

**Konzept eines Generierungssystems  
für Codeerzeugung : Die Übertragung  
des p-Codes auf wechselnde Zielmaschinen**

Diplomarbeit  
von  
Alexander Demmer

Angefertigt nach einem Thema von Prof. Reinhard Wilhelm  
am Fachbereich 10 der Universität des Saarlandes,  
Saarbrücken

## Zusammenfassung

Wir wollen P-Code als Ausgabe des Pascal P-Compilers auf wechselnde Zielmaschinen übertragen. Zu diesem Zweck wollen wir ein "back-end" bestehend aus einem auf Mustererkennung basierenden Codeerzeuger und einem Peephole-Optimierer konstruieren.

Diese Arbeit stellt ein Ansatz für die Generierung der Muster des Codeerzeugers dar.

P-Code und Zielmaschinen-Instruktionen werden von einem Generator, der ebenfalls wie der Codeerzeuger der Übersetzungszeit Mustererkennungsalgorithmen verwendet, analysiert. Ausgabe dieses Generators sind Muster vom Typ (Folge von P-Code Befehlen; Folge von Zielmaschinen-Instruktionen). Das Einfügen solcher Muster im "back-end" verspricht eine Verkürzung der Übersetzungszeit.

Das Generierungsverfahren sichert sowohl das Nicht-Blockieren des Codeerzeugers als auch die Konstruktion effizienter Muster.

## Inhaltsverzeichnis

|  |    |
|--|----|
| <u>Teil I</u> .....  | 1  |
| Einführung .....   | 1  |
| <u>Kap. 1 Grundlegende Begriffe</u> .....                                    | 3  |
| 1.1 Monolithische Compiler .....   | 3  |
| 1.2 Das UNCOL-Konzept .....  | 4  |
| 1.3 Maschinenunabhängige Compiler; Übertragbarkeit und<br>Portabilität ..... | 5  |
| 1.4 Compilerstruktur .....   | 8  |
| 1.4.1 Strukturvergleich monolithisch-portabler Compiler .....                | 8  |
| 1.4.2 Synthese .....   | 10 |
| 1.5 Compilerentwicklung .....  | 12 |
| 1.5.1 Compilerbaumodell .....  | 12 |
| 1.5.2 Implementierung .....  | 13 |
| <u>Kap. 2 Codeerzeugung; Übersicht</u> .....                                 | 15 |
| 2.1 Aufgabenbereich .....  | 15 |
| 2.2 Anforderungen .....  | 15 |
| 2.3 Probleme .....   | 17 |
| 2.4 Konventionelle Codeerzeugung .....                                       | 19 |
| 2.5 Übergabe der Kontextinformation .....                                    | 23 |
| <u>Kap. 3 Automatisierung der Codeerzeugung</u> .....                        | 26 |
| 3.1 Motivation und Entstehung .....  | 26 |
| 3.2 Beschreibung der Instruktionen der Zielmaschine .....                    | 27 |
| 3.3 Codeerzeugung als "pattern matching" .....                               | 28 |
| 3.4 Das Verfahren von Graham und Glanville .....                             | 31 |
| 3.4.1 Überblick .....  | 31 |
| 3.4.2 Tafelkonstruktion; "maximal munch"-Strategie .....                     | 33 |

|   |    |
|---|----|
| 3.4.3 Weitere Entscheidungsprobleme der Codeauswahl .....   | 34 |
| 3.4.4 Registerverteilung .....  | 37 |
| 3.4.5 Beseitigung der Blockierungssituationen .....   | 37 |
| 3.4.6 Schlußfolgerungen .....   | 37 |
| 3.5 Über Automatisierung .....  | 38 |
| <u>Teil II</u> .....  | 40 |
| II.1 Einführung .....   | 40 |
| II.2 Generierungsverfahren; Konstruktion der<br>Transformationstupel .....                            | 42 |
| <u>Kap.4 Codeerzeuger-Eingabe; P-Code als Schnittstelle zwischen<br/>front-end und back-end</u> ..... | 47 |
| 4.1 Die P-Maschine .....  | 47 |
| 4.2 P-Code Regeln; explizite Darstellung von P-Code Befehlen .....                                    | 48 |
| 4.3 P-Code Hilfs- und Echknoten .....   | 50 |
| 4.4 P-Code Transferregeln .....   | 50 |
| 4.4.1 P-Code Regeln vom Typ "load" .....  | 50 |
| 4.4.2 P-Code Regeln vom Typ "store" .....   | 52 |
| 4.5 P-Code Eingabebäume .....   | 53 |
| 4.6 Anforderung an das front-end .....  | 54 |
| <u>Kap.5 Eingabe des Konstruktionsalgorithmus für die Hilfstupel</u> .....                            | 56 |
| 5.1 Einführung .....  | 56 |
| 5.2 P-Spezifikation; Abbildung des Stack-computers auf der<br>Abstrakten Register Maschine .....      | 56 |
| 5.2.1 Definition der Abstrakten Register Maschine .....   | 56 |
| 5.2.2 P-Spezifikationstupel .....   | 57 |
| 5.2.2.1 P-Spezifikationstupel für P-Code Regeln vom Typ "load" .....                                  | 57 |
| 5.2.2.2 P-Spezifikationstupel für P-Code Regeln vom<br>Typ "store" .....                              | 59 |
| 5.3 mc-Spezifikation für den SAB 8080-Prozessor .....   | 60 |
| 5.4 Primitive Substitutionsfunktion .....   | 63 |
| <u>Kap.6 Konstruktion der Substitutionstupel</u> .....  | 66 |
| 6.1 Baumrepräsentation .....  | 66 |

|   |     |
|---|-----|
| 6.2 Regelrepräsentation .....   | 68  |
| 6.3 Erweiterte Substitutionsfunktion; Baumhomomorphismus .....                                  | 69  |
| 6.4 KRM-Metaregeln; Expansion eines P-Code Befehls .....  | 70  |
| 6.5 Codeauswahl für die KRM-Metaregeln .....  | 71  |
| 6.6 Konstruktion der "1-zu-1" Substitutionstupel .....  | 72  |
| 6.7 Konstruktion der Transformationstupel (p;mc);<br>Explizite Darstellung .....                | 74  |
| 6.8 Konstruktion der "1-zu-n" Substitutionstupel; Automatische<br>Makroexpansion .....          | 75  |
| <u>Kap.7 Konstruktion der Abstraktionstupel</u> .....   | 77  |
| 7.1 Einführung .....  | 77  |
| 7.2 Primitive Abstraktionsfunktion .....  | 77  |
| 7.3 Erweiterte Abstraktionsfunktion; Baumhomomorphismus .....                                   | 78  |
| 7.4 ARM-Metaregel; Expansion einer Maschineninstruktion .....                                   | 78  |
| 7.5 Codeauswahl für die ARM-Metaregeln .....  | 79  |
| 7.6 über den Definitionsbereich der primitiven<br>Substitutions- und Abstraktionsfunktion ..... | 81  |
| 7.7 Verhinderung der Blockierungsmöglichkeiten<br>des Codeerzeugers .....                       | 83  |
| 7.8 Interpretation des Begriffs "Treffer" .....   | 84  |
| 7.9 Substitutions- und Abstraktionstupel; Zusammenfassung .....                                 | 84  |
| <u>Kap.8 Anforderungen an den G- und C-Executor</u> .....                                       | 87  |
| 8.1 Faktorisierung der Hilfstupel; Äquivalente Hilfstupel .....                                 | 87  |
| 8.2 Eliminierung der redundanten "1-zu-1" Transformationstupel .....                            | 88  |
| 8.3 Konstruktion von effizienten Patterns; "Minimal munch"-<br>Heuristik .....                  | 90  |
| 8.4 Transformationstupel vom Typ "m-zu-n" .....   | 92  |
| 8.5 Semantische Einschränkungen .....   | 93  |
| 8.6 Semantische Betrachtungen bei der Codeauswahl .....   | 94  |
| 8.7 Behandlung der Register-Register Transferinstruktionen .....                                | 95  |
| 8.8 Einsatz von 3-Adress-Instruktionen .....  | 98  |
| 8.9 Exotische Maschineninstruktionen .....  | 100 |
| <u>Kap.9 Generator-Schema</u> .....   | 102 |

|   |     |
|---|-----|
| 9.1 Einführung .....                                  | 102 |
| 9.2 P-Spezifikation von bedingten Sprüngen .....      | 103 |
| 9.3 Behandlung des Restkellers in ARM .....           | 104 |
| 9.3.1 Prozeduraufrufe .....                           | 104 |
| 9.3.2 Parameterberechnung und -übergabe .....         | 106 |
| 9.3.3 Funktionsaufrufe .....                          | 106 |
| 9.4 Spezifikation von komplexen P-Code Befehlen ..... | 108 |
| Schlußfolgerungen .....                               | 111 |
| Literatur .....                                       | 112 |

## Teil I

### Einführung

Lange Zeit ist die Entwicklung der automatischen Codeerzeugung und Codeoptimierung ein gegenüber der syntaktischen Analyse wenig berücksichtigtes Gebiet geblieben. Erst in den letzten Jahren hat die Automatisierung der Codeerzeugung und Codeoptimierung wichtige Ergebnisse erzielt. Vor allem Resultate aus der syntaktischen Analyse und aus der Datenflußanalyse sind in diesem Bereich erfolgreich angewendet worden. Über dieses Thema ist viel Material veröffentlicht worden. Je nach dem verfolgten Ziel und den vorhandenen Werkzeugen hat fast jeder Autor seine eigenen Begriffe und Notationen eingeführt.

Einige Compilerentwickler (Cattel, Ganapathi, Lunell) haben Klassifizierungen und Vergleiche der automatischen Codeerzeugungstechniken angestellt. Diese Autoren haben sich meist auf die Methoden ihrer Vorgänger beschränkt und daraus die Aspekte diskutiert, die für ihre eigene Arbeit relevant waren. Unter diesen Arbeiten nimmt das Werk von Hans Lunell "Code Generator Writing Systems" eine Sonderstellung ein. Verschiedene Automatisierungstechniken werden ausführlich diskutiert, klassifiziert und miteinander verglichen. Ein großes Verdienst von Lunell ist, daß er die oft verschiedenen Begriffe und Notationen auf "einen gemeinsamen Nenner" gebracht hat.

Wir haben zum Teil Notationen und Begriffe aus der Untersuchung von Lunell übernommen, soweit dies für unsere Zwecke sinnvoll erschien. Unsere Absicht war, Teil II dieser Arbeit soweit wie möglich von Diskussionen und Kommentaren zu befreien, die schon zu einem früheren Zeitpunkt erklärbar gewesen wären. Daher wird der Leser bei der Lektüre des zweiten Teils auf bestimmte Bemerkungen und Beispiele aus Teil I verwiesen.

Der erste Teil dieser Arbeit umfaßt verschiedene Aspekte, die für die Darstellung unseres Codeerzeugungssystems wichtig sind. Alle eingeführten Begriffe werden durch Beispiele illustriert.

Es werden vorgestellt:

Kap.1

- der Rahmen, in dem unser Konzept zu lokalisieren ist.

Kap.2

- die allgemeine Problematik der Codeerzeugung und

Kap.3

- die Automatisierungsprobleme der Codeerzeugung anhand des Verfahrens von R.Glanville.



## KAP.1 Grundlegende Begriffe

### 1.1 Monolithische Compiler

Ein **Compiler** ist ein Programm, das Programme aus einer höheren Programmiersprache  $S$  in den Maschinencode  $M$  eines Rechners übersetzt. Dieser Vorgang wird schematisch wie folgt dargestellt:

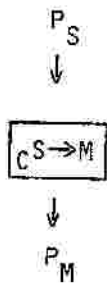


Abb.1.1 Compiler; Schematische Darstellung

Die Abkürzungen aus Abb.1.1 bedeuten:

$P_S$  : Programm in einer höheren Programmiersprache  $S$

$P_M$  : Programm in Maschinencode  $M$

$C^{S \rightarrow M}$  : Compiler, der von  $S$  nach  $M$  übersetzt

Bezogen auf den Compilierungsprozeß wird die Compilereingabe,  $P_S$ , auch **Quellprogramm** (source program) genannt. Die Compilerausgabe,  $P_M$ , wird **Objektprogramm** (target program) genannt. Die Attribute "**statisch**" (static) bzw. "**dynamisch**" (dynamic) bezeichnen Vorgänge die zur Compilezeit bzw. Laufzeit stattfinden. Compiler sind komplexe, umfangreiche Programme. In der Regel wird die Compilierungsaufgabe durch Modularisierung in einzelne Phasen zerlegt. Grob gesehen werden zwei Teile unterschieden: **Analyse** und **Synthese**. Die Analyse prüft, ob das Quellprogramm syntaktisch korrekt ist. Die Synthese erzeugt das Objektprogramm. Die Schnittstelle zwischen Analyse und Synthese wird durch eine **Zwischensprache** (intermediate code) beschrieben (vorausgesetzt es handelt sich um keinen 1-Pass Compiler). Entsprechend seiner Position innerhalb des Compilierungsprozesses bezeichnet man das Analyse-Modul als **front-end** und das Synthese-Modul als **back-end**. Compiler, die nur für eine einzige

höhere Programmiersprache und eine einzige Zielmaschine entwickelt worden sind, werden **monolithische** (monolithic) Compiler genannt. Möchte man  $m$  höhere Programmiersprachen  $S_1, \dots, S_m$  auf  $n$  Zielmaschinen,  $ZM_1, \dots, ZM_n$ , mit den zugehörigen Maschinencodes  $M_1, \dots, M_n$ , übersetzen lassen, so bedeutet dies, daß man  $n \times m$  monolithische Compiler

$$C_{S_1 \rightarrow M_1}, \dots, C_{S_1 \rightarrow M_n}, \dots, C_{S_m \rightarrow M_1}, \dots, C_{S_m \rightarrow M_n},$$

zu schreiben hat.

### 1.2 Das UNCOL-Konzept

Um den oben erwähnten Aufwand zu reduzieren, hat man versucht Bestandteile (Module) des Compilers zu entwickeln, die zu mehreren Zielmaschinen und Programmiersprachen passen. Die wichtigsten Denkanstöße in dieser Richtung hat das UNCOL -Projekt geliefert. UNCOL, Abk. für Universal Computer Language, ist der Name einer universellen Zwischensprache, die als Schnittstelle zwischen einem maschinenunabhängigen, sprachspezifischen front-end und einem maschinenspezifischen, sprachunabhängigen back-end dient. UNCOL selbst soll sprach- und maschinenunabhängig sein und somit gültig für sämtliche höhere (imperative) Programmiersprachen und Zielmaschinen.

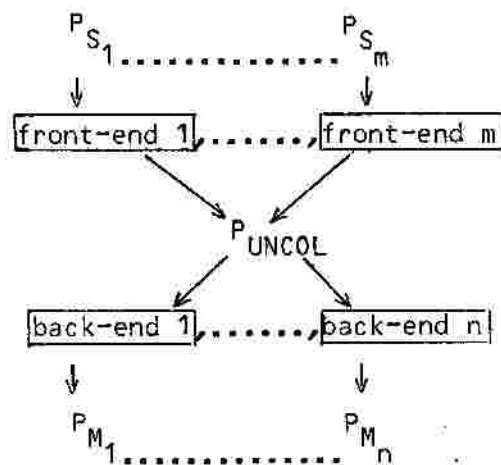


Abb. 1.2: Das UNCOL-Konzept

Aus der Kombination der  $m$  front-ends,  $fe_1, \dots, fe_m$ , und  $n$  back-ends,  $be_1, \dots, be_n$ , lassen sich alle beliebigen monolithischen Compiler,

$$C_{S_1 \rightarrow M_1}, \dots, C_{S_1 \rightarrow M_n}, \dots, C_{S_m \rightarrow M_1}, \dots, C_{S_m \rightarrow M_n}$$

zusammenstellen. Will man z.B. den Compiler gewinnen, der von  $S_i$ ,  $i \in [1, \dots, m]$ , nach  $M_j$ ,  $j \in [1, \dots, n]$ , übersetzt, dh.,  $C_{S_i \rightarrow M_j}$ , so wird das back-end  $j$  dem front-end  $i$  nachgeschaltet. Damit braucht man nur  $m+n$  ( $m$  front-ends +  $n$  back-ends) "Halbcompiler" zu schreiben statt  $m \cdot n$  Compiler, um  $m$  höhere Programmiersprache auf  $n$  Zielmaschinen zu übertragen. Der UNCOL-Versuch ist wegen der großen Unterschiede zwischen den höheren Programmier- bzw. Maschinensprachen gescheitert.

### 1.3 Maschinenunabhängige Compiler; Übertragbarkeit und Portabilität

Ausgelöst von den UNCOL-Ideen hat sich ein Gedanke im Compilerbau durchgesetzt: die Konstruktion eines Compilers aus einem maschinenunabhängigen front-end und einem maschinenabhängigen back-end. Damit läßt sich die Übersetzung einer höheren Programmiersprache  $S$  in die Maschinencodes  $M_1, \dots, M_n$  von  $n$  Zielmaschinen wie folgt vereinfachen: das gleiche front-end wird für alle Zielmaschinen eingesetzt; das back-end wird für jede Zielmaschine separat entwickelt. Abb.1.3.1 stellt diese Situation schematisch dar. Die Abkürzungen  $ZS$  und  $P_{ZS}$  stehen für Zwischensprache bzw. Programm in der Zwischensprache.

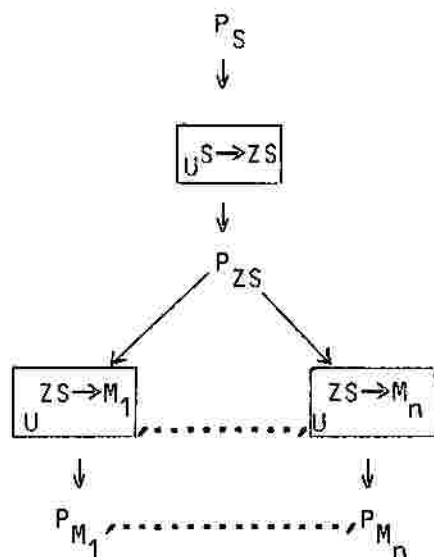


Abb.1.3.1: Übersetzung mittels eines maschinenunabhängigen front-ends

Wird die Zwischensprache ZS als Code eines abstrakten Rechners angesehen, so kann man das front-end als einen maschinenunabhängigen Übersetzer,  $U^{S \rightarrow ZS}$ , auffassen, wie auf Abb.1.3.1 zu sehen ist. Dementsprechend werden die back-ends als Übersetzer von der Zwischensprache ZS in die jeweiligen Maschinencodes,  $M_1, \dots, M_n$ , aufgefasst. Zwei bekannte Beispiele von maschinenunabhängigen Compilern sind der PASCAL [P] Compiler [Nori Amm 74] und der portable BCPL Compiler [Ri 71]. Zwei Begriffe sind in diesem Zusammenhang von Bedeutung: **Übertragbarkeit** (re-targetability) und **Portabilität** (portability).

"Übertragbarkeit" bezieht sich auf die vorher angesprochene Situation aus Abb.1.3.1: ein Compiler,  $C^{S \rightarrow M_j}$ , liegt vor,  $j \in [1, \dots, n]$ .

Frage: Kann man den vorhandenen Compiler,  $C^{S \rightarrow M_j}$ , so ändern, daß er Maschinencode  $M_h$  für eine neue Zielmaschine  $ZM_h$ ,  $h \in [1, \dots, n]$ , liefert?

#### Definition 1.3.1 (Übertragbarkeit)

Ist dies möglich, so sagt man daß  $C^{S \rightarrow M_j}$  von der Zielmaschine  $ZM_j$  auf die Zielmaschine  $ZM_h$  **übertragbar** (re-targetable) ist.

Werden die maschinenabhängigen Teile eines Compilers von den restlichen Teile getrennt gehalten, wie im Abb.1.3.1, so bedeutet dies Änderung des Compilers  $C^{S \rightarrow M_j}$  zu  $C^{S \rightarrow M_h}$ , das back-end  $U^{ZS \rightarrow M_j}$  durch das back-end  $U^{ZS \rightarrow M_h}$  zu ersetzen.

Neben der Quell- und Zielsprache S bzw. M wird auch die Sprache L, in der ein Compiler geschrieben ist, angegeben, i.Z.  $C_L^{S \rightarrow M}$ . Meist ist L eine höhere Programmiersprache. Vor dem Einsatz auf einer Zielmaschine ZM wird der Compiler von der Sprache L in den Maschinencode M übersetzt, i.Z.,  $C_L^{S \rightarrow M} \rightarrow C_M^{S \rightarrow M}$ .

Im Rahmen dieser Arbeit nehmen wir an, daß wir nur mit "startreifen" Compilern  $C_M^{S \rightarrow M}$  zu tun haben. Der untere Index M aus  $C_M^{S \rightarrow M}$  bzw. L aus  $C_L^{S \rightarrow M}$  bezeichnet die **Implementierungssprache** des Compilers.

Einen Compiler zu übertragen impliziert keine Änderung der Implementierungssprache des Compilers. Überträgt man einen Compiler  $C_{M_j}^{S \rightarrow M_j}$  von einem Rechner  $ZM_j$  mit dem zugehörigen Maschinencode  $M_j$  auf einen Rechner  $ZM_h$  mit Maschinencode  $M_h$ , so liefert der neue Compiler  $C_{M_j}^{S \rightarrow M_h}$  Maschinencode für den neuen Rechner  $ZM_h$ , läuft aber weiterhin auf der Maschine

ZM<sub>j</sub>.

Definition 1.3.2 (Cross-Compiler)

Ein Compiler, der auf einer Maschine läuft und Zielcode für eine andere Zielmaschine erzeugt, nennt man **Cross-Compiler**.

Definition 1.3.3. (Portieren)

Wird ein Compiler  $C_M^{S \rightarrow M}$  zu  $C_{M'}^{S \rightarrow M}$  geändert, so sagt man, daß der Compiler  $C_M^{S \rightarrow M}$  von der Zielmaschine M auf die Zielmaschine M' **portiert** (to port) wurde.

Dabei ist es unwichtig, welche Übersetzungsaufgabe der Compiler zu erledigen hat. Auch ein Cross-Compiler kann portiert werden.

Definition 1.3.4 (Bootstrapping)

Will man einen, auf einem Rechner ZM laufenden Compiler  $C_M^{S \rightarrow M}$  auf einem Rechner ZM' zum Einsatz bringen, so bedeutet dies  $C_M^{S \rightarrow M}$  zu  $C_{M'}^{S \rightarrow M'}$  zu ändern. Dieser Vorgang heißt **Bootstrapping**.

Bootstrapping wird in der Regel mittels zweier Schritte realisiert: Übertragen, gefolgt von Portieren des vorhandenen Compilers. Abb.1.3.2 illustriert diesen Vorgang:

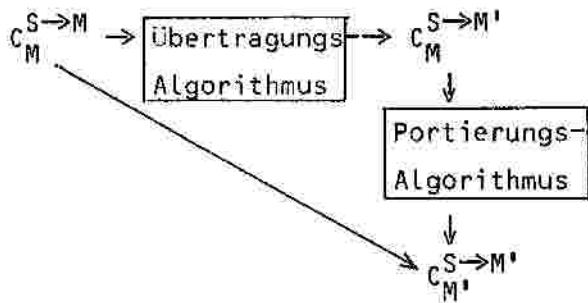


Abb.1.3.2: Compiler-Bootstrapping

Portabilität bezieht sich auf Software allgemein, Übertragbarkeit dagegen nur auf Übersetzer.

In dieser Arbeit beschäftigen wir uns mit Aspekten der Compilerübertragung entsprechend des Schemas aus Abb.1.3.1. Unser front-end ist der PASCAL [P] Compiler, P-Code ist die Zwischensprache ZS. Wie die maschinenspezifischen Teile des back-ends generiert werden, berichten wir im Teil 2 dieser Arbeit.

## 1.4 Compilerstruktur

### 1.4.1 Strukturvergleich monolithisch-portabler Compiler

In diesem Abschnitt stellen wir ein Compilermodell vor, das unseren Ansatz der Codeerzeugung unterstützt. Im weiteren Verlauf dieser Arbeit setzen wir voraus, daß das front-end und die Zwischensprache maschinenunabhängig und sprachspezifisch sind, wie in Abb.1.3.1 dargestellt.

Die zwei Hauptteile eines Compilers, Analyse und Synthese, lassen sich ihrerseits in kleinere Aufgaben, auch **Phasen** genannt, unterteilen. Abb.1.4.1.1 zeigt den Aufbau des front-ends, Abb.1.4.2 den des back-ends. Auf die Rolle der Analyse gehen wir nicht näher ein. Wir beschränken uns nur auf einige Bemerkungen, die verdeutlichen sollen, inwieweit sich die Struktur von portablen und monolithischen Compilern unterscheidet.

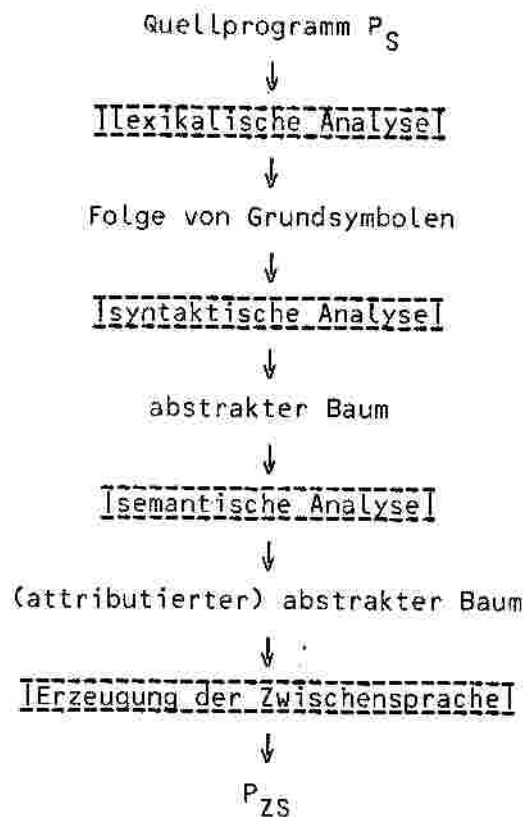


Abb.1.4.1.1: Analyse

Die "Erzeugung der Zwischensprache" wird in den meisten Darstellungen über Struktur von monolithischen Compilern als erster Schritt der Synthese betrachtet [Zi 83]. Häufig werden in dieser Phase die ersten

Maschinenabhängigkeiten eingebaut. Glanville [Gl 77] nennt diese Phase sogar "maschinenabhängige Übersetzung" (machine dependent translation). Für unsere Zwecke, d.h. Übertragung der Zwischensprache auf wechselnde Zielmaschinen, ist es wichtig, die Zwischensprache (P-Code) frei von Maschinenabhängigkeiten zu halten. Um die Unterschiede zwischen der Auffassung über Compileraufteilung im Falle des monolithischen und portablen Ansatzes zu verdeutlichen, haben wir in Abb.1.4.1.2 ein vereinfachtes Compilermodell nach einer Darstellung von Zima [Zi 83] wiedergegeben.

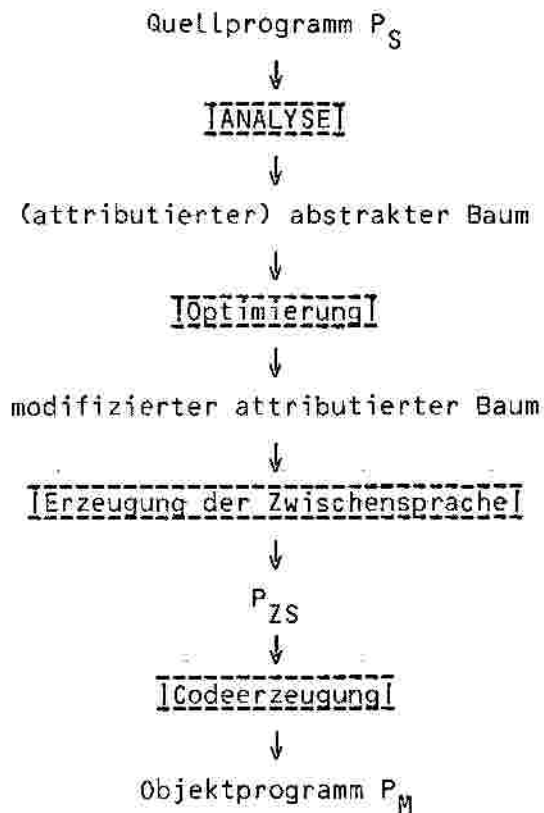


Abb.1.4.1.2: Struktur eines monolithischen Compilers

### 1.4.2 Synthese

Für die Synthese haben wir folgende Einteilung vorgenommen:

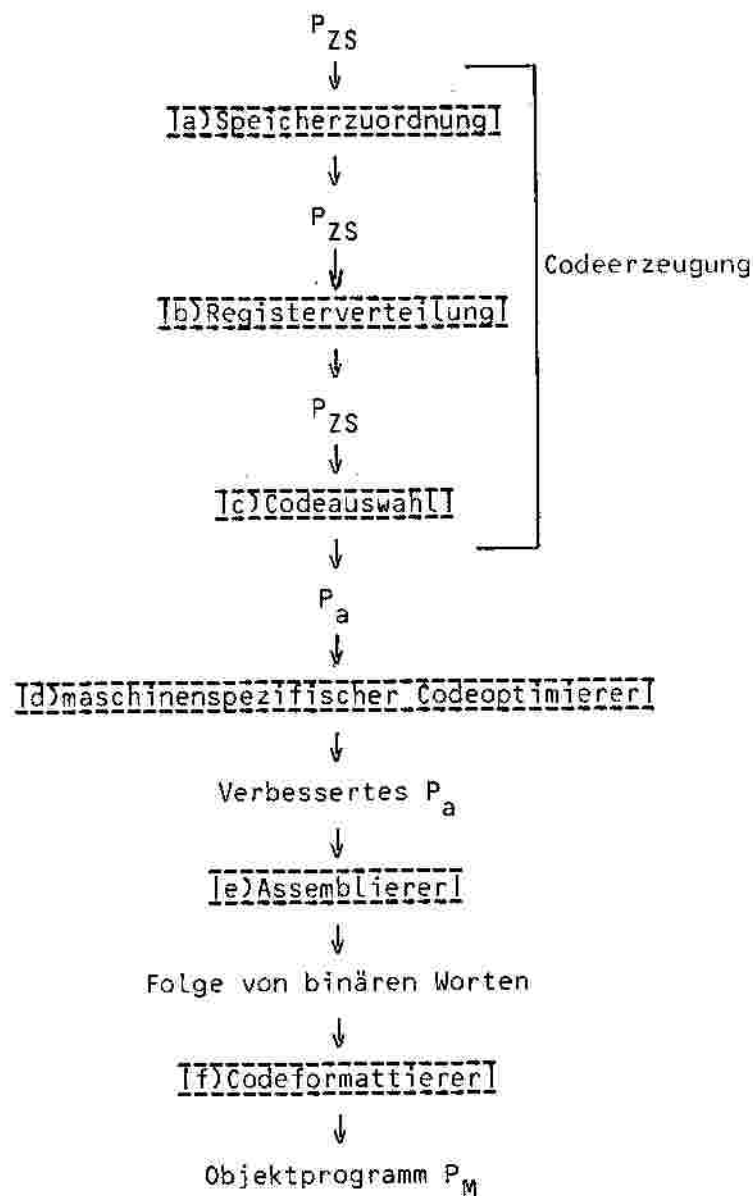


Abb.1.4.2: Synthese

Die Abkürzung  $P_a$  steht für Programm in Assembler, das wir als Objektcode betrachten.

Nachfolgend werden die Synthesephasen kurz erläutert.

- a) **Speicherzuordnung** (storage mapping)  
Diese Phase legt fest:



-wie Datenstrukturen der Zwischensprache ZS im Maschinencode M der Zielmaschine dargestellt werden. Hier wird z.B. bestimmt, daß ein integer-Abschnitt als Halb- oder Ganzwort repräsentiert wird.

-wie diese Datenstrukturen auf der Zielmaschine gespeichert werden. Hier wird z.B. entschieden, ob Feldkomponenten hintereinander oder mittels Randvektoren (edge vectors) gespeichert werden.

-wie die Zugriffspfade zu den gespeicherten Daten berechnet werden. Jetzt können den Variablen Adressen zugeteilt werden, da deren Platzbedarf bekannt ist. Prinzipiell kann dieser Vorgang unmittelbar nach Analyse der Variablen-Deklarationen erfolgen.

b) **Registerverteilung** (register management)

-bestimmt, welche Register der Zielmaschine welchen Objekten der Zwischensprache zugeordnet werden und wie diese Zuordnung verwaltet wird.

c) **Codeauswahl** (code selection)

-Die Codeauswahlphase entscheidet, welche Maschineninstruktionen ausgesucht werden, um das Programm in der Zwischensprache ZS auszuführen. In vielen automatischen Ansätzen über Automatisierung der Compilersynthese wird diese Phase als die Codeerzeugungs-Aufgabe betrachtet (siehe nächsten Abschnitt).

d) **maschinenspezifische Codeoptimierung**

Liegt das Programm in Maschinencode vor, so kann es verbessert werden. Die Notwendigkeit dieser Phase sowie die Entwicklung eines Konzepts dafür sind in [Pi 85] ausführlich dargestellt.

e) **Assemblieren** (assembler)

-Der Assemblierer, das Werkzeug, das in dieser Phase eingesetzt wird, übersetzt das Eingabeprogramm von Assembler in Binärcode. In diesem Zusammenhang ist eine Bemerkung angebracht:

In der Literatur wird unter der Bezeichnung "Maschinencode" oder "Objektcode" (target code) entweder Assembler oder Binärcode verstanden. Wir werden uns stets auf Assembler beziehen als die "benutzerfreundliche" Form des Maschinencodes.

f) **Codeformatieren** (code formatter, linker)

-Der Codeformatierer modelliert das Binärprogramm zu einem für das jeweilige Betriebssystem akzeptablen Form. Hier werden Sprungadressen und Konstanten eingetragen und Moduln zusammengefügt.

Assemblierer und Codeformatierer werden als Hilfsprogramme (utili-

ties) mit dem jeweiligen Betriebssystem geliefert. Diese beiden Aufgaben werden öfters vom gleichem Programm bewältigt.

Ob die Phasen der Synthese in einem oder mehreren Läufen (pass) erfolgen, ist für unsere Zwecke irrelevant.

Im Rahmen dieser Arbeit betrachten wir die ersten drei Phasen der Synthese a), b) und c) als Codeerzeugung (vgl. Abb.1.4.2). Diese ersten drei Phasen müssen nicht sequentiell ausgeführt werden: so kann z.B. das Codeauswahl-Programm die Registerverteilung als Unterprogramm aufrufen, in ähnlicher Weise wie der Parser den Scanner aufruft. Im Gegensatz zur Analyse ist die Reihenfolge der ersten drei back-end Aufgaben nicht fest vorgeschrieben. In den meisten Ansätzen wird die Speicherzuordnung vor der Registerverteilung und Codeauswahl unternommen. In unserem Konzept werden Speicherzuordnung und Codeauswahl generiert. Der Codeerzeuger, als back-end Teil des Compilers, wird durch das Einfügen der Registerverteilung als Coroutine zur Codeauswahl vervollständigt (siehe Abb.II.2..2).

## 1.5 Compilerentwicklung

### 1.5.1 Compilerbaumodell

In vielen Arbeiten über Codeerzeugung [Glan 77] werden Annahmen über **maschinenabhängige** und **implementierungsabhängige** Entscheidungen gemacht, ohne näher zu erklären, was damit gemeint ist. In diesem Abschnitt werden wir uns diese Aspekte ansehen. Diesmal betrachten wir den Compilerentwickslungsprozess aus der Sicht des Compilerentwicklers. Der Compiler-Designer wird mit drei Aufgaben konfrontiert. Lunell [Lun 83] nennt sie bezeichnenderweise **Erkennung** (recognition), **Implementierung** (implementation) und **Übersetzung** (translation).

