indizierung, Selektionen) zerlegt in elementare Adressierungsoperationen (Variablenzugriffe, Adreßaddition, Indizierung).

Unabhängigkeit von der Quellsprache. Wird die Zwischensprache hinreichend allgemein und unabhängig von einer bestimmten Quellsprache definiert, so ist auch der sie übersetzende Syntheseteil weitgehend quellsprachunabhängig. Er kann dann insbesondere unverändert für mehrere Übersetzer verschiedener Quellsprachen verwendet werden.

Entwurf der Zwischensprache

Nach obigen Kriterien entwirft man eine geeignete Zwischensprache, sofern sie nicht schon durch Randbedingungen der Übersetzerentwicklung vorgegeben ist. Da sie nur als interne Repräsentation verwendet wird, benötigt man dafür keine konkrete Notation. Die Definition der abstrakten Syntax und der initialen Attribute reicht aus. Im folgenden geben wir ein Schema für den Entwurf einer Zwischensprache an. Sie ist baumstrukturiert. Ihre Struktur wird wie die der Quellsprache durch eine abstrakte Syntax beschrieben. Abbildung 6.4-1 zeigt beispielhaft einige ihrer Produktionen. Zu jeder Prozedur der Quellsprache wird ein Baum der Zwischensprache erzeugt.

Die Nichtterminale unterscheiden Anweisungen (`stmt`) und Ausdrücke (`expr`). Letztere liefern ein Ergebnis und können - wie z. B. in Funktionsaufrufen - auch Anweisungen enthalten. Wie in der abstrakten Syntax der Quellsprache sind auch hier nur den Terminalen Attribute zugeordnet, die beim Baumaufbau - also hier durch die Analyseattributierung - bestimmt werden. In unserem Beispiel sind solche Terminale:

`OPR` gibt den Zieloperator an, der als Ergebnis der Operatoridentifikation bestimmt wird.

`VAL` gibt den Wert von Literalen (durch Verweis in den Literalmodul) an.

`LG` gibt den Speicherumfang (als Ergebnis der Speicherabbildung von Typen der Quellsprache) für Zuweisungen, für die Elementlänge bei Reihungsindizierungen und für den Inhaltsoperator an.

`DEF` gibt eine Referenz auf eine Objektbeschreibung in dem Definitionsmodul (bzw. deren Speicherabbildung) an. Es identifiziert Variablen- und Parameterzugriffe in Ausdrücken, die Prozedursignatur in Aufrufen und den formalen Parameter in Parameterzuweisungen.

Anhand der abstrakten Syntax in Abbildung 6.4-1 soll auf einige Entwurfsentscheidungen für die Zwischensprache hingewiesen werden:

Prozeduren

Die Schachtelung von Prozeduren in der Quellsprache wird aufgelöst in eine Folge von Prozedurbäumen. Prozeduren werden im Syntheseteil jeweils einzeln verarbeitet - außer bei interprozeduraler Optimierung. Funktions- und Prozeduraufrufe enthalten die Parameterübergabe als Anweisungsfolge. Die Reihenfolge der Parameterzuweisungen (*il_param_ass*) ist von Bedeutung, falls sie von der Quellsprache vorgeschrieben ist, oder die Übergabe auf einem Keller erfolgt. Der Ausdruck in Aufrufen gibt die Berechnung der aufzurufenden Prozedur an, das `DEF`-Attribut ihre Signatur.

Operatoren

Logische Operatoren und Vergleiche werden durch spezielle Produktionen hervorgehoben, da sie zur Kurzauswertung anders behandelt werden als die übrigen Operatoren. Zugriffe auf Variable werden nach lokalen, nicht lokalen und statisch zugeteilten Variablen unterschieden. Inhaltsoperationen (`il_cont`) und Adreßrechnung (`il_add_addr`, `il_index`) werden explizit eingesetzt. Ebenso Laufzeitprüfungen (`il_check_low`, `il_check_high`), z. B. für Reihungsgrenzen. Abbildung 6.4-2 zeigt als Beispiel den Zwischensprachbaum für eine kleine Prozedur (ohne Operationen zur Grenzenprüfung).

Erzeugung der Zwischensprachbäume

Die Erzeugung von Zwischensprachbäumen wird in der Attributierung durch Attributregeln beschrieben. Dazu ordnet man jedem Nichtterminal der attributierten Grammatik, das Anweisungen oder Ausdrücke repräsentiert, ein abgeleitetes Zielattribut `il` zu. Sein Wertebereich `il_tree` ist definiert durch die Struktur der Zwischensprachbäume.

Zu jeder Produktion der abstrakten Zwischensprachsyntax gibt es eine Funktion, die aus ihren Parametern einen neuen Teilbaum bildet. Ihre Signatur entspricht der Produktion, z. B.

il_proc:	proc ::= DEF stmt
il_stmt:	stmt ::= stmt stmt
il_lab:	stmt ::= LAB stmt
il_goto:	stmt ::= LAB
il_while:	stmt ::= expr stmt
	...
il_ass:	stmt ::= LG expr expr
il_param_ass:	stmt ::= LG DEF expr
il_call:	stmt ::= DEF expr stmt
il_funct_call:	expr ::= DEF LG expr stmt
il_dy_arith:	expr ::= OPR expr expr
il_mon_arith:	expr ::= OPR expr
il_dy_float:	expr ::= OPR expr expr
	...
il_comp:	expr ::= OPR expr expr
il_and:	expr ::= expr expr
	...
il_loc_var_:	expr ::= DEF
il_non_loc_var:	expr ::= DEF
il_static_var:	expr ::= DEF
il_const:	expr ::= VAL LG
il_cont:	expr ::= LG expr
il_add_addr:	expr ::= expr expr
il_index:	expr ::= LG expr expr
il_check_low:	expr ::= LG LAB expr expr
il_check_high:	expr ::= LG LAB expr expr

Abb. 6.4-1: Abstrakte Syntax einer Zwischensprache.

```
Produktion il_ass: stmt ::= LG expr expr

function il_ass (LG : integer, e1, e2 :il_tree) : il_tree;
```

Die Repräsentation des Zwischensprachbaumes wird mit den Funktionen zu seinem Aufbau in einem Datenmodul zusammengefaßt. Die Werte der `il`-Attribute in der Attributierung sind dann Referenzen auf die in dem Modul repräsentierten Baumknoten.


```
type t = record x, y :  integer end;

var a :  array [1 ..  10] of t;

procedure p (i, v :  integer);

begin a[i].y :=  v end;
```

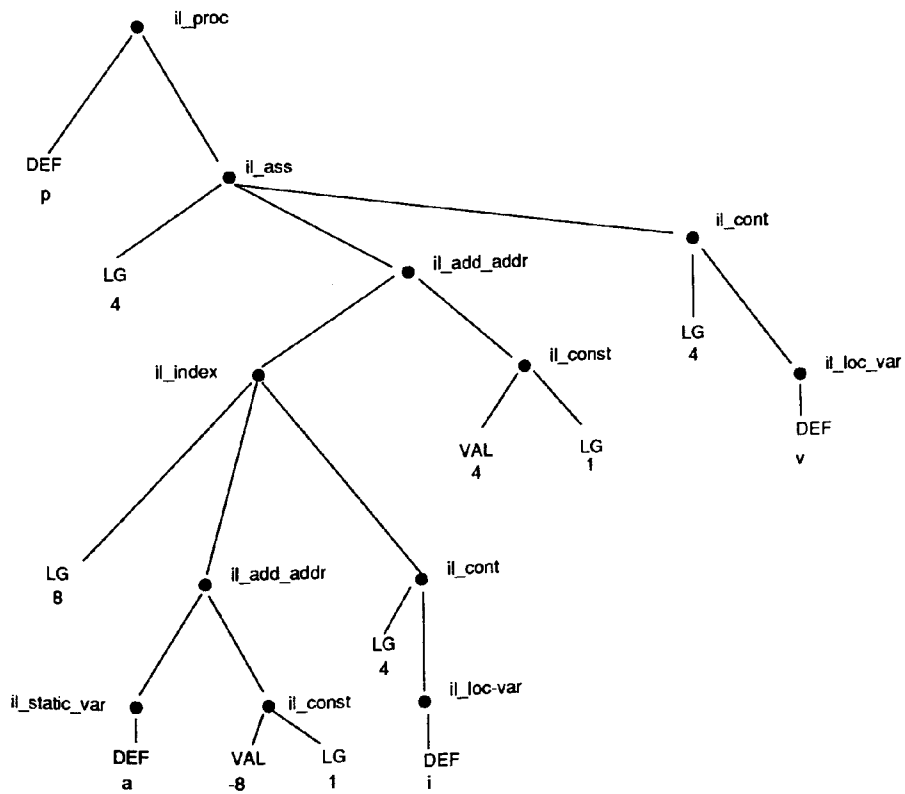


Abb. 6.4-2: Abstrakter Zwischensprachbaum für eine Prozedur.

Schnittstelle

Der Zwischensprachmodul realisiert (zusammen mit der Speicherabbildung aus dem Definitionsmodul) die Schnittstelle zwischen Analyse- und Syntheseteil. Die Art der Darstellung des Zwischensprachbau-

mes wird deshalb wesentlich durch die Strategie bestimmt, mit der er im Syntheseteil weiterverarbeitet wird. Hierzu unterscheiden wir zwei Verarbeitungsarten:
1) Der Syntheseteil verarbeitet den als Datenstruktur aufgebauten Baum. In diesem Fall benutzt die Synthese Zugriffsfunktionen und Datenstruktur des Zwischensprachmoduls.
2) Der Baum wird linearisiert und als Strom von Knoten an den Syntheseteil übergeben. Dabei wird entweder für jeden Knoten eine Verarbeitungsprozedur des Syntheseteils direkt aufgerufen, oder die Knoten werden zunächst in einem Puffer (einer Datei) zwischengespeichert.

Linearisierung

Bei der Verarbeitungsstrategie (2) kann man auf den Aufbau des Baumes als Datenstruktur im Zwischensprachmodul völlig verzichten, falls die Baumknoten bei der Attributierung in der Reihenfolge erzeugt werden, die für die Linearisierung in der Schnittstelle festgelegt wird. Die oben beschriebene Attributierung konstruiert den Zwischenbaum aufwärts (von den Blättern zur Wurzel). Ein Knoten wird also erst dann erzeugt, wenn alle seine Unterbäume vollständig aufgebaut sind. Für eine solche Postfix-Linearisierung muß man außerdem die Reihenfolge festlegen, in der die Unterbäume erzeugt werden. Wir müssen dazu in der Attributierung ausdrücken, daß die `il`-Attribute von Unterbäumen in bestimmter Reihenfolge berechnet werden. Dazu erweitern wir die Attributierung systematisch: Jedem Nichtterminal für Anweisungen und Ausdrücke ordnen wir ein Attribut `pre` zu und erzwingen mit zusätzlichen Attributregeln die gewünschte Auswertungsreihenfolge. Die Attributierung der Produktion für eine Zuweisung würde dann etwa wie folgt erweitert:

```
Anweisung ::= Variable Ausdruck
    ...
    Variable.pre := Anweisung.pre;
    Ausdruck.pre := dep (Variable.il);
    Anweisung.il := il_ass (Variable.lg, Variable.il, Ausdruck.il)
```

An den Regeln für Blätter des Baumes wird der Berechnung des `il`-Attributs eine Abhängigkeit vom Attribut `pre` hinzugefügt. Damit garantieren die Attributabhängigkeiten, daß die `il`-Attribute in Postfixreihenfolge berechnet und die `il`-Funktionen in dieser Reihenfolge aufgerufen werden.

Nachdem die Auswertungsreihenfolge des Attributauswerters (z. B. in Form von Besuchssequenzen) bestimmt ist, kann man die `pre`-Attribute und ihre Berechnung eliminieren. Da sie nur Reihenfolgeabhängigkeiten ausdrücken, sind ihre Werte nicht relevant.

7. Code-Erzeugung

Überblick

In diesem Kapitel behandeln wir die zentralen Aufgaben im Syntheseteil des Übersetzers. Die *Code-Erzeugung* übersetzt Programme, die als Ergebnis der Analyse in der Darstellung der Zwischensprache vorliegen, in Programme der Zielmaschine. Dazu müssen die Objekte der Quellsprache auf die Speicherstrukturen der Zielmaschine und die Operationen der Zwischensprache auf Instruktionsfolgen der Zielmaschine abgebildet werden. Ersteres ist im wesentlichen eine Entwurfsaufgabe, die sich in der Berechnung zusätzlicher Eigenschaften zum Definitionsmodul niederschlägt. Der Entwurf der Operationsabbildung wird durch spezielle, systematische Verfahren zur Code-Auswahl und Registerzuteilung algorithmisch umgesetzt. Im Gegensatz zu den Lösungsverfahren des Analyseteils gehen hier die Ansätze zur Systematisierung von dem Transformationsverfahren aus statt von den Strukturen der Übersetzungsquelle oder dem schwierig zu formalisierenden Übersetzungsziel. Deshalb hat in diesem Kapitel die Diskussion von Entwurfsaspekten gegenüber der Präsentation von Algorithmen eine größere Bedeutung als in den vorangehenden Kapiteln.

Die hier beschriebenen Verfahren und Entwurfstechniken sind in modifizierter Form auch anwendbar, wenn andere als Maschinensprachen das Ziel der Transformation sind, etwa niedere oder höhere Programmiersprachen oder Datenstrukturen, die von anderen Programmsystemen weiterverarbeitet werden. Mit den Verfahren der Registerzuteilung kann man z. B. die Verwendung von Hilfsvariablen bei der Übersetzung in höhere Sprachen bestimmen. Man kann sie auch abstrakter als Verfahren zur Lösung von minimierenden Zuordnungsproblemen für Bäume oder Graphen verstehen und außerhalb des Übersetzerbaus anwenden, z. B. zur Minimierung von Datenpfaden in der Hardware-Synthese.

Der Entwurf der Code-Erzeugung erfordert neben umfangreicher Detailarbeit die Festlegung zahlreicher Entwurfsentscheidungen, die häufig wechselweise voneinander abhängen (z. B. die Speicherabbildung und die Zugriffsoperationen) und die Korrektheit und die Qualität des erzeugten Codes bestimmen. Es ist deshalb unumgänglich, daß vor Beginn der Implementierung die Abbildung auf die Zielmaschine präzise und vollständig spezifiziert wird. Ziel des Entwurfs ist die Erstellung eines solchen Dokuments, das auch nach Fertigstellen des Übersetzers Grundlage seiner Wartung und eventuell notwendigen Adaption ist.

Die Eigenschaften der Zielmaschine werden im Hinblick auf die zu entwerfende Abbildung analysiert (Abschnitt 7.1). Für die Zielmaschine wird ein Speichermodell entworfen, auf das die Datentypen und Datenobjekte der Quellsprache abgebildet werden (Abschnitt 7.2). Schließlich

ordnet man jedem Operator der Zwischensprache mindestens eine dafür zu erzeugende Instruktionsfolge zu, in die der Code für die Operanden des Operators eingefügt wird (Abschnitt 7.3). Insbesondere für Operatoren in Ausdrücken werden jeweils mehrere Code-Sequenzen als mögliche Übersetzungen entworfen, zwischen denen kontextabhängig und mit Kriterien der Code-Qualität (Umfang, Ausführungszeit) ausgewählt wird.

Struktur des Syntheseteils

Das Ergebnis des Entwurfs ist ein Dokument, das die Code-Erzeugung spezifiziert und aus dem die Implementierung entwickelt wird. Abbildung 2.1-3 zeigt die modulare Struktur der Code-Erzeugung im Syntheseteil. Die Schnittstelle zum Analyseteil wird durch den Zwischencode und die Objektbeschreibungen im Definitionsmodul gebildet. Der zentrale Modul der Synthese leistet die Code-Auswahl. Er bestimmt, nach welcher Code-Sequenz jeder Baumknoten übersetzt wird. Hierfür müssen im allgemeinen umfangreiche Fallunterscheidungen systematisch implementiert werden (Abschnitt 7.4). Diese Aufgabe kann durch den Einsatz generierender Werkzeuge erleichtert werden. Bei der Übersetzung von Ausdrücken fallen Zwischenergebnisse an, die für eine große Klasse von Zielmaschinen in Registern untergebracht werden. Diese Zuordnung kann nach verschiedenen Strategien im Modul Registerzuteilung implementiert werden (Abschnitt 7.5). Für Zielprozessoren mit stark vereinfachtem Instruktionssatz (RISC) reduziert sich auch die Komplexität der Code-Auswahl, wenn es für die Abbildung von Grundoperationen nur wenige Alternativen gibt. Statt dessen muß die Zusammensetzung mehrerer Code-Sequenzen im größeren Kontext mit Optimierungsmethoden (Kap. 8) verbessert werden. Auch kommt für diese Prozessorklasse, die meist über einen großen Registersatz verfügt, der optimierenden Registerzuteilung eine besondere Bedeutung zu.

Für die Repräsentation des Zielprogramms in der Maschinensprache ziehen wir eine weitere Schnittstelle innerhalb des Syntheseteils ein: den abstrakten Maschinencode. Er repräsentiert das Maschinenprogramm als Folge von Tupeln der Maschineninstruktionen mit ihren Operanden. Sie wird vom Assemblierer in das Befehlsformat der Zielmaschine codiert und mit zusätzlichen Informationen zum Binden und für die Anwendung von Testhilfen versehen im Code-Dateiformat des Zielsystems ausgegeben (Abschnitt 7.6). Für die Aufgaben der Assemblierung stehen im allgemeinen auch eigenständige Assemblererprogramme zur Verfügung, die Codedateien aus symbolischem Maschinencode erzeugen. Um die Übersetzungsgeschwindigkeit zu steigern und den Anschluß von quellsprachbezogenen Testhilfen zu ermöglichen, integriert man diesen Modul jedoch in den Übersetzer.

Optimierung

Die Qualität des erzeugten Maschinencodes ist entscheidend für die Entwicklung von praktisch brauchbaren Übersetzern. Die Methoden zur Code-Erzeugung, die wir in diesem Kapitel vorstellen, schränken die möglichen Maßnahmen zur Code-Verbesserung auf die Code-Auswahl- und die Registeerzuteilung ein. Dabei wird hier die Annahme gemacht, daß der Code entlang der Baumstruktur des Zwischensprachprogramms zusammengesetzt und Kontextinformation nur zwischen benachbarten Knoten ausgenutzt wird. Praktische Entwicklungen haben gezeigt, daß so durchaus brauchbare Übersetzer hergestellt werden können. Höhere Anforderungen an die Code-Qualität werden erfüllt, wenn man Optimierungen, wie wir sie in Kapitel 8 besprechen, integriert. Jene Verfahren analysieren die Zwischensprachbäume in größerem Kontext mit dem Ziel sie verbessernd zu transformieren. Auch der abstrakte Maschinencode kann in einer Nachoptimierungsphase weiter verbessert werden.

Die Synthese für Prozessoren mit interner Parallelverarbeitung wirft zusätzliche Probleme der Code-Anordnung und der Schleifentransformation bei Vektorprozessoren auf. Sie können nicht mit den in diesem Kapitel beschriebenen Verfahren gelöst werden. Hierfür sind Analysen und Transformationen in größerem Kontext nötig, die auf Optimierungsverfahren aus Kapitel 8 basieren.

7.1 Eigenschaften der Zielmaschine

Voraussetzung für den Entwurf ist ein gründliches Studium des Zielprozessors und seines Instruktionssatzes, möglichst anhand der Originaldokumentation. Diejenigen Eigenschaften des Prozessors, die für die Übersetzung relevant sind, und die für die Übersetzung schließlich verwendeten Instruktionen müssen in diesen Teil des Entwurfsdokuments aufgenommen werden. Ebenso müssen für die Übersetzung fehlende Eigenschaften (z. B. Gleitpunktoperationen), die durch Software-Lösungen (Code-Sequenzen, Bibliotheksroutinen) zu ersetzen sind, aufgezeigt werden. In diesen und den folgenden Abschnitten beziehen wir uns in Beispielen hauptsächlich auf den Prozessor Motorola M68000. Trotz seines beträchtlichen Alters und existierender Nachfolgemodelle sind seine Eigenschaften noch typisch für eine große Klasse von Prozessoren. Wir gliedern die Beschreibung der Maschineneigenschaften in die Darstellung der Speicherklassen, der Instruktionsoperanden und der Instruktionen.

7.1.1 Speicherklassen

Speicher

Der *Hauptspeicher* besteht aus Speicherzellen, auf deren Inhalt mit Adressen (positive Zahlen) direkt zugegriffen werden kann. Ist die kleinste adressierbare Speicherzelle ein Byte (8 Bit), so sprechen wir von einem *byteorientierten* Speicher, sonst von *wortorientiertem* Speicher mit bestimmtem Umfang eines Speicherwortes (z. B. 32, 48, 60 Bit). Jegliche Adreßrechnung erfolgt in Einheiten der kleinsten adressierbaren Speicherzelle.

Auch im Falle byteorientierter Speicher kann meist auf aufeinanderfolgende Speicherzellen als größere Einheit zugegriffen werden. Man verwendet dafür dann auch Begriffe wie Wort, Halbwort, Langwort. Der M68000 hat einen byteorientierten Speicher und Zugriffe auf Bytes, Worte (= 2 Bytes) und Langworte (= 4 Bytes).

Häufig werden für Zugriffe auf größere Speichereinheiten Randbedingungen an die Speicheradresse gestellt: Beim M68000 kann auf Worte oder Langworte nur mit geraden Adressen zugegriffen werden. Solche Restriktionen müssen bei der Speicherabbildung durch *Ausrichtung* (*alignment*) der Objektadressen berücksichtigt werden.

Die im erzeugten Code verwendeten Hauptspeicheradressen sind Relativadressen bezogen auf den Anfang eines Speichersegments. Für den Programmcode, initialisierte und nichtinitialisierte Daten, Laufzeitkeller, Halde werden im allgemeinen verschiedene Speichersegmente angelegt. Der Binder fügt diese Segmente zusammen und adjustiert die auf ihren Anfang bezogenen Adressen.

Register

Register sind Speicherelemente des Prozessors selbst und deshalb besonders schnell zugreifbar. Sie nehmen Operanden und Ergebnisse von Instruktionen auf und werden zur Adreßrechnung benutzt. Die Menge der verfügbaren Register ist häufig in Klassen für unterschiedliche Verwendungsarten eingeteilt:

- allgemeine Register für beliebige Verwendung,
- Datenregister für arithmetische und logische Operationen (eventuell spezielle Gleitpunktregister),
- Adreßregister zur Bildung von Speicheradressen,
- Basisregister zur Adressierung von Speichersegmenten,
- Indexregister zur Verknüpfung mit Speicheradressen.

Der M68000 enthält z. B. 8 Datenregister (D0, .., D7) und 8 Adreßregister (A0, .., A7). Einige Instruktionen mancher Prozessoren (z. B. IBM 370) benutzen Registerpaare, um umfangreichere Operanden aufzunehmen (z. B. Ergebnisse der Ganzzahlmultiplikation).

Einigen Registern kann eine spezielle Bedeutung dadurch zugeordnet sein, daß sie bei bestimmten Instruktionen implizit verwendet werden. Dies sind z. B.

- Instruktionszeiger — enthält die Adresse der nächsten auszuführenden Instruktion,
- Akkumulator — nimmt einen Operanden und das Ergebnis von Berechnungen auf (nur auf älteren Prozessoren),
- Kellerpegel — für Kelleroperationen,
- Bedingungscode — nimmt das Ergebnis von Vergleichen auf und steuert bedingte Sprünge.

Einige Prozessoren, insbesondere der RISC-Klasse (z. B. RISC II), verwalten einen Registerkeller: Ein Keller im Hauptspeicher wird durch die Operationen Prozedureintritt und Prozedurrückkehr um jeweils n Elemente vergrößert bzw. verkleinert. Die obersten $m \geq n$ Elemente sind gleichzeitig als Register schnell zugreifbar. Damit steht jedem Prozeduraufruf ein Satz lokaler Register zur Verfügung. In den zwischen aufrufender und aufgerufener Prozedur überlappenden Registern können Parameter übergeben werden (*register windows*).

7.1.2 Operanden und Adressierungsarten

Operanden und Adressierungsarten

Nach der Anzahl der expliziten Operanden einer Instruktion unterscheiden wir 0-, 1-, 2- und 3-Adreßinstruktionen. Wir sprechen auch verallgemeinernd von einer *m-Adreßmaschine*, wenn typische zweistellige Verknüpfungen (`a := b op c`) durch m-Adreßinstruktionen erfolgen:

3-Adreßinstruktionen	`A3 := A1 op A2`
2-Adreßinstruktionen	`A1 := A1 op A2`
1-Adreßinstruktionen	`Akk := Akk op A1` `Akk` ist ein impliziter Akkumulator
0-Adreßinstruktionen	`push (pop op pop)` Kelleroperationen

Die Operanden können durch verschiedene *Adressierungsarten* der Instruktion angegeben werden, z. B. für den M68000:

c	konstanter Wert in der Instruktion,
R_i	Registerinhalt,
$R_i + c$	Summe aus Registerinhalt und einem konstanten Wert,
$R_i + R_j$	Summe der Inhalte zweier Register,
$R_i + R_j + c$	Summe der Inhalte zweier Register und eines konstanten Wertes,
$-R_i$	Registerinhalt nach Erniedrigen um die Operandenlänge,
$R_i +$	Registerinhalt vor Erhöhen um die Operandenlänge.

Das Ergebnis der obigen Adressierungsformeln ist die *effektive Adresse* des Operanden. Sie wird je nach Bedeutung der Instruktion selbst als Operand verwendet (*Direktoperand*) oder gibt die Speicheradresse des Operanden an (*Speicheroperand*). Mehradreßmaschinen schränken die zulässigen Kombinationen der Adressierungsarten der Operanden einer Instruktion häufig ein, z. B. `A1` in der 2-Adreßform muß ein Register sein. Einige Prozessoren (auch der M68000) erlauben auch Speicheroperanden für das Verknüpfungsergebnis. Typische RISC-Prozessoren erlauben Speicheroperanden nur in Lade- und Speicherinstruktionen.

7.1.3 Instruktionen

Instruktionsklassen

Instruktionen der folgenden Klassen sind insbesondere für die Übersetzung allgemein anwendbarer Programmiersprachen von Bedeutung:

- Datentransport zwischen Speicherzellen oder Register und Speicher (`load, store, move`), ggf. auch für größere Speicherblöcke,
- ganzzahlige Operationen unterschieden nach Operandenlängen und Interpretation der Operanden als vorzeichenbehaftet oder vorzeichenlos (z. B. für Adreßrechnungen),
- Gleitpunktoperationen verschiedener Länge (fehlen für manche Prozessoren, z. B. M68000, oder werden von Koprozessoren übernommen),
- logische Operationen als Verknüpfung von Bitfolgen,
- Shift-Instruktionen, arithmetisch (zur Multiplikation bzw. Division mit Zweierpotenzen) oder logisch (z. B. zum Isolieren von Teilen von Speicherzellen oder für Mengenoperationen),
- Vergleiche unterschieden nach vorzeichenlosen und vorzeichenbehafteten Operanden verschiedener Länge; das Ergebnis fällt ggf. im Bedingungscode-Register an,
- Sprünge, bedingt, unbedingt, mit direktem, indirektem oder indiziertem Sprungziel, Unterprogrammsprünge und Rückkehrsprünge,
- Umgebungskontrolle wie Prozeduraufruf und Rückkehr, Systemaufrufe, Unterbrechungen.

7.2 Speicherabbildung

Speicherumfang
Relativadressen

Die Speicherabbildung ordnet den Datenobjekten des Quellprogramms Speicherzellen innerhalb von *Speicherblöcken* zu. Zu jedem Objekt werden sein *Speicherumfang* und seine *Relativadresse* bezogen auf den Anfang des Speichers unter Berücksichtigung der eventuell notwendigen Ausrichtung bestimmt. Die Adressierung von Speicherblöcken berechnet man für die globalen Daten des Programms (Datensegment), der Prozedurschachteln und Verbunde. (Speicherblöcke können ineinander geschachtelt sein: Verbunde als Komponenten in Verbunden, Verbundvariable in Prozedurschachteln). Für jeden Speicherblock ist seine Adressierungsrichtung festzulegen: von 0 aus wachsende oder fallende Relativadressen. Einige Datenobjekte werden eventuell zur Verarbeitung in Register geladen, deren Umfang größer ist als der Speicherumfang des Objektes (z. B. in einem Byte gespeicherte Zeichen verarbeitet in 4 Bytes langen Registern). Hier ist sicherzustellen, daß beim Datentransport korrekt auf den Verarbeitungsumfang aufgefüllt bzw. der Speicherumfang ausgeblendet wird.

Grundlage für den Entwurf der Speicherabbildung sind die folgenden Überlegungen zur Abbildung einfacher und zusammengesetzter Datentypen sowie die Beschreibung des Maschinenzustands. Die Speicherabbildung wird implementiert, indem man in dem Definitionsmodul zu jedem Datentyp Umfang und notwendige Ausrichtung und zu jedem Datenobjekt (Variablen, Parameter, Verbundkomponenten) Umfang und Relativadresse berechnet und einträgt. Für Variablen muß außerdem angegeben werden, in welchem Speichersegment sie untergebracht werden. Um das Definitionsmodul so zu ergänzen, werden die Einträge in der Reihenfolge ihrer Anordnung in ihrem Speicherblock durchlaufen.

7.2.1 Datentypen

Grundtypen

Die Speicherabbildung der *arithmetischen Grundtypen* legt man (soweit die Quellsprache erlaubt) in Abhängigkeit der verfügbaren Instruktionen zu ihrer Verarbeitung fest, um Effizienzverluste durch Differenz zwischen Verarbeitungs- und Speicherumfang zu vermeiden. Die Darstellung arithmetischer Werte ist durch das Datenformat der Zielmaschine vorgegeben (bzw. durch Bibliotheksfunktionen für im Instruktionssatz fehlende Gleitpunktarithmetik).

Logische Werte werden in der kleinsten adressierbaren Speichereinheit (z. B. Byte) untergebracht. Die Codierung der Werte `true` und `false` kann aus den Möglichkeiten (`false = 0, true = 1`), (`false = 0, true ≠ 0`), (`false > 0, true = 0`) oder deren Vertauschung gewählt werden, falls die Quellsprache dies nicht festlegt. Dabei ist zu berücksichtigen, daß Vergleiche häufiger vorkommen als die Operatoren `and` und `or`. Werden Vergleiche durch Sprünge und `and` und `or` durch Kurzauswertung realisiert, so wählt man eine Codierung, die sich durch eine einfache Instruktion negieren läßt.

Zeichenwerte werden in Bytes gespeichert. Ihre Codierung in einem Standardformat (z. B. ASCII) wird entweder von der Quellsprache vorgeschrieben oder passend zu den E/A-Routinen des Grundsystems gewählt.

Referenztypen werden (ebenso wie Referenzparameter) auf das Adreßformat der Zielmaschine abgebildet.

Aufzählungs- und *Ausschnittstypen* werden wie ganze Zahlen behandelt. Da Aufzählungstypen im allgemeinen nur kleine Kardinalität haben und außer Inkrementieren, Dekrementieren und Vergleichen keine Operationen angewandt werden, kann auch ein passender Speicherumfang kleiner als der für ganze Zahlen gewählt werden.

Verbunde

Verbundtypen beschreiben Speicherblöcke, in denen die Komponenten hintereinander unter Berücksichtigung ihrer Ausrichtung angeordnet werden. Die Speicherabbildung bestimmt deren Relativadresse bezogen auf den Anfang des Speicherblocks und den Gesamtumfang

des Speicherblocks. Solch ein Speicherblock muß gemäß der strengsten Ausrichtungsforderung seiner Komponenten ausgerichtet werden. Falls es die Quellsprache erlaubt, kann die Reihenfolge der Komponenten so bestimmt werden, daß der Ausrichtungsverschnitt minimal wird (z. B. durch Anordnen nach fallendem Speicherumfang). Blöcke von parallelen *Varianten* werden überlagert. Die umfangreichste Variante bestimmt den Speicherumfang des Variantenblocks.

Mengen

Mengen werden auf Bitvektoren abgebildet, die durch logische Instruktionen für Mengenoperationen verknüpft werden. Wenn es die Quellsprache erlaubt, wählt man den maximalen Speicherumfang, der durch eine einzige logische Operation verknüpft werden kann. Dadurch sollte die maximale Kardinalität von Mengen jedoch nicht so beschränkt werden, daß (häufig verwendete) Mengen von Zeichen ausgeschlossen werden. Läßt sich dies nicht realisieren, wählt man je nach Kardinalität einen größeren Speicherumfang und Instruktionsfolgen für Mengenverknüpfungen. Falls die Quellsprache Mengen über negative Werte oder Mengen kleiner Kardinalität über große Werte zuläßt, wie

```
set of -10..10
set of 10000..10020
```

so müssen die Werte von Mengenelementen transponiert werden (in den obigen Beispielen auf den Bereich 0..20). Dazu werden anhand der Typanalyse bei der Zwischenspracherzeugung Anpassungsoperationen generiert.

