Ein Übersetzerbau-Praktikum

Uwe Kastens
Fakultät für Informatik der Universität Karlsruhe

1. Einführung

Im Sommersemester 1977 haben wir im Rahmen der Informatik-Ausbildung an der Universität Karlsruhe ein Übersetzerbau-Praktikum durchgeführt. Es sollte den Studenten Gelegenheit geben, die Kenntnisse aus der Übersetzerbau-Vorlesung praktisch anzuwenden und gleichzeitig Methoden der Programmkonstruktion zu erlernen. Zielsetzung und Durchführung des Praktikums unterscheiden sich wesentlich von denen ähnlicher Veranstaltungen und sollen deshalb hier diskutiert werden.

1.1 Zielsetzung

Man mag sich fragen, weshalb der Ausbildung im Bereich Übersetzerbau eine so grosse Bedeutung beigemessen wird, obwohl nur wenige Informatiker nach ihrem Studium tatsächlich Übersetzer für höhere Programmiersprachen entwickeln. Die Problemstellungen des Übersetzerbaus betreffen jedoch ein viel weiteres Gebiet: Die Implementierung komfortabler Anwendersprachen gewinnt zunehmend an Bedeutung. Darüber hinaus stellen sich bei jedem Programmsystem, das nicht-triviale Eingabedaten verarbeitet, z.B. Textverarbeitungs-, Datenbank- oder Abfrage-Systeme, typische Aufgaben aus dem Bereich des Übersetzerbaus, z.B. Symbolentschlüsselung, syntaktische Analyse und Transformation in eine Zielsprache oder in Aktionen eines Grundsystems. Eine solche Betrachtungsweise erschliesst systematische Methoden des Übersetzerbaus für die Lösung dieser Aufgaben und erhöht damit die Qualität des Software-Produktes.

Es ist wichtig, dass Studenten systematische Methoden des Übersetzerbaus nicht nur in der Theorie lernen, sondern auch beim praktischen Einsatz deren Anwendbarkeit erproben, und die Randbedingungen kennenlernen, die sich bei der Anwendung der Techniken in einem grösseren Projekt ergeben. So wird beispielsweise das Verständnis für tabellengesteuerte Zerteilungs-Methoden wesentlich vergrössert, wenn man neben dem Erlernen der Theorie tatsächlich einen solchen Zerteiler in einen Übersetzer integriert und dabei seine Ein- und Ausgabe-Schnittstellen kennenlernt.

Höhere Programmiersprachen werden in der Praxis im allgemeinen von Mehrlauf-Übersetzern implementiert. Das Praktikum soll deshalb systematische Lösungen für typische Mehrlauf-Probleme, wie Lauf-Einteilung und Behandlung von Zwischensprachen, vermitteln.

Durch die Entwicklung eines hinreichend komplexen Übersetzers erwerben die Studenten Erfahrungen im Lesen, Verstehen und Modifizieren komplexer Programmsysteme. Solche Fähigkeiten sind für die Konstruktion umfangreicher Programmsysteme - unabhängig vom Übersetzerbau - sehr wertvoll.

1.2 Durchführung

Die praktischen Übungen zum Übersetzerbau werden häufig so abgehalten, dass jeder Student für eine triviale Programmiersprache selbständig einen sehr einfachen Übersetzer implementiert. Eine solche Reduktion der Aufgabenstellung vermittelt ein verfälschtes Bild der Problematik des Übersetzerbaus: Wesentliche Aufgaben eines Übersetzers werden übersehen, da sie nur noch am Rande auftreten, z.B. semantische Analyse. Wichtige Techniken des Übersetzerbaus werden nicht angewandt, da sie erst bei der Implementierung nichttrivialer Programmiersprachen sinnvoll sind, z.B. systematische Analyse von Kontextabhängigkeiten. Viele Probleme werden unsystematisch gelöst, wenn sie für eine kleine Beispiel-Sprache nur sehr geringen Aufwand verursachen, z.B. Fehlerbehandlung.

Wir haben deshalb mit unserem Praktikum einen anderen Weg beschritten: Es wird ein Übersetzer für eine Programmiersprache implementiert, die hinsichtlich ihrer Komplexität mit modernen höheren Programmiersprachen wie PASCAL vergleichbar ist. Der Übersetzer ist ein hinreichend komplexes Software-Produkt, an dem die Notwendigkeit moderner Übersetzerbau-Techniken und Methoden des Software-Engineering demonstriert werden können.