#### a) **Erkennung**

-Diese Aufgabe entspricht dem Analyse-Teil eines Compilers: Algorithmen werden entwickelt, um Programme aus einer höheren Programmiersprache zu identifizieren und zu entscheiden, ob sie korrekt bzgl. der Definition der höheren Programmiersprache sind.

#### b) **Implementierung**

-In dieser Phase wird über die Implementierung der höheren Program-

miersprache entschieden (nicht zu verwechseln mit der Implementierung des Compilers, siehe auch Bootstrapping im vorhergehenden Abschnitt). Hier wird festgelegt, wodurch Datenstrukturen der Quellsprache auf Zielmaschinen-Ebene realisiert werden und welche Strategien Prozedur-Aufrufe und Parameter-Übergabe steuern. Hier kann der Compilerschreiber z.B. entscheiden, daß die statische Prozedurverschachtelung durch "display" oder "static links" zu realisieren ist. Das Ergebnis der Erkennungs- und Teile der Implementierungsentscheidungen widerspiegeln sich in der Architektur des abstrakten Rechners und der Zwischensprache, in unserem Fall P-Maschine und P-Code. Bezogen auf die Compileraufteilung aus Abb.1.4.1.1 und 1.4.2 entspricht die Implementierung in etwa der Phasen "Erzeugung der Zwischensprache" und "Speicherzuordnung".

c) **Übersetzung**

-entspricht einem Aufgabenbereich, der Codeauswahl und Registerverteilung umfasst. Es wird festgelegt, durch welche Maschineninstruktionen Konstrukte der Zwischensprache realisiert werden.

### 1.5.2 Implementierung

Die Implementierung verdient eine nähere Betrachtung. Bezogen auf das von uns dargestellte Compilermodell ist die Implementierung ein aus zwei Aufgaben bestehender Vorgang:

-1) Implementierung der Konstrukte der höheren Programmiersprache in der Zwischensprache entsprechend der Phase "Erzeugung der Zwischensprache" des front-ends (vgl. Abb.1.4.1.1) und

-2) Implementierung der Konstrukte der Zwischensprache im Maschinencode der jeweiligen Zielmaschine entsprechend der Speicherzuordnungs-Phase des back-ends (vgl. Abb.1.4.2)

Die Erste dieser Aufgaben ist, gemäß unserer Konvention über Compilerstruktur, sprachspezifisch und maschinenunabhängig, die zweite maschinenabhängig. Aus diesen Grund nennen wir Aufgabe 1) **maschinenunabhängige Implementierung** und Aufgabe 2) **maschinenabhängige Implementierung**. Unsere Schnittstelle, P-Code, ist sprachspezifisch auf PASCAL bezogen. So werden z.B. Adressparameter nach dem "call-by-reference" Mechanismus und nicht etwa nach "call-by-name" übergeben, was eher einen ALGOL 60-Ansatz unterstützen würde.

Allgemein wird gesagt: eine Zwischensprache sei implementierungsabhängig,

wenn darin Annahmen über Implementierung einzelner Konstrukte der höheren Programmiersprache gemacht werden [Lu 83]. So gesehen dürfen wir behaupten, P-Code sei implementierungsabhängig. Damit meinen wir aber eindeutig nur die vorher definierte maschinenunabhängige Implementierung. In der Literatur wird nicht explizit zwischen maschinenunabhängigen- und maschinenabhängigen Implementierungen differenziert. Dies kann zu einer undurchschaubaren Darstellung der Zwischensprache als Schnittstelle zwischen einem maschinenunabhängigen front-end und einem maschinenabhängigen back-end führen. So ist z.B. bei Glanville [Glan 77] zu lesen, seine Zwischensprache, IR (intermediate representation) genannt, sei implementierungsabhängig und maschinenunabhängig. Offensichtlich enthält aber IR maschinenunabhängige- wie auch maschinenabhängige Implementierungsentscheidungen: "IR is a very low level and local language" (Zitat aus [Gl 77]).

Allgemein ist die Zwischensprache nicht völlig maschinenunabhängig zu gestalten, auch wenn der front-end Schreiber keine bestimmte Zielmaschine im Auge hat. Die Praxis zeigt, daß dies richtig ist. Implementierung und Übersetzung sind eng miteinander verbunden. Dies ist für die maschinenabhängige Implementierung offensichtlich klar, gilt aber ebenfalls für die maschinenunabhängige Implementierung. Liegt die maschinenunabhängige Implementierung fest, so wird die Übersetzung in die eine oder andere Richtung kanalisiert oder zumindest erleichtert. Dazu ein Beispiel:

#### Beispiel 1.5.2

Die Entscheidung, daß lokale Variablen in P-Code relativ adressiert werden gegenüber dem Anfang eines "activation record" ist ohne Rücksicht auf irgend eine Zielmaschine getroffen worden. Abschnitte, die lokale Variablen enthalten, sind dementsprechend einfacher zu übersetzen auf Zielmaschinen, die über relative Adressierung verfügen, wie z.B. die PDP-11 Familie als auf Rechner, die diese Adressierungsart nicht besitzen, wie der SAB 8080.

Aus diesem Grund scheint es uns korrekt zu sagen, eine Zwischensprache sei maschinenunabhängig; sie wird aber wegen der großen Unterschiede zwischen den Rechnerarchitekturen mit unterschiedlichem Erfolg auf die einzelnen Zielmaschinen übertragen.

## Kap.2 Codeerzeugung; Übersicht

### 2.1. Aufgabenbereich

Der Begriff **Codeerzeugung** (code generation) ist vom verwendeten Compilermodell abhängig. In einfachen Compilern wird der Codeerzeugung alles was hinter der syntaktischen Analyse folgt, zugeordnet.

Unter Codeerzeugung haben wir die ersten 3 Phasen der Synthese erfasst: Speicherzuordnung, Registerverteilung und Codeauswahl. Wie aus Abb.1.4.2 zu ersehen ist, haben wir Maschinencode-Optimierung als eine separate Phase betrachtet. Ein Maschinencode-Optimierer basierend auf Datenfluß-Analyse übernimmt diese Aufgabe [Pi 85].

Codeauswahl eignet sich gut zum Automatisieren. Aus diesem Grund wird in den meisten Ansätzen über automatische Codeerzeugung nur die Codeauswahl-Aufgabe, eventuell mit Registerverteilung als Unterprogramm, als Codeerzeugung aufgefasst. Wie wir im nächsten Kapitel sehen werden, läßt sich Codeauswahl als symbolisches **"pattern matching"** aus einer Beschreibung der Zielmaschinen-Instruktionen ableiten.

Im Folgenden zeigen wir, welche Erwartungen an einen Codeerzeuger gestellt werden und warum deren Erfüllung problematisch ist.

### 2.2. Anforderungen

Ein "guter" Codeerzeuger zeichnet sich durch folgende Eigenschaften aus:

- a) **Schnelles Abarbeiten der Zwischensprache** (rapid compilation)  
-damit wird der Wunsch nach Algorithmen geäußert, deren Komplexität linear von der Länge der Eingabe abhängt.
- b) **Zuverlässigkeit** (reliability)  
- der Codeerzeuger läuft keine Gefahr zur Compilierungszeit zu blockieren und sichert ausreichende Laufzeit-Kontrolle (z.B. dynamische Überprüfung des Typ- und Stacküberlaufs).

c) **Einfache Fehlerbehandlung** (error handling)

-direkte Korrespondenz zwischen Zwischensprache und Maschinencode zur schnellen Fehlerdiagnose.

d) **Effizienz** (efficiency)

-bezieht sich auf Zeitaufwand und Speicherplatz des erzeugten Codes. Angaben über diese zwei Größen macht jedes Hersteller-Manual; Für jede Instruktion wird der Zeitaufwand in Maschinen-Zyklen und der Speicherplatz in Bytes gemessen.

Es ist bis jetzt noch nicht gelungen, alle diese Anforderungen in befriedigender Weise gleichzeitig zu erfüllen. Schuld daran ist die Tatsache, daß sie sich teilweise widersprechen. Am deutlichsten ist der Konflikt zwischen Effizienz und Zuverlässigkeit auf der einen Seite und Fehlerbehandlung auf der anderen Seite (näheres in [Lu 83]).

Effizienz des Objektcodes bedeutet wenig "unnötigen" Code und optimierten "produktiven" Code. "Optimierter" Code heißt : schneller und billiger Code entsprechend der definierten Kostenkriterien. Bei der Optimierung wird aber die Reihenfolge der Maschineninstruktionen geändert. Dadurch geht die direkte Korrespondenz zwischen Maschinencode und Zwischensprache verloren, was die Fehlerbehandlung sehr erschwert.

Unter "produktivem" Code werden die Maschinen-Instruktionen verstanden, die unmittelbar für die Realisierung der Operationen aus der höheren Programmiersprache stehen. "Hilfsaktionen" vom Typ: Behandlung eines Übertrags (carry) des Resultats einer arithmetischen Operation, Register-Register Transfer-Instruktionen als Vorbereitung für eine Multiplikation (auf vielen Maschinen wird gefordert, daß der Multiplikand sich im ungeraden Teil eines Doppelregisters befindet) sowie Laufzeitkontrolle (Nachprüfen, ob Feldkomponenten sich z.B. innerhalb der angegebenen Grenzen befinden) sind für korrekte Codeerzeugung unentbehrlich. Die Hardware der meisten Zielmaschinen bietet wenig Möglichkeiten die Laufzeitkontrolle automatisch zu übernehmen. Der Assembler-Programmierer ist daher gezwungen sich mit diesen lästigen Codesequenzen auseinanderzusetzen. Dieser Code wird als "unnötig" bezeichnet. Beide Begriffe sind von der Sicht des Anwendungs-Programmierers definiert. Codeerzeugung erfordert also eine Kompromißlösung.

Der oben erwähnte Konflikt spiegelt nur den Unterschied zwischen der Struktur der höheren Programmiersprachen und der Rechnerarchitektur der Zielmaschinen wieder.

Innerhalb dieser Arbeit sind wir den Anforderungen der Zuverlässigkeit und Effizienz nachgegangen: Wir zeigen, daß der von uns vorgestellte Codeerzeuger nicht blockieren kann und in der Lage ist, effiziente Alternativen auszusuchen. Fehlerbehandlung wird nicht besprochen. Was das schnelle Abarbeiten der Zwischensprache betrifft, so hängt diese allein von der verwendeten festen Komponente des Codeerzeugers ab, einem erweiterten "pattern matcher" in unserem Fall. Kap.8 behandelt, welche Anforderungen ein guter "pattern matcher" zu erfüllen hat.

Innerhalb der Codeerzeugungs-Techniken basierend auf "pattern matching" wird der Konflikt zwischen Effizienz und schnellem Abarbeiten der Zwischensprache ersichtlich: um die beste Lösung auszusuchen, muß ein "pattern matcher" mehrere (eventuell alle) Alternativen aussuchen. Im schlimmsten Fall hängt die Zeitkomplexität des "pattern matching"-Algorithmus exponentiell von der Länge der Eingabe ab.

Zum Abschluß dieses Abschnitts sei noch erwähnt, wie man in der Praxis mit den konflikt-generierenden Anforderungen a)-d) fertig wird. Der Compiler wird, je nach Wunsch, der einen oder anderen Situation durch Angabe von Optionen angepasst. Ein gutes Beispiel dazu bietet der IBM PASCAL/VS Compiler: Die DEBUG/NODEBUG-Option erlaubt dem Benutzer die Information einzuschalten, die ein interaktiver "Debugger" benötigt. Die CHECK-Option ermöglicht die Generierung der Laufzeit-Kontrolle.

### 2.3 Probleme

In diesem Abschnitt werden wir die Probleme aufzählen, die den Codeerzeugungs-Prozeß zu einer nicht trivialen Aufgabe machen.

#### a) **Auswahl der richtigen Alternative**(code selection)

-Codeauswahl ist **mehrdeutig**: damit ist gemeint, daß es auf einem Rechner mehrere Folgen von Maschineninstruktionen gibt, die ein und dasselbe Konstrukt einer Quellsprache realisieren. In der Regel schreiben Kostenkriterien vor, welche Alternative ausgesucht wird. Orientiert man sich z.B. am Speicherplatz-Bedarf, so ist entscheidbar welche Alternative die beste ist, allerdings zur Zeit nur mit exponentiellen Aufwand, da das Problem NP-vollständig ist [Br Se 76]. Dieses Problem ist mit Anforderungen a) und d) verbunden. In der Praxis werden Algorithmen eingesetzt, deren Berechnungsaufwand linear von der Länge der Eingabe abhängt. Dies bedeutet, daß man sich mit "nicht optimaler" Codeauswahl

begnügen muß. Ein erheblicher Aufwand ist daher in die Entwicklung **effizienter Heuristika** investiert worden.

b) **Register-Management**

-bezieht sich auf Registerverteilung und ist mit Codeauswahl aufs engste verbunden. Dieses Problem ist, wie die Codeselektion, NP-vollständig. Für "guten" Code ist die Zusammenarbeit von Registerverteilung und Codeauswahl entscheidend. Je nach der verwendeten Zielmaschine und Ziel des Compilierungs-Prozesses kann sogar die Angabe der Kostenkriterien für effizienten Code schwierig werden, wenn von Registerverteilung und Codeauswahl eine gute Zusammenarbeit erwartet wird. Für die Umsetzung eines Quell-Konstrukts z.B. ist es oft problematisch zu entscheiden was besser ist: zwei Maschineninstruktionen, die drei Register benötigen oder drei Maschineninstruktionen mit zwei Registern?

c) **Reihenfolge der Auswertung** (computation order)

-ein ebenfalls NP-vollständiges Problem analysiert in welcher Reihenfolge die Operatoren der Zwischensprache in Maschinencode umgesetzt werden. Je bescheidener die Ressourcen der Zielmaschine sind, desto mehr Überlegungen müssen getroffen werden, wie die Register sinnvoll eingesetzt werden, wie der Speicherplatz sparend ausgenutzt wird,...etc. Dies, zusammen mit den Effizienz-Anforderungen, verbunden mit a) und b), können die Reihenfolge der Auswertung einschränken.

d) **Mangelnde Orthogonalität** (lack of orthogonality)

-das Problem tritt auf Maschinencode-Ebene auf. **Orthogonalität** ist ein Maßstab für die Regelmäßigkeit mit der Maschineninstruktionen mit **primitiven Datentypen** und **Adressierungsarten** vorhanden sind. **Primitive Datentypen** sind Datentypen, die eine direkte "hardware"-mäßige Realisierung besitzen. Ein Byte oder ein Register ist z.B. ein primitiver Datentyp, nicht aber eine Adressierungsart vom Typ "pi" (pi: STAK[SP];SP:=SP+2) wie in der Maschinenbeschreibung von INTEL 8080 von Giegerich [Gi 81] (siehe Begriff des operativen Speichers) zu lesen ist.

Jede Maschineninstruktion setzt sich zusammen aus einem Befehls- oder Operorteil (sogenannte **Assembler-Mnemonic**) und einem Operandenteil. Allgemein gilt: je grösser die Anzahl der möglichen Operanden des Befehlsteils desto "orthogonaler" die Maschineninstruktion. Auf dem SAB 8080 ist zB. keine Speicher-zu-Speicher Ad-



dition möglich: mindestens ein Additions-Term muß sich in einem Register befinden. Auf der PDP-11/70 dagegen sind Speicher-zu-Speicher Addition und Subtraktion möglich. Nach ähnlichen Vergleichen des ganzen Befehlssatzes von SAB 8080 und PDP-11/70 kann man behaupten: Der Befehlssatz vom PDP-11/70 ist "orthogonaler" als der des SAB 8080.

e) **Spezielle Eigenschaften der Zielmaschine** (machine idioms)

-Die meisten Zielmaschinen verfügen über spezielle Eigenschaften, die sich als sehr effizient erweisen. Beispiele dazu: Inkrement- und Autoinkrement-Instruktionen, spezielle Adressierungsarten, 3-Adress Instruktionen, usw. Die Verwendung dieser Instruktionen erschwert die Codeauswahl: Um z.B. festzustellen, wann ein Additions-Befehl durch einen Inkrement-Befehl ersetzbar ist, hat der Codeerzeuger zu prüfen, ob einer der Summanden den konstanten Wert 1 besitzt.

f) **Exotische Befehle** (exotic instructions)

-gibt es zunehmend auf moderneren Architekturen, etwa VAX 11, IBM 370, NS 16032. Beispiel dafür: String-Suche, Matrix-Multiplikation, Block-Bewegung,...,etc. Obwohl sehr effizient, werden sie wegen der unerwünschten Neben-Effekte, wie z.B. Vernichten der Inhalte mehrerer Register nicht ohne weiteres eingesetzt.

g) **Kontext-Problem**

-der Kontext in dem sich das Konstrukt befindet, für das Code erzeugt wird, beeinflußt die Codeauswahl. Ein klassisches Beispiel dazu: Der **Mc Carthy Auswertungs-Stil** von Bool'schen Ausdrücken (short-circuit Boolean evaluation). Um einen Bool'schen Ausdruck auszuwerten, der "und" (and) und "oder" (or) Operatoren enthält, braucht man nicht jedes Mal alle zugehörigen Operanden zu betrachten. Ein bedingter Sprung kann z.B. erfolgen, nachdem der erste Operand der folgenden "and"-Sprungbedingung "(a=0) and (b=0)" ausgewertet wurde.

## 2.4 Konventionelle Codeerzeugung

Die Mehrheit der Lehrbücher über Compilerbau beschreiben ähnliche Verfahren für Codeerzeugung, die in vielen, vorwiegend monolithischen Compilern, angewandt werden. Typisch für diese Verfahren ist die sogenannte

**"Makroexpansion"** : jeder Operator der Zwischensprache wird, separat für jede Zielmaschine, in eine Folge von Maschineninstruktionen umgesetzt. Jedem Operator der Zwischensprache wird also eine Makrodefinition zugeordnet. Zur Compilierungszeit werden diese Makros gemäß ihrer Definition expandiert (siehe dazu die Behandlung von **TripleIn** und **QuadrupleIn** in [Ah U 77].)

Einige Autoren verwenden Prozeduren statt Makros, die zur Compilierungszeit aufgerufen werden.

Der Nachteil dieser Methode ist, daß die Operatoren der Zwischensprache als unabhängige Aufgaben betrachtet werden, ohne den **Kontext** zu berücksichtigen, in dem sie sich befinden. Zu einem Operator der Zwischensprache werden (höchstens) seine Operanden (unmittelbare Nachfolger) betrachtet. Für effiziente Codeauswahl ist dies aber unzureichend.

Ein einfaches Beispiel soll diese Situation illustrieren:

Beispiel 2.4:

Betrachten wir einen Ausschnitt eines arithmetischen Ausdrucks aus einer höheren Programmiersprache. Die Darstellung in einer Zwischensprache (etwa IR von Glanville) könnte folgenden **Strukturbaum** liefern: (siehe auch **abstrakter Syntaxbaum** in [Zi 83] oder [Wi 82] und **Operatorbaum** in OPTRAN aus [GL Mö Wi 80].)

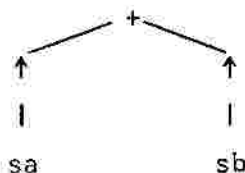


Abb.2.4: Strukturbaum für Addition

Der Strukturbaum ist Eingabe für den Codeerzeuger. Aus diesem Grund nennen wir ihn auch **Eingabebaum**.

Es wird angenommen, "a" und "b" sind globale Variablen, gespeichert auf den absoluten Adressen "sa" und "sb" (vgl. auch Implementierungsentscheidungen aus Kap.1). Der Einfachheit halber identifizieren wir die absolute Adressen sa und sb aus der Zwischensprache mit den konkreten absoluten Adressen der Zielmaschine.

Bemerkung 2.4.1:

Für den Rest des ersten Teils dieser Arbeit nehmen wir an, daß die Speicherzuordnung stattgefunden hat. "sa" und "sb" aus Abb.2.4 können jede beliebige Speicheradresse bezeichnen: z.B.  $sa, sb \in \{1, \dots, 65536\}$  für den SAB 8080 Datenspeicher.

Der **Zugriffs-Operator "↑"** (dereferencing operator), angewandt auf eine Adresse, berechnet den Inhalt dieser Adresse. Der obige Teilbaum wird gelesen: "Addiere die Inhalte der absoluten Adressen sa und sb." Weiterhin nehmen wir an, wir hätten einen Rechner, der über folgende Transfer- und Arithmetik-Befehle verfügt:

| <u>Maschinencode</u> | <u>Wirkung</u>  |
|----------------------|---|
| LOAD Ri,(sm)         | Lade Inhalt der Speicheradresse sm auf Register Ri    |
| ADD Ri,Rj            | Addiere Inhalt von Register Ri und Rj; Resultat in Ri |
| ADD (sm),(sn)        | Addiere Inhalt der Adressen sm und sn; Resultat in sm |

Bemerkung 2.4.2:

Im Laufe dieser Arbeit werden wir das Problem der Registerverteilung nicht behandeln. In sämtlichen Beispielen gehen wir von der Annahme aus, daß wir genügend Register zur Verfügung hätten.

"Ri,Rj" sind Stellvertreter für eine Klasse von Registern, z.B.  $Ri, Rj \in \{\text{Register1}, \dots, \text{Register16}\}$  für die PDP 11/70.

Maschinencode wird in einem **bottom-up** Baumdurchlauf erzeugt, indem jeder begegnete Knoten (Makro)- expandiert wird. Der Grund für die bottom-up Auswertung ist, daß Operanden eines Operators vor dessen Expansion bekannt sein müssen. Gemäß dieser Strategie erhalten wir für den Operator-Baum aus Abb.2.4 folgende Codesequenz:

```
LOAD R1,(sa)
LOAD R2,(sb)
ADD R1,R2
```

(Ende des Beispiels 2.4).

Dasselbe Resultat, dh. äquivalenten\* Code, liefert die Maschineninstruktion

```
ADD (sa),(sb)
```

Um die effizientere (1 Maschineninstruktion statt 3) Speicher-zu-Speicher

\*Äquivalent bezieht sich hier nur auf die Berechnung der Addition,

Addition zu verwenden, muß dem Codeerzeuger folgendes bekannt sein:

erstens

-daß der Vater beider "↑"-Operatoren der "+"-Operand ist, und

zweitens

-daß für die Söhne ("↑"-Operatoren) kein Code erzeugt wird, solange der Vater ("+"-Operator) nicht bekannt ist.

Makroexpansion ist relativ einfach zu gestalten, solange man jeden Knoten der Zwischensprache als ein separates Problem betrachtet. Möchte man Maschinencode für mehrere benachbarte Knoten erzeugen und gleichzeitig ineffizienten Code wie im vorherigen Beispiel ausschließen, so muß man in den Makro-Definitionen Fallunterscheidungen einfügen, um Kontext zu differenzieren. Dies führt aber zu langen, unüberschaubaren Makro-Definitionen. Hilfe in dieser Richtung bieten spezielle **problem-orientierte Sprachen** (problem oriented languages). Problem-orientierte Sprachen werden für eine besondere Aufgabe entwickelt, in diesem Fall Codeerzeugung [Do Noo Fe 79]. Makros werden in einer problem-orientierten Sprache geschrieben, von einem "preprocessor" in die Implementierungssprache des Compilers übersetzt und anschließend als **Interpreter** dem front-end nachgeschaltet.

Auch der PASCAL [P] Compiler wird zusammen mit einem P-Code Interpreter geliefert.

In diesem Zusammenhang sind die problem-orientierten Sprachen CGL von Elson & Rake, SLM von Wilcox und CGPL von Donegan zu erwähnen. Näheres darüber in [Gr 80], [Gana Fi Henn 82] und [Lu 83]. In den Arbeiten, die sich mit Klassifizierungen verschiedener Techniken der Codeerzeugung beschäftigen, wird die oben besprochene Kategorie dem "Interpreter"-Versuch zugeordnet [Cat 77], [Gana Fi Henn 82] und [Gr He 84].

---

nicht aber auf die Lokalisierung des Resultats (im ersten Fall in Register R1, im zweiten auf Adresse (sa)).

## 2.5 Übergabe der Kontextinformation

Der Mangel an Kontext-Information wird aufgehoben, indem man Maschinencode für mehrere Knoten des abstrakten Syntaxbaumes erzeugt.

### Beispiel 2.5.1

Betrachten wir erneut Abb.2.4. Ein Algorithmus, der den Eingabebaum von oben nach unten durchläuft, liest zunächst den "+"-Operator. Zu diesem Zeitpunkt kann er noch keinen Maschinencode erzeugen, weil ihm die Operanden nicht bekannt sind. Nach Begegnung der beiden "^"-Operanden, wird der ADD (sa),(sb)-Befehl ausgegeben. Der Codeerzeuger ist somit "gezwungen", die "bessere Alternative auszusuchen.

Im vorigen Abschnitt haben wir gesehen, daß der Codeerzeuger in einem bottom-up Baumdurchlauf ebenfalls die ADD (sa),(sb)-Instruktion auswählt, falls er fähig ist die Codeausgabe zu verschieben bis er den "+"-Operator gesehen hat.

Die meisten Ansätze über automatische Codeauswahl verwenden **tafelgesteuerte** (table driven) Algorithmen. Die eingelesenen Knoten werden auf einem Keller gespeichert. Der "top-down" Baumdurchlauf ist in diesem Fall äquivalent mit dem Einlesen des linearisierten Baums (string) in Präordnung (von links nach rechts). "Bottom-up" Baumdurchlauf heißt dagegen den linearisierten Baum als Postfixstring zu lesen. Die Verzögerung der Codeausgabe zugunsten des Kontextsammelns kann sich in einer "bottom-up"-Strategie als nutzlos erweisen. Folgendes Beispiel zeigt, wie dies passieren kann.

### Beispiel 2.5.2

Wir erweitern unseren hypothetischen Rechner aus dem vorigen Abschnitt um eine zusätzliche Instruktion:

MUL Ri,Rj, mit der Wirkung: Multipliziere Inhalt der Register  
Ri und Rj; Resultat in Ri.

Gleichzeitig nehmen wir an, es gebe keine Speicher-zu-Speicher Multiplikation, etwa MUL (sa),(sb); ähnliche Voraussetzungen gibt es, zB. auf PDP-11 oder VAX-11 Maschinen.

Ersetzen wir den "+"-Operator aus Abb.2.4 durch den "\*" (Multiplikations-Operator), so erhalten wir folgenden Teileingabebaum:

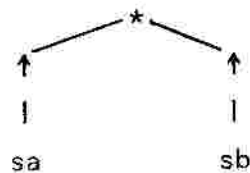


Abb.2.5: Strukturbaum für Multiplikation

Mit der bottom-up "Verzögerungs"-Strategie aus dem letzten Abschnitt gerät der Codeerzeuger in eine "Sackgasse": die beiden "↑"-Operatoren werden eingelesen ohne dafür Code zu liefern. Bei Begegnung des "\*" - Operators stellt der Codeerzeuger fest, daß es keine Maschineninstruktion gibt, die den Teileingabebaum aus Abb.2.5 abdeckt. Um diese falsche Entscheidung wiedergutzumachen, muß der Algorithmus zurück (**back-track**) zu den "↑"-Operatoren, diese in Register aufladen und erst dann die Register-Register Multiplikation ausführen. Folgende Maschinencode-Sequenz wird erzeugt:

```
LOAD R1,(sa)
LOAD R2,(sb)
MUL R1,R2
```

Ein top-down Algorithmus hätte dieselbe Codesequenz geliefert ohne back-track; Speicherzugriffe werden in diesem Fall als Operanden abgeschlossen sobald der "\*" - Operator eingelesen wird. Aus diesem Beispiel ergibt sich, daß die an einem Knoten des Eingabebaums relevante Information durch die Menge der Zielmaschinen-Ressourcen, die bei der Umsetzung aller potentieller Nachfolger des Knotens in Frage kommen, bestimmt wird.

Bemerkung 2.5.1:

In den verbleibenden Beispielen dieser Arbeit nehmen wir an mit einem "pattern matcher" zu arbeiten, der die Eingabebäume "top-down" durchläuft.

Aus dem Beispiel 2.5.2 wird die Bedeutung der Orthogonalität ersichtlich. Die ADD-Instruktion ist mit dem primitiven Datentyp "Register" und der Adressierungsart "absolute Adresse" verwendbar, MUL dagegen nur mit "Register". Hätten die ADD- und MUL-Befehle dieselbe "Orthogonalität", würde die Durchlauferrichtung des Strukturbaums in einem tafelgesteuerten Algorithmus keine Rolle spielen.

Die Annahme, daß jeder innere Knoten des Eingabebaums sich in eine Maschineninstruktion umsetzen läßt, ist nicht immer realistisch. In den obigen Beispielen sind wir von einer "1-zu-1" Korrespondenz zwischen den Knoten des Eingabebaums und den Maschineninstruktionen ausgegangen. In vielen Fällen brauchen wir eine Folge von Maschineninstruktionen für die Umsetzung eines Knotens des Eingabebaums.

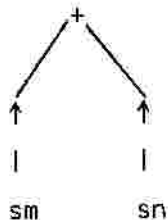
### Kap.3 Automatisierung der Codeerzeugung

#### 3.1 Motivation und Entstehung

Der Codeerzeuger enthält einen Codeauswahl-Algorithmus, der, wie aus dem letzten Abschnitt resultiert, folgendes bestimmt:

- welche Alternativen sich bei der Codeauswahl ergeben und
- welche davon die beste ist, wobei die Entscheidung nach Kostenkriterien erfolgt.

Wie wir an Beispiel 2.4 und 2.5 gesehen haben, bestimmen die vorhandenen Maschineninstruktionen welche (Teil)-Eingabebäume bei der Codeauswahl in Frage kommen und ausgesucht werden. Betrachtet man den "+"-Knoten als Makro, so wird er für den Fall, daß er als Operanden die "+"-Operatoren hat, durch die Maschineninstruktion ADD (sm),(sn) definiert. Man sagt in diesem Fall, der Teileingabebaum



ist äquivalent zur ADD (sm),(sn)-Instruktion. Umgekehrt kann man diesen Baum als die Beschreibung der obengenannten Maschineninstruktion auffassen. Codeauswahl bedeutet aus dieser Sicht:

Bestimme in einem Baumdurchlauf die Teileingabebäume entsprechend der Beschreibung der Maschineninstruktionen und suche die "besten" aus. Einzige Voraussetzung dabei: Die Symbole der Beschreibung der Maschineninstruktionen und des Eingabebaums müssen dieselbe Bedeutung haben. Dadurch reduziert sich das Codeauswahl-Problem auf symbolische **Mustererkennung** ("pattern matching"). Die Menge der beschriebenen Maschineninstruktionen dient als **Suchmuster** (patterns).



### 3.2 Beschreibung der Instruktionen der Zielmaschine

Um die Maschineninstruktionen vollständig zu beschreiben, müssen wir noch angeben, wo das Resultat einer Operation lokalisiert wird. Aus diesem Grund unterscheiden wir innerhalb der Beschreibung zwei Teile:

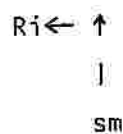
-die **rechte Seite** zeigt, welche Operationen ausgeführt werden. Dieser Teil stellt das "pattern" dar.

-die **linke Seite** gibt an, wo das Resultat der rechten Seite lokalisiert wird.

Linke und rechte Seite der Beschreibung werden durch das Metasymbol " $\leftarrow$ " getrennt, das nicht der Beschreibungssprache angehört.

#### Beispiel 3.2:

Wir beschreiben die LOAD R1,(sm) durch:



Die restlichen Instruktionen des hypothetischen Rechners werden wie folgt beschrieben:

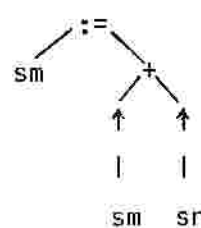
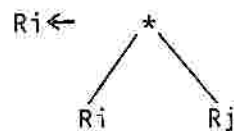
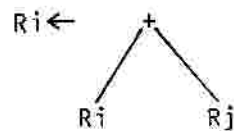
#### Maschineninstruktion

ADD R<sub>i</sub>,R<sub>j</sub>

MUL R<sub>i</sub>,R<sub>j</sub>

ADD (sm),(sn)

#### Beschreibung



"ADD (sm),(sn)" ist ein komplexer Befehl, der 4 Operatoren der Zwischensprache (2 Speicherzugriffe, eine Addition und ein "store") in Maschinencode umsetzt.

Das Symbol ":= " (assign) ist der binäre Zuweisungsoperator der Zwischensprache mit der Wirkung: Speichere den Inhalt des rechten Teilbaums in der Speicherzelle mit der Adresse, die im linken Teilbaum angegeben ist.

Wie wir im nächsten Abschnitt sehen werden, stellen die "assignments" die Wurzel der Eingabebäume dar. Diese Eingabebäume werden als "End-patterns" erkannt und nicht weiter durch neue patterns ersetzt. Aus diesem Grund wird hier nicht explizit zwischen linker und rechter Seite unterschieden, so daß in diesem Fall die komplette Beschreibung der Maschineninstruktion als pattern betrachtet wird.

#### Bemerkung 3.2:

Im Laufe dieser Arbeit werden wir, wenn nicht explizit anders gesagt, unter der Bezeichnung Maschinenbeschreibung die Beschreibung der Maschineninstruktionen verstehen.

### 3.3. Codeerzeugung als "pattern matching"

Ein tafelnesteuerter Codeauswahl-Algorithmus basierend auf "top-down pattern matching" funktioniert, grob betrachtet, nach dem Schema aus Abb.3.3.1.

#### Eingabe

Programm als Folge von "assignments" in der Zwischensprache Die Eingabe kann als Baum oder als string angegeben werden.

#### Ausgabe

Programm als Folge von Maschineninstruktionen (Assembler)

#### Bemerkung 3.3.1:

Wir haben uns für den Baum als Eingabe entschieden, da wir über "tree pattern matcher" verfügen, die wir als feste Komponenten unseres Codeerzeuger-Generator Systems einzusetzen beabsichtigen (für den Aspekt der P-Code Bäume siehe Kap.4).