Reihungen

Reihungen mit beliebig vielen Indexstufen werden entweder auf einen zusammenhängenden Speicherbereich linearisiert abgebildet oder als Baumstruktur gestreut im Speicher untergebracht.

Die Linearisierung einer s-stufigen Reihung

```
a : array [m1 .. n1 , ... , ms .. ns] of t
```

wird durch folgende Kenngrößen bestimmt:

E	Speicherumfang eines Reihungselementes vom Typ t (einschließlich des ggf. notwendigen Ausrichtungsverschnitts),
m_i , n_i	Unter- und Obergrenze der i-ten Indexstufe,
$k_i = n_i - m_i + 1$	Indexspanne der i-ten Stufe,
$K = k_1 * \ldots * k_s$	Anzahl der Elemente,
$U = K*E$	Speicherumfang der Reihung,
$A =$ adr (a[m_1 , ..., m_s])	Anfangsadresse der Reihung innerhalb des umgebenden Speicherblocks.

Bei *zeilenweiser* Anordnung der Elemente ergibt sich folgende Speicherabbildungsfunktion für den Zugriff auf ein Element:

$$\text{adr}(a[j_1, \ldots, j_s]) = A + (j_1 - m_1) * k_2 * \ldots * k_s * E + \ldots + (j_s - m_s) * E$$

Die in diesem Ausdruck nicht von den Indizes j_i abhängigen Terme können isoliert und zusammengefaßt werden:

$$F = A - (m_1 * k_2 * \ldots * k_s * E) - \ldots - m_s * E$$

$$\begin{aligned}\text{adr}(a[j1, \ldots, js]) &= F + (j_1 * k_2 * \ldots * k_s * E) + \ldots + j_s * E \\ &= F + (\ldots (j_1 * k_2 + j_2) * k_3 + \ldots + j_s) * E\end{aligned}$$

Für F gilt offensichtlich $F = \text{adr}(a[0, \ldots, 0])$. Da dieses Element nicht notwendig zu den definierten Reihungselementen gehört, bezeichnen wir F auch als *fiktive Anfangsadresse*. Sie kann aus den Kenngrößen der Reihung berechnet und zur Vereinfachung der Indizierungsfunktion verwendet werden. Sie wird in Zwischensprachoperationen zur Adreßrechnung umgesetzt. Dabei kann zusätzlich die Zulässigkeit jedes Index $m_i \le j_i \le n_i$ überprüft werden.

Sind die Indexgrenzen der Reihung a zur Übersetzungszeit bekannt, so können F, E und die k_i als konstante Operanden in den Zugriffscode eingesetzt werden.

Werden die Indexgrenzen erst zur Laufzeit bei der Auswertung der Reihungsdeklaration (wie in Algol 60) bestimmt, so legt man zusätzlich einen *Deskriptor* als Datenstruktur an, der die Kenngrößen

$$F, m_1, k_1, \ldots, m_s, k_s$$

aufnimmt. E und s sind im allgemeinen zur Übersetzungszeit bekannt. Die Werte der Deskriptorkomponenten werden durch geeigneten Code für die Reihungserzeugung bestimmt und in den Zugriffsoperationen entsprechend verwendet. Struktur und Umfang des Deskriptors liegen zur Übersetzungszeit fest und werden bei der Speicherzuteilung in den aufnehmenden Speicherblock eingefügt. Der Speicherbereich für die Reihungselemente, dessen Umfang sich erst zur Laufzeit ergibt, wird entweder am Ende des Speicherblocks untergebracht oder dynamisch (z. B. auf der Halde) beschafft.

Fordert die Quellsprache, daß sich mehrstufige Reihungen bei Zugriffen auf Teilreihungen so verhalten, als ob sie in der Form

```
a: array [m1 .. n1] of array [m2 .. n2] of ... of t
```

deklariert wären, so müssen sie wie oben beschrieben zeilenweise linearisiert werden. Fortran hingegen fordert z. B. explizit die *spaltenweise* Anordnung der Elemente. Die Speicherabbildung dafür ergibt sich durch Transposition der Indexstufen in den obigen Formeln.

Alternativ zu der linearisierten Form kann man mehrstufige Reihungen auch als Baumstruktur gestreut im Speicher unterbringen. Dabei wird eine Reihung

```
a: array [m1 .. n1 , ..., ms .. ns] of t
```

implementiert, als ob sie in der Form

```
b: ↑array [m1 .. n1] of
   ↑array [m2 .. n2] of
       ...
   ↑array [ms .. ns] of t
```

deklariert wäre. Die Indizierung auf jeder Indexstufe enthält dann einen zusätzlichen Indirektionsschritt:

```
a[j1 , ..., js]
```

wird implementiert als

```
b ↑ [j1] ↑ [j2]  ... ↑ [js].
```

Die Reihungen von Referenzen der einzelnen Stufen und die Elementblöcke der höchsten Stufe s werden dynamisch bei der Generierung erzeugt. Dieses Verfahren ist dann vorteilhaft, wenn auf nur wenige Elemente großer Reihungen tatsächlich zugegriffen wird und die dynamische Speicherverwaltung dafür sorgt, daß nur die dafür notwendigen Teilbäume aufgebaut werden.

Prozeduren

Prozeduren müssen auch als Datenobjekte bei der Speicherabbildung berücksichtigt werden, falls die Quellsprache Prozedurvariablen (wie in C und Modula-2) oder Prozeduren als Parameter (Pascal, C, Modula-2) erlaubt. Im einfachsten Fall wird ein solches Objekt durch die Anfangsadresse des Prozedurcodes implementiert. Dies ist ausreichend, falls bei jedem Aufruf des Parameter- oder Variablenwertes die Umgebung, in der die aufgerufene Prozedur auszuführen ist, zur Übersetzungszeit bestimmt ist: In C ist dies immer das Hauptprogramm, da Prozeduren nicht geschachtelt werden. In Modula-2 ist dies ebenfalls immer das Hauptprogramm, da nur dort deklarierte Prozeduren Werte von Variablen und Parametern sein dürfen. In Sprachen wie Pascal, wo geschachtelte Prozeduren Werte von Parametern sein können, wird eine

Prozedur erst durch das Paar Code-Adresse und Adresse der Umgebungsschachtel auf dem Laufzeitkeller identifiziert.

7.2.2 Maschinenzustand

Die dynamische Semantik der Quellsprache definiert die Abarbeitung eines Quellprogramms in den Begriffen einer abstrakten Maschine. Ihr Zustandsmodell wird auf die Speicherstrukturen der realen Zielmaschine abgebildet. Diese Abbildung legt die Verwendung des Hauptspeicheradreßraumes und der Maschinenregister fest.

Datenobjekte des Quellprogramms, die die gleiche Lebensdauer haben oder auf die gemeinsam zugegriffen wird, faßt man in Speicherblöcken zusammen, z. B. für globale Daten, Prozedurschachteln und Verbunde. Ihre innere Struktur (Adressierung ihrer Komponenten) wird mit der Abbildung der Datentypen bestimmt. Die Speicherblöcke werden entweder in einem statisch adressierten Bereich des Hauptspeichers untergebracht, im Laufzeitkeller oder auf der Halde verwaltet.

Statisch adressierte Speicher

Statisch adressierte Speicherbereiche nehmen die globalen Datenobjekte des Hauptprogramms (bzw. der Übersetzungseinheiten) auf. Da Binder und Lader des Grundsystems im allgemeinen erlauben, den Speicher sehr effizient zu initialisieren, ist es zweckmäßig bei der Bildung der Speicherblöcke globale Objekte, deren Initialwerte zur Übersetzungszeit bekannt sind, separat zusammenzufassen. Ebenso kann man auch die Werte der im Programm benutzten Literale behandeln, die nicht als Direktoperanden in Instruktionen eingesetzt, sondern als Datenobjekte gespeichert werden (z. B. Zeichenreihen, Gleitpunktzahlen). Für Quellsprachen ohne rekursive Aufrufe (wie Fortran) können die Speicherblöcke aller Prozeduren statisch adressiert werden. Jeder Aufruf einer Prozedur verwendet dann dieselben Speicherelemente für die lokalen Prozedurobjekte. Ein Laufzeitkeller entfällt in diesem Fall.

Laufzeitkeller

Der *Laufzeitkeller* nimmt für Quellsprachen mit rekursiven Aufrufen die Speicherblöcke mit den lokalen Prozedurobjekten (*Prozedurschachteln, activation records*) auf. Bei Aufruf einer Prozedur wird ein für sie in Umfang und Struktur bestimmter Speicherblock gekellert und nach Abarbeiten des Prozedurrumpfes wieder entkellert. Als Voraussetzung für die Korrektheit dieser Implementierung muß garantiert sein, daß auf Objekte eines Prozeduraufrufs nach dessen Abarbeiten - dem Ende ihrer Lebensdauer - nicht mehr zugegriffen wird. Diese Bedingung kann z. B. durch folgende Spracheigenschaften verletzt werden:

- Referenzvariablen, die Adressen von Objekten als Werte annehmen können, welche eine kürzere Lebensdauer als die Variablen selbst haben (*dangling reference*),
- Prozeduren als Funktionsergebnis oder Prozedurvariablen, die auf lokal gebundene Objekte bei ihrem Aufruf außerhalb ihrer Umgebung zugreifen.

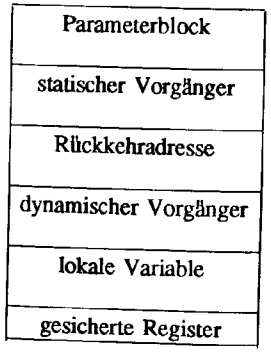


Abb. 7.2-1: Schachtelstruktur.

Struktur der Prozedurschachteln

Der Laufzeitkeller enthält zu jedem Zeitpunkt der Abarbeitung des Programms eine Prozedurschachtel für jeden noch nicht abgeschlossenen Prozeduraufruf. Neben den beiden Speicherblöcken für Parameter und Funktionsergebnis sowie für die lokalen Variablen, enthält die Schachtel weitere Objekte zur Implementierung der Kelleroperationen und des Aufrufs sowie zum Zugriff auf Objekte, die nicht lokal zur aufgerufenen Prozedur sind und nicht statisch adressiert werden. Dies sind

- die *Rückkehradresse*, an der die Programmausführung nach Abarbeiten des Prozedurrumpfs fortgesetzt wird,
- die Adresse der auf dem Keller vorangehenden Prozedurschachtel (*dynamischer Vorgänger*), aus der heraus der Aufruf erfolgte,
- die Adresse der Prozedurschachtel, in der die aufgerufene Prozedur deklariert ist (*statischer Vorgänger*); in Sprachen mit geschachtelten Prozeduren gibt die Kette der statischen Vorgängerschachteln ausgehend von der Schachtel auf der Kellerspitze die auf jeder Schachtellungstiefe adressierbaren Objekte an; in Sprachen ohne Prozedurschachtelung, wie C, kann der statische Vorgänger entfallen,
- ein Speicherbereich zum Sichern der Inhalte von Registern, die in der Prozedur wiederbenutzt werden.

Die Anordnung der beiden Speicherblöcke für Parameter und lokale Variablen und obiger Verwaltungsobjekte innerhalb der Prozedurschachtel wird im Zusammenhang mit der Code-Sequenz für Prozeduraufrufe entworfen, um etwa vorhandene Maschineninstruktionen für den Schachtelaufbau auszunutzen und unnötiges Umspeichern zu vermeiden. Die Schachtelstruktur in Abbildung 7.2-1 ist typisch für eine Aufrufsequenz, in der zunächst die aktuellen Parameter in aufeinanderfolgenden Speicherzellen gekellert werden, dann der Unterprogrammsprung erfolgt und anschließend die Register gesichert werden.

Es kann zweckmäßig sein, den Bezugspunkt der Prozedurschachtel (Relativadresse 0) nicht an den Schachtelanfang zu legen, wenn es eine Maschineninstruktion gibt, (z. B. `lnk` beim M68000), die den Inhalt des Registers mit dem aktuellen Schachtelzeiger kellert und durch den Kellerpegel ersetzt. In diesem Fall erhält der dynamische Vorgänger die Relativadresse 0 und die übrigen Größen in der Schachtel werden relativ dazu (mit positiven und negativen Relativadressen, je nach Adressierungsrichtung) auf dem Keller adressiert.

Blockindexregister

Um den Zugriff auf nicht lokale Prozedurobjekte über die Kette der statischen Vorgänger zu beschleunigen, kann man deren Werte in Registern halten, ein *Blockindexregister* für jede Schachtelungstiefe. Ihr Inhalt muß bei jedem Umgebungswechsel, d. h. Prozedureintritt und -rückkehr aktualisiert werden. Die Blockindexregister können natürlich auch auf Speicherzellen statt auf reale Register abgebildet werden. Allerdings ist diese sogenannte *Display-Technik* im allgemeinen weniger effizienzsteigernd als man erwarten könnte, da nur ein recht geringer Anteil der Zugriffe auf Objekte führt, die nicht lokal und nicht statisch adressiert sind.

Die *Halde* nimmt Objekte auf, die dynamisch generiert werden (z. B. durch Ausführen des Generators `new` in Pascal). Ihre Lebensdauer wird entweder durch eine explizite Operation (`dispose` in Pascal) beendet oder sie endet, sobald die Objekte in der Programmausführung unzugänglich werden. Zur Implementierung der Halde unterscheidet man im wesentlichen folgende Techniken:

- gestreute Speicherung, wobei Generierung und Freigabe durch Operationen des Grundsystems vorgenommen werden,
- Unterbringung in einem linearen Speicherbereich, wobei freigegebene Objekte in *Freispeicherlisten* zur Wiederverwendung eingereiht werden,
- Unterbringung in einem lineare Speicherbereich, wobei der Speicher freigegebener und unzugänglicher Objekte durch Techniken der Speicherbereinigung wiedergewonnen wird.

Verwendung der Register

Zur Vervollständigung der Beschreibung des Maschinenzustands müssen schließlich die Verwendungen der Register festgelegt werden. Dabei müssen sowohl Maschineninstruktionen, die bestimmte Register mit spezieller Bedeutung verwenden, als auch Aufrufkonventionen für Systemfunktionen sowie für von anderen Übersetzern übersetzte Funktionen berücksichtigt werden. Spezielle Verwendungsarten für Register sind z. B.

- Instruktionszeiger, falls er explizit zugänglich ist,
- Pegel des Laufzeitkellers,

- Adresse der aktuellen Schachtel,
- Register für Funktionsergebnisse,
- Pegel der Halde, falls sie in einem zusammenhängenden Speicherbereich implementiert wird.

Die nicht durch solche speziellen Verwendungen gebundenen Register werden bei der Code-Erzeugung für die Aufnahme von Zwischenergebnissen frei vergeben.

7.3 Abbildung der Operationen

Entwurf von Code-Sequenzen

Zu den Operationen der Zwischensprache werden *Code-Sequenzen* entworfen, welche die Übersetzung in die Zielsprache beschreiben. Eine Code-Sequenz für eine Operation gibt man als Folge von Maschinenintruktionen an, in der an bestimmten Stellen der für die Unterbäume zu erzeugende Code (z. B. zur Berechnung von Operanden) eingesetzt wird. Zu jedem Zwischensprachoperator wird mindestens eine Code-Sequenz angegeben. Um die Qualität des erzeugten Codes zu verbessern, entwirft man für einige Operationen mehrere Code-Sequenzen für unterschiedliche Randbedingungen. Es ist Aufgabe der *Code-Auswahl* (Abschnitt 7.4), die Anwendbarkeit der Varianten zu prüfen und die günstigste auszuwählen.

Prozeduraufruf

7.3.1 Prozeduraufrufe

Die Code-Sequenzen für Prozeduraufrufe müssen mit denen für den Prozedureintritt und die Rückkehr aus Prozeduren (Vor- und Nachspann zum Prozedurcode) sowie mit der Schachtelstruktur abgestimmt sein und deshalb mit diesen zusammen entworfen werden. Passend zu der Schachtelstruktur aus Abschnitt 7.2.2 ist z. B. die in Abbildung 7.3-1 angegebene Folge von Operationen. Die linke Spalte enthält die Operationen, für die an der Aufrufstelle Code erzeugt wird, die rechte Spalte gibt den Vor- und Nachspann der Prozedur an. Die Operationen 1, 3, 4, 6, 8, 10 bauen die Schachtel schrittweise auf dem Laufzeitkeller auf; 12, 13, 14, 15, 17 entkellern die Schachtel wieder.

Im allgemeinen ist in (2) die aufzurufende Prozedur zur Übersetzerzeit bekannt. Beim Aufruf von Prozedurparametern oder Prozedurvariablen ist hier jedoch Code zur Auswertung des Zugriffsweges auf das Prozedurobjekt nötig.

Die in den Schritten (9) und (18) aktualisierten Blockindexregister enthalten die Anfangsadressen der statischen Vorgänger der aktuellen Schachtel. Sie werden zur schnellen Adressierung nicht lokaler Variablen verwendet. Sie können im Speicher oder in dafür reservierten Registern untergebracht werden. Verzichtet man auf die Blockindexregi-

Aufrufcode	Prozedurcode
(1) aktuelle Parameter berechnen und kellern	
(2) aufzurufende Prozedur berechnen	
(3) deren statischen Vorgänger kellern	
(4) Rückkehradresse kellern	
(5) Unterprogrammsprung	
	(6) dynamischen Vorgänger kellern
	(7) Schachtelregister := Kellerpegel
	(8) Kellerpegel für lokale Variable erhöhen
	(9) Blockindexregister umsetzen
	(10) Register sichern, kellern
	(11) Code für Prozedurrumpf
	(12) Register restaurieren, entkellern
	(13) Kellerpegel für lokale Variable zurücksetzen
	(14) Schachtelregister entkellern
	(15) Rückkehradresse entkellern
	(16) Rücksprung
(17) Kellerpegel für statischen Vorgänger und Parameter zurücksetzen	
(18) Blockindexregister zurücksetzen	
(19) Funktionsergebnis übernehmen	

Abb. 7.3-1: Aufruf- und Prozedurcode.

ster, so muß bei jedem nichtlokalen Zugriff die statische Vorgängerkette verfolgt werden.

Statischen Vorgänger bestimmen

Der statische Vorgänger der aufzurufenden Prozedur p wird in (3) nach einem der folgenden Verfahren bestimmt (q sei die aufrufende Prozedur):

1) p ist in Schritt (2) direkt angegeben und deshalb in q oder einem seiner statischen Vorgänger deklariert. Die Schachtelungstiefe n_p von p ist zur Übersetzungszeit bekannt, ebenso die Schachtelungstiefe n_q von q. Der statische Vorgänger von p (auf Schachtelungstiefe n_p -1) wird durch Code für das Rückverfolgen der statischen Vorgängerkette um $n_q - n_p + 1$ Schritte, bzw. Zugriff auf das (n_p -1)-te Blockindexregister bestimmt.

2) p ist das Ergebnis der Berechnung in Schritt (2) und Restriktionen der Quellsprache garantieren, daß der statische Vorgänger von p das Hauptprogramm ist.

3) Das Ergebnis der Berechnung in Schritt (2) ist ein Prozedurobjekt, das den statischen Vorgänger enthält. Dieser wird gekellert.

4) Die Berechnung eines statischen Vorgängers entfällt völlig.

Für die Übersetzung von Pascal werden die Varianten (1) und (3), für Modula-2 (1) und (2) benötigt; für C gilt (4). Die Variante (4) kann als Optimierungsmaßnahme auch bei Quellsprachen mit geschachtelten

Prozeduren angewandt werden, falls sichergestellt wird, daß aus der aufgerufenen Prozedur nicht auf nichtlokale Objekte zugegriffen wird.

Falls die Blockindexregister verwendet werden, setzt man für die Operationen (9) und (18) den gleichen Algorithmus ein, um sie zu aktualisieren: Ausgehend von der aktuellen Schachtel durchläuft er die Kette der statischen Vorgänger und trägt jeden in das zugehörige Blockindexregister ein. Der Durchlauf kann vor Erreichen der äußersten Programmumgebung abgebrochen werden, falls alter und neuer Wert eines Blockindexregisters übereinstimmen. Diese Schleife wird zweckmäßig als parameterlose Routine formuliert, die bei (9) und (18) aufgerufen wird.

Unterprogrammsprung

Die Operationen (4, 5) bzw. (15, 16) für den Unterprogrammsprung mit Kellern bzw. Rücksprung mit Entkellern der Rückkehradresse können im allgemeinen durch jeweils eine einzige Maschineninstruktion realisiert werden (z. B. M68000: `jsr` bzw. `rts`). Gibt es solche Instruktionen nicht, so wird in (4) die Programmstelle (17) gekellert, in (15) in ein Register entkellert und in (16) darauf gesprungen. Ebenso werden beim M68000 die Operationen (6, 7, 8) bzw. (13, 14) durch einzelne Instruktionen (`lnk`, `unlk`) erledigt.

Register sichern

In Schritt (10) bzw. (12) werden die Inhalte von solchen Registern, die im Code des Prozedurrumpfes für Zwischenergebnisse benutzt werden, gesichert bzw. restauriert. Hierfür kann - falls vorhanden - eine Spezialinstruktion eingesetzt werden. Da erst nach Erzeugen des Codes für (11) feststeht, welche Register gesichert werden müssen, kann (10) erst nachträglich in den Code eingesetzt werden. Als Alternative zu (10) bzw. (11) könnten auch im Aufrufcode diejenigen Register auf dem Keller gesichert bzw. restauriert werden, deren Inhalte vor dem Aufruf berechnet und nach dem Aufruf benutzt werden. Um die Schachtelstruktur zu erhalten müßten diese Operationen vor (1) und nach (19) eingefügt werden. Bei Maschinen mit einen Registerkeller brauchen höchstens globale Register gesichert zu werden. Das Sichern der Register kann vollständig entfallen, wenn durch Maßnahmen der Registerzuteilung sichergestellt wird, daß keine Zwischenergebnisse über Aufrufe hinweg in Registern gehalten werden.

Funktionsergebnisse werden entweder in dafür festgelegten Registern oder wie Parameter auf dem Keller übergeben. Durch geschickte Anpassung der Code-Sequenzen kann man versuchen, das Umspeichern (19) in möglichst vielen Fällen zu vermeiden.

7.3.2 Ablaufstrukturen

Ablaufstrukturen übersetzt in Sprünge

Höhere Ablaufstrukturen der Zwischensprache (wie `if`, `case`, `while`, `repeat`, `for`) werden auf Code-Sequenzen mit Marken und Sprüngen zurückgeführt. Die Plazierung solcher, im Zuge der Code-Erzeugung generierter Marken und die Adressierung der Sprungziele ist Aufgabe des Assemblierer-Moduls.

In den folgenden Code-Sequenzen nehmen wir an, daß logische Ausdrücke, die als Bedingungen in Ablaufstrukturen vorkommen, so übersetzt werden, daß abhängig vom Ergebnis der Bedingung *B* auf eine Marke *M* gesprungen oder der auf den Code von *B* folgende Code ausgeführt wird. Wir notieren den einzusetzenden Bedingungscode

```
Code (B, true, M) bzw.
Code (B, false, M)
```

für den Fall, daß beim Ergebnis `true` bzw. `false` auf *M* gesprungen wird. (Für diese Art der Übersetzung logischer Ausdrücke eignet sich die Technik der Kurzauswertung besonders gut, siehe Abschnitt 7.3.3).

Im folgenden geben wir Code-Sequenzen für die wichtigsten Ablaufstrukturen an.

Zweiseitige Bedingung: `if B then T else E`

```
        Code (B, false, M1)
        Code (T)
        jump M2
M1:     Code (E)
M2:
```

`jump M2` steht für einen unbedingten Sprung auf `M2`.

einseitige Bedingung: `if B then T`

```
        Code (B, false, M1)
        Code (T)
M1:
```

`while`-*Schleife*: `while B do R`

```
M1:     Code (B, false, M2)
        Code (R)
        jump M1
M2:
```

Eine alternative Code-Sequenz erhält man durch Vertauschen von Rumpf und Bedingung:

```
        jump M2
M1:     Code (R)
M2:     Code (B, true, M1)
```

Beide Sequenzen erzeugen gleich langen Code. In der zweiten wird jedoch bei jedem Schleifendurchgang ein Sprungbefehl weniger ausgeführt. Zur Anwendung dieser Code-Sequenz muß jedoch der Algorithmus zur Code-Erzeugung in der Lage sein, die Reihenfolge des Codes gegenüber der in der Zwischensprache zu verändern.

Konstrukte zum Verlassen von Schleifen (z. B. `exit`) werden durch einen Sprung auf M2 in der ersten Code-Sequenz übersetzt. (In der zweiten muß eine zusätzliche Marke dafür eingeführt werden.) Entsprechende Code-Sequenzen für weitere Schleifenformen (z. B. `repeat`) kann man leicht nach diesem Muster konstruieren.

`Zählschleife: for v:= A to E do R`

Zählschleifen

Zählschleifen treten in Programmiersprachen in verschiedenen Formen mit unterschiedlicher Semantik auf und erfordern besondere Sorgfalt beim Entwurf der Code-Sequenzen. Wir betrachten hier nur Code-Sequenzen für Schleifen mit Schrittweite 1:

```
        Code (A, ta)
        Code (E, te)
        cmp ta, te
        bra >, M2
M1:     mov ta, v
        Code (R)
        cmp ta,te
        bra ≥,M2
        inc ta
        jump M1
M2:
```

In dieser Code-Sequenz verwenden wir Hilfsvariablen `ta` und `te`. Sie können entweder durch zusätzliche, in der Prozedurschachtel untergebrachte Objekte oder durch Register implementiert werden. Im ersten Fall muß dazu die Speicherzuteilung für die Prozedur nachträglich ergänzt werden. Die Zuteilung von Registern zu diesem Zweck ist im allgemeinen höchstens bei den innersten von mehreren geschachtelten Schleifen möglich. Die Notation `Code (A, ta)` bedeutet, daß Code

für die Auswertung von `A` eingesetzt wird, der das Ergebnis in `ta` liefert. `cmp` ist eine Vergleichsinstruktion, der darauf folgende bedingte Sprung (`bra`) wird ausgeführt, falls die angegebene Bedingung (>) im vorangehenden Vergleich erfüllt war. `inc` erhöht den Wert des Operanden um 1. Dies geschieht für `ta` erst nach dem Prüfen der Abbruchbedingung, um zu vermeiden, daß `ta` über den Zahlbereich hinaus vergrößert werden könnte. Dafür nimmt man die Ausführung von zwei Sprungbefehlen in jeder Iteration in Kauf.

In der folgenden Code-Sequenz wird mit der für die `while`-Schleife angewandten Technik der unbedingte Sprung eingespart. Außerdem kann bei manchen Maschinen anstelle der drei letzten Instruktionen eine einzige Spezialinstruktion eingesetzt werden.

```
        Code (A, ta)
        Code (E, te)
        dec ta          ta = A-1
        sub ta, te      te = E-A+1 = Anzahl der Durchläufe
        jump M2
M1:     inc ta
        mov ta, v
        Code (R)
M2:     dec te
        cmp te, 0
        bra ≥, M1
```

`te` gibt hier die Anzahl der Schleifendurchläufe an und wird in jedem Schritt dekrementiert.

Auch hier kann bei der Berechnung von `ta=ta-1` sowie `te=te-ta` ein Überlauf verursacht werden, falls `A=minint` oder `E-A≥maxint`. Für diesen Fall könnte eine recht aufwendige Überlaufprüfung nach dem Code für die beiden Ausdrücke eingefügt werden, ohne die Schleifendurchläufe zu belasten.

Fallunterscheidung:

```
case A  of
k11 , ..., k1n :    S1
km1 , ..., kmp :    Sm
else                Se
end
```

Fallunterscheidung

Für die Übersetzung von Fallunterscheidungen kommen zwei unterschiedliche Techniken in Frage: *Sprungverteiler* oder *Verzweigungskaskade*. Der Sprungverteiler ist im allgemeinen die schnellere, die Ver-

zweigungskaskade in manchen Fällen die weniger speicheraufwendige Implementierung. Welche von beiden günstiger ist, hängt von der Verteilung der Fallmarken k_{ij} ab.

Wir stellen zunächst eine Code-Sequenz für den Sprungverteiler vor:

```
        Code (A, r)
        cmp r, kmax
        bra >, Me
        sub kmin, r
        bra <, Me       Sprung falls r < 0
        jump v(r)       Sprung auf Instruktionszeiger
                        + v[r]
M1:     Code (S1)
        jump N
M2:     Code (S2)
        jump N
         ...
Mm:     Code (Sm)
        jump N
Me:     Code (Se)
N:
```

Für den indizierten Sprung wird ein Vektor der Länge `kmax -kmin + 1` benutzt. Er enthält für jede Fallmarke die Distanz im Code von der `jump`-Instruktion bis zur Marke M_i des zugehörigen Falles. Vektorelemente, zu denen keine Fallmarke existiert, werden mit der Distanz zu M_e belegt. Der Inhalt des Vektors kann selbstverständlich erst nach Erzeugen des Codes für die Fälle bestimmt werden. Der Vektor wird im Segment für initialisierte Daten untergebracht.

Diese Implementierungsvariante ist günstig, falls die Fallmarken zahlreich sind und den Bereich zwischen `kmin` und `kmax` recht dicht ausfüllen. Ist der Bereich sehr groß im Verhältnis zur Zahl der Fallmarken, so ist der Speicheraufwand für den Verteilervektor zu hoch. Gibt es nur sehr wenige Fallmarken, so ist eine Folge von Vergleichen des Wertes von A mit den Fallmarken auch schneller. Sind zahlreiche Fallmarken in einem großen Wertebereich weit verstreut, kann man eine Verzweigungskaskade erzeugen, in der die Vergleiche mit den Fallmarken nach dem Prinzip der binären Suche angeordnet sind.

7.3.3 Kurzauswertung logischer Ausdrücke

Kurzauswertung

Die Auswertung von Ausdrücken mit logischen Operatoren kann dadurch beschleunigt werden, daß man so viele Teilausdrücke von links nach rechts auswertet bis das Gesamtergebnis feststeht und dann den

Code für die Auswertung der verbleibenden Operanden überspringt. Da solche Ausdrücke häufig den Kontrollfluß von Ablaufstrukturen steuern, kann dann als Sprungziel eine geeignete Marke im Kontext vorgegeben und damit ein weiterer Sprung eingespart werden. Diese Technik der *Kurzauswertung* ist dann anwendbar, wenn die Quellsprache es zuläßt, daß nicht alle Operanden eines Ausdrucks ausgewertet werden (z. B. Pascal), oder dies für logische Ausdrücke sogar fordert (z. B. Modula-2). Abbildung 7.3-2 gibt ein Beispiel für die Abfolge der Berechnung abhängig von dem Ergebnis der Operanden. Für die Pfeile a → T, b → E und c → E werden Sprunginstruktionen erzeugt. Die übrigen führen zu der im Code unmittelbar folgenden Instruktion.

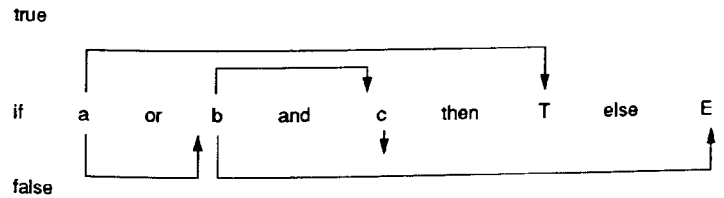


Abb. 7.3-2: Kurzauswertung logischer Ausdrücke.

Wir können dieses Schema systematisch durch Code-Sequenzen beschreiben, indem wir in jedem Teilausdruck vorgeben, ob beim Ergebnis `true` oder `false` gesprungen wird und welches dann die Zielmarke ist. Wir verwenden dafür die oben eingeführte Notation

```
Code (B, true, M) bzw.
Code (B, false, M).
```

Für jeden logischen Operator geben wir eine Code-Sequenz für jeden der beiden durch den Kontext bestimmten Fälle an:

```
Code (A and B, true M):     Code (A, false, N)
                            Code (B, true, M)
                        N:

Code (A and B, false, M):   Code (A, false, M)
                            Code (B, false, M)

Code (A or B, true, M):     Code (A, true, M)
                            Code (B, true, M)
```

```
Code (A or B, true, M):     Code (A, true, N)
                            Code (B, false, M)
                        N:

Code (not A, true, M):      Code (A, false, M)

Code (not A, false, M):     Code (A, true, M)
```

Obige Code-Sequenzen kann man auch als Attributierung des Ausdrucksbaumes auffassen. Jeder Baumknoten hat zwei erworbene Attribute, die für die Unterbäume jeweils mit den Werten `true` oder `false` bzw. der Zielmarke bestimmt werden.

