ÜBERSETZERBAU

EIN KLEINER ÜBERBLICK

Rubino Geiß
rubino@ipd.info.uni-karlsruhe.de

Sebastian Hack
hack@ipd.info.uni-karlsruhe.de

IPD Lehrstuhl Goos
Fakultät für Informatik
Universität Karlsruhe (TH)

Interner Bericht 2003-18
ISSN 1432-7864

VORWORT

Übersetzerbau-Forschung ist heutzutage hauptsächlich durch die Entwicklung von Optimierungs- und Codeerzeugungs-Verfahren geprägt. Dadurch sollen Übersetzer in die Lage versetzt werden, die vielfältigen Eigenschaften moderner Prozessoren effizient zu nutzen. Die Grundlagen moderner Übersetzer, insbesondere die Überprüfung des Eingabeprogramms hinsichtlich syntaktischer und semantischer Regeln sind wohlverstanden und beruhen auf Erkenntnissen aus den 60er und 70er Jahren. Das Erzeugen von lauffähigem Code ist ebenso gut erforscht. Diese Verfahren sind auch über den Übersetzerbau hinaus grundlegend für die Informatik.

In den folgenden Kapiteln geben wir eine kleine Einführung in die Arbeitsweise von Übersetzern. Wir verzichten bewusst auf die meisten (formalen) Details und Algorithmen, da wir Einsteigern ermöglichen wollen, die verschiedenen Komponenten dieser komplexen Systeme zu überblicken und sich eine intuitive Vorstellung von Übersetzern zu bilden.

Für das detaillierte Studium der vorgestellten Konzepte, Verfahren und Algorithmen werden wir jeweils Literatur benennen. Die im Anhang besprochenen Standardwerke sind von allgemeinem Interesse.

Zwar ist die detaillierte Beschreibung der einzelnen Verfahren unerlässlich für ihr genaues Verständnis, wir sind jedoch der Meinung, dass eine ungefähre Vorstellung von dem, was ein Verfahren bewirkt, das tiefere Verständnis erleichtert. Es besteht dennoch die Gefahr, dass eine fehlgeleitete Intuition dem tieferen Verstehen im Wege steht. Falls dies durch diesen Text geschehen sein sollte, bitten wir um die Nachsicht des Lesers. Wir sind für Kritik und Verbesserungsvorschläge äußerst dankbar, da uns an einer ständigen Verbesserung dieses Textes gelegen ist.

Karlsruhe, im September 2003

Rubino Geiß und Sebastian Hack

INHALTSVERZEICHNIS

KAPITEL 1

EINLEITUNG

Übersetzer sind Programme, welche verschiedene Darstellungen von Informationen ineinander überführen. So ist z.B. das LaTeX-Textsatzsystem ein Übersetzer, der dieses Dokument aus einer mit Formatierungsanweisungen angereicherten Textdatei in eine DVI-Datei überführt hat. Wichtig für uns sind aber jene Übersetzer, die Computerprogramme einer imperativen Hochsprache (wie Java, C, C++, Pascal, etc.) in eine Repräsentation umwandeln, die von Mikroprozessoren ausgeführt werden kann (Maschinensprache).

Übersetzer sind ein unverzichtbares Hilfsmittel der Software-Entwicklung, da die meiste Software nicht „von Hand“ in Maschinensprache, sondern in einer Hochsprache, wie z.B. Java oder C++ geschrieben wird. Für den Benutzer des Übersetzers steht meist nicht die als gegeben vorausgesetzte korrekte Übersetzung[1], sondern die Übersetzung in ein möglichst effizientes Programm, im Vordergrund. Mit „effizient“ können mehrere Dinge gemeint sein. Zum Beispiel ist man bei eingebetteten Systemen an einem kleinen Umfang des übersetzten Programms oder auch einem geringen Energieverbrauch interessiert, da Speicherplatz und Energie dort knapp und teuer sind. Andererseits ist bei einem Video-Codec Speicherplatz (sowohl die Größe der Programmdatei, also auch der Laufzeitspeicherverbrauch) nicht ausschlaggebend. Hier ist es eher von Bedeutung, wie schnell das Programm arbeitet.

1.1 Terminologie

In dieser kleinen Übersicht werden einige Begriffe, die charakteristisch für Programme und Übersetzer imperativer Programmiersprachen sind, immer wieder verwendet. An dieser Stelle wollen wir diese Begriffe kurz klären.

Als Vorbemerkung sei noch gesagt, dass wir für das *Rechnen*[2] das Modell des *von Neumann-Rechners* verwenden. Unsere Recheneinheit hat einen Befehlszeiger, der auf den Befehl zeigt, der gerade ausgeführt wird, und greift über einen Bus auf einen Hauptspeicher zu. Die Befehle und die Daten liegen gemeinsam in diesem Hauptspeicher. Dies entspricht den modernen (relevanten) Rechnerarchitekturen, für die Übersetzerbau interessant ist.

Ein Programm einer imperativen Programmiersprache besteht aus Steuerstrukturen (Anweisungen, Verzweigungen und Schleifen) die den Befehlszeiger steuern. Ferner besteht es aus Variablendefinitionen, die Stellen im Hauptspeicher des Rechners entsprechen (siehe unten).

Quellprogramm
: Das Programm, das der Übersetzer (meist in Form einer Textdatei) als Eingabe erhält.

Zielarchitektur
: Die Rechnerarchitektur (meist ein Prozessor), in deren Maschinensprache das Quellprogramm transformiert werden soll.

[1] Korrekte Übersetzung ist u.a. bezüglich eines Sprachberichts (engl.: language report) zu definieren

[2] das englische „Computing“ drückt vielleicht besser aus, was damit gemeint ist

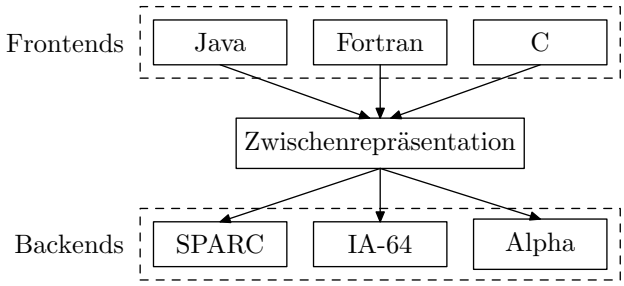


Abbildung 1.1: Schematischer Aufbau eines Übersetzers

Werte und Variablen
: Variablen benennen die Speicherzellen eines von Neumann-Rechners und Werte entsprechen deren Inhalt. Im Allgemeinen legt der Übersetzer während der Übersetzung noch viele weitere Variablen und Werte an, zum Beispiel zur Adressrechnung.

Berechnung
: Unter Berechnung verstehen wir die Anwendung einer Operation, die n Werte auf m Werte abbildet, wobei $n, m \in \mathbf{N}_0$. Für die Additionsoperation + ist $n = 2, m = 1$.

Auf den folgenden Seiten tauchen immer wieder kleine Codebeispiele auf, die in einer Pseudo-Maschinensprache formuliert sind. Anhang A gibt einen Überlick über diese Maschinensprache.

1.2 Architektur

Übersetzer sind heutzutage modular aufgebaut und in Phasen eingeteilt. Zentraler Bestandteil des Übersetzers ist die so genannte *Zwischendarstellung*[3]. Sie stellt das Quellprogramm so weit wie möglich unabhängig von der Quellsprache einerseits, und der Zielarchitektur andererseits dar. Die meisten Analysen und Optimierungen des Übersetzers arbeiten auf der Zwischendarstellung.

Das *Frontend* hat die Aufgabe, ein Eingabeprogramm (im weiteren Quellprogramm genannt) auf *Wohlgeformtheit*[4] im Sinne der Sprachdefinition der Quellsprache zu überprüfen und die Programmsemantik in die Zwischendarstellung zu überführen.

Das *Backend* erzeugt aus dem in der Zwischendarstellung dargestellten Programm Befehle für eine bestimmte Zielarchitektur, das sogenannte Zielprogramm. Das muss nicht zwangsläufig die Maschinensprache eines Prozessors (der sogenannten Zielarchitektur) sein. Man könnte zum Beispiel Java nach C übersetzen.

Diese Architektur hat den Vorteil, dass man nicht für jede Quellsprachen-Zielarchitektur-Kombination einen eigenen Übersetzer schreiben muss. Man konzipiert als Hauptbestandteil eine Zwischendarstellung, an die man „nur noch“ Frontends und Backends anschließen muss. Abbildung 1.1 stellt dies schematisch dar. Abbildung 6.1 zeigt die Übersetzer-Architektur am IPD im Detail.

[3] engl.: intermediate representation (IR)
[4] Man spricht auch von der Korrektheit der Syntax und der statischen Semantik.

KAPITEL 2

DAS FRONTEND

Das Frontend ist für die Überprüfung der Syntax und der statischen Semantik des Eingabetextes und das Überführen desselben in die Zwischendarstellung zuständig. Das Frontend besteht im wesentlichen aus zwei Teilen: Der *syntaktischen Analyse* und der *semantischen Analyse*[1]. Das Aufteilen in zwei Teile ist nicht zwingend notwendig, sondern geschieht hauptsächlich aus softwaretechnischen Gründen.

2.1 Syntaktische Analyse

Wie jede Sprache besitzt die Quellsprache (z.B. Java, C oder Fortran) eine Syntax, also ein Regelwerk, das bestimmt, wie man Sätze in dieser Sprache formuliert. Zum Beispiel schreibt die Syntax der deutschen Sprache vor, dass Wörter durch Leerzeichen getrennt werden, somit ist

```
dasisteinbaum
```

kein Satz der deutschen Sprache. Die Syntax der Programmiersprache Java schreibt z.B. vor, dass für jedes Zeichen „`(`“ immer ein darauffolgendes schließendes Zeichen „`)`“ auffindbar sein muss.

Die syntaktische Analyse besitzt eine vollständige Spezifikation der Syntax einer Quellsprache und prüft, ob der Eingabetext dieser Spezifikation genügt. Hierzu hat sich aus „pragmatischen“ Gründen die Verwendung einer Teilmenge der kontextfreien formalen Sprachen[2] durchgesetzt.

Um zu überprüfen, ob ein Eingabetext einer Syntaxspezifikation genügt, muss der Übersetzer das *Wortproblem* der formalen Sprache lösen. Man sagt auch, der Übersetzer *zerteilt*[3] den Eingabetext. Nun wird auch deutlich, warum nur ein Teil der kontextfreien Sprachen verwendet werden kann. Das Wortproblem der kontextfreien Sprachen ist im Allgemeinen nur von *indeterministischen* Kellerautomaten lösbar. Aber indeterministisch zerteilbare Sprachen sind praktisch ungeeignet, da sie für Menschen eine nur schwer durchschaubare Syntax der Quellsprache zur Folge hätten, und die Implementierung eines indeterministischen Kellerautomaten ineffizient ist. Entscheidend für den Übersetzerbau sind die Untermengen LL(k) und LALR(1) (siehe [GW84]), da die entsprechenden Kellerautomaten das Wortproblem *deterministisch* in $O(n)$ lösen, wobei n die Länge des Eingabetextes ist.

Da die manuelle Erstellung dieser Kellerautomaten jedoch sehr fehleranfällig und zeitraubend ist, benutzt man dafür Werkzeuge[4] . Sie erzeugen aus einer gegebenen Grammatik ein Programm, das den deterministischen Kellerautomaten implementiert. Dieses Programm führt dann die syntaktische Analyse durch. Das Unix Werkzeug YACC erstellt zum Beispiel LALR(1) Kellerautomaten.