Wie erreicht man, dass ein Student in der Lage ist, in einer ein-semestrigen Lehrveranstaltung eine solch umfangreiche Aufgabe zu bewältigen? Eine Verteilung von Unteraufgaben an verschiedene Studenten kommt nicht in Frage, da jeder Student den gesamten Übersetzer kennenlernen soll. Wir haben deshalb einen modular strukturierten Übersetzer vollständig entworfen und implementiert. Die Teilmoduln sind zu einzelnen Themenbereichen zusammengefasst. Aufgabe der Studenten ist es, zu jedem Themenbereich einige kleinere, zentrale Moduln zu entwerfen, zu implementieren und in die übrigen Moduln, die ihnen zur Verfügung gestellt werden, zu integrieren. Auf diese Weise wird der Lehrstoff sowohl durch selbständige Entwicklung von Algorithmen als auch durch Lesen und Verstehen von fertigen Programmteilen vermittelt. Mit diesem Verfahren erreichen wir gleichzeitig weitere wesentliche Ziele: Die Studenten lernen fest vorgegebene Aufgaben- und Schnittstellen-Spezifikationen präzise einzuhalten - eine Erfahrung, die in allen Bereichen der Programmkonstruktion wertvoll ist. Die Struktur und der Programmierstil der vorgegebenen Moduln prägt die Entwicklung der eigenen Programmteile. (Das ging in einigen Fällen soweit, dass Unsauberkeiten in den vorgegebenen Moduln

- berechtigt - kritisiert wurden.)

Aufgrund dieser Vorgehensweise sind wir in der Lage, gleichzeitig moderne Techniken, des Übersetzerbaus und Methoden der Programmkonstruktion in relativ kurzer Zeit zu vermitteln.

2. Übersetzer-Struktur

Die wichtigste Voraussetzung für alle Ziele, die mit dem Praktikum erreicht werden sollen, ist eine modulare Übersetzer-Struktur mit klar definierten, überschaubaren Schnittstellen. Nur ein wohl-strukturiertes Programm kann mit relativ geringem Aufwand beim Lesen verstanden werden. Die Schnittstellen der Moduln müssen präzise spezifiziert und überschaubar sein, damit die Studenten in der Lage sind eigene Moduln zu integrieren.

Die Struktur eines Übersetzers entwickelt man entweder ausgehend von den Elementen der zu implementierenden Sprache (wie in [Wi 77] demonstriert) oder ausgehend von den unterschiedlichen Aufgaben, die für die Übersetzung zu lösen sind. Im ersten Fall wird der Übersetzer in Moduln gegliedert, die jeweils ein Sprachelement weitgehend vollständig behandeln. Diese Mothode führt im allgemeinen zu unsystematischen Lösungen, da vielfach in verschiedenen Moduln gleichartige Teilaufgaben erledigt werden, ohne dass die zugrundeliegende Systematik erkannt und ausgenutzt wird. Gliedert man den Übersetzer stattdessen so, dass jeder Modul eine inhaltlich abgeschlossene Teilaufgabe löst (z.B. Zerteilung, Bezeichner-Identifikation, Artabgleich) und auf alle Sprachelemente angewandt wird, für die diese Aufgabe relevant ist, so werden die wesentlichen Übersetzeraufgaben deutlich. Es können dann systematische Lösungsmethoden angewandt werden, die sich leicht für andere Anwendungen verallgemeinern lassen. Eine solche Übersetzerstruktur ist ausserdem weitgehend unabhängig von der implementierten Sprache und der Zielmaschine.

Der Entwurf der Übersetzerstruktur ist eine recht schwierige Aufgabe, die einige Vorkenntnisse und Erfahrungen im Übersetzerbau erfordert. Will man zeitaufwendige Fehlentscheidungen vermeiden, so kann man sie nicht von den Studenten selbstständig lösen lassen. Wir legen deshalb die Struktur des Übersetzers von vornherein fest und beteiligen die Studenten, indem wir ihnen die Entwurfsentscheidungen verdeutlichen, so dass sie den Entwurf nachvollziehen können: Die Teilaufgaben des Übersetzers werden aus der Sprachdefinition erarbeitet. Es wird aufgezeigt, welche Abhängigkeiten zwischen ihnen bestehen. Diese veranschaulicht man in einem Abhängigkeitsdiagramm [Abb. 1].

