DIPLOMARBEIT

# Übersetzung von Forth in Maschinensprache

augeführt am Institut für Computersprachen
der Technischen Universität Wien
durch

**Christian Pirker**

Matr.Nr.: 87 26 612

Troststraße 100/2/10
1100 Wien

unter Anleitung von
O. Univ. Prof. Dipl.-Ing. Dr. techn. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Martin Anton Ertl

Wien, 6. Oktober 1995 . . . . . . . . . . . . . . . .

# Kurzfassung

Forth ist eine erweiterbare und interaktive Sprache. Sie stellt zwei programmierersichtbare Stacks (Daten- und Rücksprungstack) zur Verfügung. Die Befehle (Wörter) bearbeiten die Daten auf den Stacks. Die Effizienz der Stackzugriffe und des Programmflusses bestimmen zum großen Teil das Laufzeitverhalten der Forth-Umsetzungen.

Im Rahmen dieser Arbeit erfolgt die Umsetzung eines Übersetzers der Maschinencode für *MIPS RISC Prozessoren* erzeugt. Mit modernen Übersetzungstechniken werden die Forth Programme in Maschinencode übersetzt. Dieser Code ist sofort ausführbar.

Der Maschinencode-Übersetzer konstruiert einen **Datenflußgraphen** für jeden Grundblock des Forth Programms. Einfache **Befehlsauswahls-**, **Befehlsanordnungs-** und **Registerbelegungsalgorithmen** erzeugen dann den Maschinencode. Die Algorithmen versuchen die Speicherzugriffe auf Stacks zu minimieren. Das Aktualisieren der Stackpointer wird nur durchgeführt, wenn es nötig ist.

Der gesamte Maschinencode-Übersetzer ist in Forth geschrieben und kann sich selbst übersetzen. Der Übersetzer ist in den Interpreter integriert, sodaß auch Interpreterworte bearbeitet werden können.

Die vom Maschinencode-Übersetzer erzeugten Programme sind im Bereich von 13 bis 196 % schneller als die interpretierten Programme. Die Übersetzung in Maschinencode dauert aber momentan um 220 % länger als die Übersetzung in den interpretierten Zwischencodes.

# Abstract

Forth is an extensible and interactive language. It provides two programmer-visible stacks (data and returnstack). The supplied instructions (words) manipulate data on the stacks. The efficiency of the stack access and control flow determine mainly the performance of Forth implementations.

This thesis builds a compiler that generates native code for *MIPS RISC processors*. The compiler translates Forth programs into native code using state of the art compiler technology. The code is directly executable on the processor.

The compiler generates a **data flow graph** for each basic block of the program. Then simple **instruction selection**, **instruction scheduling** and **register allocation** algorithms produce the native code. The algorithms try to reduce the stack operations and eliminate unneccesary stack pointer updates.

The native code compiler is written in Forth and can compile itself. The compiler is integrated into the interpreter. Therefore it also handles interpreter words.

Forth programs compiled by this compiler run about 13 to 196 % faster compared to interpreted programs. Currently compiling takes 220 % longer than compiling into interpreting code.

# Danksagung

Besonders danken möchte ich meinen Betreuern, Herrn O. Univ. Prof. Dipl.-Ing. Dr. techn. Manfred Brockhaus und Herrn Univ.-Ass. Dipl.-Ing. Martin Anton Ertl, für Anregungen und Möglichkeiten zur Diskussion.

Diese Arbeit ist meinen Eltern, Elfriede und Martin Pirker, gewidmet, die mir den Besuch einer höheren Schule und das Studium ermöglicht haben.

# Inhaltsverzeichnis

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Sourcenverzeichnis

# Kapitel 1

# Einleitung

Die Programmiersprache Forth ist ein erweiterbare und interaktive Sprache. Sie stellt zwei programmierersichtbare Stacks (Daten- und Rücksprungstack) zur Verfügung. Die Befehle (Wörter) bearbeiten die Daten auf den Stacks. Daraus ergibt sich Postfix. Die Effizienz der Stackzugriffe und des Programmflusses bestimmen zum großen Teil das Laufzeitverhalten der Forth-Umsetzungen.