Letztlich ist man aber nicht nur an der Lösung des Wortproblems in Form einer ja/nein-Antwort (Eingabetext ist Wort der Sprache oder nicht) interessiert, sondern man will auch die vom Kellerautomaten vorgenommenen Schritte nachvollziehen können. Man kann sich leicht überlegen,

[1] Man nennt das Frontend deswegen auch *Analysephase*
[2] auch bekannt als Chomsky-2
[3] engl.: to parse
[4] wie Bison (bzw. Yacc) oder ANTLR (oder einen mächtigeren Übersetzerbaukasten, wie Cocktail oder Eli)

$$
\begin{aligned}
G &= (S, N, T, P) \\
T &= \{*, +, (,), x\} \\
N &= \{E, T\} \\
P &= \{S \to E, E \to E + T\} \\
&\cup \{E \to T, T \to T * x, T \to x\}
\end{aligned}
$$

`x * x + x` →

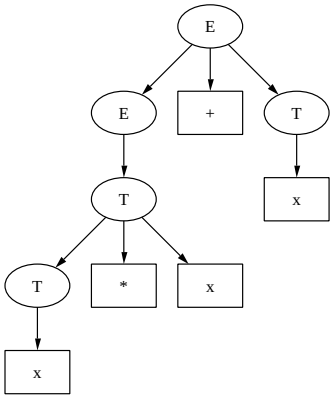


Abbildung 2.1: Grammatik der Quellsprache, Eingabetext und resultierender Strukturbaum

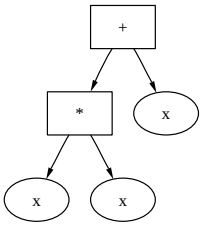


Abbildung 2.2: AST

dass diese Informationen als Baum dargestellt werden können. Die Blätter des Baums sind die Terminale, die inneren Knoten im Baum sind die Nichtterminale und die Wurzel bildet das Startsymbol der Grammatik. Dieser Baum heißt *Strukturbaum*[5]. Abbildung 2.1 zeigt die Grammatik einer Quellsprache, einen möglichen Eingabetext und den resultierenden Strukturbaum.

Das Nichtterminal T in der Grammatik aus Abbildung 2.1 ist nur eingeführt, um die Priorität der Operatoren + und $*$ zu bestimmen und somit die Grammatik eindeutig (deterministisch zerteilbar) zu machen. Es trägt keine Information, die für den Übersetzer weiter wichtig wäre. Von Bedeutung ist für den Übersetzer nur, dass durch + jene zwei Ausdrücke addiert werden, die durch x und durch die Multiplikation zweier weiterer Ausdrücke gegeben sind. Deswegen wird der Strukturbaum in den *Abstrakten Strukturbaum*[6] überführt. Der AST zum Strukturbaum aus Abbildung 2.1 ist in Abbildung 2.2 zu sehen.

Es bleibt noch die lexikalische Analyse zu erwähnen, die als Optimierung der syntaktischen Analyse in den meisten Übersetzern zum Einsatz kommt. Es hat sich nämlich gezeigt, dass weite Teile der kontextfreien Grammatik, die zur Beschreibung der Syntax der Quellsprache verwendet wird, sogar *regulär*[7] ist, also mit Hilfe von regulären Ausdrücken beschreibbar ist. Man schaltet also gewöhnlich dem Kellerautomaten einen DEA[8] vor, der den Eingabetext in größere „Häppchen“[9] unterteilt. Zum Beispiel werden Ziffernfolgen zu Zahlen zusammengefasst. Der Kellerautomat sieht statt Ziffernfolgen dann nur noch Nichtterminale wie ⟨Ganzzahl⟩ oder ⟨Gleitkommazahl⟩. Die Terminale der Grammatik des Kellerautomaten sind also nicht die einzelnen Zeichen der Eingabedatei,

[5] engl.: structure tree (ST) oder parse tree
[6] engl.: abstract syntax tree, oder kurz: AST
[7] Chomsky-3
[8] Deterministischer Endlicher Automat, engl.: DFA (Deterministic Finite Automaton)
[9] im englischen Token genannt

sondern daraus gebildete Agglomerate.

2.2 Semantische Analyse

Leider sind die *kontextfreien Grammatiken*[10] nicht in der Lage die Regeln der statischen Semantik zu handhaben. So ist zum Beispiel in Java nur ein Funktionsaufruf der Form `x.foobar(a, b, c)` möglich, wenn `foobar` tatsächlich als Funktion mit drei Argumenten in der Klasse von `x` definiert ist[11]. Um diese Regeln ausdrücken zu können reicht die Ausdrucksstärke der kontextfreien Grammatiken nicht aus.

Alle benötigten Informationen sind entweder im AST vorhanden oder können daraus berechnet werden. Mit diesen Wissen kann nun endgültig geprüft werden ob das Programm wohlgeformt ist. Dabei kann man alle Bedingungen, die eine Programmiersprache vorschreibt von Hand prüfen, sprich man programmiert die Überprüfung jeder Bedingung im Übersetzer aus[12], oder man bedient sich *attributierten Grammatiken* (AG). Man versieht die Knoten des AST mit Attributen (ein Knoten, der einen Ausdruck darstellt kann z.B. mit dem Typ dieses Ausdrucks attributiert werden) und gibt an, von welchen anderen Attributen (auch anderer Knoten) die Berechnung der Attribute dieses Knoten abhängt. Diese Spezifikation wird von einem Werkzeug gelesen, das eine Besuchsreihenfolge der Knoten erstellt, so dass alle Attribute ausgewertet werden können. Als Ausgabe liefert dieses Werkzeug ein Programm, dass das Besuchen der Knoten in einem AST vornimmt.

Es ist sinnvoll, nur eine eingeschränkte Klasse von AGs zu betrachten[13], da i.A. eine Prüfung auf Wohlgeformtheit nur mir exponentiellem Aufwand möglich ist, und überdies die Besuchsreihenfolge nicht allgemein festgelegt werden kann. Näheres zu AGs findet man wiederum in [GW84].

2.3 Zusammenfassung

Das Frontend eines Übersetzers ist für imperative und objektorientierte Sprachen für die Forschung nicht mehr von zentralem Interesse. (Bei funktionalen Sprachen ist das anders.) Für die Effizienz des Zielprogramms ist es nicht von Bedeutung. Es leistet lediglich eine Überprüfung der Eingabe auf Wohlgeformtheit, und eine Überführung des Eingabetextes in die internen Datenstrukturen des Übersetzers.

Alle wesentlichen wissenschaftlichen Arbeiten stammen aus den Jahren 1960–1980. Mit den LALR(1) und LL(k) Untermengen der kontextfreien Sprachen, den dazugehörigen Konstruktionsalgorithmen der Kellerautomaten und den Geordneten Attributierten Grammatiken stehen heute Techniken zur Verfügung, die sich leicht in Werkzeuge[14] umsetzen lassen, somit ist das Erstellen eine Übersetzer-Frontends eher unspektakulär.

[10] auch Chomsky-2 genannt

[11] Hierbei müssen auch noch die Typen von `a`, `b` und `c` mit denen der Funktionsdeklaration kompatibel sein

[12] GCC macht das tatsächlich so

[13] Die allgemeinste Klasse die sich gut handhaben lässt, heißt geordnete attributierte Grammatik (OAG).

[14] Die aus unserer Sicht vollständigste und gelungenste Sammlung dieser Werkzeuge ist ELI (siehe [GLH+92] `http://www.uni-paderborn.de/project-hp/eli.html`).

KAPITEL 3

DIE ZWISCHENDARSTELLUNG

Für Zwischendarstellungen gibt es keinen „Standard“ wie die LALR(1) Grammatiken für die syntaktische Analyse. Die Zwischendarstellung ist meistens von Übersetzer zu Übersetzer unterschiedlich umgesetzt, da der heilige Gral der Zwischendarstellungen noch nicht gefunden worden ist. Es gibt aber gewisse Verfahren und Optimierungen, die allgemein anerkannt sind und „einfach dazugehören“.

Den Anschluss des Frontends an die weiteren Phasen des Übersetzers bildet die sogenannte Transformation, die die Zwischendarstellung aus dem AST erzeugt.

3.1 Datenfluss und Steuerfluss

Die Bedeutung eines Programms einer (imperativen) Hochsprache[1] kann im wesentlichen durch Daten- und Steuerfluss ausgedrückt werden.

Datenfluss

Der Datenfluss ist der „Fluss“ der Werte (siehe Abschnitt 1.1) durch die Variablen, wobei Berechnungen Zusammenflüsse des Datenflusses darstellen. Konstanten und parameterlose Funktionen sind Quellen des Datenflusses. Funktionsrücksprünge (z.B. das `return` in C, C++ und Java), Funktionen ohne Rückgabewert und das Schreiben in den Speicher sind i.A. dessen Senken. Man kann den Datenfluss als gerichteten Graphen darstellen, wobei die Werte die Kanten, und die Berechnungen die Knoten des Datenflussgraphen[2] sind.

> Ein `+` führt zwei Datenflüsse zusammen, indem es aus zwei Werten einen neuen berechnet. Abbildung 3.1 zeigt ein Beispiel für einen Datenflussgraphen.

Wenn man den dualen Graphen[3] betrachtet, so stellen die Kanten nun die Datenabhängigkeiten dar. Dieser Datenabhängigkeitsgraph[4] drückt aus, welche Werte vor anderen berechnet werden müssen.

Steuerfluss

Der Steuerfluss ist letztlich eine Abstraktion des Befehlszeigers unserer von Neumann-Maschine. Er bestimmt den Inhalt des Befehlszeigers und damit die Abarbeitungsreihenfolge der Berechnungen. Auch der Steuerfluss ist als gerichteter Graph darstellbar. Man spricht dann vom Steuerflussgraphen[5]. Die Knoten des CFG sind die Berechnungen, die Kanten drücken die Abarbeitungsreihenfolge der Berechnungen aus. Der CFG hat zwei ausgezeichnete Knoten *start* (Quelle) und *end* (Senke), wobei die Abarbeitung des Programms bei *start* beginnt und bei *end* endet.

[1] Imperativ steht in Klammern, da es noch andere Paradigmen wie z.B. logisches und funktionales Programmieren gibt, hier aber nur Aspekte des Übersetzerbaus für imperative Sprachen besprochen werden

[2] engl.: Data Flow Graph (DFG)

[3] Alle Kanten zeigen in die entgegengesetzte Richtung

[4] engl.: Data Dependence Graph (DDG)

[5] engl.: Control Flow Graph (CFG)

```
x = a + 2;
y = a - b;
z = x * c;
return z + y;
```

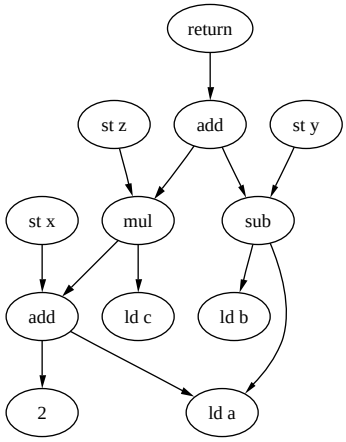


Abbildung 3.1: Datenabhängigkeitsgraph

Man fasst zusammenhängende Knoten des CFGs, die jeweils nur einen Nachfolger und eine Vorgänger haben, zu einem *Grundblock* zusammen. Da in einem Grundblock per definitionem keine Berechnung über das Ausführen einer anderen entscheidet, wird die Berechnungsreihenfolge innerhalb des Grundblocks nur durch die Datenabhängigkeiten bestimmt.

> Abbildung 3.3 zeigt einen Steuerfluss/Datenabhängigkeitsgraphen, wie ihn die Bibliothek FIRM erzeugt. Deutlich zu sehen sind die ausgezeichneten Knoten des CFG (*start* und *end*), die blau gefärbt sind und die Nummern 288 und 262 tragen. Die großen gelben Flächen, die andere Knoten umrahmen, sind Grundblöcke. Die roten Kanten sind die Steuerflusskanten (sie zeigen hier in die verkehrte Richtung). Die schwarzen Kanten sind Datenabhängigkeitskanten. Wenn eine Kante gestrichelt ist, so stellt sie eine Rückwärtskante dar, d.h. sie schließt einen Zyklus.