Die Code-Sequenzen zeigen, daß für die Operatoren keine Instruktionen erzeugt werden. Die Sprünge werden an den Teilbäumen für die Operanden generiert. Im Falle von Vergleichen als Operanden ist dies besonders einfach, da sie im allgemeinen ohnehin durch einen Sprung realisiert werden:

```
Code (a verglop b, true, M):    cmp  a, b
                                bra  verglop, M

Code (a verglop b, false, M):   cmp  a, b
                                bra  verglop', M
```

Dabei ist `verglop'` der komplementäre Operator zu `verglop`.

Wird ein Operand direkt als logischer Wert z. B. durch einen Variablenzugriff bestimmt, so muß zusätzlich ein bedingter Sprung abhängig von seinem Wert erzeugt werden (z. B. wie oben durch Vergleich mit einem geeigneten Operanden je nach Codierung logischer Werte).

7.3.4 Ausdrücke

Übersetzung von Ausdrücken

Für den Entwurf der Übersetzung von Ausdrücken müssen folgende Festlegungen getroffen werden:

1) Die Unterbringung von Zwischenergebnissen (in Registern verschiedener Klassen) und Operanden wird durch *Wertdeskriptoren* für die verschiedenen Adressierungsarten der Maschine beschrieben. Die Wertdeskriptoren werden als Operanden der erzeugten Instruktionen verwendet.

2) Die Zwischensprachoperatoren werden auf Maschineninstruktionen abgebildet. In einfachen Fällen (arithmetische Operationen) ist dies eine 1:1-Zuordnung. Für Komplexoperationen (lange Zuweisungen, Mengenoperationen) werden Code-Sequenzen entworfen.

3) Die in der Zwischensprache expliziten Adressierungs- und Inhaltsoperatoren werden entweder auf Instruktionen (zur Adreßrechnung oder

zum Laden in Register) oder auf Transformationen der Wertdeskriptoren abgebildet (z. B. Verwendung eines Speicheroperanden statt explizitem Ladens).

4) Die Auswertungsreihenfolge der Operanden mehrstelliger Operationen wird festgelegt und die für Zwischenergebnisse verwendeten Register werden bestimmt. Diese Aufgabe wird gesondert in Abschnitt 7.3 behandelt.

Für Maschinen mit sehr orthogonalem Instruktionssatz können die Festlegungen zu (1) und (2) unabhängig voneinander getroffen werden. Dies ist z. B. der Fall, wenn für jeden Operanden und das Ergebnis einer Instruktion eine beliebige Adressierungsart gewählt werden kann. Das gleiche gilt für typische RISC-Architekturen, bei denen Operanden von Verknüpfungen grundsätzlich in Registern untergebracht werden und Speicheroperanden nur für Lade- und Speicherinstruktionen zulässig sind.

Vielfach ist der Instruktionssatz weniger orthogonal: Es bestehen Restriktionen für die zulässigen Adressierungsarten einzelner Instruktionen (z. B. höchstens ein Speicheroperand) oder es gibt spezielle zusätzliche, besonders schnelle oder kurze Instruktionen für bestimmte Adressierungsarten (z. B. `addq` für kleine Konstanten beim M68000). Für solche Maschinen wird die Auswahl der am besten geeigneten Instruktionen abhängig von den Wertdeskriptoren der Operanden getroffen.

Wertdeskriptoren

Wertdeskriptoren für Operanden

Grundlage für den Entwurf der Wertdeskriptoren in (1) sind die Adressierungsarten der Zielmaschine (vgl. 7.1.2). Ein Wertdeskriptor ist ein Verbund mit Varianten für jede in den Code-Sequenzen verwendete Adressierungsart von Operanden, Ergebnissen und Zwischenergebnissen. Die Verbundkomponenten jeder Variante entsprechen den für die jeweilige Adressierungsart notwendigen Angaben (Register, Relativadresse, konstanter Wert).

Zur Illustration geben wir einige Wertdeskriptorvarianten (mit ihren Komponenten) an:

`Const (v)`	für konstante, ganzzahlige Werte *v*; weitere Varianten könnten verschiedene Wertebereiche der Konstante unterscheiden,
`DReg (r)`, `AReg (r)`	für in Registern der Klasse *D* oder *A* untergebrachte Werte,
`RegDisp (r, c)`	für Adressen, die durch Addition der Relativadresse *c* zum Inhalt des Registers *r* gebildet werden,

`RegIndex (r, i)` für Adressen, die durch Addition der Inhalte der Register *r* und *i* gebildet werden,

`RegIndexDisp (r, i, c)` für Adressen, die durch Addition der Inhalte der Register *r* und *i* und der Relativadresse *c* gebildet werden.

Um zu unterscheiden, ob ein adreßwertiger Wertedeskriptor (`AReg`, `RegDisp`, `RegIndex`, `RegIndexDisp`) für die Adresse oder deren Inhalt steht, führen wir zusätzliche Wertdeskriptoren für den letzteren Fall mit gleichen Komponenten ein: `CAReg (r)`, `CRegdisp (r, c)`, `CRegIndex (r, i)`, `CRegIndexDisp (r, i, c)`. Weitere Wertdeskriptoren werden z. B. für absolute Daten- und Programmadressen benötigt.

Operatoren in Ausdrucksbäumen

Ausdrucksoperatoren

In einem Ausdrucksbaum beschreiben die Wertdeskriptoren die Unterbringung der Operanden und Zwischenergebnisse. Für einen binären Operator im Baum sind dies z. B. `wl`, `wr` und `wres`. Die Übersetzung eines Operators durch eine Instruktion (oder Instruktionsfolge) unterliegt Restriktionen bezüglich der Operanden-Wertdeskriptoren: So muß z. B. bei der `add`-Instruktion `wl` ein *D*- oder *A*-Register sein, `wr` ist beliebig und `wres=wl`. Wir beschreiben die Übersetzung durch eine Code-Sequenz, deren Kopf die beteiligten Wertdeskriptoren angibt:

```
Code (+; wl: DReg (d), wr) wres : wl :    add wr, wl
```

Den Kopf der Code-Sequenz können wir als ihre Signatur auffassen (Operanden- und Ergebnistypen). Ist die Code-Sequenz mit verschiedenen Kombinationen von Wertdeskriptoren anwendbar, so geben wir mehrere alternative Signaturen an:

```
Code (+; wl : DReg (d), wr) wres : wl
Code (+; wl : AReg (a), wr) wres : wl
Code (+; wr, wl : DReg (d)) wres : wl
Code (+; wr, wl : AReg (a)) wres : wl:    add wr, wl
```

Die zwei letzten Signaturen nutzen die Kommutativität des Operators aus. Ist der rechte Operand eine Konstante (aus einem begrenzten Wertebereich), so kann z. B. eine Spezialinstruktion eingesetzt werden:

```
Code (+; wl : DReg, wr : Const (c)) wres : wl
Code (+; wl : AReg, wr : Const (c)) wres : wl :  addq wr, wl
```


Sind beide Operanden konstant, drücken wir Konstantenfaltung durch eine Code-Sequenz aus, die keine Instruktion erzeugt:

```
Code (+; wl : Const (a); wr : Const (b)) wres : Const (c) :
                                          c= a+b
```

In den obigen Code-Sequenzen ist der für die Teilbäume der Operanden zu erzeugende Code nicht explizit angegeben. Es wird angenommen, daß er (in geeigneter Anordnung, siehe auch Abschnitt 7.3) dem Operatorcode vorangeht. Zur Vereinfachung des Entwurfs der Code-Sequenzen für alle Operatoren der Zwischensprache ist es zweckmäßig, gleich behandelte Operatoren zusammenzufassen und die Zahl der Signaturen durch Zusammenfassung von Wertdeskriptoren zu reduzieren.

Inhaltsoperation

Verzögerte Inhaltsoperation

Die Inhaltsoperation der Zwischensprache `(il_cont)` für einfache Werte könnte in einen Ladebefehl übersetzt werden, z. B.

```
Code (il_cont; wr : RegDisp (r, d)) wres : DReg (dr):
                                       mov r, dr
```

Damit würde jedoch die Verwendung des Operanden `r` als Speicheroperand in einer übergeordneten Operation ausgeschlossen. Der Ladebefehl ist z. B. bei rechten Operanden binärer Operationen überflüssig. Wir *verzögern* deshalb die Erzeugung des Ladebefehls bis im oberen Kontext entschieden wird, ob er erforderlich ist. Die Code-Sequenzen für die Inhaltsoperation transformieren deshalb nur den Wertdeskriptor und erzeugen keine Instruktion:

```
Code (il_cont; wr : RegDisp (r, d)) wres : CRegDisp (r, d):
                                       leer
```

Tritt eine solche Operation als rechter Operand z. B. einer binären arithmetischen Operation auf, so ist der Wertdeskriptor des Ergebnisses unmittelbar passend und es wird ein Speicheroperand verwendet. Tritt er aber als linker Operand auf, so muß er doch in ein Register geladen werden, da nur die Wertdeskriptoren `DReg` für linke Operanden von binären arithmetischen Operationen passen.

Aus diesem Grund müssen wir auch für die in solchen Fällen notwendigen *Anpassungen von Wertdeskriptoren* Code-Sequenzen einführen. Sie sind keinen Operatoren der Zwischensprache zugeordnet, sondern werden angewandt, falls die Wertdeskriptoren der Operanden nicht unmittelbar passen:

```
Code (r : CRegDisp (r, d)) wres : DReg (dr) :        mov r, dr
```

Entsprechende Code-Sequenzen gibt man für die übrigen Wertdeskriptoren an, die verzögerte Inhaltsoperationen beschreiben.

Adreßrechnung

Adreßrechnung

Operationen zur Addition von Zahlwerten zu Speicheradressen sind in der Zwischensprachrepräsentation explizit eingesetzt (`il_aadd`). Ihre effiziente Übersetzung spielt für die Qualität des erzeugten Codes eine entscheidende Rolle. Vielfach braucht für eine Adreßaddition keine Instruktion erzeugt zu werden, falls sie bei der Berechnung der effektiven Adresse eines Operanden geeigneter Adressierungsart implizit erfolgt. So kann man z. B. die Addition einer konstanten Relativadresse *c* zu einer Adresse im A-Register durch Einsetzen von *c* in einem Speicheroperanden der Form `RegDisp (r, c)` erzielen. Wir geben deshalb für die Adreßaddition eine Reihe von Code-Sequenzen an, die möglichst weitgehend diese impliziten Adreßrechnungen durch Transformation der Wertdeskriptoren beschreiben; hier nur einige Beispiele dazu:

```
Code  (il_aadd; wl: AReg (r); wr: Const (c))
      wres: RegDisp (r, c): leer

Code  (il_aadd; wl: RegDisp (r, k); wr: Const (c))
      wres: RegDisp (r, d): d= k+c
```

(Die Anwendbarkeit dieser Code-Sequenz wird im allgemeinen durch die Größe von d begrenzt.)

```
Code  (il_aadd; wl: RegDisp (r, k); wr: AReg (s))
      wres: RegIndexDisp (r, s, k): leer

Code  (il_aadd; wl: RegIndex (r, i, k); wr: CAReg (s)
      wres: wl:   add wr, s
```

Nur im letzten der angegebenen Fälle ist eine explizite Instruktion für die Adreßaddition nötig.

7.4 Code-Auswahl

In Abschnitt 7.3 haben wir beschrieben, wie für jeden Operator der Zwischensprache Code-Sequenzen entworfen werden. Dieser Entwurf führt unmittelbar zu der hier dargestellten systematischen Implementierung der Code-Auswahl. Sie ist von besonderer Bedeutung für Ausdrucksoperatoren, denen häufig mehrere, alternativ anwendbare Code-

Sequenzen zugeordnet sind (z. B. für Adressierung). Am Ende des Abschnitts weisen wir darauf hin, wie Übersetzermodule für diese Aufgabe automatisch generiert werden können.

7.4.1 Ausdrucksbäume

Code-Auswahl für Ausdrücke

Für jeden Operator der Zwischensprache entwerfen wir eine Funktion nach einheitlichem Schema: Sie hat für jeden Operanden einen Parameter vom Wertdeskriptortyp. Das Funktionsergebnis ist ebenfalls vom Wertdeskriptortyp. Ihr Rumpf besteht aus einem *Auswahlteil*, der die anzuwendende Code-Sequenz selektiert, und einem *Generierungsteil*, der die Instruktion ausgibt und den Ergebniswertdeskriptor berechnet.

Im Auswahlteil wird anhand der Wertdeskriptoren für die Operanden (Parameter der Funktion) entschieden, nach welcher Code-Sequenz im zweiten Teil generiert wird. Wir entwerfen diese Auswahl allgemein zunächst als n-fach geschachtelte Fallunterscheidung bei einem n-stelligen Operator. Jede Fallunterscheidung erstreckt sich über alle Typen von Wertdeskriptoren. In den Anweisungen der Fälle wird eine Codierung der jeweils anzuwendenden Code-Sequenz zu dem betrachteten Operator bestimmt.

Für einige Wertdeskriptorkombinationen ist eine Code-Sequenz nicht unmittelbar anwendbar, sondern erst nach Anpassung der Wertdeskriptoren (z. B. Laden eines Speicheroperanden in ein Register). Wir berücksichtigen dies durch zusätzliche Aufrufe der nach dem gleichen Schema konstruierten Anpassungsfunktionen. Häufig werden mehrere der möglichen Wertdeskriptorkombinationen auf die gleiche Code-Sequenz abgebildet, oder andere können aus strukturellen Gründen ausgeschlossen werden. Wir vereinfachen dann die Fallunterscheidung entsprechend durch Zusammenfassen von Fällen und Einführen von Fehlerausgängen.

Der Generierungsteil der Funktion besteht aus einer Fallunterscheidung für die einzelnen Code-Sequenzen. Jeder Fall enthält Anweisungen zur Ausgabe der Instruktion und zur Berechnung des Wertdeskriptors für das Ergebnis. Abbildung 7.4-1 zeigt das Schema solcher Auswahlfunktionen.

Die hier beschriebenen Funktionen werden je nach Implementierung der Zwischensprachschnittstelle unterschiedlich aufgerufen:

- Zur Erzeugung der Zwischensprache ruft der Analyseteil für jeden Operator die Code-Auswahlfunktion unmittelbar auf. In diesem Fall wird der Zwischensprachbaum nicht als Datenstruktur aufgebaut. Der Code wird direkt erzeugt. Die Code-Reihenfolge wird durch die Aufruffolge im Analyseteil bestimmt. Die Wertdeskriptoren der Aufrufergebnisse werden bis zu ihrer Verwendung als Parameter in geeigneten Datenstrukturen der Zwischenspracherzeugung gespeichert.)

```
function
  il_aadd_code (opl, opr : Wertdeskr) : Wertdeskr;

  var res : Wertdeskriptor; cs : integer;
  begin
      (* Auswahlteil *)
      case opl.tp of
           AReg: case opr.tp of
                      DReg: cs := 0;
                      ...
                 end;
           ...
      end;

      (* Generierungsteil *)
      case cs of
      (* 1. Code-Sequenz fuer il_aadd *)
         ...
      end;
  end;
```

Abb. 7.4-1: Schema von Funktionen zur Code-Auswahl.

• An der Schnittstelle zwischen Analyse- und Syntheseteil wird ein Zwischensprachbaum (ggf. stückweise) aufgebaut. Dieser wird zur Code-Auswahl in der Reihenfolge des zu erzeugenden Codes durchlaufen. Dabei ruft man in der Aufwärtsphase die Auswahlfunktionen auf. Die Wertdeskriptoren speichert man entweder in einem Keller oder als Attribute der Baumknoten. Dieses Verfahren kann auch auf zwei Baumdurchläufe ausgedehnt werden, um die Registerzuteilung zu verbessern (siehe auch Abschnitt 7.5): In der ersten Aufwärtsphase wird nur der Auswahlteil der Funktionen ausgeführt und der Ergebniswertdeskriptor (ohne konkrete Register) bestimmt. Abhängig von deren Ergebnis bestimmt man Registerbedarf und Auswertungsreihenfolge der Operanden. Im zweiten Baumdurchlauf teilt man abwärts die Register zu und generiert aufwärts den Code gemäß der ausgewählten Code-Sequenz.

7.4.2 Ablaufstrukturen

Code-Erzeugung für Ablaufstrukturen

Bei der Implementierung der Code-Erzeugung für Ablaufstrukturen spielt die Auswahl von Code-Sequenzen nur eine untergeordnete Rolle, da meist für jeden Zwischensprachoperator nur ein oder zwei Code-Sequenzen entworfen werden. Hier ist die Realisierung der Zwischensprachschnittstelle von entscheidender Bedeutung.

Wird an der Schnittstelle zum Syntheseteil ein Zwischensprachbaum aufgebaut, so konstruiert man nach dem Muster der entworfenen Code-Sequenzen für Ablaufstrukturen einen Satz von rekursiven Prozeduren. Der Rumpf einer solchen Prozedur wird systematisch gemäß der

Code-Sequenz programmiert. Er enthält Aufrufe zur Erzeugung von Code für die Unterbäume in der angegebenen Reihenfolge, und dazwischen Anweisungen zur Ausgabe von Instruktionen und zur Plazierung von Marken. Abbildung 7.4-2 gibt als Beispiel eine Prozedur für die zweiseitige Bedingungen an.

```
procedure il_if2_code (k: Knoten);
begin
   m1 := generiere_Marke;
   m2 := generiere_Marke;
   il_cond_code (k^.sohn[1], false, m1);
   il_stmt_code (k^.sohn[2]);
   emit1 (jump, m2);
   plaziere_Marke (m1);
   il_stmt_code (k^.sohn[3]);
   plaziere_Marke (m2)
end;
```

Abb. 7.4-2: Prozedur zur Code-Sequenz für zweiseitige Bedingungen.

Die Prozedur `il_stmt_code` verzweigt abhängig vom Operator des Knotens auf die entsprechende Prozedur. Auch für die Kurzauswertung logischer Ausdrücke werden Prozeduren nach diesem Muster, mit zusätzlichen Parametern wie in Abschnitt 7.3.3 aufgerufen.

Sollen auch Ablaufstrukturen ohne Aufbau eines Zwischensprachbaumes direkt durch Aufrufe von Code-Erzeugungsprozeduren übersetzt werden, so muß die Zwischensprache dafür systematisch erweitert werden.

Wir zerlegen dazu die Code-Sequenz für jeden Ablaufstrukturoperator in Abschnitte, die jeweils nur Sequenzen von Anweisungen zur Ausgabe von Instruktionen bzw. zum Generieren oder Plazieren von Marken enthalten. Für jeden dieser Abschnitte führen wir einen neuen Zwischensprachoperator ein und entwerfen eine Code-Prozedur. Für das Beispiel der zweiseitigen Bedingung ergibt dies z. B. die Operatoren `il_if2_1`, `il_if2_2` und `il_if2_3` mit den Prozeduren

```
procedure il_if2_1_code
begin
   push (generiere_Marke);
   push (generiere_Marke);
end;
```

```
procedure il_if2_2_code;
begin
    emit1 (jmp, top)
    plaziere_Marke (top-1)
end;
procedure il_if2_3_code;
begin
    plaziere_Marke (top);
    pop; pop
end;
```

Anstelle einer einzigen Funktion für einen Zwischensprachoperator `il_if2` ruft die Zwischenspracherzeugung die Funktionen für die neuen Operatoren `il_if2_1` vor dem Gesamtkonstrukt, `il_if2_2` vor dem `else`-Teil und `il_if2_3` nach dem Gesamtkonstrukt auf. Die generierten Marken werden in einem Keller verwaltet. Aus der Sicht der Zwischenspracherzeugung entspricht dieses Vorgehen einem Aufwärtserzeugen des Zwischensprachbaumes, wobei der Operator il_if2 aufgespalten wurde in einen Präfix-, einen Infix- und einen Postfixoperator.

7.4.3 Generatoren zur Code-Auswahl

Die Code-Auswahl für Ausdrücke ist wegen der kombinatorischen Komplexität der zu unterscheidenden Fälle die fehlerträchtigste Implementierungsaufgabe der Code-Erzeugung, andererseits ist sie auch am einfachsten systematisierbar. Aus diesem Grunde existieren dafür die bisher am weitesten entwickelten generierenden Werkzeuge im Bereich der Synthese.

Ersetzen von Baummustern

Die meisten Generatoren basieren auf dem Prinzip der *Ersetzung von Baummustern*. Wir wollen es zunächst an der oben eingeführten Systematik darstellen. Abbildung 7.4-3 zeigt den Ausschnitt eines Zwischensprachbaumes mit einer Ganzzahladdition an der Wurzel. Die beiden Operanden werden durch Inhaltsoperation (*cont*) angewandt auf Adreßausdrücke bestimmt ($+_a$ steht für die Adreßaddition). Die Kästen in der Abbildung repräsentieren jeweils die Anwendung einer Code-Sequenz aus unserem Entwurf in Abschnitt 7.3 mit den Typen der Wertdeskriptoren aus ihrer Signatur. Wir können sie als Anwendung von Baummustern für die Code-Sequenzen auffassen. An den Wertdeskriptoren überlappen sich die Muster: Der Ergebniswertdeskriptor des Unterbaumes ist entweder gleich dem entsprechenden Operandenwertdeskriptor oder er wird durch eine Code-Sequenz daran angepaßt (wie im linken oberen Teilbaum). Das Problem der Code-Auswahl wird hier gelöst, indem man eine vollständige, passende Überdeckung des Baumes durch Muster zu den Code-Sequenzen findet. In unserer Implementierung haben wir diese Überdeckung bei einem Aufwärtsdurchgang durch den Baum anhand der Wertdeskriptoren bestimmt.

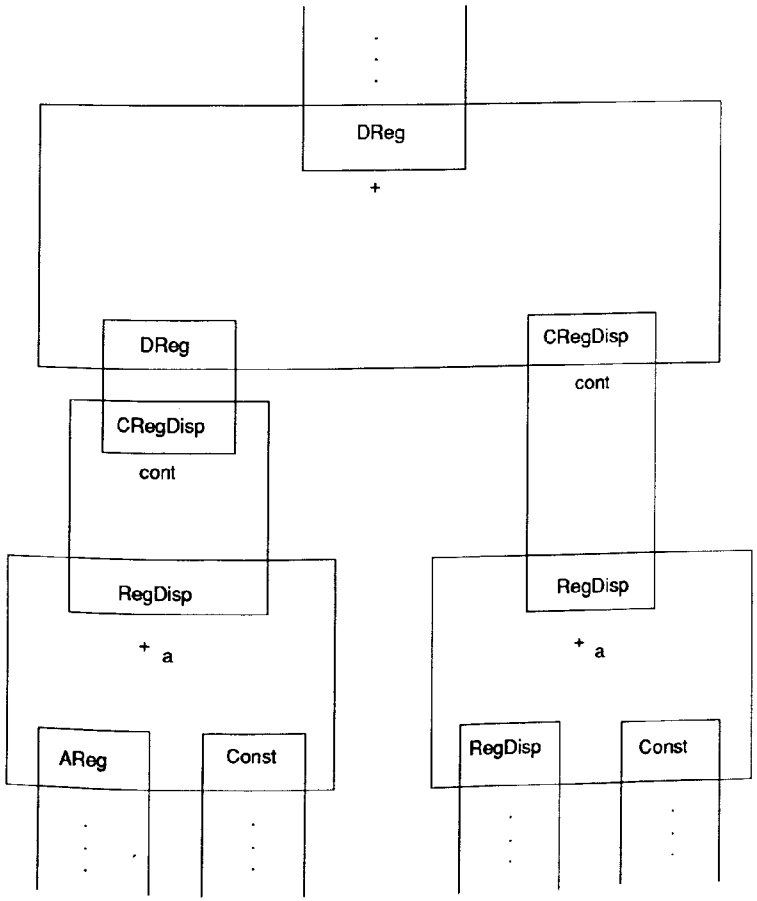


Abb. 7.4-3: Überdeckung mit Baummustern.

Bei der von Graham und Glanville entwickelten Methode (GLAN78) und den darauf basierenden Generatoren z. B. (GRAH82, LAND82) wird ein Syntaxanalyseverfahren zur Bestimmung der Überdeckung angewandt. Der Zwischensprachbaum wird mit seinen Operatoren in Präfixform dargestellt. Die Baummuster zu den Code-Sequenzen werden als Produktionen einer kontextfreien Grammatik angege-

ben. Die Wertdeskriptoren der Signaturen übernehmen hier die Rolle der Nichtterminale. In unserem Beispiel aus Abbildung 7.4-3 werden Code-Seqenzen zu folgenden Produktionen verwendet:

```
1) DReg ::= + DReg CRegDisp
2) RegDisp ::= +a AReg Const
3) RegDisp ::= +a RegDisp Const
4) CRegDisp ::= cont RegDisp
5) DReg ::= CRegDisp
```

Zu einer solchen (vervollständigten) Grammatik berechnet ein quellbezogener Zerteiler (nach der LALR(1)-Methode) einen Ableitungsbaum. Er repräsentiert die gesuchte Baumüberdeckung. Man baut ihn jedoch nicht auf, sondern es werden jeweils beim Anwenden einer Produktion die zugeordnete Code-Sequenz erzeugt und Operationen zur Berechnung von Wertdeskriptorkomponenten ausgeführt. Die hier beschriebene Übertragung unseres Signaturentwurfs auf diese Methode nutzt deren Mächtigkeit nicht aus. Es können auch Baummuster angegeben werden, die sich über mehrere Ebenen im Baum erstrecken und mehrere Operatoren enthalten. So könnte man z. B. die Produktionen (3) und (4) zusammenfassen:

```
CRegDisp ::= cont + RegDisp Const
```

Damit können auch Einzelinstruktionen als Code-Sequenz beschrieben werden, in die größere Baumausschnitte übersetzt werden.

Im Gegensatz zur Syntaxanalyse der Quellsprache ist die hier entwickelte Grammatik im allgemeinen nicht eindeutig. Im Gegenteil: Mehrdeutigkeiten drücken Varianten der Übersetzung aus, die zur Code-Verbesserung ausgenutzt werden sollen. Der Zerteiler wird deshalb so generiert, daß Konflikte aufgelöst werden. Dies geschieht so, daß längere Muster, die größere Baumteile überdecken, bevorzugt werden. Außerdem kann die Auswahl durch den Code-Sequenzen zugeordnete Kosten und Anwendbarkeitsbedingungen (z. B. Wertebereich von Konstanten im Wertdeskriptor) beeinflußt werden.

Da der Code im Aufwärtsdurchgang sofort erzeugt wird, ist die Auswertungsreihenfolge mit dem Zwischensprachbaum fest vorgegeben. Falls die Registerzuteilung in die Aktionen der Code-Sequenzen integriert wird, muß sie ebenfalls in dem Aufwärtsdurchgang erfolgen.

Aho und Ganapathi haben eine Methode entwickelt, die ein effizientes Mustererkennungsverfahren mit der Methode der dynamischen Programmierung verbindet (AHOG85). Dabei werden im Aufwärtsdurchgang verschiedene Teilüberdeckungen mit ihren Kosten gespeichert und erst in einem zweiten Durchgang die günstigste daraus ausgewählt. Nach dieser Methode arbeitet z. B. der Generator *twig*. In GANA82 wird eine Übersicht über systematische Verfahren zur Code-Auswahl gegeben.

7.5 Registerzuteilung

Eine große Klasse von Maschinenarchitekturen verwendet Register als Speicherelemente innerhalb des Prozessors. Instruktionen, die ihre Operanden aus Registern entnehmen oder ihr Ergebnis in Register schreiben sind im allgemeinen schneller und häufig auch kürzer als solche, die über den Systembus auf den Hauptspeicher zugreifen. Die Anzahl der verfügbaren Register ist im allgemeinen beschränkt. Ein bedeutender Faktor für die Qualität des erzeugten Codes ist deshalb die gute Ausnutzung der verfügbaren Register, um die Zahl der Speicherzugriffe gering zu halten. Dies ist von besonderer Bedeutung bei RISC-Architekturen, die Speicherzugriffe ausschließlich in Lade- und Speicherinstruktionen erlauben.

Verwendung der Register

Beim Entwurf der Code-Erzeugung legt man die Verwendung der Register fest:

1) Zwischenergebnisse bei der Berechnung von Ausdrücken,
2) Werte von Ausdrücken, die mehrfach verwendet werden,
3) Variablen,
4) Prozedurparameter,
5) Funktionsergebnis,
6) Adressen von Speicherbereichen wie Kellerpegel, Haldenpegel, aktuelle Schachtel usw.

Für die Verwendungen (6) und eventuell auch (5) werden bestimmte Register vorgesehen, die dann nicht mehr anderweitig eingesetzt werden. Die verbleibenden Register teilt man nach unterschiedlichen Strategien den übrigen Verwendungsarten zu. Da dies in verschiedenen Modulen des Übersetzers geschehen kann, werden mit der Strukturierung des Übersetzers auch Entscheidungen über die Registerzuteilung gefällt. Die wichtigsten Zusammenhänge wollen wir hier aufzeigen. Da alle diese Maßnahmen auf eine Verbesserung des erzeugten Codes abzielen und damit Gegenstand der Optimierung sind, müssen wir hier einige Techniken im Vorgriff auf Kapitel 8 erwähnen.

Variablen, Prozedurparametern und Funktionsergebnissen kann für ihre gesamte Lebensdauer ein Register statt eines Speicherplatzes in der Prozedurschachtel zugeordnet werden. Diese Entscheidung fällt man bei der Speicherzuteilung und vermerkt sie in deren Datenstruktur. Die Auswahl solcher Objekte kann durch Informationen über ihre Verwendungshäufigkeit aus der Datenflußanalyse verbessert werden. Da die so zugeordneten Register nicht mehr für weitere Verwendungen im Rumpf der Prozedur zur Verfügung stehen, ist diese Maßnahme nur bei Prozessoren mit sehr großem Registersatz oder Registerkeller angezeigt. Alternativ kann man diese Objekte auch zusammen mit denen der Verwendungsarten (1) und (2) während der Code-Auswahl Registern zuteilen.

Register zur Unterbringung von Werten mit relativ kurzer Lebensdauer (Verwendungsarten (1) und (2)) werden meist bei der Erzeugung der Instruktionen verzahnt mit der Code-Auswahl zugeteilt. Die dazu verwendeten Verfahren unterscheiden sich im wesentlichen in der Art der Information, die zur Entscheidung über eine möglichst günstige Zuteilung herangezogen wird. Reichen die hierfür verfügbaren Register nicht aus, so muß Code zum *Zwischenspeichern* und Wiederladen belegter Register beigefügt werden (*spill code*).

Auswertungsreihenfolge

Im einfachsten Fall werden die Register für Zwischenergebnisse und Ausdrücke beim Erzeugen jeder einzelnen Instruktion in der Auswertungsreihenfolge ohne weitere vorausschauende Information ausgewählt (on-the-fly). Werden Ausdrucksbäume zweimal durchlaufen, so kann man den Code für Teilbäume so umordnen, daß die verfügbaren Register günstig genutzt werden. Weitergehende Information steht zur Verfügung, wenn man Abschnitte des Codes, die keine Marken und Sprünge enthalten (Grundblöcke) zweimal durchläuft, um zukünftige Verwendungen von Werten bei der Zuteilung zu berücksichtigen. Als Ergebnis der Datenflußanalyse kann Information über einen noch größeren Kontext (Prozedurrumpf, Programm) bei der Registerzuteilung genutzt werden.

Mehrfach verwendete Ausdrücke

Sollen mehrfach verwendete Ausdrücke bei der Registerzuteilung berücksichtigt werden, so müssen sie in einer vorangehenden Analyse erkannt und identifiziert werden. Dies kann wiederum in unterschiedlich großem Kontext geschehen: innerhalb eines Ausdrucks, Grundblocks, Prozedurrumpfs oder des Programms. Für alle diese Maßnahmen (mit oben genannter Ausnahme) geht man im allgemeinen davon aus, daß zum Zeitpunkt der Registerzuteilung die Auswertungsreihenfolge festgelegt ist. Um dabei trotzdem Aspekte der Registerzuteilung zu berücksichtigen, kann man die Auswertungsreihenfolge vor der Code-Auswahl in Abhängigkeit von Abschätzungen des erwarteten Registerbedarfs bestimmen.