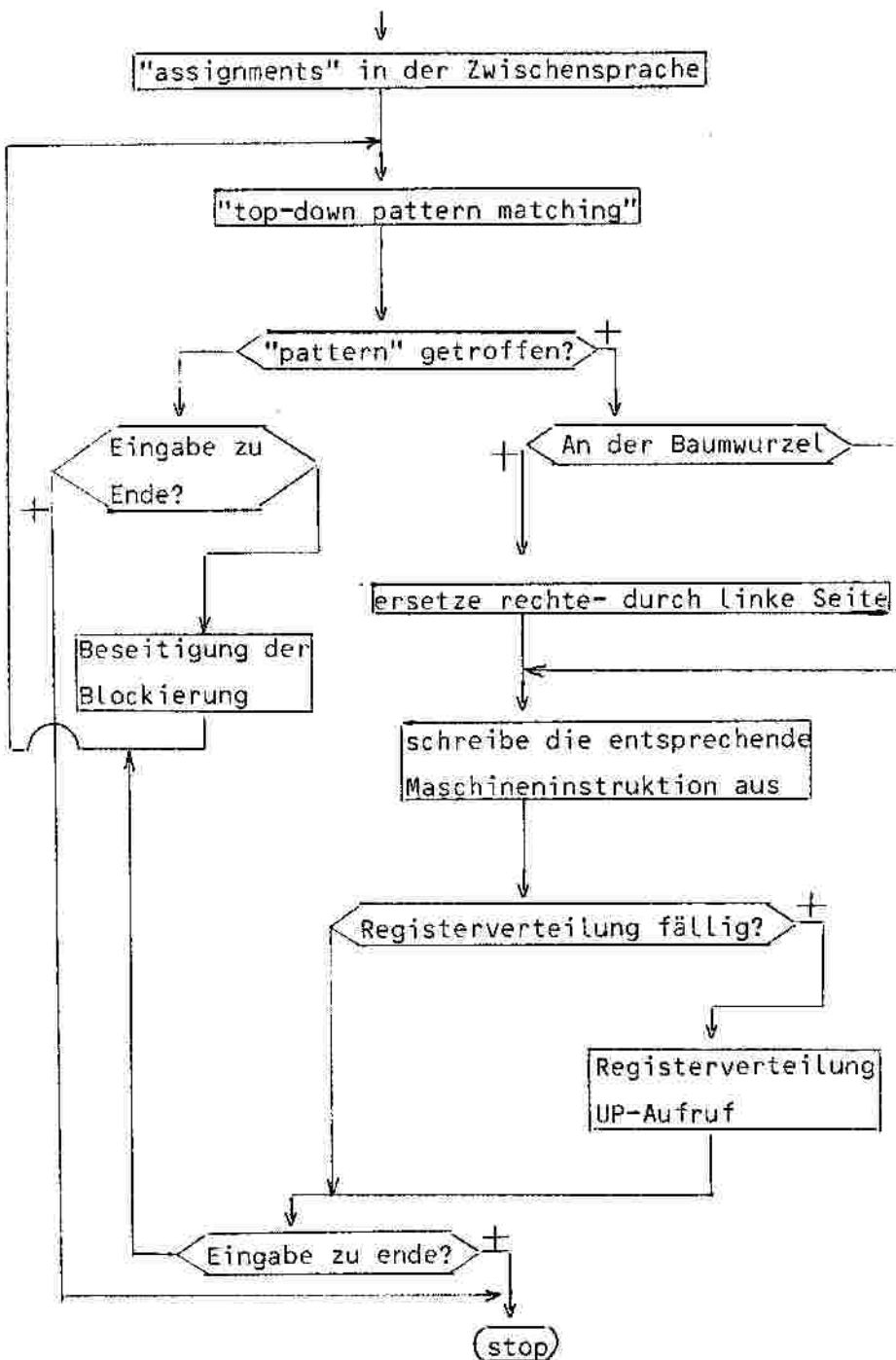


Abb.3.3.1: Codeauswahl als pattern matching

Ein einfaches Beispiel unseres hypothetischen Rechners soll den Algorithmus verdeutlichen.

Beispiel 3.3

Angenommen wir möchten Maschinencode für eine Zuweisung vom Typ "a := a\*b" erzeugen mit a und b als globale Variablen. Die Übersetzung in der Zwischensprache könnte in etwa folgenden Baum ergeben:

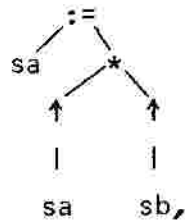


Abb.3.3.2: Eingabebaum für a:=a\*b

unter der Annahme, globale Variablen werden auf absoluten Adressen gehalten.

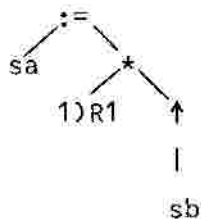
Wir fügen eine neue Maschineninstruktion ein:

"STORE sm,Ri" mit der Wirkung: Speichere Inhalt des Registers Ri auf Speicheradresse sm.

Um einfacher zu verfolgen, an welchem Knoten ein "pattern" getroffen hat, haben wir Knoten und "patterns" mit derselben Zahl gekennzeichnet. Das Ende der Eingabe wird durch das "#" -Symbol angegeben.

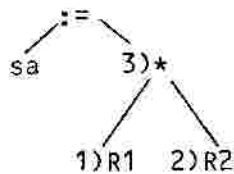
Nach dem Algorithmus aus Abb.3.3.1. ergibt sich:

Eingabe



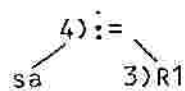
Ausgabe

1)LOAD R1,(sa)



1)LOAD R1,(sa)

2)LOAD R2,(sb)



LOAD R1,(sa)

LOAD R2,(sb)

3)MUL R1,R2

```
#          LOAD R1,(sa)
          LOAD R2,(sb)
          MUL R1,R2
4)STORE R1,sa
```

### Bemerkung 3.3.2:

Der Eingabebaum wird "top-down" durchlaufen. Die Reduktionen (ersetze rechte Seite durch linke Seite) erfolgen "bottom-up" (vgl. auch LR-Parser Techniken: Einlesen der Eingabe von links nach rechts mit "bottom-up" Reduktion unter Berücksichtigung der Diskussion aus 2.5).

Wir haben angenommen, daß unser hypothetischer Rechner mindestens zwei gleichberechtigte Register, R1 und R2, besitzt.

Dem obigen Beispiel ist noch ein wichtiges Ergebnis zu entnehmen: Die Abarbeitung des Eingabebaums kann durch einen "push-down" Automaten simuliert werden.

-Die eingelesenen Symbole werden im Keller gespeichert.

-Der Übergang zum nächsten Zustand wird vom aktuellen Zustand und vom zuletzt eingelesenen Symbol bestimmt.

Der Vorteil tafalgesteuerter Algorithmen ist die problemlose Automatisierung: Anhand einer Beschreibung der Maschineninstruktionen baut ein Generator die Übergangstafeln; der "pattern matcher" bestimmt, wie der Eingabebaum damit transformiert wird. Das Übertragen des Compilers auf einen neuen Rechner reduziert sich damit auf das Ersetzen der generierten Tabellen. Dieser Prozess erfordert vom Benutzer die Zusammenstellung der Eingabe für den Generator, d.h. die Beschreibung der Maschineninstruktionen. In groben Zügen ist dies die Idee der Methode von Graham und Glanville für Codeerzeugung [Glan 77], [Glan Gr 78] und [Gr 80].

## 3.4 Das Verfahren von Graham und Glanville

### 3.4.1 Überblick

1977 hat Glanville ein Generierungs-Verfahren für Codeerzeugung ausgearbeitet, das Ergebnisse aus der syntaktischen Analyse ausnutzt:

-Der "pattern matcher" basiert auf einem modifiziertem SLR(1) Parser Treiber.

-Die Beschreibungen der Maschineninstruktionen werden zu kontext-freien Produktionen umgeformt, aus denen ein Generator, "preprozessor" genannt, die Übergangs-Tabellen des Parsers konstruiert.

Das Gesamtbild des generierenden Systems sieht folgendermaßen aus:

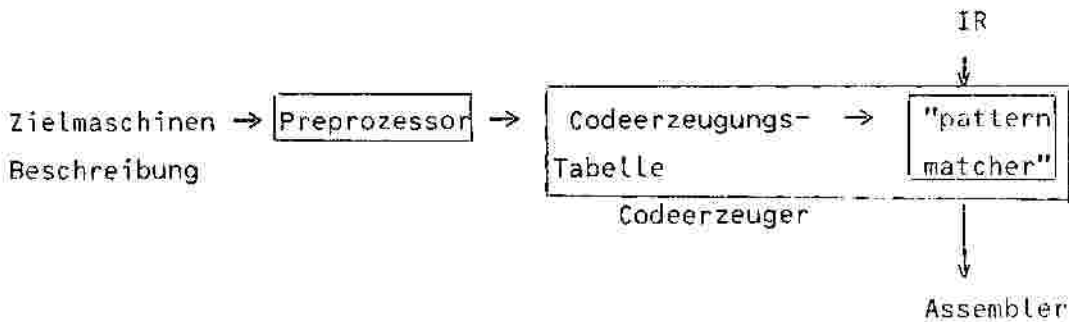


Abb.3.4.1.1 Das Generierungs-System von Glanville

Wie auf dem obigen Bild zu sehen ist, setzt sich der Codeerzeuger aus zwei Teilen zusammen:

- Der Treiber (table-driven pattern matcher) ist die feste Komponente.
- Die Tafel (code generator table) ist die generierte Komponente.

Glanville nennt seinen Treiber "pattern matcher". In Wirklichkeit handelt es sich um einen **"erweiterten pattern matcher"**: Mit jedem Treffer sind semantische Aktionen (Reduktionen, Aufruf der Registerverteilungs-Routine,...) verbunden wie aus Abb.3.3.1 zu sehen ist.

Der Generator von Glanville (Preprozessor) ist ein komplexer Algorithmus, der folgende Aufgaben übernimmt:

- a) -baut aus der Maschinenbeschreibung die kontext-freie Grammatik (hier "instruction set grammar" genannt) auf;
- b) -lokalisiert und beseitigt die Zustände, in denen der Algorithmus Gefahr läuft in eine unendliche Schleife zu geraten (looping detection and removal);
- c) -lokalisiert und beseitigt die Zustände, in denen der Algorithmus Gefahr läuft zu blockieren (blocking detection and removal);
- d) -konstruiert die Codeerzeugungstafeln.

Glanville betrachtet nur die Aufgabe der Codeauswahl. Speicherzuordnung wird einem Zwischen-Übersetzungsschritt (machine dependent

translator) überlassen, der einen APT (abstract parse tree) in IR (intermediate representation) umsetzt. IR ist explizit maschinenabhängig.

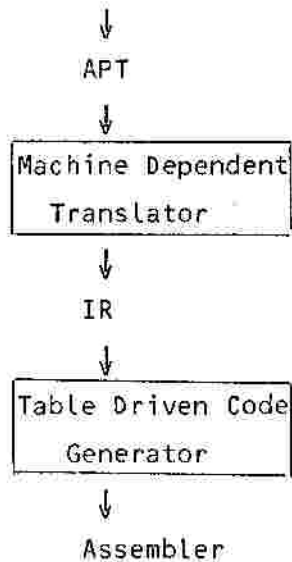


Abb.3.4.1.2: back-end Einteilung von Glanville

### 3.4.2 Tafelkonstruktion; "maximal munch"-Strategie

Wie wir bereits gesehen haben, ist die Auswahl einer "guten" Codesequenz ein wichtiges Problem der Codeerzeugung. Für den "preprocessor" bedeutet die Entscheidung für die eine oder die andere Alternative die Beseitigung der Mehrdeutigkeit der konstruierten Grammatik. In diesem Fall kann die Mehrdeutigkeit nicht durch Umformen von Produktionen aufgehoben werden: die kontext-freien Produktionen stammen aus den Maschineninstruktionen und diese können nicht geändert werden. Das Problem wird vom Generator entsprechend der Heuristik gelöst: treffe die längsten "patterns"! Im Sinne dieser Idee werden die auftretenden Konflikte wie folgt behandelt:

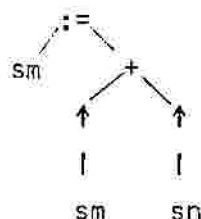
**shift/reduce Konflikte** werden zugunsten von shift entschieden;

**reduce/reduce Konflikte** werden zugunsten des längsten Musters entschieden.

Folgendes Beispiel zeigt die Anwendung dieser Heuristik.

#### Beispiel 3.4.2.:

Wir betrachten erneut den Eingabebaum entsprechend der ADD (sm),(sn)-Instruktion



-Ein shift/reduce Konflikt entsteht beim Einlesen des "sm"-Operanden. Hier angekommen, kann der "pattern matcher" weiterlesen (shiften) gemäß dem ADD (sm),(sn)-Befehl oder reduzieren entsprechend der LOAD Ri,(sm)-Instruktion.

-Ein reduce/reduce Konflikt entsteht bei der Begegnung des "sn"-Operanden. Der Treiber kann nach der LOAD Ri,sn- oder ADD (sm),(sn)-Instruktion reduzieren.

In der Literatur ist diese Heuristik unter dem Namen "maximal munch"-Methode bekannt. Der Name ist von R.G.Cattell in [Cat 78] eingeführt worden. Fast alle in der Praxis eingesetzten Codeauswahl-Algorithmen orientieren sich danach. Die Idee, möglichst große Abschnitte der Zwischensprache in einer Maschineninstruktion umzusetzen entspricht den zwei gängigsten Kostenkriterien: wenig Maschinencode bedeutet wenig Speicherplatz und wenige Maschinenzyklen.

Diese Heuristik bietet nicht in allen Fällen befriedigende Ergebnisse. Die Abdeckung eines großen Unterbaums zu einem früheren Zeitpunkt kann eine günstigere Reduktion zu einem späteren Zeitpunkt annullieren.

Im zweiten Teil dieser Arbeit werden wir sehen, daß u.U. genau die "Gegen"-Heuristik, d.h. bilde möglichst kurze Abschnitte der Eingabe auf einer Maschineninstruktion innerhalb des Generierungsschrittes die besseren "patterns" liefert.

### 3.4.3 Weitere Entscheidungs-Probleme der Codeauswahl

#### a) Zu reduce/reduce Konflikte; Semantische Einschränkungen

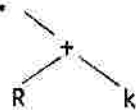
Ein weiteres Problem entsteht bei den reduce/reduce Konflikten, wenn Produktionen die gleiche Länge haben. Eine einfache Heuristik, die sich ebenfalls an Kostenkriterien orientiert, wird in diesem Fall angewandt: Der Preprozessor baut eine "Prioritätenliste" der Maschineninstruktionen mit gleichen syntaktischen Mustern auf. Instruktionen mit besonderen



Eigenschaften wird eine höhere Priorität zugewiesen als Instruktionen mit allgemeinerem Charakter (siehe auch Codeerzeugungs-Probleme; "machine idioms").

Beispiel 3.4.3.1:

Der Teileingabebaum ... wird stets in eine



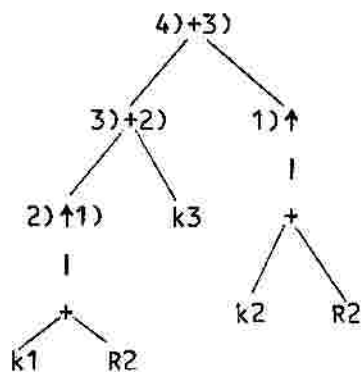
Inkrement-Instruktion umgesetzt, falls die Konstante "k" den Wert 1 hat und nicht etwa in einer Additions-Instruktion vom Typ "Addiere eine Konstante zum Inhalt eines Registers".

b) Zur Reihenfolge der Auswertung der Baumoperatoren

Die Verfahren basierend auf dem Ansatz von Glanville benutzen ein "top-down" Baumdurchlauf um "back-track" zu vermeiden (siehe dazu auch Diskussion in 2.5). Daß diese Durchlaufrichtung nicht immer den "besten" Code produziert zeigt folgendes Beispiel:

Beispiel 3.4.3.2:

Folgender Eingabebaum wird dem Codeerzeuger geliefert:



Wir erweitern unsere hypothetische Maschine um relative Adressierung. Eine relative Adresse,  $(R_b + K_j)$ , wird aus der Addition des Inhalts des Basisregisters  $R_b$  und einem konstanten "displacement"  $K_j$  berechnet. Wir nehmen an,  $R_2$  sei das Basisregister.

Folgende zwei Maschineninstruktionen mit relativer Adressierung werden im Befehlssatz des hypothetischen Rechners aufgenommen:

| <u>Maschineninstruktion</u> | <u>Beschreibung</u>  |
|-----------------------------|--|
| LOAD Ri,Kj(Rb)              | $R_i \leftarrow \uparrow$<br> <br>+<br>/ \<br>Rb Kj                      |
| ADD Ri,Kj(Rb)               | $R_i \leftarrow \uparrow$<br>+<br>/ \<br>Ri Ri<br> <br>+<br>/ \<br>Kj Rb |

Wird der Eingabebaum "top-down" durchlaufen, so ergibt sich nach der Strategie von Glanville folgender Maschinencode:

- 1)LOAD R1,K2(R2)
- 2)LOAD R3,K1(R2)
- 3)ADD R3,K3
- 4)ADD R3,R1;

Das Treffen der "patterns" wird durch die Zahl links von den Knoten angegeben.

Wird dagegen der linke Unterbaum des ersten "+"-Operators vor dem zweiten reduziert, so erhalten wir:

- 1)LOAD R1,K1(R2)
- 2)ADD R1,K3
- 3)ADD R1,K2(R2).

Die Zahl rechts von den Knoten zeigt an, welche "patterns" getroffen haben.

Die erste Codesequenz benutzt 4 Maschineninstruktionen und 3 Register, die zweite dagegen nur 3 Maschineninstruktionen und 2 Register. Der Grund für die unterschiedliche Registerverteilung ist die Tatsache, daß der Inhalt von R2 als Basisregister nicht überschrieben werden darf. Benutzt man, aus welchem Grund es auch sei (z.B. die Zielmaschine hat nur zwei Register) für die erste Codesequenz zwei Register, so muß der Codeerzeuger versichern, daß der Inhalt von R2 vor der Ausgabe der Codesequenz gerettet und anschließend wieder hergestellt wird. Das Beispiel ist typisch für die Zusammenarbeit zwischen Codeauswahl und Registerverteilung.

#### 3.4.4 Registerverteilung

Im obigen Beispiel wird Code erzeugt unter der Annahme daß 3 Register zur Verfügung stehen. Wenn innerhalb der Codeerzeugung die Register "ausgehen", so werden alte Register wieder freigegeben durch Abspeichern ihrer Inhalte auf markierten Adressen. Diese Technik ist unter den Namen "on-the-fly" bekannt. Diese typische "ad hoc"-Methode liefert befriedigende Resultate für hinreichend kleine Eingabebäume.

Im Gegensatz dazu wird von Glanville als globale Strategie die "preplanning"-Methode erwähnt [Gr 80] als Verbesserungsvorschlags für den Aufbau seines Codeerzeugers. In diesem Fall wird vor der Codeauswahl die Anzahl der benötigten Register abgeschätzt.

#### 3.4.5 Beseitigung der Blockierungs-Situationen

Der Preprozessor kann anhand der konstruierten Grammatik prüfen, ob die Codeauswahl blockiert oder nicht. Glanville beweist, daß der Algorithmus nicht blockiert gdw. die Grammatik **uniform** ist. Uniformität bedeutet intuitiv: Die Operanden eines jeden Operators sind unabhängig vom Kontext gültig. Kap.5 aus [Gl 77] zeigt, wie der Uniformitäts-Test ausgeführt wird. Somit hat der Systembenutzer die Möglichkeit seine Generator-Eingabe auf Vollständigkeit zu prüfen. Das Ergebnis der Uniformität ergibt: Die abgeleiteten Grammatiken der untersuchten Zielmaschinen ergeben sich als uniform aufgrund des kontext-freien Einsatzes der Maschineninstruktionen. Eine Grammatik, die nicht uniform ist, deutet auf eine inkomplette Beschreibung der Maschineninstruktionen hin.

#### 3.4.6 Schlußfolgerungen

Das Verfahren von Glanville hat viele Nachfolger gefunden. Diesen Erfolg verdankt es in erster Linie [Lu 83] der Popularität der LR-Parsing Techniken. Anwendungen und Experimente sind in [La Jan 80], [Bir 82], [He 81], [Craw 82], [Gr He Schu 82] zu finden.

In der angegebenen Literatur werden vorwiegend folgende Probleme behandelt:

-Faktorisieren der Eingabe-Grammatik (eine komplette VAX 11 Beschreibung würde schätzungsweise an die  $8 \cdot 10^6$  Produktionen erfordern) in [He 81] und

[Gr He 84] und

-Verallgemeinerung der semantischen Aktionen in [Gr he Schu 82] und [Craw 82].

Einen natürlichen Erweiterungsansatz des Verfahrens von Glanville unternimmt Ganapathi in [Gana 80]: der Codeauswahl-Algorithmus wird durch Attributierung erweitert.

Die Generierungsverfahren dieser Kategorie benutzen Ergebnisse der syntaktischen Analyse. Aus diesem Grund werden sie von Lunell [Lu 83] dem "Grammatik-Ansatz" (grammar approach) zugeordnet. Der "Grammatik-Ansatz" hat relativ schnell seine Grenzen gezeigt: diese hängen hauptsächlich mit den Einschränkungen des "string pattern matching" zusammen. In dieser Hinsicht bieten "tree pattern matcher" ein mächtigeres Werkzeug an.

### 3.5 Über Automatisierung

Im Laufe dieser Arbeit werden wir öfters über die Komponenten eines generierenden System zu sprechen kommen. Was Namen und Funktion der einzelnen Komponenten betrifft, halten wir uns an das von Lunell [Lu 83] vorgeschlagene System:

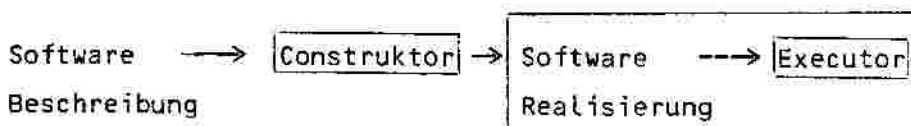


Abb.3.5.1: Constructor-Executor Schema

Ein Vergleich mit Abb.3.4.1 zeigt:

- der "Constructor" ist der Preprozessor,
- die Software-Realisierung (bezeichnet das generierte Produkt) ist die Codeerzeugungs-Tabelle,
- der "Executor" ist der "erweiterte pattern matcher".

In der Praxis werden verschiedene separat entwickelte Lösungsansätze von Teilen eines "Constructors-Executors" Systems als semantische Aktionen (supplementary routines) sowohl zur Generierungszeit als auch zur Compilierungszeit eingefügt. Ein etwas realistischeres Generierungssystem sieht dann folgendermaßen aus:

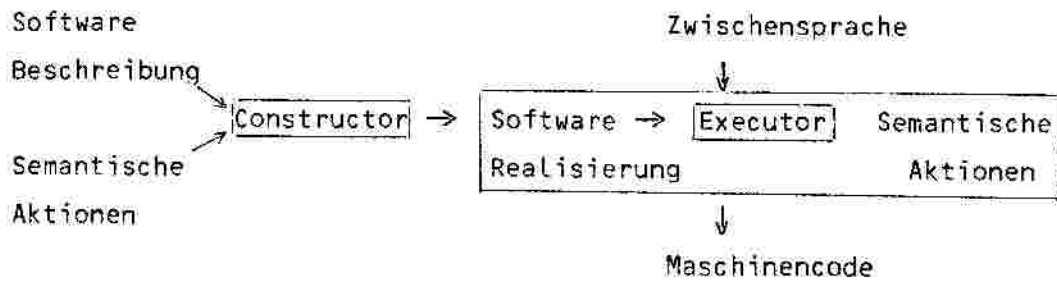


Abb.3.5.2: Erweitertes "Constructor-Executor" System

## TEIL II

### II.1 Einführung

Die folgenden Kapitel enthalten die Beschreibung eines Generatorsystems für Codeerzeugung. In Kap.2 haben wir gesehen, was wir unter Codeerzeugung verstehen. Kap.3 stellt vor, wie Codeauswahl als "pattern matching" realisiert wird und wie sich diese automatisieren läßt.

Wir betrachten die Codeerzeugung-Aufgabe als einen erweiterten "pattern matching"-Prozess (siehe Kap.3).

Der Codeerzeuger setzt sich zusammen aus einem erweiterten "pattern matcher" ("pattern matcher" plus semantische Aktionen, in Abb.3.5.1 als "Executor" bezeichnet) plus patterns in p-Code.

Die "pattern matcher"-Eingabe besteht aus P-Code Bäumen (siehe Kap.4). Um diese Eingabe zu durchsuchen braucht der "pattern matcher" P-Code "patterns". Diese werden für jede Zielmaschine generiert. Unsere Aufgabe besteht in der Generierung dieser maschinenabhängigen Komponente (P-Code "patterns") des Codeerzeugers. Der Generierungsvorgang wird, ähnlich der Codeerzeugung zur Compilierungszeit, zu einem "pattern matching"-Prozess reduziert. Um den erweiterten "pattern matcher" der Compilierungszeit von dem der Generierungszeit zu unterscheiden, bezeichnen wir den ersten mit **C-Executor** und den zweiten mit **G-Executor**.

Um Mißverständnisse zu vermeiden:

-Ein Vergleich mit dem Generierungsschema von Glanville (Abb.3.5.1) zeigt, daß es sich im diesen Fall um einem 2-stufigen Generierungsprozess handelt:

-Patterns werden in einem ersten Durchgang generiert, bevor sie (eventuell) z.B. einem Parsergenerator als Eingabe präsentiert werden, der daraus in einem zweiten Durchgang die Parse-Tafeln konstruiert, die schließlich vom C-Executor manipuliert werden.

Folgendes Schema beschreibt näherungsweise unseren Aufgabenbereich.

Ziel: Übertragung von P-Code auf wechselnde Zielmaschinen

Lösungsansatz: Codeerzeugung durch "pattern matching"

Benötigtes Werkzeug: erweiterter "pattern matcher" (C-Executor)  
+ P-Code "patterns"

Vorhandenes Werkzeug: erweiterte "pattern matchers" (C- und G-Executor)

Herzustellen: "patterns" in P-Code

Herstellungsmethode: Generierung mit Hilfe eines erweiterten  
"pattern matcher" (G-Executor)

Grob gesehen halten wir uns an den Algorithmus aus Abb.3.3.1. Die generierten P-Code "patterns" sind mit Maschineninstruktionen verbunden: Trifft ein "pattern", so wird die damit assoziierte Folge von Maschineninstruktionen ausgeschrieben. Die Information, die besagt, welche Folge von Maschinenbefehlen welchem P-Code "pattern" zugeordnet wird, halten wir in Paaren vom Typ (p;mc) fest. Wir nennen diese Paare **Transformationstupel**. Die erste Komponente "p" ist die Abkürzung für das P-Code "pattern" bestehend aus einem oder mehreren P-Code Befehlen. Die zweite Komponente "mc" steht für die mit p assoziierte Folge von Maschineninstruktionen.

Die Transformationstupel (p,mc) sind das generierte Produkt unseres Generatorsystems (vgl. Abb.3.5.2). Prinzipiell können wir jeden "pattern matcher" mit den entsprechenden Erweiterungen als G-Executor benutzen. Erwartungsgemäß hängt die Güte der Transformationstupel (p,mc) und somit des erzeugten Codes von der Qualität des G-Executors ab. Das Problem wird in Kap.8 angesprochen.

#### Bemerkung II.1:

Sämtliche Beispiele dieses Teils der Arbeit haben wir "zu Fuß" bearbeitet, ohne dabei einen bestimmten "Executor" im Auge zu haben. Die einzigen Kriterien, nach denen wir uns orientiert haben, waren die Kosten gemäß der "maximal munch"- (Kap.3) und "minimal munch" (Kap.8) Heuristika.

Um die "patterns" in P-Code zu generieren, braucht der Generator außer der Maschinenbeschreibung auch die Beschreibung von P-Code, von uns **p-Spezifikation** genannt. Die p-Spezifikation wird vom Systementwickler als ein maschinenunabhängiges Produkt ein für allemal festgelegt. Aus diesem Grund wird die p-Spezifikation zu einer festen Komponente des Generators. Die Generator-Eingabe besteht aus der Beschreibung der Zielmaschine, die wir **mc-Spezifikation** nennen. Mit Hilfe dieser Information

können wir das "Constructor-Executor"-Paar aus Abb.3.6 in unserem Fall wie folgt ausbauen:

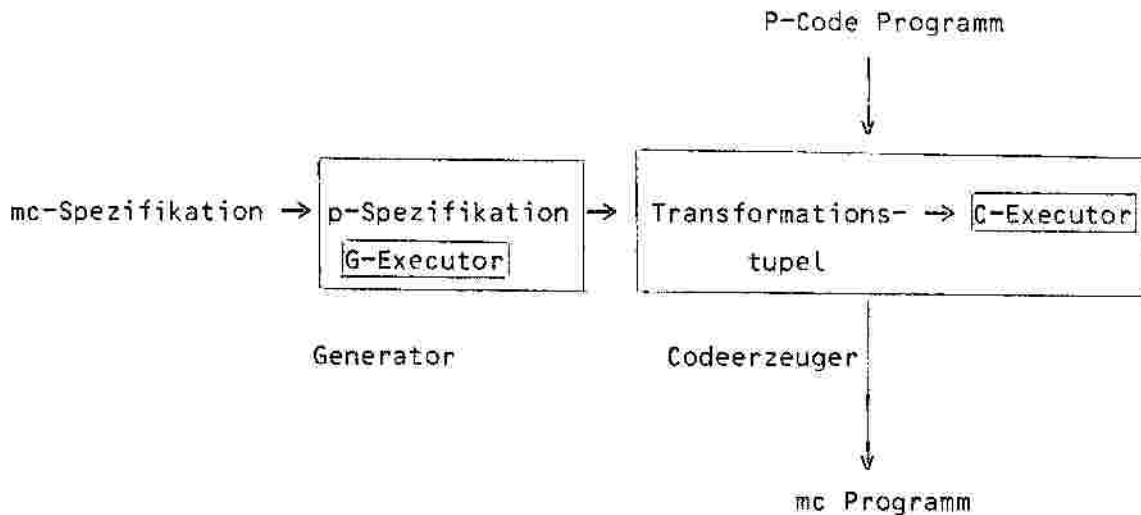


Abb.II.1: Codeerzeuger-Generator System

## II.2 Generierungsverfahren; Konstruktion der Transformationstapel

In diesem Abschnitt wollen wir auf die Konstruktion der Transformationstapel näher eingehen. Die Idee der Konstruktion ist einfach: Die Generierung der Transformationstapel wird einem Codeauswahl-Prozess gleichgestellt. In beiden Fällen, sowohl bei der Generierung von "patterns" als auch bei der Codeauswahl auf Übersetzerebene, haben wir es mit einem erweiterten "pattern matching"-Vorgang zu tun. Die "patterns" und die "pattern matcher"-Eingabe sind aber jedes Mal verschieden: Auf Übersetzerebene werden die p-Komponenten der Transformationstapel (p;mc) als "patterns" eingesetzt. Auf Generierungsebene werden einfachere "patterns" eingesetzt, die in der mc-Spezifikation und in der p-Spezifikationskomponente des Generators enthalten sind.

Die Schwierigkeit bei der Konstruktion der Transformationstapel (p;mc) liegt in dem unterschiedlichen "semantischen Aufbau" von P-Code und Maschinencode.

Wir haben P-Code und Maschinencode auf eine gemeinsame semantische Basis gebracht, indem wir von beiden Seiten, d.h. sowohl von P-Code als auch von Maschinencode, ausgegangen sind.

Der Hauptunterschied zwischen P-Code und Maschinencode ist die Tatsache, daß P-Code sich auf die obersten Stackadressen bezieht, Maschinencode dagegen auf Register (siehe auch [Ah U 77]: "The prime difference between intermediate code and assembly code is that the intermediate code need not specify the registers used for each operation"). Diese Beobachtung



haben wir hauptsächlich ausgenutzt, um die gemeinsame semantische Basis von P-Code und Maschinencode festzulegen.

Die P-Maschine (stack computer) wird zu einer ebenfalls virtuellen Maschine, **Abstrakte Register Maschine** genannt, umgeformt. Dieser Schritt ist in der p-Spezifikationskomponente des Generators enthalten. Der Generator liest die mc-Spezifikation (Beschreibung der Zielmaschine) ein und interpretiert sie als die **Konkretisierung** der Abstrakten Register Maschine.

Durch die Definition der Abstrakten Register Maschine sind wir von P-Code aus einen Schritt näher in Richtung Maschinencode gekommen. Die Abstrakte Register Maschine ist maschinenunabhängig wie P-Code, besitzt aber Register wie die Zielmaschine.

Umgekehrt sind wir durch die mc-Spezifikation von Maschinencode einen Schritt näher in Richtung P-Code gekommen.

Das Paar, bestehend aus der Maschineninstruktion und ihrer Beschreibung nennen wir **mc-Spezifikationstupel**, i.Z.  $(\pi; mc)$ . Die erste Komponente,  $\pi$ , ist die Beschreibung der Maschineninstruktion "mc". Den Beschreibungscode nennen wir **KRM** (wie konkretisierte Register Maschine), um deutlich auszudrücken daß hiermit die Konkretisierung der Abstrakten Register Maschine gemeint ist. Die zweite Komponente "mc" ist die Zielmaschineninstruktion in Assembler. KRM ist maschinenabhängig. Maschinenunabhängig ist dagegen **ARM**, der Code der Abstrakten Register Maschine.

Das Paar, bestehend aus einem P-Code Befehl "p" und den äquivalenten ARM-Befehlsfolgen "σ" nennen wir **p-Spezifikationstupel**, i.Z.  $(p; \sigma)$ .

Sind die p-Spezifikations- bzw. mc-Spezifikationstupel definiert, so brauchen wir uns nicht mehr (bis auf den Zusammenbau der Transformationstupel) mit P-Code und Maschinencode zu beschäftigen sondern nur mit deren äquivalenten Code ARM bzw. KRM. Damit reduziert sich die Frage: "Wie wird die Äquivalenz von P-Code und Maschinencode bestimmt?" zu: "Wie läßt sich die Abstrakte Register Maschine auf die Zielmaschine konkretisieren?" bzw. "Wie läßt sich die Zielmaschine abstrahieren?". **Abstrahieren** ist der umgekehrte Vorgang von Konkretisieren, d.h: Welche ARM-Befehlssequenz entspricht einer in KRM beschriebenen Maschineninstruktion?

Die Konkretisierung der Abstrakten Register Maschine wird vom Benutzer initiiert und von Generator fortgesetzt. Der Benutzer legt die Korrespondenz zwischen den (hauptsächlich primitiven) Datentypen von ARM und KRM fest durch Angabe der **primitiven Substitutionsfunktion**  $f_{\sigma}$ . Mit

Befehlsfolge umgesetzt wird. Das Ergebnis wird als ein Zweitupel, **Substitutionstupel** genannt, i.Z.  $(\sigma;\pi)$ , ausgegeben.

Gleichzeitig mit der Definition der primitiven Substitutionsfunktion  $f_\sigma$  wird die **primitive Abstraktionsfunktion**  $f_\pi$  bestimmt ( $f_\pi$  ist nicht die inverse Funktion von  $f_\sigma$ ). Die primitive Abstraktionsfunktion  $f_\pi$  legt für jeden betrachteten Datentyp der Zielmaschine fest, aus welchem Datentyp der Abstrakten Register Maschine dieser konkretisiert wurde. Analog zu  $(\sigma;\pi)$  baut der Generator die **Abstraktionstupel**  $(\pi;\sigma)$  auf. Ein Abstraktionstupel besagt, welche Folge von ARM-Befehlen auf einem KRM-Befehl abgebildet werden.

Aus den Substitutions- und Abstraktionstupeln erhalten wir die Transformationstupel  $(p,mc)$ , indem wir die  $\sigma$ -Komponente von  $(\sigma;\pi)$  und  $(\pi;\sigma)$  durch die äquivalente P-Code Sequenz aus den p-Spezifikationstupeln  $(p;\sigma)$  ersetzen bzw. die  $\pi$ -Komponente durch den äquivalenten Maschinencode aus den mc-Spezifikationstupeln  $(\pi;mc)$ . Aus diesem Grund nennen wir die  $(\sigma;\pi)$  und  $(\pi;\sigma)$ -Tupel auch **Hilfstupel**.

Sämtliche, in diesem Abschnitt eingeführten Tupel sind dadurch entstanden, daß zwei äquivalente Codesequenzen zusammengruppiert wurden. Wir bezeichnen sie alle als **Äquivalenzstupel**.