3.2 Darstellung

Essentiell für die Effizienz der Zwischendarstellung ist ihre Darstellung, also die Art und Weise, wie das Quellprogramm im Übersetzer dargestellt wird. Dies ist eine Gratwanderung, denn die Zwischendarstellung sollte einerseits nicht zu viele Informationen „verschenken“, da sie bei der Optimierung benötigt werden könnten, andererseits sollte sie das Quellprogramm in eine Form bringen, dass die Erzeugung effizienten Codes möglich macht. Folgendes Beispiel mag diesen Konflikt verdeutlichen:

> Fast jede imperative Programmiersprache besitzt Reihungen[6] in ihren Sprachumfang. Angenommen, in einem Programm werden Reihungen für die Implementierung von Vektoren benutzt und einige Methoden für das Rechnen mit Vektoren implementiert. Etwa eine Methode, die zwei Vektoren addiert:
>
> ```
> void add(float x[], float y[], float z[])
> {
>   int i;
>   for(i = 0; i < length(x); i++)
>     z[i] = x[i] + y[i];
> }
> ```
>
> Moderne (RISC) Mikroprozessoren besitzen aber keinen Befehl, der einen Reihungszugriff implementiert. Insbesondere dann nicht, wenn Reihungen in der Programmiersprache als Klassen

[6] engl.: Arrays

dargestellt sind (wie z.B. in Java). Der Reihungszugriff muss also in einen Speicherzugriff umgesetzt werden, dessen Adresse vorher berechnet werden muss (`data` stellt die *Relativadresse* [7] der eigentlichen Reihung in der Reihungsklasse dar. `x` ist ein Zeiger auf ein Objekt der Reihungsklasse):

```
add  t1 = x, data
shl  t2 = i, 2
add  t3 = t1, t2
ld   t1 = [t3]
```

Das ist aber nicht effizient, da der erste Additionsbefehl von der Schleifenvariable `i` völlig unabhängig ist und in jedem Schleifendurchlauf berechnet wird, was völlig unnötig ist.

In obigem Beispiel kam vielleicht eine Zwischensprache zum Einsatz, die die Adressarithmetik, die das Laden von `x[i]` steuert, nicht in die Optimierungen miteinbezogen hat. Es wäre nötig gewesen, den Reihungszugriff frühzeitig in die Adressarithmetik umzusetzen und einige Optimierungen zu starten, welche die oben beschriebene Berechnung des Offsets der eigentlichen Reihung im Objekt `x` vor die Schleife zieht.

Andererseits kann man argumentieren, dass es vielleicht eine Optimierung gibt, die Reihungszugriffe optimieren soll, zum Beispiel so, dass Zugriffe auf aufeinanderfolgende Reihungselemente in SIMD Konstrukte umsetzt werden. Da ist es wichtig, dass der Reihungszugriff noch nicht in Adressarithmetik umgesetzt ist.

Man sieht, dass unterschiedliche Aspekte der Optimierung unterschiedliche Darstellungen erfordern. Deswegen lässt man die Zwischendarstellung oft unterschiedliche Phasen durchlaufen (so auch Firm [TLB99]).

3.2.1 Lineare Darstellungen

In einfacheren Übersetzern geschieht die Befehlserzeugung oft direkt aus dem AST[8], indem ein Postfix-Lauf über den AST gestartet wird und bei jedem Knoten einige Befehle erzeugt werden. Das ist natürlich völlig ineffizient, da man immer nur den Knoten im Baum betrachten kann, an dem man sich befindet.

Einfache Zwischendarstellungen erzeugen in diesem AST-Lauf nicht direkt Maschinensprachbefehle, sondern eine Art architekturunabhängige Maschinensprache (den sogenannten Tripelcode), der das Programm dann sehr maschinennah repräsentiert. Alle Hochsprachenelemente, wie zum Beispiel Zugriff auf Verbundfelder, Reihungszugriffe und das polymorphe Aufrufen von Methoden sind schon komplett in Adressarithmetik umgesetzt. So würde z.B. ein Zugriff auf ein Feld in einem C-Struct so dargestellt:

```
                                mov  t1 = a
                                ld   t2 = [t1]
                                add  t3 = t1, 4
struct { int x, y; } a;         ld   t4 = [t3]
int i, j;                       add  t5 = t4, t2
i = (a.x + a.y) / 2;            div  t6 = t5, 2
j = i + 1;                      st   i = t6
                                ld   t7 = i
                                add  t8 = t7, 1
                                st   j = t8
```

Diese Darstellung ist wenig geeignet um effiziente Befehlssequenzen zu erzeugen, da die Befehle in einer gewissen Weise schon angeordnet sind. Pipelineeffekte und Registerzuteilung (siehe Kapitel 5) sind daher schlecht zu berücksichtigen.

[7] engl.: Offset
[8] Prominente Vertreter sind: Java, MSIL von .NET und Pascal-P-Code

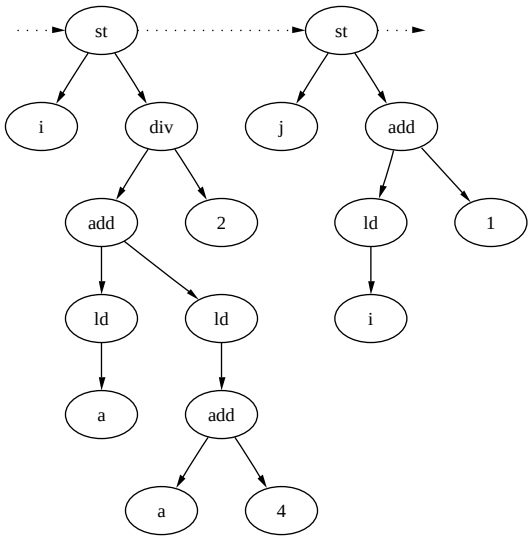


Abbildung 3.2: Baumdarstellung

3.2.2 Baum-Darstellungen

Eine Verfeinerung der obigen Tripelcodedarstellung sind die Baum-Darstellungen. Sie benutzen keine Pseudo-Maschinensprachbefehle, sondern stellen das Programm als Sequenz von Ausdrucksbäumen dar. Dies stellt schon eine Verbesserung gegenüber den linearen Darstellungen dar, da zumindest die Ausdrücke nicht als Sequenz hingeschrieben sind, sondern als Baum dargestellt sind. Jedoch sind die einzelnen Ausdrücke immer noch sequenziell aneinander gereiht. Abbildung 3.2 zeigt obiges Beispiel in einer Baumdarstellung.

3.2.3 Moderne Zwischendarstellungen

Es hat sich gezeigt, dass man für besonders erfolgreiches Optimieren sehr viele Informationen des Quellprogramms beibehalten muss. Zum Beispiel sind Typinformationen von entscheidender Bedeutung, wenn es um die Optimierung objektorientierter Programme geht. Solche Informationen sind in einer Tripelcodedarstellung natürlich nicht mehr vorhanden, da z.B. die Felder eines Verbundes schon in Offsets umgerechnet sind. In Verbunden nicht benutzte Felder kann man dann nicht mehr „wegoptimieren“, da ja die Information, dass es sich um einen Verbund gehandelt hat, nicht mehr explizit verfügbar ist.

Darüber hinaus ist die Tripelcodedarstellung zu „eindimensional“, da diese Form der Zwischenrepräsentation schon eine gewisse Auswahl und Anordnung der letztlich erzeugten Maschinensprachbefehle suggeriert. Man geht deswegen heute dazu über, in der Zwischendarstellung nur noch Datenabhängigkeiten darzustellen. Zwangsläufig endet man dann bei einer graphbasierten Zwischendarstellung. Der Graph stellt dann die „Wert-hängt-von-einem-anderen-Wert-ab“-Relation dar. Diese Darstellung lässt die Anordnung der später erzeugten Befehle völlig offen und ist wie in Abbildung 3.1 ersichtlich, nicht mehr intuitiv mit einer Maschinensprache vergleichbar. Neben den vielen Vorteilen einer solchen Darstellung (wie z.B. das erheblich einfachere Implementieren vieler Optimierungen) ist sie auch ästhetisch der Tripelcodedarstellung vorzuziehen, weil sie nur die für die richtige Übersetzung des Programms notwendigen Informationen klar deutlich macht.

Der Steuerfluss wird in solchen Darstellungen auch als Graph dargestellt. Die gebräuchlichste Form ist, jedem Datenflussknoten einen Grundblock zuzuordnen und die Grundblöcke untereinander durch Steuerflusskanten zu verbinden.

3.2.4 SSA

Eine Eigenschaft für Zwischendarstellungen ist in den letzten 15 Jahren sehr populär geworden: SSA[9] [CFR+91]. SSA ist *keine* Zwischendarstellung, sondern eine Eigenschaft, dem eine Zwischendarstellung genügen kann. SSA bedeutet nämlich, dass jede Variable im ganzen Programm nur eine Zuweisung besitzt. Das hat den großen Vorteil, dass man an jeder Stelle klar weiß, mit welchem Wert die Variable belegt ist. Ohne SSA ist diese Information nur implizit durch den Steuerfluss gegeben. Man denke zum Beispiel an eine Variable, welcher in einem `then`- und einem `else`-Zweig einer `if`-Anweisung ein unterschiedlicher Wert zugewiesen wird, wie das Beispiel weiter unten zeigt.

Natürlich kann nicht jedes Programm die SSA-Bedingung erfüllen. Folgendes Beispiel demonstriert dies:

```
...
if(a > 0)
  x = a;
else
  x = 0;

y = f(x);
```

`x` wird einmal im `if`-Teil und einmal im `else`-Teil ein anderer Wert zugewiesen, also besitzt `x` zwei Zuweisungen, und das Programm ist nicht in SSA-Form. Es ist auch nicht klar, welchen Wert x nach dem `if`-Block hat. Hat es den Wert `a`, oder den Wert `0`?

Ein erster Schritt in Richtung SSA-Form ist es, bei jeder Zuweisung an eine Variable, eine neue einzuführen, um sicher zu stellen, dass jede Variable nur eine Zuweisung besitzt. Dann sähe das obige Beispielprogramm so aus:

```
...
if(a > 0)
  x1 = a;
else
  x2 = 0;
```

Jetzt ist noch offen, wie die Zeile `y = f(x)` hingeschrieben werden kann. Denn es gibt zwei Variablen für `x`: `x1` und `x2`. Dafür wird in der SSA-Darstellung die ϕ-Funktion eingeführt. Die ϕ-Funktion ist zunächst ein ganz normaler Operator, dessen Stelligkeit der Anzahl der Steuerflussvorgänger des Blockes entspricht, in der er sich befindet. Der Wert der ϕ-Funktion hängt von der Steuerflusskante ab, über die der Grundblock erreicht wurde: Wurde der Grundblock über die n-te Kante betreten, so ist der Wert der ϕ-Funktion der des n-ten Arguments. Somit kann man obiges Beispiel vollständig in SSA-Form schreiben:

```
...
if(a > 0)
  x1 = a;
else
  x2 = 0;

y = φ(x1, x2);
```

3.3 Zusammenfassung

Unterschiedliche Arten und Eigenschaften von Zwischendarstellungen wurden vorgestellt. Da ein Bild oft mehr sagt als tausend Worte, wollen wir an dieser Stelle ein C-Programm bringen, dass von einem Frontend in eine moderne, graphbasierte Zwischendarstellung und in eine lineare Darstellung transformiert wurde.