Symbolische Register

Man kann die Registerzuteilung auch in eine Phase nach der Code-Auswahl verschieben, indem man zunächst *symbolische Register* aus einem unbegrenzten Vorrat zuteilt. In einem nachfolgenden Durchgang durch die Instruktionsfolge ersetzt man die symbolischen Register konsistent durch reale. Reichen diese nicht aus, so müssen zusätzliche Instruktionen zum Zwischenspeichern eingefügt werden. Als Konsequenz solcher Einfügungen können die Marken und Sprünge in dem ggf. noch zu modifizierenden Code noch nicht endgültig adressiert sein.

Schließlich sei noch auf eine häufig wirksame Einzelmaßnahme hingewiesen: In den meisten Fällen ist es gleichgültig in welchem Register einer Klasse ein Wert berechnet wird. In Fällen wie dem Funktionsergebnis gilt dies jedoch nicht. Man vermeidet ein Umladen zwischen Registern, indem die Berechnung von vornherein in dem Zielregister

ausgeführt wird (*targeting*). Während bei 3-Adreßmaschinen diese Verbesserung einfach durch Einsetzen des Zielregisters in der letzten Instruktion eines Ausdrucks erreicht werden kann, muß bei 2-Adreßmaschinen das Zielregister bei der Zuteilung für den gesamten Ausdruck berücksichtigt werden.

In den folgenden Abschnitten stellen wir einige der wichtigsten Verfahren zur Registerzuteilung vor.

7.5.1 Registerzuteilung in einem Durchgang

Ein Aufwärtsdurchgang

Dieses sehr einfache Verfahren teilt die Register in einem einzigen Durchgang *(on-the-fly)* durch die Instruktionsfolge (z. B. bei deren Erzeugung in der Code-Auswahl) zu. Die Auswertungsreihenfolge liegt dabei fest. Die Lebensdauer von Werten in Registern ist auf die Verwendung in einzelnen Ausdrücken oder maximal in Grundblöcken begrenzt.

Wir nehmen an, daß die Operationen der Registerzuteilung mit der Verwaltung der freien und verfügbaren Register in einem Modul zusammengefaßt sind. Dies sind im wesentlichen die Operationen

`get_reg`	liefert ein freies Register und kennzeichnet es als belegt, und
`free_reg (r)`	gibt ein zugeteiltes Register wieder frei.

Der Zuteilungszustand wird durch die Menge der jeweils freien Register beschrieben. Die Operationen werden bei der Erzeugung der einzelnen Instruktionen aufgerufen. Für die Anwendung im Zusammenhang mit der im Abschnitt 7.4 beschriebenen Code-Auswahl ergänzt man die Code-Sequenzen durch solche Aufrufe. Die zugeteilten Register werden in die erzeugten Instruktionen und die Wertdeskriptoren eingesetzt. Die Code-Sequenz zum Laden eines Wertes v hat dann z. B. die Form

```
res := get_reg
emit2 (mov, v, res)
```

Der Ergebniswertdeskriptor `res` beschreibt das zugeteilte Register zur späteren Verwendung. Für eine Addition in 2-Adreßform geben die Operandenwertdeskriptoren zugeteilte Register an. Das des rechten Operanden wird freigegeben, das des linken als Ergebnisdeskriptor übernommen:

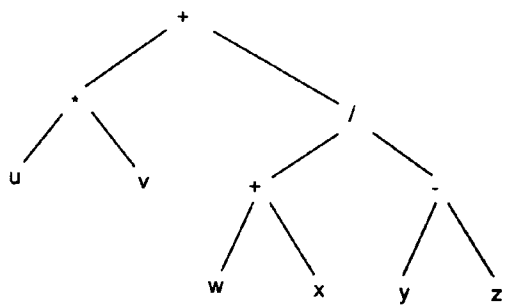
```
res := wl;
free_reg (wr);
emit2 (add, wr, res)
```

Im Falle von 3-Adreßinstruktionen gibt man beide Operandenregister frei und teilt das Ergebnis neu zu:

```
free_reg (wl);
free_reg (wr);
res := get_reg;
emit3 (add, wl, wr, res)
```

Zur Vereinfachung haben wir hier nur Wertdeskriptoren für Register betrachtet. Bei komplexer strukturierten Wertdeskriptoren nimmt man in deren Komponenten die zugeteilten Register auf.

Abbildung 7.5-1 zeigt z. B. einen Ausdrucksbaum, für den, wie in Abschnitt 7.4 beschrieben, der ausgegebene 2-Adreßcode aufwärts von links nach rechts mit den zugeteilten Registern erzeugt wird. Zur Vereinfachung nehmen wir an, daß alle Operanden von Verknüpfungen in Registern sein müssen.




```
mov   u,   d1
mov   v,   d2
mul   d2,  d1
mov   w,   d2
mov   x,   d3
add   d3,  d2
mov   y,   d3
mov   z,   d4
sub   d4,  d3
div   d3,  d2
add   d2,  d1
```

Abb. 7.5-1: Registerzuteilung in einem Durchgang für einen Ausdrucksbaum.

Das Verfahren ist ebenso auf Grundblöcke anwendbar, in denen in Registern anfallende Zwischenergebnisse auch mehrfach verwendet werden. Bei der Erzeugung der Instruktionen werden wie oben Register durch `get_reg` beschafft, im Wertdeskriptor bis zu ihrer letzten Verwendung gespeichert und dann durch `free_reg` freigegeben. Abbildung 7.5-2 gibt die Operationsfolge eines Grundblocks und den dazu erzeugten Code an. Die Buchstaben `a - i` stehen für Wertdeskriptoren der Zwischenergebnisse, `w, x, y, z` für Werte, die nicht im Register vorliegen. Wegen der hier vorgegebenen Auswertungsreihenfolge werden vier Register benötigt. Weiter unten geben wir ein Verfahren an, das den Registerbedarf minimiert.


```
a := x          mov  x,  d1
b := y          mov  y,  d2
c := b + a      add  d1, d2
d := z          mov  z,  d3
e := d*c        mul  d2, d3
f := w          mov  w,  d4
g := e/f        div  d4, d3
h := g + a      add  d1, d3
i := h*c        mul  d2, d3
```

Abb. 7.5-2: Registerzuteilung in einem Durchgang für Grundblock.

Zwischenspeichern

Der Registerzuteilungsmodul versagt, falls mehr als die verfügbaren Register zugleich belegt werden sollen (in beiden Beispielen, falls nur drei Register zur Verfügung stehen). Da der Modul keine Kontrolle mehr über die vergebenen Register (bis zu ihrer Freigabe) hat, kann er auch nicht die Erzeugung von Code zum Zwischenspeichern veranlassen.

Wir können diesen Defekt beheben, indem wir symbolische Register und eine zusätzliche Moduloperation einführen: Der Modul vergibt mit `get_reg` symbolische statt realer Register (z. B. fortlaufende ganze Zahlen); `free_reg` gibt diese wieder frei. Der Zuteilungszustand wird durch eine Liste der belegten symbolischen Register dargestellt. Jedes Listenelement gibt ein reales Register und ggf. eine Speicherstelle an, in die der Registerinhalt zwischengespeichert wurde. Unmittelbar vor Erzeugen einer Instruktion wird durch Aufruf einer zusätzlichen Operation `use_reg` (s_1 , ..., s_n) für alle in der Instruktion verwendeten symbolischen Register s_i ein reales Register `r` zugeteilt. Dazu wird, falls nötig, der Inhalt von Registern zwischengespeichert bzw. zurückgeladen. Der so erzeugte Code ist nicht minimal. Diese Lösung ist höchstens dann vertretbar, wenn hinreichend viele Register zur Zuteilung zur Verfügung stehen, so daß selten Code zum Zwischenspeichern erzeugt wird.

7.5.2 Registerzuteilung für Grundblöcke in zwei Durchgängen

Werden Register auf der Ebene von Grundblöcken oder Ausdrücken mit gleichen Teilausdrücken zugeteilt, so können einmal berechnete Zwischenergebnisse in Registern gehalten und mehrfach verwendet werden. Sind alle Verwendungen der Werte im Grundblock bei der Registerzuteilung von vornherein bekannt, so kann man das Zwischenspeichern mit einem Verfahren minimieren, das in seinen Grundzügen von Belady als Seitentauschverfahren vorgestellt wurde (BELA66). Es basiert auf zwei Durchgängen durch den Code: Im ersten werden alle Verwendungen der Werte festgestellt. Im zweiten werden

die Register zugeteilt. Falls dabei Registerinhalte zwischengespeichert werden müssen, wählt man solche Werte, die erst möglichst spät wiederverwendet werden.

Lebensdauergraph

Dazu wird im ersten Durchgang ein Graph der Lebensdauer der Werte bezogen auf die Abfolge der Instruktionen aufgestellt. Abbildung 7.5-3 zeigt den Lebensdauergraphen für das Beispiel aus Abbildung 7.5-2. Der Zuweisung von Operationsergebnissen ordnet man einen späteren Zeitpunkt zu als der Operation selbst, um auszudrücken, daß die Lebensdauer eines Operanden enden kann, bevor die des Ergebniswertes beginnt.

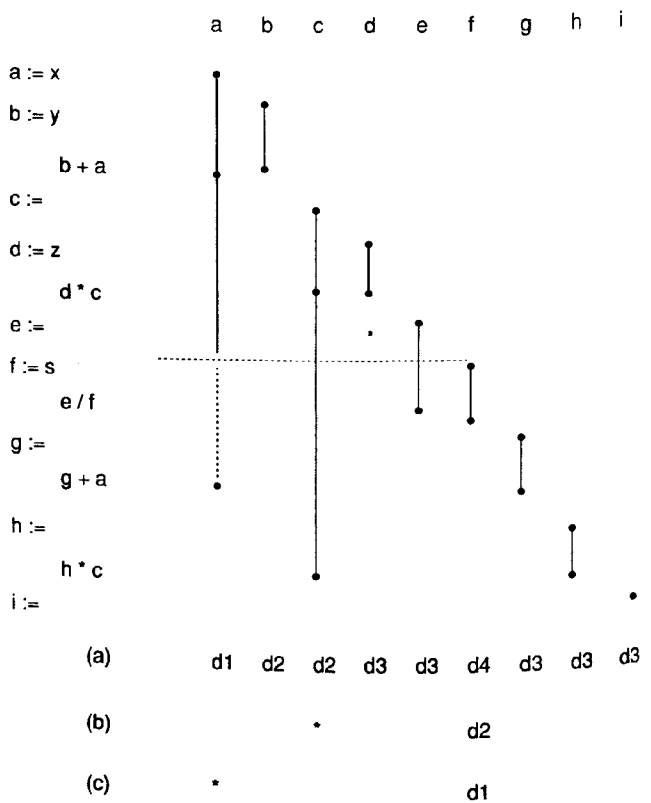


Abb. 7.5-3: Lebensdauer der Werte eines Grundblocks.

Abbildung 7.5-3 (a) gibt eine Zuteilung von mindestens vier verfügbaren Registern an. Das Maximum wird beim Laden von `f` erreicht. Stehen nur drei Register zur Verfügung, so muß ein Wert zwischengespeichert werden. Wählt man `c` als den am spätesten wiederverwendeten Wert, so muß er gespeichert und vor der Berechnung von `i` wieder geladen werden (Fall (b)). Es ist jedoch günstiger, den Wert `a` zu wählen (Fall (c)), denn er kann ohne zusätzliches Speichern aus der Variablen `x` wiedergeladen werden. Die allgemeine Strategie lautet deshalb: Wähle bei Registermangel ein Register, dessen Inhalt schon im Speicher steht. Falls kein solches existiert, wähle das am spätesten wiederverwendete Register.

7.5.3 Optimale Registerzuteilung für Ausdrucksbäume

Registerbedarf und Auswertungsreihenfolge

Für Ausdrucksbäume, in denen jedes Zwischenergebnis genau einmal verwendet wird, kann man die Auswertungsreihenfolge so bestimmen, daß die Zahl der verwendeten Register und damit der zu erzeugende Code zum Zwischenspeichern minimal sind (SETH70). Dazu berechnet man in einem Aufwärtsdurchgang für jeden Teilbaum den Registerbedarf und für jeden mehrstelligen Operator die Reihenfolge, in der die Operandenbäume ausgewertet werden. In einem nachfolgenden zweiten Baumdurchgang teilt man abwärts die Register zu und erzeugt aufwärts den Code in der vorher bestimmten Reihenfolge.

Abbildung 7.5-4 beschreibt diese Berechnung durch Attributierungsregeln für Ausdrucksbäume. Hier sind nur die Regeln für zweistellige Operationen angegeben. Die für einstellige Operationen und für Blätter des Baumes kann der Leser leicht ergänzen. Zur besseren Übersicht sind die Nichtterminale für die Wurzel des betrachteten Teilbaumes (`Form`) und seiner Operandenteilbäume (`Li`, `Re`) unterschiedlich benannt. Die verwendeten Attribute haben folgende Bedeutung:

`Bedarf`	Anzahl der zur Auswertung des Baumes benötigten Register, nicht größer als die Zahl `Max` der insgesamt vorhandenen Register,
`Folge`	Reihenfolge in der die Operanden ausgewertet werden (`links` oder `rechts`),
`Speicher`	Entscheidung, ob zwischengespeichert wird,
`Reg`	das Register, in dem das Ergebnis des Teilbaumes anfällt,
`Verfügbar`	Menge der Register, die zur Auswertung des Teilbaumes benutzt werden können,
`Code`	Folge von Instruktionen, die gemäß der Auswertungsreihenfolge zusammengesetzt wird.

Die angegebenen Regeln beruhen auf folgenden Überlegungen: Das Ergebnis des zuerst ausgewerteten Operanden belegt während der Auswertung des zweiten Operanden ein Register. Deshalb wertet man den

```
Produktion:        Form ::= Li Opr Re

Auswertungsreihenfolge und Registerbedarf:
   Form.Folge :=
      if Li.Bedarf < Re.Bedarf
      then rechts else links fi;
   Form.Speicher :=
      (Li.Bedarf = Max) and (Re.Bedarf = Max);
   Form.Bedarf :=
      if Form.Speicher then Max else
      if Form.Folge = links
         then max (Li.Bedarf, Re.Bedarf+1)
         else max (Li.Bedarf+1, Re.Bedarf)
      fi fi;

Registerzuteilung:
  Li.Reg := Form.Reg;
  Re.Reg := Waehle_Reg (Form.Verfuegbar);

  Li.Verfuegbar :=
      if Form.Folge = links or Form.Speicher
      then   Form.Verfuegbar
      else   Form.Verfuegbar - {Re.Reg} fi;

  Re.Verfuegbar :=
      if Form.Folge = rechts or Form.Speicher
      then   Form.Verfuegbar
      else   Form.Verfuegbar - {Li.Reg} fi;

Code-Zusammensetzung:
  Form.Code :=
      if Form.Speicher then
           Re.Code & Speichern (Re.Reg) &
           Li.Code & Laden (Reg.Reg)  &
           Instr (Opr.Op, Li.Reg, Re.Reg)
      else if Form = links then
           Li.Code & Re.Code &
           Instr (Opr.Op, Li.Reg, Re.Reg)
      else Re.Code & Li.Code &
           Instr (Opr.Op, Li.Reg, Re.Reg)
      fi fi
```

Abb. 7.5-4: Registeroptimale Auswertungsreihenfolge von Ausdrucksbäumen.

Operanden mit höherem Registerbedarf zuerst aus (Regeln für `Folge` und `Bedarf`). Zur Bestimmung der Register, in denen die Operandenergebnisse anfallen, wird hier die Restriktion für 2-Adreßinstruktionen angenommen. Das Register für den rechten Operanden kann beliebig aus der Menge der für den Teilbaum verfügbaren Register gewählt werden. Je nach Auswertungsreihenfolge vermindert man für den zweiten Operanden die verfügbaren Register um das Ergebnisregister des ersten Operanden. An den Blättern des Baumes bestimmt man den Registerbedarf zu 1, falls Operandenwerte immer im Register vorliegen müssen. Kann der

rechte Operand einer Instruktion auch ein Speicheroperand sein, so ist der Bedarf rechter Blätter zweistelliger Operationen 0.

Man muß genau dann zwischenspeichern, wenn beide Operanden alle vorhandenen Register benötigen. Das Ergebnis des zuerst ausgewerteten Operanden wird dann während der Auswertung des zweiten Operanden zwischengespeichert. Code für das Wiederladen kann entfallen, wenn die Zwischenspeicherstelle als Speicheroperand in die Instruktion eingesetzt werden kann (hier nicht berücksichtigt). Deshalb ist es in diesem Fall günstiger, erst den rechten Operanden auszuwerten. Statt die Code-Stücke zu konkatenieren, wie in der Attributierung mit & angegeben, kann man auch den Baum im zweiten Durchgang in der vorher bestimmten Auswertungsreihenfolge durchlaufen.

Abbildung 7.5-5 gibt zu dem Ausdrucksbaum aus Abbildung 7.5-1 den an jedem Knoten berechneten Registerbedarf und den bei Verwendung von drei Registern erzeugten Code an. Nur an der Baumwurzel wird der rechte Teilbaum vor dem linken ausgewertet. Stünden in diesem Beispiel nur zwei Register zur Verfügung, so würde zweimal zwischengespeichert werden: für die Ergebnisse der Subtraktion und der Division.

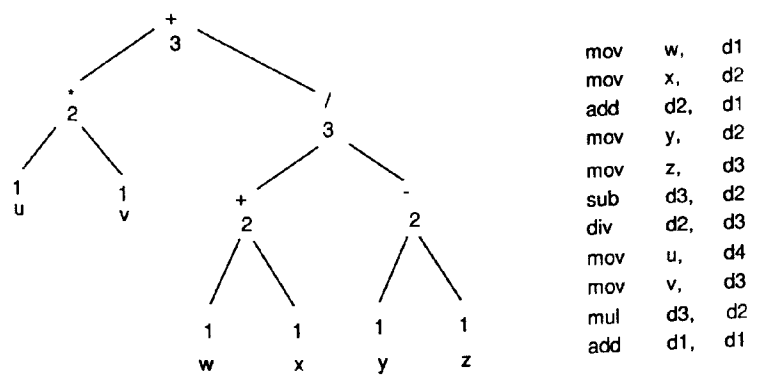


Abb. 7.5-5: Registerbedarf und optimaler Code für einen Ausdrucksbaum.

Suboptimale Fälle

Das hier vorgestellte Verfahren liefert optimalen Code für Ausdrucksbäume, falls nur Register einer einzigen Klasse gleichwertiger Register verwendet werden. Man kann zeigen, daß in diesem Fall optimaler Code durch rekursives, am Baum orientiertes Zusammensetzen der Code-Stücke der Teilbäume erzeugt werden kann. Falls verschiede-

ne Registerklassen zu berücksichtigen sind (z. B. ganzzahlige und Gleitpunktregister) oder Zwischenergebnisse mehr als ein Register belegen, kann es sein, daß für gewisse Ausdrucksbäume ein optimaler Code nicht diese Struktur hat. Man kann das oben beschriebene Verfahren durchaus auf solche Randbedingungen erweitern. Es erzeugt dann in einigen Fällen nur suboptimalen Code. Das Verfahren ist natürlich nicht anwendbar, wenn durch Identifikation von gleichen Teilbäumen ein azyklischer Graph anstelle eines Baumes vorliegt.

Da in diesem Verfahren die Register in einem Abwärtsdurchgang durch den Baum zugeteilt werden, können dabei Vorgaben aus dem äußeren Kontext berücksichtigt werden (targeting). So kann man festlegen, daß das Gesamtergebnis z. B. in einem für die Parameterübergabe oder für das Funktionsergebnis vorgesehenen Register anfällt. Auch im inneren des Ausdrucks können ggf. Restriktionen der erzeugten Operation berücksichtigt und zusätzliches Kopieren von Registerinhalten vermieden werden.

7.5.4 Registerzuteilung durch Graphfärbung

Wir stellen in diesem Abschnitt ein Verfahren der Registerzuteilung vor, das im Gegensatz zu den bisher betrachteten auf verzweigte Programmstücke (z. B. Prozedurrümpfe) anwendbar ist. Man kann damit entscheiden, welche mehrfach verwendeten Werte von Variablen und Ausdrücken in Registern gehalten werden, und ihnen die Register so zuteilen, daß möglichst wenig Code zum Zwischenspeichern erzeugt werden muß. Das auf Graphfärbung basierende Verfahren wurde von Chaitin in CHAI82 beschrieben.

Betrachten wir als Beispiel die Programmstruktur in Abbildung 7.5-6. Die Kästen repräsentieren Grundblöcke, deren Ablaufstruktur durch die verbindenden Pfeile dargestellt ist. Zu jedem Block ist angegeben, welche der Werte *a-f* dort berechnet werden (mit := gekennzeichnet) oder dort existieren, weil sie in dem Block oder in einem seiner Nachfolger benutzt werden. Zur Vereinfachung der Darstellung vergröbern wir hier die Lebensdauer der Werte auf ganze Grundblöcke. Man überlegt sich leicht, daß man für einen so verzweigten Ablauf kein Diagramm für die Intervalle der Lebensdauer der Werte wie in Abbildung 7.5-3 aufstellen kann.

Färbung des Unverträglichkeitsgraphen

Will man den symbolisch benannten Werten jeweils ein reales Register zuordnen, so müssen die in einem Block verwendeten Werte in paarweise verschiedenen Registern untergebracht werden. Die Restriktion beschreiben wir durch einen *Unverträglichkeitsgraphen* mit den vorkommenden Werten als Knoten, Abbildung 7.5-7. Zwei im gleichen Block verwendete Werte verbinden wir durch eine Kante.

Die Registerzuteilung reduziert sich damit auf das bekannte Problem der *Graphfärbung*: Jedem Knoten ist ein Register (eine Farbe) so

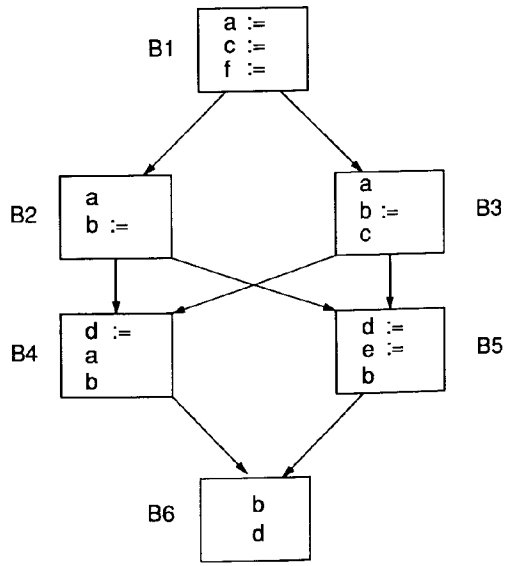


Abb. 7.5-6: Ablaufgraph mit Benutzung von Werten.

zuzuordnen, daß es verschieden von denen seiner Nachbarn ist. Abbildung 7.5-7 gibt eine Zuordnung von drei Registern an. Zu entscheiden, ob ein Graph mit k Farben färbbar ist, ist ein NP-vollständiges Problem. In der Praxis sind heuristische Verfahren ausreichend, z. B. eines, das aus folgender Überlegung hergeleitet wird: Ist ein Graph G mit k Farben gefärbt, so kann man diese Färbung auf einen Graphen G' erweitern, der einen zusätzlichen Knoten kleineren Grades als k hat. Offensichtlich kommt man mit k Farben aus, wenn der Graph keinen Knoten vom Grade k oder größer enthält. Aus dem ursprünglichen Graphen entfernt man deshalb nacheinander solche Knoten mit ihren Kanten, deren verbliebener Grad kleiner als k ist; in Abbildung 7.5-7 mit $k = 3$ z. B. in der Reihenfolge *f-e-d-c-b-a*. Wird damit der Graph geleert, so ist er k-färbbar. Wir ordnen dann den Knoten in umgekehrter Reihenfolge die Register so zu, daß sie mit den Nachbarknoten verträglich sind. Wir erhalten dann z. B. die in Abbildung 7.5-7 angegebene Zuordnung der Register *d1, d2, d3*.

Wird der Graph nicht geleert, so muß zwischengespeichert werden. Dazu eliminieren wir Kanten aus dem Graphen, indem wir in einigen Blöcken Werte im Speicher statt im Register unterbringen. Das Zuteilungsverfahren wird mit dem modifizierten Graphen dann erneut angewandt. In unserem Beispiel kommt man so mit zwei Registern aus,

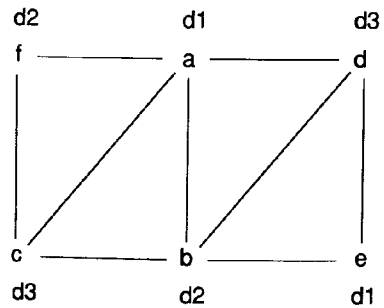


Abb. 7.5-7: Unverträglichkeitsgraph zu Abbildung 7.5-6.

wenn man *a* in Block 3 zwischenspeichert. Die Entscheidung für welche Werte zwischengespeichert wird, fällt man zweckmäßig unter Berücksichtigung der Kosten, die von der Verwendung der Werte in den Blöcken abhängen.

Insgesamt setzt die Anwendung dieses Verfahrens Kenntnisse über die Existenz von Werten im Programmablauf voraus, die durch globale Datenflußanalyse (s. Kap. 8) ermittelt werden.

7.6 Assemblierung

In der letzten Phase der Übersetzung, der *Assemblierung*, werden die Instruktionen in das Bitmuster des Instruktionsformates transformiert, Sprungadressen bestimmt und das übersetzte Programm als Code-Datei ausgegeben. Sie enthält zusätzliche Informationen, insbesondere über Speicheradressen, die es erlauben, Übersetzungseinheiten mit anderen zusammenzubinden, das Programm zur Ausführung zu laden und ggf. bei der Ausführung Testhilfen anzuwenden. Das Format der Code-Datei wird durch den Binder und Lader des Betriebssystems bestimmt, das Instruktionsformat durch die Spezifikation des Zielprozessors.

Assemblierung im Übersetzer

Zur Lösung dieser Aufgaben kann man entweder einen in der Grundsoftware im allgemeinen vorhandenen Assemblierer an den Übersetzer anschließen oder einen Assemblierermodul entwickeln und in den Übersetzer integrieren. Im ersten Fall wird eine Datei mit *symbolischem Assemblercode* erzeugt und durch Aufruf des Systemassemblierers in eine Code-Datei transformiert. Dieses Vorgehen erfordert nur geringen Entwicklungsaufwand. Allerdings wird die Gesamtlaufzeit des Übersetzers durch das zusätzliche Schreiben und Lesen der symbolischen Datei erheblich verlängert. Auch lassen sich auf diesem Wege im allgemeinen keine quellsprachbezogenen Informationen für Testhilfen

in die Code-Datei einbringen. Für die Entwicklungsphase des Übersetzers und für Übersetzerprototypen ist der Anschluß des Systemassemblierers zweckmäßig, um Übersetzungsergebnisse einfach überprüfen zu können. Verwendet man zur Erzeugung des symbolischen Codes die im folgenden beschriebenen Schnittstellen, so kann man später statt dessen einen Assemblierermodul integrieren.

7.6.1 Formatierung der Instruktionen

Der Zielprozessor spezifiziert die Repräsentation von Maschineninstruktionen als Bitmuster in aufeinanderfolgenden Speicherzellen. Solch ein Instruktionsformat ist ein Verbund mit Bitfeldern für den Operationscode und im allgemeinen unterschiedlich strukturierten Feldern für verschiedene Adressierungsarten der Operanden. Je nach Kombination der Adressierungsarten können die Formate auch unterschiedliche Länge haben.

Der Assemblierer speichert den Code für eine Übersetzungseinheit in einer internen Datenstruktur. Falls die Code-Auswahl die Instruktionen in ihrer endgültigen Reihenfolge generiert, genügt eine einfache Reihung. Andernfalls legt man Code-Blöcke in einer verketteten Struktur an und fügt sie bei der Ausgabe zusammen. Je nach Implementierungssprache wählt man den Elementtyp so, daß die Indizes möglichst den Relativadressen entsprechen und daß die Werte einzelner Instruktionsfelder durch einfache Operationen (Zuweisung an Bitfelder in niederen Programmiersprachen) bestimmt werden können.

Code-Speicher

Als Schnittstelle zur Code-Auswahl stellt der Assemblierermodul einen Satz von Prozeduren zur Verfügung:

Schnittstelle

```
procedure emit0 (instr : Opcode);
procedure emit1 (instr : Opcode; op1 : Wertdeskr);
procedure emit2 (instr : Opcode; op1, op2 :
Wertdeskr);
   . . .
```

Jeder Aufruf der Prozeduren repräsentiert die Abstraktion einer zulässigen Maschineninstruktion, die in der spezifizierten Codierung an der aktuellen Stelle in der Code-Datenstruktur eingesetzt wird. Die Operanden werden durch Wertdeskriptoren angegeben, wie sie in Abschnitt 7.3 für die Adressierungsarten eingeführt wurden. Etwaige Restriktionen der Kombinierbarkeit von Operationscode und Adressierungsarten müssen schon bei der Code-Auswahl berücksichtigt werden. Die Rümpfe der Prozeduren bilden aus den Parametern das Bitmuster des entsprechenden Instruktionsformates, tragen es in den internen Code-Speicher ein und inkrementieren den Zeiger auf die aktuelle Position.

7.6.2 Sprungadressierung

Die Code-Erzeugung plaziert Marken im Code und generiert Instruktionen mit Marken als Sprungzielen. Da insbesondere bei der Erzeugung von Vorwärtssprüngen die Stelle der Marke im Code noch nicht bestimmt ist, werden aus der Sicht der Code-Erzeugung symbolische Marken verwendet, denen erst der Assemblierer Code-Stellen zuordnet.

Der Assemblierer stellt deshalb an der Schnittstelle die zwei Prozeduren zur Verfügung:

```
function generiere_Marke: SymbMarke;
procedure plaziere_Marke (m : SymbMarke);
```

Markentabelle

Ein Aufruf von `generiere_Marke` führt eine symbolische Marke zur Verwendung in Sprunginstruktionen ein. Durch Aufruf von `plaziere_Marke` wird ihr die aktuelle Position im Code zugeordnet. Im Assemblierer wird eine Markentabelle angelegt, die den symbolischen Marken (als Tabellenindex) ihre Code-Stelle zuordnet. Solange eine Marke noch nicht plaziert ist, enthält der Tabelleneintrag den Anfang einer Liste aller noch nicht aufgelösten Verwendungen der Marken. Bei der Erzeugung einer Sprunginstruktion durch eine `emit`-Prozedur setzt man für die angegebene symbolische Marke den Tabelleneintrag ein. Dies ist entweder die zugeordnete Code-Stelle (Rückwärtssprung) oder der Verweis auf die Liste nicht aufgelöster Verwendungen (Vorwärtssprung). Im zweiten Fall wird die Liste um die neue Instruktion verlängert. Beim Plazieren einer Marke wird die zugehörige Liste durchlaufen und dabei die Code-Stelle in die Sprunginstruktionen eingetragen. In Sprunginstruktionen, die relativ zum Instruktionszeiger adressieren, wird die Differenz zwischen der Code-Stelle der Marke und der Position der Instruktion als Relativadresse eingesetzt.

Kurze und lange Sprünge

Der Wertebereich von Markenadressen, die wie oben beschrieben als konstante Operanden in Instruktionen eingesetzt werden, ist durch den Umfang des dafür vorgesehenen Instruktionsfeldes begrenzt. Damit ist die maximal erreichbare Sprungdistanz limitiert. Wird sie überschritten, kann man meist eine andere Sprunginstruktion wählen (z. B. Laden der Sprungadresse in ein Register und indirekten Sprung). Solch eine Instruktionsfolge ist im allgemeinen länger als ein einfacher Sprung. Die Länge einer Sprunginstruktion ist dann distanzabhängig. Die Entscheidung über die Sprungart wird wiederum durch die Länge der zwischen Sprung und Marke liegenden distanzabhängigen Sprünge bestimmt. Man kann den Entscheidungszyklus durch zwei grobe Maßnahmen aufbrechen: Alle distanzabhängigen Sprünge werden in der Langform realisiert. Dadurch wird neben der Code-Länge auch meist die Ausführungszeit unnötig vergrößert. Realisiert man alle distanzabhängigen Sprünge kurz, so sind einige von ihnen nicht korrekt übersetzbar.

Eine solche Übersetzerbeschränkung, die der Assemblierer erkennen, aber dann nicht mehr beheben kann, ist nur akzeptabel, wenn der Wertebereich der Sprungmarken so groß ist, daß er nur in sehr extremen Fällen überschritten wird.