Schematisch sieht der oben dargestellte Vorgang folgendermaßen aus:

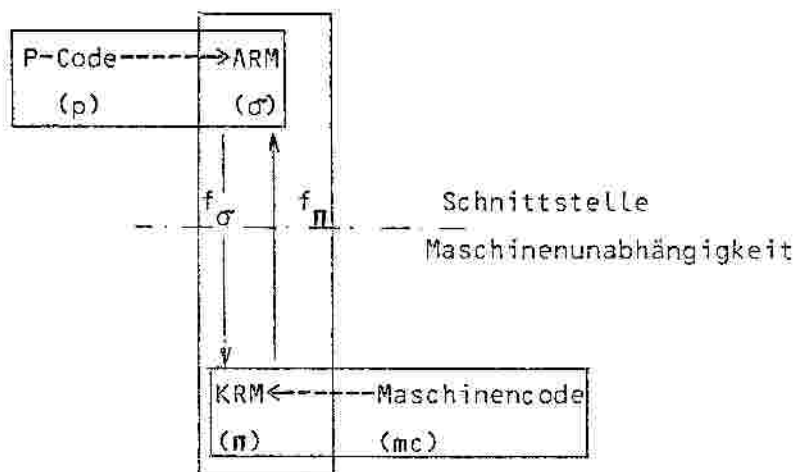


Abb.II.2.1: Konstruktion der Transformationstupel ; Prinzip

Der Abb.II.2.2.1 ist folgendes zu entnehmen:

- In Klammern sind die Abkürzungen angegeben, mit denen die verschiedenen Codes innerhalb der Äquivalenzstupel bezeichnet werden.
- Die Pfeile deuten auf die Definitionsrichtung der Äquivalenzstupel hin. So werden z.B. die p-Spezifikationstupel  $(p;\sigma)$  von P-Code aus definiert.
- Die Konstruktionsrichtung der Substitutions- und Abstraktionstupel

Richtung (von ARM nach KRM) definiert,  $(\pi; \sigma)$  in  $\pi$ -Richtung (von KRM nach ARM). In Kap.7 werden wir auf die Bedeutung der beiden Richtungen zu sprechen kommen. Die Abarbeitung in  $\sigma$ -Richtung sorgt dafür, daß der Codeerzeuger nicht blockiert. Die Abarbeitung in  $\pi$ -Richtung dagegen liefert "effiziente patterns". Damit können wir das Bild unseres Generierungssystems vervollständigen.

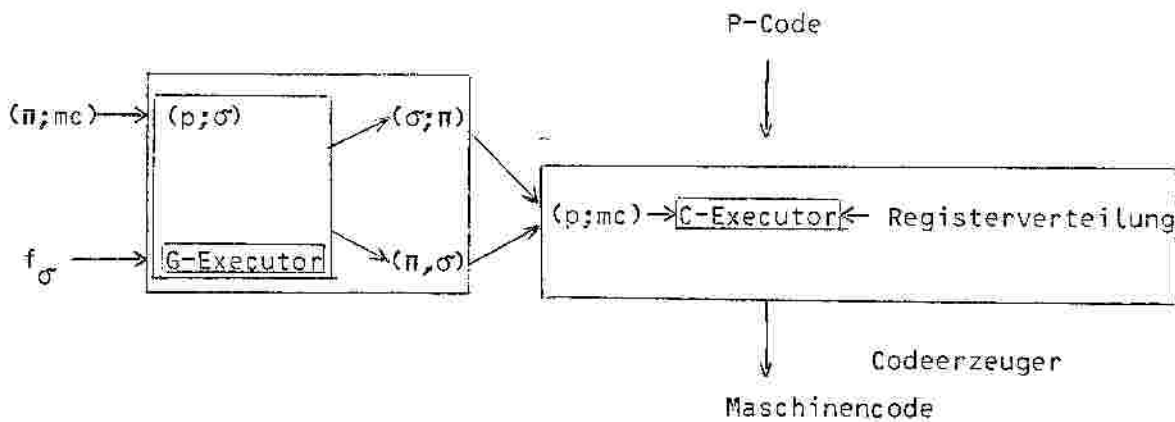


Abb. II.2.2 Codeerzeuger-Generator System; Aufbau

Die primitive Substitutionsfunktion  $f_\sigma$  wird als separate Eingabe des Generators angegeben. Die Registerverteilungsroutine wird als semantische Aktion innerhalb des obigen Generierungsschemas eingefügt (vgl. auch Abb.3.5.2). In den nächsten Kapiteln erklären wir die Funktionsweise und Zusammenarbeit der einzelnen Komponenten des Codeerzeuger-Generator Systems.

Kap.4 beschäftigt sich mit P-Code aus unserer Sicht, d.h. als Schnittstelle für Codeerzeugung. Kap.5 beschreibt die gemeinsame Eingabe für die Konstruktion von Substitutions- und Abstraktionstupeln. Die Konstruktion dieser Hilfstupel wird in Kap.6 und 7 dargestellt. Kap.8 behandelt Probleme, die sich bei der Konstruktion der Hilfstupel ergeben und deren Auswirkungen auf Übersetzerenebene. Kap.9 beschreibt einen Ansatz für die Erweiterung des Automatisierungskonzepts für Problem- und Ausnahmefälle.

Übersichtshalber fassen wir die definierten Äquivalenzstupel zusammen.

| Äquivalenzstapel; Komponenten des Codeerzeuger-Generator Systems |                        |  |
|--|------------------------|--|
| Abkürzung  | Bezeichnung            | Funktion innerhalb des Systems   |
| (p;σ)  | p-Spezifikationstapel  | spezifiziert P-Code in ARM;<br>P-Code Beschreibung                     |
| ((π;mc)  | mc-Spezifikationstapel | spezifiziert Maschinencode in<br>KRM; mc-Beschreibung                  |
| (σ;π)  | Substitutionstapel     | konkretisiert ARM; Verhindert<br>das Blockieren des Codeer-<br>zeugers |
| (π;σ)  | Abstraktionstapel      | abstrahiert KRM; sorgt für<br>"bessere patterns"                       |
| (p;mc)   | Transformationstapel   | generiertes Produkt; patterns<br>für Compilierungszeit                 |

## Kap.4 Codeerzeuger-Eingabe; P-Code als Schnittstelle zwischen front-end und back-end

### 4.1 Die P-Maschine

P-Code ist Objektcode für die virtuelle P-Maschine. Eine kompakte und übersichtliche Darstellung des "stack computers" befindet sich in [Pe 83]. Hier werden wir nur auf ein paar Eigenschaften eingehen, die für unsere Arbeit wichtig sind.

Der "stack computer" hat keine Arbeitsregister (Die 5 vorhandenen Register SP, NP, EP, MP, PC werden für die Verwaltung des Kellers, der Halde und als Programmzähler eingesetzt). Sämtliche Berechnungen werden auf den obersten Kellerelementen ausgeführt. Für jede Prozedur und Funktion wird auf dem Keller ein **Kellerrahmen** (stack frame) reserviert.

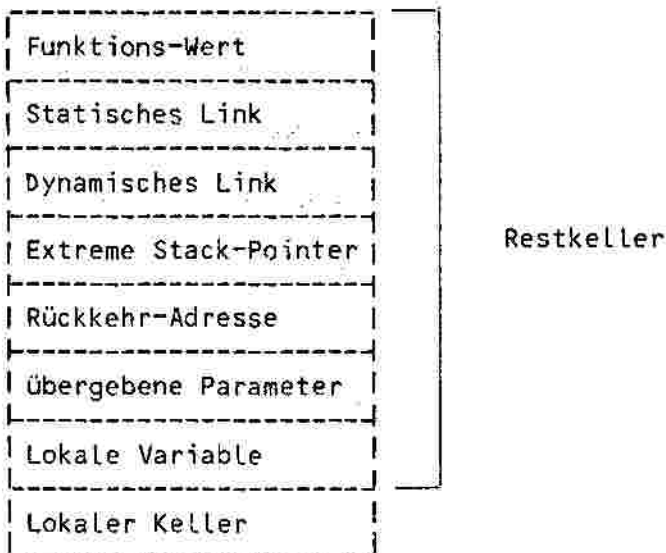


Abb.4.1: Kellerrahmen für Prozeduren und Funktionen

Die aktuelle Stackadresse wird im "stack pointer" Register SP aufbewahrt. Der **lokale Keller** (local stack) wird für die Ausführung von Operationen und für die Aufbewahrung von "temporaries" benutzt. Während der Abarbeitung einer Prozedur oder Funktion ändert sich der lokale Keller ständig. Im Gegensatz dazu ändert sich der Kellerabschnitt

Variable"-Adresse nicht. Diesen festen Kellerbereich nennen wir **Restkeller**. Auf dem lokalen Keller werden die Prozedurrümpfe behandelt, auf dem Restkeller dagegen der Verwaltungsmechanismus der Prozeduren (Aufruf, Rückkehr, Parameter-Übergabe und Abspeichern der lokalen Variablen). In dieser Arbeit behandeln wir P-Code Programmabschnitte entsprechend den Prozedurrümpfen (lokaler Keller). Ein Lösungsansatz für die Behandlung des Restkellers wird in Kap.9 skizziert.

#### 4.2 P-Code Regel; explizite Darstellung von P-Code Befehlen

Der PASCAL [P] Compiler liefert eine Wortkette von P-Code Befehlen spaltenweise aus ([No Am74]).

##### Beispiel 4.2.1

Die P-Code Befehle in Abb.4.2.1 entsprechen folgender "if"-Anweisung eines PASCAL-Programms:

if i≠n then i:=1 else n:=2

"i" und "n" sind globale, integer Variablen. "i" wird auf absoluter Adresse 10, "n" auf absoluter Adresse 9 aufbewahrt.

ldoi 10

ldoi 9

neqi

Abb.4.2.1: String-Darstellung eines P-Code  
Programmausschnitts

fjp l7

ldci 1

sroi 10

ujp l8

l7

ldci 2

sroi 9

l8

Diese Darstellung von P-Code Programmen nennen wir **String-Darstellung**.

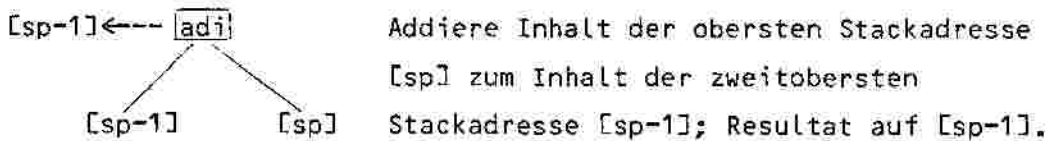
Die Darstellung von P-Code Befehlen aus der String-Darstellung eines P-Code Programms nennen wir **implizite Darstellung** von P-Code Befehlen. In der impliziten Darstellung erscheinen nur die Operatoren eines Befehls. Aus diesem Grund bezeichnen wir die implizite Darstellung eines P-Code Befehls als **P-Code Operator**.

Für unsere Zwecke empfiehlt sich eine explizitere Darstellung von P-Code als die obige Ausgabe des PASCAL [P] Compilers. Unsere explizite

Darstellung gibt zu jedem Operator die zugehörigen Operanden an.

Beispiel 4.2.2

Den P-Code "adi" Befehl (addiere integer) stellen wir wie unten dar:



Das "←" Symbol, sowie die linke- und rechte Seite der obigen "adi"-Beschreibung haben dieselbe Bedeutung wie in 3.2. Die Ähnlichkeit zwischen der Struktur des oben dargestellten "adi" P-Code Befehls und des "ADD Ri,Rj"-Befehls ist nicht zufällig; sie soll dem Systementwickler die Erstellung der p-Spezifikationskomponente des Generators erleichtern.

Definition 4.2 (p-Code Regel)

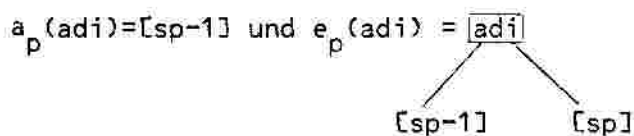
Diese explizite Darstellung eines P-Code Befehls "op" (wie Operator) nennen wir **P-Code Regel von op**, i.Z.  $r_p(op)$ . Die linke Seite nennen wir **Ausgabeschablone**, i.Z.  $a_p(op)$ , die rechte Seite **Eingabeschablone**, i.Z.  $e_p(op)$ .

Jedem P-Code Operator assoziieren wir eine P-Code Regel. Eine P-Code Regel ist nur eine andere Darstellung eines P-Code Befehls, in der explizit unterschieden wird zwischen Operator, Operanden und Lokalisierung des Resultats.

Beispiel 4.2.3

Für den "adi"-Befehl aus Bsp.4.2.2 erhalten wir:

$r_p(adi) := a_p(adi) ← e_p(adi)$  mit



Ähnlich wie im Abschnitt 3.2 ist die Eingabeschablone (rechte Seite)  $e_p(op)$  ein Baum, die Ausgabeschablone  $a_p(op)$  (linke Seite) dagegen stets ein Blatt.

### 4.3 P-Code Hilfs- und Ecktknoten

Die Blätter  $[sp-1]$  und  $[sp]$  der Eingabeschablone  $e_p(adi)$  aus Bsp.4.2.2 sind in der üblichen impliziten Darstellung (vgl. Bsp 4.2.1) von P-Code nicht identifizierbar. Aus diesem Grund nennen wir die von uns explizit eingeführten Knoten **Hilfsknoten**.

Die P-Code Befehle (Operatoren), die in der impliziten Darstellung erscheinen, nennen wir **Ecktknoten**.

Die Ecktknoten werden durch ein Rechteck gekennzeichnet.

Die Menge aller P-Code Regeln notieren wir mit  $R_p$ .

$R_p = \{r_p(op) \mid op \in EK_p\}$

mit  $EK_p = \{abi, abr, \dots, uni, xjp\}$ , Menge der Ecktknoten.

Der Zeichensatz  $Z_p$  der expliziten P-Code Darstellung besteht aus der Vereinigung der Eck- und Hilfsknoten von P-Code:

$Z_p = EK_p \cup HK_p$  mit  $HK_p = \{[sp], [sp-1], +, \dots\}$ , Menge der Hilfsknoten.

### 4.4 P-Code Transferregeln

Weiterhin beschreiben wir die Operatoren, die im Verlauf dieser Arbeit häufig vorkommen. Dies sind in erster Linie die Transferbefehle vom Typ "lade" (load) und "speichere" (store). Weitere in den nachfolgenden Beispielen erwähnte Befehle werden unmittelbar nach ihrem Auftreten erklärt. Alle P-Code Operationen, die auf dem lokalen Keller erfolgen, beziehen sich (höchstens) auf die obersten drei Stackadressen:  $[sp-2]$ ,  $[sp-1]$  und  $[sp]$ . Die Operanden werden durch "lade"-Befehle auf die oberen Stackadressen gebracht. Nach erfolgter Operation wird das Resultat auf die alte oder auf eine neue Adresse gespeichert.

In den folgenden Abschnitten dieses Kapitels haben wir sowohl die implizite (Operator) als auch die explizite (Regel) Darstellung von P-Code wiedergegeben.

#### 4.4.1 P-Code Regeln vom Typ "load"

Folgende drei P-Code Operatoren laden eine Adresse auf die oberste Stackposition.



| <u>P-Code Operator</u> | <u>P-Code Regel</u>                             | <u>Wirkung</u>                             |
|------------------------|---|--|
| lao q                  | $[sp+1] \leftarrow \boxed{\text{lao}} q$        | Lade absolute Adresse q                    |
| lca q                  | $[sp+1] \leftarrow \boxed{\text{lca}} q$        | Lade Adresse einer Konstante q             |
| lda p q                | $[sp+1] \leftarrow \boxed{\text{lda}} (B(p)+q)$ | Lade relative Adresse q gegenüber Niveau p |

Abb. 4.4.1.1 P-Code Regel vom Typ "Lade Adresse"

Anhand der P-Code Regeln aus Abb. 4.4.1.1 können wir folgendes feststellen:

- Der Stackpointer wird autoinkrementiert mit jedem "Lade"-Befehl. Die Ausgabeschablone [sp+1] gibt dies wieder.
- Die Eingabeschablonen bestehen nur aus einem Blatt.
- B(p) bezeichnet die Funktion "base" (vgl. auch P-Code Interpreter).
- B(p)+q steht für relative Adressierung: Parameter p ist die Schachtelungstiefe der aktuellen Prozedur, q ein konstantes "displacement" (siehe auch Kap.9).

Folgende drei Befehle laden den Inhalt einer Adresse (direkte Adressierung) oder einer Konstanten, die keine Adresse ist (unmittelbare Adressierung) auf die oberste Stackposition.

| <u>P-Code Operator</u> | <u>P-Code Regel</u>                              | <u>Wirkung</u>                                       |
|------------------------|--|--|
| ldoc q                 | $[sp+1] \leftarrow \boxed{\text{ldoc}} q$        | Lade Inh. der absoluten Adresse q                    |
| lodc p q               | $[sp+1] \leftarrow \boxed{\text{lodc}} (B(p)+q)$ | Lade Inh. der relativen Adresse q gegenüber Niveau p |
| ldcc q                 | $[sp+1] \leftarrow \boxed{\text{ldcc}} q$        | Lade Konstante q                                     |

Abb. 4.4.1.2 P-Code Regel vom Typ "Lade Konstante oder Inh. einer Adresse"

Der c-Parameter bezeichnet den Typ des Adressinhalts: a=adress, c=konstante, d=displacement

oben erwähnten Typen.

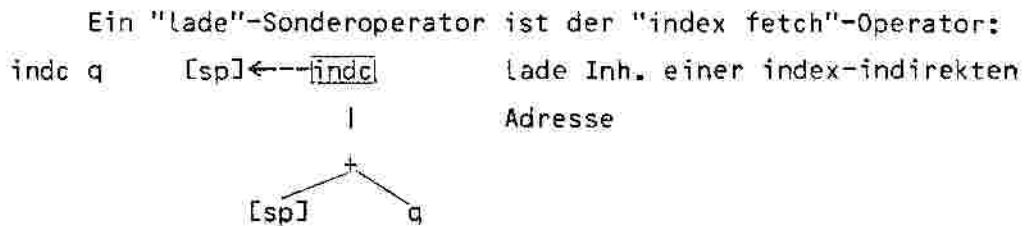


Abb. 4.4.1.3 P-Code Regel (indc q)

Mit diesem Operator ist keine Stackbewegung verbunden. Die Adresse, die sich aus dem Inhalt der obersten Stackposition + q berechnet, wird durch ihren Inhalt ersetzt.

Die P-Code Regeln  $r_p(\text{ldoc } p \ q)$  und  $r_p(\text{indc } q)$  sind syntaktisch ähnlich, unterscheiden sich aber in der Semantik. Beide Regeln laden den Inhalt einer Adresse, die auf unterschiedliche Art berechnet wird: Für

$r_p(\text{ldoc } p \ q)$  mit Hilfe von "static link" (Base-index Adressierung), für  $r_p(\text{indc } q)$  dagegen mittels der obersten Stackadresse [sp] (index-indirekte Adressierung).

Zur Notation von P-Code Regeln sei noch folgendes vermerkt:

Die Eingabeschablonen der P-Code Regeln vom Typ "lade Adresse" bzw. "lade Konstante" wurden als Blätter dargestellt (Abb. 4.4.1.1), die Eingabeschablonen der Regeln vom Typ "lade Inhalt einer Adresse" dagegen als Bäume (Abb. 4.4.1.2). Diese Schreibweise soll dem Systementwickler die Konstruktion der p-Spezifikationstupel  $(p; \sigma)$  erleichtern. Im Kap. 3 dieser Arbeit haben wir für die Beschreibung von Maschineninstruktionen vom Typ "lade unmittelbar" kein Symbol für den "lade"-Operator verwendet, wohl aber für Maschineninstruktionen vom Typ "lade indirekt" (" $\uparrow$ "-Symbol). Die Eckknoten der Eingabeschablonen von  $r_p(\text{ldoc } q)$  und  $r_p(\text{ldoc } p \ q)$  sind ähnlich wie die " $\uparrow$ "-Symbole aus 3.2 innere Knoten. Ein Vergleich mit den Befehlen aus 3.2 zeigt, daß die obersten Stackadressen die Rolle der Register eines herkömmlichen Rechners haben. Innerhalb der P-Code Beschreibung (p-Spezifikation) werden wir diese Beobachtung ausnutzen.

#### 4.4.2 P-Code Regeln vom Typ "store"

In diesem Fall haben wir nur mit Eingabeschablonen zu tun (vgl. auch den "store"-Befehl aus 3.2)

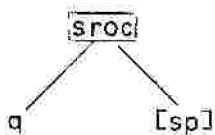
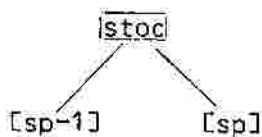
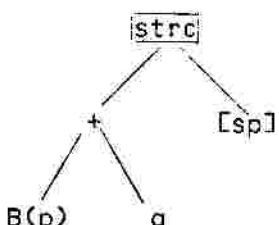
| <u>P-Code Operator</u> | <u>P-Code Regel</u>  | <u>Wirkung</u>   |
|------------------------|--|--|
| sroc q                 |   | speichere [sp] auf absolute Adresse q;                             |
| stoc                   |   | speichere [sp] auf absolute Adresse, die sich auf "sp-1" befindet; |
| str p q                |  | Speichere [sp] auf relative Adresse q gegenüber Niveau p;          |

Abb.4.4.2: P-Code Regeln vom Typ "store"

#### 4.5 P-Code Eingebäume

Wie im Kap.3 erwähnt haben wir uns für Bäume entschieden als Eingabe für den Codeerzeuger. Aus der String-Darstellung eines P-Code Programms werden die P-Code Bäume auf einfache Weise konstruiert unter Ausnutzung folgender zweier Beobachtungen:

- die Wurzel der Bäume sind "assignments"- und "jump"-Operatoren.
- die String-Darstellung eines P-Code Programms ist eine Folge von linearisierten P-Code Bäumen in Postdurchlauf.

Das Regelformat der Sprungoperatoren besteht, ähnlich wie die "store"-Regeln, nur aus Eingabeschablonen. Prozeduraufrufe- und Rückkehr Regeln ("call" und "return") werden ebenfalls als Sprünge behandelt (Näheres darüber in Kap.9).

#### Beispiel 4.5.1

Für den P-Code Programmabschnitt aus Bsp.4.2.1 ergibt sich folgende Folge von P-Code Bäumen:

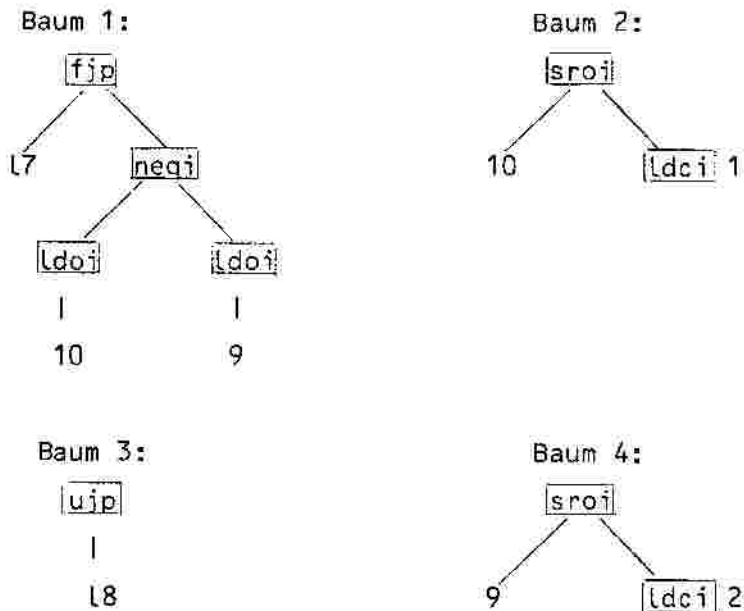


Abb.4.5: P-Code Wald für den Programmabschnitt aus Bsp.4.2.1

Die P-Code Regeln "neqi", "fjp" und "ujp" haben folgende Bedeutung:

| P-Code Regel   | Wirkung   |
|--|---|
| <pre> [sp-1] ← neqi       /  \      [sp-1] [sp]                     </pre> | logischer "not equal"-Operator;<br>prüft ob [sp-1] ≠ [sp],<br>Resultat nach "sp-1"; |
| <pre>       fjp      /  \     lx   [sp]                     </pre>         | "jump if false"<br>springe nach lx,<br>falls [sp-1] den Wert "false" hat;           |
| <pre>       ujp               lx                     </pre>                | unbedingter Sprung nach lx;   |

#### 4.6 Anforderungen an das front-end

Was die Korrespondenz zwischen P-Code String- und Baumdarstellung betrifft, gibt es eine Ausnahme: der "flo"-Operator (float next to the top, transformiert [sp-1] von "integer" in "real"). Der PASCAL [P] Compiler gibt den "flo"-Operator nicht unmittelbar (Postdurchlauf) nach Besetzung von "sp-1" aus, sondern erst nach Umwandlung von [sp] wie folgendes Beispiel zeigt:

Beispiel 4.6.1

Betrachten wir folgenden Ausdruck eines PASCAL-Programms:

$r := r + 1/i$  mit  $r, i$  globale Variable,  $r$  "real" und  $i$  "integer". Die Übersetzung in P-Code ergibt:

ldor 10  
ldci 1  
ldoi 9  
flt  
flo  
dvr  
adr  
srdr 10

oder als Baum

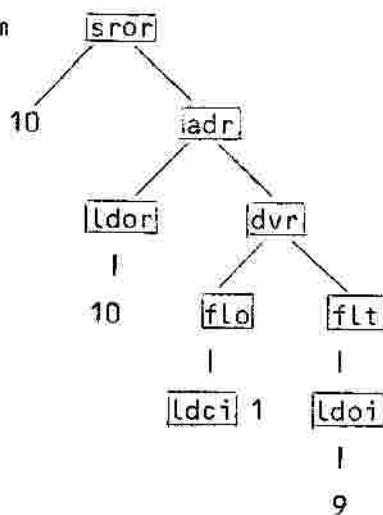


Abb.4.6: Behandlung des "flo"-Operators beim Übergang von der String- zur Baumdarstellung

Der "flt"-Operator hat dieselbe Wirkung wie der "flo"-Operator, bezieht sich aber auf [sp]. Bei der Konstruktion des P-Code Baums haben wir die Post-Durchlaufrichtung verletzt. Der Baum entspricht einem String, in dem sich "flo" zwischen "ldci 1" und "ldoi 9" befinden würde. Das Problem kann einfach beseitigt werden, indem wir von der front-end-Ausgabe P-Code Programme als eine Folge von Eingabebäumen fordern.

Bemerkung 4.6

In dieser Arbeit haben wir vorausgesetzt, daß der Pascal P-Compiler maschinenunabhängig ist. Prinzipiell ist dies möglich, ist aber in der aktuellen Implementierung von Nori und Ammann nicht der Fall (siehe die Definition der Size- und Alignment-Bedingungen in [No Am 74]).

## Kap.5 Eingabe des Konstruktionalgorithmus für die Hilfstupel

### 5.1 Einführung

Für die Konstruktion der Hilfstupel  $(\sigma; \pi)$  und  $(\pi; \sigma)$  untersuchen wir wie ARM, Code der Abstrakten Register Maschine, auf KRM, Code der konkretisierten Register Maschine, abgebildet wird und umgekehrt. ARM und KRM werden durch die Beschreibung von P-Code und Maschinencode definiert, d.h. durch Angabe der p-Spezifikations- und mc-Spezifikationstupel  $(p; \sigma)$  bzw.  $(\pi; mc)$ . Zusammen mit der vom Benutzer angegebenen primitiven Substitutionsfunktion  $f_{\sigma}$  bilden  $(p; \sigma)$  und  $(\pi; mc)$  die Eingabe des Konstruktionalgorithmus für die Hilfstupel.

#### Bemerkung 5.1:

Von ihrer Funktion her sind die p-Spezifikationstupel  $(p; \sigma)$  ein Teil der Eingabe des Konstruktionalgorithmus, werden aber auf dem Schema aus Abb.II.2.2 als feste Komponente des Generators betrachtet, da sie vor der Generierungszeit bekannt sind.

### 5.2 p-Spezifikation; Abbildung des Stack-computers auf der Abstrakten Register Maschine

#### 5.2.1 Definition der Abstrakten Register Maschine

Wir wollen P-Code und die p-Maschine für unsere Zwecke "maschinenfreundlicher" (nicht zu verwechseln mit maschinenabhängig) gestalten mit möglichst wenig Änderungen. Bei der Definition der Abstrakten Register Maschine versuchen wir so dicht wie möglich an der Semantik der p-Maschine zu bleiben.

Die Abstrakte Register Maschine besteht aus Registern und Keller. Die Register nennen wir **abstrakte Register**. Ausgehend vom "stack computer" wird die Abstrakte Register Maschine unter Berücksichtigung folgender zwei Postulate definiert:

1) sämtliche Änderungen des lokalen Stacks (vgl. 4.1) werden auf abstrakten Registern simuliert.

2) der Reststack des "stack computers" wird auf dem Keller der Abstrakten Register Maschine simuliert (Kap.9).

In diesem Kapitel behandeln wir nur P-Code Regeln, die sich auf den lokalen Stack beziehen. Dies reicht aus, um die Konstruktion der Transformationstapel zu verfolgen. Ein Vorschlag für die Behandlung des Restkellers wird in Kap.9 skizziert.

### 5.2.2 p-Spezifikationstapel

ARM, Code der Abstrakten Register Maschine wird durch die Beschreibung der P-Code Regel definiert. Jeder P-Code Regel wird dabei eine ARM-Regel assoziiert. Diese Korrespondenz wird vom Systementwickler in den p-Spezifikationstapel  $(p; \sigma) \in \Sigma$  festgehalten, wobei  $\Sigma$  die Menge der p-Spezifikationstapel bezeichnet.

In diesem Abschnitt geben wir die p-Spezifikationstapel an für die P-Code Transferregeln aus 4.4.1 und 4.4.2. Weitere ARM-Regeln, die im Verlauf dieser Arbeit auftreten, werden unmittelbar nach ihrem Erscheinen erläutert.

#### 5.2.2.1 p-Spezifikationstapel für P-Code Regeln vom Typ "Load"

| <u>P-Code Regel</u>                  | <u>ARM-Regel</u>                             | <u>Wirkung</u>   |
|--------------------------------------|--|--|
| $[sp+1] \leftarrow [la] q$           | $ADRREG \leftarrow ABSADR$                   | Lade absolute Adresse ABSADR auf Adressregister ADRREG |
| $[sp+1] \leftarrow [lca] q$          | $ADRREG \leftarrow ADRKONS$                  | Lade Adresse einer Konstante                           |
| $[sp+1] \leftarrow [lda] (B(p)+q)$   | $ADRREG \leftarrow BXADR(p,q)$               | Lade relative Adresse $BXADR(p,q)$                     |
| $[sp+1] \leftarrow [ldcc] q$         | $REGIS \leftarrow KONS$                      | Lade Konstante auf Register REGIS                      |
| $[sp+1] \leftarrow [ldoc]$<br> <br>q | $REGIS \leftarrow   \uparrow$<br> <br>ABSADR | Lade Inh.einer absoluten Adresse                       |

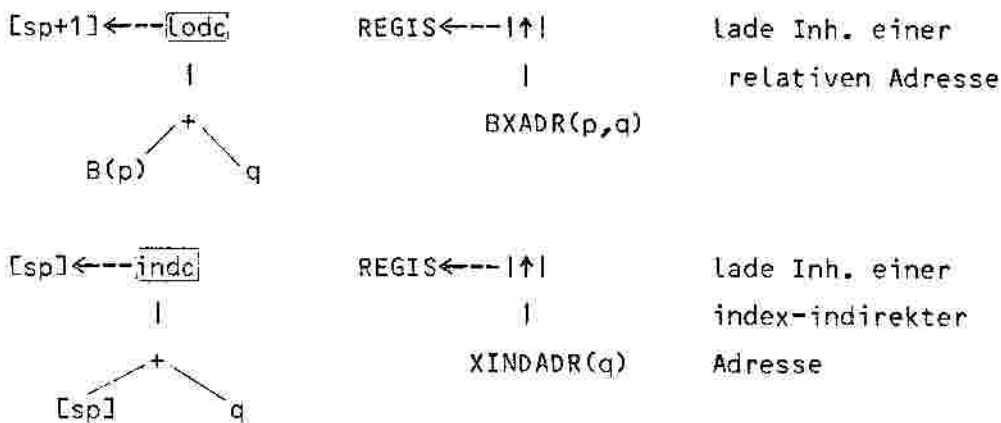


Abb. 5.2.2.1 p-Spezifikationstupel für "Load"-Regeln

Zu den obigen Regeln ist folgendes zu präzisieren:

- ARM-Zeichensatz wird in Großbuchstaben angegeben;
- ARM-Operatoren werden von zwei vertikalen Strichen umrandet;
- Die Register der Abstrakten Register Maschine sind in nicht notwendig disjunkten Klassen eingeteilt: so bezeichnet z.B.
- ADRREG eine Klasse von abstrakten Registern, deren Inhalt Adressen sind und
- REGIS die Klasse aller Register der Abstrakten Register Maschine;
- Ein- und Ausgabeschablone sowie das " $\leftarrow$ "-Symbol haben dieselbe Bedeutung wie in den P-Code Regeln.

Sei  $Z_\sigma$  der Zeichensatz der Abstrakten Register Maschine:  $Z_\sigma = \{ | \uparrow |, | + |, \dots, KONS, ADRREG, \dots \}$ . Solange wir kein konkretes Element einer Registerklasse zu präzisieren brauchen, können wir diese Registerklasse als einen primitiven Datentyp der Abstrakten Register Maschine betrachten, dargestellt durch ein Element aus  $Z_\sigma$ . Solange wir z.B. kein Sonderregister (etwa Basisregister) innerhalb der ADRREG spezifiziert haben, können wir ADRREG als einen primitiven Datentyp betrachten.

Ist DATAREG die Klasse der Register, die keine ADRREG sind, so gilt:

$\underline{REGIS} = \{ ADRREG, DATAREG \}$  und  $ADRREG \in Z_\sigma$ ,  $\underline{REGIS} \in 2^{Z_\sigma}$  und  $REGIS \in \underline{REGIS}$ , ein Bezeichner für ein Element der Menge aller Registerklassen.

Zusammengesetzte Datentypen werden als Mengen aufgefasst. Die Mengennamen werden unterstrichen.

Die Adressierungsarten "relative Adressierung"  $BXADR(p,q)$  und index-indirekte Adressierung  $XINDADR(q)$  werden nicht näher spezifiziert (sondern eher verallgemeinert) innerhalb des p-Spezifikationsschrittes, da die Substitutionen auf diese Adressierungsarten im P-Code



maschinenunabhängig zu halten (siehe auch Diskussion in 1.5).

Man kann sich  $BXADR(p,q)$  und  $XINDADR(q)$  als Makros vorstellen, die maschinenabhängig expandiert werden.  $BXADR(p,q)$  wird z.B. auf der PDP11/70 mittels eines konstanten "displacements" und eines Basisregister realisiert, auf der IBM 370 dagegen mit Hilfe zweier Register (Basis- und Adressregister) und eines konstanten displacements.