Wie man sieht berechnet folgendes Programm den ggT zweier Zahlen.

[9] Static Single Assignment

```
int gcd(int a, int b) {
  while(a != b) {
      if(a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  return a;
}
```

Ein Übersetzer mit linearer Zwischendarstellung hätte wohl etwas folgender Art erstellt:

```
    arg    t0 = 0
    arg    t1 = 1
L1: sub    t2 = t0, t1
    be     t2, L2
    cmp    t3 = t0, t1
    ble    t3, L3
    sub    t0 = t0, t1
    jmp    L4
L3: sub    t1 = t1, t0
    jmp    L4
L4: jmp    L1
L2: ret    t0
```

Hier sieht man ganz deutlich, wie der Steuerfluss in Sprungmarken und Sprunganweisungen ausgedrückt ist. Diese Darstellung ist sehr umständlich, da man die Grundblöcke erst noch finden muss. Eine Information, die im AST implizit enthalten ist, geht hier verloren.

Desweiteren hat man den Eindruck, dass schon Maschinensprache entsteht. In Wahrheit imitiert die Zwischensprache des Übersetzers nur eine bevorzugte Zielarchitektur. Das Backend wird später viel Aufwand treiben müssen, um die durch den AST suggerierte Anordnung der Befehle in der Zwischendarstellung aufzulösen, und eine eigene effiziente Anordnung zu finden. Es ist eigentlich nicht einzusehen, so viele Informationen zu verwerfen und sie später wieder aufzubauen.

Ein modernerer Übersetzer hätte einen Abhängigkeitsgraphen wie in Abbildung 3.3 erstellt. (Das Bild stammt aus der am IPD Goos entwickelten Zwischenrepräsentation Firm [TLB99] [Lin02], angeschlossen an das C-Frontend des Gcc [FSF].)

Die „kleinen“ Knoten mit Namen wie etwa: `Start`, `CmpT`, `PhiI`, ... stellen die Datenflussknoten dar. Sie sind in „Rechtecke“ eingeschlossen (mit den Nummern 287, 285, 267, 271, 280, 269, 263, 265). Diese „Rechtecke“ sind die Grundblöcke. Durch die `Cond`- und `Jmp`-Knoten wird der Steuerfluss verzweigt.

Schön sieht man auch die SSA-Eigenschaft von Firm. Im Schleifenkopf (Block mit der Nummer 267), werden `a` und `b` verglichen, da dieser Block aber mehrere Steuerflussvorgänger hat (nämlich 285, 280 und 269; sie entsprechen dem Block vor dem Schleifenkopf, dem `then`- und `else`-Zweig), und `a` und `b` in diesen Blöcken beschrieben werden, werden hier ϕ-Knoten eingesetzt.

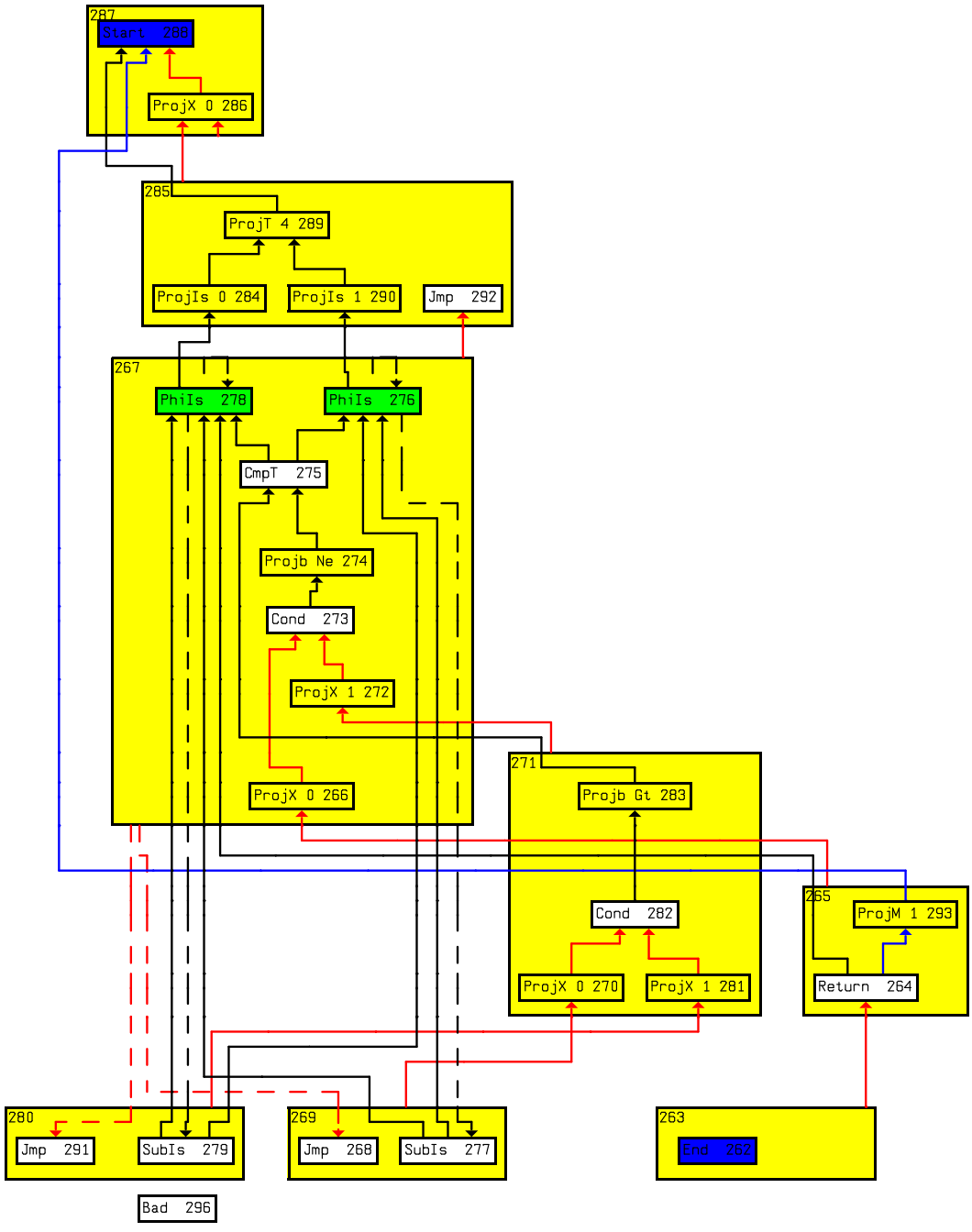


Abbildung 3.3: Das ggT-Programm in der Zwischendarstellung Firm

KAPITEL 4

OPTIMIERUNGEN

Optimierung ist eigentlich ein falsch gewählter Begriff, denn viele Probleme im Übersetzerbau sind NP-vollständig, oft sogar nicht berechenbar. Insofern kann von einer Optimierung (also das Finden des besten aller möglichen Zielprogramme) keine Rede sein. Es wird allenfalls versucht, die Qualität des Zielprogramms hinsichtlich bestimmter Kriterien (z.B. Laufzeit, Speicherverbrauch, Codegröße, oder auch Stromverbrauch des Prozessors) zu verbessern. Folgender Gedanke verdeutlicht diese Schwierigkeit:

> Gegeben sei ein Programm, das endlos läuft und dabei keine Ausgabe fabriziert. Die optimale Übersetzung hinsichtlich des Platzbedarfs wäre etwas in der Art (formuliert in einer Maschinensprache):
>
> ```
> L1: jmp    L1
> ```
>
> Der Übersetzer kann i.A. aber nicht wissen, dass das Programm nicht terminiert, da das Halteproblem unentscheidbar ist, somit kann er diese Übersetzung i.A. nicht finden.

Im folgenden sollen ein paar Optimierungen vorgestellt werden, die in keinem Übersetzer fehlen sollten. Manche Optimierungen finden schon auf der Zwischensprachen-Ebene statt, da sie hier am leichtesten zu implementieren sind. Andere Optimierungen werden auf dem vom Backend erzeugten Code eingesetzt.

Man muss bedenken, dass durch das Auflösen von Typen der Hochsprache (siehe Lowering, Abschnitt 5.1) sehr viel Adressarithmetik entstehen kann, was bedeutet, dass letztlich wesentlich mehr Code in der Zwischendarstellung repräsentiert wird, als dem Programmierer bewusst ist. Zum Beispiel wird der Zugriff auf eine zweidimensionale Reihung vom Übersetzer in drei Multiplikationen und eine Addition umgesetzt. Deswegen sind einige Optimierungen, die zunächst ziemlich überflüssig wirken, unabdingbar.

Die im folgenden vorgestellten Optimierungen kommen meist auf der Zwischendarstellung zum Einsatz. Der Einfachheit halber verzichten wir auf die Erläuterung der Mechanismen der Optimierungen und schildern nur deren Effekt. Für genauere Beschreibungen verweisen wir auf die jeweils angegebene Literatur.

4.1 Konstantenfaltung und algebraische Vereinfachung

Hier werden Konstanten soweit wie möglich zusammen gefasst, denn anstelle

```
a = 4 * 4 * 1 + 4 * j
```

kann man auch

```
a = 16 + 4 * j
```

setzen. Natürlich würde ein Programmierer so etwas eher selten schreiben, aber der Übersetzer wird dies aus folgendem C-Fragment erzeugen, um das Offset des Elements in der Reihung `arr` zu berechnen:

```
int j;
int arr[4][4];
...
x = arr[1][j];
```

Außerdem werden durch diese Optimierung die für die Grundrechenarten gültigen Umformungsgesetze wie das Assoziativgesetz und das Distributivgesetz verwendet, um so viele Konstanten wie möglich zusammen zu fassen.

Desweiteren können „teure“ Befehle, wie Multiplikation und Division, in billigere, wie Addition, Subtraktion und Bitshifts umgesetzt werden[1]. Zum Beispiel wird man

```
a = 16 + 4 * j
```

immer durch

```
a = 16 + j << 2
```

darstellen, da die Shift-Left Operation von Mikroprozessoren meist schneller ausgeführt werden kann, als die Multiplikation.

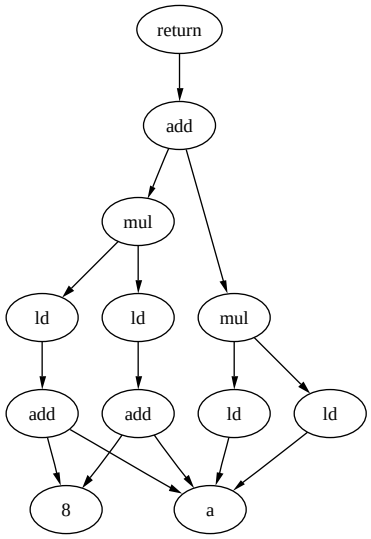
> Oft rechnet es sich auch, die Multiplikation einer Variablen mit Zahlen der Form $2^n + 1$ in Shifts und Additionen umzusetzen, da viele Prozessoren Befehle der Bauart
>
> ```
> (x << c) + y
> ```
>
> haben, wobei c eine Konstante ist.
>
> Mit der algebraischen Vereinfachung muss man behutsam umgehen, da zum Beispiel die Addition auf Gleitkommazahlen *nicht* assoziativ ist, wegen der Rundungsfehler, die entstehen können. Eine gute Einführung in die Gleitkomma-Arithmetik bietet [Gol91].