Das Projekt RAFTS hat einen Übersetzer zum Ziel, der mit Hilfe moderner Übersetzungstechniken die Forth Programme in Maschinencode übersetzt [Ert92].

Das Forth Programm wird in mehrere **Datenflußgraphen** umgewandelt. Eine **Befehlsauswahl**, eine **Befehlsanordnung** und eine **Registerbelegung** wird darauf angewandt um den Maschinencode zu erzeugen.

Zur Optimierung werden folgende Strategien verfolgt:

**Minimierung der Stackverwaltung:**

- Lesen und Schreiben auf die Stacks optimieren.
- Aktualisieren von Stackpointern minimieren.

**schnellere Wortaufrufe.**

In dieser ersten Umsetzung werden nur einfache Algorithmen verwendet. Langfristig gesehen steht die Steigerung der Laufzeitgeschwindigkeit im Vordergrund. Im Rahmen dieser Arbeit ist die Geschwindigkeit vorerst noch nicht das Ziel, sondern nur die korrekte Lauffähigkeit des Übersetzers.

Der Maschinencode ist für *MISP RISC Prozessoren* [KH92] ausgelegt. Als Forth-System wurde `gforth` [1] verwendet. Dieses System [Ert93] arbeitet mit Threaded Code [Loe81], [Kog85].

In weiteren Arbeiten sollen Möglichkeiten untersucht werden, globalere oder sogar interprozedureale Optimierungsverfahren in den Maschinencode-Übersetzer einzubauen. Die Verwendung von *Inlining* und *Tail Call* Optimierung wird auch untersucht werden.

[1] Die Source für `gforth` findet man in *http://www.complang.tuwien.ac.at/forth/gforth-0.1alpha.tar.gz* oder *ftp://ftp.complang.tuwien.ac.at/pub/forth/gforth-0.1alpha.tar.gz*.

## 1.1 Kapitelübersicht

Die Beschreibung von Forth in Kapitel 2 stellt eine Erklärung der, für diese Arbeit notwendigen, Zusammenhänge von Forth dar.

Die Übersetzungstechniken und Strategien für die Umsetzung sind in Kapitel 3 angeführt. Anhand eines durchgehenden Beispiels werden die einzelnen Übersetzungsschritte anschaulich dargestellt.

In Kapitel 4 werden die verwendeten Datenstrukturen der Umsetzung und weitere Umsetzungsdetails beschrieben.

Anhand eines Beispiels werden momentane Schwachstellen in Kapitel 5 aufgezeigt, die in einer Erweiterung des Maschinencode-Übersetzer verbessert werden.

Die Performancetests in Kapitel 6 beinhalten das Zeitverhalten bei der Selbstübersetzung, bei vier allgemeinen Benchmarkprogrammen und beim gesamten Spektrum der Kontrollstrukturen.

## 1.2 Notation

Im gesamten Text wird eine einheitliche Texthervorhebung von Begriffen des selben Bereiches gemacht.

| Begriffe | Beispielausgaben |
|---|---|
| Forthbegriffe | CFA |
| Forthwörter | `dup` |
| Forthausgaben | *`ok`* |
| Maschinencodewörter | `add %sp, %sp, 4` |
| Englische Fachbegriffe | *Tail Call* |

# Kapitel 2

# Forth

Forth [Bro81, Bro84], [Woe92] [RCM93] ist ein höhere Programmiersprache, die in ihren Anfängen auf Arbeiten von Charles H. Moore aus dem Jahre 1969 am *National Radio Astronomy Observatory, Charlottesville (USA)* zurückgeht.

Stackorientierte Sprachen zeichnen sich durch einen ständigen Gebrauch der programmierersichtbaren Stacks aus. Forth-Systeme sind traditionell mit interpretierenden Techniken umgesetzt.

Viele Anwendungen würden von einer schnelleren Umsetzungen von stackorientierten Sprachen profitieren. Steuerungsanwendungen oder Laserdrucker könnten mit billigeren Prozessoren gleiche Performance erreichen.

**Wort**

Dieser Begriff hat eine sehr allgemeine Bedeutung. Es kann ein Befehl sein, der eine Konstante, eine Variable oder eine Prozedure darstellt. Des weiteren kann es ein Kontrollstruktur- oder ein Definitionswort sein. Eine Folge von Zeichen, außer dem Leerzeichen stellt ein Wort dar.