Eine differenzierte Behandlung der Situation erfordert erheblichen algorithmischen und programmtechnischen Aufwand. Die Datenstruktur für die Instruktionsfolge muß so gewählt werden, daß zunächst in der Kurzform erzeugte Sprünge später in die Langform umgewandelt werden können. Zu jedem distanzabhängigen Sprung s_i bestimmt man nun die Menge A_i der distanzabhängigen Sprünge, von denen die Entscheidung für s_i abhängt. A_i enthält alle s_j, die zwischen s_i und seiner Zielmarke m_i liegen. Außerdem berechnet man zu jedem s_i den Wert d_i, um den sich m_i von s_i entfernen darf (durch Verlängern von Sprüngen in A_i), ohne daß für s_i die Langform gewählt werden muß. Nun kann man sukzessive für solche s_i die Kurzform wählen, die d_i nicht überschreiten, falls alle in A_i noch nicht entschiedenen Sprünge lang realisiert werden. Auch aus den dann noch verbleibenden unentschiedenen Sprüngen können mit einem allerdings komplexen Optimierungsverfahren einige als kurz realisierbar ermittelt werden.

7.6.3 Code-Datei

Informationen zum Code

Neben der Instruktionsfolge enthält eine Code-Datei weitere Informationen, die zum Laden des Programms, zum Binden getrennt übersetzter Module und Bibliotheksfunktionen, sowie zur Benutzung von Testhilfen notwendig sind. Weil die Repräsentation dieser Information durch Binder, Lader und Testhilfen des Betriebssystems vorgegeben und nicht wie das Instruktionsformat prozessorabhängig ist, sollte sie von einem separierbaren Modul innerhalb des Assemblierers erzeugt werden. Dadurch wird die Anpassung des Assemblierers an Systemänderungen und die Portierung auf ein anderes Betriebssystem für den gleichen Zielprozessor vereinfacht.

Strukturelle Information. Angaben wie Umfang und Lage der Segmente für die Instruktionsfolge, initialisierte und nichtinitialisierte Daten sowie Erzeugungsdatum, System- und Übersetzerversion werden im Kopf der Code-Datei zusammengefaßt.

Relativadressen. Alle Felder von Instruktionen, die Adressen relativ zum Anfang des Instruktions- bzw. Datensegmentes enthalten, werden aufgezählt, damit sie beim Zusammenfügen von Modulen entsprechend adjustiert werden können.

Externreferenzen. Alle Felder von Instruktion, die Referenzen auf Adressen außerhalb des Moduls enthalten, werden aufgezählt und der für sie verwendete symbolische Name angegeben. Alle aus anderen Modulen referierbaren Adressen werden zusammen mit ihrem symboli-

schen Namen aufgezählt. Der Binder löst diese Querbezüge zwischen den Übersetzungseinheiten auf.

Testinformation. Die Struktur von Datenbereichen (globalen Daten, Prozedurschachteln, Verbunde) wird ähnlich wie im Definitionsmodul durch Namen und Typangaben des Quellprogramms beschrieben. Programmstellen im Quellprogramm werden Adressen in der Instruktionsfolge zugeordnet. Mit solchen Informationen kann ein Testhilfewerkzeug den Ablauf des Programms in Termen der Quellsprache sichtbar machen und beeinflussen.

Zur Erzeugung der Code-Datei wird obige Information zunächst in eine interne Datenstruktur eingetragen und nach Abschluß der Code-Erzeugung ausgegeben. Die auf Instruktionen bezogenen Angaben (Relativadressen, Externreferenzen) werden bei der Verarbeitung entsprechender Wertdeskriptoren in den `emit`-Prozeduren bestimmt. Testinformationen sowie Einträge für von außen referierbare Objekte können im wesentlichen unmittelbar von dem Speicherzuteilungsmodul angestoßen und mit Angaben aus dem Definitions- und dem Bezeichnermodul berechnet werden. Die Zuordnung von Programmadressen entnimmt man der Markentabelle.

8. Optimierung

Überblick

Übersetzer, die unter realistischen Bedingungen praktikabel einsetzbar sein sollen (Produktionsübersetzer), müssen Zielprogramme möglichst guter Qualität erzeugen. Höchstens für Prototypübersetzer und solche, die ausschließlich in der Programmentwicklung eingesetzt werden, kann man sich mit Code geringerer Qualität zufrieden geben. Das dominierende Qualitätskriterium ist die Ausführungszeit der Programme. Das Maß des Code-Umfangs wird meist nachrangig berücksichtigt. Im Übersetzerbau hat sich der Begriff Optimierungen für Maßnahmen zur Verbesserung der Zielprogramme eingeprägt, obwohl optimale Lösungen für realistische Zielmaschinen und beliebige Programme mit vertretbarem Aufwand nicht erreichbar sind.

Das Problem der Programmoptimierung kann in drei Teilaufgaben zerlegt werden: Verbessernde Transformationen, die redundante Berechnungen elimieren oder vereinfachen, müssen entwickelt und die Voraussetzungen und Randbedingungen für ihre Anwendung spezifiziert werden. Über große Bereiche der Ablaufstruktur des Programms müssen Informationen berechnet werden, um die Anwendbarkeit bestimmter Transformationen festzustellen. Diese Datenflußanalyse ist eine algorithmisch komplexe Aufgabe. Schließlich werden die Transformationen, die sich häufig gegenseitig beeinflussen und die Optimierungsinformation zum Teil wieder verändern, zu einer insgesamt wirkungsvollen Optimierungsphase zusammengefügt. Im Zentrum dieses Kapitels stehen systematische Methoden zur Berechnung der Optimierungsinformation (Abschnitt 8.2). In Abschnitt 8.3 zeigen wir, wie diese für einige der wichtigsten Transformationen genutzt wird.

Datenflußanalyse

Die Systematik der Datenflußanalyse beruht auf folgenden Prinzipien: Die verzweigte Kontrollstruktur des Programms wird durch einen Ablaufgraphen abstrahiert. Anhand seiner Struktur stellt man ein Gleichungssystem für bestimmte ablaufabhängige Eigenschaften auf. Mit standardisierten Algorithmen wird das Gleichungssystem gelöst. Die Ergebnisse liefern Informationen, die für jeden beliebigen Ablauf des Programms gültig sind. Dieses Schema wird auf beeindruckend viele verschiedene Probleme der Programmoptimierung angewandt. Darüber hinaus können nach dem gleichen Schema auch andere ablaufbedingte Eigenschaften z. B. Aussagen zur Programmvalidierung oder für das systematische Testen berechnet werden. Außerhalb des Übersetzerbaus kann man mit diesem Schema Probleme lösen, die durch gerichtete Graphen beschrieben werden, deren Knoten bestimmte Informationen transformieren, wie etwa zur Analyse der Eigenschaften von Schaltkreisen.

Programmtransformation

Maßnahmen zur Programmverbesserung werden meist als Transformationen formuliert, die ein Programmstück unter gewissen Voraussetzungen in ein verbessertes mit gleicher Wirkung überführen. Solche Transformationen verlagern Berechnungen in die Übersetzungszeit, entfernen überflüssige Berechnungen, vereinfachen sie oder verschieben sie an andere Programmstellen. In Abschnitt 8.1 stellen wir einige wichtige Optimierungsmaßnahmen vor, die zunächst auf den recht engen Kontext sogenannter Grundblöcke beschränkt werden.

Übersetzerstruktur

Verbessernde Transformationen setzen auf verschiedenen Ebenen der Programmrepräsentation an. Einige Verbesserungen sind durchaus auf Quellsprachniveau formulierbar. Da sie bei Anwendung in der Programmentwicklung Lesbarkeit und Klarheit des Programms vermindern würden, sind auch sie Gegenstand der Übersetzeroptimierung. Andere sind erst in der Zwischensprache anwendbar, in der z. B. Adressierungsoperationen explizit eingesetzt sind. Solche Transformationen werden in einer Optimierungsphase durchgeführt, die zwischen der Erzeugung der Zwischensprache und der Code-Auswahl in die Übersetzerstruktur eingefügt wird. Diese Optimierungen sind weitgehend unabhängig von der Quell- und Zielsprache. Die Voraussetzungen zur Anwendung solcher Transformationen erfordern meist Analysen des Programms mit seinen Ablaufstrukturen. Deshalb werden in dieser Optimierungsphase Verfahren zur Datenflußanalyse eingesetzt (Abschnitt 8.2). In Abschnitt 8.3 zeigen wir, wie die Datenflußanalyse die Wirksamkeit von Transformationen aus 8.1 erhöht und weitere Verbesserungen ermöglicht.

Prozeduren

Die Datenflußanalyse kann meist keine exakte Optimierungsinformation berechnen, die für alle Programmausführungen gilt. Es muß deshalb garantiert werden, daß die Information in jedem Fall pessimistisch bestimmt wird, damit darauf basierende Transformationen die Korrektheit der Übersetzung erhalten. Besondere Probleme verursachen Sprachelemente wie Referenzobjekte, Referenzparameter und Prozeduraufrufe, die verhindern, daß die Identität von Variablen und ihre Benutzung statisch unmittelbar festgestellt werden kann. Durch spezielle Analysen auch der Aufrufbeziehungen zwischen Prozeduren kann man dazu präzisere Optimierungsinformation gewinnen. Wir gehen hierauf am Ende von Abschnitt 8.3 näher ein. Bis dahin ignorieren wir diese Effekte zur Vereinfachung der Darstellung.

Wechselbeziehungen

Zwischen den in diesem Kapitel beschriebenen verbessernden Transformationen bestehen vielfältige Wechselbeziehungen: Die Anwendung einer Transformation kann Voraussetzungen für andere Transformationen an anderen Programmstellen schaffen. Das Gesamtergebnis der Optimierungsphase wird deshalb wesentlich durch die Abfolge und eventuelle Wiederholung der einzelnen Transformations- und Analyseschritte bestimmt. Hierzu wendet man empirisch begründete Strategien an, auf die wir hier nicht weiter eingehen.

Nachoptimierung

Eine andere Klasse von Optimierungsmaßnahmen ist typisch maschinenabhängig und wird auf der Folge abstrakter Instruktionen als sogenannte Nachoptimierung zwischen Code-Auswahl und Assemblierung durchgeführt. Hier werden zusätzliche Verbesserungen im wesentlichen aus zwei Gründen erzielt: Die Code-Auswahl setzt die Instruktionsfolge aus Teilstücken gemäß der Code-Sequenzen zusammen. In den Grenzbereichen der Teilstücke können vorher nicht erkennbare Redundanzen eliminiert werden. Weiter können spezielle Maschineninstruktionen mit komplexer Wirkung, die bei der Code-Auswahl nicht berücksichtigt wurden, in der Nachoptimierung für kurze Instruktionsfolgen substituiert werden (Abschnitt 8.4).

Parallelverarbeitung

Prozessorarchitekturen, die auf interner Parallelverarbeitung basieren, stellen zusätzliche Anforderungen an die Erzeugung des Codes. Die Instruktionen müssen unter den Randbedingungen der parallelen Abarbeitung geeignet angeordnet werden. In Abschnitt 8.5 führen wir Anordnungsverfahren ein, die mehrere parallele Instruktionsströme für Prozessoren mit mehreren Funktionseinheiten oder einen Instruktionsstrom für prozessorinterne Fließbandverarbeitung herstellen. Häufig ist die passende Anordnung der Instruktionen Voraussetzung für eine korrekte Ausführung. In jedem Fall wird die Leistungsfähigkeit solcher Prozessoren erst durch diese Optimierungen wirksam. Die betrachteten Verfahren basieren auf allgemeinen Methoden der Anordnungstheorie.

8.1 Verbessernde Transformationen in Grundblöcken

Grundblock

In diesem Abschnitt beschreiben wir einige verbessernde Transformationen im Kontext von Ausdrücken und Grundblöcken. Man spricht hier auch von *lokaler Optimierung*. Beispiele geben wir soweit möglich zur Vereinfachung der Darstellung in Termen einer Quellsprache an, obwohl die Maßnahmen in der Optimierungsphase auf der Programmdarstellung in der Zwischensprache durchgeführt werden.

Ein *Grundblock (basic block)* ist eine Anweisungsfolge maximaler Länge, deren Anweisungen alle genau dann ausgeführt werden, wenn die erste ausgeführt wird.

Ein Grundblock beginnt also mit einer als Sprungziel markierten Auswertung, endet mit einem Sprung oder einer Programmverzweigung zu anderen Grundblöcken und enthält keine weiteren Marken oder Sprünge. Die Verzweigungsstruktur wird durch Ablaufgraphen mit Grundblöcken als Knoten beschrieben. Sie sind Grundlage der Datenflußanalyse in Abschnitt 8.2.

8.1.1 Konstantenfaltung und algebraische Vereinfachung

Konstante Operanden

Sind die Werte sämtlicher Operanden einer Formel zur Übersetzerzeit bekannt, so kann der Übersetzer ihr Ergebnis berechnen und dies anstelle der Formel einsetzen (*Konstantenfaltung*). Solche Formeln kommen auf Quellsprachniveau recht selten unmittelbar vor, z. B. 2*pi*, mit *pi* als definierter Konstante. Innerhalb von Adressierungsausdrükken in den Zwischensprachoperationen treten sie jedoch häufig auf. So wird die Speicheradresse von `a[5].k` nach der Formel $s + ra + (5 - ug) * e + rk$ berechnet, wobei s der Schachtelanfang, ra die Anfangsadresse von `a`, ug dessen untere Indexgrenze, e seine Elementlänge und rk die Relativadresse innerhalb eines Verbundes ist. Außer s sind alle Werte zur Übersetzungszeit bestimmbar. Die Formel kann auf eine einzige Adreßaddition reduziert werden.

Algebraische Gesetze

Auch Formeln deren Operanden nicht alle konstant sind, und deren Auswertung keine Seiteneffekte verursacht, können durch Anwendung *algebraischer Gesetze* vereinfacht werden.

So gelten z. B. folgende Identitäten

```
x + 0 = x        x * 0 = 0
x * 1 = x        x or true = true
x and true = x.
```

Ebenso kann man Operationen so umformen, daß Operatoren mit geringerer Ausführungsdauer verwendet werden (*strength reduction*), z. B.

```
x ** 2    =    x * x
x * 2     =    x + x
x/2       =    x * 0.5.
```

Auf Instruktionsniveau zählt auch das Ersetzen von Multiplikationen und Divisionen mit Zweierpotenzen durch Shift-Instruktionen zu dieser Klasse von Vereinfachungen.

Überlauf

Für die Durchführung der Konstantenfaltung im Übersetzer ist besondere Vorsicht geboten, damit die Semantik des Quellprogramms erhalten bleibt. Alle Rechnungen müssen in der Arithmetik der Zielmaschine durchgeführt werden. Führt die Auswertung einer konstanten Formel zu einer Wertebereichsüberschreitung, so darf die Übersetzung nicht dadurch terminieren! Entsprechende Prüfungen müssen deshalb vor Ausführen der Rechnung vorgenommen werden. Auch sollte das Programm in solch einem Fall nicht als fehlerhaft gekennzeichnet werden, da zur Laufzeit die Formel möglicherweise nicht ausgewertet wird. Eine Warnung und Verzicht auf die Faltung wären angebracht. Es ist deshalb zweckmäßig, sämtliche solche Operationen für die verschiedenen Datentypen in den Literalmodulen zusammenzufassen. Dadurch

wird auch die Portierung des Übersetzers und die Anpassung von Querübersetzern an andere Zielmaschinen erleichtert.

Konstantenweitergabe

Die Wirksamkeit der Konstantenfaltung kann auch außerhalb von Adressierungsausdrücken wesentlich erhöht werden, wenn man den Analysekontext über die Formeln selbst hinaus vergrößert. Die Kenntnis, daß ein konstanter Wert w an eine Variable v zugewiesen wird, kann in nachfolgenden Formeln zur Faltung ausgenutzt werden. Wir nennen solch ein Verfahren *Konstantenweitergabe*. Zu der Anwendung einer Variablen v in einer Formel stellt man fest, ob die letzte im Grundblock vorangehende Zuweisung an v - falls eine solche existiert - der Variablen einen konstanten Wert w zuweist. Wir ersetzen dann in der Formel v durch w. Dieses Verfahren kann auch über Grundblöcke hinaus ausgedehnt werden, indem man mit Hilfe der Datenflußanalyse die letzten vorangehenden Zuweisungen an v auf allen Wegen innerhalb des Ablaufgraphen bestimmt (s. Abschnitt 8.3).

8.1.2 Eliminieren von Bereichsprüfungen

Laufzeitprüfungen

Eine der Konstantenfaltung verwandte Optimierungsmaßnahme ist die Eliminierung von Bereichsprüfungen. In statisch typisierten, höheren Programmiersprachen können die Wertebereiche von ganzzahligen (oder auf ganze Zahlen abgebildeten) Aufzählungstypen explizit eingeschränkt werden. Es müssen dann Laufzeitprüfungen für die Einhaltung des Wertebereichs bei jeder Zuweisung an die Variable in den Code eingesetzt werden. Entsprechendes gilt für die Grenzenprüfung bei Reihungsindizierungen auch in nicht statisch typisierten Sprachen. Sind die Grenzen und der zu prüfende Wert bei der Übersetzung bekannt, so kann auf den Code für den Laufzeittest verzichtet werden. So wird bei der Zuweisung in folgendem Ausschnitt aus einem Pascal-Programm z. B. kein Laufzeittest benötigt:

```
var i : integer; j : 1..10;
    ...
for i := 1  to 5 do  ... j := 2 * i ...
```

Zur Entscheidung über die Notwendigkeit von Laufzeitprüfungen untersuchen wir ebenso wie bei der Konstantenweitergabe die letzten vorangehenden Zuweisungen an Variablen, um festzustellen, ob der Wert in einem zur Übersetzungszeit bestimmbaren Bereich liegt und dieser in dem geforderten enthalten ist. Selbstverständlich ist auch das Weglassen der Prüfung nur einer der beiden Grenzen eine Verbesserung der Übersetzung.

Ein Übersetzer, der diese Optimierungen durchführt, trägt auch wesentlich zur Sicherheit der übersetzten Programme bei. Er belohnt die möglichst präzise Begrenzung von Wertebereichen in Quellpro-

grammen durch geringe Laufzeitverluste und bewahrt den Programmentwickler vor der Versuchung, Laufzeitprüfungen durch eine Übersetzeroption abzuschalten, wenn das Programm angeblich ausgetestet ist und nun ernsthaft angewandt wird.

8.1.3 Gemeinsame Teilausdrücke

Häufig treten an verschiedenen Programmstellen Formeln auf, deren Auswertung das gleiche Ergebnis liefert. Wir sprechen dann von *gemeinsamen Teilausdrücken*, (*GTA*, *common subexpressions*). Ziel der Optimierung ist es, sie möglichst nur einmal auszuwerten, das Ergebnis (z. B. in einem Register) zu speichern und bei späterem Auftreten ohne Neuberechnung zu verwenden. Gemeinsame Teilausdrücke können im Kontext von Ausdrücken, Grundblöcken und Ablaufgraphen erkannt und ihre erneute Berechnung eliminiert werden.

Struktureller Vergleich

In einem Ausdruck erkennt man gemeinsame Teilausdrücke durch strukturellen Vergleich der Teilbäume - vorausgesetzt, die Auswertung von Operanden verursacht keine Seiteneffekte. Indentifiziert man gleiche Teilbäume, so wird der Ausdrucksbaum zu einem gerichteten azyklischen Graphen (directed acyclic graph, DAG). Im Ausdruck zu Abbildung 8.1-1 treten `a`, `b` und `a+b` mehrfach auf. Da `b` nur in dem größeren Ausdruck `a+b` vorkommt, braucht sein Wert nicht über die Berechnung von `a+b` hinaus gespeichert zu werden. Stehen genügend Register zur Verfügung, so speichert man darin die Werte mehrfach auftretender Ausdrücke. Im Gegensatz zur Bestimmung einer Auswertungsreihenfolge mit minimalem Registerbedarf für Bäume, gibt es im Falle von azyklischen Graphen keinen entsprechenden allgemeinen, optimalen und effizienten Algorithmus. Man wendet deshalb Heuristiken zur Bestimmung der Auswertungsreihenfolge an.

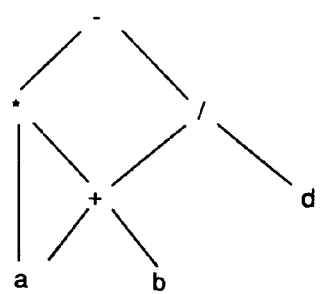


Abb. 8.1-1: DAG für den Ausdruck a*(a+b) - (a+b)/d.

Komplexe Ausdrücke mit gemeinsamen Teilausdrücken, wie der in Abbildung 8.1-1, kommen so in Quellprogrammen recht selten vor. Auf Zwischensprachebene ergeben aber insbesondere Adressierungsausdrükke (wie Reihungsindizierungen) zahlreiche gemeinsame Ausdrücke. Die Analyse lohnt sich deshalb erst auf dieser Ebene.

Analysiert man gemeinsame Teilausdrücke im Kontext von Grundblöcken, so ist die strukturelle Gleichheit weder hinreichend noch notwendig, wie die beiden folgenden Zuweisungssequenzen zeigen:

1) `a := b+c; b := e; d := b+c`

2) `a := b+c; d := b; e := d+c`

Symbolische Auswertung

In (1) tritt zwar der Ausdruck `b+c` zweimal auf, seine Auswertungsergebnisse sind jedoch nicht notwendig gleich. In (2) sind `b+c` und `d+c` zwar strukturell verschieden, aber semantisch gleichwertig. Um gemeinsame Teilausdrücke in einem Grundblock zu bestimmen, wendet man eine Technik der symbolischen Auswertung an: In einem Durchgang durch die Zuweisungsfolge numeriert man jeden neu auftretenden Wert eindeutig und ordnet die *Wertnummer* (*value number*) der zugewiesenen Variablen zu. Die obigen Beispiele lauten dann mit Wertnummern statt Variablenbezeichnern:

1) `a := "3" ; b := "4" ; d := "5"`
   mit `"3" = "1" + "2"` und `"5" = "4" + "2"`

2) `a := "3" ; d := "1" ; e := "3"`
   mit `"3" = "1" + "2"`

Mit dieser Technik werden alle Ausdrücke auf die am Anfang des Grundblockes gültigen Variablenwerte zurückgeführt (z. B. "1" = b, "2" = c, "4" = e in (1)). Nach solch einer Transformation können die Berechnungen der Ausdrücke und die Zuweisung beliebig umgeordnet werden, solange Ausdrücke vor ihrer Verwendung berechnet werden und keine Seiteneffekte auftreten. Ebenso wie im Falle von azyklische Graphen ist hier die Optimierung des Registerbedarfs ein nicht effizient lösbares Problem.

Zur Implementierung der Ausdrucksanalyse legt man zweckmäßig eine Datenstruktur an, die jedem Teilausdruck seine Identifikation zuordnet. Neu hinzukommende Teilausdrücke werden sukzessive von innen nach außen mit schon vorhandenen anhand ihres Operators und ihrer Operanden verglichen und nur aufgenommen, falls sie sich von allen anderen unterscheiden. Die Anzahl der Kandidaten für den Vergleich kann durch Hash-Verfahren reduziert werden. Für spätere Analysen ist es weiter zweckmäßig, zu jeder Variablen eine Liste der Ausdrücke an-

zulegen, in denen sie vorkommt. Die Werte dieser Ausdrücke werden durch eine Zuweisung an die Variable verändert.

Die Ergebnisse der Analyse von gemeinsamen Teilausdrücken können weiter verbessert werden, wenn der Vergleich von Ausdrücken (strukturell oder symbolisch) durch Anwendung algebraischer Gesetze (z. B. Kommutativität und Assoziativität) verfeinert und der Kontext auf den Ablaufgraphen ausgedehnt wird.

8.1.4 Überflüssige Zuweisungen

Lebendige Variablen

Die Berechnung und Zuweisung eines Variablenwertes, der in nachfolgenden Anweisungen nicht mehr benutzt wird, ist überflüssig und kann eliminiert werden. Dies gilt z. B. in Abbildung 8.1-2 für die mit (*) markierte Zuweisung. Wie in den vorangegangenen Abschnitten betrachten wir diese Verbesserung zunächst im Kontext von Grundblöcken, obwohl sie erst für Ablaufgraphen mit der Methode der Datenflußanalyse signifikante Wirkung entfaltet.

Um überflüssige Zuweisungen systematisch zu erkennen, unterscheiden wir zwischen der *Definition* einer Variablen auf der linken Seite einer Zuweisung und ihrer *Benutzung* in einer Formel.

Wir nennen eine Variable v an einer Programmstelle p *lebendig*, falls auf p eine Benutzung von v folgt bevor v erneut definiert wird.

Für die Programmstellen vor und nach einer Zuweisung können wir Mengen lebendiger Variablen als Vor- und Nachbedingung angeben:

$$\{ V \}\ x := E_{yi}\ \{ N \},$$

wobei E_{yi} eine Formel über den Variablen y_i sei. V wird dann aus N berechnet:

$$V = (N\text{-}\{x\}) \cup \{y_i\}.$$

Nehmen wir an, daß nach der letzten Zuweisung in Abbildung 8.1-2 alle im Block auftretenden Variablen lebendig sind, so ergeben sich nach obiger Formel die angegebenen Mengen lebendiger Variablen an den Programmstellen. Eine Zuweisung wie die markierte ist dann überflüssig, wenn ihre linke Seite nicht in der Nachbedingung enthalten ist.

Die Analyse kann auch auf andere Anweisungsformen im Grundblock ausgeweitet werden. So treten z. B. in Prozeduraufrufen Variablen in aktuellen Wertparametern als Benutzung und in aktuellen Ergebnisparametern als Definition auf. Referenzparameter muß man pessimistisch als Benutzung auffassen, solange nicht garantiert ist, daß sie in

		lebendig
		b, c
	a := b+c	
		a, c
(*)	b := c+1	
		a, c
	c := c+a	
		a, c
	b := a+5	
		b, c
	a := 3	
		a, b , c

Abb. 8.1-2: Lebendige Variablen im Grundblock.

der Prozedur vor einer Benutzung zugewiesen werden. Auf solche Untersuchungen gehen wir am Ende von Abschnitt 8.3 näher ein.

8.2 Datenflußanalyse

Die in Abschnitt 8.1 vorgestellten Maßnahmen zur Code-Verbesserung (Konstantenfaltung, gemeinsame Teilausdrücke, Eliminierung von Bereichsprüfungen und Zuweisungen) setzen voraus, daß gewisse Anwendbarkeitsbedingungen im Kontext gelten. Während wir die Analyse dieser Voraussetzungen bisher auf Grundblöcke beschränkt haben, erweitern die hier vorgestellten Techniken der *Datenflußanalyse* den Kontext auf Ablaufgraphen. Durch die Betrachtung eines größeren Kontextes werden weitere Anwendungsstellen für die Verbesserungen festgestellt und die Wirksamkeit der Maßnahmen wesentlich erhöht. Es muß betont werden, daß die Datenflußanalyse nur der Beschaffung von Information dient, die zur verbessernden Transformation der Programme herangezogen wird. Muchnick und Jones geben in MUCH81 eine umfassende Darstellung und Bibliographie zur Datenflußanalyse.

8.2.1 Ablaufgraphen

Programmstruktur als Ablaufgraphen

Die grundlegende Datenstruktur für die Algorithmen der Datenflußanalyse ist der *Ablaufgraph*. Seine Knoten sind die Grundblöcke, seine Kanten die bedingten und unbedingten Programmverzweigungen zwischen den Grundblöcken. Er beschreibt alle strukturell möglichen Ablaufpfade. Abbildung 8.2-1 gibt als Beispiel den Ablaufgraphen zu einem kleinen Programmstück mit vier Grundblöcken an. Die Anweisungssequenzen S_2 und S_4 bilden die Grundblöcke B_2 und B_4. Die Be-

```
S1;                          S1;
if C1                        if not C1 then goto M3;
    then S2                  S2;
                             goto M4;
    else repeat S3       M3: S3;
    until C3;                if not C3 then goto M3;
S4;                      M4: S4;
```

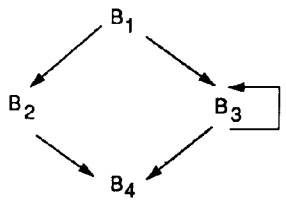


Abb. 8.2-1: Beispiel für einen Ablaufgraphen.

dingungen C_1 und C_3 werden jeweils mit der vorangehenden Anweisungsfolge zu den Blöcken B_1 und B_3 zusammengefaßt. Ablaufgraphen können entweder aus geschachtelten Ablaufstrukturen anhand des Strukturbaumes erzeugt werden, oder sie werden aus einer Programmrepräsentation niedrigeren Niveaus mit Marken und Sprüngen berechnet. Abbildung 8.2-1 gibt Programmstücke in beiden Varianten an, die zum gleichen Ablaufgraphen führen.

In den folgenden Überlegungen verwenden wir die Funktionen *Vorg (B)* und *Nachf (B)*, die jeweils die Menge der unmittelbaren Vorgänger- bzw. Nachfolgerknoten von B im Ablaufgraphen angeben. Wir setzen voraus, daß jeder Ablaufgraph jeweils einen eindeutigen Anfangsknoten B_a und Endknoten B_e hat, in dem alle Wege durch den Graphen beginnen bzw. enden. Außerdem sei *Vorg* $(B_a) = \emptyset$ und *Nachf* $(B_e) = \emptyset$.

Unerreichbarer Code

In einem Ablaufgraphen sollte jeder Knoten vom Anfangsknoten erreicht werden können. Grundblöcke, die nicht erreichbar sind, kann man weglassen, ohne die Wirkung des Programms zu verändern. Wir streichen deshalb sukzessive alle Knoten $B \neq B_a$ mit *Vorg* $(B) = \emptyset$. Durch die Elimination solchen *unerreichbaren Codes* wird der Code-Umfang reduziert und die Laufzeit nachfolgender Übersetzerphasen verringert. Die Ursache für die Entstehung unerreichbaren Codes liegt meist in der Berechnung der Ergebnisse von Verzweigungs- oder Schleifenbedingungen zur Übersetzungszeit, z. B. durch Konstantenfaltung. Liefert etwa in Abbildung 8.2-1 C_1 immer den Wert `false`, so können B_2 und die Verzweigung dahin eliminiert werden. Es ist deshalb zweckmäßig, nach der Konstantenfaltung den Ablaufgraphen zu aktualisieren.

Ebenso kann ein Grundblock B, der nur aus einem unbedingten Sprung besteht, eliminiert und der Sprung direkt in allen Blöcken aus *Vorg* (B) eingesetzt werden. Solche Sprünge auf Sprünge können z. B. bei der schematischen Zerlegung von Ablaufstrukturen oder durch Optimierungen, die Code eliminieren, entstehen.

8.2.2 Schleifenerkennung

Eine wichtige Klasse von wirkungsvollen Transformationen zielt auf die Verbesserung von Schleifen. Geht man von allgemeinen Ablaufgraphen aus, die aus niederen Ablaufstrukturen entstanden sind, so müssen darin zunächst Teilgraphen als Schleifen erkannt werden.

Dominanz

Um Schleifen zu beschreiben führen wir zunächst eine Relation über den Knoten des Ablaufgraphen ein: Wir sagen ein Knoten *a dominiert* einen Knoten *b*, *a dom b*, falls jeder Weg vom Anfangsknoten des Graphen zu *b* über *a* führt. Es gilt *a dom a* für alle Knoten.

Schleifenkopf

Damit ist eine Menge S von Knoten genau dann eine *Schleife*, wenn S einen eindeutigen Eingangsknoten h enthält, der alle Knoten in S dominiert, und wenn es von jedem Knoten in S einen Weg zu h gibt. h heißt auch *Kopf* der Schleife.

Rückwärtskanten

Jede Schleife wird durch eine *Rückwärtskante* charakterisiert. Dies sind Kanten $a \rightarrow h$ mit *h dom a*. Der Knoten h ist dann Kopf einer Schleife. Alle Schleifen eines Graphen bestimmt man nun, indem man zu einer Rückwärtskante $a \rightarrow h$ die Menge S so berechnet, daß gilt $a \in S$,$h \in S$ und *Vorg* $(b) \in S$, falls $b \in S$ und $b \neq h$. Abbildung 8.2-2 zeigt einen Ablaufgraphen mit seinen Schleifen.

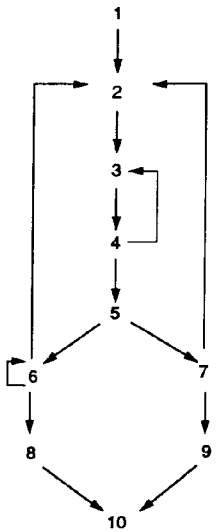
Für je zwei so berechnete Schleifen S_i , S_j gilt eine der folgenden Aussagen:

- S_i und S_j sind disjunkt (z. B. S_1 und S_4),
- S_i und S_j sind ineinander geschachtelt (z. B. S_1 in S_2) oder
- S_i und S_j haben denselben Kopf (z. B. S_2 und S_3).