4.2 Eliminierung gemeinsamer Teilausdrücke

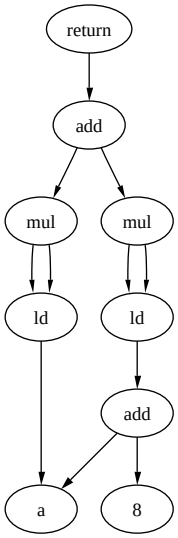
Häufig finden sich in Programmen Ausdrücke, die immer wieder aufs Neue berechnet werden. Auch hier gilt, dass der Programmierer wohl weniger dafür verantwortlich ist, obwohl man `a-b` auch öfters verwendet, um nicht eine dritte Variable `c=a-b` einzuführen. Die Eliminierung gemeinsamer Teilausdrücke[2] leistet genau dies. Man unterscheidet die CSE, die nur auf Grundblockbasis operiert von der GCSE[3], die ganze Funktionen berücksichtigt.

> Angenommen, man schreibt ein Programm, dass mit geometrischen Figuren im zweidimensionalen Raum arbeitet. Dann wird man sich wahrscheinlich einen Verbund der Form
>
> ```
> typedef struct {
>   double x, y;
> } vec_t;
> ```
>
> definieren. Man wird sicher auch eine Funktion, die den Betrag eines solchen Vektors berechnet, schreiben.
>
> ```
> double norm(vec_t *a) {
>   return a->x * a->x + a->y * a->y;
> }
> ```
>
> Das Lowering (siehe Abschnitt 5.1) wird nun die Mitglieder des Verbunds (`x` und `y`) in die Offsets 0 und 8 umrechnen. Danach wird Adressarithmetik eingefügt. Der DDG (siehe Abschnitt 3.1) für die Funktion `norm` sieht dann so aus:

[1] Diese Optimierung wird auch oft „strength reduction“ genannt
[2] engl.: Common Subexpression Elimination (CSE)
[3] Global Common Subexpression Elimination



Hier sieht man ganz deutlich, dass die Berechnung der Adresse `a+8` zweimal vorhanden ist. Auch die Ladebefehle sind jeweils doppelt vorhanden, was unnötig ist. Der durch CSE optimierte DDG hätte folgendes Aussehen:



4.3 Eliminierung partieller Redundanzen

Die Eliminierung partieller Redundanzen[4] [MR79] ist eine Art "Super"-Optimierung, deren Effekte die Eliminierung gemeinsamer Teilausdrücke, das Herausziehen von schleifenunabhängigem Code[5] sowie einige weitere Verbesserungen des Codes, umfassen. Eine partielle Redundanz ist hierbei eine Berechnung, die in mehreren Pfaden durch den Steuerflussgraphen (aber nicht notwendigerweise allen) enthalten ist. PRE erreicht durch geschicktes Platzieren (genauer das Einfügen und Löschen) von Berechnungen, dass nach der Optimierung auf jedem Pfad durch den Steuerflussgraphen nicht mehr — im allgemeinen sogar weniger — solche Berechnungen liegen.

Den Unterschied zwischen GCSE und PRE illustriert Abbildung 4.1 am besten.

[4] engl.: Partial Redundancy Elimination (PRE)
[5] engl.: loop invariant code motion

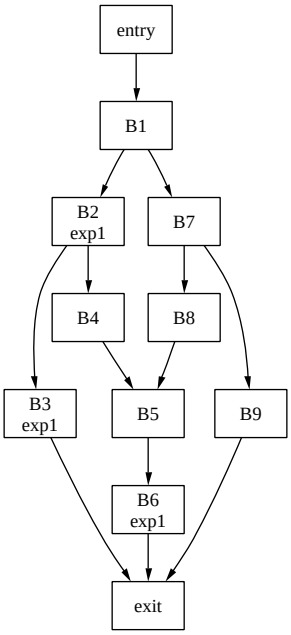


Abbildung 4.1: Beispiel für PRE

In diesem Beispiel wird die Berechnung `exp1` in den Blöcken B2, B3 und B6 durchgeführt. Wir nehmen an, dass der verbleibende Code keinen Effekt auf die Auswertung `exp1` hat. Die klassische GCSE würde nur die Berechnung von exp1 in B3 eliminieren, da B6 über (entry, B1, B7, B8, B5, B6) erreicht werden kann ohne exp1 vorher zu berechnen. PRE würde die Berechnung von exp1 in den Blöcken B3 und B6 eliminieren. Um die Berechnung von B6 zu entfernen zu können, würde PRE die Berechnung von exp1 in B8 einfügen.

Obschon PRE eine mächtige Optimierung ist, tendiert sie dazu, die Größe des Codes zu erhöhen, um seine Laufzeit zu verringern. Darum sollte GCSE eingesetzt werden, wenn man die Codegröße optimieren möchte.

4.4 Schleifenoptimierungen

In vielen Fällen sind die Rümpfe von Schleifen ziemlich klein. Da aber der Rücksprungbefehl am Ende der Schleife in modernen RISC-Prozessoren häufig ein Leerlaufen der Pipeline verursacht, ist man interessiert, mehr Befehle pro Schleifendurchlauf auszuführen.

Zum Beispiel wäre die Addition zweier Vektoren

```
for(i = 0; i < n; i++)
  a[i] = b[i] + c[i];
```

Für einen RISC-Prozessor wäre folgende Implementierung sicher besser:

```
for(i = 0; i < n >> 2; i+=4) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}

for(j = 0; j < n & 3; j++, i++)
  a[i] = b[i] + c[i];
```

Schleifenoptimierungen versuchen, unter Berücksichtigung der Zielarchitektur (Pipeline Penalties bei Sprüngen, Größe des Befehls-Cache) Schleifen umzuformen. Dazu gehört das oben angedeutete Schleifenausrollen, aber auch andere Transformationen, wie zum Beispiel das Teilen von Schleifen in zwei. Einen guten Überblick über Schleifenoptimierungen findet man in [BGS94].

4.5 Inlining von Funktionen

Viele Funktionen in einem Programm sind nicht besonders lang. Unser Beispiel für den Vektorbetrag aus Abschnitt 4.2 ist nur eine Zeile lang und umfasst wenig Befehle.

Dies würde noch kein Problem darstellen, wenn der Funktionsaufruf nicht so teuer wäre. Ein Funktionsaufruf birgt nämlich einige Leistungseinbußen in sich:

- Je nach Prozessor müssen Register für die Parameter frei gemacht werden, deren Inhalt in den Speicher geschrieben werden muss (nach der Rückkehr von der Funktion müssen diese Register wieder geladen werden, was vielleicht Cache Misses zur Folge hat).
- Auf anderen Prozessoren müssen die Parameter auf dem Stack übergeben werden, was vielleicht kostbare Cachezeilen kostet.
- Dann muss die Funktion ja noch gerufen werden, was ggf. einen Instruction-Cache Miss und sicherlich einen Pipeline-Stall zur Folge hat. Selbiges gilt natürlich auch bei der Rückkehr zum Rufer.

Gerade in objektorientierten Sprachen gibt es viele Funktionen (z.B. setter und getter), deren Rümpfe kürzer als zehn Zeilen sind. In Anbetracht obiger Nachteile beim Rufen einer Funktion drängt sich der Gedanke auf, den Rumpf der Funktion direkt an der Aufrufstelle einzusetzen. Dadurch wird der Code vielleicht ein wenig länger, aber auf jeden Fall schneller. Dies wird durch das *Inlining* bewerkstelligt. Problematisch ist natürlich die Grenze zu finden, wann Inlining noch lohnt und wann nicht.

4.6 Peephole-Optimierung

Peephole-Optimierung [McK65] ist eine Art Nachverarbeitung des erzeugten Codes. Sie geht mit einem kleinen Fenster (zwei oder drei Befehle) über den erzeugten Code und versucht unnötige Redundanzen im Code zu entfernen. Die meisten Effekte können mittlerweile schon von modernen Codeerzeugern (siehe Abschnitt 5.3.4 und 5.3.3) erreicht werden, jedoch kostet die Peephole-Optimierung kaum Zeit und ist somit immer gut, um dem erzeugten Code den letzten Schliff zu geben. Folgendes Beispiel zeigt einige suboptimale Codesequenzen und deren Ersetzungen.

Code	Ersatz
`jmp L1` `L1:`	
`mov r1 = r1`	
`mul r1 = r2, 2`	`shl r1 = r2, 1`

4.7 Zusammenfassung

Dies ist nur ein kleiner Auszug aus den existierenden Optimierungen, jedoch sind CSE, PRE und Inlinig absoluter Standard und in jedem ernsthaften Übersetzer implementiert, da sie einen großen Effekt auf die Laufzeit des Programms haben.

Da jede Optimierung das Programm transformiert, werden vielleicht manche Optimierungen erst durch die Ergebnisse anderer möglich oder gar *unmöglich*. Es gibt kein Patentrezept, welche Optimierungen wann und in welcher Reihenfolge durchgeführt werden sollen. Oft werden bestimmte Optimierungen (z.B. CSE) auch mehrmals gestartet (nach anderen Optimierungen). Eine kleine Sammlung von Optimierungen findet sich unter

`http://www.nullstone.com/htmls/category/address.htm`

KAPITEL 5

DAS BACKEND

Das Backend erfüllt mehrere Aufgaben. Die wohl Wichtigste ist das Erzeugen von Code, der auf einem Prozessor ausgeführt werden kann. Darüberhinaus muss das Backend (falls noch nicht geschehen), alle Datenstrukturen in eine „flache“ Darstellung bringen, so dass die einzelnen Elemente direkt mit Adressen korrespondieren. Dies leistet das sogenannte *Lowering*.

5.1 Lowering

In einfacheren Zwischensprachen wird das Lowering häufig vorweg genommen. Es hat sich aber gezeigt, dass bestimmte Informationen, die dadurch vernichtet werden (z.B. das Speicherlayout von Klassen) durchaus relevant für bestimmte Optimierungen sind. Da der Codeerzeuger aber nichts von Klassen, Verbänden und Reihungen versteht, sondern nur etwas von Funktionen, Speicherstellen, Offsets und Werten, wird vor der eigentlichen Codeerzeugung das Lowering durchgeführt. Es legt unter anderem die Abbildung der zusammengesetzten Datentypen in den Speicher fest. Das Lowering kann in die Zwischendarstellung integriert werden und somit von den Optimierungen der Zwischendarstellung profitieren. Folgende Informationen sind für das Lowering von Bedeutung:

Offset
: Die Adresse eines Mitglieds innerhalb eines zusammengesetzen Datentyps.

Alignment
: An welchen Adressen muss der Datentyp ausgerichtet sein? Viele Prozessoren fordern, dass ein Wort mit n Bytes im Speicher an einer Adresse steht, die durch n teilbar ist. Manche Prozessoren unterstützen auch den Zugriff auf nicht ausgerichtete Worte; jedoch bringt eine Ausrichtung meist eine erhebliche Geschwindigkeitssteigerung beim Zugriff.

Virtuelle Methodentabellen
: Prozessoren bieten im Allgemeinen keine Unterstützung für polymorphe Methodenaufrufe. Deswegen muss das Lowering diese Aufrufe in Konstrukte, die Prozessoren bekannt sind, umformen.

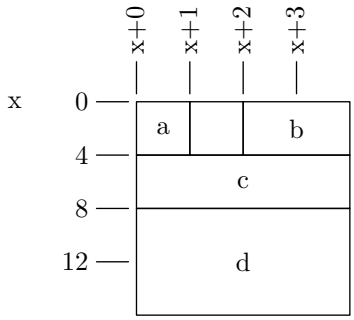
Folgendes Beispiel zeigt das Lowering eines Structs in C.

```
struct a_struct {
  char a;
  short b;
  int c;
  double d;
};
```

Angenommen, es gelte:

Typ	Größe in Bytes	Ausrichtung
`char`	1	1
`short`	2	2
`int`	4	4
`double`	8	8

so wäre das Speicher-Layoutdes Typs `a_struct`:



Der Verbund müsste desweiteren an einer acht-Byte Adresse ausgerichtet werden, da die Ausrichtung von `double` an 8 Byte Grenzen geschieht.