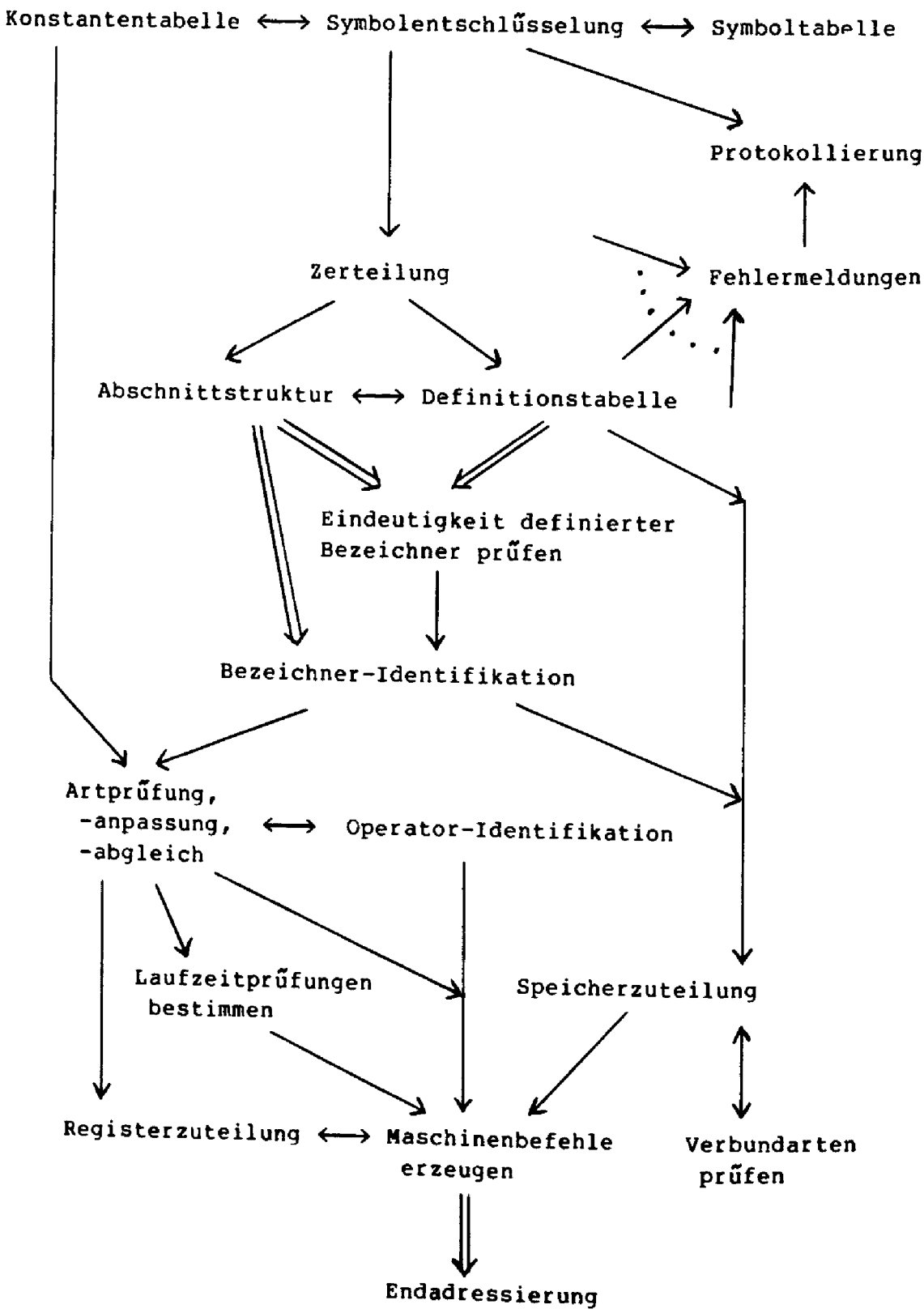

Abb. 1: Abhängigkeiten der Übersetzeraufgaben

Aufgaben	minimal	maximal	implementiert
Symbolentschl. Konstantentab. Symboltab.		1	1
Zerteilung	1	2	2
Definitionstab. Abschnittstruktur		3	3
Eindeutigkeit definierter Bezeichner prüfen Bezeichner-Identifikation		4	4
Artprüfung, -abgleich, -anpassung Operator-Identifikation		5	5
Laufzeitprüfungen bestimmen	2	6	
Speicherzuteilung Verbundarten prüfen		7	6
Maschinenbefehle erzeugen Registerzuteilung		8	7
Endadressierung	3	9	8
Protokollierung		10	9

Abb. 2: Lauf-Einteilungen

2.1 Lauf-Einteilung

Man erzielt eine logisch einfache Struktur des Übersetzers, wenn man der Lauf-Einteilung folgendes Schema, das auf den Abhängigkeiten beruht, zugrunde legt:

Für zwei Moduln A und B gilt:
- A und B gehören zu demselben Lauf, wenn sie wechselweise voneinander abhängen (A <--> B);
- A gehört zu demselben Lauf wie B oder zu einem früheren, wenn B von A abhängt (A --> B);
- A gehört zu einem früheren Lauf als B, wenn B von der vollstän-

digen Bearbeitung von A abhängt und für A ein Durchgang durch eine Zwischensprache nötig ist (A ==> B).