## 2.1 Speicherverwaltung

### 2.1.1 Datentypen

Eine Zelle stellt eine einfache Informationseinheit in der Forth-Architektur dar. Die Größe einer solchen Zelle in Byte ist systemabhängig. Es gibt Systeme mit zwei, vier und acht Byte großen Zellen. Ein Zeichen stellt eine weitere Informationseinheit dar. Ihre Größe beträgt ein Byte.

In Forth werden unterschiedliche Datentypen verwendet. Dabei handelt es sich um Flags für Vergleiche, um Zeichen, um ein- und zweizellige Zahlen, die vorzeichenbehaftet oder vorzeichenlos sein können. Es gibt auch noch Adressen, die auf Zellen- oder auf Zeichengrenzen ausgerichtet sind.

| Daten-Typ | Beschreibung |
|---|---|
| *i*x, j*x* | eine Folge von Elementen |
| *flag* | `true` (-1), `false` (0) |
| *char* | Zeichen |
| *n* | vorzeichenbehaftete einzellige Zahl |
| *+n* | nicht negative einzellige Zahl |
| *u* | vorzeichenlose einzellige Zahl |
| *x* | einzellige Zahl |
| *d* | vorzeichenbehaftete zweizellige Zahl |
| *+d* | nicht negative zweizellige Zahl |
| *ud* | vorzeichenlose zweizellige Zahl |
| *xd* | zweizellige Zahl |
| *xt* | Ausführungstoken |
| *addr* | Adresse |
| *a-addr* | zellenausgerichtete Adresse |
| *c-addr* | charakterausgerichtete Adresse |

Tabelle 2.1.1: Liste von Daten-Typen

In der Tabelle 2.1.1 (diese Seite) werden die vom Standard verwendeten Daten-Typen aufgelistet. Der Speicherbedarf der unterschiedlichsten Stackelemente ist in der Tabelle 2.1.2 (nächste Seite) aufgelistet.

### 2.1.2 Stacks

#### Datenstack

In Forth werden die Parameter über den Datenstack, deshalb auch manchmal Parameterstack genannt, übergeben. Das oberste Element des Datenstack wird mit TOS (*top of stack*) abgekürzt.

Zur Manipulation der Stackelemente gibt es die unterschiedlichsten Wörter, z.B. `dup`, `drop` oder `swap`, die eine Kopie des TOS Elementes am Datenstack erzeugen, das TOS Element am Datenstack entfernen oder die zwei obersten Stackelemente vertauschen. Alle Wörter benötigen zum besseren Verständnis eine Beschreibung welche und wieviel Elemente vom Stack genommen werden und anschließend wieder am Stack abgelegt werden (siehe Abschnitt 2.1.3 Stackdiagramme, nächste Seite). Es gibt auch Wörter, die eine beliebige Anzahl von Elementen, mit einem Zähler als oberstes Element (TOS), entfernen oder ablegen, z.B. `set-order` und `get-order`.

#### Rücksprungstack

Der Rücksprungstack wird zum Speichern von laufzeitrelevanten Daten verwendet. Das oberste Element des Rücksprungstack wird mit TOR (*top of return stack*) abgekürzt.

| Daten-Typ | Größe am Stack | Größe im Speicher |
|---|---|---|
| *flag* | 1 Zelle | 1 Zelle |
| *char* | 1 Zelle | 1 Charakter |
| *n* | 1 Zelle | 1 Zelle |
| *+n* | 1 Zelle | 1 Zelle |
| *u* | 1 Zelle | 1 Zelle |
| *x* | 1 Zelle | 1 Zelle |
| *d* | 2 Zellen | 2 Zellen |
| *+d* | 2 Zellen | 2 Zellen |
| *ud* | 2 Zellen | 2 Zellen |
| *xd* | 2 Zellen | 2 Zellen |
| *xt* | 1 Zelle | 1 Zelle |
| *addr* | 1 Zelle | 1 Zelle |
| *a-addr* | 1 Zelle | 1 Zelle |
| *c-addr* | 1 Zelle | 1 Zelle |

Tabelle 2.1.2: Speicherbedarf, der unterschiedlichsten Stackelemente

Die Daten sind primär die Rücksprungadressen bei Wortaufrufen. Des weiteren werden bei den Zählschleifen die Grenzen bzw. die aktuellen Zähler auf dem Rücksprungstack gespeichert. Es besteht aber auch die Möglichkeit den Rücksprungstack zum Speichern von temporären Daten zu verwenden. Dabei sind einige Einschränkungen zu beachten, da sonst der Programmablauf eventuell gestört wird (siehe [ANS94], Seite 13).