5.2 Codeerzeugung

Die Codeerzeugung erstellt aus der Zwischensprache eine Sequenz von Maschinenbefehlen. Die Schwierigkeit ist nicht das Finden einer gültigen Sequenz (eine Sequenz, die semantisch dem in der Quellsprache formulierten Programm entspricht), sondern eher das Finden einer „guten“ Sequenz bezüglich der erwähnten Optimierungskritierien (Geschwindigkeit, Speicherverbrauch, etc.). Meistens liegt der Fokus auf dem Finden einer möglichst schnellen Sequenz.

Die Codeerzeugung besteht im wesentlichen aus der Auswahl der geeigneten Befehle für die Einheiten der Zwischendarstellung, der Anordnung dieser Befehle, und der Zuteilung von verfügbaren Prozessorregistern für die in der Zwischendarstellung auftretenden Werte. Das Hauptproblem ist, dass sich alle drei Phasen gegenseitig beeinflussen[1]. So hat zum Beispiel das Umordnen von Befehlen einen Einfluss auf die zugewiesenen Register, da unter Umständen mehr Register gebraucht werden könnten.

5.2.1 Befehlsanordnung

Gerade bei modernen RISC-Architekturen hängt die Ausführungsgeschwindigkeit einer Sequenz ganz entscheidend von der Anordnung der Befehle ab. RISC-Prozessoren besitzen Pipelines, die die Bearbeitungsschritte eines Befehls durchführen, z. B.

1. Dekodieren
2. Operanden holen
3. Befehl ausführen
4. Wert zurückschreiben

In einem Taktzyklus wird *nicht* ein Befehl ausgeführt, sondern eine Stufe der Pipeline

Angenommen, der Befehl A wird zum Zeitpunkt t_A in die Pipeline eingefügt[2] und stellt sein Ergebnis nach r_A Takten zur Verfügung. Dann ist das Ergebnis zum Zeitpunkt $t_A + r_A$ verfügbar. Nutzt nun ein zweiter Befehl B (in die Pipeline eingefügt zum Zeitpunkt t_B) das Ergebnis von A, so sollte der Zeitpunkt, an der B seine Operanden liest größer als $t_A + r_A$ sein, sonst fügt der Prozessor Wartezyklen ein, was Zeit kostet.

Hier kommt der Übersetzer ins Spiel. Im Idealfall hat er Kenntnis von den Pipeline-Belegungen aller Befehle und aller Bypässe, die existieren. Er versucht eine möglichst optimale Anordnung zu finden, so dass der Prozessor keine Leertakte einfügen muss (siehe auch [Mül95]).

[1] Man spricht auch vom Phasenproblem des Backends
[2] engl.: Instruction issue

5.2.2 Registerzuteilung

In der Zwischendarstellung wird angenommen, dass unendlich viele Speicherplätze für die Werte des Programms zur Verfügung stehen. Für die Ausführungsgeschwindigkeit des Programms ist es aber von entscheidender Bedeutung, dass möglichst viele Werte des Programms in den Registern des Prozessors gehalten werden, denn ein Speicherzugriff ist unter Umständen um ein Vielfaches langsamer als ein Zugriff auf ein Register. Register gibt es aber nur in beschränkter Anzahl. Die Aufgabe des Registerzuteilers ist es, zu versuchen, allen Werten des Programms Register zuzuweisen. Falls das nicht möglich ist, weil keine Register mehr frei sind, muss er dafür Sorge tragen, dass die Werte in den Hauptspeicher ausgelagert werden[3].

Die entscheidende Information für die Registerzuteilung ist die Lebendigkeit einer Variablen. Wird einer Variable das erste mal ein neuer Wert zugewiesen, so wird sie „lebendig“ (bei SSA ist das per definitionem auch das einzige Mal). Nach der letzten Benutzung der variable erlischt ihre Lebenszeit, da ihr Wert ja nicht mehr benötigt wird.

Folgendes Programmfragment zeigt eine Sequenz von Befehlen und die an der jeweiligen Stelle lebendigen Variablen.

Zeitpunkt	Befehl	Lebendige Werte
1	`mov t0 = 5`	`t0`
2	`add t1 = t0, 9`	`t0 t1`
3	`add t2 = t0, t1`	`t0 t1 t2`
4	`sub t3 = t0, t1`	`t2 t3`
5	`add t4 = t2, 6`	`t3 t4`
6	`sub t5 = t3, t4`	`t5`

Man sieht, dass man für eine korrekte Registerzuteilung 3 Register benötigt, da an Stelle 3 der Sequenz die Werte a, d, e gleichzeitig lebendig sind und nicht im selben Register gehalten werden können.

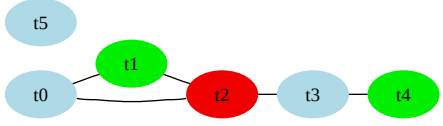
Variablen, die zur selben Zeit lebendig sind, deren Lebenszeiten sich also überlappen, können logischerweise nicht im selben Register gehalten werden, sonst würden sie sich überschreiben. Der Registerzuteiler weist jeder Lebenszeit ein Register zu. Falls er nicht genügend Register hat, um jeder Lebenszeit ein Register zuzuweisen, muss er manche Werte in den Hauptspeicher auslagern und somit deren Lebenszeiten in Registern verkürzen.

Die Registerzuteilung ist NP-vollständig und lässt sich sehr leicht auf das Problem des Färbens eines Graphen mit k Farben reduzieren (wobei k die Anzahl der verfügbaren Register ist).

Jeder Lebenszeit entspricht ein Knoten in diesem sogenannten Interferenzgraphen. Überschneiden sich zwei Lebenszeiten, so werden die entsprechenden Knoten durch eine Kante verbunden. Da beim Graphfärben zwei Knoten, die durch eine Kante verbunden sind niemals die selbe Farbe erhalten können, entsprechen die Farben den Registern und der vollständig gefärbte Graph einer Zuteilung von Registern an die einzelnen Lebenszeiten. Zum Lösen dieses Problems existiert eine brauchbare Heuristik, die sich in der Vergangenheit bewährt hat [CAC+81].

Um Missverständnissen vorzubeugen sei noch erwähnt, dass die Komplexität der Registerzuteilung nicht darin besteht, festzustellen, wie viele Register ein Programm benötigt. Vielmehr ist das Zuweisen der Register an die einzelnen Lebenszeiten (was dem Färben des Interferenzgraphen entspricht) schwierig.

Der gefärbte Interferenzgraph zu obigem Codefragment sähe z.B. so aus:



[3] man nennt diese Vorgehen auch spilling

Eine Zuteilung, die von einem graphfärbenden Registerzuteilung gefunden worden wäre, ist z.B.:

Wert	t0	t1	t2	t3	t4	t5
Register	r1	r2	r3	r1	r2	r1

5.2.3 Befehlsauswahl

Die verfügbaren Operatoren einer Programmiersprache können nicht immer 1:1 auf entsprechende Maschinensprachenbefehle abgebildet werden. Zwar haben die meisten in Programmiersprachen vorhandenen Operatoren entsprechende Prozessorbefehle; oft finden sich für bestimmte Operatoren, oder Operatorkombinationen aber auch einfachere Maschinensprache-Sequenzen. Hier einige Beispiele:

Name	Befehl	C-Äquivalent
Multiply-Add	`fma f1 = f2, f3, f4`	`f1 = f2 + f3 * f4`
And-Not	`andn r1 = r2, r3`	`r1 = r2 & ~r3`
Shift-Left-Add	`shladd r1 = r2, r3, c`	`r1 = (r2 << c) + r3`

Es ist nicht gesagt, dass man immer einen Befehl auswählen sollte, der möglichst viele Operatoren der Quellsprache in sich vereinigt, obwohl das zunächst verlockend ist. Man könnte anstelle des Multiply-Add-Befehls beispielsweise auch eine Multiplikation und eine Addition auswählen. Vielleicht würden diese Befehle in anderen Funktionseinheiten des Prozessors ausgeführt werden, oder könnten so angeordnet werden, dass insgesamt eine andere und bessere Anordnung der Befehle gefunden werden kann. Vielleicht lässt sich so auch die Anzahl der benötigten Register senken (Registerdruck).

Man sieht, dass sich alle drei Phasen gegenseitig beeinflussen. Es ist deshalb auf jeden Fall „suboptimal“, die drei Phasen stur hintereinander zu durchlaufen.

5.3 Verfahren der Codeerzeugung

Die hier vorgestellten Verfahren bewältigen vor allem die Befehlsauswahl. Die Befehlsanordnung und die Registerzuteilung laufen als „Optimierung“ danach über die erzeugte Befehlssequenz. Die Integration der Befehlsauswahl, Befehlsanordnung und Registerzuteilung ist der Hauptbestandteil der momentanen Backend-Forschung. In den meisten verfügbaren Übersetzern kommen aber nur klassische Verfahren[4] zum Einsatz, die diese Integration kaum bewerkstelligen können. Desweiteren arbeiten diese Verfahren immer lokal in Grundblöcken, können also Zusammenhänge, die über die Grundblockgrenzen hinausgehen, nicht erkennen.

Ein wichtiger Aspekt des Backends ist der Einsatz von Generatoren, ähnlich wie beim Frontend. Man möchte den gesamten Codegenerator nicht von Hand programmieren, da dies sehr viel Zeit kostet, fehleranfällig ist und die entstandenen Codegeneratoren nur schwer auf andere Zielarchitekturen portierbar sind.

Seit ungefähr 25 Jahren beschäftigt man sich deshalb mit Verfahren der Erzeugung von Codegeneratoren aus Spezifikationssprachen, in denen die Beziehung der Zwischendarstellung zur Zielarchitektur festgehalten wird. Diese Spezifikationen sind wesentlich kleiner als ausprogrammierte Codegeneratoren und deswegen weniger fehleranfällig und leichter wart- und portierbar.

5.3.1 Einfache Formen

Einfache Formen der Codeerzeugung können nur kleine Bereiche der Zwischendarstellung betrachten und somit nur lokal Befehle auswählen. Meistens betrachten sie nur einen Zwischendarstellungsbefehl und wählen für diesen einen konkreten Maschinensprachenbefehl aus. Das ist kaum mehr als als gewöhnliche Makrosubstitution. Spezielle Adressierungsmodi, oder komplexere Befehle, die mehrere Zwischendarstellungsoperationen in einem durchführen könnten, bleiben unberücksichtigt.

[4] Meistens Bottom Up Pattern Matcher

5.3.2 Graham/Glanville-Codegeneratoren

Die Graham/Glanville-Codegenerator-Generatoren [GG78] waren die ersten Werkzeuge, die Codegeneratoren aus einer Spezifikation erzeugen konnten. Sie basieren auf der LR-Zerteilertechnik, die schon bei den Frontends zur syntaktischen Analyse zum Einsatz kam. Zum Einsatz dieser Art von Generatoren benötigt man eine Zwischendarstellung in Baumform, wie in Abschnitt 3.2.2 beschrieben. Der Code wird für jeden Ausdrucksbaum einzeln erzeugt. Zunächst muss man die einzelnen Bäume aber textuell hinschreiben. Dazu bedient man sich der Präfixform, indem man die Knoten des Baumes rekursiv besucht, und vor dem Abstieg zu den Kindern den Wert des besuchten Knoten ausgibt. Der linke Baum aus Abbildung 3.2 sähe z.B. so aus:

```
st i div add ld a ld add a 4 2
```

Klammert man die einzelnen Knoten wird deutlicher, was gemeint ist:

```
(st i (div (add (ld a) (ld (add a 4))) 2))
```

Der Trick ist nun, eine Grammatik anzugeben, mit der man die textuell Darstellung (ohne Klammern) zerteilt. Zu jeder Produktion dieser Grammatik annotiert man eine Sequenz von Maschinensprachebefehlen (meist einen), die ausgegeben werden, wenn die Produktion angewandt wird. Folgendes Beispiel soll das Verfahren verdeutlichen:

Folgende Tabelle ist ein Auszug den Produktionen des LR-Automaten eines Graham-Glanville Codegenerators für den MMIX-Prozessor [Knu99]:

Nummer	Produktion	Maschinensprachebefehl
	...	
1	t1 ⇒ add t2 t3	ADD t1, t2, t3
2	t1 ⇒ add t2 <Zahl>	ADD t1, t2, <Zahl>
3	t1 ⇒ div t2 2	SR t1, t2, 1
4	t1 ⇒ ld t2	LDT t1, t2, 0
5	t1 ⇒ ld add t2 t3	LDT t1, t2, t3
6	t1 ⇒ ld add t2 <Zahl>	LDT t1, t2, <Zahl>
7	ϵ ⇒ st t1 t2	STT t1, t2, 0
	...	