Diese Regeln lassen noch Freiheiten, die Läufe nach unterschiedlichen Entwurfskriterien einzuteilen, z.B. möglichst wenige Läufe mit wenigen Durchgängen durch Zwischensprachen oder möglichst kleine Läufe. Da wir uns von vornherein für einen Mehrlauf-Übersetzer entschieden haben, um dessen spezielle Problematik zu demonstrieren, kann die Anzahl der Läufe allein aufgrund von didaktischen und technischen Überlegungen bestimmt werden. Wir haben eine grosse Anzahl von Läufen (9) gewählt, die jeweils durch ein separates Programm implementiert werden (Abb. 2). Die Studenten kommen dann möglichst schnell zu abgeschlossenen und ablauffähigen Programmen. Der organisatorische Aufwand für Erweiterungen und Änderungen ist gering.

Diese Einteilung entspricht auch der thematischen Gliederung des Praktikums: Nach Abschluss eines Themas, z.B. Bezeichner-Identifikation, ist auch der zugehörige Lauf fertiggestellt. Ausserdem erhalten die Studenten durch die Lauf-Einteilung eine grosse Anzahl von Wiederaufsetzpunkten. Falls ihre Moduln nicht korrekt funktionieren, können sie den kompletten Lauf übernehmen. (Von dieser Möglichkeit wurde bisher kein Gebrauch gemacht.)

2.2 Übersetzer-Moduln und Schnittstellen

Ein zentrales Schema der Programmstrukturierung ist die Gliederung in Moduln, die der Aufgabenzerlegung entspricht. Eine grobe Unterscheidung führt zu zwei Klassen von Moduln: Den Programmoduln, die einen einheitlichen, abgeschlossenen Algorithmus implementieren, und den Datenmoduln, die mehrere einfachere Algorithmen (z.B. Zugriffsfunktion) mit der Datenstruktur, auf der sie operieren, zusammenfassen. (In [GK 77] wird dieser Aspekt genauer ausgeführt.)

Die Übersetzer-Tabellen (Symboltabelle, Konstantentabelle, Definitionstabelle) werden als selbständige Datenmoduln aufgefasst. Sie bestehen aus einer Datenstruktur - der Tabelle im engeren Sinne - und Prozeduren, die darauf operieren, z.B. Einträge generieren oder Bezeichner identifizieren im Falle der Definitionstabelle. Die Tabellen-Moduln fassen Daten zusammen, die in mehreren Läufen verwendet werden. Sie gehören deshalb zu den Schnittstellen zwischen den Läufen. Bei den Programm-Moduln steht der algorithmische Aspekt im Vordergrund. Sie transformieren Zwischensprachen (Symbolentschlüsselung, Zerteilung) oder erledigen kleinere algorithmische Teilaufgaben (Artabgleich, Operatoridentifikation).

Die einzelnen Übersetzer-Aufgaben sind so voneinander abgegrenzt, dass sich inhaltlich abgeschlossene Moduln mit einfachen Schnitt-

stellen ergeben. Da jeder Lauf nur aus wenigen, inhaltlich zusammengehörigen Moduln besteht, gibt es keine komplexen Schnittstellen innerhalb von Läufen. Die Schnittstellen zwischen den Läufen werden durch die Zwischensprachen beschrieben. Die erste Zwischensprache ist die Folge verschlüsselter Grundsymbole die der Symbolentschlüssler ausgibt. Alle weiteren Zwischensprachen sind Postfix-Darstellungen des Strukturbaumes. Die Struktur der Zwischensprache wird deshalb im wesentlichen durch die Grammatik der Quellsprache definiert. Ausserdem tragen die Tabellen-Moduln zu den Schnittstellen zwischen den Läufen bei.

3. Techniken des Übersetzerbaus

Es ist ein wesentliches Ziel des Praktikums, den Studenten moderne Techniken des Übersetzerbaus so zu vermitteln, dass sie in der Lage sind, die Techniken auch bei geänderter Aufgabestellung anzuwenden. Voraussetzung dafür ist, dass der Übersetzer nach seinen Aufgaben unabhängig von der übersetzten Sprache gegliedert ist. Nur dann erkennt man den wesentlichen Kern der Techniken, der auch auf anderen Anwendungen übertragbar ist. In manchen Fällen verhindert die Zerlegung des Übersetzers in Moduln, die jeweils ein Sprachelement bearbeiten, die Anwendung bestimmter moderner Techniken: Diese Gliederung führt z.B. zu Zerteilungsverfahren nach dem rekursiven Abstieg [Wi 77] - nicht aber zu tabellengesteuerten Verfahren.