### 5.2.2.2 p-Spezifikationstupel für P-Code Regeln vom Typ "store"

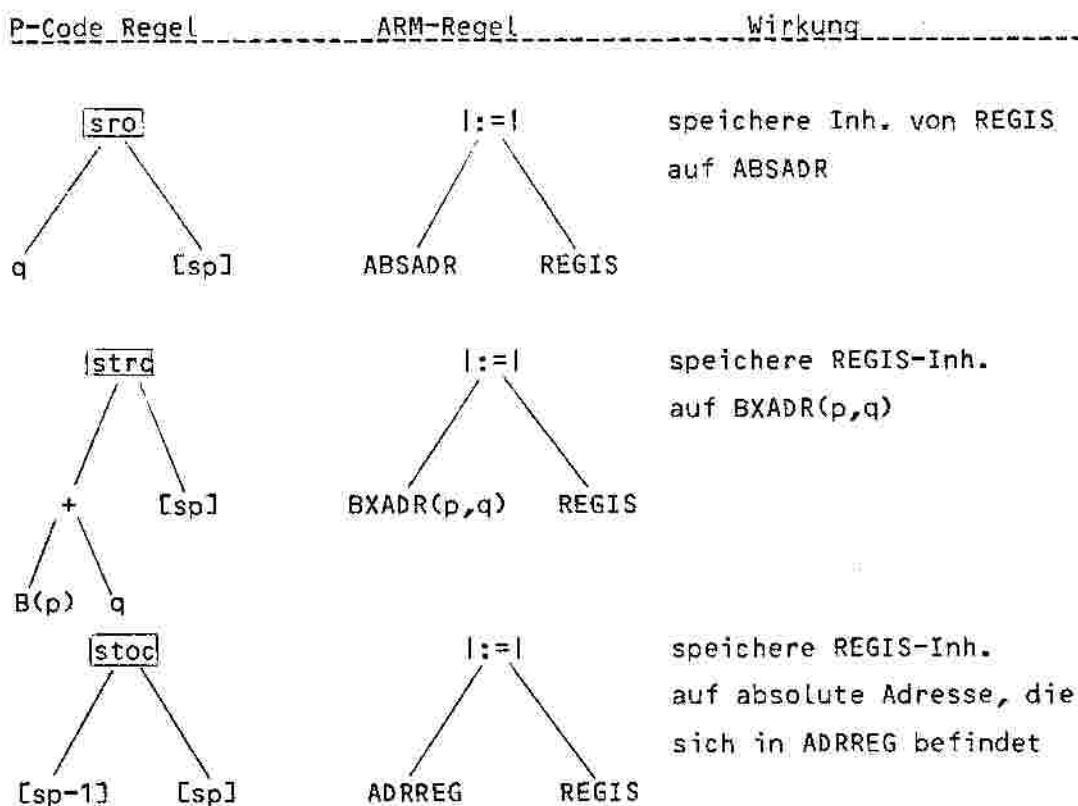


Abb.5.2.2.2: p-Spezifikationstupel für P-Code "store"-Regeln

Abb.5.2.2.2 zeigt, daß die ARM-Regeln vom Typ "store" nur aus Eingabeschablonen bestehen. In ARM, im Gegensatz zur expliziten P-Code Darstellung, wird nicht mehr zwischen Echt- und Hilfsknoten differenziert.

Die zu einem P-Code Operator "op" assoziierte ARM-Regel schreiben wir als:  $r_{\sigma}(op) := a_{\sigma}(op) \leftarrow e_{\sigma}(op)$ .

Die Menge der ARM-Regeln notieren wir mit  $R_{\sigma}$

$R_{\sigma} = \{r_{\sigma}(op) \mid op \in EK_p\}$  mit  $EK_p$  Menge der P-Code Eckknoten. Somit gilt:  $|R_{\sigma}| = |R_p|$ , d.h. eine "1 zu 1"-Korrespondenz zwischen den P-Code Eckknoten und ARM-Regeln.

Diese Annahme ist nicht immer realistisch. Wie wir im Kap.9 sehen werden, erfordert die p-Spezifikation mancher P-Code Operatoren eine Folge

von ARM-Regeln. Die Darstellung des Konstruktionsalgorithmus für die Substitutions- und Abstraktionstupel aus Kap.6 und 7 ist durch diesen vereinfachten Ansatz nicht berührt.

#### Bemerkung 5.2.2.2:

Um die Diskussion zu vereinfachen haben wir den "c"-Parameter aus den P-Code Regeln weggelassen. In Wirklichkeit übernehmen wir den "c"-Parameter in die ARM-Notationen. So schreiben wir z.B. REGIS ←-- KONSc für die "ldcc q"-Regel, wobei "c" dieselbe Bedeutung wie in P-Code (4.4.1) hat.

Im nächsten Abschnitt stellen wir die Beschreibung der Zielmaschine als Eingabe für den Generator der Hilfstupel dar.

### 5.3 mc-Spezifikationstupel für den SAB 8080-Prozessor

Für die Beschreibung der Zielmaschine haben wir eine, dem IR (intermediate representation) von Glanville (siehe [Gl 77], Kap.3: The Target Machine Description), ähnliche Notation ausgewählt, einerseits weil wir damit vertraut sind, andererseits weil IR mittlerweile sich als eines der einfachsten nicht-prozeduralen Beschreibungsmittel gezeigt hat. Dies schließt allerdings nicht aus, daß die mc-Spezifikation auch durch andere Beschreibungscode angegeben werden kann. Wir denken dabei an die Maschinenbeschreibungen von Henry und Aigrain ([Gr He 84]), die eine viel genauere semantische Beschreibung der Maschineninstruktionen erlauben, allerdings mit mehr Aufwand.

Die Menge der mc-Spezifikationstupel bezeichnen wir mit  $\mathbb{M}$ . Die mc-Spezifikationstupel  $(\pi; mc)$  enthalten die Beschreibung der Zielmaschine. Der Generator interpretiert die  $(\pi; mc)$ -Tupel als die Substitution der Abstrakten Register Maschine. Aus diesem Grund nennen wir den Beschreibungscode der Zielmaschine  $KRM$  (konkretisierte Register Maschine).

In diesem Abschnitt werden wir einige häufig verwendete Maschineninstruktionen des SAB 8080-Prozessors beschreiben.

Zunächst einige Worte über den SAB 8080. Für eine ausführliche Beschreibung siehe Kapitel "Aufbau des Mikroprozessors SAB 8080" aus dem SAB 8080 Siemens-Manual [SAB 78].

SAB 8080 ist ein 8 Bit Mikroprozessor der über folgende Ressourcen

verfügt:

- 7 Arbeitsregister (A, B, C, D, E, H, L), aus denen drei Registerpaare (Doppelregister) zusammengesetzt werden können (BC, DE, HL);
- Befehlszähler PC (program counter) und Stackpointer SP.

Der Speicherraum ist folgendermaßen organisiert:

- Der Programmspeicher (ROM) enthält Anwendungsprogramme und Konstanten;
- Der Datenspeicher (RAM) enthält Daten und Zwischenergebnisse (temporaries);
- 256 ( $2^8$ ) Datenkanäle (ports) für Datenaustausch zwischen Mikroprozessor und seiner Umgebung.

Das Registerpaar PSW (program status word) setzt sich zusammen aus dem Akkumulator A und einem Sonderbyte. PSW beschreibt den momentanen Zustand der Maschine; das Sonderbyte entspricht in etwa dem "condition code"-Register anderer Prozessoren.

Durch den 16 Bit Adress-Bus können 65536 ( $2^{16}$ ) Bytes (1/2 Worte) adressiert werden. Die Speicheradressierung erfolgt unmittelbar, direkt oder indirekt durch den SP oder eines der Registerpaare BC, DE oder HL. Die Adressierung mit Hilfe des HL-Registerpaares wird implizite Adressierung genannt. Der Name kommt daher, daß "HL" nicht explizit im Operandenteil der Maschineninstruktion auftaucht sondern impliziter Bestandteil des Befehlscodes ist.

Die Zielmaschineninstruktionen werden in der Assembler-Notation des Herstellers angegeben. Die Wirkung der Maschineninstruktion wird aus Platzgründen unmittelbar unterhalb der Assembler-Notation angegeben.

Assembler

KRM-Regel

1) lxi adrreg,a16

adrreg←---a16

-Lade ein Element aus der Menge adrreg mit absoluter Adresse a16 (unmittelbare Adressierung);

2) mvi r,k8

r←---k8

-Lade ein Element aus r mit Konstante k8 (unmittelbare Adressierung);

3) mov r,(HL)

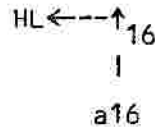
r←---↑<sub>8</sub>

|

HL

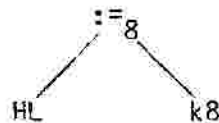
-Lade ein Element aus r mit dem Inh. der Adresse, die sich in HL befindet (implizite Adressierung);

4) lhld a16



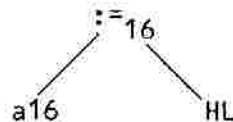
-Lade HL mit Inhalt von Adressen "a16" und "a16+1" (direkte Adressierung);

5) mvi (HL),k8



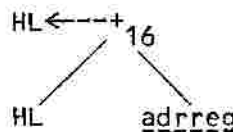
-Speichere eine Konstante k8 auf der Adresse, die sich in HL befindet (implizite Adressierung);

6) shld a16



-Speichere HL-Inh. unter Adressen "a16" und "a16+1" (direkte Adressierung);

7) dad addrreg



-Addiere zu HL-Inhalt ein Registerpaar aus addrreg, Resultat in HL.

Abb.5.3: mc-Spezifikationstupel für einige SAB 8080 Instruktionen

Die KRM-Symbole aus Abb.5.3 haben folgende Bedeutung:

$r \in \underline{r} = \{A, B, C, D, E, H, L\}$ ,  $\text{addrreg} \in \underline{\text{addrreg}} = \{HL, BC, DE, SP\}$ , d.h.  $r$  und  $\text{addrreg}$  sind stellvertretende Bezeichner für Elemente aus  $\underline{r}$  bzw.  $\underline{\text{addrreg}}$ .

a16 ist 16 Bit Adresse, k8 8 Bit Konstante.

Wir unterscheiden zwischen Operatoren, die sich auf Byte (8 Bits) oder auf "Wort" (16 Bits) beziehen. So bedeutet z.B. " $\uparrow_8$ " lade Byte und " $\uparrow_{16}$ " lade Wort.

Mit  $Z_{n8080}$  bezeichnen wir den Zeichensatz der KRM-Regeln des SAB 8080.  $Z_{n8080} = \{k8, a16, A, B, \dots, HL, \dots\}$  ("terminals" und "nonterminals" bei Glanville) und es gilt:

$k8, a16, A, \dots \in Z_{n8080}$ , während  $\underline{r}, \underline{\text{addrreg}} \in 2^{Z_{n8080}}$ .

Bemerkung 5.3:

Der Einfachheit halber schreiben wir künftig einfach  $Z_{\Pi}$  statt  $Z_{n8080}$ , behalten aber im Auge daß  $Z_{\Pi}$  maschinenabhängig ist.

Mit jeder Maschineninstruktion "ins" wird eine KRM-Regel assoziiert. For-

mal schreiben wir:

$r_{\Pi}(ins) := a_{\Pi}(ins) \leftarrow e_{\Pi}(ins)$  mit  $a_{\Pi}(ins)$  und  $e_{\Pi}(ins)$  Ein- und Ausgabeschablone der KRM-Regel  $r_{\Pi}(ins)$ . Die Menge aller KRM-Regel schreiben wir als:

$R_{\Pi} = \{r_{\Pi}(ins) \mid ins \in Z_{mc}\}$  wobei  $Z_{mc}$  das Zeichensatz der Befehlssteile (Assembler-Mnemonic) der Maschineninstruktionen ist.  $|Z_{mc}|$  ist die Anzahl der beschriebenen Maschineninstruktionen.

Das KRM-Zeichensatz  $Z_{\Pi}$  ist verschieden vom Assembler-Zeichensatz  $Z_{mc}$ . Einem Zeichen aus  $Z_{mc}$  können mehrere Zeichen aus  $Z_{\Pi}$  entsprechen. Der SAB 8080-Instruktion "Lhld a16" entsprechen die Symbole HL,  $\uparrow_{16}$ , a16 wie aus der Abb.5.3 zu sehen ist. Dasselbe Situation tritt bei der p-Spezifikation auf: Die Menge der P-Code Eckknoten  $EK_p$  (implizite Darstellung) ist verschieden von  $Z_p$ , Zeichensatz der expliziten P-Code Darstellung ( $Z_p = EK_p \cup HK_p$ ).

Wir haben uns innerhalb dieser Arbeit auf eine sehr primitive und direkte Umformung von p- (p-Spezifikation) und Maschinencode (mc-Spezifikation) beschränkt. Es ging uns in erster Linie um die syntaktische Annäherung von p- und Maschinencode.

#### 5.4 Primitive Substitutionsfunktion

Die primitive Substitutionsfunktion  $f_{\sigma}$  bestimmt wie ARM-Datentypen auf der Zielmaschine konkretisiert werden. Sie wird vom Systembenutzer als separate Eingabe dem Generator zugefügt (vgl. Abb.II.2.2). Die primitive Substitutionsfunktion  $f_{\sigma}$  ist auf einer Untermenge  $D_{\sigma}$  von  $2^{Z_{\sigma}}$ , Potenzmenge des ARM-Zeichensatzes, definiert mit Werten in einer Untermenge  $W_{\Pi}$  von  $2^{Z_{\Pi}}$ , Potenzmenge des KRM-Zeichensatzes einer Zielmaschine.

Folgende Abbildung enthält einen Ausschnitt aus der Substitution der Abstrakten Register Maschine auf dem SAB 8080.

| $f_{\sigma}$                          |                                       |
|---------------------------------------|---------------------------------------|
| $D_{\sigma} \subseteq 2^{Z_{\sigma}}$ | $W_{\Pi} \subseteq 2^{Z_{\Pi 8080}}$  |
| {KONS}                                | {k8, a16}                             |
| {ABSADR}                              | {a16}                                 |
| {ADRKONS}                             | {a16}                                 |
| {ADRREG}                              | addrreg                               |
| {DATAREG}                             | r                                     |
| REGIS                                 | regis                                 |
| {l:=l}                                | {:= <sub>8</sub> , := <sub>16</sub> } |

Abb.5.4 Ausschnitt aus der Substitutionsfunktion für den SAB 8080-Prozessor

Wie aus Abb.5.4 zu sehen ist, sind, mit Ausnahme von REGIS  $\in 2^{Z_{\Pi 8080}}$ , alle Argumente der Funktion  $f_{\sigma}$  primitive Datentypen der Abstrakten Register Maschine. Aus diesem Grund haben wir  $f_{\sigma}$  primitive Substitutionsfunktion genannt.

Die Anwendung der Funktion  $f_{\sigma}$  bedeutet für den Benutzer die Antwort auf die Frage: Welche Korrespondenten hat ein Datentyp der Abstrakten Register Maschine auf der Zielmaschine?

Für eine n-lementige Menge  $\{a_1, \dots, a_n\}$ , entsprechend einem zusammengesetzten Datentyp (wie REGIS z.B.), gilt:

$$f_{\sigma}(\{a_1, \dots, a_n\}) = f_{\sigma}(\{a_1\}) \cup \dots \cup f_{\sigma}(\{a_n\}).$$

Beispiel 5.4

$$\text{REGIS} = \{\text{ADRREG}\} \cup \{\text{DATAREG}\}$$

$$f_{\sigma}(\text{REGIS}) = f_{\sigma}(\{\text{ADRREG}\}) \cup f_{\sigma}(\{\text{DATAREG}\}) = \text{addrreg} \cup \{r\} = \text{regis}.$$

Aus Abb.5.4 ergibt sich, daß der SAB 8080 nicht zwischen der Adresse einer Konstanten ADRKONS und der absoluten Adresse ABSADR zu unterscheiden braucht:

$$(f_{\sigma}(\{\text{ADRKONS}\}) = f_{\sigma}(\{\text{ABSADR}\}) = \{a16\}).$$

Daraus darf nicht geschlossen werden, daß ADRKONS und ABSADR allgemein äquivalent sind. Auf andere Zielmaschinen, Siemens 7700 z.B., ist  $f_{\sigma}(\{\text{ADRKONS}\}) \cap f_{\sigma}(\{\text{ABSADR}\}) = \{\}$ ; in diesem Fall wird ADRKONS mit Hilfe relativer Adressierung realisiert. Wir dürfen aber behaupten, daß

ADRKONS und ABSADR äquivalent sind bzgl. der Substitution auf dem SAB 8080-Prozessor (siehe auch Diskussion über Implementierung in 1.5.2).

Die primitive Substitutionsfunktion  $f_{\sigma}$  ordnet Maschinenressourcen (KRM) zu P-Code Ressourcen (ARM). Die Angabe der primitiven Substitutionsfunktion  $f_{\sigma}$  stellt keine große Anforderungen an den Benutzer. Der Aufwand dafür ist, in den meisten Fällen, in etwa vergleichbar mit der Angabe der ersten drei Sektionen (#registers, #variables, #terminals) aus der Zielmaschinen-Beschreibung von Glanville ([GL 77]). Schwieriger wirds für einfache Prozessoren mit starken nicht-orthogonalen Befehlssatz. So wird z.B. der ARM Multiplikationsoperator [\*] auf dem SAB 8080 durch ein Unterprogramm substituiert. In der Regel darf der Benutzer in solchen Situationen mit Standard-Unterstützung vom Hersteller rechnen.

Der Systementwickler muß dem Benutzer den kommentierten Zeichensatz  $Z_{\sigma}$  der Abstrakten Register Maschine zur Verfügung stellen. Dies bedeutet aber nicht, daß der Benutzer etwas über die p-Spezifikation von P-Code zu wissen braucht.

## Kap.6 Konstruktion der Substitutionstapel

### 6.1 Baumrepräsentation

In diesem Kapitel verfolgen wir die Konstruktion der Substitutionstapel  $(\sigma, \pi) \in F_{\sigma}$  mit  $F_{\sigma}$  Menge der Substitutionstapel. Dafür ist es notwendig, die primitive Substitutionsfunktion  $f_{\sigma}$  auf Bäume zu erweitern. Zu diesem Zweck führen wir in diesem Abschnitt die benötigten Begriffe ein.

Wir benutzen folgende Notationen:

Sei  $K$  ein Knoten aus der Knotenmenge  $KN$  eines Baums  $B$ .

$NF(K)$  ist die Menge der unmittelbaren Nachfolger von  $K$ .

$|NF(K)|$  ist Ausgangsgrad von  $K$  (gibt Stelligkeit des Knotens an, d.h. Anzahl der Operanden).

#### **Dewey-Repräsentation**

Wir betrachten Bäume in Dewey-Notation (siehe [Zi 83]).

Die Struktur des Baums  $B$  geben wir durch die Dewey-Notation des Baums an, i.Z.  $DN(B) = \{d_i | 1 \leq i \leq |KN|\}$ , mit  $d_i$  Dewey-Zahl der Baumknoten.

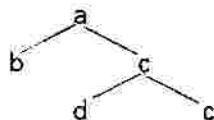
Mit  $DR(B)$  bezeichnen wir die Darstellung eines Baums  $B$ .

$DR(B) = \{(d_i, M_i) | 1 \leq i \leq |KN|\}$ , wobei  $d_i$  die Dewey-Zahl des Knotens  $K_i$  und  $M_i$  die zugehörige Knotenmarkierung ist.

$DR(B)$  nennen wir Dewey-Repräsentation eines Baums  $B$ .

#### Beispiel 6.1.1

Folgender Baum  $B$ :



wird als  $DR(B) = \{(1, a), (1.1, b), (1.2, c), (1.2.1, d), (1.2.2, c)\}$  dargestellt.

Die Baumstruktur wird durch:

$DN(B) = \{1, 1.1, 1.2, 1.2.1, 1.2.2\}$  wiedergegeben.

Die Dewey-Notation ist eine Prefixnumerierung. Sind die Knoten eines Baums  $B$  in Dewey-Notation durchnummeriert, so ist die Struktur



dieses Baums aus seiner linearisierten Form eindeutig rekonstruierbar.

### Strukturelle Gleichheit

Zwei Bäume  $B$  und  $B'$  sind strukturell gleich falls  $B$  und  $B'$  die gleiche Dewey-Notation besitzen, i.Z.  $DN(B)=DN(B')$ .

Strukturelle Gleichheit von Bäumen gilt unabhängig von den Knotenmarkierungen.

### Positionsgleichheit

Zwei Knoten von zwei verschiedenen Bäumen (nicht unbedingt strukturell gleich) sind positionsgleich, falls sie von derselben Dewey-Zahl identifiziert werden.

Sind die zugehörigen Markierungen ebenfalls gleich, so sagen wir, daß die Knoten identisch sind.

### Baumidentität

Zwei Bäume  $B$  und  $B'$  sind identisch, falls  $DR(B)=DR(B')$ , d.h.  $B$  und  $B'$  haben dieselbe Struktur und positionsgleiche Knoten haben dieselbe Markierung.

### Bemerkung 6.1.1:

Die Knotenmarkierungen der Bäume, die wir künftig betrachten werden, sind Mengen bestehend aus den Argumenten und Werten der primitiven Substitutions- und Abstraktionsfunktion (Kap.7). Aus diesem Grund schreiben wir künftig die Knotenmarkierung als Mengen  $\underline{M}$ .

In diesem Zusammenhang unterscheiden wir:

### Knotenzugehörigkeit

Für positionsgleiche Knoten stellt sich die Frage, ob die Markierung  $\underline{M}_i$  eines Knotens  $K_i$  eine Teilmenge aus der Markierung  $\underline{M}_j$  eines Knotens  $K_j$  ist.

Gilt  $\underline{M}_i \subseteq \underline{M}_j$ , so sagen wir, daß der Knoten  $(d_i, \underline{M}_i)$  ein Unterknoten des Knotens  $(d_j, \underline{M}_j)$  ist.

### Bauminklusion

Seien  $B$  und  $B'$  zwei strukturell gleiche Bäume mit

$DR(B)=\{(d_i, \underline{M}_i) \mid 1 \leq i \leq |KN|\}$  und  $DR(B')=\{(d_i, \underline{M}'_i) \mid 1 \leq i \leq |KN'|\}$

Wir sagen, daß  $B$  in  $B'$  enthalten ist, falls für alle positionsgleichen Knoten gilt:  $\underline{M} \subseteq \underline{M}'$ .

enthält.

Bemerkung 6.1.2:

Der obige Begriff ist sprachlich nicht zu verwechseln mit Teil- bzw. Unterbaum.

**Baumschnitt**

Seien B und B' zwei strukturell gleiche Bäume mit DR(B) und DR(B') wie oben. Gilt für alle positionsgleiche Knoten  $M_i, M'_i: M_i \bar{\cap} M'_i \neq \{\}$ , so bezeichnen wir  $B \bar{\cap} B'$  als den Baumschnitt von B und B'.

$$B \bar{\cap} B' = \{(d_i, M_i \bar{\cap} M'_i) \mid M_i \bar{\cap} M'_i \neq \{\}\}.$$

**Baumvereinigung**

Analog zum Baumschnitt definieren wir die Baumvereinigung:

$$B \cup B' = \{(d_i, M_i \cup M'_i)\}$$

Folgendes Beispiel illustriert die eingeführten Begriffe.

Beispiel 6.1.2

Betrachten wir folgende zwei Bäume B und B':



oder in der Dewey-Repräsentation

$$DR(B) = \{(1, \{a,b\}), (1.1, \{c,d\}), (1.2, \{e,f\})\}$$

$$DR(B') = \{(1, \{a\}), (1.1, \{c\}), (1.2, \{e,f\})\}$$

Für B und B' gilt:

- B und B' sind strukturell gleich,  $DN(B) = DN(B')$ ;
- B' ist in B enthalten,  $B' \subseteq B$ , oder B enthält B',  $B \supseteq B'$ ;
- der Schnittbaum  $B \bar{\cap} B' = B'$ .

6.2 Regelrepräsentation

Die Dewey-Repräsentation einer Regel  $r_\alpha$  definieren wir als die Vereinigung der Dewey-Repräsentationen der Ein- und Ausgabeschablonen, i.Z.  $DR(r_\alpha) = DR(a_\alpha) \cup DR(e_\alpha)$ . Das gleiche gilt für die Dewey-Notationen:  $DN(r_\alpha) = DN(a_\alpha) \cup DN(e_\alpha)$ .

$\in \{p, \sigma, \pi\}$ , erweitert. Dafür fordern wir, daß die Definitionen gleichzeitig für Ein- und Ausgabeschablonen  $e_\alpha$  bzw.  $a_\alpha$  gültig sind. So ergibt sich z.B. für die Regelidentität der Regeln

$$(DR(r_\alpha) = DR(r'_\alpha)) \iff (DR(a_\alpha) = ((DR(r_\alpha)) \text{ und } (DR(e_\alpha) = DR(e'_\alpha))))$$

Analog werden Regelinklusion ( $\subseteq$ ) und Regelschnitt ( $\cap$ ) definiert. Wir legen fest, daß der Ausgabeschablone (ein Blatt) die Dewey-Zahl 0 zugeordnet wird.

### Beispiel 6.2.1

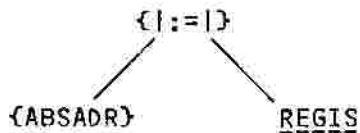
Die P-Code Regel  $r_p(\text{adi})$  aus Beispiel 4.2.2 schreiben wir in Dewey-Repräsentation:  $DR(r_p(\text{adi})) = \{(0, [\text{sp}-1]), (1, \text{adi}), (1.1, [\text{sp}-1]), (1.2, [\text{sp}])\}$ , wobei  $DR([\text{sp}-1]) = \{(0, [\text{sp}-1])\}$  die Dewey-Repräsentation der Ausgabeschablone ist.

### Bemerkung 6.2:

Ab jetzt schreiben wir die Knotenmarkierungen der in Kap.4 und 5 definierten Regeln (P-Code, ARM und KRM) als Mengen aus.

### Beispiel 6.2.2

Anders als in Abb.5.2.2.2 schreiben wir  $r_p(\text{sro } q)$  als:



## 6.3 Erweiterte Substitutionsfunktion; Baumhomomorphismus

In 5.4 haben wir gesehen, wie Datentypen der Abstrakten Register Maschine konkretisiert werden, d.h. wie sie auf der Zielmaschine abgebildet werden. Wie aber werden ARM-Regeln konkretisiert? Der nächste Schritt beschreibt die Abbildung der ARM-Regeln auf KRM-Ebene. Dafür müssen wir die Funktion  $f_\sigma$  auf die ARM-Regeln erweitern.

Die Erweiterung der Substitutionsfunktion  $f_\sigma$  auf ARM-Regeln nennen wir **erweiterte Substitutionsfunktion**, i.Z.  $\tilde{f}_\sigma$ .

Die Anwendung der  $\tilde{f}_\sigma$ -Funktion auf einer ARM-Regel  $r_\sigma(\text{op})$  definieren wir als die Anwendung von  $\tilde{f}_\sigma$  auf die Aus- und Eingabeschablone, i.Z.:  $\tilde{f}_\sigma(r_\sigma(\text{op})) = \tilde{f}_\sigma(a_\sigma(\text{op})) \leftarrow \tilde{f}_\sigma(e_\sigma(\text{op}))$ ,  $\text{op} \in \text{EK}_p$ .

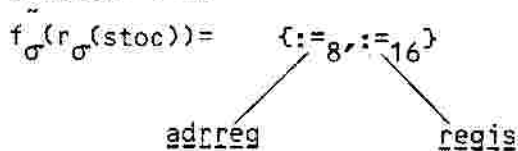
Sei  $DR(e_\sigma(\text{op})) = \{(d_j, M_j) \mid 1 \leq j \leq \text{IKN}((e_\sigma(\text{op})))\}$  die Dewey-Repräsentation der Eingabeschablone ( $e_\sigma(\text{op})$ :  $\text{IKN}((e_\sigma(\text{op})))$  ist Knotenanzahl von  $e_\sigma(\text{op})$ ).

Die Anwendung der erweiterten Substitutionsfunktion  $f_{\sigma}^{\sim}$  auf einer Eingabeschablone (Baum) definieren wir als die Anwendung der primitiven Substitutionsfunktion  $f_{\sigma}$  an jeder Knotenmarkierung des Baums, i.Z.  $f_{\sigma}^{\sim}(DR(e_{\sigma}(op))) = \{(d_j, f_{\sigma}(M_j)) \mid 1 \leq j \leq |KN((e_{\sigma}(op)))|\}$ . Die Baumstruktur wird dadurch nicht geändert, d.h. die Dewey-Notationen bleiben identisch:  $DN(e_{\sigma}(op)) = DN(f_{\sigma}^{\sim}(e_{\sigma}(op)))$ .

Folgendes Beispiel zeigt die Anwendung der erweiterten Substitutionsfunktion  $f_{\sigma}^{\sim}$  auf der ARM-Regel  $r_{\sigma}(stoc)$ .

### Beispiel 6.3

Für die Baumdarstellung von  $(DR(r_{\sigma}(stoc)))$  (siehe Abb.5.2.2.2) erhalten wir:



$r_{\sigma}(stoc) = e_{\sigma}(stoc)$ , da keine Ausgabeschablone in diesem Fall.

Der Homomorphismus führt die Umbenennung aller Knotenmarkierungen aus. Der Übersicht halber werden wir die Eingabeschablone, wenn möglich, als Bäume statt in der Dewey-Darstellung (Mengen von Knoten) wiedergeben.

## 6.4 KRM-Metaregeln; Expansion eines P-Code Befehls

Durch die Anwendung der erweiterten Substitutionsfunktion  $f_{\sigma}^{\sim}$  sind wir von der ARM- auf die KRM-Ebene gelangt.

### Definition 6.4 (KRM-Metaregel)

Das Ergebnis der Funktionsanwendung von  $f_{\sigma}^{\sim}$  auf einer ARM-Regel  $r_{\sigma}(op) \in R_{\sigma}$ ,  $op \in EK_p$ , nennen wir **KRM-Metaregel**, i.Z.  $f_{\sigma}^{\sim}(r_{\sigma}(op)) = mr_{\pi}(op)$ .

Die Knotenmarkierungen  $f_{\sigma}(M)$  einer Metaregel sind Elemente aus der Potenzmenge  $2^{Z_{\pi}}$  des KRM-Zeichensatzes,  $f_{\sigma}(M) \in 2^{Z_{\pi}}$  (vgl. auch Abb.5.4).

Mit  $me_{\pi}(op)$  und  $ma_{\pi}(op)$  bezeichnen wir die Ein- bzw. die Ausgabeschablonen einer KRM-Metaregel  $mr_{\pi}(op)$ .

Die Menge der KRM-Metaregeln bezeichnen wir mit  $MR_{\pi}$  mit

$$MR_{\pi} = \{mr_{\pi}(op) \mid op \in EK_p\}$$

Mit Hilfe der erweiterten Substitutionsfunktion  $f_{\sigma}^{\sim}$  haben wir jeder ARM-Regel (und damit indirekt jedem P-Code Befehl) eine KRM-Metaregel asso-

ziert.

Die Konstruktion der KRM-Metaregel bedeutet das kontextfreie Einbetten der Zielmaschinen-Resourcen in die Semantik der P-Code Befehle (ARM-Regeln). Folgendes Bild zeigt schematisch die Entstehung einer KRM-Metaregel aus einem P-Code Befehl:

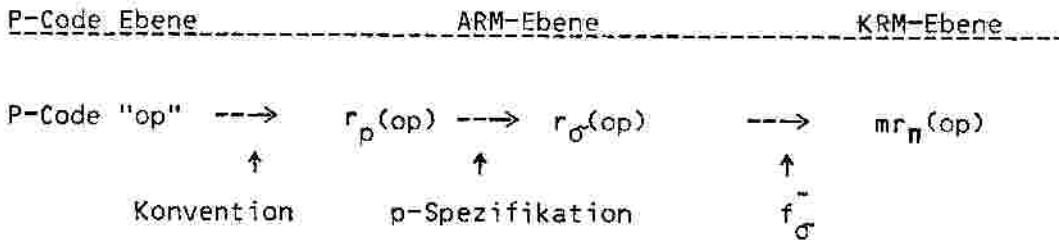
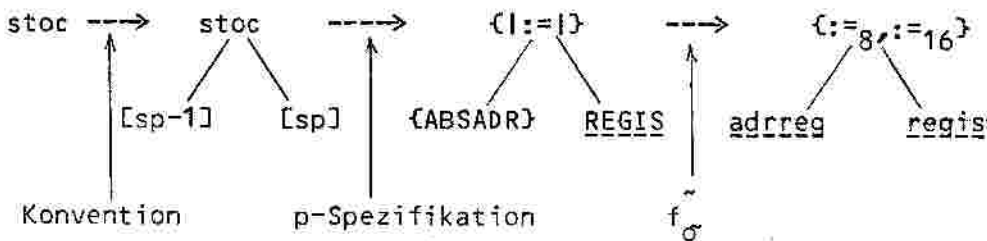


Abb.6.4: Expansion eines P-Code Befehls

Die Entstehung einer KRM-Metaregel aus einer ARM-Metaregel nennen wir **Expansion eines P-Code Befehls**. Die Expansion des P-Code "stoc" Befehls ergibt:

Beispiel 6.4



6.5 Codeauswahl für die KRM-Metaregeln

Um festzustellen wie eine ARM-Regel  $r_\sigma(op)$  auf KRM-Ebene realisiert wird, müssen wir herausfinden, welche KRM-Regeln und in welcher Reihenfolge in der KRM-Metaregel  $mr_\pi(op)$  enthalten sind (also ein Codeselektionsproblem). Ähnlich wie zur Compilierungszeit betrachten wir diese Aufgabe als erweitertes "pattern matching":

- Der erweiterte "pattern matcher" ist im "G-Executor" des Systemgenerators enthalten (Abb.II.2.2);
- Die Eingabe besteht aus den KRM-Metaregeln  $mr_\pi(op)$ ;
- Die "patterns" sind KRM-Regeln  $r_\pi(ins)$ .

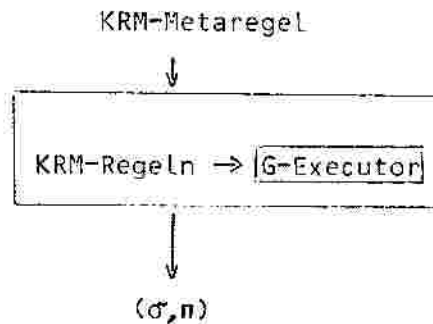


Abb. 6.5 Codeauswahl für KRM-Metaregel

Der Prozess verläuft ähnlich wie in dem Schema aus Abb. 3.3.1. Der "G-Executor" durchsucht die KRM-Metaregeln  $mr_{\pi}(op)$ ,  $op \in EK_p$ , hintereinander nach KRM-Regeln  $r_{\pi}(ins)$ ,  $ins \in Z_{mc}$ . Trifft eine KRM-Regel bzw. Regel-folge, so schreibt der Generator das Paar, bestehend aus den ARM- und KRM-Regeln zusammen als ein Substitutionstupel  $(\sigma, \pi) \in F_{\sigma}$ , wobei  $F_{\sigma}$  die Menge der Substitutionstupel bezeichnet. Die erste Komponente  $\sigma$  dieses Zwei-Tupels ist die ARM-Regel  $r_{\sigma}(op)$ , aus der die Metaregel  $mr_{\pi}(op)$  "entwickelt" wurde (siehe Abb. 6.4). Die zweite Komponente ist die KRM-Regel(folge), die vom "G-Executor" beim Treffen bestimmt wird. In folgenden erklären wir, wie ein "Treffer" bestimmt wird.

Definition 6.5 (Treffer auf KRM-Ebene)

Wir sagen, ein "pattern"  $r_{\pi}(ins) \in R_{\pi}$ ,  $ins \in Z_{mc}$ , trifft, wenn

- 1)  $r_{\pi}(ins) \cap \bigcap mr_{\pi}(op) \neq \emptyset$ ,  $op \in EK_p$ , oder
- 2)  $e_{\pi}(ins) \cap \bigcap tme_{\pi}(op) \neq \emptyset$ ,

wobei  $tme_{\pi}(op)$  Teilbaum der Metaeingabeschablone  $me_{\pi}(op)$ .

Im ersten Fall erhalten wir ein Substitutionstupel vom Typ "1-zu-1", in zweiten dagegen vom Typ "1-zu-n". Folgende Abschnitte 6.6 und 6.7 behandeln beide Fälle.

6.6 Konstruktion der "1-zu-1" Substitutionstupel

Gegeben: eine Metaregel  $mr_{\pi}(op)$ ,  $1 \leq i \leq |EK_p|$ .