Eine mögliche ausgewählte Codesequenz für obiges Beispiel wäre:

```
LDT   t1, a, 4
LDT   t2, a, 0
ADD   t3, t1, t2
SR    t4, t3, 1
STT   i, t4, 0
```

In obigem Beispiel sieht man, dass die Grammatik nicht eindeutig ist. Das `ld` in Produktion 6 „überdeckt“ auch das im Baum darunterliegende `add`. Es gibt aber auch zwei Regeln (2 und 4), die das `add` und das `ld` getrennt überdecken würden. Um diese Mehrdeutigkeiten (die in diesen Automaten sehr häufig vorkommen) zu beseitigen, muss man die Produktionen mit Kosten versehen, die dann zwischen mehreren möglichen Anwendungen entscheiden. Diese Entscheidung wird *statisch* bei der Erzeugung des Codegenerators getroffen. Nachteile des Verfahrens sind:

- Die Tabellen der generierten LR-Automaten werden i.A. sehr groß.
- Aufgrund des Einsatzes von kontextfreien Sprachen geschieht bei einem Befehl mit zwei Operanden (z.B. `add t1 t2`) die Auswahl der Befehle für den linken Teilbaum (`t1`) unabhängig vom rechten Teilbaum (`t2`), was in einigen Fällen zu suboptimalen Code führen kann.
- Es ist *unentscheidbar*, ob alle möglichen Bäume der Zwischendarstellung von dem Codegenerator zerteilt werden können. Dieses Problem lässt sich auf das Äquivalenzproblem von kontextfreien Grammatiken zurückführen.

5.3.3 Bottom Up Pattern Matching (BUPM)

Bei BUPM [AGT89] kommt keine Zerteilertechnik zum Einsatz, obwohl die Spezifikation für den Codegenerator-Generator der der Graham/Glanville-Generatoren ähnelt. BUPM kann auf Ausdrucksbäumen optimalen Code im Sinne der Spezifikation erzeugen. Es implementiert ein allgemeines Baumersetzungsverfahren, dass Teilbäume finden kann und dann durch einen einzelnen Knoten ersetzt. Die Ersetzung kann mit Kosten, die *dynamisch* (während des Codeerzeugungsprozesses) berechnet werden, gesteuert werden. Eine Regel in diesem Baumersetzungssystem besteht im wesentlichen aus folgenden Bestandteilen:

Ziel
: Ein einzelner Knoten, meist ein Register des Prozessors.

Muster
: Ein Baum, der in dem Ausdrucksbaum gefunden werden soll und dann durch das obige Ziel ersetzt wird.

Kosten
: Eine Funktion, die die Kosten für die Ersetzung berechnet.

Aktion
: Eine Funktion, die ausgeführt wird, wenn die Regel zum Einsatz kam, und das Muster durch das Ziel ersetzt wurde. Hier wird beispielsweise eine Maschinensprach-Sequenz ausgegeben.

Eine weit verbreitete Implementierung dieses Verfahrens mit dem Namen BEG wurde am IPD, Lehrstuhl Goos von Emmelmann entwickelt [ESL89].

5.3.4 Bottom Up Rewrite Systems (BURS)

Das BURS-Verfahren [PLG88] erweitert das BUPM um Termersetzungssysteme. Man kann nun Teilbäume auch in andere Teilbäume transformieren und Variablen verwenden. Zum Beispiel sind nun Regeln der folgenden Art denkbar:

```
add X Y ⇒ add Y X
```

Diese Regel verwendet zwei Variablen und stellt die Kommutativität der Addition dar. Dies ist sehr praktisch, da man nun Regeln, die die eine Addition mit zwei verschiedenen Operanden darstellen, nur einmal hinschreiben muss, und nicht beide Fälle explizit erwähnen muss.

Aus

```
t1 ⇒ add t2 <Zahl>
t1 ⇒ add <Zahl> t2
  t1 ⇒ add t2 1
  t1 ⇒ add 1 t2
```

wird

```
t1 ⇒ add t2 <Zahl>
  t1 ⇒ add t2 1
add X Y ⇒ add Y X
```

Diese zusätzliche Funktionalität birgt natürlich auch höhere Komplexität in sich, was sich in einer längeren Laufzeit während der Codeerzeugung niederschlägt. Deswegen werden Suchverfahren, wie in [NKWA96] beschrieben, unabdingbar. Eine Verallgemeinerung des Verfahren von Ausdrucksbäume auf Abhängigkeitsgraphen wurde von Boris Boesler in CGGG [Boe98] implementiert.

5.4 Assemblierung, Binden und Laden

Am Ende des Übersetzungsprozesses steht meist eine Textdatei mit Maschinensprachebefehlen, Sprungmarken, Adressmarken und Bereichsbeschreibungen. Diese Textdatei wird dann vom sogenannten *Assembler* eingelesen, der daraus eine Binärdatei produziert, die nun vom Betriebssystem (und ggf. dem *Binder*[5]) geladen werden kann. In dieser Binärdatei sind die sogenannten *Mnemonics*, die Kürzel für die Maschinensprachbefehle (z.B. `add` und `sub`) und die Operanden in Zahlenform für den Prozessor kodiert. Auch die globalen Variablen, die nicht in Registern oder auf dem Stack gehalten werden können, sind in dieser Binärdatei repräsentiert, indem ihr initialer Wert dort vermerkt ist.

Es kann natürlich sein, dass ein Programm auf Variablen oder Funktionen einer Bibliothek, die nicht im Quelltext vorliegt, zugreifen will. Der Übersetzer kennt die Implementierung dieser Funktionen nicht, kann also keinen Code dafür erzeugen. Stattdessen markiert er die entsprechenden Stellen in der Assemblerdatei. Nach dem Assemblieren betrachtet der Binder die erzeugte Binärdatei und versucht, die markierten Funktionen und Variablen in den ihm übermittelten Bibliotheken zu finden. Man unterscheidet zwei Arten von Binden:

Statisches Binden
: Am Ende der Übersetzung werden die fehlenden Variablen und Funktionen vom Binder aus der Bibliothek herauskopiert und mit der übersetzten Datei zusammengefügt. Dies hat den Nachteil, dass die resultierenden Binärdateien sehr groß werden, da die meisten Programme großen Gebrauch von Bibliotheken machen. Fast jedes C-Programm nutzt `printf`. Mit dieser Art des Binden fände man den Code von `printf` in jeder ausführbaren Datei dupliziert. Das kostet nur Platten- und Hauptspeicherplatz.

Dynamisches Binden
: Hier wird der Linker erst beim Start des Programms vom Betriebssystem gerufen. Er lädt dann alle benötigten Bibliotheken in den Speicher und trägt die richtigen Adressen für die benötigten Funktionen und Variablen ein.

[5] engl.: Linker

KAPITEL 6

AKTUELLE FORSCHUNG AM IPD

Zur Zeit bauen wir ein offenes optimierendes Übersetzer-System (Crs[1]), das einen Gutteil des hier Vorgestellten und noch weiteres umsetzt. Es basiert auf den Ergebnissen von über 10 Dissertationen am IPD der letzten Jahre. Um alle diese Einzelleistungen zu einen System zu integrieren beschäftigen wir auch verstärkt HiWis.

Neben diesen eigentlichen Übersetzerbau-Aktivitäten liegt ein weiterer Schwerpunkt auf der Anwendung von Theorien und Werkzeugen des Übersetzerbaus, z.B. der XML-Verarbeitung von Dokumenten oder der werkzeugbasierten Softwaretechnik. Näheres über unsere Forschungsaktivitäten findet sich auf unseren Webseiten: `http://www.info.uni-karlsruhe.de`

Nun wollen wir im einzelnen auf die Architektur von Crs eingehen (siehe Abbildung 6.1). Da wie schon gesagt für den Bau von Frontends im Wesentlichen kein weiterer Forschungsbedarf besteht, haben wir exemplarisch zwei Frontends existierender Übersetzer an Crs angeschlossen. Das eine ist Jack, ein Frontend für eine eingeschränkte Version von Java, sowie Gcc-Firm ein C Frontend, das die volle Unterstützung des Gcc-C-Dialekts bieten soll. Die Optimierungsphase und das Backend sind unsere eigentlichen Forschungsinteressen. Die Kerndatenstruktur hierbei ist Firm. Auf ihr arbeiten alle existierenden und geplanten Optimierungen und Transformationen:

Klassische Optimierungen
: Konstantenfaltung, algebraische Vereinfachungen, CSE, PRE und Inlining.

Trapp-Optimierungen
: wurden von Martin Trapp am IPD entwickelt und bewirken interprozedurale Optimierungen insbesondere bei objektorientierten Programmen.

SIMD-/EPIC-Optimierungen
: bestehen aus einer geplanten Sammlung an Erweiterungen von Firm, um Zielarchitekturspezifische Optimierungen für SIMD- und EPIC-Prozessoren zu beherrschen.

Cache Optimierungen
: verbessern das Laufzeitverhalten unter Ausnutzung der Cacheparameter der Zielarchitektur.

Graphtransformationen
: sind ein geplantes Rahmenwerk zur Manipulation von Firm-Graphen mittels Graphersetzungsregeln. Dies soll das Erkennen und Angeben von Mustern im Firm-Graphen vereinfachen, insbesondere beim Erkennen von impliziten SIMD-Konstrukten.

CATE
: möchte unsere Infrastruktur benutzen, um softwaretechnische Analysen durchzuführen und besseres halbautomatisches Refactoring zu erreichen.

Manche Optimierungen arbeiten auf verschiedenen „Ebenen“ der Zwischensprache, die durch das Lowering (siehe Abschnitt 5.1) ineinander überführt werden. Dabei ist generell die Tendenz

[1] engl: Compiler Research System

vorhanden, dass zuerst noch mehr Konzepte der Hochsprache direkt abgebildet werden (z.B. Zugriffe auf Arrays) und in späteren Phasen in eher maschinennahe Konstrukte aufgelöst werden (z.B. Zeigerarithmetik und Ladebefehle). Allerdings muss dieser Prozess nicht monoton sein; so kann es vorkommen, dass erst gewisse Optimierungen z.B. Schleifen erkennen und dies in Firm explizit gemacht wird.

Die Backends unseres Crs sind entweder von Cggg generiert oder handgeschrieben[2]. Wir sind gerade bemüht, für zahlreiche Prozessoren Codegenerator-Spezifikationen für Cggg zu erstellen, um ein vollständiges, von C auf verschiedene Prozessoren übersetzendes Forschungsvehikel zu haben. Eine der Zielarchitekturen ist C, sie dient im wesentlichen für Testzwecke.