In unserem Übersetzer werden systematische Verfahren zur Lösung der zentralen Aufgaben Symbolentschlüsselung, Zerteilung (syntaktische Analyse), semantische Analyse und Code-Erzeugung eingesetzt. Symbolentschlüssler entwickelt man heute unter Zugrundelegung von endlichen Automaten [JR68]; das Verfahren wird hier nicht weiter diskutiert.

3.1 Zerteilung

Die Auswahl eines Zerteilungsverfahrens wird wesentlich durch die verfügbaren Hilfsmittel bestimmt. Grundsätzlich ist ein tabellengesteuertes Zerteilungsverfahren der Programmierung eines Zerteilers nach der Methode des rekursiven Abstiegs vorzuziehen, falls ein Programmsystem verfügbar ist, das die Tabellen automatisch generiert. Die Anwendung tabellengesteuerter Verfahren hebt durch die modulare Trennung von syntaktischer und semantischer Analyse und Code-Erzeugung die verschiedenartige Behandlung unterschiedlicher Spracheigenschaften hervor. Mächtigere tabellengesteuerte Verfahren schränken die syntaktischen Eigenschaften der Sprache weniger ein, als die Methode des rekursiven Abstiegs.

Durch die automatische Erzeugung der Zerteilertabellen erhält man mit geringem Aufwand zuverlässige Zerteiler, die leicht gewartet und an Sprachänderungen angepasst werden können. Setzt man tabellengesteuerte Zerteilungsverfahren ein, so kann in den Zerteiler eine systematische Behandlung syntaktischer Fehler integriert werden, die gegenüber heuristischen Verfahren bessere Ergebnisse liefert und dadurch die Qualität des Übersetzers erhöht. Die Steuerungsdaten für die Fehlerbehandlung können sogar zusammen mit den Zerteilertabellen automatisch generiert werden [Rö 76].

In diesem Praktikum setzen wir einen Zerteiler ein, der nach der LALR(1)-Methode arbeitet. Aufgabe der Studenten ist es, die Grammatik der Sprache an die LALR(1)-Bedingung und die vorgegebenen Ein- und Ausgabe-Schnittstellen für den Zerteiler anzupassen. Durch die Implementierung des Zerteiler-Kerns, in dem die Zustandsübergänge durchgeführt werden, vertiefen sie ihr Verständnis für dieses Zerteilungsverfahren. Durch die Anwendung der LALR(1)-Methode werden sie ausserdem mit den speziellen Problemen der quellorientierten Zerteilung vertraut.

3.2 Semantische Analyse

Die semantische Analyse umfasst Übersetzeraufgaben wie Bezeichner-Identifikation, Artprüfung und Operator-Identifikation, die aus den Kontextabhängigkeiten der übersetzten Sprache resultieren. In den Lehrveranstaltungen und Lehrbüchern zum Übersetzerbau wird diesen Aufgaben häufig eine zu geringe Bedeutung beigemessen, und es werden keine systematischen Lösungen angeboten [Wi 77], [Gr 71]. Man geht davon aus, dass mit der syntaktischen Struktur alle wesentlichen Eigenschaften des Programms erfasst sind, und durch die Code-Erzeugung auf die Zielsprache abgebildet werden können. Diese Betrachtungsweise ist jedoch nicht richtig für moderne höhere Programmiersprachen (z.B. PASCAL) oder problemorientierte Anwendersprachen (z.B. Entwurfssprachen oder Abfragesprachen in Datenbanksystemen). In solchen Sprachen wird die Transformation der syntaktischen Sprachelemente in die Zielsprache wesentlich durch die Analyse der Kontextabhängigkeiten bestimmt.

Zur Bezeichner-Identifikation setzen wir ein systematisches Verfahren ein, das für alle in Programmiersprachen gebräuchlichen Gültigkeitsbereichsregeln anwendbar ist. Insbesondere sehen wir die Möglichkeit, dass ein Bezeichner vor seiner Definition angewandt auftritt, nicht als bedauerlichen Betriebsunfall der Sprachdefinition an, der heuristisch repariert wird, sondern wir lösen die Situation systematisch.

Trotz der in höheren Programmiersprachen üblichen Vielfalt von Regeln zu Arten und Artanpassungen kann man auch hier systematische

Lösungen angeben: Aufgaben wie Artabgleich, Ermitteln von Artanpassungssequenzen und Operatoridentifikation lassen sich modular abtrennen. Die Kontextbedingungen, die zu verschiedenen Sprachelementen Aussagen über Arten machen, können dann systematisch umgesetzt werden in Anwendungen solcher Funktionen auf bestimmte Teile des Strukturbaumes.