Zur Manipulation der Stackelemente gibt es die unterschiedlichsten Wörter, z.B. `>r`, `r@` oder `r>`, die das TOS Element vom Datenstack auf den Rücksprungstack verschieben, die eine Kopie des TOR Elementes auf den Datenstack kopieren, oder die das TOR Element vom Rücksprungstack auf den Datenstack verschieben.

**Andere Stacks**

Es können noch weitere Stacks in Forth vorhanden sein. Z.B. wird für Gleitkomma-Berechnungen ein eigener Gleitkommastack verwendet.

### 2.1.3 Stackdiagramme

Um das Verhalten eines Wortes nach außen zu beschreiben, ist es von Vorteil, eine genaue Definition des Stackverhaltens in Kommentaren zu vermerken (*stack diagrams*). Es hat sich hiefür eine eigene Notation durchgesetzt.

Es handelt sich hierbei um einen Kommentar, der vom Interpreter/Übersetzter ignoriert wird. Das Wort `(` leitet ein Stack Diagram ein. Anschließend werden die Daten am Stack vor und nach der Ausführung des Wortes aufgelistet, wobei die Daten vor und nachher mittels `--` getrennt werden. Der Kommentar wird dann mit einer `)` abgeschlossen.

`swap` ( $x_1$ $x_2$ -- $x_2$ $x_1$ )

In einigen Fällen ist es notwendig nicht nur die Anzahl der Elemente zu wissen, sondern auch den Daten-Typ. In der Tabelle 2.1.1 (Seite 4) werden die vom Standard verwendeten Daten-Typen aufgelistet.

#### Rücksprungstackdiagramme

Um zwischen dem Daten- und Rücksprungstack in Stackdiagrammen zu unterscheiden, hat sich für Rücksprungstackelemente die Notation `R:` durchgesetzt.

`>r` ( *x* -- ) ( *R:* -- *x* )

### 2.1.4 Lexikon (*dictionary*)

Das Lexikon stellt eine Liste von vorgegebenen oder selbst geschriebenen Wörter dar. Jedes Wort wird zusammen mit seiner Definition in das Lexikon eingetragen. Zum Eintragen eines neuen Wortes in das Lexikon werden die Definitionswörter (siehe Abschnitt 2.2 Definitionswörter, diese Seite) verwendet.

Das Wort `here` ( -- *addr* ) gibt die Speicherzelle des Lexikons an, über die als nächstes verfügt werden kann.

#### Wortliste (*wordlist*)

Ein Lexikon besteht aus mehreren Wortlisten, die teilweise unabhängig voneinander existieren. Den Unterschied zwischen einem Lexikon und einer Wortliste könnte man mit dem Unterschied der beiden Bedeutungen des Wortes “Sprache” vergleichen. “Sprache” ist einmal die menschliche Fähigkeit, sich zu verständigen, zum anderen die Zusammenfassung einer Gruppe von Wörtern (eine Wortliste) und grammatikalischen Regeln (deutsche Sprache, englische Sprache).

Grundsätzlich stehen die Definitionen in der Reihenfolge im Lexikon, in der sie definiert werden, unabhängig davon, welcher Wortliste sie angehören. Die einzelnen Wortlisten sind also keine abgeteilten Gruppen im Lexikon, sondern unabhängig verkettete Listen. Eine Wortliste muß nicht unbedingt eine verkettete Liste sein, sondern kann auch eine andere Umsetzung sein, z.B. in Form einer Hash-Tabelle.