Die Integration einer Feedback-Komponente ist geplant. Die Idee dabei ist: Der Codegenerator erzeugt nicht blind Code, sondern teilt seine Probleme der Zwischensprachenschicht mit, um dort ggf. gewisse andere Optimierungstaktiken verfolgen zu können. So ist es z.B. denkbar, dass das Ausrollen einer Schleife den Registerdruck derart erhöht hat, dass kein "guter" Code mehr erzeugt werden kann. Es wäre also klug, eine solche Optimierung zurücknehmen zu können.

Die Korrektheit des Übersetzers und seine Leistung sind bei der Implementierung von neuen Komponenten ständig zu prüfen. Daher wird eine automatische Leistungsmessungs- und Validationseinheit benötigt.

Weiterhin beschäftigt sich das Institut mit der formalen Verifikation von Übersetzern. So wurde zum Beispiel die in Firm implementierte Konstantenfaltung in einer Studienarbeit als korrekt bewiesen [GB03].

[2]Das Handschreiben ist allenfalls für Serialisierer des Firmgraphen (engl.: Dumper) oder Codegeneratoren für kellerbasierte Zwischensprachen sinnvoll

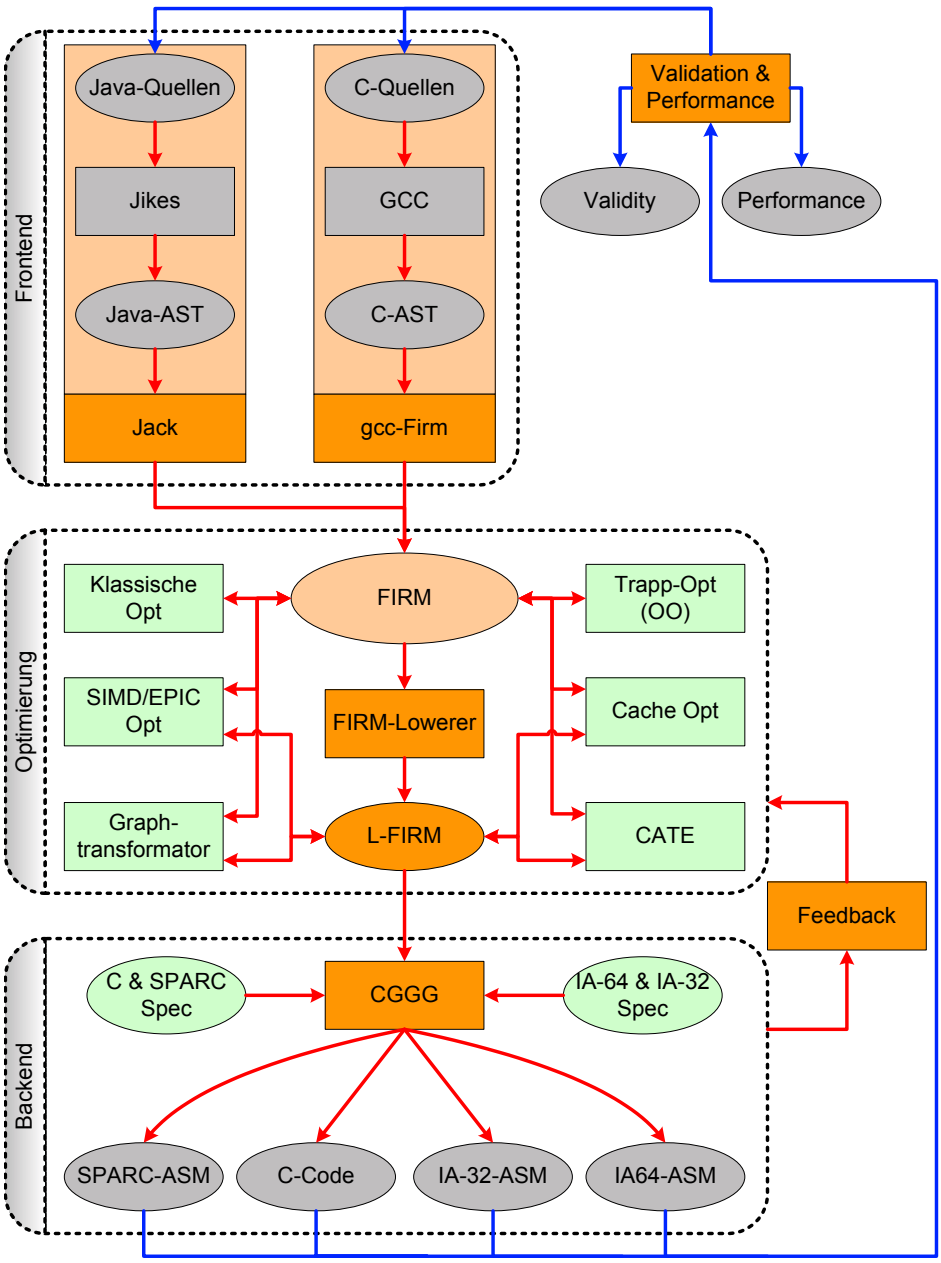


Abbildung 6.1: CRS — Compiler Research System am IPD

ANHANG A

PSEUDOCODE

Wir möchten hier kurz die Bedeutung der in diesem Text verwendeten Pseudo-Maschinensprache erläutern. Sie entspricht einem Grundschatz an Befehlen, der auf jeder herkömmlichen RISC-Architektur verfügbar ist. Fast alle Befehle sind Drei-Adress-Befehle, das heißt, die Befehle haben drei Argumente, zwei Operanden und ein Ziel. Dabei haben die drei Argumente folgende Form:

Darstellung	Bedeutung
`t<Zahl>`	Temporäre Variable. Vom Übersetzer eingefügt
`<Zahl>`	Numerische Konstante
`<Buchstabe>` (nicht `t`)	Adresse einer Variablen

Ferner kann man einem Befehl eine Sprungmarke voranstellen, was die Adresse des Befehls markiert. Man kann dann mit einem Sprungbefehl an diese Stelle springen. Folgende Tabelle gibt einen Überblick über die verwendeten Befehle. Dabei steht ein `n` für eine Zahl, `Ln` für eine Sprungmarke.

Befehl	Berechnung
`add t1 = t2, t3`	$t_1 := t_2 + t_3$
`sub t1 = t2, t3`	$t_1 := t_2 + t_3$
`mul t1 = t2, t3`	$t_1 := t_2 + t_3$
`div t1 = t2, t3`	$t_1 := t_2 + t_3$
`and t1 = t2, t3`	$t_1 := t_2 \,\&\, t_3$
`andn t1 = t2, t3`	$t_1 := t_2 \,\&\, \tilde{}t_3$
`or t1 = t2, t3`	$t_1 := t_2 \mid t_3$
`orn t1 = t2, t3`	$t_1 := t_2 \mid \tilde{}t_3$
`shl t1 = t2, t3`	$t_1 := t_2 << t_3$
`shr t1 = t2, t3`	$t_1 := t_2 >> t_3$
`ld t1 = [t2]`	Lade den Inhalt der Adresse t_2 nach t_1
`st [t1] = t2`	Schreibe t_1 in den Speicher an Adresse t_1
`jmp Ln`	Unbedingter Sprung zu einer Marke
`be t1, Ln`	Sprung zu Marke, wenn $t_1 = 0$
`bne t1, Ln`	Sprung zu Marke, wenn $t_1 \neq 0$
`bl t1, Ln`	Sprung zu Marke, wenn $t_1 < 0$
`ble t1, Ln`	Sprung zu Marke, wenn $t_1 \leq 0$
`bg t1, Ln`	Sprung zu Marke, wenn $t_1 > 0$
`bge t1, Ln`	Sprung zu Marke, wenn $t_1 \geq 0$
`arg t1 = n`	Schreibe das n-te Argument der Funktion in t_1
`ret t1`	Verlasse die Funktion und liefere t_1 zurück.

Darüber hinaus gibt es noch zwei Abkürzungen, die zur besseren Lesbarkeit verwendet werden:

Abkürzung	Bedeutung
`mov t1 = t2`	`or t1 = t2, 0`
`not t1 = t2`	`orn t1 = 0, t2`

ANHANG B

STANDARDWERKE

Appel. *Compiler Construction with Java, C, ML*

Dies sind eigentlich gleich drei Bücher: Jeder Band veranschaulicht die Theorie mit beispielhaften Implementierungen entweder in der Sprache Java, C oder ML. Die Bücher umfassen alle Gebiete des modernen Übersetzerbaus, wobei der Schwerpunkt bei den fortgeschrittenen Themen wie z.B. die Übersetzung von funktionalen und objektorientierten Sprachen, der SSA-Eigenschaft sowie Cache Optimierungen liegt.

Goos, Waite. *Compiler Construction*

Bietet vor allem eine Einführung in attributierte Grammatiken, die von allen anderen Büchern vernachlässigt werden. Auch die formalen Details der Konstruktion von LL(k), LR(1), SLL(k), SLR(1) und LALR(1)-Automaten werden behandelt. Im Backend-Bereich ist die Forschung jedoch schon weiter fortgeschritten. Es ist für nicht-kommerzielle Zwecke unter

`ftp://i44ftp.info.uni-karlsruhe.de/pub/papers/ggoos/CompilerConstruction.ps.gz`

frei zu haben.

Morgan. *Implementing an Optimizing Compiler*

Beschränkt sich nur auf die Optimierungen, die dafür im Detail und sehr gut erklärt vorgestellt werden. Hier findet sich alles, was ein moderner Übersetzer bieten muss.

Muchnik. *Advanced Compiler Design & Implementation*

Steven Muchnik macht dort weiter, wo Andrew Appel aufhört. Mit großer Liebe zum Detail werden hier die einzelnen Optimierungen sowie die ihnen zugrundeliegen Analysen und Darstellungen behandelt. Dieses Buch eignet sich nicht für den Frontendbauer, sehr wohl aber zum weitergeheden Studium optimierender Übersetzer.

Sethi, Aho, Ullman. *Compilerbau*

Das „Drachenbuch“ galt lange als das absolute Standardwerk im Übersetzerbau. Mittlerweile ist vieles darin veraltet. Die Grundlagen, gerade im Frontend-Bereich, sind immer noch brauchbar.

LITERATURVERZEICHNIS

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[Boe98] Boris Boesler. Codeerzeugung aus abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe (TH), IPD, Jun 1998.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[ESL89] H. Emmelmann, F.-W. Schröer, and R. Landwehr. Beg - a generator for efficient back ends. In *ACM Proceedings of the Sigplan Conference on Programming Language Design and Implementation*, June 1989.

[FSF] Free Software Foundation. The GNU Compiler Collection.

[GB03] Sabine Glesner and Jan Olaf Blech. Classifying and formally verifying integer constant folding. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 82(2), April 2003.

[GG78] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254. ACM Press, 1978.

[GLH+92] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[GW84] Gerhard Goos and William Waite. *Compiler Construction*. Springer, Jan 1984.

[Knu99] D. E. Knuth. *MMIXware*, volume 1750 of *Lecture Note In Computer Science*. Springer-Verlag, 1999.

[Lin02] Götz Lindenmaier. libfirm – a library for compiler optimization research implementing firm. Technical Report 2002-5, Universität Karlsruhe, Fakultät für Informatik, Sep 2002.

[McK65] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[Mül95] Thomas Müller. *Effiziente Verfahren zur Befehlsanordung.* PhD thesis, Universität Karlsruhe (TH), IPD, June 1995.

[NKWA96] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code generation = a* + BURS. In *Computational Complexity*, pages 160–176, 1996.

[PLG88] E. Pelegri-Llopart and Susan Graham. Optimal code generation for expression trees: An application of burs theory. In *15th Symposium on Principles of Programming Languages*, pages 294–308, New York, 1988. ACM.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the intermediate representation firm. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[WM92] R. Wilhelm and D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung.* Springer-Verlag, 1992.

INDEX