Eine weitere Systematisierung der semantischen Analyse kann durch Formalisierung der Kontextabhängigkeiten erzielt werden: Beschreibt man die Kontextabhängigkeiten durch eine attributierte Gramatik [Kn 68], so kann man die semantische Analyse auffassen als eine Vervollständigung des syntaktischen Strukturbaumes durch Attribute. Diese Aufgabe ist sogar tabellengesteuert lösbar [Ka 76]. Eine Weiterentwicklung des Praktikums in dieser Richtung wird angestrebt.

3.3 Code-Erzeugung

Die Code-Erzeugung für eine höhere Programmiersprache ist ein sehr komplexes Problem, das eine Reihe verschiedenartiger Teilaufgaben umfasst, für deren Lösung eine Vielzahl einzelner Verfahren anwendbar sind ([Gr 71], [Sch 75]). Nur wenige der Verfahren sind unabhängig von der Quellsprache und der Zielmaschine, z.B. Belegungsstrategien für die Registerzuteilung und die Abbildung von Objekten auf Speicheradressen. Da ausserdem bei den Teilnehmern des Praktikums keine vertieften Kenntnisse von Maschinensprachen vorausgesetzt werden können, ist es vernünftig, den Schwerpunkt der Ausbildung in diesem Bereich nicht auf spezielle Techniken zu legen, sondern eine Entwurfsmethode zu vermitteln:

Zunächst legt man zu jedem Sprachelement eine "Code-Sequenz" fest (siehe Abb. 3). Sie gibt an, wie das zu erzeugende Code-Stück aus einzelnen Befehlen und Code-Stücken zu Teilen des Sprachelements unter Berücksichtigung der Kontextabhängigkeiten zusammengesetzt wird. Die in einer formalisierten Sprache formulierten Code-Sequenzen werden dann systematisch in die Code-Erzeugungs-Algorithmen übertragen. Da dies eine reine Fleissaufgabe ist, die auch automatisch gelöst werden könnte, beschränkt man sich auf die exemplarische Umsetzung einiger Code-Sequenzen.

while Bedingung loop Anweisung repeat

```
neue Marke (M1)
Code für (Bedingung)
Bedingter Sprung nach M2
Code für (Anweisung)
Sprung nach (M1)
neue Marke (M2)
```

Abb. 3: Code-Sequenz für while-Schleifen

Dieses Vorgehen vereinfacht den Entwurf, verbessert die Überprüfbarkeit der Algorithmen und liefert mit den Code-Sequenzen einen wertvollen Beitrag zur Dokumentation. Es ist klar, dass in einer einsemestrigen Veranstaltung komplexe Aufgaben wie Optimierung, Endadressierung und Binden nicht behandelt werden können.

4. Implementierte Sprache

Programmiersprachen früherer Übersetzerbau-Praktika erfüllen nicht die didaktische Forderung nach hinreichender Komplexität. Die Anwendung der Techniken, die vermittelt werden sollen, muss in natürlicher Weise durch die Eigenschaften der gewählten Beispielsprache begründet sein. Anhand einer Sprache mit der trivialen Kontextabhängigkeiten lassen sich keine systematischen Verfahren zur semantischen Analyse demonstrieren. Ebenso kann die Anwendung eines LALR(1)-Zerteilers nicht vernünftig motiviert werden, wenn man eine Sprache implementiert, in der wie in BASIC alle Anweisungen mit einem Schlüsselwort beginnen.

Andererseits sollte die Sprache nur solche Elemente enthalten, deren Implementierung wichtige Erfahrungen vermittelt. Der Gesichtspunkt der praktischen Anwendbarkeit der Sprache spielt hier eine untergeordnete Rolle. Es reicht z.B. aus, eine einzige Schleifenform aufzunehmen und sich auf wenige Operatoren zu beschränken.

Wir sind ausgegangen von der Programmiersprache LEX [Go 75], die im wesentlichen dem in [BG 71] verwendeten ALGOL 68-Dialekt entspricht. Durch konsistentes Weglassen von Sprachelementen haben wir daraus die Sprache MINILEX entwickelt. Sie enthält nur noch solche Eigenschaften, die wir unter didaktischen Gesichtspunkten für notwendig halten:

Die elementaren Arten int, real, bool reichen aus, um Artanpassungen durchführen und bedingte Formeln formulieren zu können. Anhand freidefinierbarer Verbundarten und Reihungsarten zeigt man die verschiedenen Probleme bei der Übersetzung zusammengesetzter Datenobjekte. Blockstrukturen und Prozeduren mit Parametern sind grundle-

gende Elemente höherer Programmiersprachen, die die Anwendung wichtiger Analyse- und Synthese-Verfahren erfordern, z.B. Bezeichner-Identifikation und kellerartige Speicherorganisation.

Man hätte eine ähnliche Sprache, die unseren Anforderungen genügt, durch Vereinfachen von PASCAL erhalten können. Wir haben LEX vorgezogen, weil wir dafür eine klare und knappe Sprachdefinition verfügbar hatten.