Gesucht: Treffermenge für die gegebene Metaregel, d.h. KRM-Regeln

$r_{\pi}(ins)$ ,  $ins \in Z_{mc}$ , s.d.  $r_{\pi}(ins) \cap \bigcap mr_{\pi}(op) \neq \emptyset$ .

ein, indem wir:

- a) zuerst nach zur Metaregel strukturell gleichen Regeln suchen
- b) aus deren Menge die KRM-Regeln aussortieren, die der Trefferbedingung genügen.

zu\_a) Der "G-Executor" sucht nach KRM-Regeln, die strukturell gleich zur KRM-Metaregel  $mr_{\pi}(op)$  sind. Die Menge dieser KRM-Regeln notieren wir mit  $S_{\pi}(op)$ .

$S_{\pi}(op) = \{r_{\pi}(ins\ k) \mid DN(r_{\pi}(ins\ k)) = DN(mr_{\pi}(op))\}$ ,  
mit  $1 \leq k \leq |S_{\pi}(op)|$ .

zu\_b) Welche Elemente aus  $S_{\pi}(op)$  erfüllen die Trefferbedingung?

Diese Untermenge von  $S_{\pi}(op)$ , deren Elemente über die oben erwähnte Eigenschaft verfügen, bezeichnen wir mit  $I_{\pi}(op)$ .

$I_{\pi}(op) = \{r_{\pi}(ins\ l) \mid r_{\pi}(ins\ l) \bar{\cap} mr_{\pi}(op) \neq \{\}\}$ ,  
mit  $1 \leq l \leq |I_{\pi}(op)|$ .

Folgendes Beispiel zeigt wie  $S_{\pi}(op)$  und  $I_{\pi}(op)$  bestimmt werden.

#### Beispiel 6.6.1

Die Metaregel entsprechend dem "lao q" P-Code Operator ergibt auf dem SAB 8080:

$mr_{\pi}(lao\ q) = f_{\sigma}(r_{\sigma}(lao\ q)) = f_{\sigma}(\{(0, \{f_{\sigma}(\{ADRREG\})\}), (1, \{f_{\sigma}(\{ABSADR\})\})\}) =$   
 $addrreg \leftarrow \{a16\}$ , oder falls  $addrreg$  explizit ausgeschrieben:  
 $\{HL, BC, DE, SP\} \leftarrow \{a16\}$ . Lediglich Regel  $r_{\pi}(lxi\ addrreg, a16)$  ist strukturell gleich zur Metaregel  $mr_{\pi}(lao\ q)$ .

$S_{\pi}(lao\ q) = \{r_{\pi}(lxi\ addrreg, a16)\}$ .

$I_{\pi}(lao\ q) = S_{\pi}(lao\ q) = \{r_{\pi}(lxi\ addrreg, a16)\}$ . Wir erhalten somit das Substitutionstupel  $(r_{\sigma}(lao\ q), r_{\pi}(lxi\ addrreg, a16))$ .

Falls  $|I_{\pi}(op)| > 1$ , dann besteht die  $\pi$ -Komponente des Substitutionstupels  $(\sigma, \pi)$  aus mehreren Alternativen. Ergeben sich z.B. für die ARM-Regel  $r_{\sigma}(op)$ ,  $op \in EK$ ,  $k$  Alternativen,  $r_{\pi}(ins\ 1), \dots, r_{\pi}(ins\ k)$ , so schreiben wir das entsprechende Substitutionstupel  $(\sigma; \pi_1 / \dots / \pi_k)$ .

#### Beispiel 6.6.2

Die Metaregel entsprechend dem "ldoc q"-P-Code Operator ergibt auf dem SAB8080:

$$mr_{\pi}(ldoc\ q) = f_{\sigma}^{-1}(r_{\sigma}(ldoc\ q)) = \text{regis} \leftarrow \left\{ \uparrow_8, \uparrow_{16} \right\}$$

|  
{a16}

$S_{\pi}(ldoc\ q) = \{r_{\pi}(lda\ a16), r_{\pi}(lhld\ a16)\}$  (vgl. Abb. 5.3) und

$I_{\pi}(ldoc\ q) = S_{\pi}(ldoc\ q)$ .

Somit erhalten wir das Transformationstupel  $(ldoc\ q; lda\ a16/lhld\ a16)$ .

In Kap. 8 zeigen wir, wie und nach welchen Kriterien die Alternativen zur Compilierungszeit ausgesucht werden.

### 6.7 Konstruktion der Transformationstupel; explizite Darstellung

Sind die Substitutions- bzw. die Abstraktionstupel (Kap. 7) festgelegt, so ist die Konstruktion der Transformationstupel  $(p; mc) \in \Theta$ , wobei  $\Theta$  die Menge der Transformationstupel bezeichnet, nur noch eine Formalität. Die  $\sigma$ - und  $\pi$ -Komponente der Substitutions- bzw. Abstraktionstupel werden durch die assoziierten P-Code Operatoren und Maschineninstruktionen aus der p- und mc-Spezifikationstupel ersetzt.

#### Beispiel 6.7

Aus dem Substitutionstupel von Beispiel 6.6.1 erhalten wir (vgl. auch Abb. 5.2.2.1 und 5.3) das  $(lao\ q; lxi\ \underline{addrreg}\ a16)$ -Transformationstupel.

Was Entscheidungen auf Übersetzerzebene (Auswahl der richtigen Alternative, Bestimmung der Register-Register Transferbefehle) betrifft, so brauchen wir als zusätzliche Information zu den Transformationstupeln die Korrespondenz zwischen den Ressourcen der p- und Zielmaschine. Aus diesem Grund haben wir uns für eine ausführliche Darstellung der Transformationstupel entschlossen.

Wir schreiben die P-Code Operatoren als P-Code Regeln, geben an wie die "stack computer"-Ressourcen durch die Zielmaschinen-Ressourcen konkretisiert werden und welche Kosten (Bytes und Taktzyklen) mit den Maschineninstruktionen verbunden sind.

Diese Darstellung der Transformationstupel nennen wir **explizite Darstellung** der Transformationstupel. Für das obige Transformationstupel erhalten wir:



| P-Code Regel                | Maschineninstruktion          | Resourcen-Äquivalenz                         | Bytes | Taktzyklen |
|-----------------------------|-------------------------------|--|-------|------------|
| $[sp+1] \leftarrow [Lda] q$ | <code>lxi addrreg, a16</code> | $[sp+1] \sim \text{addrreg}$<br>$q \sim a16$ | 3     | 10         |

Abb. 6.7: Transformationstupel; Explizite Darstellung

Die dritte Spalte der Abb. 6.7 ist wie folgt zu interpretieren: der Inhalt der obersten Stackadresse "sp+1" wird durch ein Element der Menge  $\text{addrreg} = \{BC, DE, HL, SP\}$  konkretisiert. Das Symbol "~" steht für "äquivalent zu" und ist als eine Abkürzung für  $f_{\sigma}(\{ADRREG\}) = \text{addrreg}$  zu sehen, wobei  $\{ADRREG\}$  durch die p-Spezifikation (5.2.2.1) dem "[Lda+1]" P-Code Resource zugeordnet wurde.

### 6.8 Konstruktion der "1-zu-n" Substitutionstupel; Automatische Makroexpansion eines P-Code Befehls

Sind die Regelmengen  $S_{\pi}(op)$  oder  $I_{\pi}(op) = \{\}$ ,  $op \in EK_p$ , so können wir kein "1-zu-1" Treffer erhalten. In diesem Fall durchsucht der "pattern matcher" des "G-Executors" die Metaeingabeschablone  $me_{\pi}(op)$  nach Teilbäumen, die von KRM-Regeln  $r_{\pi}(ins)$ ,  $ins \in Z_{mc}$ , getroffen werden. Jeder Treffer ist mit folgenden semantischen Aktionen verbunden: (vgl. auch Algorithmus aus 3.3.1)

- ersetze in der KRM-Metaregel die Eingabe- durch die Ausgabeschablone;
- schreibe die KRM-"Trefferregel"  $r_{\pi}(ins)$  aus;
- rufe Registerverteilungs-Routine auf, falls fällig (siehe Kap. 8).

Folgendes Beispiel illustriert die Realisierung der relativen Adressierung auf dem SAB 8080.

#### Beispiel 6.8

Die P-Code Regel  $r_p(Lda\ p\ q)$ , für  $p=0$ , wird wie folgt auf KRM-Ebene expandiert: (siehe auch Realisierung der "base"-Funktion in Kap. 9)

$$mr_{\pi}(Lda\ 0\ q) = f_{\sigma}(r_{\sigma}(Lda\ 0\ q)) = \text{addrreg} \leftarrow \{+16\}$$

$\swarrow$   
**breg**

$\searrow$   
 $\{a16(q)\}$

Dabei haben wir als Basisregister eines der Registerpaare  $\in \{BC, DE\}$  be-

abhängige 16-Bit Konstante (siehe auch Diskussion über Prozeduraufrufe in Kap.9). Die Abarbeitung der KRM-Metaregel  $mr_{II}(lda\ 0\ q)$  durch den "G-Executor" ergibt:

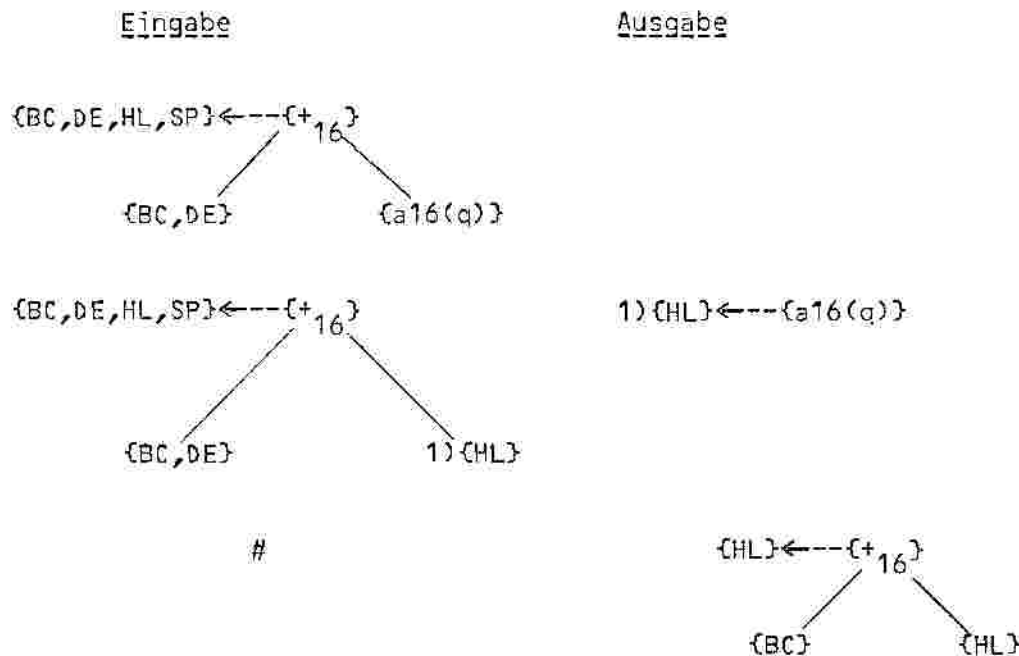


Abb.6.8: Automatische Makroexpansion von  $r_p(lda\ 0\ q)$

Das Ende der Eingabe haben wir mit dem Symbol "#" markiert.

In diesem Beispiel haben wir angenommen, der "G-Executor" sei in der Lage zu erkennen, daß der Inhalt des Registerpaares BC (Basisregister) nicht überschrieben werden darf. Das entsprechende Transformationstupel ( $lda\ 0\ q; lxi\ HL, a16(q).\ dad\ BC$ ) ("." trennt zwei aufeinanderfolgende Instruktionen, die zur selber Alternative gehören) sieht in expliziter Darstellung folgendermaßen aus:

| P-Code Regel                      | Maschineninstruktion | Ressourcen-Äquivalenz | Bytes | Taktzyklen |
|-----------------------------------|----------------------|-----------------------|-------|------------|
| $[sp+1] \leftarrow lda\ (B(0)+q)$ | $lxi\ HL, a16(q)$    | $[sp+1] \sim HL$      | 3     | 10         |
|                                   | $dad\ BC$            | $q \sim a16(q)$       | 1     | 10         |

Abb.6.8: Automatische Makroexpansion; explizite Darstellung

Die in 6.6 und 6.7 besprochenen Fälle entsprechen der Makroexpansion eines P-Code Operators auf Maschinen-Ebene. Die Konstruktion der "1-zu-n" Transformationstupel,  $1 \leq n$ , nennen wir **automatische Makroexpansion eines p-Code Befehls**, weil der Prozess vom Generator ausgeführt wird.

## Kap.7 Konstruktion der Abstraktionstupel

### 7.1 Einführung

In vorigen Kapitel haben wir gesehen, wie ein P-Code Operator auf eine Maschineninstruktion ("1-zu-1" Tupel) oder eine Folge von Maschineninstruktionen ("1-zu-n" Tupel) abgebildet wird. Wie wir in 7.7 sehen werden, ist dieser Schritt wichtig um zu sichern, daß der Codeerzeuger nicht blockiert. Was die Effizienz des Codeerzeugers betrifft, bleiben wir dabei auf dem Niveau der Makroexpansion.

Wünschenswert sind Transformationstupel vom Typ "m-zu-1" die besagen, welche Folge von m P-Code Operatoren auf eine Maschineninstruktion abgebildet wird (siehe "maximal munch"-Heuristik in 3.4.2). Dafür müssen wir untersuchen, welche P-Code Befehle einer Maschineninstruktion assoziiert werden.

Die Generierung der "m-zu-1 patterns" erfolgt ähnlich wie die Generierung der "1-zu-n patterns". Diesmal gehen wir aber in umgekehrter Richtung vor, d.h. von der KRM- zur ARM-Ebene (siehe  $\pi$ -Richtung auf Abb.II.2.1). Es ist, als ob Zwischensprache und Maschinencode die Rollen vertauscht hätten, was die Position innerhalb des back-ends betrifft.

Analog zu Definition von  $f_{\sigma}$ , führen wir in 7.2 die **primitive Abstraktionsfunktion**  $f_{\pi}$  ein. Die Erweiterung von  $f_{\pi}$  zu  $\tilde{f}_{\pi}$  wird in 7.3 dargestellt. Abschnitt 7.4 beschreibt, wie Metaregeln auf ARM-Ebene konstruiert werden und Abschnitt 7.5 wie Codeauswahl für diese Konstrukte erfolgt. In 7.8 wird intuitiv erklärt, welche "Philosophie" hinter dem Begriff "Treffer" steckt und wie dies mit der Verhinderung der Blockierungsmöglichkeit des Codeerzeugers aus Abschnitt 7.7 zusammenhängt. Eine Zusammenfassung des Algorithmus für die Konstruktion der Hilfstupel ist in 7.9 enthalten.

### 7.2 Primitive Abstraktionsfunktion

Mit der Angabe der primitiven Substitutionsfunktion  $f_{\sigma}$  wird dem Generator bekannt wie ARM-Datentypen auf der Zielmaschine konkretisiert

der Abstrakten Register Maschine die Datentypen der Zielmaschine abstimmen. Dieser Vorgang wird durch die Definition der Funktion  $f_{\Pi}$  festgelegt.

Auf einer einelementigen Menge  $\{x\}$  definieren wir  $f_{\Pi}$  wie folgt:

$$f_{\Pi}(\{x\}) = \{y_j \mid \{x\} \in f_{\sigma}(\{y_j\})\}, \text{ mit } 1 \leq j \leq |Z_{\sigma}|.$$

Auf einer k-elementigen Menge  $\{x_1, \dots, x_k\}$  definieren wir  $f_{\Pi}$ :

$$f_{\Pi}(\{x_1, \dots, x_k\}) = f_{\Pi}(\{x_1\}) \sqcup \dots \sqcup f_{\Pi}(\{x_k\}),$$

$$f_{\Pi} : Z^{\Pi} \rightarrow Z^{\sigma}.$$

### Beispiel 7.2

Die Abstraktion des Registerpaares HL und der 8-Bit Konstante k8 des SAB 8080-Prozessors ergeben (betrachte Abb.5.4):

$$f_{\Pi}(\{HL\}) = \{ADRREG\}, \text{ da } \{HL\} \subset \{BC, DE, HL, SP\} = \text{adr\_reg} = f_{\sigma}(\{ADRREG\});$$

$$f_{\Pi}(\{k8\}) = \{KONS\}, \text{ da } \{k8\} \subset \{k8, a16\} = f_{\sigma}(\{KONS\}).$$

### 7.3 Erweiterte Abstraktionfunktion; Baumhomomorphismus

Wir benutzen dieselben Notationen wie in 6.1 und 6.2. Die Erweiterung der  $f_{\Pi}$  auf KRM-Regeln nennen wir **erweiterte Abstraktionsfunktion**, i.Z.  $f_{\Pi}^{\sim}$ .

Die Definitionen dieses Abschnitts erfolgen analog zu den Definitionen aus 6.3. Die Anwendung der erweiterten Abstraktionfunktion  $f_{\Pi}^{\sim}$  auf einer KRM-Regel  $r_{\Pi}(ins)$ ,  $ins \in Z_{mc}$ , heißt, die Anwendung von  $f_{\Pi}^{\sim}$  auf die Aus- und Eingabeschablone, i.Z.:

$$f_{\Pi}^{\sim}(r_{\Pi}(ins)) = f_{\Pi}^{\sim}(a_{\Pi}(ins)) \leftarrow f_{\Pi}^{\sim}(e_{\Pi}(ins)).$$

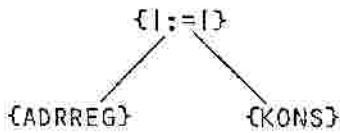
Die Anwendung von  $f_{\Pi}^{\sim}$  auf einer Eingabeschablone  $e_{\Pi}(ins)$  definieren wir als die Anwendung der primitiven Abstraktionfunktion  $f_{\Pi}$  an jeder Knotenmarkierung der Eingabeschablone:

$$f_{\Pi}^{\sim}(DR(e_{\Pi}(ins))) = \{(d_l, f_{\Pi}(\{M_l\}) \mid 1 \leq l \leq |KN(e_{\Pi}(ins))|\} \text{ mit } |KN(e_{\Pi}(ins))| \text{ Knotenanzahl von } e_{\Pi}(ins).$$

Die Anwendung von  $f_{\Pi}^{\sim}$  auf der Ausgabeschablone ergibt sich als ein Sonderfall der Anwendung von  $f_{\Pi}^{\sim}$  auf der Eingabeschablone (Baum bestehend aus einem Blatt),

### Beispiel 7.3

Die Anwendung von  $f_{\Pi}^{\sim}$  auf der KRM-Regel  $r_{\Pi}(mvi(HL), k8)$  ergibt:



#### 7.4 ARM-Metaregel; Expansion einer Maschineninstruktion

Das Ergebnis der Anwendung der erweiterten Abstraktionsfunktion auf einer KRM-Regel  $r_{\Pi}(ins)$  nennen wir **ARM-Metaregel**, i.Z.

$$f_{\Pi}^{\sim}(r_{\Pi}(ins)) = mr_{\sigma}(ins).$$

$MR_{\sigma} = \{mr_{\sigma}(ins) \mid ins \in Z_{mc}\}$ , Menge der ARM-Metaregel und

$$mr_{\sigma}(ins) := ma_{\sigma}(ins) \leftarrow me_{\sigma}(ins).$$

Jeder KRM-Regel  $r_{\Pi}(ins)$  und damit indirekt jeder Zielmaschineninstruktion wird eine ARM-Metaregel  $mr_{\sigma}(ins)$  assoziiert durch Anwendung der erweiterten Abstraktionsfunktion  $f_{\Pi}^{\sim}$  auf  $r_{\Pi}(ins)$ .

Die zu Maschinenressourcen gebundenen P-Code Ressourcen werden kontextfrei in der Semantik der Maschineninstruktion eingesetzt.

Die "Entwicklung" einer Maschineninstruktion zu einer ARM-Metaregel nennen wir **Expansion der Maschineninstruktion**.

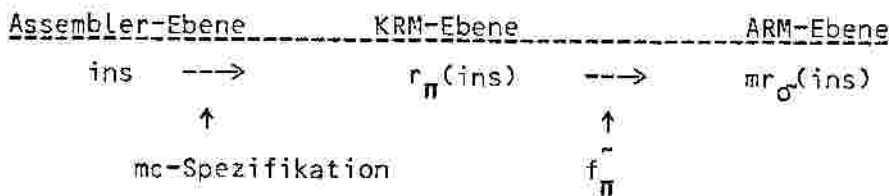


Abb.7.4: Expansion einer Maschineninstruktion

#### 7.5 Codeauswahl für die ARM-Metaregeln

Um festzustellen welche Folge von ARM-Regeln einer Maschineninstruktion zugeordnet werden, müssen wir, nach der Konstruktion der ARM-Metaregel, bestimmen welche ARM-Regeln und in welcher Reihenfolge in der ARM-Metaregel enthalten sind.

Analog zu 6.5 betrachten wir diese Aufgabe als "erweitertes pattern matching".

Der dafür notwendige "pattern matcher" ist im Systemgenerator enthalten. Die Eingabe des "pattern matchers" besteht diesmal aus ARM-Metaregeln und die "patterns" aus ARM-Regeln.

Definition 7.5 (Treffer auf ARM-Ebene)

Ein "pattern"  $r_{\sigma}(op)$ ,  $op \in EK_{\sigma}$ , trifft, falls,

- 1)  $r_{\sigma}(op) \bar{\cap} mr_{\sigma}(ins) \neq \{\}$ ,  $ins \in Z_{mc}$ , oder
- 2)  $e_{\sigma}(op) \bar{\cap} tme_{\sigma}(ins) \neq \{\}$ , mit  $tme_{\sigma}(ins)$  Teilbaum der Metaeingabeschablone  $me_{\sigma}(ins)$ .

Nach demselben Muster wie in 6.5 erhalten wir im Fall 1) Abstraktionstapel vom Typ "1-zu-1", und vom Typ "m-zu-1" im Fall 2), falls für die Abdeckung der ARM-Metaregel mehrere Treffer benötigt werden. Mit jedem Treffer sind folgende semantischen Aktionen verbunden:

- Ersetze Ein- durch Ausgabeschablone;
- Schreibe ARM-"Trefferregel" aus beim vollständigen "matching".

Bemerkung 7.5:

In diesem Fall brauchen wir keine Registerverteilung vorzunehmen.

Analog zu 6.6 wird die Suche nach "passenden" ARM-Regeln folgendermaßen eingeschränkt:

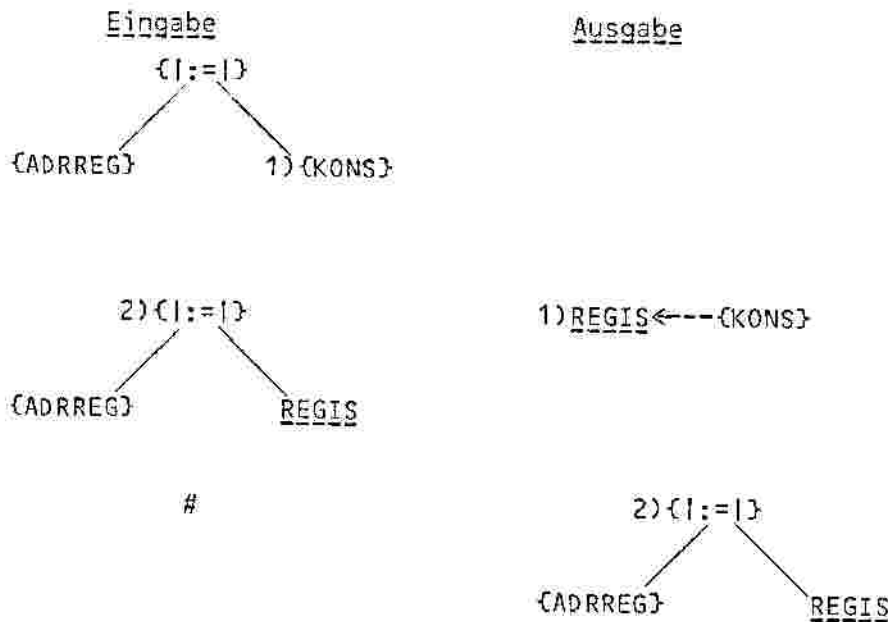
Mit  $S_{\sigma}(ins) = \{r_{\sigma}(op \ k) \mid DN(r_{\sigma}(op \ k)) = DN(mr_{\sigma}(ins))\}$ ,  $1 \leq k \leq |S_{\sigma}(ins)|$ , bezeichnen wir die ARM-Regeln, die zu einer gegebenen ARM-Metaregel strukturell gleich sind.

$I_{\sigma}(ins)$  bezeichnet die Menge, bestehend aus der ARM-Regeln, die an der Bildung der Schnittregel mit  $mr_{\sigma}(ins)$  beteiligt sind.

$I_{\sigma}(ins) = \{r_{\sigma}(op \ l) \mid r_{\sigma}(op \ l) \bar{\cap} mr_{\sigma}(ins) \neq \{\}\}$ , mit  $1 \leq l \leq |I_{\sigma}(ins)|$ . Wie in 6.6 gilt:  $I_{\sigma}(ins) \subseteq S_{\sigma}(ins)$ . Folgendes Beispiel zeigt wie die Konstruktion der Abstraktionstapel erfolgt.

Beispiel 7.5

Für die ARM-Metaregel aus Beispiel 7.3  $mr_{\sigma}(mvi \ (HL), k8) = f_{\Pi}(\tilde{r}_{\Pi}(mvi \ (HL), k8))$ , ergibt sich  $S_{\sigma}(mvi \ (HL), k8) = I_{\sigma}(mvi \ (HL), k8) = \{\}$ , also keine Chance auf die Konstruktion eines "1-zu-1"-Tupels. Die Abarbeitung dieser ARM-Metaregel ergibt:



Das zugehörige Transformationstupel  $(p;mc)$  ergibt in expliziter Darstellung:

|  |      |
|--|------|
| <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: left;"> <div style="border: 1px solid black; padding: 2px; margin-bottom: 2px;">stoc</div> <div style="display: flex; justify-content: space-between; width: 100%;"> <span>[sp-1]</span> <span>[ldc] q</span> </div> </div> <div style="text-align: center;">                 mvi (HL),k8             </div> <div style="text-align: right;"> <div style="border: 1px solid black; padding: 2px; margin-bottom: 2px;">[sp-1]~HL</div> <div style="display: flex; justify-content: space-between; width: 100%;"> <span>2</span> <span>10</span> </div> </div> </div> | q~k8 |
|--|------|

Abb.7.5: Transformationstupel für die "mvi (HL),k8"-Instruktion

Voraussetzung für die Konstruktion des obigen Transformationstupels: HL "liegt" auf [sp-1] (näheres darüber in 8.7).

Es kann vorkommen, daß eine ARM-Metaregel  $mr_{\sigma}(ins)$  von mehreren Folgen von ARM-Regeln "getroffen" wird, so daß ein Abstraktionstupel vom Typ  $(\pi; \sigma_1 / \dots / \sigma_k)$ ,  $1 \leq k \leq |EK_p|$ . Dies ist, im Gegensatz zur Situation aus 6.6, nicht mehrdeutig. Jedes Mal, wenn der Codeerzeuger der P-Code Befehlsfolge  $(\sigma_1 / \dots / \sigma_k)$  begegnet, wird diese durch die  $\pi$ -Instruktion ersetzt. Aus diesem Grund reden wir in diesem Zusammenhang nicht von P-Code Alternativen innerhalb der p-Komponente der Transformationstupel  $(p_1 / \dots / p_k; mc)$ .

7.6 Über den Definitionsbereich der primitiven Substitutions- und Abstraktionsfunktion

In den letzten zwei Kapiteln haben wir zusammengesetzte Datentypen

Dementsprechend haben wir  $f_\sigma : 2^{Z_\sigma} \rightarrow 2^{Z_\pi}$  und  $f_\pi : 2^{Z_\pi} \rightarrow 2^{Z_\sigma}$  definiert.

Die zusammengesetzte Datentypen  $C \subset 2^{Z_\sigma}$  und  $2^{Z_\pi}$  nennen wir **zusammengesetzte Datentypen erster Ordnung**.

Vorsicht ist verlangt, wenn wir mit zusammengesetzte Datentypen **zweiter**

**Ordnung**  $C \subset 2^{Z_\alpha}$ ,  $\alpha \in \{\sigma, \pi\}$ , und höher zu tun haben (vgl. auch Definition des operativen Speichers von Giegerich in [Gi 81]). Der Maschinenbeschreiber darf nicht willkürlich die Ressourcen der Zielmaschine zusammengruppieren.

#### Beispiel 7.6:

Auf dem SAB 8080 gibt es eine Reihe von Instruktionen, die von den Registerpaaren BC und DE Gebrauch machen. Gruppieren wir diese beide

Registerpaare zusammen zu  $\underline{\text{indir}} = \{BC, DE\} \subset 2^{Z_\pi}$ , so gilt (vgl. 5.3):

$\underline{\text{addrreg}} = (\underline{\text{indir}} \cup \{HL, SP\}) \subset 2^{Z_\pi}$ .

Bei der Erweiterung der Funktion  $f_\pi$  müssen wir, vorausgesetzt  $f_\pi : 2^{Z_\pi} \rightarrow 2^{Z_\sigma}$ , stets  $\underline{\text{addrreg}}$  als Vereinigung von zwei Mengen,  $\underline{\text{indir}}$  und  $\{HL, SP\}$

betrachten, und nicht etwa als  $\underline{\text{addrreg}} = \{\underline{\text{indir}}, \{HL, SP\}\} \in 2^{Z_\pi}$ . Konkret bedeutet dies, daß wir jede Maschineninstruktion, die  $\underline{\text{addrreg}}$  als Operand enthält in zwei Maschineninstruktionen zerlegen entsprechend der beiden Komponenten  $\underline{\text{indir}}$  und  $\{HL, SP\}$ . Dies würde den Konstruktionsalgorithmus für die Abstraktionstapel verlangsamen. Eine angebrachte Lösung wäre,

den Definitionsbereich von  $f_\pi$  zu  $D \subset 2^{Z_\pi} \cup 2^{Z_\pi}$  zu erweitern. Für den Benutzer ist die Lösung des Problems von der Assemblerform der Zielmaschine und dem Präzisionsgrad der Beschreibungstechnik abhängig. Durch die Erweiterung des Definitionsbereiches der Substitutions- und Abstraktionsfunktion werden die Überlegungen betreffend der Konstruktion der Transformationstapel nicht beeinflußt.



### 7.7 Verhindern der Blockierungsmöglichkeiten des Codeerzeugers

Zur Übersetzungszeit müssen wir dafür sorgen, daß jeder P-Code Eingabebaum durch P-Code "patterns" aus der Transformationstupel abgedeckt wird. Auf die Generierungszeit übertragen bedeutet dies, daß für jeden P-Code Befehl eine äquivalente Folge von Zielmaschineneinstruktionen "entdeckt" wird. Ist dies gewährleistet, d.h. gelingt es dem Codeerzeuger-Generator für jeden P-Code Befehl ein Transformationstupel (p;mc) zu konstruieren, so läuft der Codeerzeuger zur Compilierungszeit keine Gefahr zu blockieren.

Die Konstruktionsmethode für die Substitutionstupel sichert, daß jeder P-Code Befehl in Maschinencode übersetzt wird mittels folgender drei Schritte:

- 1) Die p-Spezifikation wandelt jede P-Code Regel in eine ARM-Regel um.
- 2) Die primitive Substitutionsfunktion ordnet jeder Knotenmarkierung einer ARM-Regel korrespondierende Maschinenressourcen zu.
- 3) Für jede KRM\_Metaregel muß eine Überdeckung mit KRM-Regel gefunden werden.

Der dritte Schritt läßt sich leider nicht vollständig automatisieren. Die KRM-Regeln entsprechen den Maschineninstruktionen, die vom Benutzer als "beschreibbar" angesehen werden. Für die Übersetzung einiger P-Code Befehle ("mpi" für den SAB 8080, "flo",...; siehe auch Kap.9), müssen wir auf Maschineninstruktionen zurückgreifen, die nicht innerhalb der mc-Spezifikation angegeben wurden.

Für die KRM-Metaregeln, die sich nicht automatisieren lassen, muß der Benutzer eine äquivalente Folge von Maschineninstruktionen finden (häufig als Unterprogramme in den Hersteller-Manualen vorhanden; Siehe auch 5.4). Der Generator kann allerdings melden, ob dieser Schritt erforderlich ist. Der Automatisierungsgrad des Verfahrens würde sich mit Sicherheit erhöhen, wenn statt der hier angegebenen mc-Spezifikation eine "raffinierte" Maschinenbeschreibung (z.B basierend auf denotationeller Semantik wie bei Willmertinger in [Willm 84]) benutzt wird, die instande ist Haupt- und Nebeneffekte einer Maschineninstruktion exakt zu beschreiben.

Im Sinne der Idee dieses Abschnitts läßt sich die Philosophie des Begriffs "Treffer" erklären.

### 7.8 Interpretation des Begriffs "Treffer"

In Abschnitt 6.5 haben wir den Begriff "Treffer" für die  $\sigma$ -Richtung (P-Code  $\rightarrow$ ) definiert als:

$$mr_{\pi}(op) \cap r_{\pi}(ins) \neq \emptyset.$$

#### Bemerkung 7.8:

Während dieser Diskussion werden wir uns, der Einfachheit halber, nur auf den Fall der "1-zu-1" Hilfstupel beschränken.

Vom Standpunkt des Abschnittes 7.7 ist es wünschenswert, wenn die KRM-Metaregel in der KRM-Regel enthalten ist, i.Z.  $mr_{\pi}(op) \subseteq r_{\pi}(ins)$ , was der Schnittbedingung

$$(*) \quad mr_{\pi}(op) \cap r_{\pi}(ins) = mr_{\pi}(op) \text{ entspricht.}$$

In diesem Fall wird der P-Code Operator "op" vollständig von der Maschineninstruktion "ins" abgefangen. Dieser Fall kommt für Maschineninstruktionen vor, die eine breite Klasse von Operanden enthalten. Allgemein heißt das, je orthogonaler der Befehlssatz der Zielmaschine, desto grösser ist die Chance, daß die (\*)-Bedingung zutrifft.

Für Maschineninstruktionen mit eingeschränkten Operandenteilen ist dagegen die Chance groß, daß  $mr_{\pi}(op) - r_{\pi}(ins) \neq \emptyset$ , d.h. die KRM-Metaregel nicht vollständig durch KRM-Regeln abgedeckt wird. Damit der Codeerzeuger nicht blockiert, müssen wir dafür sorgen, daß jede, in P-Code auftretende Situation, ein Äquivalent auf Maschinencode-Ebene hat. Dieser Aspekt wird in 8.5 besprochen.

In Beispiel 6.6.1 haben wir, unter Berücksichtigung dieses Gedanken, den P-Code Operator "lao q" vollständig abgedeckt. Wir sagen auch, "lao q" ist **semantisch uneingeschränkt** auf die "lxi addrreg,a16"-Instruktion abgebildet worden.

Maßgebend für den Begriff "Treffer" ist die Absicht P-Code vollständig durch Maschinencode abzudecken.

Der Begriff "Treffer" in  $\pi$ -Richtung (Maschinencode  $\rightarrow$  P-Code) unterstützt denselben Gedanken wie in  $\sigma$ -Richtung: wünschenswert ist der Fall  $r_{\sigma}(op) \subseteq mr_{\sigma}(ins)$ .