Im letzten Fall ist es zweckmäßig, die Schleifen miteinander zu verschmelzen.

Irreduzible Graphen

In einem Ablaufgraphen, wie dem in Abbildung 8.2-3, können auch zyklische Teilgraphen existieren, die nicht die Schleifenbedingung erfüllen, da sie keinen eindeutigen Kopfknoten haben. Solch ein Graph



$4 \rightarrow 3$: $S_1 = \{3,4\}$
$6 \rightarrow 2$: $S_2 = \{2,3,4,5,6\}$
$7 \rightarrow 2$: $S_3 = \{2,3,4,5,7\}$
$6 \rightarrow 6$: $S_4 = \{6\}$

Abb. 8.2-2: Rückwärtskanten und Schleifen im Ablaufgraph.

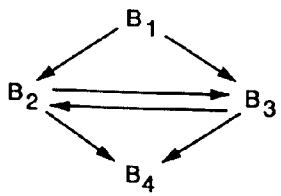


Abb. 8.2-3: Irreduzibler Ablaufgraph.

ist *irreduzibel*. Schleifenoptimierungen können in diesem Fall erst nach Umstrukturierung unter Verdoppelung von Knoten berechnet werden.

8.2.3 Code-Verschiebung aus Schleifen

Code-Verschiebung

Bei einigen Schleifenoptimierungen wird Code aus der Schleife herausgezogen oder neu konstruiert, der dann immer vor der Schleife

genau einmal ausgeführt werden soll. Solche Art der *Code-Verschiebung* erfordert strukturelle Veränderungen des Ablaufgraphen.

Soll Code unmittelbar vor einer Schleife plaziert werden, deren Kopf von mehreren Vorgängern erreicht wird, so bringen wir ihn in einem neuen Grundblock unter, der zwischen Schleifenkopf und seinen Vorgängern eingefügt wird. Der Knoten B' in Abbildung 8.2-4 ist ein solcher *Vorkopf*.

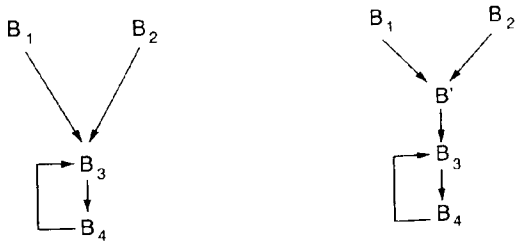


Abb. 8.2-4: Vorkopf einer Schleife.

Schleifen, die aus `while`- oder `for`-Konstrukten entstanden sind, werden im aktuellen Programmlauf abhängig von der Schleifenbedingung eventuell gar nicht ausgeführt. Verschiebt man wie oben beschrieben Code aus dem Rumpf in den Vorkopf, so können unerwünschte Effekte entstehen: Falls die Schleife nicht ausgeführt wird, verlängert sich die Laufzeit gegenüber dem nicht transformierten Programm. Bewirkt der verschobene Code Seiteneffekte (z. B. Überlauf), so treten diese unter Umständen nur im transformierten Programm auf. Die Wirkung des Programms ist dann unzulässig verändert worden. So ist z. B. eine Transformation von

```
while C do ... x/y ...
```

in

```
u := x/y; while C do ...u...
```

allgemein nicht zulässig. Wandeln wir die ursprüngliche Schleife zunächst unter Verdopplung der Bedingungen in ein `repeat`-Konstrukt um,

```
if C then repeat ... x/y ... until not C
```

so ist darauf die Transformation anwendbar. In Termen allgemeiner Ablaufgraphen ausgedrückt bedeutet dies: Aus einem Block *B* einer

Schleife darf nur dann Code vor die Schleife verschoben werden, wenn B auf jedem Weg durch die Schleife liegt. Diese Bedingung kann durch Umstrukturieren des Ablaufgraphen, wie oben gezeigt, erfüllt werden. Die Code-Verschiebung aus Schleifen wird in MORE79 formal beschrieben.

8.2.4 Datenflußgleichungen

Optimierungsinformation

Die Datenflußanalyse geht von Eigenschaften einzelner Grundblöcke (z. B. im Block berechnete Variablenwerte) aus, verknüpft diese für im Ablaufgraphen benachbarte Blöcke und liefert als Ergebnis präzisierte Optimierungsinformation zu den Grundblöcken. Die Datenflußinformation zu jedem Grundblock wird durch Mengen von Objekten beschrieben. Je nach Art des Datenflußproblems sind dies z. B. Mengen von Variablen, Ausdrücken oder Zuweisungen an Variablen. In der Implementierung von Datenflußalgorithmen stellen wir solche Mengen als Bitvektoren oder Listen dar.

Datenflußgleichung

Wir ordnen je eine Menge $In\ (B)$ dem Eingang des Blockes B und $Out\ (B)$ seinem Ausgang zu. Für jeden Block stellen wir eine Gleichung der folgenden Form auf:

$$(8.2\text{-}1)\quad Out\ (B) = Gen\ (B) \cup (In\ (B) - Kill\ (B)).$$

Sie definiert die Berechnung von $Out\ (B)$ abhängig von $In\ (B)$. Die Mengen $Gen\ (B)$ und $Kill\ (B)$ sind Konstrukte, die unabhängig vom Ablaufgraphen für jeden Block lokal bestimmt werden. $Gen\ (B)$ enthält die Informationsobjekte, die durch eine Ausführung von B erzeugt werden und damit immer auch in $Out\ (B)$ enthalten sind. $Kill\ (B)$ enthält die Objekte, die durch eine Ausführung von B ungültig werden, falls sie bei Eintritt in B gelten. (Man könnte die Gleichung auch mit dem Komplement von $Kill$ (B), häufig $Thru\ (B)$ genannt, formulieren, das dann mit $In\ (B)$ geschnitten wird.)

Die In-Menge jedes Blockes wird durch die Out-Mengen seiner Vorgänger bestimmt, beschrieben durch Gleichungen der Form:

$$(8.2\text{-}2)\quad In\ (B) = \underset{V=Vorg\ (B)}{\theta}\ Out\ (V)$$

Existenz- und Allprobleme

Dabei ist θ je nach Datenflußproblem der Vereinigungs- oder Durchschnittsoperator. Im ersten Fall ($\theta = \cup$) sprechen wir von einem *Existenzproblem*: $In\ (B)$ faßt die Eigenschaften zusammen, die am Ausgang eines beliebigen Vorgängers gelten. Im Fall $\theta = \cap$ liegt ein *Allproblem* vor: $In\ (B)$ beschreibt Eigenschaften, die für alle Vorgänger gelten. Durch Berechnung von Lösungswerten für die Variablen In und Out in einem so aufgestellten Gleichungssystem wird das zugehörige Datenflußproblem gelöst. Da die Ablaufgraphen im allgemeinen Schleifen

enthalten, können die Lösungen nicht durch einfache Substitution der Gleichungen berechnet werden.

Vorwärts- und Rückwärtsanalyse

In den obigen Gleichungsformen folgen die Abhängigkeiten zwischen den In- und Out-Mengen den Kanten des Ablaufgraphen. Wir sprechen deshalb bei so beschriebenen Datenflußproblemen von *Vorwärtsanalyse*. Insbesondere zur Bestimmung lebendiger Variablen ist ein Datenflußproblem mit entgegengesetzt gerichteten Abhängigkeiten durch *Rückwärtsanalyse* zu lösen. Die entsprechenden Gleichungen lauten dafür:

$$(8.2\text{-}3)\quad In\ (B) = Gen\ (B) \cup (Out\ (B) - Kill\ (B)),$$

$$(8.2\text{-}4)\quad Out\ (B) = \underset{N=Nachf\ (B)}{\theta}\ In\ (N).$$

Vorwärtsanalyse

Da die Probleme symmetrisch sind, beschränken wir uns im folgenden auf die Untersuchung der Vorwärtsanalyse. Als Beispiel für das Aufstellen eines Gleichungssystems betrachten wir das Datenflußproblem der *erreichenden Definitionen* (*reaching definitions*). Diese Information kann z. B. zur Konstantenweitergabe oder Elimination von Bereichsprüfungen verwendet werden (Abschnitt 8.1). Weitere Anwendungen werden wir in Abschnitt 8.3 zeigen.

Erreichende Definitionen

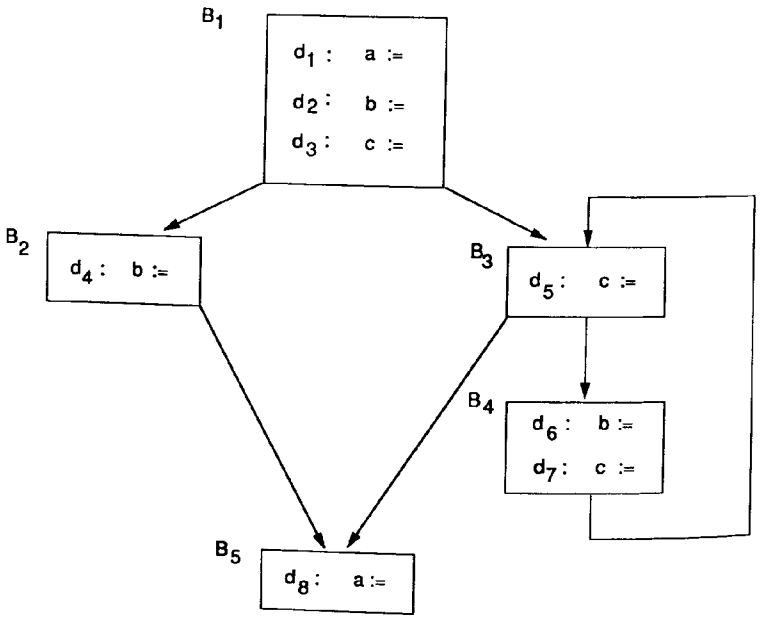
An eine Variable x wird an verschiedenen Programmstellen d_i ein Wert zugewiesen. Dies sind die Definitionen von x. Zu einem Block B sollen $In\ (B)$ und $Out\ (B)$ Mengen von Definitionen aller betrachteten Variablen angeben, die bei beliebigem Programmablauf möglicherweise am Eingang bzw. Ausgang von B gültig sind. Abbildung 8.2-5 zeigt einen Ablaufgraphen mit den Zuweisungen d_1 bis d_8 an die Variablen a, b, c. Die Mengen $Gen\ (B)$ und $Kill\ (B)$ haben hier folgende Bedeutung:

$Gen\ (B)$: alle Definitionen d_i in B, auf die in B keine Definition derselben Variable folgt,

$Kill\ (B)$: alle Definitionen d_j in Blöcken ungleich B an Variable, an die auch in B zugewiesen wird.

Existenzproblem

Mit dieser Definition beschreibt die Gleichung (8.2-1) den gewünschten Zusammenhang zwischen den In- und Out-Mengen: Am Ausgang von B sind die jeweils letzten Zuweisungen aus B gültig sowie diejenigen, die am Eingang von B gelten und an deren Variablen in B nicht zugewiesen wird. In die Gleichung (8.2-2) setzen wir für θ den Vereinigungsoperator ein, um auszudrücken, daß die Definitionen jedes Vorgängers zu denen der Nachfolger beitragen können. Die Menge $In\ (B_a)$ ist leer. Abbildung 8.2-5 gibt für das Beispiel die Mengen Gen und $Kill$ sowie eine Lösung für In und Out an.



	Gen	Kill	In	Out
B_1	d_1, d_2, d_3	d_4, d_5, d_6, d_7, d_8	ϕ	d_1, d_2, d_3
B_2	d_4	d_2, d_6	d_1, d_2, d_3	d_1, d_3, d_4
B_3	d_5	d_3, d_7	d_1, d_2, d_3, d_6, d_7	d_1, d_2, d_5, d_6
B_4	d_6, d_7	d_2, d_3, d_4, d_5	d_1, d_2, d_5, d_6	d_1, d_6, d_7
B_5	d_8	d_1	$d_1, d_2, d_3, d_4, d_5, d_6$	$d_2, d_3, d_4, d_5, d_6, d_8$

Abb. 8.2-5: Mengen erreichender Definitionen.

Minimale Lösung

Im allgemeinen gibt es mehrere Lösungen eines Systems von Datenflußgleichungen. Im obigen Beispiel für erreichende Definitionen ist die minimale Lösung angegeben. Fügt man z. B. in den Mengen *In* (B_3), *Out* (B_3), *In* (B_4), *Out* (B_4) und *In* (B_5) jeweils das Element d_8 hinzu, so ist das Gleichungssystem ebenfalls erfüllt. Diese nicht minimale Lösung entspricht jedoch nicht der gesuchten Optimierungsinformation.

Allproblem, maximale Lösung

Das Datenflußproblem der erreichenden Definitionen ist ein Existenzproblem. Es wird festgestellt, ob ein Weg von einer Variablendefinition zu einer Programmstelle existiert. Solche Probleme werden mit $\theta = \cup$ in der Gleichung (8.2-2) bzw. (8.2-4) formuliert. Unter den möglichen Lösungen wird die minimale gesucht. Im Gegensatz dazu muß in Allproblemen eine Eigenschaft auf allen Wegen zu einer Programmstelle gelten.

Hier ist dann $\theta = \cap$ und die maximale Lösung wird gesucht. Kennedy gibt in KENN81 eine Übersicht über Algorithmen zur Lösung von Datenflußproblemen.

8.2.5 Iteratives Lösungsverfahren

Iteration, Anfangswert

Ein einfaches Verfahren zur Lösung von Datenflußgleichungen basiert auf folgendem Iterationsprinzip: Man wähle geeignete Anfangswerte für die Mengen *In* (*B*) und *Out* (*B*) und wiederhole die Berechnung aller *In*- und *Out*-Mengen solange, bis sich keine mehr ändert. Suchen wir in der Vorwärtsanalyse eine minimale Lösung (Existenzproblem), so wählen wir als Anfangswerte *In* (*B*) = $\emptyset$ und *Out* (*B*) = *Gen* (*B*) für alle *B*. Damit ergibt sich der Algorithmus in Abbildung 8.2-6. Im Falle eines Allproblems wird *In* (*B*) mit der vollen Menge *U* und *Out* (*B*) mit *Gen* (*B*) $\cup$ (*U* - *Kill* (*B*)) initialisiert und in der inneren `for`-Schleife der Operator $\cap$ eingesetzt.

```
for all B do
begin In[B] := {}; Out[B] := Gen[B] end;

repeat stabil := true;
       for all B <> Ba do
       begin
             for all V in Vorg[B] do
                 In[B] := In[B] + Out[V];
             oldout := Out[B];
             Out[B] := Gen[B] + (In[B] - Kill[B]);
             stabil := stabil and Out[B] = oldout
       end
until stabil
```

Abb. 8.2-6: Iterative Lösung der Datenflußgleichungen.

Komplexität

Der Algorithmus terminiert, da in keinem Schritt ein Element aus einer der Mengen entfernt wird. Ist *n* die Anzahl der Knoten des Ablaufgraphen, so ist die theoretische Komplexität des Algorithmus $O(n^3)$ bzw. $O(n^2)$, falls für die maximale Zahl der Vorgänger eines Knotens eine im Vergleich mit *n* kleine Konstante angenommen wird. Die tatsächliche Anzahl der Durchläufe der `repeat`-Schleife wird wesentlich durch die Reihenfolge bestimmt, in der die Blöcke in der äußeren der beiden geschachtelten `for`-Schleifen betrachtet werden. Es ist günstig, wenn möglichst viele Änderungen der Mengen in einem einzigen Durchgang berechnet werden. Dies ist dann der Fall, wenn man einen Knoten erst nach seinen Vorgängern betrachtet - sieht man von Rückwärtskanten in Schleifen ab. Eine solche Anordnung der Knoten erhält

man, wenn für den Ablaufgraphen ausgehend vom Anfangsknoten B_a ein Baum berechnet wird, der alle Knoten enthält (*spannender Baum*) und darin die Vorgänger vor den Nachfolgern angeordnet werden. Legt man dem Algorithmus diese Anordnung der Blöcke zugrunde, so wird die Zahl der Durchläufe der `repeat`-Schleife nur noch durch die Rückwärtskanten der Zyklen im Ablaufgraphen bestimmt.

8.2.6 Hierarchische Lösungsverfahren

Hierarchisches Verfahren

Die Laufzeitkomplexität des iterativen Algorithmus zur Lösung von Datenflußgleichungen kann reduziert werden, wenn man ausnutzt, daß den Ablaufgraphen meist eine hierarchische Struktur zugrunde liegt. Sie kann durch die syntaktische Struktur des Programmes vorgegeben sein oder im Nachhinein für einen Ablaufgraphen ermittelt werden. Wie im vorigen Abschnitt betrachten wir auch hier nur die Vorwärtsanalyse für Existenzprobleme.

p_1 :	P ::= S	S.In := ∅
p_2 :	S ::= B	S.Gen := B.Gen S.Kill := B.Kill S.Out := B.Gen ∪ (S.In - B.Kill)
p_3 :	S_1 ::= S_2; S_3	S_1.Gen := S_3.Gen ∪ (S_2.Gen - S_3.Kill) S_1.Kill := S_3.Kill ∪ (S_2.Kill - S_3.Gen) S_2.In := S_1.In S_3.In := S_2.Out S_1.Out := S_3.Out
p_4 :	S_1 ::= S_2 **alt** S_3	S_1.Gen := S_2.Gen ∪ S_3.Gen S_1.Kill := S_2.Kill ∩ S_3.Kill S_2.In := S_1.In S_3.In := S_1.In S_1.Out := S_2.Out ∪ S_3.Out
p_5 :	S_1 ::= **loop** S_2	S_1.Gen := S_2.Gen S_1.Kill := S_2.Kill S_2.In := S_1.In ∪ S_2.Gen S_1.Out := S_2.Out

Abb. 8.2-7: Datenflußattributierung.

Attributierung

Wir nehmen zunächst an, daß die Ablaufstruktur eines Programmes durch die in Abbildung 8.2-7 enthaltene, vereinfachte Syntax beschrieben wird. Die Strukturen sind aus Sequenzen (p_3), Verzweigungen

(p_4), Schleifen (p_5) und Grundblöcken (p_2) zusammengesetzt. Marken und Sprünge sind ausgeschlossen. Die Ablaufstruktur eines Programms wird so durch einen abstrakten Strukturbaum zu obiger Grammatik beschrieben, dessen Blätter die Grundblöcke sind. Wir können nun die Berechnung der Datenflußmengen als Attributierung solcher Bäume wie in Abbildung 8.2-7 spezifizieren. Dabei ordnen wir allen inneren Knoten (*S*) die abgeleiteten Attribute *Gen, Kill, Out* und das erworbene Attribut *In* mit ihrer üblichen Bedeutung zu. Im Gegensatz zum iterativen Verfahren werden hier zusammengesetzte Konstrukte zu einem hierarchisch übergeordneten Knoten zusammengefaßt und auch für diesen die Mengenattribute berechnet.

Die Attributregeln für Sequenzen und Verzweigungen (p_3 , p_4) sind unmittelbar einsichtig. In p_2 wird die Gleichung (8.2-1) mit den konstanten Mengen *B.Gen* und *B.Kill* übernommen. p_1 gibt die Initialisierung für den Anfangsknoten an.

Eine besondere Überlegung erfordert die Attributierung der Schleife in p_5 : Überträgt man die Gleichung (8.2-2) unmittelbar auf diesen Fall, so hätte die Attributregel für S_2.*In* lauten müssen:

$$S_2.In := S_1.In \cup S_2.Out$$

Damit wäre die Attributierung zyklisch. Wir können hier jedoch die Mengen S_2. *Gen* und S_2. *Kill* als berechnet voraussetzen und S_2. *Out* gemäß (8.2-2) substituieren:

$$S_2.In := S_1.In \cup (S_2.Gen \cup (S_2.In - S_2.Kill))$$

Diese Attributregel ist direkt rekursiv. Entwickelt man die Rekursion mit ∅ als Anfangswert für S_2.*In*, so ergibt sich die Gleichung:

$$S_2.In := S_1.In \cup S_2.Gen$$

Die Attributierung ist damit zyklenfrei.

Aus den Attributabhängigkeiten erkennt man unmittelbar, daß jeder Strukturbaum in zwei Durchläufen attributiert werden kann. *Gen* und *Kill* werden im ersten Durchlauf aufwärts, *In* und *Out* im zweiten Durchlauf abwärts berechnet. D.h., im ersten Durchlauf werden die lokalen Eigenschaften der Blöcke zu größeren hierarchischen Strukturen propagiert. Im zweiten Durchlauf werden die *In*- und *Out*-Mengen schrittweise zu den Grundblöcken hin verfeinert. Die Laufzeit dieses Verfahrens ist offensichtlich proportional zur Zahl der Knoten im Strukturbaum .

Die angegebene Attributierung kann systematisch so verändert werden, daß sie für Allprobleme anwendbar ist. Dazu werden in den Attributierungen der Schleife und der Verzweigung alle Mengenoperatoren durch den komplementären Operator ersetzt, und in p_1 wird mit der vollen Menge initialisiert. Die Rückwärtsanalyse läßt sich ebenso systematisch beschreiben.

Sprünge und Prozeduren

Prinzipiell läßt sich dieses Attributierungsverfahren auch auf Strukturen mit Marken und Sprüngen sowie Prozeduren und Aufrufen ausdehnen. Die Optimierungsinformation muß dann zwischen einer Marke und allen Sprüngen darauf bzw. zwischen Prozeduranfang und -ende und allen ihren Aufrufen wie im entsprechenden Ablaufgraph weitergegeben werden. Damit erhält man dann allerdings eine Attributierung mit zyklischen Abhängigkeiten. Da hier jedoch die Konvergenz der Mengen gesichert ist, kann die Lösung durch iterierte Attributberechnung mit einem speziellen Auswerteralgorithmus bestimmt werden.

Intervallanalyse

Auch für allgemeine Ablaufgraphen können hierarchische Lösungsverfahren angewandt werden. Das bekannteste, die *Intervallanalyse*, soll hier kurz skizziert werden. Zur Vertiefung sei auf MUCH81 verwiesen. Im Gegensatz zu obigem Verfahren muß hier zunächst dem Ablaufgraphen eine hierarchische Struktur aufgeprägt werden. Man überdeckt ihn vollständig durch eine Menge von Teilgraphen, sogenannte *Intervalle*. Jedes Intervall hat einen eindeutigen Eingangsknoten, der alle Knoten des Intervalls dominiert. Auf der nächsten Hierarchieebene wird jedes Intervall zu einem Knoten zusammengefaßt, der in dem so vereinfachten Graphen die Datenflußeigenschaften des Intervalls abstrahiert. Dieses Verfahren wird hierarchisch fortgesetzt bis der Graph zu einem einzigen Wurzelknoten reduziert ist. Für die große Klasse der reduziblen Graphen ist dies möglich.

Auf der so gewonnenen hierarchischen Struktur wird das Prinzip des oben beschriebenen syntaktischen Verfahrens angewandt. Im Aufwärtsdurchgang werden die Mengen *Gen* und *Kill* vergröbert, im Abwärtsdurchgang *In* und *Out* verfeinert. Allerdings sind die Berechnungen auf jeder Hierarchie- ebene komplexer, da die Knoten beliebig viele Nachfolger haben können. Das Verfahren ist deshalb nicht linear in der Zahl der Knoten.

8.3 Anwendungen der Datenflußinformation

In diesem Abschnitt stellen wir verbessernde Transformationen vor, die auf der durch Datenflußanalyse berechneten Optimierungsinformation basieren. Dabei legen wir Lösungen unterschiedlicher Datenflußprobleme zugrunde. Weiter zeigen wir Transformationen, die speziell auf Schleifen angewandt werden und deshalb zur Reduktion der

Ausführungszeit besonders wirksam sind. Schließlich beziehen wir Variablen mit mehreren Zugriffswegen und Aufrufe von Prozeduren in die Analyse ein.

8.3.1 Erreichende Definitionen

Erreichende Definitionen

Das Datenflußproblem der erreichenden Definitionen bestimmt für jeden Grundblock die Menge der in möglichen Abläufen jeweils letzten Variablendefinitionen. Wir haben dieses Problem in Abschnitt 8.2.4 als Beispiel für die Vorwärtsanalyse verwendet. Mit der so berechneten Information kann man jeder Anwendung einer Variablen in einem Block ihre möglichen Definitionsstellen zuordnen. Da diese Information als Voraussetzung für mehrere verschiedene Optimierungsmaßnahmen verwendet wird, ist es zweckmäßig, sie durch eine spezielle Datenstruktur zu repräsentieren: Zu jeder Anwendungsstelle legt man eine Liste von Verweisen auf die Definitionsstellen an, sogenannte *Definitionsketten* (*use-def chains*). Im folgenden zeigen wir einige der hierauf basierenden Transformationen auf.

Anwendung von Definitionsketten

```
                                          s1 := i * 3 + 1;
                                          s2 := i * 15 + 5;
                                          s3 := i * 105 + 35;

                    Tripel:

repeat                                    repeat
  k := i * 3+1;     (i, 3, 1)               k  := s1;
  j := k * 5;       (i, 3*5, 5)             j  := s2;
  x := a[j*7];      (i, 3*5*7, 5*7)         x  := a[s3];
  i := i+2          (i, 1, 2)               i  := i+2;
until C                                   s1 := s1 + 6;
                                          s2 := s2 + 30;
                                          s3 := s3 + 210
                                        until C
```

Abb. 8.3-1: Transformation von Induktionsvariablen.

Konstantenweitergabe. Sind alle Definitionen in einer solchen Kette Zuweisungen des gleichen konstanten Wertes, so ersetzt man die Anwendung der Variablen durch diesen Wert. Dadurch kann möglicherweise Konstantenfaltung auf den Ausdruck angewandt werden und eine weitere konstante Zuweisung entstehen.

Bereichsprüfungen. Ist die obere oder untere Grenze der in der Definitionskette zugewiesenen Werte bekannt, so berechnet man daraus die minimalen unteren und maximalen oberen Grenzen und ordnet diese der Anwendungsstelle zu. Damit können Laufzeitprüfungen als überflüssig erkannt und Grenzen für andere Variable ermittelt werden.

Schleifenoptimierung. Mit Hilfe der Definitionsketten kann man schleifeninvariante Berechnungen und Induktionsvariable erkennen. Die Transformationen hierzu werden in Abschnitt 8.3.4 erläutert.

Kopierzuweisungen

Kopierzuweisungen. Bei einigen Transformationen werden neue Zuweisungen der Form $x := y$ erzeugt. Einige davon können wieder eliminiert werden, indem man Anwendungen von x durch y ersetzt. Zu einer Definition $s: x := y$ prüfen wir, ob alle Anwendungen $u: z := A_x$ von x, die s erreicht, ausschließlich s in ihrer Definitionskette haben. Wenn außerdem auf keinem Pfad von s zu einem u eine Zuweisung an y erfolgt, kann s eliminiert und x in jedem A_x durch y ersetzt werden. Die zweite Bedingung prüfen wir, indem wir ein neues Datenflußproblem lösen, das dem der erreichenden Definitionen sehr ähnlich ist. Seine Lösung gibt mit $In\ (B)$ genau die Menge von Kopierzuweisungen der Form $s: x := y$ an, für die am Anfang von B obige Bedingung gilt. *Gen* und *Kill* werden wie folgt bestimmt:

$s \in Gen\ (B)$ wenn s zu B gehört und in B keine Zuweisung an y folgt.
$s \in Kill\ (B)$ wenn s nicht zu B gehört und B eine Zuweisung an y enthält.

Das Problem ist offensichtlich ebenfalls ein Vorwärtsproblem, aber anders als das der erreichenden Definitionen ein Allproblem mit $\theta = \cap$.

8.3.2 Globale gemeinsame Ausdrücke

Verfügbare Ausdrücke

Ausdrücke, die in verschiedenen Grundblöcken gleich auftreten, brauchen nicht in jedem erneut berechnet zu werden. Man weist das Ergebnis ihrer Berechnung an eine neu eingeführte Hilfsvariable zu und substituiert diese an anderen Stellen für den Ausdruck. Mit dem folgenden Verfahren bestimmen wir, welche Berechnungen ersetzt werden können und wo für verbleibende Berechnungen Zuweisungen eingefügt werden müssen.

Wir formulieren dazu ein Datenflußproblem, dessen Lösung zu jedem Grundblock B mit $In\ (B)$ die Menge der am Eingang *verfügbaren Ausdrücke* angibt. Sei A z. B. der Ausdruck $x * y$, dann ist A am Eingang von Block B verfügbar, falls A auf allen Wegen zu B berechnet wird, ohne daß danach x oder y neu berechnet werden. Es handelt sich hier also um ein Allproblem ($\theta = \cap$), das durch Vorwärtsanalyse zu lösen ist. Die Mengen $Gen\ (B)$ und $Kill\ (B)$ werden zu jedem Block wie folgt bestimmt:

$A \in Gen\ (B)$, falls es in B eine Berechnung von A gibt, auf die keine Zuweisung an eine der Variablen von A folgt.
$A \in Kill\ (B)$, falls es in B Zuweisungen an Variablen von A gibt, auf die keine Berechnung von A folgt.

Diese Regeln machen deutlich, daß hier Ausdrücke unter struktureller Gleichheit identifiziert werden. Verschiedene Auftreten von A können durchaus unterschiedliche Werte liefern.

Haben wir eine Lösung des Datenflußproblems, so wählen wir Anweisungen $s: z := A$ in Blöcken B_s mit $A \in In\ (B_s)$. Außerdem darf s keine Zuweisung an Variablen von A vorangehen. s wird durch $z := u$ ersetzt, wobei u eine für A neu eingeführte Hilfsvariable ist. Um die Blöcke B_t zu finden, in denen u berechnet werden muß, verfolgen wir ausgehend von B_s alle Wege zurück bis jeweils zum ersten Block mit $A \in Gen\ (B_t)$ und $B_t \neq B_s$. In jedem B_t ersetzen wir das Auftreten von A in $w := A$ durch $u := A\ ;\ w := u$.

Es muß angemerkt werden, daß diese Transformation den Code unter ungünstigen Umständen nicht verbessern oder sogar verschlechtern kann. Dies ist z. B. der Fall, wenn innerhalb einer Schleife zusätzliche Zuweisungen für A erzeugt werden, um die Berechnung von A nach der Schleife zu ersparen. Man wendet die Transformation deshalb nicht für alle Anweisungen s an.

8.3.3 Lebendige Variablen

Lebendige Variablen

In Abschnitt 8.1.4 wurden Zuweisungen an Variablen eliminiert, die nachfolgend nicht mehr verwendet werden. Dort haben wir für einzeln betrachtete Grundblöcke angenommen, daß am Ende des Blockes alle auftretenden Variablen lebendig sind. Die Annahme kann mit der Datenflußanalyse präzisiert werden. In diesem Fall wird die Datenflußinformation entgegen der Ablaufrichtung berechnet. Wir verwenden deshalb die Gleichungsformen 8.2-3 und 8.2-4 für die Rückwärtsanalyse. Die *Gen*- und *Kill*-Mengen haben hier folgende Bedeutung:

$Gen\ (B)$ Menge der Variablen, die in B benutzt, aber nicht vor ihrer Benutzung in B definiert werden.
$Kill\ (B)$ Menge der Variablen, die in B definiert, aber nicht vor ihrer Definition benutzt werden.

Ist eine Variable nur in einigen Nachfolgern eines Blockes B lebendig, so müssen wir pessimistisch annehmen, daß sie am Ende von B lebendig ist. Es liegt also ein Existenzproblem vor, und wir setzen $\theta = \cup$. Im iterativen Lösungsalgorithmus suchen wir eine minimale Lösung mit den Anfangswerten $Out\ (B) = \emptyset$ und $In\ (B) = Gen\ (B)$.

8.3.4 Redundante Berechnungen in Schleifen

Schleifeninvariante Berechnungen

Mit Hilfe von Datenflußinformation kann man in einfacher Weise solche Berechnungen erkennen, die *schleifeninvariant* sind, d. h. in jedem Durchlauf das gleiche Ergebnis liefern. Unter gewissen Bedingungen können solche Berechnungen vor die Schleife verlagert werden.

Sei $x + y$ ein Ausdruck A, der in einem Grundblock B berechnet wird. B gehöre zur Schleife S. Liegen alle Definitionen von x und y, die A erreichen, außerhalb von S, so ist A in S invariant. Die Eigenschaft stellt man durch Untersuchen der Definitionsketten fest. Man kann nun eine neue Hilfsvariable u einführen, im Vorkopf von S $u := x + y$ berechnen und A in B durch u ersetzen. Dabei entstehende Kopierzuweisungen werden ggf. durch entsprechende Optimierungen wieder entfernt.