5. Testhilfen

Die beim Entwurf des Übersetzers angewandten Konstruktionsprinzipien sorgen dafür, dass keine strukturbedingten Fehler vorkommen. Die systematische Entwicklung der Moduln und ihrer Schnittstellen aus den Teilaufgaben des Übersetzers gewährleisten eine hohe Zuverlässigkeit. Trotzdem muss ein Programmsystem, das modifiziert, weiterentwickelt und gewartet werden soll, von vornherein so geschrieben werden, dass Informationen über Ablauf und Dateninhalte zu Testzwecken ausgegeben werden.

Die Testinformation ist dann vernünftig anwendbar, wenn sie Begriffe und Struktur des Programms widerspiegelt. Ein wohlstrukturiertes, in einer höheren Programmiersprache geschriebenes Programm sollte auch klar gegliederte, gut lesbare Testausgabe produzieren - nicht aber einen hexadezimalen Speicherabzug. Da solche Forderungen nur selten von Sprachübersetzern und Laufzeitsystemen unterstützt werden, ist der Programmierer auf die Massnahmen angewiesen, die er selber in das Programm integriert. Die Testausgabe unseres Übersetzers genügt folgenden Forderungen:

- Der Rumpf jeder Prozedur beginnt mit der Ausgabe der Eingangsparameter und endet mit der Ausgabe der Ausgangsparameter und des Prozedurergebnisses.
- Das Einlesen und das Ausgeben von Zwischensprachelementen wird von den Läufen protokolliert.
- Anfang und Ende der zu einem Modul gehörigen Testausgabe werden deutlich hervorgehoben.
- In der Ausgabe erscheint kein Wert ohne Erklärung seiner Bedeutung oder einen Bezug auf den Programmtext (z.B. Parameterbezeichner).
- Die Testausgabe kann für jeden Modul separat dynamisch an- und abgeschaltet werden. Zu diesem Zweck werden zusätzliche Steuer-Elemente in die Zwischensprachen eingefügt.
- Die Programmteile, die Testausgabe erzeugen, können durch einfache Textsubstitution aus dem Programm entfernt werden. (Sie werden durch spezielle Kommandos bei der Übersetzung ausgeblendet oder in Kommentare umgewandelt.) Diese Eigenschaft ist notwendig, damit man zugleich eine Testversion und eine effi-

zientere Normalversion des Programms zur Verfügung hat, die immer gleich aktuell sind, obwohl nur eine Version gewartet wird.

Mit der strikten Anwendung dieser Regeln haben wir den Studenten demonstriert, wie man ein Programmsystem so instrumentieren kann, dass es von anderen als dem Autor verstanden, weiterentwickelt und getestet werden kann.

6. Beispiel

Am Beispiel der Bezeichner-Identifikation (Lauf 4) zeigen wir die Gliederung eines Laufes in Moduln, die fertig vorgegeben bzw. von den Studenten implementiert werden. In diesem Lauf werden in einem Durchgang durch die Zwischensprache (Strukturbaum in Postfix-Darstellung) Bezüge in die Symboltabelle, die Bezeichner repräsentieren, durch Bezüge in die Definitionstabelle auf die jeweils gültige Definition ersetzt. Die Definitionstabelle wird im Lauf 3 vollständig aufgebaut und steht hier zur Verfügung.

Die hier angewandte Methode löst systematisch die Identifikation insbesondere für blockstrukturierte Programmiersprachen mit geschachtelten Gültigkeitsbereichen: Zu jedem im Programm auftretenden Bezeichner wird ein Keller angelegt, der die Definitionen des Bezeichners aufnimmt. Die Keller werden durch Verkettung der Definitionen als Listen implementiert. Man sorgt dafür, dass beim Durchgang durch den Strukturbaum zu jedem Zeitpunkt die gerade gültigen Definitionen als oberste Elemente in den Kellern stehen. Damit ergeben sich drei zentrale Aufgaben:

- Kellern aller lokalen Definitionen beim Eintritt in einen Abschnitt,
- Entkellern aller lokalen Definitionen beim Verlassen eines Abschnitts und
- Identifizieren eines angewandt auftretenden Bezeichners mit der obersten Definition im zugehörigen Keller.