### 7.9 Substitutions- und Abstraktionstupel; Zusammenfassung

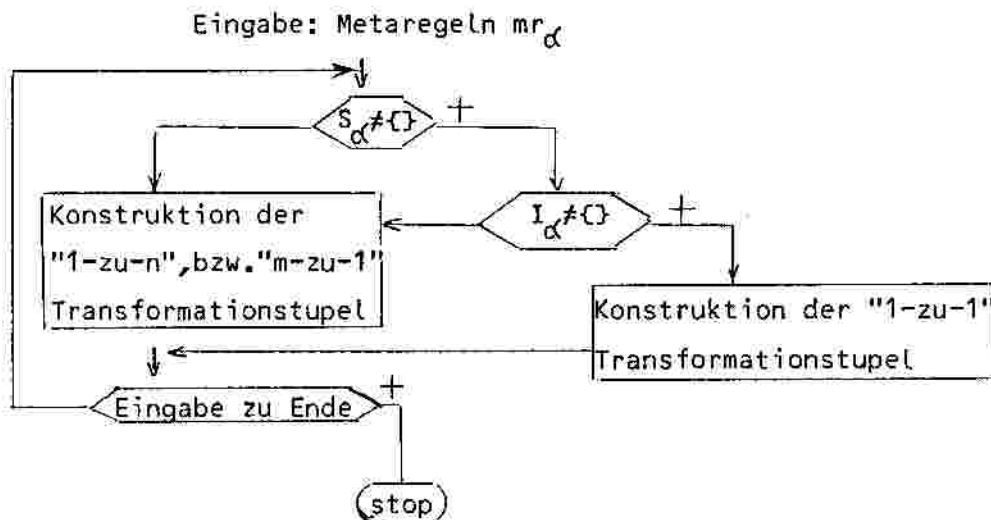
Der Übersicht halber fassen wir die Konstruktion der Substitutions- und Abstraktionstupel schematisch zusammen.

| Hilfstupel      | "G-Executor"-Eingabe                   | "G-Executor"-Ausgabe                        | "Patterns" |
|-----------------|--|---|------------|
| $(\sigma; \pi)$ | KRM-Metaregel<br>$mr_{\pi}(op)$        | Folge von KRM-Regeln<br>$r_{\pi}(ins)$      | KRM-Regeln |
| $(\pi; \sigma)$ | ARM-Metaregel<br>$mr_{\sigma}(ins\ i)$ | Folge von ARM-Regeln<br>$r_{\sigma}(op\ j)$ | ARM-Regeln |

Abb.7.9.1: Hilfstupel; Übersicht

Um zwischen den "pattern matcher" des Generators, die in  $\sigma$ - und  $\pi$ -Richtung eingesetzt werden zu unterscheiden, nennen wir den ersten  $\sigma$ -"pattern matcher" und den zweiten  $\pi$ -"pattern matcher". Der  $\sigma$ -"pattern matcher" arbeitet auf KRM-Ebene mit KRM-Regeln als "patterns". Der  $\pi$ -"pattern matcher" dagegen wird auf ARM-Ebene eingesetzt und benutzt ARM-Regeln als "patterns".

Die Konstruktion der Hilfstupeln  $(\sigma; \pi)$  und  $(\pi; \sigma)$  erfolgt nach demselben Muster, schematisch in Abb.7.9.2 wiedergegeben.



mit  $\alpha \in \{\pi, \sigma\}$

Abb.7.9.2: Konstruktion der Hilfstupel

Selbstverständlich sind die "1-zu-1" Hilfstupel ein Sonderfall der "1-zu-n" bzw. "m-zu-1" Hilfstupel. Die Tatsache, daß wir die "1-zu-1" Hilfstupel separat betrachten, hat einen besonderen Grund. Damit wollen wir redundante Arbeit vermeiden. Angenommen bei der Konstruktion der Substitutionstupel (Abarbeitung der  $\sigma$ -Richtung, siehe Abb.II.2.1) findet der Generator ein Paar  $(r(\sigma) \cdot r(ins))$  vom Typ "1-zu-1". Bei der

Konstruktion der Abstraktionstupel ( $\pi$ -Richtung auf Abb. II.2.1) stellt sich die Frage: "Braucht der Generator die Maschineninstruktion "ins" zu berücksichtigen oder kann er sie vernachlässigen?" Der Generator kann "ins" vernachlässigen, falls die Abarbeitung der Abstraktionstupel kein Paar vom Typ  $(r_{\pi}(\text{ins}); r_{\sigma}(\text{op}'))$ , mit  $\text{op}' \neq \text{op}$ , liefert.

Unter welchen Bedingungen, die oben erwähnte Annahme gilt und wie dieses Problem mit der Komplexität des "G-Executors" zusammenhängt, beschreiben wir im nächsten Kapitel.

## Kap.8 Anforderungen an den G- und C-Executor

In diesem Kapitel stellen wir vor, was man mit den vorhandenen Werkzeugen, G- und C-Executor, erreichen kann, um guten Code zu liefern.

### 8.1 Faktorisierung der Hilfstupel; Äquivalente Hilfstupel

Bisher haben wir noch keine Annahmen gemacht, ob die Konstruktion der Substitutionstupel ( $\sigma$ -Richtung) vor der Konstruktion der Abstraktionstupel ( $\pi$ -Richtung) erfolgen soll oder umgekehrt. Wir haben lediglich angenommen, beide Richtungen werden vollständig abgearbeitet, d.h. für jeden P-Code Operator  $op$ ,  $op \in EK_p$ , wird ein Substitutionstupel  $(\sigma; \pi)$  und für jede Maschineninstruktion  $ins$ ,  $ins \in Z_{mc}$ , ein Abstraktionstupel  $(\pi; \sigma)$  konstruiert.

Für die Übersetzungszeit ist es wünschenswert, wenn Transformationstupel mit gleichen p- und (oder) mc-Komponenten zusammengruppiert werden. Auf diese Weise wird die semantisch zusammenhängende Information aus p- und Maschinencode in einem einzigen "pattern" gehalten. Dies erreichen wir, indem Substitutionstupel mit derselben  $\pi$ -Komponente und Abstraktionstupel mit derselben  $\sigma$ -Komponente **faktoriert** werden.

#### Beispiel 8.1

Die Substitutionstupel  $(\sigma; \pi)$  und  $(\sigma'; \pi)$  werden zu  $(\sigma/\sigma'; \pi)$  faktoriert.

Aus dem Substitutionstupel  $(r_\sigma(op); r_\pi(ins))$  oder aus dem Abstraktionstupel  $(r_\pi(ins); (r_\sigma(op)))$  erhalten wir dasselbe Transformationstupel. Hilfstupel  $(\sigma; \pi)$  und  $(\pi; \sigma)$ , die über diese Eigenschaft verfügen, nennen wir **äquivalent**. Äquivalente Hilfstupel  $(\sigma; \pi)$  und  $(\pi; \sigma)$  unterscheiden sich nur in der Reihenfolge der Aufschreibung ihrer  $\sigma$ - und  $\pi$ -Komponenten. Für das daraus konstruierte Transformationstupel ist es gleichgültig, ob der Generator in  $\sigma$ - oder  $\pi$ -Richtung vorgegangen ist.

8.2 Eliminierung der redundanten "1-zu-1" Transformationstupel

Wir wollen jetzt untersuchen unter welchen Bedingungen die eine oder die andere Richtung bei der Konstruktion der "1-zu-1" Transformationstupel vernachlässigt werden darf.

Satz 8.2

Bei der Konstruktion der Transformationstupeln vom Typ "1-zu-1" braucht nur eine Richtung betrachtet werden.

Wir unterscheiden zwei Fälle entsprechend der zwei Konstruktionsrichtungen:

a) Nach vollständiger Abarbeitung der  $\sigma$ -Richtung

Gegeben: Ein Substitutionstupel  $(\sigma; \pi) = (r_\sigma(op); r_\pi(ins))$ ,

$op \in EK_p$  und  $ins \in Z_{mc}$ .

Zu zeigen: Es gibt kein  $op'$ ,  $op' \in EK_p$ , so daß ein "1-zu-1" Abstraktionstupel  $(r_\pi(ins); r_\sigma(op'))$  bei der vollständigen Abarbeitung der  $\pi$ -Richtung entsteht.

Beweis: Durch Widerspruch

Angenommen es existiert ein  $op'$ ,  $op' \in EK_p$ , so daß  $(r_\pi(ins); r_\sigma(op'))$  ein Abstraktionstupel ist. Dies ist gleichbedeutend mit  $r_\sigma(op') \in I_\sigma(ins)$ .

Nach Lemma 8.2 (nächste Seite) gilt aber:

$$(r_\sigma(op') \in I_\sigma(ins)) \iff (r_\pi(ins) \in I_\pi(op'))$$

Aus der Existenz des Substitutionstupels  $(\sigma; \pi) = (r_\sigma(op); r_\pi(ins))$ , ergibt sich, daß  $r_\pi(ins) \in I_\pi(op)$ .

Aus  $r_\pi(ins) \in I_\pi(op')$   
und  
 $r_\pi(ins) \in I_\pi(op)$

$\implies$  Es existiert ein  $(\sigma; \pi)$ -Tupel vom  
Typ "1-zu-1" mit zwei  $\sigma$ -Komponenten  
 $(r_\sigma(op') / r_\sigma(op); r_\pi(ins))$

d.h. das Substitutionstupel  $(r_\sigma(op'); r_\pi(ins))$  wäre schon bei der Abarbeitung der  $\sigma$ -Richtung entdeckt worden.

Folglich: Es gibt kein  $op'$  mit der angenommenen Eigenschaft.

b) Nach vollständiger Abarbeitung der  $\pi$ -Richtung:

Gegeben: Ein Abstraktionstupel  $(\pi; \sigma) = (r_\pi(ins); r_\sigma(op))$ ,  $op \in EK_p$  und  $ins \in Z_{mc}$ .

Zu zeigen: Es gibt kein  $ins'$ ,  $ins' \in Z_{mc}$ ,  $ins' \neq ins$  so daß ein "1-zu-1" Substitutionstupel  $(r_\sigma(op); r_\pi(ins'))$  bei der Abarbeitung der  $\sigma$ -Richtung entsteht.

Beweis: Analog zu a).

Lemma 8.2

$$(r_{\sigma}(op) \in I_{\sigma}(ins)) \iff (r_{\pi}(ins) \in I_{\pi}(op)),$$

Beweis:

$$I_{\pi}(op) = \{r_{\pi}(ins\ k) \mid r_{\pi}(ins\ k) \mid \bar{\cap} mr_{\pi}(op) \neq \emptyset\}, \quad 1 \leq k \leq |I_{\pi}(op)| \text{ und}$$

$$I_{\sigma}(ins) = \{r_{\sigma}(op\ l) \mid r_{\sigma}(op\ l) \mid \bar{\cap} mr_{\sigma}(ins) \neq \emptyset\}, \quad 1 \leq l \leq |I_{\sigma}(ins)|.$$

Mit anderen Worten:

$$r_{\sigma}(op) \in I_{\sigma}(ins) \iff r_{\sigma}(op) \mid \bar{\cap} mr_{\sigma}(ins) \neq \emptyset \text{ und}$$

$$r_{\pi}(ins) \in I_{\pi}(op) \iff r_{\pi}(ins) \mid \bar{\cap} mr_{\pi}(op) \neq \emptyset.$$

Dadurch wird Lemma 8.2 zu:

$$(r_{\sigma}(op) \mid \bar{\cap} mr_{\sigma}(ins) \neq \emptyset) \iff (r_{\pi}(ins) \mid \bar{\cap} mr_{\pi}(op) \neq \emptyset)$$

Laut Definition der Metaregeln gilt:

$$mr_{\sigma}(ins) = f_{\pi}^{-1}(r_{\pi}(ins)) = f_{\pi}(a_{\pi}(ins)) \leftarrow f_{\pi}^{-1}(e_{\pi}(ins)) \text{ und}$$

$$mr_{\pi}(op) = f_{\sigma}(r_{\sigma}(op)) = f_{\sigma}(a_{\sigma}(op)) \leftarrow f_{\sigma}(e_{\sigma}(op))$$

Unter Berücksichtigung der Definition von Metaregeln und Anwendung des Regelschnitts auf Ein- bzw. Ausgabeschablonen, wird Lemma 8.2 zu:

$$a) \quad (a_{\pi}(ins) \mid \bar{\cap} f_{\sigma}(a_{\sigma}(op)) \neq \emptyset) \iff (a_{\sigma}(op) \mid \bar{\cap} f_{\pi}(a_{\pi}(ins)) \neq \emptyset)$$

und

$$b) \quad (e_{\pi}(ins) \mid \bar{\cap} f_{\sigma}(e_{\sigma}(op)) \neq \emptyset) \iff (e_{\sigma}(op) \mid \bar{\cap} f_{\pi}(e_{\pi}(ins)) \neq \emptyset)$$

Wir beweisen zunächst Fall a).

a) Für die Ausgabeschablonen (einelementige Bäume)

Wir beweisen zunächst die " $\Leftarrow$ "-Richtung:

Der Einfachheit halber führen wir folgende Notationen ein:

$$a_{\pi}(ins) = \underline{x} = \{x_1, \dots, x_m\} \in 2^{\mathbb{Z}_{\pi}} \text{ und } a_{\sigma}(op) = \underline{y} = \{y_1, \dots, y_n\} \in 2^{\mathbb{Z}_{\sigma}}$$

Somit gilt für a) in " $\Leftarrow$ "-Richtung:

$$\underline{x} \mid \bar{\cap} f_{\sigma}(\underline{y}) \neq \emptyset \iff \underline{y} \mid \bar{\cap} f_{\pi}(\underline{x}) \neq \emptyset.$$

Beweis:

$$f_{\pi}(\underline{x}) = f_{\pi}(\{x_1, \dots, x_m\}) = f_{\pi}(\{x_1\}) \cup \dots \cup f_{\pi}(\{x_m\}) \text{ und}$$

$$f_{\pi}(\{x_1\}) = \{y_{1k} \mid x_1 \in f_{\sigma}(\{y_{1k}\})\}$$

$$f_{\pi}(\{x_m\}) = \{y_{mk} \mid x_m \in f_{\sigma}(\{y_{mk}\})\}.$$

Aus  $\underline{y} \mid \bar{\cap} f_{\pi}(\underline{x}) \neq \emptyset$  folgt:

Es existiert ein  $q$ ,  $1 \leq q \leq n$ , und es existiert ein  $p$ ,  $1 \leq p \leq m$ , so daß  $y_q = y_{pk}$ .

Dies bedeutet, daß  $y_q \in f_{\pi}(\{x_p\})$ . Daraus ergibt sich gemäß der Definition von  $f_{\pi}$ :  $x_p \in f_{\sigma}(\{y_q\})$ , was  $\underline{x} \mid \bar{\cap} f_{\sigma}(\underline{y}) \neq \emptyset$  impliziert.

Für die " $\Rightarrow$ "-Richtung erfolgt der Beweis analog.

b) Für die Eingabeschablonen:

In diesem Fall wenden wir den Beweis von a) an jeder Knotenmarkierung der Eingabeschablone an.

### 8.3 Konstruktion von "effizienten patterns"; "Minimal munch"-Heuristik

In Abschnitt 8.2 haben wir gesehen wie redundante Arbeit bei der Konstruktion der Transformationstupel ( $p;mc$ ) vermieden wird. Dabei haben wir vorausgesetzt, daß der "pattern matcher" des G-Executors in beiden Richtungen gemäß derselben Suchmethode arbeitet, in diesem Fall entsprechend der "maximal munch"-Heuristik (3.4.2).

Nächstes Beispiel zeigt, daß die Anwendung dieser Heuristik in  $\pi$ -Richtung zu Effizienzverlusten führen kann.

#### Beispiel 8.3

Angenommen wir arbeiten mit einem Rechner, der über eine Instruktion vom Typ "inr addrreg" (inkrementiere Inhalt eines Adressregisters) verfügt. Bei der Abarbeitung der  $\sigma$ -Richtung erhalten wir das Transformationstupel (inc 1;inr addrreg). Laut Satz 8.2 brauchen wir die "inr addrreg"-Instruktion bei der Abarbeitung der  $\pi$ -Richtung nicht mehr zu betrachten.

Eine geschickte "ad hoc"-Übersetzung von P-Code in Maschinencode zeigt, daß die Maschineninstruktion "inr addrreg" auch die P-Code Folge "ldci 1, adi" abdeckt, vorausgesetzt die Operanden von "adi" sind Adressen. Unter diesen Umständen ergibt sich nach Abarbeitung der  $\sigma$ - und  $\pi$ -Richtungen und Faktorisierung der Hilfstupel folgendes Transformationstupel: (inc 1/ldci 1, adi;inr addrreg). Was die Kosten betrifft, ist die "ldci 1,adi"-Komponente des Transformationstupels ein "besseres pattern" als "inc 1", (unser Ziel ist, möglichst lange Folgen von P-Code Operatoen auf einer Maschineninstruktion abzubilden).

Weiterhin zeigen wir, wie die "ldci 1,adi"-Komponente bei der Abarbeitung der  $\pi$ -Richtung entsteht. Die Eingabe besteht aus der ARM-Metaregel  $mr_{\sigma}(\text{inr } \underline{\text{addrreg}})$ . Die "patterns" sind die ARM-Regeln  $r_{\sigma}(\text{ldci } 1)$  und  $r_{\sigma}(\text{adi})$ .

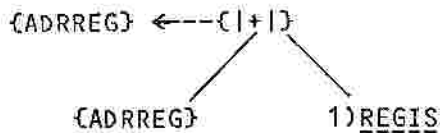
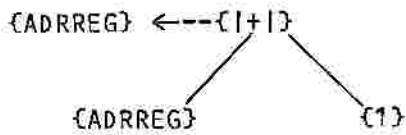
In diesem Fall haben wir den "adi"-Befehl nur auf Berechnung von Adressen eingeschränkt. Im Abschnitt 8.5 werden wir dieses Problem ausführlich behandeln.

Für den ARM-Datentyp {KONS} haben wir direkt {1} eingesetzt, weil diese Konstante schon auf P-Code Ebene bekannt ist.



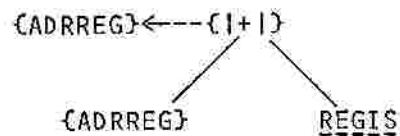
Eingabe: ARM-Metaregel

Ausgabe: ARM-Regeln



#

1) REGIS ←-- {1}



Aus dem obigen Beispiel ist es einfach zu sehen wie die zweite P-Code Folge "ldci 1,adi" entstanden ist: der "shift/reduce"-Konflikt (vgl. 3.4.2) am rechten Sohn vom { | + | }-Knoten wird zugunsten von "reduce" entschieden, also genau umgekehrt wie bei der "maximal munch"-Strategie. Dadurch erreichen wir, daß eine ARM-Metaregel durch viele ARM-Regeln abgedeckt wird.

### Definition 8.3 ("minimal munch"-Heuristik):

Die Suchmethode nach der

- "shift/reduce"-Konflikte zugunsten von "reduce" und
- "reduce/reduce"-Konflikte zugunsten des kürzesten "pattern" entschieden werden, nennen wir "minimal munch"-Heuristik.

Daraus ergibt sich eine differenzierte Anforderung an unserem G-Executor:

- die  $\sigma$ -Richtung soll nach der "maximal munch"-,
  - die  $\pi$ -Richtung nach der "minimal munch"-Heuristik behandelt werden.
- Die beiden Such-Heuristika unterscheiden sich nicht, solange keine "shift/reduce"- und "reduce/reduce"-Konflikte begegnet werden.

Für den Satz 8.2 bedeutet dies: Die Konstruktion der Transformationstupel braucht nur in einer Richtung betrachtet werden, solange dabei keine Entscheidungskonflikte begegnet werden.

Beide Heuristika, "minimal- und maximal munch", spiegeln dasselbe Prinzip wieder: möglichst große Teile der Eingabe mit einer Reduktion abzudecken.

Außer der hier erwähnten Heuristika gibt es Suchmethoden, die mehrere Überdeckungen zulassen. Ideal wäre ein "pattern matcher", der bei der Abarbeitung einer Metaregel sämtliche Überdeckungen findet. Der Aufwand

dafür dürfte vertretbar sein, wenn man bedenkt, daß die Eingabe für ein solches "pattern matcher" aus kleinen Bäumen (entsprechend jeweils einem P-Code Befehl in  $\sigma$ -Richtung oder einer Maschineninstruktion in  $\pi$ -Richtung) besteht.

#### 8.4 Transformationstupel vom Typ "m-zu-n"

Bisher haben wir innerhalb der dargestellten Beispiele in  $\sigma$ -Richtung ein "pattern matching"-Verfahren gemäß der "maximal munch"-Heuristik angewandt. Dadurch erreichen wir, daß ein P-Code Operator durch möglichst wenige Maschineninstruktionen übersetzt wird. Umgekehrt versuchen wir in  $\pi$ -Richtung mit Hilfe der "minimal munch"-Heuristik, möglichst viele P-Code Befehle auf eine Maschineninstruktion abzubilden. Dadurch entstehen Transformationstupel vom Typ "1-zu-1", "1-zu-n" und "m-zu-1", keine aber vom Typ "m-zu-n",  $m, n \neq 1$ , da wir in beiden Richtungen stets von einem P-Code Operator bzw. Maschineninstruktion ausgehen.

Eine Möglichkeit Transformationstupel vom Typ "m-zu-n" zu konstruieren, wenn auch indirekt, d.h. ohne von einer Folge von P-Code Operatoren oder Maschineninstruktionen auszugehen, ergibt sich, indem wir in  $\sigma$ -Richtung neben der "maximal munch"- auch die "minimal munch"-Heuristik zulassen. Folgendes Beispiel zeigt, wie ein solches Tupel entsteht.

#### Beispiel 8.4

Wird der p-Code Operator "inc 1" (siehe Beispiel 8.3) entsprechend der "minimal munch"-Heuristik behandelt ( $\sigma$ -Richtung), so erhalten wir das Transformationstupel

```
(inc 1; load addrreg,1. add addrreg,addrreg)
```

Den "." haben wir verwendet, um zwei aufeinanderfolgende Instruktionen, die zur selben Alternative gehören, zu unterscheiden.

Die zwei Maschineninstruktionen aus dem obigen Transformationstupel haben folgende Semantik:

-lade eine Konstante (1 in diesem Fall) auf Adressregister addrreg;  
Resultat in addrreg.

-Addiere die Inhalte von zwei Adressregister; Resultat in ersten Operand.

Faktorisieren wir das soeben erhaltene Transformationstupel mit den Transformationstupeln (inc 1; inr addrreg) und (ldci 1. adi; inr addrreg) aus Beispiel 8.3 die, aus äquivalenten Hilfstupeln abstammen, so erhalten

wir das etwas aufwendigere Transformationstupel

(inc 1/ldci 1. adi; inr addrreg/load addrreg,1. add addrreg,addrreg).

Dieses Transformationstupel ist die abgekürzte Schreibweise für 4 nicht-faktorierte Transformationstupel vom Typ "1-zu-1, 1-zu-2, 2-zu-1" und "2-zu-2".

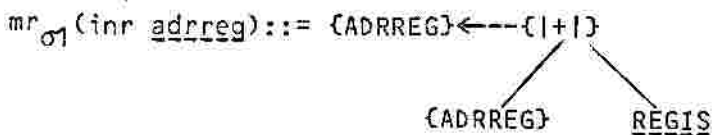
Die "maximal munch"-Methode liefert zwar guten Code in einer lokalen Umgebung, was allerdings die Erzeugung von schlechten Code zu einem späteren Zeitpunkt nicht ausschließt (siehe 3.4.2).

Ein guter Executor simuliert, bevor die erste Transformation (Reduktion plus Ausgabe des Objektcodes in unserem Fall) erfolgt, sämtliche "Treffer" durch. Solche Werkzeuge wurden an dem Lehrstuhl vom Prof. Reinhard Wilhelm von Ulrich Möncke, Beatrix Weisgerber und Reinhard Wilhelm entwickelt. Der Aufwand, zur Generierungszeit mächtige Suchstrategien (bei der Konstruktion der Transformationstupel) anzuwenden, kann sich zur Compilierungszeit bezahlt machen.

### 8.5 Semantische Einschränkungen

Hätten wir im Beispiel 8.3  $r_{\sigma}(adi)$  als "pattern" benutzt, so hätten wir keinen "Treffer" im Sinne der Idee aus 7.7 erzielen können.

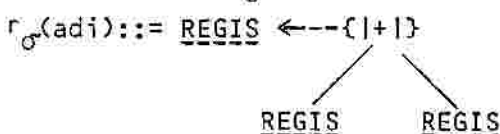
Sei  $mr_{\sigma_1}(inr \underline{addrreg})$  die transformierte ARM-Metaregel,



Notation: Der Index "1" hinter " $\sigma$ " deutet darauf hin, daß eine Reduktion (Transformation) stattgefunden hat: ein Unterbaum (Eingabeschablone) wurde durch einen Knoten (Ausgabeschablone) ersetzt.

Allgemein schreiben wir  $mr_{\sigma_k}(ins)$ ,  $ins \in Z_{mc}$ , wobei  $k$  die Anzahl der bisher angewandten Transformationen bezeichnet.

Die ARM-Regel  $r_{\sigma}(adi)$  bezieht sich auf REGIS:

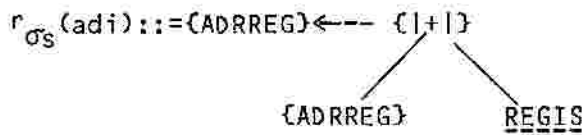


und es gilt:  $r_{\sigma}(adi) \not\subseteq mr_{\sigma_1}(inr \underline{addrreg})$ .

Somit ist  $r_{\sigma}(adi)$  kein "Treffer" im Sinne der Forderung nach vollständiger Abdeckung eines P-Code Operators. Erst nach Einschränkung

der REGIS-Knoten (Ausgabeschablone und Linker Sohn von "{|+|}-Knoten) auf {ADRREG} läßt sich das "pattern" anwenden.

Notation: Das eingeschränkte "pattern" schreiben wir symbolisch durch Angabe eines zusätzlichen Index "s" nach "σ", in diesem Fall:



Die oben erwähnte Einschränkung übertragen auf P-Code Ebene bedeutet, daß [sp-1] eine Adresse enthalten muß.

Dieser Fall entspricht der Situation  $mr_{\sigma}(\text{ins}) \subseteq r_{\sigma}(\text{op})$  bei der Konstruktion der Abstraktionstapel (in  $\sigma$ -Richtung).

Wir reden in diesem Fall von **semantischen Einschränkungen auf P-Code Ebene**.

Gilt dagegen  $r_{\sigma}(\text{op}) \subseteq mr_{\sigma}(\text{ins})$ , so sprechen wir von **semantischen Einschränkungen auf Maschinencode Ebene**.

Allgemein, für  $mr_{\sigma}(\text{ins}) \cap r_{\sigma}(\text{op}) \neq \emptyset$ , haben wir mit semantischen Einschränkungen auf beiden Ebenen zu tun.

Somit entspricht unsere Definition des Begriffs "Treffer" aus 6.5 und 7.5 dem semantisch eingeschränkten Fall auf beiden Ebenen, P-Code und Maschinencode.

Dieselbe Situation tritt bei der Konstruktion der Substitutionstapel in  $\sigma$ -Richtung auf, für  $r_{\Pi}(\text{ins}) \subseteq mr_{\Pi}(\text{op})$  (semantischen Einschränkungen auf P-Code Ebene),  $mr_{\Pi}(\text{op}) \subseteq r_{\Pi}(\text{ins})$  (Einschränkungen auf Maschinencode Ebene), oder allgemein  $r_{\Pi}(\text{ins}) \cap mr_{\Pi}(\text{op}) \neq \emptyset$  (Einschränkungen auf beiden Ebenen).

Berücksichtigen müssen wir nur die Einschränkungen auf P-Code Ebene, um Transformationstapel zu konstruieren, die jeder vorkommenden Situation in P-Code "gewachsen" sind.

Semantische Einschränkungen auf Maschinencode-Ebene bedeuten, daß nicht alle Operanden einer Maschineninstruktion zum Einsatz kommen.

Das Problem der semantischen Einschränkungen ist eine Folge der unterschiedlichen Orthogonalität von P-Code Befehlen und Maschineninstruktionen.

## 8.6 Semantische Betrachtungen bei der Codeauswahl (Übersetzungszeit)

In Beispiel 6.6.2 haben wir gesehen, daß für ein P-Code Operator "op" sich mehrere Alternativen an Maschinencode ergeben können, falls

$|I_{II}| > 1$ . Die explizite Darstellung des Transformationstupels sieht wie folgt aus:

|                      |             |          |                               |   |    |
|----------------------|-------------|----------|-------------------------------|---|----|
| $[esp+1] \leftarrow$ | <u>ldoc</u> | Lda a16  | $[esp+1] \sim A; q \sim a16$  | 3 | 13 |
|                      | l           | Lhld a16 | $[esp+1] \sim HL; q \sim a16$ | 3 | 16 |
|                      | q           |          |                               |   |    |

Abb.8.6 Transformationstupel für den "ldoc" P-Code Operator auf dem SAB 8080

Die Auswahl der richtigen Alternative erfolgt zur Compilierungszeit nach dem Wert der Adresse  $q$ ,  $1 \leq q \leq 2^{16}-1$ , und der Länge von  $[esp+1]$ ,  $l([esp+1])$ . Ist z.B.  $l([esp+1]) \geq 8$  Bit, so scheidet die erste Alternative aus (A bezeichnet den Akkumulator, 8 Bit groß, HL das Registerpaar HL, 16 Bit lang). Gibt es dagegen keine Einschränkungen was  $[esp+1]$  und  $q$  betrifft, so wird die billigste Alternative ausgesucht, in diesem Fall "Lda a16".

### 8.7 Behandlung der Register-Register Transferinstruktionen

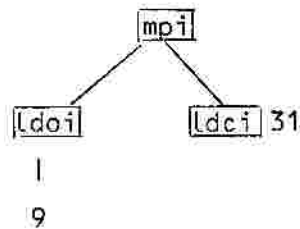
Bis jetzt haben wir Register-Register Transferinstruktionen in keinem Transformationstupel eingesetzt. Das Problem rührt von den unterschiedlichen Architekturen des "stack computers" und der gängigen Register-Stack Maschinen. Die Register-Register Transferinstruktionen haben keinen Äquivalent in P-Code (es gibt in P-Code keinen Transfer zwischen den Inhalten der oberen Stackadressen).

Die Lösung ergibt sich zur Übersetzerzeit, indem vor der Ausgabe einer Maschineninstruktion (mc-Komponente eines Transformationstupels (p;mc)) geprüft wird, ob die Äquivalenz zwischen den Ressourcen der p-Maschine und Zielmaschine erhalten bleibt.

Folgendes Beispiel illustriert, wie eine solche Entscheidung getroffen wird bei der Codeerzeugung für eine PDP-11/70-Maschine (vgl.auch PDP-11/70 Beschreibung von Glanville in [GL 77]).

#### Beispiel 8.7

Angenommen der Codeerzeuger hat folgenden Teileingabebaum zu bearbeiten:



Folgende Transformationstupel werden dazu gebraucht

|                 |                 |          |
|-----------------|-----------------|----------|
| [sp+1] ← [ldoi] | mov *k, r       | q~k und  |
|                 | lade Inh. von   | [sp+1]~r |
| q               | Adresse k auf r |          |

|                   |                  |          |
|-------------------|------------------|----------|
| [sp+1] ← [ldci] q | mov #k, r        | q~k und  |
|                   | lade Konstante k | [sp+1]~r |
|                   | auf r            |          |

|                |                    |            |
|----------------|--------------------|------------|
| [sp-1] ← [mpi] | mul r, o           | [sp]~r und |
|                | Multipliziere Inh. | [sp-1]~o   |
| [sp-1] [sp]    | von r mit o;       |            |
|                | Resultat in o      |            |

Register  $r$  ist ein "allgemeines Zweck" (general purpose) Register,  $r = \{r1, r2, r3, r4\}$  und  $o$  ein ungerades ( $o$  wie  $odd$ ) Register, hauptsächlich für Multiplikation und Division eingesetzt.

Die Kosten der Instruktionen wurden vernachlässigt, da sie irrelevant in diesem Zusammenhang waren.

Die Semantik der Maschineninstruktionen ist in Spalte II der Transformationstupel angegeben.

Nach der Anwendung der ersten zwei Transformationstupel (vorausgesetzt der C-Exeutor arbeitet gemäß der "maximal munch"-Heuristik) sieht sich der Codeerzeuger mit folgender Situation konfrontiert:



Wir haben angenommen, die Registerzuweisungsroutine stellt Register  $r3$

und r4 zur Verfügung. Was die Äquivalenz zwischen den Ressourcen der p-Maschine und Zielmaschine betrifft, gilt zu diesem Zeitpunkt:

[esp-1]~r4 und [esp]~r3.

Das dritte Transformationstapel fordert aber [esp-1]~o. Folglich wird vor der Ausgabe von "mul r3,o"-Instruktion noch die Register-Register Transferinstruktion "mov r4,o" erzeugt.

Die Möglichkeit, den Inhalt eines Registers R1 auf ein Register R2 zu transportieren, ist immer gegeben.

Eine nicht vorhandene Register-Register Transferinstruktion vom Typ: bringe Inhalt von Register R1 auf Register R2 kann durch "speichere"- und "lade"-Instruktionen realisiert werden:

- Speichere Inhalt von R1 unter Adresse a1;
- lade R2 mit Inhalt von a1.

In vielen Fällen können nicht vorhandene Register-Register Transferinstruktionen durch geschickten Gebrauch der Maschinenidiome realisiert werden.

Ein Beispiel dazu: lade Registerpaar DE mit Inhalt von SP (stack pointer) auf dem SAB 8080.

```
lxi HL,0      Lade HL mit Wert "0";  
dad SP       SP-Inhalt + 0 in HL;  
xchg        Vertausche Inhalt von HL und DE.
```

Allgemein gilt auch in diesem Fall: je primitiver (d.h. weniger orthogonaler Befehlssatz) die Zielmaschine, desto mehr muß der Benutzer "zu Fuß" erledigen.

Wie wir gesehen haben, müssen die Register-Register Transferinstruktionen erst zur Übersetzungszeit dem Codeerzeuger bekannt sein.

Wir haben nicht untersucht, ob es sinnvoller ist, die Behandlung der Register-Register Transferinstruktionen der Registerzuweisungsroutine zu übergeben oder separat zu unternehmen. Wahrscheinlich ist es kein großer Unterschied solange die Registerzuweisungsroutine als Unterprogramm des C-Executors arbeitet. Eine gute Registerverteilungsroutine vermeidet redundante "lade"- und Register-Register Transferbefehle zur Compilierungszeit, indem z.B. vor der Ausgabe einer solchen Instruktion der Registerinhalt mit dem zu ladenden Wert verglichen wird.

Konvertierungsinstruktionen vom Typ: mache aus einem "short"-Datentyp ein "long"-Datentyp werden ebenfalls wie die Register-Register Transferinstruktionen behandelt.

Redundante Befehle lassen sich durch Datenflußanalyse auch über die Grenzen der Eingabebäume erkennen. Die Methode wird in Rahmen der Maschinen-

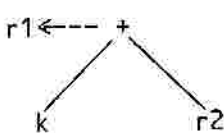
codeoptimierung verwendet ([Gi 82] und [Pi 85]).

Eine weitere Möglichkeit Register-Register Transferbefehle zu vermeiden, ergibt sich aus dem geschickten Gebrauch der Zielmaschineneigenschaften (machine idioms). Ein Beispiel dafür ist der Einsatz von 3-Adress-Instruktionen wie aus dem nächsten Abschnitt zu sehen ist.

### 8.8 Einsatz von 3-Adress-Instruktionen

3-Adress-Instruktionen sind dadurch charakterisiert, daß das Resultat der Operation einer Maschineninstruktion auf eine andere Maschinenresource lokalisiert wird als die Operanden. Für uns bedeutet dies (Maschineninstruktion als KRM-Regel): Die Ausgabeschablone der KRM-Regel ist verschieden von den Operanden der Eingabeschablone. Folgendes Beispiel stellt eine 3-Adress-Instruktion aus dem Befehlsatz einer PDP 11/70 Maschine dar.