Das Wort `set-current` ( *wid* -- ) setzt die Wortliste, in das die neuen Definitionen geschrieben werden. Die Sichbarkeit und Reihenfolge der Wortlisten beim Suchen von Wörtern kann mit dem Wort `set-order` ( $wid_n$ $wid_1$ $n$ -- ) geändert werden. *wid* stellt dabei einen Identifier der Wortliste dar.

## 2.2 Definitionswörter

Mit den Definitionswörtern können neue Wörter erzeugt werden.

### 2.2.1 Definition

```
: name-1            ( -- )
          ...worte...
;

name-1 Ausführung:  ( i*x -- i*y )
```

Mit der Definition können neue Wörter erzeugt, die andere Wörter aufrufen. Es entspricht einer Prozedur in anderen Programmiersprachen.

Das Wort `:` leitet eine Definition ein. Als erstes muß eine Definition erstellt werden, damit das neue Wort in weitere Folge vom Interpreter aufgerufen werden kann. Es wird vorerst einmal der Name des Wortes im Lexikon gespeichert, unter dem es verwendet werden kann. Damit kann das neu übersetzte Wort vom Interpreter aufgerufen oder vom Übersetzer zum Übersetzen von weiteren Wörtern verwendet werden.

Anschließend werden die Wortaufrufe und/oder Kontrollstrukturen übersetzt.

Mit dem Wort `;` wird die Definition abgeschlossen. Ab diesem Zeitpunkt ist das neue Wort mit dem Namen `name-1` übersetzt und ausführbar.

### 2.2.2 Konstanten

```
constant name-2     ( x -- )
name-2 Ausführung:  ( -- x )
```

Mit dem Wort `constant` wird ein Wort mit dem Namen `name-2` erzeugt, das den konstanten Wert *x* retouniert, der während der Definition des Wortes gespeichert wird.

Wenn eine zweizellige Konstante benötigt wird, kann man mit dem Wort `2constant` eine erzeugen.

### 2.2.3 Variablen

```
variable name-3     ( -- )
name-3 Ausführung:  ( -- a-addr )
```

Mit dem Wort `variable` wird ein Wort mit dem Namen `name-3` erzeugt, das die Adresse des Parameterfeldes (PFA) des Wortes retouniert. Während der Definition wird eine Speicherzelle ab der PFA reserviert. Die retounierte Adresse kann dann entweder lesend oder schreibend (mit den Wörtern `@` ( *a-addr* -- *x* ) oder `!` ( *x a-addr* -- )) bearbeitet werden.

Wenn eine Variable mit zwei Speicherzellen benötigt wird, kann man mit dem Wort `2variable` eine erzeugen.

### 2.2.4 Datenstrukturen

```
create name-4       ( -- )
name-4 Ausführung:  ( -- a-addr )
```

```
allot  ( n -- )
```

Mit dem Wort `create` wird ein Wort mit dem Namen `name-4` erzeugt, das die Adresse des Parameterfeldes (PFA) des Wortes retouniert. Dabei wird im Unterschied zu `variable` noch keine Speicherzelle reserviert. Mit dem Wort `allot` kann man das Parameterfeld des zuletzt definierten Wortes vergrössern. Mit der retounierten Adresse kann dann die Adresse einer beliebigen nachflogenden Speicherzellen berechnet werden. Es kann dann lesend oder schreibend auf diese Speicherzelle zugegriffen werden.

Das nachfolgende Beispiel zeigt zwei gleiche Definitionen einer einzelligen Variablen. Einmal mit dem Wort `variable` und einmal mit den Wörtern `create` und `allot` erzeugt.

```
variable var-1

create var-1 cell allot
```

### 2.2.5 `does>` Struktur

Zur Erzeugung von neuen Definitionswörtern kommt das Wort `does>` zur Anwendung. `does>` steht innerhalb der Definition von Definitionswörtern und markiert das Ende des Definitionszeit-Teils und den Beginn des Verwendungszeit-Teils.

`does>` kommt in Kombination mit `create` zur Anwendung. Beim Definieren eines neuen Wortes wird der Definitionsteil aufgerufen. Bei der Verwendung des neuen Wortes wird die Adresse des Parameterfeldes (PFA) auf dem Datenstack gelegt und anschließend der Verwendungsteil ausgeführt.