Bei der Implementierung dieser Aufgaben durch drei Prozedurmoduln lernen die Studenten das Verfahren kennen und anwenden. Der gesamte Rahmen, in den die Moduln eingebettet werden, wird fertig vorgegeben. Er enthält im wesentlichen Programmteile, deren Implementierung recht aufwendig ist aber keine neuen Erkenntnisse vermittelt, z.B. Prozeduren und Vereinbarungen zum Lesen der Definitionstabelle und zur Ein- und Ausgabe der Zwischensprachen. Die Grobstruktur des Identifikations-Laufes ist in Abb. 4 dargestellt.

```
Vereinbarungen

procedure dtnachtrag;
begin (*Nachvereinbarungen im Fehlerfall*) end;

procedure dtlesen;
begin  (*DT einlesen *) end;

procedure identifikation;

   procedure ausidentvekt (dtbez: dtbezug);
   begin (*Entkellern einer Liste von Definitionen*) end;

   procedure inidentvekt (dtbez: dtbezug);
   begin (*Kellern einer Liste von Definitionen*) end;

   function identifizierebez (stbez: stbezug;
                              bzn: dtbzn): dtbezug;
   begin (*Bezeichner identifizieren*) end;

   procedure identifarten (dtel: dtbezug);
   begin (*identifiziert Artangaben in Definitionen*) end;

begin (*identifikation*)
  repeat (*SB-Durchlauf*)
    case kntyp of
    bezeichner:    identifizierebez;
    abschnittanf:  inidentvekt(lok. Obj.);
                   identifarten(lok. Obj);
    abschnittende: ausidentvekt(lok. Obj.)
    end
  until eof
end

begin (*lauf 4*) end.
```

Die durch | gekennzeichneten Teile werden von den Studenten implementiert.

Abb. 4: Struktur des 4. Laufes (Bezeichner-Identifikation)

7. Zusammenfassung

Für Informatiker ist es wichtig, systematische Methoden des Übersetzerbaus kennenzulernen und in der Praxis zu erproben. Die Programmiersprache, die in einem Übersetzerbau-Praktikum implementiert wird, muss so komplex sein, dass die Aufgaben nur unter Anwendung systematischer Techniken und Methoden der Programmkonstruktion gelöst werden können. Der Arbeitsaufwand für den einzelnen Studenten kann auf ein vernünftiges Mass reduziert werden, wenn der Übersetzer modular gegliedert ist, und die Studenten nur einzelne Moduln mit zentraler Bedeutung implementieren und in fertige Rahmenprogramme integrieren.

Im Sommersemester 1977 haben wir ein Praktikum in dieser Weise erfolgreich durchgeführt. Von 30 Studenten, die in 10 Dreier-Gruppen arbeiteten, haben 8 Gruppen bis zum Ende durchgehalten, 7 Gruppen stellten einen lauffähigen Übersetzer fertig. Dieses Ergebnis ist ungewöhnlich hoch im Vergleich zu der "Schwundrate" üblicher Übungen und unter Berücksichtigung grosser Schwierigkeiten beim Rechnerzugang. Die Erfahrungen dieser Veranstaltung werden wir ausnutzen, um für spätere Wiederholungen die Aufgabestellungen zu präzisieren und die Implementierung zu verbessern.

Prof. Dr. G. Goos und meine Kollegen Dr. P. Kammerer, H. Neugebauer und Dr. H. Rohlfing haben sich wesentlich an der Entwicklung und Durchführung des Praktikums beteiligt. Ihnen gebührt mein aufrichtiger Dank.

8. Literatur

BG 71 Bauer, F.L., Goos, G., Informatik, Eine einführende Übersicht, Heidelberger Taschenbücher Bd. 80 und 91, Springer, 1971

GK 77 Goos, G., Kastens, U., Programming Languages und the Design of Modular Programs, Proc. of the Working Conference on Constructing Quality-Software, Novosibirsk 1977

Go 75 Goos, G., Die Programmiersprache LEX, Fak. f. Informatik, Universität Karlsruhe, Int. Bericht Nr. 1, 1975

Gr 71 Gries, D., Compiler Construction for Digital Computers, Wiley 1971

JR 68 Johnson, W.L., Ross, D.T., Automatic Generation of Efficient Lexical Processors Using Finite State Techniques, CACM 11, 805-813, 1968

Ka 76 Kastens, U., Ein Übersetzer-erzeugendes System auf der Basis attributierter Grammatiken, Fak. f. Informatik, Universität Karlsruhe, Int. Bericht Nr. 10, 1976

Kn 68 Knuth, D.E., Semantics of context-free languages, in Math.Syst.Th. 2, 127-145, 1968 und Math.Syst.Th. 5, 95, 1971

Rö 76 Röhrich, J., Syntax-error Recovery in LR-Parsers, in Informatik Fachberichte, Bd. 1, Springer, 1976

Sch 75 Schneider, H.J., Compiler - Aufbau und Arbeitsweise, W. de Gruyter, Berlin 1975

Wi 77 Wirth, N., Compilerbau, Teubner Studienbücher Informatik, Bd. 36, Stuttgart 1977