#### Beispiel 8.8.1

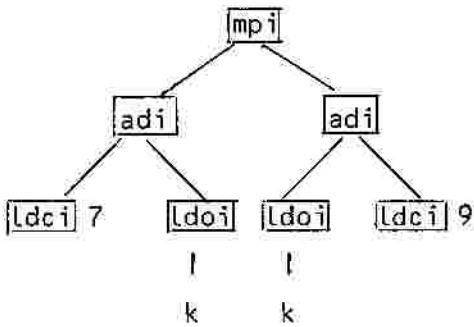
| <u>KRM-Regel</u>  | <u>Assembler-Mnemonic</u> | <u>Wirkung</u>   |
|---|---------------------------|--|
|  | add #k,r1,r2              | Addiere Konstante "k" zum Inhalt von r2; Resultat in r1. |

Geschickter Gebrauch von 3-Adress-Instruktionen kann Transferinstruktionen überflüssig machen. Vor allem bei der Bearbeitung **gemeinsamer Unterausdrücke** macht sich der Einsatz von 3-Adress-Instruktionen bezahlt ([GL 77] und [GL Gr 78]). Das nächste Beispiel zeigt welche Codeverbesserung sich aus dem Gebrauch von 3-Adress-Instruktionen ergibt.

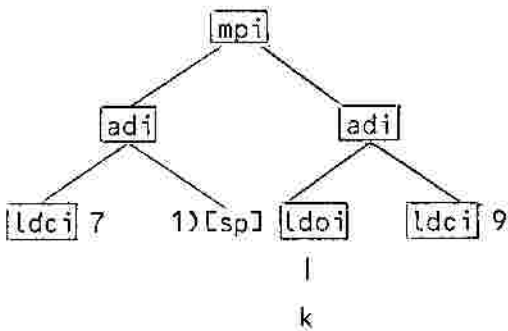
#### Beispiel 8.8.2

Angenommen der Codeerzeuger erhält folgenden P-Code Unterbaum zur Bearbeitung:

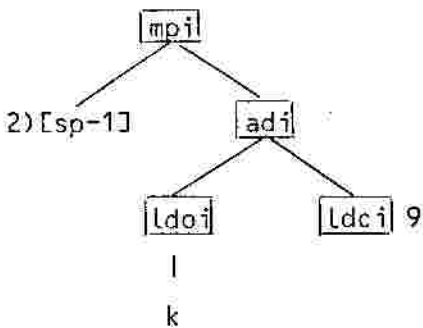




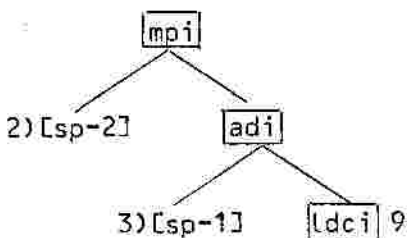
Störend bei diesem Baum ist die Tatsache, daß der Inhalt der Adresse "k" zweimal auf die oberste Stackposition geladen wird. In P-Code läßt sich dies nicht vermeiden, weil der Stackinhalt nach erfolgter Operation gelöscht wird. Die Abarbeitung dieses Baumes nach der Strategie des stack-computers ergibt auf einem PDP 11/70 Rechner:



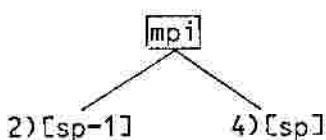
1) mov \*k,r1 ([sp]~r1)  
lade Inh. der Adresse "k" in r1



2) add #7,r1 ([sp-1]~r1)  
addiere Zahl 7 zu Inh. von r1

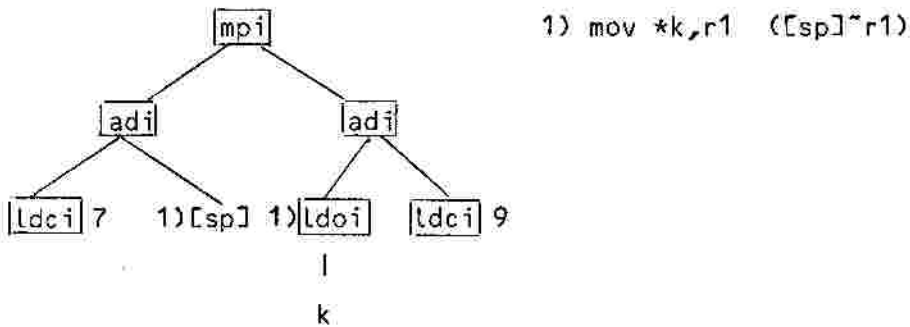


3) mov \*k,r2 ([sp-1]~r1 und [sp-2]~r2)

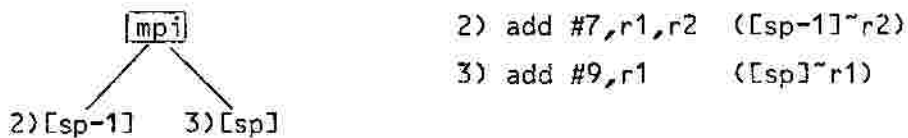


4) add #9,r2 ([sp-1]~r1 und [sp-2]~r2)

Unter Verwendung der 3-Adress-Instruktion aus Beispiel 8.8.1 erhalten wir:



1) `mov *k,r1 ([sp]~r1)`



2) `add #7,r1,r2 ([sp-1]~r2)`

3) `add #9,r1 ([sp]~r1)`

Dadurch ersparen wir uns die "mov \*k,r2"-Instruktion.

Um die kürzere Folge von Maschineninstruktionen zu erzeugen muß der C-Executor erkennen, daß der rechte Operand des linken "adi"-Befehls und der linke Operand des rechten "adi"-Befehls aus Beispiel 8.8.2 denselben Wert darstellen.

Dafür gibt es prinzipiell zwei Möglichkeiten:

- indirekt, durch Übergabe dieser Information vom front-end (siehe auch Behandlung der gemeinsamen Unterausdrücke in [Gl 77]).
- direkt, parallel zum pattern matching auf dem Eingabebaum.

### 8.9 Exotische Maschineninstruktionen

Eine Lösung für die Ausnutzung exotischer Instruktionen innerhalb eines automatischen Ansatzes für Codeerzeugung ist von Morgan in [Mo 84] vorgeschlagen worden.

Das von Morgan & Co entwickelte und implementierte System EXTRA (Exotic Instruction Transformational Analysis System) basiert auf folgendem Prinzip: Beschreibungen von Operatoren höherer Programmiersprachen und exotischer Maschineninstruktionen werden miteinander verglichen und transformiert (source-to-source) bis sie äquivalent werden. Die Ergebnisse der EXTRA-Analyse werden der Maschinenbeschreibung beigelegt.

Wir haben nicht untersucht, wie das Einfügen von EXTRA-Ergebnisse in unsere mc-Spezifikation zu erfolgen hat und welche Erweiterung dies für unser Generierungssystem impliziert. Sicher ist allerdings, daß in einem Fall zusätzliche Informationen vom front-end benötigt wird, um in-

nerhalb der P-Code Eingabebäume die Operatoren der höheren Programmiersprache zu identifizieren.

Eine Möglichkeit exotische Maschineninstruktionen einzusetzen ergibt sich für uns bei der Konstruktion der Abstraktionstupel. Das Problem haben wir nicht behandelt. In diesem Abschnitt wollen wir nur die Idee beschreiben.

Eine exotische Maschineninstruktion läßt sich durch eine Folge von einfachen (primitiven) Maschineninstruktionen ersetzen. Gelingt uns innerhalb der mc-Spezifikation eine exotische Maschineninstruktion "ins ex" durch eine Folge von  $n$  primitiven, beschreibbaren Maschineninstruktionen  $ins\ 1, \dots, ins\ n$  darzustellen, so kann der Generator aus den Abstraktionstupeln  $(r_{\Pi}(ins\ 1); r_{\sigma}(op\ 1)), \dots, (r_{\Pi}(ins\ n); r_{\sigma}(op\ n))$ , die zu "ins ex" entsprechende  $\sigma$ -Komponente zusammensetzen.

Die Hauptschwierigkeit dabei ist, daß die zu einer exotischen Maschineninstruktion äquivalente Folge von primitiven Maschineninstruktionen sich aus zum Teil nicht beschreibbaren Komponenten (z.B. Rotieren eines Registerinhalts) zusammensetzt.

Um dieses Problem zu analysieren ist eine Maschinenbeschreibung nötig, die vollständig die Semantik der Maschineninstruktionen (Haupt- und Nebeneffekte) berücksichtigt.

## Generator-Schema

### 9.1 Einführung

Bisher haben wir angenommen, die p-Spezifikationskomponente  $(p; \sigma)$  des Generators (Abb.II.2.2) wird vom Systementwickler ein für allemal festgelegt. Wegen der zum Teil großen Unterschiede zwischen der P- und Zielmaschine kann eine feste "P-Code -ARM Übersetzung" in manchen Fällen zu Effizienzverlusten führen. Betroffen davon sind P-Code Operatoren wie Sprünge, Prozeduraufruf- und Rückkehrbefehle, "floating point"- sowie komplexe P-Code Befehle.

Bedingte Sprünge werden z.B. in P-Code mittels effizienten Vergleichsoperatoren wie :  $les\ c (<)$ ,  $equ\ c (=)$ , etc ..., und des "fjp"-Operators (false jump) realisiert. Auf vielen Zielmaschinen werden aber bedingte Sprünge mittels einer Vergleich (compare)- und bedingten Sprunginstruktion (jump if equal, usw, ...) realisiert. Wird die P-Code Realisierung auf ARM-Ebene übernommen, so wird vom Benutzer erwartet, daß er bei der Beschreibung von Maschinen mit bedingten Sprunginstruktionen die oben beschriebene Situation erkennt und die primitive Substitutionsfunktion richtig angibt.

Eine Lösung, um in diesen Fällen dem Benutzer die Arbeit zu erleichtern, ist eine "flexible" P-Code-ARM Übersetzung. Dies wird durch Angabe von Optionen innerhalb des p-Spezifikationsschrittes realisiert.

Für den soeben beschriebenen Fall bedeutet dieser Gedanke die Angabe von zwei Übersetzungsvarianten in ARM:

- eine basierend auf effizienten Vergleichsoperatoren plus booleschem Sprung (entsprechend der P-Code Realisierung);
- die andere basierend auf einen nicht näher spezifiziertem Vergleichsoperator plus bedingten Sprüngen.

Bei der Generierung der Transformationstupel wird die Option gesetzt, die der Zielmaschinenarchitektur entspricht.

In diesem Zusammenhang reden wir nicht mehr von einem festen Übersetzungsschritt P-Code -ARM sondern von einem **Generator-Schema**.

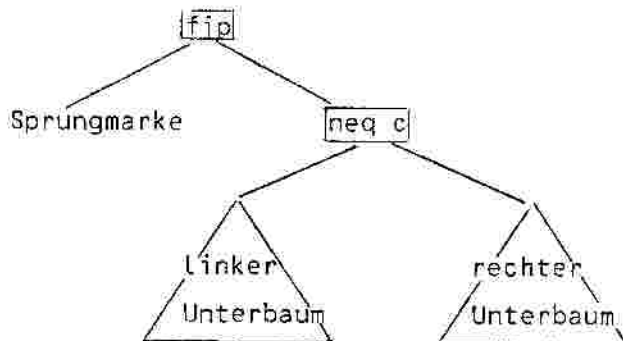
Um eventuellen Mißverständnissen vorzubeugen: die Angabe des Generator-Schemas bedeutet nicht, daß die p-Spezifikationskomponente

wechselnde Architekturen (siehe auch Diskussion in 1.5).

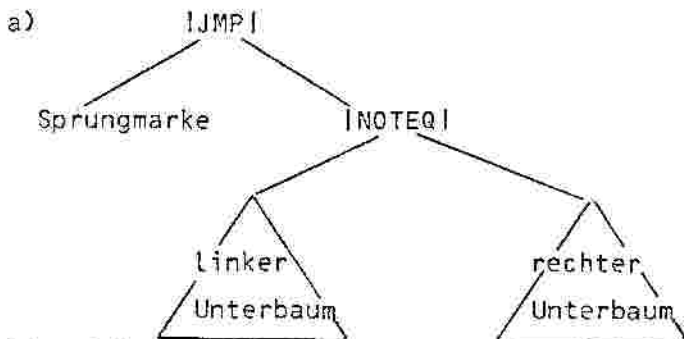
### 9.2 p-Spezifikation von bedingten Sprüngen

Effiziente Vergleichsoperatoren gefolgt vom "fjp"-Befehl betrachten wir als eine Einheit entsprechend einer bedingten Verzweigung in der höheren Programmiersprache.

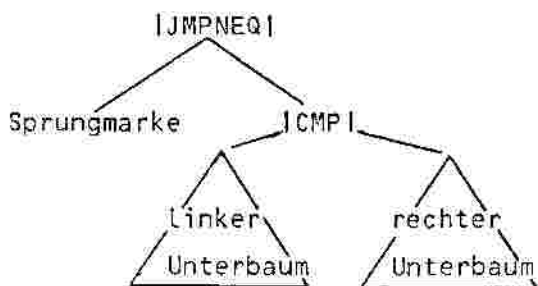
Die Folge von zwei P-Code Befehlen



wird zu:



für Zielmaschinen, die über effiziente Vergleichsoperatoren verfügen und zu:



für Zielmaschinen, die über bedingte Sprünge verfügen.

Ein ähnliches Problem entsteht bei der p-Spezifikation von Prozeduraufruf- und Rückkehrbefehlen.

### 9.3 Behandlung des P-Code Restkellers in ARM

Der Kellerbereich, der mit dem Aufruf und Rückkehr einer Prozedur (Funktion) assoziiert wird, haben wir Restkeller genannt, da er sich während der Abarbeitung eines Prozedurrumpfes im Gegensatz zum Lokalen Keller nicht ändert (4.1).

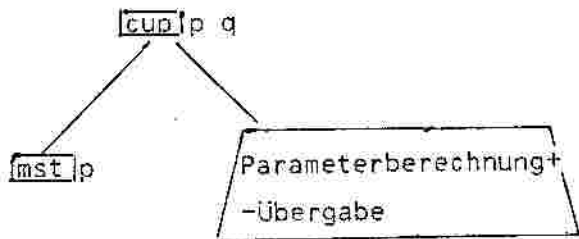
In diesem Abschnitt schlagen wir eine mögliche Realisierung für Prozeduraufrufe (9.3.1), Parameterberechnung und -übergabe (9.3.2) sowie Funktionsaufrufe (9.3.3) vor.

#### 9.3.1 Prozeduraufrufe

In diesem Fall sieht sich der Systementwickler ebenfalls mit zwei Ansätzen konfrontiert: "static link" und "display".

Einen Vorschlag für die Implementierung der auf static link basierenden Kellerorganisation wird hier dargestellt.

Ein Prozeduraufruf mit Berechnung und Übergabe von Parametern bildet in P-Code eine Einheit, die wir als Eingabebaum wie folgt darstellen:



Die zwei Befehle "mst" (mark stack) und "cup" (call user procedure) haben folgende Bedeutung (siehe auch P-Code Interpreter in [Pe 83]):

"mst p", mit  $p = \text{Niveau der aufrufenden Prozedur} - \text{Niveau der aufgerufenen Prozedur} + 1$ , ist für folgendes verantwortlich:

- trage  $sl$  (static link, d.h.  $\text{base}(p)$ ) und  $dl$  (dynamic link, ehemaliger MP-Wert) auf dem Stack.

- reserviere Restkellerplatz für die aufgerufte Prozedur (Funktion).

- setze EP auf neuen Wert um zu verhindern, daß "store"- (Siehe 4.1) und "heap"-Bereiche des Stacks überlappen.

"cup p q", mit  $p = \text{Parameteranzahl}$  und  $q = \text{Anfangsmarke (entry point)}$ , sorgt für:

- setzt MP auf Anfang des Kellerrahmens für die aktuelle Prozedur.

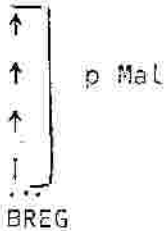
- rettet PC-Inhalt auf "Return"-Adresse.

- setzt PC auf q.

"sl" wird benutzt um Variablen in äußere Blöcke zu erreichen, "dl" um den Kellerrahmen nach Bearbeitung einer Prozedur wieder frei zu geben sowie als Zugriff auf lokale Variablen.

Der Restkeller der p-Maschine wird auf dem ARM-Stack, ein Bereich des ARM-Speichers, spezifiziert. Zu diesem Zweck legen wir folgende Korrespondenz zwischen den SC- und ARM-Ressourcen fest:

MP (dynamic link) wird mit dem Basisregister BREG assoziiert und SP mit dem ARM-Stack Pointer SPREG. Die Funktion base(p) wird durch wiederholten (p Mal) Zugriff auf das BREG realisiert, i.Z.:



Die Funktion base(0) wird dadurch zu BREG; bezieht sich stets auf dem aktuellen Prozedurkellerrahmen. In diesem Fall ist dl = sl.

Die ARM-Kellerkonfiguration für eine Prozedur (Funktion) sieht wie folgt aus:

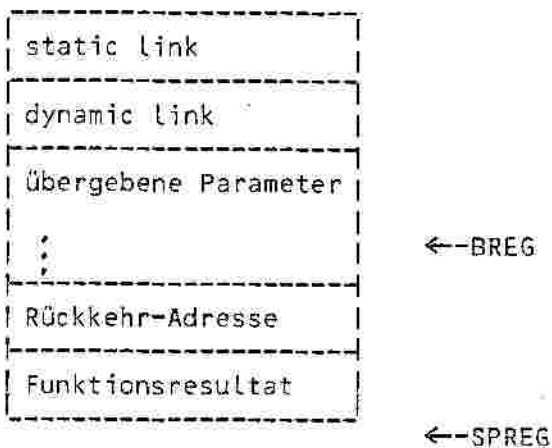


Abb.9.3.1 ARM-Kellerrahmen für Prozeduren (Funktionen)

Die P-Code "mst"- und "cup"-Befehle werden durch den ARM-Aufrufoperator |CALL| spezifiziert, der folgendes realisiert:

- Eintragen von sl, dl und der Rückkehr-Adresse auf dem ARM-Stack;
- Setzen des PC auf Anfang des Unterprogramms;
- Aktualisieren des BREG und SPREG.

Auf ARM-Ebene wird der |CALL|-Operator nicht näher spezifiziert, um, genau wie für XINDADR(p) (siehe 5.2.2.1) die Maschinenunabhängigkeit der ARM zu bewahren und eine effiziente Konkretisierung auf verschiedenen Zielmaschinen zu erreichen.

### 9.3.2 Parameterberechnung und -übergabe

Ein Abschnitt eines P-Code Eingabebaums entsprechend der Parameterberechnung und -übergabe (durch die P-Code Befehle "mst" und "cup" lokalisiert) wird vom C-Executor mit Hilfe der Transformationstapel abgearbeitet wie jeder andere Eingabebaum zur Compilierungszeit. Die "load"-Befehle unmittelbar vor dem "cup"-Befehl sind für die Parameterübergabe in P-Code verantwortlich.

Auf den meisten Zielmaschinen erfolgt die Übergabe von Parametern mit Hilfe von Instruktionen vom Typ "push".

In diesem Kontext haben wir P-Code "load"-Befehle durch ein [PUSH]-Operator in ARM spezifiziert.

Der Benutzer konkretisiert den ARM [PUSH]-Operator durch "push"-Instruktionen der Zielmaschine.

Ein Problem kann jedoch entstehen für Zielmaschinen mit einem kleinen Stack. In diesem Fall werden nur die Rückkehradressen auf dem Stack aufbewahrt. Der Stackbereich für die übergebenen Parameter wird auf einem freiem Speicherabschnitt simuliert. Die zwei Optionen der p-Spezifikation für "load"-Befehle, die Parameter übergeben, verwirklichen diesen Gedanken:

- die [PUSH]-Variante wird für Zielmaschinen mit einem ausreichend großen Stack gewählt,

- für Zielmaschinen mit einem kleinen Stack führen wir einen ARM-Transferbefehl [MOVL] ein.

Mit dem ARM [IRET]-Operator entsprechend dem P-Code "ret"-Befehl wird der aktuelle Kellerrahmen wieder freigegeben, static- und dynamic Link auf die alten Werte und der Programmzähler (PC) auf die Rückkehradresse gesetzt.

Bei der Konkretisierung des [IRET]-Operators muß der Benutzer für die Freigabe des Kellerschnitts entsprechend den übergebenen Prozedurparametern (durch "pop"- oder andere Transferinstruktionen) sorgen.

### 9.3.3 Funktionsaufrufe

Einen besonderen Fall stellt der Funktionsaufruf dar.

Die "cup"-Befehle, die den Funktionsaufruf bezeichnen, werden nicht als Wurzel der P-Code Eingabebäume betrachtet, weil die Funktion ein Resultat liefert, mit dem weitergearbeitet wird.



Folgendes Beispiel zeigt, wie die Abarbeitung eines Eingabebaums, der einen rekursiven Funktionsaufruf enthält (aus [Eu 84]), erfolgen könnte.

Beispiel 9.3.3

Folgender Baum entspricht dem "fakultät:=Nxfakultät(N-1)"-Statement:

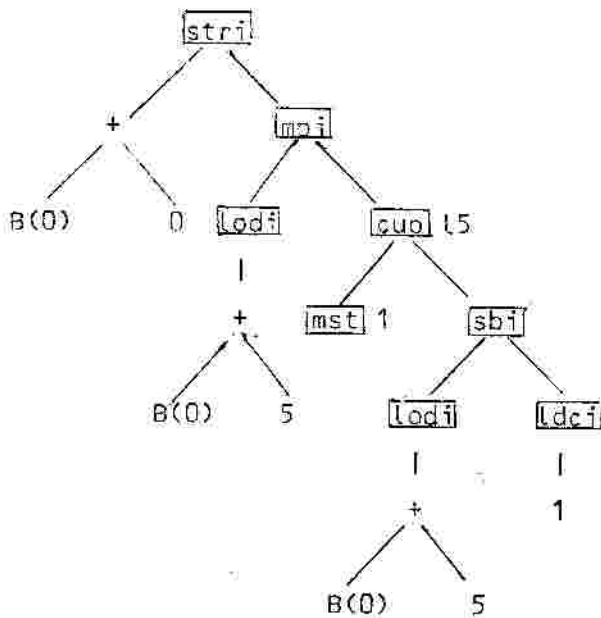


Abb. 9.3.3.1 P-Code Eingabebaum für rekursiven Funktionsaufruf

Nach Berechnung des Funktionsresultats wird der Eingabebaum zu:

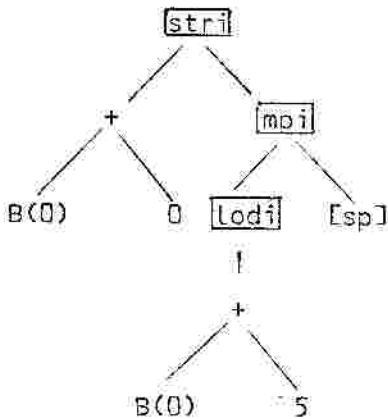
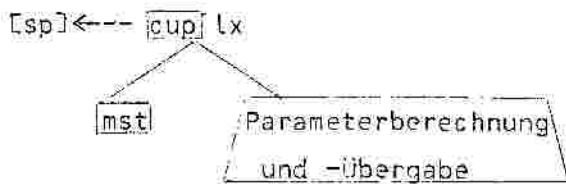


Abb. 9.3.3.2 Baum aus 9.3.3.1 nach abgearbeitetem Funktionsaufruf und die Abarbeitung des Baumes fortgesetzt.

Aus dem Beispiel 9.3.3 ergibt sich für die P-Code Regel  $r_p(\text{cup})$  eines Funktionsaufrufs:



wobei in diesem Fall "cup" kein Wurzelknoten des Eingabebaums ist. Die Ausgabeschablone von  $r_p(\text{cup})$  wird zur Compilierungszeit bestimmt. So ist z.B.  $a_p(\text{cup})=[\text{sp}-1]$ , falls "fakultät(N-1)xN" die rechte Seite des Statements aus Beispiel 9.3.3 gewesen wäre.

In diesem Fall haben wir keine ARM-Sonderregel der "cup"-Regel assoziiert. Der C-Executor braucht keinen Hinweis, daß es sich hier um einen Funktionsaufruf handelt, damit das Funktionsresultat an der richtigen Stelle weitergereicht wird. Der Aspekt des Eingabebaums ("cup" als innerer Knoten statt Wurzel) und der Vergleich zwischen den Ressourcen von p-Code - und Zielmaschine aus der expliziten Form der Transformationstapel (6.7) reichen dafür aus.

#### 9.4 p-Spezifikation von komplexen P-Code Befehlen

Eine Reihe von P-Code Operatoren realisieren komplexe Berechnungen auf dem stack-computer. Repräsentativ dafür sind: "mod" (Modulo), "chk" (prüft nach, ob ein Wert zwischen zwei angegebenen Grenzen liegt), "ixa" (berechnet eine indexierte Adresse), "mov" (Block-Transfer von hintereinander gespeicherten Werten), etc.

In der Regel haben diese P-Code Befehle keine direkte korrespondierenden Maschineninstruktionen.

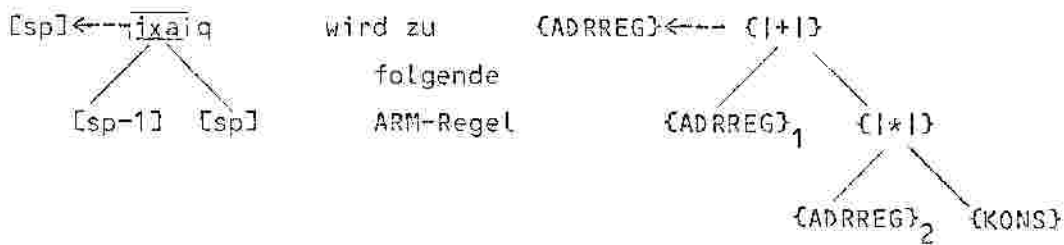
Würden wir korrespondierende ARM-Operatoren einfügen, z.B. |IXA| für "ixa", so sieht sich der Benutzer bei der Angabe der primitiven Substitutionsfunktion einer aufwändigen Aufgabe gegenübergestellt: der Makroexpansion eines komplexen ARM-Operators. Dies bedeutet aber dasselbe wie die Makroexpansion eines P-Code Befehls auf Zielmaschinenebene.

Aus diesem Grund haben wir uns entschlossen, die Makroexpansion eines komplexen P-Code Befehls schon auf ARM-Ebene zu unternehmen.

##### Beispiel 9.4.1

Der P-Code Befehl "ixa q" berechnet eine Adresse aus den beiden obersten Stackinhalten wie folgt: die Basisadresse (auf [sp-1]) wird zum Resultat von [sp]\*q addiert.

Die P-Code Regel ist:



Die ARM-Regel  $r_p(ixai q)$  wird zu einer KRM-Metaregel expandiert und anschließend dem  $\sigma$ -pattern matcher als Eingabe geliefert wie in Kap.6 beschrieben.

Die Indizes "1" und "2" rechts von den Mengenklammern von  $\{ADRREG\}$  aus Beispiel 9.4.1 weisen darauf hin, daß die substituierten Adressregister der Zielmaschine verschieden sein müssen (so wie  $[sp-1]$  und  $[sp]$  in P-Code). Im Gegensatz zu P-Code aber, muß die Ausgabeschablone der ARM-Regel nicht identisch mit einem der Adressregister der Eingabeschablone sein.

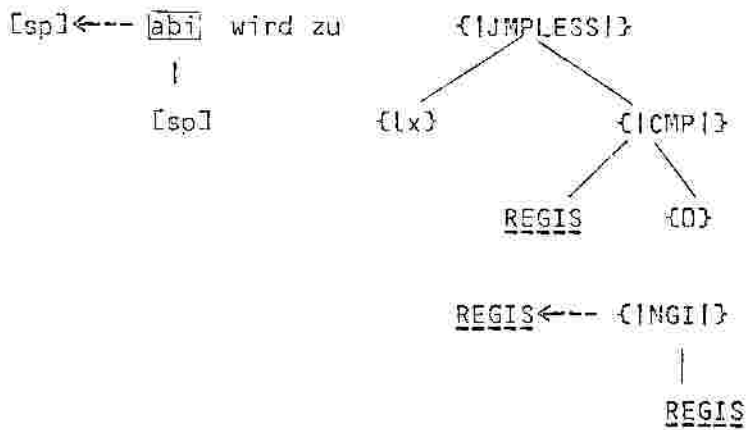
Es gibt natürlich keine Garantie dafür, daß die in Beispiel 9.4.1, angegebene Lösung für die  $r_p(ixai q)$ -Regel auch die effizienteste für eine breite Klasse von Zielmaschinen ist.

Um komplexe P-Code Operatoren effizient auf verschiedene Zielmaschinen umzusetzen, muß der Systementwickler je nach Zielmaschinen-Architektur mehrere Lösungen, d.h. Optionen des Generators-Schemas, parat haben. Dieser Ansatz unterstützt die automatische Makroexpansion von komplexen P-Code Befehlen, allerdings unter Verlust der "echten" Maschinenunabhängigkeit des p-Spezifikationsschrittes.

Anders als in Beispiel 9.4.1 erfordern die meisten komplexen P-Code Befehle eine Folge von ARM-Regeln bei Angabe der p-Spezifikation. Folgendes Beispiel stellt diese Situation dar.

#### Beispiel 9.4.2

Die P-Code Regel  $r_p(abi)$  ("abi" heißt: berechne absolut Betrag aus einer "integer" Zahl)



{!x ...

Dabei haben wir angenommen, die "integer" Zahl befindet sich in REGIS. Die Interpretation der ARM-Codesequenz lautet: Invertiere das Vorzeichen des REGIS-Inhaltes, falls dieser Wert <0 ist.

Die Makroexpansion von P-Code Regeln in ARM erfolgt mit Hilfe elementarer (primitiver) ARM-Operatoren. Dadurch vermeiden wir die Einführung komplexer ARM-Operatoren in  $Z_{\sigma}$  den ARM-Zeichensatz.

## Schlußfolgerungen

Die meisten Codeerzeuger-Generatoren, die auf pattern matching basieren, konzentrieren sich auf die Verarbeitung der Maschinenbeschreibung. Unabhängig davon muß der Zwischencode zur Übersetzungszeit auf das semantische Niveau der Zielmaschine gebracht werden.

In unserem Ansatz werden diese beiden Aufgaben zur Generierungszeit gelöst. Aus diesem Grund haben wir nicht gefordert, daß das front-end direkt ARM liefert. Wie aus Kap.9 zu sehen ist, können wir die "Anpassung" des P-Codes an verschiedene Zielmaschinen im p-Spezifikation-Schritt übernehmen, was die Verkürzung der Übersetzung von P- in Maschinencode impliziert.

Gegenüber anderen generativen Ansätzen enthält unser System zusätzlich die Konstruktion der Substitutionstupel (automatische Makroexpansion).

Die automatische Makroexpansion eines jeden P-Code Befehls sichert dem Codeerzeuger die Abarbeitung jeder beliebigen Folge von P-Code Befehlen. Das generierte Produkt sind patterns in P-Code. Damit kann der pattern matcher der Übersetzungszeit (C-Executor) direkt P-Code Programme durchsuchen.

Die Generierung von P-Code patterns ist ebenfalls ein pattern matching-Prozess. Das Verfahren ist nicht an einen bestimmten Typ von pattern matcher gebunden. Dasselbe gilt auch für die P-Code patterns. Dadurch läßt sich das generierte Produkt zusammen mit einem beliebigen C-Executor zum Codeerzeuger ausbauen.

### Literaturverzeichnis

[Ah Ga 84]

A. V. Aho, M. Ganapathi: Efficient Tree Pattern Matching: an Aid to Code Generation, ACM Vol. 1, 1985

[Ah U 77]

A. V. Aho, J. D. Ullman: Principles of Compiler Design, Addison Wesley, 1977

[Ak Le 82]

T. A. Akin, R. Leblanc: The Design and Implementation of a Code Generation Tool, Software-Practice and Experience Vol. 12, 1982

[Bir 82]

P. Bird: An Implementation of a Code Generator Specification Language for Table Driven Code Generators, Compiler Construction 82, Boston, Mass. 1982

[Br Se 76]

J. Bruno, R. Sethi: Code Generation for a One-Register Machine, Journal of the ACM Vol. 23, 1976

[Cat 77]

R. G. Cattell: A Survey and Critique of Some Models of Code Generation, Department of Computer Science Report, Carnegie-Mellon University, 1977

[Craw 82]

J. Crawford: Engineering a Production Code Generator; ACM Sygplan, Sygplan Symposium, Juni 1982

[Do Noo Fe 79]

M. K. Donegan, R. Noonan, S. Feyock: A Code Generation Language, ACM Sigplan Notices 14(8), 1979

M. Eulenstein: POCO, ein portables System zur Generierung portabler Compiler; Dissertation, Univ. des Saarlandes, Saarbrücken, 1984

[Gana 80]

M. Ganapathi: Retargetable Code Generation and Optimization Using Attribute Grammars, Ph. D. Dissertation, Univ. of Wisconsin-Madison, 1980

[Gie 81]

R. Giegerich: Automatische Erzeugung von Maschinencode-Optimierern, TU München IS112, 1981

[GL 77]

R. S. Glanville: A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers, Ph. D. Thesis, Berkley, Univ. of California, 1977

[GL Gr 78]

R. S. Glanville, S. L. Graham: A New Method for Compiler Code Generation, Fifth ACM Symposium, 1978

[GL MÖ Wi 80]

I. Glasner, U. Mönke, R. Wilhelm: OPTRAN, a Language for the Specification of Program Transformations, Informatik-Fachberichte 34, Springer 1980

[Gr 80]

S. L. Graham: Table Driven Code Generation, IEEE Computer 13(8), 1980

[Gr He Schu 82]

S. L. Graham, R. R. Henry, R. A. Schulman: An Experiment in Table Driven Code Generation, ACM Sigplan Notices 17, 1982

[He 81]

R. R. Henry: The Code Generator Generator's Work Station: Experiments with the Graham-Glanville Machine Independent Code Algorithmus for Code Generation, Master's Project Report, Berkeley, Univ. of California, 1981

[Ho Sche 83]

R. N. Horspool, A. Scheunemann: A Methodology for Automatic Code Generation, Technical Report, Univ. of Victoria, 1983

[La Jan Go 82]

R. Landwehr, H. Jansohn, G. Goos: Experience with an Automatic Code Generator Generator, ACM Sigplan Notices 17, 1982

[Lu 83]

H. Lunell: Code Generator Writing Systems, Ph. D. Thesis, Software Systems Research Center, S-581 83, Linköping, 1983

[No Am 74]

K. Nori, U. Amman, K. Jensen, H. Nägeli: The Pascal P-Compiler; Implementation Notes, Inst. für Informatik Zürich, 1974

[Mo Ro 82]

T. M. Morgan, L. A. Rowe: Analysing Exotic Instructions for a Retargetable Code Generator, ACM Sigplan Notices 17, 1982

[Pi 85]

S. Pistorius: Die Generierung von Datenflußanalyse zum Zwecke der Maschinencode-Optimierung, Diplomarbeit, Institut für Informatik, Saarbrücken, 1985

[PDP 84]

PDP-11/70 Architecture Handbook, Digital, 1984

[SAB 78]

System SAB 8080, Siemens 1978

[Wilm 84]

W. Wilmertinger: Denotationelle Maschinenbeschreibung, TU München, 1984

[Wilh 85]

Compilerbau II, Vorlesungsskript, Institut für Informatik, Saarbrücken, 1985

[Zi 83]

H. Zima: Compilerbau I und II, Wissenschaftsverlag BI, 1983