Solch eine Transformation ist jedoch nur dann sicher, wenn B auf jedem Weg durch die Schleife ausgeführt wird. Gegenfalls muß der Ablaufgraph dazu wie in Abschnitt 8.2.3 beschrieben, transformiert werden.

Induktionsvariablen

Eine Variable i ist *Induktionsvariable* einer Schleife S, wenn sie bei jedem Schleifendurchgang um einen konstanten Wert erhöht oder erniedrigt wird. Innerhalb von S können Berechnungen, die von Induktionsvariablen abhängen, z. B. Reihungsindizierungen, vereinfacht werden. Außer den Zählvariablen in `for`-Schleifen kann man weitere Variablen auch in anderen Schleifenformen als Induktionsvariable erkennen. Wir setzen dazu voraus, daß Definitionsketten und konstante und schleifenvariante Berechnungen bestimmt sind.

Zunächst ermitteln wir alle *direkten Induktionsvariablen*. Sie haben in S genau eine Definition der Form p: $i := i \pm c$, wobei c konstant ist und p auf jedem Weg durch die Schleife ausgeführt wird. Jeder Ausdruck der Form $j^* \; a \pm b$ ist eine lineare Funktion in der Induktionsvariablen j, wenn a und b konstant sind. Eine Variable, die durch solch einen Ausdruck berechnet wird, ist eine *indirekte Induktionsvariable*.Wir ordnen jeder Induktionsvariablen und jedem solchen linearen Ausdruck ein Tripel (i, a, b) zu, der den Wert als Funktion einer direkten Induktionsvariablen beschreibt: $i*a+b$. Mit Hilfe der Tripel propagieren wir die Induktionsvariableninformation durch den Schleifenrumpf. Abbildung 8.3-1 zeigt eine Schleife mit den berechneten Tripeln und ihre Transformation. Alle diese Berechnungen müssen in jedem Schleifendurchgang ausgeführt werden.

Wir führen nun zu jedem Tripel (v_i, a_i, b_i) eine neue Hilfsvariable s_i ein und ersetzen den linearen Ausdruck durch s_i. Im Vorkopf der Schleife wird jedes s_i initialisiert durch $s_i := v_i * a_i + b_i$.

In der Schleife fügen wir unmittelbar hinter der einzigen Zuweisung für die direkte Induktionsvariable $v_i := v_i + c$ die Zuweisung $s_i := s_i + c * a_i$ ein. Auf diese Weise haben wir Multiplikationen innerhalb der Schleife durch Additionen ersetzt.

Lineare Adreßfortschaltung

Ein wichtiger Anwendungsfall für Transformationen auf der Basis von Induktionsvariablen ist die Reihungsindizierung in Schleifen. Hier können Multiplikationen mit Spannen und Elementlängen durch Additionen ersetzt werden. Diese Maßnahme ist auch unter dem Namen *lineare Adreßfortschaltung* bekannt. Ebenso ist die Erkennung von Induktionsvariablen wesentliche Voraussetzung für die Zusammenfassung

von Schleifenanweisungen zu komplexen Operationen mit Vektoren, die auf Vektorrechnern in Hardware realisiert sind.

8.3.5 Aliasnamen

Aliasnamen

In den vorangehenden Abschnitten haben wir einige einschränkende Annahmen unterstellt, um die Darstellung der Optimierungsverfahren zu vereinfachen. Wir haben die Existenz von Prozeduraufrufen zunächst ignoriert und unterstellt, daß Variablen immer eindeutig identifizierbar sind. Hier sollen nun Techniken der *interprozeduralen* Analyse und der Behandlung von verschiedenen Zugriffswegen zu gleichen Variablen, sogenannten *Aliasnamen*, aufgezeigt werden.

Referenzvariablen

In einem Programm kann eine Variable als Speicherobjekt außer über ihren deklarierten Bezeichner auch über andere Zugriffswege erreicht werden. Solche Aliasnamen entstehen z. B. durch die Verwendung von Referenzparametern und Referenzvariablen, deren Werte Stellen von Variablen sind. Kommen diese Referenzobjekte in Ausdrücken oder Zuweisungen vor, so ist nicht unmittelbar feststellbar, welche Variable betroffen ist. Da man den Wert von Referenzobjekten im allgemeinen nicht zur Übersetzungszeit eindeutig feststellen kann, ordnet man dem Auftreten eines Referenzobjektes eine pessimistisch berechnete Menge von Variablen zu, auf die es sich beziehen könnte. Die Analyse für Referenzparameter zeigen wir weiter unten im Zusammenhang mit Prozeduren. Zur Berechnung der möglichen Werte von Referenzvariablen löst man das folgende Datenflußproblem.

Jedem Grundblock B ordnen wir Mengen $In\,(B)$ und $Out\,(B)$ von Paaren (r, v) zu, die jeweils einen möglichen Wert v der Referenzvariablen r repräsentieren. Die Abhängigkeit zwischen den In- und Out-Mengen wird durch eine Funktion für jeden Block beschrieben:

$$Out\,(B) = Trans_B\,(In\,(B))$$

Die Funktion $Trans_B$ berechnet man durch sukzessive Analyse der Anweisungsfolge von B: Eine Zuweisung der Form $r := Stelle\,(a)$ entfernt alle Paare für r aus der Vorbedingung und fügt das Paar (r, a) ein. Eine Zuweisung zwischen Referenzvariablen $r := q$ entfernt alle Paare für r und fügt Paare (r, a) ein, falls es ein (q, a) in der Vorbedingung gibt. Gleichungen der Form

$$In\,(B) = \bigcup_{v \in Vorg\,(B)} Out\,(B)$$

vervollständigen das Gleichungssystem. Es wird dann mit minimalen Initialwerten z. B. iterativ gelöst. Anders als in bisher betrachteten Da-

tenflußproblemen kann hier die berechnete Information (Menge von Paaren) nicht durch Bitvektoren dargestellt werden.

Datenflußprobleme pessimistisch aufstellen

Auch bei der Aufstellung von Datenflußproblemen, wie in Abschnitt 8.2 beschrieben, muß in Gegenwart von Aliasnamen berücksichtigt werden, daß Bezüge auf Variablen nicht immer eindeutig sind. Berechnet man erreichende Definitionen, so tragen alle möglichen Ziele einer Zuweisung im Block B an eine Referenzvariable zu $Gen\ (B)$ bei, nicht aber zu $Kill\ (B)$, da sie nicht mit Sicherheit in B ausgeführt werden. Berechnet man verfügbare Ausdrücke, so muß angenommen werden, daß jede der möglichen Zuweisungen ausgeführt werden kann und dadurch Ausdrücke invalidiert werden. Bei der Analyse der Lebendigkeit von Variablen tragen alle möglichen Variablen einer Anwendung einer Referenzvariable zu $Gen\ (B)$ bei, die möglichen Zuweisungsziele jedoch nicht zu $Kill\ (B)$.

Auch zusammengesetzte Variablen können Aliasprobleme verursachen: Die Identität von Reihungselementen kann im allgemeinen nicht festgestellt werden, falls Indizierungsausdrücke nicht zur Übersetzungszeit ausgewertet werden. Man betrachtet daher Reihungen als ganzheitliche Variable. Eine Zuweisung an ein Element wird als Zuweisung an die Variable aufgefaßt. Auch für Verbunde kann es zweckmäßig sein, sie nur ganzheitlich zu betrachten, falls Zuweisungen sowohl an einzelne Komponenten als auch an den gesamten Verbund möglich sind.

8.3.6 Prozeduren

Seiteneffekte von Aufrufen

Der Aufruf einer Prozedur verursacht im allgemeinen (neben der Berechnung eines Ergebnisses im Falle einer Funktion) Seiteneffekte, die bei der Berechnung der Datenflußinformation berücksichtigt werden müssen. Man könnte den Ablaufgraphen des Prozedurrumpfes an den Aufrufstellen einsetzen und die bisher vorgestellten Analysemethoden anwenden. Dieses Vorgehen ist jedoch nur für sehr kleine, selten aufgerufene Prozeduren praktikabel, da es sonst zu unvertretbar hohem Analyseaufwand führt. Wir erweitern deshalb die möglichen Anweisungsformen in Grundblöcken um Prozeduraufrufe und bestimmen deren Auswirkung auf die Datenflußinformation des Blockes. Dabei beschränken wir uns auf die Bestimmung von Variablendefinitionen und -benutzungen. Die Feststellung gemeinsamer Ausdrücke über Prozedurgrenzen hinweg ist meist wenig ergiebig und zu aufwendig.

Menge veränderbarer Variablen

Seiteneffekte auf die Umgebung der Aufrufstelle werden durch Zuweisungen an globale Variablen oder Referenzparameter im Prozedurrumpf bewirkt. Wir suchen deshalb zu jeder Prozedur p eine Menge von Variablen $Mod\ (p)$, deren Wert durch einen Aufruf von p möglicherweise verändert wird. Dabei müssen auch alle von p direkt oder indirekt aufgerufenen Prozeduren berücksichtigt werden.

$Mod\ (p)$ kann anhand der Aufrufbeziehung rekursiv definiert werden. Eine Variable v ist in $Mod\ (p)$, falls eine der folgenden Bedingungen gilt:
1) v ist global zu p oder formaler Referenzparameter von p und wird in p explizit zugewiesen.
2) p enthält einen Aufruf von q, v ist global zu q und $v \in Mod\ (q)$.
3) p enthält einen Aufruf von q mit v als aktuellem Parameter zum formalen Referenzparameter f von q und $f \in Mod\ (q)$.

Nach diesen Regeln kann man alle $Mod\ (p)$ iterativ bestimmen: Ausgehend von Mengen mit Elementen, die (1) erfüllen, werden die $Mod\ (p)$ solange gemäß (2) und (3) vergrößert, bis die Lösung stabil ist. Die Iteration wird beschleunigt, wenn man die Prozeduren in einer Reihenfolge entgegen ihrer Aufrufbeziehung betrachtet, abgesehen von Zyklen darin (vgl. Abschnitt 8.2.5).

Pessimistische Datenfluß-information

Die Mengen $Mod\ (p)$ werden dann in die Bestimmung der Datenflußinformation für Grundblöcke einbezogen. Der Aufruf einer Prozedur p im Block B wird als Menge möglicher Zuweisungen an die Variablen in $Mod\ (p)$ aufgefaßt. Es gelten die gleichen Konsequenzen für die Berechnung von $Gen\ (B)$ und $Kill\ (B)$ wie in Abschnitt 8.3.5 für Zuweisungen an Referenzvariablen.

Die Untersuchung der Lebendigkeit von Variablen kann nach dem gleichen Verfahren über Prozedurgrenzen hinweg ausgedehnt werden. Anstelle von Zuweisungen im Rumpf einer Prozedur p betrachtet man dann Benutzungen globaler Variablen und formaler Referenzparameter und berechnet $Use\ (p)$ analog zu $Mod\ (p)$.

Prozeduren als Parameter

Erlaubt die zu übersetzende Programmiersprache auch Prozeduren als Parameter von Prozeduren, so kann die Aufrufbeziehung nicht unmittelbar aus der Programmstruktur entnommen werden. Nehmen wir an, daß alle formalen Parameter verschiedener Prozeduren unterschiedlich benannt (bzw. umbenannt) sind. Dann berechnet man zu jedem formalen Parameter f einer Prozedur p, der in p unmittelbar aufgerufen wird, eine Menge $Akt\ (f)$ von deklarierten und formalen Prozeduren, so daß für $r \in Akt\ (f)$ eine der folgenden Bedingungen gilt:
- Es gibt einen Aufruf von p mit aktuellem Parameter r für f.
- g ist ein formaler Parameter mit $g \in Akt\ (f)$ und $r \in Akt\ (g)$.

Dann ist die für f tatsächlich aufgerufene Prozedur unter den deklarierten Prozeduren in $Akt\ (f)$. Diese müssen wir alle in die Aufrufbeziehung übernehmen, um sichere Annahmen zu machen.

Prozedurvariablen

Sind auch Prozedurvariablen in der Sprache zugelassen, so bezieht man sie in obige Analyse mit ein und versucht durch Rückverfolgen der Definitionsketten ihre möglichen Werte einzugrenzen. Gelingt dies nicht,

muß man pessimistisch annehmen, daß ihr Wert eine beliebige, im Kontext gültige, deklarierte oder formale Prozedur passenden Typs ist.

Separate Übersetzung

Sind Quellprogramme in mehrere Übersetzungseinheiten gegliedert, die separat übersetzt werden, dann stehen im allgemeinen keine Informationen über Variablendefinitionen und -benutzungen sowie über Aufrufe in externen Prozeduren zur Verfügung. Man kann aber aus einer expliziten Schnittstellenbeschreibung oder implizit aus dem Quelltext die Menge der in anderen Übersetzungseinheiten sichtbaren Variablen und Prozeduren feststellen. Man muß dann pessimistisch annehmen, daß ein Aufruf einer externen Prozedur alle diese Objekte definiert, benutzt bzw. aufruft. Für den gerade übersetzten Modul kann solche Information präzisiert werden. Legt man diese Information zusätzlich zum Übersetzungsergebnis ab, so kann sie bei nachfolgenden Übersetzungen anderer Module verwendet werden.

8.4 Nachoptimierung

Die Verfahren zur *Nachoptimierung* wenden verbessernde Transformationen auf der Ebene des erzeugten Assemblercodes an. Sie sind damit vollständig abhängig vom Instruktionssatz des Zielprozessors und unabhängig von Quell- und Zwischensprache. Der Übersetzermodul zur Nachoptimierung wird als Filter zwischen der Code-Auswahl und der Assemblierung eingefügt. Er puffert eine kurze Folge von meist zwei bis drei Instruktionen, auf denen er die Transformationen anwendet. Der englische Name *peephole optimization* charakterisiert das Verfahren treffend. In DAVI80 und TANE82 werden systematische Verfahren zur Nachoptimierung beschrieben.

Wirksamkeit der Nachoptimierung

Auch wenn in vorangehenden Übersetzerphasen lokale und globale Optimierungen sowie optimierende Code-Auswahl angewandt werden, kann eine Nachoptimierung weitere wirksame Verbesserungen erzielen. Ihre Stärken entfaltet sie dann im wesentlichen in zwei Klassen von Situationen: Werden die Entscheidungen zur Code-Auswahl jeweils für einzelne Teilbäume unabhängig getroffen, dann kann der Code an den Grenzen zwischen Teilbäumen redundant sein. Außerdem kann die Nachoptimierung die Anwendbarkeit von Spezialinstruktionen feststellen, deren Berücksichtigung bei der Code-Auswahl zu aufwendig ist.

Darüber hinaus kann die Nachoptimierung auch einige der Transformationen anwenden, die wir in den vorigen Abschnitten besprochen haben, z. B. Konstantenfaltung, algebraische Umformungen, Elimination von unerreichbarem Code und von Sprüngen auf Sprünge. Allerdings steht dafür hier nur ein wesentlich engerer Kontext zur Verfügung. Außerdem können sich solche Verbesserungen nicht mehr positiv auf weitere Optimierungen auswirken, falls die Nachoptimierung nicht iteriert

wird. Diese Maßnahmen sind deshalb in früheren Phasen wirkungsvoller.

Instruktionsmuster

Zur Nachoptimierung sind zwei unterschiedliche Methoden bekannt: Die erste vergleicht die Instruktionsfolge mit Instruktionsmustern für die Vorbedingung von Transformationen und führt dann entsprechende Transformationen durch. Die zweite löst die Wirkung der betrachteten Instruktionen in eine Folge elementarer Operationen auf und setzt diese dann zu möglichst effizienten Instruktionen wieder zusammen. Im folgenden geben wir einige Beispiele für Muster nach der ersten Methode an.

Typisches Beispiel für Redundanzen in aufeinanderfolgenden Code-Sequenzen sind Paare von Speicher- und Ladeinstruktionen, wie sie etwa bei der Übersetzung folgender Anweisungen entstehen können:

```
a := b+c    mov b, r
            add c, r
            mov r, a

d := a-e    mov a, r   (*)
            sub e, r
            mov r, d
```

Das Transformationsmuster

```
mov u,v     =>   mov u, v
mov v,u
```

entfernt die überflüssige, mit (*) markierte Instruktion.

Häufig werden bei der Code-Auswahl Seiteneffekte von Instruktionen, z. B. das Setzen des Bedingungscodes durch arithmetische Instruktionen, nicht ausgenutzt. Dies kann etwa durch Anwenden von

```
sub u, v    =>   sub u, v
cmp v, 0
```

oder

```
mov u, v    =>   mov u, v
cmp u, v
```

in der Nachoptimierung nachgeholt werden. Ebenso können hier Inkrement- und Dekrementinstruktionen erzeugt werden:

```
mov u, v
add 1, v     =>    inc u,
mov v, u
```

falls *v* danach nicht lebendig ist.

Um die Zahl der zu spezifizierenden Muster gering zu halten, sollte man Muster zusammenfassen, die für Klassen mehrer Instruktionen anwendbar sind, z. B. alle arithmetischen Instruktionen. Solch eine Nachoptimierungsphase ist aus der Spezifikation der Muster und der Repräsentation der Instruktionen auch mit einfachen Werkzeugen automatisch generierbar.

8.5 Anordnung von Instruktionen

Prozessorinterne Parallelität

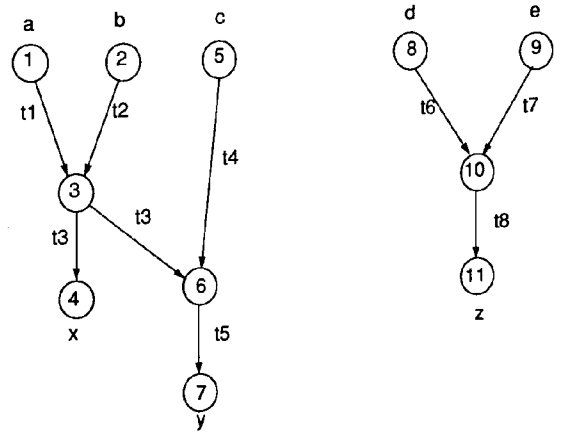
Prozessoren einiger Architekturklassen führen mehrere Instruktionen durch verschiedene Hardware-Komponenten parallel aus. Um den Leistungsgewinn solcher prozessorinterner Parallelität zu nutzen, muß der Übersetzer die Instruktionen in möglichst günstiger Reihenfolge anordnen. Für einige Prozessoren ist dies nicht nur eine Frage der Code-Optimierung, sondern der Korrektheit des Zielprogrammes. Verletzen Instruktionsfolgen Parallelisierbarkeitsbedingungen, ohne daß dies vom Prozessor erkannt und verhindert wird, so führt er sie zwar schnell, aber mit falschen Ergebnissen aus.

Anordnung von Instruktionen

Wir betrachten in diesem Abschnitt Anordnungsprobleme für zwei verschiedene Parallelisierungsprinzipien: parallele Funktionseinheiten und Fließbandverarbeitung. Für beide Strukturen gehen wir von gerichteten, azyklischen Graphen aus, die die Abhängigkeiten der auszuführenden Berechnungen abstrahieren. Sie werden vorweg an einem Beispiel eingeführt. Die *Anordnung von Instruktionen* (*instruction scheduling*) stellt sich damit als ein Optimierungsproblem der Graphanordnung dar. Zur Lösung solcher Probleme sind Verfahren aus der allgemeinen Anordnungstheorie anwendbar. Eine Übersicht gibt COFF76. Unter den vorliegenden Randbedingungen sind sie durchweg nicht gleichzeitig optimal und effizient lösbar. Es werden suboptimale, heuristische Algorithmen angewandt. In den beiden folgenden Abschnitten stellen wir die Probleme stark vereinfacht und modellhaft dar und zeigen nur einige einfache Lösungen auf.

Azyklische Graphen

Wir repräsentieren die Berechnungen eines Grundblockes als einen gerichteten, azyklischen Graphen. Er besteht im allgemeinen aus mehreren unverbundenen Teilgraphen. Abbildung 8.5-1 zeigt als Beispiel einen Grundblock mit seinem Graphen. Jeder Knoten steht für eine Operation. Die benannten Zwischenergebnisse in der Berechnungsfolge werden zu Kanten des Graphen. In einen Knoten einmündende Kanten repräsentie-




```
1:  t1  := a                1:  t1  := a
2:  t2  := b                2:  t2  := b
3:  t3  := t1 + t2          5:  t4  := c
4:  x   := t3               8:  t6  := d
5:  t4  := c                9:  t7  := e
6:  t5  := t3 + t4          3:  t3  := t1 + t2
7:  y   := t5               6:  t5  := t3 + t4
8:  t6  := d               10:  t8  := t6 + t7
9:  t7  := e                4:  x   := t3
10: t8  := t6 + t7          7:  y   := t5
11: z   := t8              11:  z   := t8
```

Abb. 8.5-1: Zwei Instruktionsfolgen mit ihrem Graphen.

ren die Operanden einer Operation. Die von einer Operation ausgehenden Kanten führen zu Operationen, die das Ergebnis verwenden. Knoten, in die keine Kanten münden, laden einen Wert aus dem Speicher, Knoten, aus denen keine Kanten entspringen, speichern einen Wert. Zur Vereinfachung nehmen wir zunächst an, daß alle im Block verwendeten Werte auf diese Weise gelesen und die Endergebnisse der Berechnungen so gespeichert werden. Weitere Wechselbeziehungen über den Speicher (einen Wert speichern und wieder laden) schließen wir zunächst innerhalb eines Blockes aus. Mit diesen Randbedingungen modelliert der Graph die Abhängigkeiten in der Berechnungsfolge. Die ursprünglich sequentielle Anordnung der Berechnungen ist in eine Halbordnung überführt, die nur noch inhaltlich notwendige Abfolgen beschreibt und damit größtmögliche Freiheiten zur Umordnung bietet. Eine solche Graphdarstellung kann durch Analyse der Ausdrücke (Abschnitt 8.1.3 und 8.3.2) gewonnen wer-

den. Zur Verdeutlichung dieser Redundanz der linearen Anordnung ist in Abbildung 8.5-1 eine zweite Operationsfolge angegeben, die zum gleichen Graphen führt und die gleiche Wirkung hat. Die Numerierung der Knoten zeigt die Zuordnung der Operationen.

In den beiden folgenden Abschnitten ordnen wir die Operationen unter Aspekten der Parallelverarbeitung neu an. Dabei nehmen wir in Abschnitt 8.5.1 an, daß mehrere Funktionseinheiten gleichzeitig Operationen verarbeiten. In Abschnitt 8.5.2 werden Anordnungen für die Fließbandverarbeitung in einem Prozessor hergestellt. Wir behandeln hier zur Vereinfachung beide Probleme unabhängig voneinander, obwohl im allgemeinen auch die parallelen Funktionseinheiten jeweils als Fließband organisiert sind.

8.5.1 Parallele Funktionseinheiten

Parallele Instruktionsströme

Prozessoren mit n parallelen Funktionseinheiten können im allgemeinen n verschiedene Operationen mit jeweils verschiedenen Operanden gleichzeitig bearbeiten. Der Prozessor wird dazu durch n parallele Instruktionsströme gesteuert, bzw. durch einen Strom von n-elementigen, zusammengesetzten Instruktionen. Man spricht deshalb auch von *horizontalem Code* oder *breiten Instruktionsworten* (*very long instruction words, VLIW*).Um die Leistungsfähigkeit des Prozessors auszunutzen, sollte der Code möglichst kompakt sein, ohne die Reihenfolgerestriktionen des Graphen zu verletzen. Die Zahl der nacheinander auszuführenden n-fachen Instruktionen und der Lücken in den breiten Instruktionen ist dann möglichst klein. Abbildung 8.5-2 zeigt eine Instruktionsfolge für einen Prozessor mit drei Funktionseinheiten, die zu dem Beispiel aus Abbildung 8.5-1 hergestellt wurden.

Wir beschreiben hier ein Verfahren zur Anordnung unter stark vereinfachten Randbedingungen. Insbesondere nehmen wir an, daß jede Operation von jeder beliebigen Funktionseinheit in einem Schritt, einem Prozessorzyklus, ausgeführt werden kann. Man kann das Anordnungsproblem dann anhand des Graphen wie folgt formulieren:

	Funktionseinheit		
	1	2	3
Instruktion			
1	t1 := a	t2 := b	t4 := c
2	t3 := t1 + t2	t6 := d	t7 := e
3	t5 := t3 + t4	t8 := t6 + t7	x := t3
4	y := t5	z := t8	leer

Abb. 8.5-2: Folge von dreifachen Instruktionen.

Anordnungsproblem

Der Graph wird in zwei Dimensionen ausgelegt. Seine Kanten verlaufen alle in Richtung der senkrechten Zeitachse. Die Knoten werden zu jedem Zeitschritt nebeneinander angeordnet. Die gleichzeitig von verschiedenen Funktionseinheiten in einem Schritt auszuführenden Operationen werden nebeneinander plaziert. Die Zahl der maximal nebeneinander angeordneten Knoten ist die *Breite der Anordnung*.

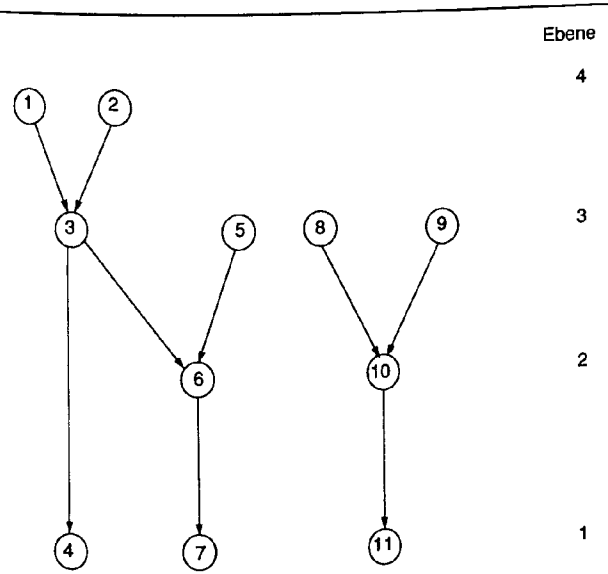


Abb. 8.5-3: Ebenenanordnung des Graphen aus Abbildung 8.5-1.

Ebenenanordnung

Als Ziel der Optimierung wird die Schrittzahl bei vorgegebener maximaler Breite minimiert. Grundlage vieler Verfahren und Heuristiken zur Lösung solcher Anordnungsprobleme sind sogenannte *Ebenenanordnungen*. Dazu bestimmt man zu jedem Knoten seine um 1 erhöhte maximale Entfernung von einem Endknoten des Graphen. Den Endknoten, von denen keine Kanten ausgehen, wird die Ebene 1 zugeordnet. Abbildung 8.5-3 zeigt unseren Graphen in seiner Ebenenanordnung. Hieran erkennt man für die Anordnung wichtige Eigenschaften. Die Anzahl der Ebenen ist die Zahl der mindestens auszuführenden Schritte, die auch mit beliebig vielen Funktionseinheiten nicht unterschritten werden kann. Ist die Anzahl der Operationen in einer Ebene größer als die Zahl der verfügbaren Funktionseinheiten, so müssen einige davon einem anderen Schritt zugeordnet werden. Der Graph enthält kritische Pfade. Dies sind alle Wege durch den Graphen, die je einen Knoten in jeder Ebene haben. Werden

Operationen des kritischen Pfades verschoben, benötigt die Ausführung mehr Schritte als durch die Zahl der Ebenen mindestens vorgegeben sind. Pfahler beschreibt in PFAH88 verschiedene Anordnungsverfahren anhand dieses Modells der Graphauslegung.

Das folgende einfache Verfahren von Hu (HU61) liefert optimale Ergebnisse für baumstrukturierte Graphen und beliebige maximale Breite k. Man wählt dabei für den nächsten Schritt der Anordnung aus den noch nicht angeordneten Operationen höchstens k mit größten Ebenennummern aus. Dabei dürfen Operationen aus verschiedenen Ebenen, die in einem Schritt angeordnet werden, nicht voneinander abhängen. Angewandt auf azyklische Graphen, die keine Bäume sind, liefert dieses Verfahren suboptimale Ergebnisse. Abbildung 8.5-4 zeigt eine so berechnete Auslegung unseres Graphen für $k = 3$ Funktionseinheiten. Sie liefert die Instruktionsfolge aus Abbildung 8.5-2. Für dieses Beispiel ist sie optimal. Da es für die allgemeine Lösung dieses Optimierungsproblems keine effizienten Algorithmen gibt und in realen Anwendungen weitere Restriktionen berücksichtigt werden müssen, verwendet man diese Ebenenanordnung als Grundlage vieler modifizierter, heuristischer Verfahren.

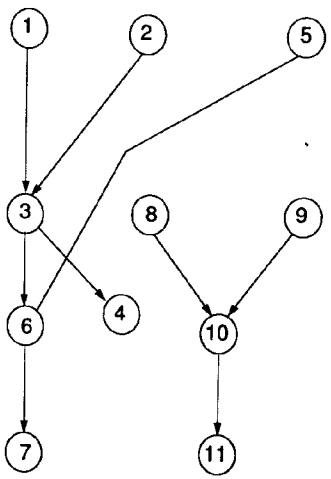


Abb. 8.5-4: Auslegung des Graphen für k=3 Funktionseinheiten.

Registerbedarf

Eine wichtige, zusätzliche Randbedingung für die Anordnung ist die Zahl der Register zur Unterbringung von Zwischenergebnissen. Auch diese Eigenschaft wird durch die zweidimensionale Auslegung

des Graphen modelliert. Die Kanten des Graphen repräsentieren Zwischenergebnisse und verbinden die erzeugenden und verwendenden Operationen.

Schnittweite

Legen wir wie in Abbildung 8.5-5 zwischen zwei Schritten der Anordnung einen waagerechten Schnitt durch den Graphen, so gibt die Zahl der geschnittenen Kanten die Anzahl der belegten Register an. Es muß dabei allerdings beachtet werden, daß ein Wert in mehreren Operationen verwendet werden kann. In der Graphik fassen wir deshalb die zu dem Wert gehörigen Kanten bis zu ihrem Verwendungsschritt zusammen. In einem formalen Modell zählt man entsprechend nur die geschnittenen Kanten, die unterschiedlichen Ursprung haben. Der Registerbedarf einer Graphauslegung ist damit durch die *Schnittweite* (*cut width*) der Anordnung des Graphen bestimmt. Das ist die maximale Zahl von Kanten, die jeweils zwischen zwei Schritten geschnitten werden. Das Problem der Registerminimierung kann damit auf das aus der Graphentheorie bekannte Problem der Minimierung der Schnittweite (*mincut*) zurückgeführt werden. Es ist allerdings unter den gegebenen Randbedingungen nicht effizient optimal lösbar. Man muß suboptimale Näherungsverfahren anwenden.

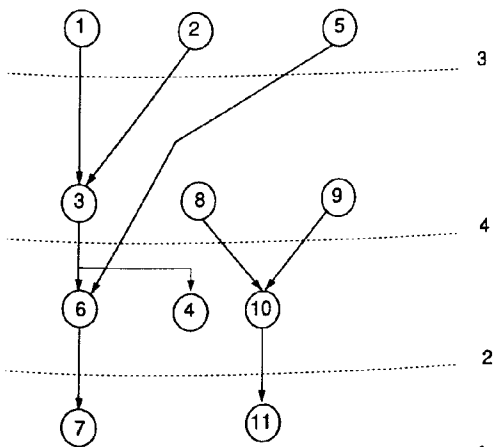


Abb. 8.5-5: Registerbedarf als Schnittweite der Graphauslegung.

Die unveränderte Anwendung des Ebenenverfahrens zur Auslegung kann sich ungünstig auf den Registerbedarf auswirken, da Operationen früher als nötig angeordnet werden. Dadurch kann die Lebensdauer der Werte unnötig verlängert werden. Man kann diesen Effekt teilweise ausgleichen, indem man mehrstellige Operationen aus der

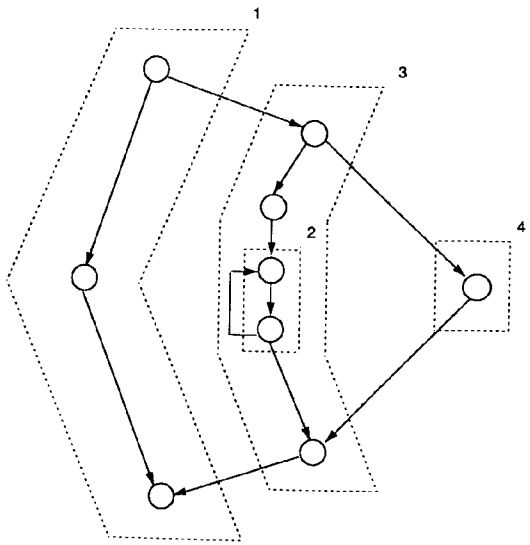


Abb. 8.5-6: Ablaufgraph mit Pfaden zur Kompaktierung.

gleichen Ebene bevorzugt oder in einer nachfolgenden Phase Knoten in spätere Schritte verschiebt, falls diese nicht ausgefüllt sind.