Das nachfolgende Beispiel zeigt zwei gleiche Definitionen einer einzelligen Konstanten. Einmal mit dem Wort `constant` und einmal mit dem Wort `con`. Das Definitionswort `con` wird mit den Wörtern `create` und `does>` erzeugt.

```
123 constant con-1

: con
  create ,
  does> @ ;
123 con con-1
```

Ein Beispiel zur `does>` Struktur wird in Source 2.2.1 (nächste Seite) gezeigt. Bei dieser Ausgabe handelt es sich um eine Kombinierte Ausgabe des Source-Teiles und den Ausgaben von `gforth`. Die Ausgabe *`complex:create`* wird vom Übersetzungs-Teil erzeugt. Die Ausgabe *`complex:does>`* wird vom Laufzeit-Teil erzeugt.

## 2.3 Kontrollstrukturen

Es werden folgende Kontrollstrukturen unterstützt:

```
: complex ( flag - )  compiled
  create  compiled
    ." complex: create" cr  compiled
    , ,  compiled
  does>  compiled
    ." complex: does>" cr  compiled
    dup cell+ @  compiled
    swap @ ;  ok
  ok
  ok
1 2 complex c1 complex: create
 ok
3 4 complex c2 complex: create
 ok
c1 .s cr 2drop complex: does>
<2> 1 2
 ok
c2 .s cr 2drop complex: does>
<2> 3 4
 ok
```

Source 2.2.1: Beispiele für `does>`

- Bedingte Sprünge
- Einfache Schleifen
- Zählschleifen

Bei allen Überprüfungen von Bedingungen wird das TOS als Flag interpretiert. Dabei wird `-1` als `true` und `0` als `false` verwendet (als `true` wird im allgemeinen alles ungleich 0 angesehen).

Diese Bedingungen können mittels Vergleichswörter (<, =, <>, > …) berechnet werden. Es besteht natürlich auch die Möglichkeit mehrer Vergleiche mittels logischer Wörter (`and`, `or`, `xor`, `invert`) zu verbinden.

### 2.3.1 Bedingte Sprünge

Die bedingten Sprünge haben folgenden syntaktischen Aufbau:

- ≪*flag-1*≫ `if` ≪*code-1*≫ `endif`
- ≪*flag-2*≫ `if` ≪*code-2*≫ `else` ≪*code-3*≫ `endif`

- *≪val≫* `case` *≪valof-1≫* `of` *≪codeof-1≫* `endof` …
  *≪valofn≫* `of` *≪codeofn≫* `endof` *≪default≫* `endcase`

Wenn *≪flag-1≫* `true` ist, wird bei einem einfachen `if` *≪code-1≫* ausgeführt. Wenn *≪flag-2≫* `true` ist, wird bei einem alternativen `if` *≪code-2≫* ausgeführt, wenn *≪flag-2≫* `false` ist wird *≪code-3≫* ausgeführt.

Bei einem `case` wird *≪val≫* vom Stack genommen und mit allen *≪valof-i≫* verglichen. Wenn eine Übereinstimmung gefunden wird, wird der entsprechende *≪codeof-i≫* ausgeführt. An sonsten wird der *≪default≫* ausgeführt.

Beispiele zu den bedingten Sprüngen werden in Source 2.3.1 (nächste Seite) gezeigt.

### 2.3.2 Einfache Schleifen

Die einfachen Schleifen haben folgenden syntaktischen Aufbau:

- `begin` *≪code-1≫* `again`
- `begin` *≪code-2≫* *≪flag-2≫* `until`
- `begin` *≪flag-3≫* `while` *≪code-3≫* `repeat`

`begin` …`again` stellt eine Endlosschleife dar. Sie kann mit einem `exit` verlassen werden.

Bei der `begin` …`until` Schleife wird mindestens einmal der *≪code-2≫* ausgeführt bis die Abbruchbedingung *≪flag-2≫* überprüft wird. Bei Nichterfüllung der Bedingung wird die Schleife erneut ausgeführt.

Bei der `begin` …`while` …`repeat` Schleife wird zuerst die Abbruchbedingung *≪flag-3≫* überprüft. Bei Erfüllung der Bedingung wird der *≪code-3≫* ausgeführt. Anschließend wird die Abbruchsbedingung erneut überprüft.

Beispiele zu den einfachen Schleifen werden in Source 2.3.2 (Seite 12) gezeigt.

### 2.3.3 Zählschleifen

Die Schleifen haben folgenden syntaktischen Aufbau:

- *≪bis-1≫* *≪von-1≫* `do` *≪code-1≫* `loop`
- *≪bis-2≫* *≪von-2≫* `do` *≪code-2≫* *≪inc≫* `+loop`
- *≪bis-3≫* *≪von-3≫* `?do` *≪code-3≫* `loop`
- *≪bis-4≫* *≪von-4≫* `?do` *≪code-4≫* *≪inc≫* `+loop`

Bei den Zählschleifen wird am Stack ein Bereich *≪von-i≫* *≪bis-i≫* angegeben. Beim Wort `do` werden diese Grenzen einfach ohne zu überprüfen gesetzt. Das Wort `?do` hingegen überprüft vor dem Start der Schleife, ob die Grenzen gleich sind und beendet sie gegebenenfalls sofort.

```
: foo-if ( flag - )  compiled
  if  compiled
    ." true" endif ;  ok
  ok
true foo-if cr true
 ok
false foo-if cr
 ok
```

```
: foo-ifelse ( flag - )  compiled
  if  compiled
    ." true" else  compiled
    ." false" endif ;  ok
  ok
true foo-ifelse cr true
 ok
false foo-ifelse cr false
 ok
```

```
: foo-case ( n - )  compiled
  case  compiled
    0 of  compiled
      ." null" endof  compiled
    1 of  compiled
      ." eins" endof  compiled
    ." default"  compiled
    endcase ;  ok
  ok
0 foo-case cr null
 ok
1 foo-case cr eins
 ok
99 foo-case cr default
 ok
```

Source 2.3.1: Beispiele für bedingte Sprünge

```
: foo-again ( n - )  compiled
  begin  compiled
    dup u.  compiled
    dup 16 >= if  compiled
      exit endif  compiled
    1+ again ;  ok
  ok
7 foo-again cr 7 8 9 10 11 12 13 14 15 16
 ok
16 foo-again cr 16
 ok
```

```
: foo-until ( n - )  compiled
  begin  compiled
    dup u.  compiled
    dup 1+  compiled
    swap 16 >= until ;  ok
  ok
7 foo-until cr 7 8 9 10 11 12 13 14 15 16
 ok
16 foo-until cr 16
 ok
```

```
: foo-while ( n - )  compiled
  begin  compiled
    dup 16 <= while  compiled
    dup u.  compiled
    1+ repeat ;  ok
  ok
7 foo-while cr 7 8 9 10 11 12 13 14 15 16
 ok
16 foo-while cr 16
 ok
```

Source 2.3.2: Beispiele für einfache Schleifen

| Daten-Typ | Beschreibung |
|---|---|
| *colon-sys* | für neue Wortdefinitionen |
| *dest* | für Schleifen |
| *orig* | für Sprünge |
| *case-sys* | für `case` |
| *of-sys* | für `of` |
| *do-sys* | für `do`, `loop` |
| *loop-sys* | für `do`, `loop` |

Tabelle 2.3.1: Liste von Kontrollstruktur-Typen

Mit `loop` wird die Schleife um eins erhöht und überprüft, ob die Grenze erreicht ist. Dadurch wird die Schleife ≪*bis-i*≫ - ≪*von-i*≫ mal ausgeführt.

Das Wort `+loop` erhält einen Inkrementierzähler ≪*inc*≫ übergeben. Dadurch lassen sich auch andere Schrittweiten als eins ausführen. Da die Schritte sowohl positive als auch negative sein können muß die Grenzprüfung etwas komplizierter ausfallen. Hier wird nicht auf Gleichheit der Grenzen geprüft, sondern ob die Grenze überschritten wurde, und zwar ist das möglich von beiden Seiten.

Auf den Zähler der aktuellen Zählschleife kann mit dem Wort `i` ( -- *n* ) zugegriffen werden. Es besteht auch die Möglichkeit bei zwei geschachtelten Zählschleifen mit dem Wort `j` ( -- *n* ) auf den Zähler der äußeren Zählschleife zugreifen.