Zuteilung von Funktionseinheiten und Registern

Die berechnete Auslegung des Graphen bestimmt die Länge der Instruktionsfolge, die Ausnutzung der Funktionseinheiten und den Registerbedarf. Es ist noch offen, auf welcher Funktionseinheit jede Operation ausgeführt und in welchem Register jeder Wert gespeichert wird. Ein nachfolgender Zuteilungsschritt trifft diese Festlegungen. Dabei können in unserem Modell die Funktionseinheiten beliebig gewählt und die Register nach einem Verfahren für Grundblöcke (Abschnitt 7.5.2) zugeteilt werden. Instruktionen zum Umladen zwischen getrennten Registersätzen und zum Zwischenspeichern bei Registermangel müssen ggf. nachträglich eingeführt werden. Können sie nicht in Instruktionslücken eingefügt werden, so verlängern sie die Instruktionsfolge.

Kompaktierung für Programmpfade

Der erreichbare Kompaktheitsgrad des Codes hängt auch von der Zahl der Funktionseinheiten und der Länge der Grundblöcke ab. Da ein Grundblock im Mittel kaum mehr als acht Operationen umfaßt, die zum Teil voneinander abhängen, können damit schon drei oder mehr Funktionseinheiten kaum sinnvoll genutzt werden. Es ist deshalb notwendig, die Kompaktierung über Grundblöcke hinweg auf größere Strukturen auszudehnen. Fisher hat dazu in FISH81 ein Verfahren vorgestellt, das auf Programmpfaden basiert (*trace scheduling*). Man wählt im Ablaufgraphen des Programmes einen Pfad vom Anfangs- zum Endblock aus

und kompaktiert alle seine Operationen, als ob sie einem einzigen Grundblock angehören würden. Dabei werden Operationen auch über die Grenzen der ursprünglichen Grundblöcke verschoben. Die Semantik des Programmes wird durch zusätzliche Operationen in den angrenzenden Grundblöcken wiederhergestellt (*Kompensationscode*). Man kompaktiert dann iterativ nach dem gleichen Verfahren die verbleibenden Seitenpfade des Ablaufgraphen. Abbildung 8.5-6 gibt eine Gliederung eines Ablaufgraphen in Pfade mit der Reihenfolge ihrer Bearbeitung an.

Zyklische Programmpfade werden von diesem Verfahren nicht erfaßt, so daß Schleifen isoliert behandelt werden müssen. Es ist aber bemerkenswert, daß man aber Verbesserungen erzielen kann, wenn Lücken im kompaktierten Code einer Schleife durch Operationen von außen aufgefüllt werden. Da sich bei diesem Verfahren der Kompensationscode in später zu bearbeitenden Seitenpfaden akkumulieren kann, sinkt die Code-Qualität mit zunehmender Entfernung vom ersten gewählten Pfad. Es ist deshalb zweckmäßig, die Pfade in der Reihenfolge ihrer Ausführungshäufigkeit zu bearbeiten. Ellis beschreibt in ELLI86 ausführlich die Konstruktion eines Übersetzers nach dieser Methode. Inzwischen sind Verbesserungen der Methode entwickelt worden, welche die Verschiebung von Instruktionen anhand der Struktur des Ablaufgraphen beschränken, und damit die Verschlechterung des Codes in Seitenpfaden begrenzen (z. B. tree compaction in LAHA83)

8.5.2 Fließbandverarbeitung

In vielen Prozessoren wird die Arbeitsgeschwindigkeit durch *Fließbandverarbeitung* (*pipelining*) gesteigert. Die Abarbeitung der Instruktionen ist in mehrere Schritte gegliedert, z. B.

1) Instruktion beschaffen und decodieren,
2) Operanden beschaffen,
3) Operation ausführen,
4) Ergebnisse bereitstellen.

Fließbandstufen

Für jeden dieser Schritte ist eine spezielle Verarbeitungsstufe des Prozessors zuständig. In diesem Beispiel sprechen wir dann von einem vierstufigen Fließband. Die Prozessorstufen können parallel an ihrem Verarbeitungsschritt für verschiedene Instruktionen arbeiten. Zur Modellierung nehmen wir vereinfachend an, daß die Verarbeitungsdauer in jeder Stufe gleich ist. Dies ist die Zeit für einen Prozessorzyklus. In jedem Zyklus kann der Prozessor die Verarbeitung einer Instruktion mit der ersten Stufe beginnen. Sie wird dann an die nächsten Stufen weitergegeben und ist nach vier Zyklen abgearbeitet. Im eingeschwungenen Zustand werden jeweils vier aufeinanderfolgende Instruktionen gleichzeitig von verschiedenen Stufen bearbeitet. Abbildung 8.5-7 zeigt diese Fließbandverarbeitung schematisch.

Zyklus	Stufe 1	2	3	4
1	I1			
2	I2	I1		
3	I3	I2	I1	
4	I4	I3	I2	I1
5	I5	I4	I3	I2
6	I6	I5	I4	I3
7	I7	I6	I5	I4

Abb. 8.5-7: Schema der Fließbandverarbeitung.

Konflikte

Ist die Ausführung einer Operation auf mehrere Stufen verteilt ((2) - (4) in unserem Beispiel), so führt das Fließbandverfahren auf ein Problem der Anordnung der Instruktionen. Nehmen wir z. B. an, daß in zwei aufeinanderfolgenden Instruktionen von der ersten ein Wert berechnet wird, der Operand der zweiten ist, etwa

```
I1:   R1 := R2 + R3
I2:   R4 := R1 + R5,
```

dann stehen diese Operationen in einem *Schreib-Lese-Konflikt*. Abbildung 8.5-7 zeigt, daß das Ergebnis von *I1* erst im vierten Zyklus geschrieben, aber von *I2* schon im dritten benötigt wird. Die Instruktionen können so nicht korrekt ausgeführt werden. Fügen wir eine andere Instruktion zwischen den beiden ein, müßte der Wert im selben Zyklus von der vierten zur zweiten Stufe übertragen werden. Es wäre sogar möglich, daß das Ergebnis der Addition für *I1* in Stufe 3 unter Umgehung der übrigen Stufen direkt der Stufe 3 für die Addition von *I2* im nächsten Zyklus zugeleitet wird. Dann könnten beide Instruktionen unmittelbar nacheinander ausgeführt werden. Einige Prozessoren leisten solche Überbrückung der Stufen *(bypass)* automatisch. Wird keine dieser Techniken angewandt, muß der Abstand zwischen den Instruktionen *I1* und *I2* um zwei Zyklen vergrößert werden.

In der Prozessor-Hardware werden solche Konflikte unterschiedlich behandelt. Eine Klasse von Prozessoren ignoriert etwaige Konflikte zu dicht aufeinanderfolgender Instruktionen und führt sie schematisch

aus. Hier ist es für die Korrektheit des Zielprogrammes notwendig, daß der Übersetzer die Instruktionen konfliktfrei anordnet. Andere Prozessoren erkennen Konflikte und blockieren daraufhin das Fließband vorübergehend *(pipeline interlock)* bis alle Stufen konfliktfrei weiterarbeiten können. Diese von der Hardware durchgeführte Maßnahme hat den gleichen Effekt, der durch Einfügen von leeren Instruktionen an den Konfliktstellen erzielt werden kann. Der Übersetzer könnte jedoch stattdessen Instruktionen einfügen, die ohnehin ausgeführt werden müssen. In diesem Fall wird durch günstige Anordnung des Codes seine Ausführungsgeschwindigkeit verbessert.

Abstand

Je nach Struktur und Zahl der Fließbandstufen sind auch andere als der oben beschriebene Konflikt möglich. Dies gilt insbesondere, wenn abhängig von der Operation Register in unterschiedlichen Stufen gelesen oder geschrieben werden, oder wenn das Fließband bei Speicherzugriffen angehalten werden muß. Im folgenden abstrahieren wir deshalb von den speziellen Prozessoreigenschaften durch eine Funktion

Abstand (I1, I2) = k.

Sie gibt an, daß in der Instruktionssequenz *I2* nach mindestens *k* Zyklen konfliktfrei auf *I1* folgen kann.

Zur Behandlung eines Anordnungsverfahrens gehen wir hier wieder von den Operationen eines Grundblockes aus, die in einem azyklischen Graphen repräsentiert sind. Wie in der Einführung zum Abschnitt 8.5 nehmen wir wieder an, daß die Register für Zwischenergebnisse noch nicht zugeteilt sind, alle benötigten Werte aus dem Speicher geladen und alle Ergebnisse des Blockes in den Speicher geschrieben werden. Wir gehen von einem vierstufigen Fließband mit oben beschriebener, einfacher Charakteristik aus. Hier sind nur die Stufen 2 und 4 für die Anordnung der Instruktionen wichtig:

	Stufe 2	Stufe 4
arithmetische Instruktionen	Operanden lesen	Ergebnis schreiben
Ladeinstruktion	Adresse lesen	Inhalt laden
Speicherinstruktion	Adresse und Wert lesen	Wert speichern

Falls der Prozessor Werte von Stufe 4 nach Stufe 2 im selben Zyklus übertragen kann, nimmt die Funktion *Abstand* nur Werte 1 oder 2 an, andernfalls kann auch ein konfliktfreier Abstand von 3 notwendig sein. Für unser Beispiel nehmen wir die erstere Variante an. Abbildung 8.5-8 zeigt die Instruktionen aus Abbildung 8.5-1 in gleicher Reihenfolge mit eingefügten leeren Instruktionen zur Vermeidung von Konflikten. Im zugehörigen Graphen müssen alle durch Kanten verbundene Operationen einen

Abstand von mindestens 2 haben. Das Ziel von Anordnungsverfahren ist es, mit möglichst wenigen leeren Instruktionen auszukommen.

Anordnungsverfahren

Das folgende einfache, heuristische Verfahren nach GIBB86 basiert auf der in Abschnitt 8.5.1 vorgestellten Ebenenanordnung. Die nächste anzuordnende Instruktion *I* wird nach folgenden Kriterien ausgewählt:

1) Alle Vorgänger von *I* müssen angeordnet sein.
2) *I* steht nicht in Konflikt mit einer der *m* vorangehenden angeordneten Instruktionen. Dies wird mit der Funktion *Abstand* überprüft. Dabei ist *m* um 1 kleiner als der maximale Wert, den *Abstand* annehmen kann.
3) *I* hat möglichst viele Nachfolger im Graphen. Diese stehen mit *I* in Konflikt. Wird *I* früh angeordnet, so besteht größere Aussicht alle diese Konflikte ohne Leerinstruktionen zu vermeiden.
4) *I* hat die höchste Ebenennummer der zur Auswahl anstehenden Operationen.

Registerzuteilung

Erfüllt in einem Anordnungsschritt keine Operation die Kriterien (1) und (2), so muß eine leere Instruktion eingefügt werden. Die zweite Instruktionsfolge in Abbildung 8.5-8 ist nach diesem Verfahren mit der Ebenenanordnung aus Abbildung 8.5-3 angeordnet worden. Sie ist um 6 Instruktionen kürzer als die ursprüngliche Anordnung und für dieses Beispiel optimal. Das Verfahren liefert nicht immer optimale Anordnungen. Weitere Restriktionen des Fließbandes können in der Funktion Abstand oder durch zusätzliche Kriterien eingebracht werden. Können Abhängigkeiten über Speicherzugriffe nicht ausgeschlossen werden, so muß deren Reihenfolge aus der ursprünglichen Sequenz eingehalten werden. In der so angeordneten Instruktionsfolge werden die Register für Zwischenergebnisse nach einem Verfahren für Grundblöcke zugeteilt. Reichen die verfügbaren Register nicht aus, so muß Code zum Zwischenspeichern mit zusätzlichen Leerinstruktionen konfliktfrei eingefügt werden.

Anders als in obigem Verfahren kann der Übersetzer auch zunächst den Code mit endgültig zugeteilten Registern ohne Rücksicht auf die Fließbandverarbeitung erzeugen. Eine nachfolgende Übersetzerphase verbessert dann ihre Anordnung. Ein solches Verfahren wird in HENN83 vorgestellt. Es ist wie die Nachoptimierung an der Assemblierschnittstelle anwendbar. Dazu wird ebenso wie oben beschrieben ein Graph für die Operationsabhängigkeiten aufgestellt. Hier kann in der ursprünglichen Instruktionsfolge dasselbe Register nacheinander für verschiedene Zwischenergebnisse verwendet werden. Bei der Umordnung dürfen die unterschiedlichen Verwendungen nicht miteinander verzahnt werden. Der Anordnungsalgorithmus hält dazu Informationen über noch zu verwendende Registerinhalte und die Registerverwendungen in den noch anzuordnenden Instruktionen. Außerdem wird mit die-

Instruktionsfolge mit notwendigen leeren Instruktionen			umgeordnete Instruktionsfolge		
1:	t1	:= a	1:	t1	:= a
2:	t2	:= b	2:	t2	:= b
	leer		5:	t4	:= c
3:	t3	:= t1 + t2	3:	t3	:= t1 + t2
	leer		8:	t6	:= d
4:	x	:= t3	9:	t7	:= e
5:	t4	:= c	6:	t5	:= t3 + t4
	leer		10:	t8	:= t6 + t7
6:	t5	:= t3 + t4	4:	x	:= t3
	leer		7:	y	:= t5
7:	y	:= t5	11:	z	:= t8
8:	t6	:= d			
9:	t7	:= e			
	leer				
10:	t8	:= t6 + t7			
	leer				
11:	z	:= t8			
6 leere Instruktionen			keine leere Instruktion		
20 Zyklen			14 Zyklen		

Abb. 8.5-8: Instruktionsfolgen für Fließbandverarbeitung.

ser vorausschauenden Information vermieden, daß der Algorithmus in eine Situation gerät, in der er eine angefangene Instruktionsfolge nicht mehr korrekt fortsetzen kann. Dies wäre dann der Fall, wenn die Anfänge zweier Teilgraphen verzahnt angeordnet sind, in denen zwei Register in unterschiedlicher Reihenfolge geschrieben und gelesen werden.

Verzögerte Sprünge

Abschließend sei noch auf das Problem von Sprunginstruktionen bei Fließbandverarbeitung hingewiesen. Das Sprungziel und die Sprungentscheidung bei bedingten Sprüngen werden erst in einer späteren Fließbandstufe, z. B. in der zweiten oder dritten, bestimmt. Entweder hält der Prozessor immer beim Erkennen eines Sprunges das Fließband an, bis diese Stufe ausgeführt ist. Dann sind keine besonderen Maßnahmen zur Instruktionsanordnung notwendig. Allerdings werden dadurch Sprünge langsamer als nötig abgearbeitet. Oder der Prozessor führt noch eine bestimmte Anzahl *k* Instruktionen, die auf den Sprung folgen und sich schon auf dem Fließband befinden, vollständig aus. Erst danach wird der so *verzögerte Sprung* (*delayed branch*) wirksam. Für diesen Fall ist wieder ein Anordnungsproblem zu lösen: Statt hinter jeden Sprung *k* Leerinstruktionen einzufügen, wird versucht, dorthin Instruktionen zu verschieben, die

vor dem Sprung auszuführen sind, aber die Sprungentscheidung nicht beeinflussen. Bei unbedingten Sprüngen kann man auch die ersten k Instruktionen nach dem Sprungziel dorthin kopieren und das Sprungziel um k Instruktionen verschieben.

Anhang

1. Literaturhinweise

Zur Vertiefung des Inhaltes dieses Bandes seien zunächst die beiden derzeit aktuellsten umfassenden Lehrbücher zum Übersetzerbau empfohlen:

W.M. Waite, G. Goos: **Compiler Construction.** Berlin-Heidelberg-New York: Springer, 1984.

Dieses Werk umspannt das Themenspektrum von den Grundlagen formaler Systeme bis hin zu Implementierungstechniken. Es betont die Gesamtstruktur und Schnittstellen der Übersetzer und behandelt ausführlich wichtige Entwurfsaspekte wie Sprach- und Maschineneigenschaften und Fehlerbehandlung. Im Bereich der Analyseaufgaben wird neben der syntaktischen auch die semantische Analyse mit attributierten Grammatiken angemessen ausführlich dargestellt. Das Lehrbuch

A.V. Aho, R. Sethi, J.D. Ullman: **Compilers, Principles, Techniques and Tools.** Reading, Ma: Addison-Wesley, 1986.

behandelt recht umfassend und ausführlich eine Vielzahl von Verfahren und Algorithmen zu allen Übersetzeraufgaben. Die Schwerpunkte liegen in den Bereichen Syntaxanalyse, Code-Erzeugung und Optimierung. Das schon klassische Werk

A.V. Aho, J.D. Ullman: **The Theory of Parsing, Translation and Compiling.** Englewood Cliffs, N.J.: Prentice-Hall, 1972.

enthält theoretische Grundlagen insbesondere zu den älteren Standardverfahren der Syntaxanalyse und Optimierung. Das zweibändige Lehrbuch

H. Zima: **Compilerbau I u. II,** Reihe Informatik/36 u. 37. Mannheim: BI Wissenschaftsverlag, 1982 u. 1983.

betont stark formalisierte Darstellungen und präsentiert die zum Teil älteren Techniken sehr detailliert.

Zum Teilgebiet Optimierung und Datenflußanalyse gibt

S.S. Muchnick, N.D. Jones: **Program Flow Analysis: Theory and Applications.** Englewood Cliffs; N.J.: Prentice-Hall, 1981.

eine gute zusammenfassende Darstellung.

Systematische Methoden und Werkzeuge zum Übersetzerbau sind in

B. Lorho (ed.): **Methods and Tools for Compiler Construction.** Cambridge University Press, 1984.

in einer Reihe von Einzelbeiträgen zu einem Seminar über das Thema zusammengestellt.

Ferner sei hier auf die Berichte der Konferenzreihen der ACM

Symposium on Compiler Construction,

Symposium on Principles of Programming Languages,

Symposium on Architectural Support for Programming Languages and Operating Systems

hingewiesen, die regelmäßig aktuelle Beiträge zum Übersetzerbau enthalten.

Für die Konstruktion von Übersetzern ist ein tiefgehendes Verständnis von Programmiersprachen und ihren Grundkonzepten eine unverzichtbare Voraussetzung. Hierzu sei zunächst auf die weiteren Bände im Hauptgebiet 3 dieses Handbuches verwiesen. Darüber hinaus können

E. Horowitz: **Fundamentals of Programming Languages.** Berlin-Heidelberg-New York: Springer, 1983.

M. Marcotty, H. Ledgard: **The World of Programming Languages.** Berlin-Heidelberg-New York: Springer, 1987.

ein grundlegendes Verständnis vermitteln. Ersteres ist stark an reale Programmiersprachen angelehnt, während das zweite Sprachkonzepte in abstrahierten Beispielsprachen vorstellt.

Schließlich sei noch auf eine Reihe wichtiger Sprachdefinitionen mit Referenzen auf das Literaturverzeichnis verwiesen: Ada (ANSI83a), Algol 60 (NAUR63), Algol 68 (WIJN75), APL (IVER62), C (KERN78, ANSI88), COBOL (ANSI68), Fortran (ANSI66, ANSI78), Lisp (MCCA65), Modula-2 (WIRT85). Pascal (JENS85, ANSI83b), Smalltalk (GOLD83), Snobol (GRIS71).

2. Literaturverzeichnis

AHOU72 Aho, A. V., Ullman, J. D.: The Theory of Parsing, Translation and Compiling. Englewood Cliffs: Prentice Hall, 1972.

AHOG85 Aho, A. V., Ganapathi, M.: Efficient tree pattern matching: an aid to code generation. Proc. of the Twelfth ACM Symp. on Prin. of Prog. Lang., 1985, 334-340.

AHOS86 Aho, A. V., Seti, R., Ullman, J. D.: Compilers Principles, Techniques and Tools. Reading, MA: Addison Wesley, 1986.

ALLE71 Allen, F. E., Cocke, J.: A Catalogue of Optimizing Transformation. In: Rustin, R. R. (editor): Design and Optimization of Compilers. Englewood Cliffs, NJ: Prentice Hall, 1972.

ANSI66 FORTRAN, X3.9-1966, American National Standards Institute, New York, 1966.

ANSI68 COBOL, X3.23-1968, American National Standards Institute, New York, 1968.

ANSI78 FORTRAN, X3.9-1978, American National Standards Institute, New York, 1978.

ANSI83a Reference Manual for the Ada Programming Language, American National Standards Institute/MIL-STD 1815, American National Standards Institute, New York, 1983.

ANSI83b Programming Language PASCAL, American National Standards Institute/IEEE 770 X3.97-1983, American National Standards Institute, New York, 1983.

ANSI88 Programming Language C, X3. 159-1988, American National Standards Institute, New York, 1988.

BAUE76 Bauer, F. L.: Historical Remarks on Compiler. In: Bauer, F. L., Eickel, J. (editor): Compiler Construction /196/ An Advanced Course. Lecture Notes in Computer Science 21, Berlin-Heidelberg-New York: Springer, 1976, 603-621.

BELA66 Belady, L. A.: A Study of Replacement Algorithms for a Virtual Storage Computer. IBM Systems Journal 5, 2 (1966), 78-101.

CHAI82 Chaitin, G. J.: Register Allocation & Spilling via Coloring. SIGPLAN Notices 17, 6 (1982), 98-105.

CLEM88 Clemm, G. M: The Odin Specification Language. In: International Workshop on Software Version and Configuration Control '88. Stuttgart: Teubner, 1988.

COFF76 Coffman, E. G., (ed.): Computer and Job-Shop Scheduling Theory. New York: Wiley & Sons, 1976.

DAVI80 Davidson, J. W., Fraser, C. W.: The Design and Application of a Retargetable Peephole Optimizer. Trans. Prog. Lang and Systems 2, 2 (1989), S. 191-202.

DENC84 Dencker, P., Dürre K., Heuft, J.: Optimization of Parser Tables for Portable Compilers. Trans. Prog. Lang and Systems 6, 4 (1984), S. 546-572.

DERA88 Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars. Lecture Notes in Computer Science 323. Berlin-Heidelberg-New York: Springer, 1988.

DERE69 DeRemer, F. L.: Practical Translators for LR(k) Languages. MAC-Tech. Rep.-65, MIT, Cambridge, MA, 1969.

DERE71 DeRemer, F. L.: Simple LR(k) Grammars. Comm. of the ACM 14, 7 (1971), S. 453-460.

ELLI86 Ellis, J. R.: Bulldog. A Compiler for VLIW Architectures, MIT, Cambridge, MA, 1986.

FARR82 Farrow, R: LINGUIST-86, Yet Another Translator Writting System based on Attribute Grammars. SIGPLAN Notices 17, 6 (1982), S. 160-171.

FISH81 Fisher, J. A.: Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Trans. on Computers C-30, 7 (1981), S. 478-490.

GANA82 Ganapathi, M., Fischer, C. N., Hennessy, J. L.: Retargetable compiler code generation. Computing Surveys 14, 4 (1982), S. 573-592.

GANZ82 Ganzinger, H., Giegerich, R., Möncke, U., Wilhelm, R.: A Truly Generative Semantics /196/ directed Compiler Generator. SIGPLAN Notices 17, 6 (1982), S. 172-184.

GIBB86 Gibbons, P. B., Muchnick, S. S.: Efficient Instruction Scheduling for a Pipelined Architecture. SIGPLAN Notices 21, 9 (1986), S. 11-18.

GLAN78 Glanville, R. S., Graham, S. L.: A New Method for Compiler Code Generation. In: Conf. Record of the Fifth ACM Symp. on Prin. of Prog. Lang. Association for Computing Machinery, New York, 1978, S. 231-240.

GOLD83 Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Reading, Mass.: Addison Wesley 1983.

GORD76 Gordon, M.: The Denotational Definition of Programming Languages. An Introduction. Berlin-Heidelberg-New York: Springer, 1976.

GRAH82 Graham, S. L., Henry, R. R., Schulman, R. A.: An Experiment in Table Driven Code Generation. SIGPLAN Notices 17, 6 (1982), S. 32-43.

GRAY87 Gray, R. W.: Generating Fast, Error Recovering Parsers. Boulder, CO.: MS Thesis, Dpt. of Computer Science, University of Colorado, 1987.

GRIS71 Griswold, R., Paage, J., Polonsky, J.: The SNOBOL 4 Programming Language, 2nd ed. . Englewood Cliffs, N. J.: Prentice-Hall 1971.

GROS89 Grosch, J.: Generators for High Speed Front-Ends. In: Hammer, D. (ed.): Workshop on Compiler, Compiler and High Speed Compilation. Lecture Notes in Computer Science 371. Berlin-Heidelberg-New York: Springer, 1989.

HENN83 Hennessy, J., Gross, T.: Postpass Code Optimization of Pipeline Constraints. Trans. Prog. Lang and Systems 5, 3 (1983), S. 422-448.

HEUR86 Heuring, V. P.: The Automatic Generation of Fast Lexical Analyzers. Software-Practice & Experience 16, 9 (1986), S. 801-808.

HOAR73 Hoare, C. A. R., Wirth, N.: An Axiomatic Definition of the Programming Language PASCAL. Acta Inf. 3 (1973), S. 335-355.

HORO83 Horowitz, E.: Fundamentals of Programming Languages. s. S. 232.

HU61 Hu, T. C.: Parallel Sequencing and Assembly Line Problems. Operations Research 9, 6 (1961).

IVER62 Iverson, K.: A Programming Language. New York: Wiley 1962.

JENS85 Jensen, K., Wirth, N., Mickel, A. B., Miner, J. F.: Pascal User Manual an Report. Third Edition. Berlin-Heidelberg-New York: Springer, 1985.

JOHN75 Johnson, S. C.: Yacc /196/ Yet Another Compiler-Compiler. Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

JOUR84 Jourdan, M.: Recursive Evaluators for Attribute Grammars: an Implementation. In: Lorho, B. (ed.): Methods and Tools for Compiler Construction. Cambridge, UK: Cambridge University Press, 1984, S. 139-164.

KAST80 Kastens, U.: Ordered Attribute Grammars. Acta Inf. 13, 3 (1980), S. 229-256.

KAST82 Kastens, U., Hutt, B., Zimmermann, E.: GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141. Berlin-Heidelberg-New York: Springer, 1982.

KAST89a Kastens, U.: Abstract Interfaces for Compiler Generating Tools. In: Hammer, D. (ed.): Workshop on Compiler-Compiler and High Speed Compilation. Lecture Notes in Computer Science 371. Berlin-Heidelberg-New York: Springer, 1989.

KAST89b Kastens, U.: LIGA: A Language Independent Generator for Attribute Evaluators. Bericht Nr. 63 der Reihe Informatik, Universität-GH Paderborn, 1989.

KENN81 Kennedy, K.: A Survey of Data Flow Analysis Techniques. In: Muchnick, S. S., Jones, N. D. (ed.): Program Flow Analysis: Theory and Applications. Englewood Cliffs, NJ: Prentice Hall, 1981, S. 5-54.

KERN78 Kernighan, B. W., Ritchie, D. M.: The C Programming Language. Englewood Cliffs, NJ: Prentice Hall, 1978.

KNUT65 Knuth, D. E.: On the Translation of Languages from Left to Right. Inf. and Control 8, 6 (1965), S. 607-639.

KNUT68 Knuth, D. E.: Semantics of Context-Free Languages. Mathematical Systems Theory 2, 2 (1968), S. 127-146.

LAHA83 Lah, J., Atkins, D. E.: Tree Compaction of Microprogramms. In: Proc. of the 16th Annual Microprogramming Workshop. New York: IEEE, 1983, 922-933.

LAND82 Landwehr, R., Jansohn, H., Goos, G.: Experience With an Automatic Code Generator Generator. SIGPLAN Notices 17, 6 (1982).

LESK75 Lesk, M. E.: LEX – A Lexical Analyzer Generator. Computing Science Technical Report 39. Murray Hill, NJ: Bell Telephone Laboratories, 1975.

LORH84 Lorho, B., (ed.): Methods and Tools for Compiler Construction. Cambridge, UK: Cambridge University Press, 1984.

LUCA69 Lucas, P., Walk, K.: On the Formal Description of PL/I. Annual Review in Automatic Programming 6, 3 (1969), S. 105-181.

MARC87 Marcotty, M., Ledgard, H.: The World of Programming Languages. Berlin-Heidelberg-New York: Springer, 1987.

MCCA65 Mc Carthy, J., Levin, S.: LISP 1.5 Programmer's Manual, 2nd ed. . Camebridge, Mass.: M.I.T. Press 1965.

MOREE79 Morel, E., Renvoise, C.: Global Optimization by Suppression of Partial Redundancies. Comm. of the ACM 22, 11 (1979), S. 96-103.

MUCH81 Muchnick, S. S., Jones, N. D., (eds.): Program Flow Analysis: Theory and Applications. Englewood Cliffs, NJ: Prentice Hall, 1981.

NAUR63 Naur, P.: Revised Report on the Algorithmic Language ALGOL 60. Comm. of the ACM 6, 1 (1963), S. 1-17.

PAYT82 Payton, T., Keller, S., Perkins, J., Rowan, S., Mardinly, S.: SSAGS, a Syntax and Semantics Analysis Generation System. IEEE COMPSAC'82, 1982, S. 424-432.

PFAH88 Pfahler, P.: Übersetzermethoden zur automatischen Hardware-Synthese. Dissertation, Universität-GH Paderborn, FB 17, 1988.

PURD80 Purdom, P., Brown, C. A.: Semantic Routines and LR(k) Parsers. Acta Inf., 1980, S. 299-316.

RÄIH84 Räihä, K.: Attribute Grammar Design using the Compiler Writing System HLP. In: Lorho, B. (ed.): Methods and Tools for Compiler Construction. Cambridge, UK: Cambridge University Press, 1984, S. 183-206.

RECH85 Rechenberg, P., Mössenböck, M.: Ein Compiler-Generator für Mikrocomputer. München: Hanser, 1985.

RÖHR80 Röhrich, J.: Methods for the Automatic Construction of Error Correcting Parsers. Acta Inf. 13, 2 (1980), S. 115-139.

ROSE70 Rosenkrantz, D. J., Stearns, R. E.: Properties of Deterministic Top-Down Grammars. Inf. and Control 17 (1970), S. 226-256.

RUST72 Rustin, R. R., (ed.): Design and Optimization of Compilers. Englewood Cliffs, NJ: Prentice Hall, 1972.

SETH70 Sethi, R., Ullman, J. D.: The Generation of Optimal Code for Arithmetic Expressions. J. ACM 17, 4 (1970), S. 715-728.

TANE82 Tanenbaum, A. S., Staveren, H., Stevenson, J. W.: Using Peephole Optimization on Intermediate Code. Trans. Prog. Lang and Systems 4, 1 (1982), S. 21-36.

TANE83 Tanenbaum, A. S., Staveren, H., Keizer, E. G., Stevenson, J. W.: A practical toolkit for making portable compilers. Comm. of the ACM 26, 9 (1983), S. 654-660.

TENN76 Tennent, R. D.: The Denotational Semantics of Programming Languages. Comm. of the ACM 19, 8 (1976), S. 437-453.

WAIT84 Waite, W. M., Goos, G.: Compiler Construction. Berlin-Heidelberg-New York: Springer, 1984.

WAIT85 Waite, W. M., Carter, L. R.: The Cost of a Generated Parser. Software-Practice & Experience 15, 3 (1985), S. 221-239.

WAIT86 Waite, W. M.: The Cost of Lexical Analysis. Software-Practice & Experience 16, 5 (1986), S. 473-488.

WAIT88 Waite, W. M., Heuring, V. P., Kastens, U.: Configuration Control in Compiler Construction. In: International Workshop on Software Version and Configuration Control '88. Stuttgart: Teubner, 1988.

WEGN72 Wegner, P.: The Vienna Definition Language. Computing Surveys 4, 1 (1972), S. 5-63.

WIJN75 Wijngaarden, A., et. al.: Revised Report on the Algorithmic Language ALGOL 68. Acta Inf. 5 (1975), S. 1-236.

WIRT85 Wirth, N.: Programming in Modula-2, Third Edition. Berlin-Heidelberg-New York: Springer, 1985.

ZIMA82 Zima, H.: Compilerbau I und II, Reihe Informatik/36 u. 37. Mannheim: BI Wissenschaftsverlag 1982 u. 1983.

3. Register