Beispiele zu den Zählschleifen werden in Source 2.3.3 (nächste Seite) gezeigt.

## 2.3.4 Kontrollflußstack

Bei der Bearbeitung von Kontrollstrukturen fallen auch Daten an. Es handelt sich dabei um Zieladressen von Sprüngen. Zur Speicherung dieser Daten wird ein eigener Kontrollflußstack eingerichtet. Der Kontrollflußstack muß im System nicht unbedingt physisch vorhanden sein. Falls er nicht existiert, kann auch der Datenstack dafür verwendet werden, da er zur Übersetzungszeit sonst nur unbenutzt vorhanden ist.

Zur Manipulation der Stackelemente gibt es die unterschiedlichsten Wörter, z.B. `cs-pick` und `cs-roll`, die eine Kopie des n-ten Elementes vom Kontrollflußstack auf den Kontrollflußstack kopieren oder den Kontrollflußstack um n Elemente umordnen.

Beim Kontrollflußstack werden zur Beschreibung des Stackverhaltens spezielle Datentypen verwendet. In der Tabelle 2.3.1 (diese Seite) werden die vom Standard verwendeten Kontrollstruktur-Typen aufgelistet.

### Kontrollflußstackdiagramme

Um zwischen dem Daten- und Kontrollflußstack in Stackdiagrammen zu unterscheiden, hat sich für Kontrollflußstackelemente die Notation `C:` durchgesetzt.

```
: foo-do-loop ( n - )  compiled
  16 swap do  compiled
    i u.  compiled
    loop ;  ok
  ok
7 foo-do-loop cr 7 8 9 10 11 12 13 14 15
 ok
16 foo-do-loop cr 16 17 18 ... 4294967294 4294967295 0 1 2
... 13 14 15
 ok
```

```
: foo-do-+loop ( n - )  compiled
  16 swap do  compiled
    i u.  compiled
    2 +loop ;  ok
  ok
7 foo-do-+loop cr 7 9 11 13 15
 ok
16 foo-do-+loop cr 16 18 20 ... 4294967292 4294967294 0 2
... 10 12 14
 ok
```

```
: foo-?do-loop ( n - )  compiled
  16 swap ?do  compiled
    i u.  compiled
    loop ;  ok
  ok
7 foo-?do-loop cr 7 8 9 10 11 12 13 14 15
 ok
16 foo-?do-loop cr
 ok
```

```
: foo-?do-+loop ( n - )  compiled
  16 swap ?do  compiled
    i u.  compiled
    2 +loop ;  ok
  ok
7 foo-?do-+loop cr 7 9 11 13 15
 ok
16 foo-?do-+loop cr
 ok
```

Source 2.3.3: Beispiele für Zählschleifen

| Daten-Typ | Beschreibung |
|---|---|
| *<char>* | ein Zeichen |
| *<chars>* | mehrere Zeichen |
| *<”>* | Sequenz bis zum Anführungszeichen |
| *<)>* | Sequenz bis zur rechten Klammer |
| *<eol>* | Sequenz bis *end of line* |
| *ccc* | Sequenz von beliebigen Zeichen |
| *name* | Sequenz von Zeichen (ohne Leerzeichen) |

Tabelle 2.4.1: Liste von Texteingabeabkürzungen

`if` ( *C: -- orig* )

## 2.4 Äußerer Interpreter/Übersetzer

Forth ist traditionell ein interpretierendes System. Interpreter können langsam sein, aber Forth versucht mit einer Interpretation in zwei Ebenen ein schnelles Interpretieren zu ermöglichen. Es handelt sich dabei um den inneren und den äußeren Interpreter.

### 2.4.1 Äußerer Interpreter

Der Äußere Interpreter ist sehr einfach aufgebaut. Der Interpreter liest eine Zeile vom Terminal oder von einer Datei ein. Forth unterteilt diese Zeile in Wörter. Diese Wörter werden dann von links nach rechts schrittweise ausgeführt. Zuerst wird im Lexikon nach dem auszuführendem Wort gesucht. Wenn es gefunden wird, wird es ausgeführt und anschließend beim nächsten Wort fortgefahren. Wird es nicht gefunden, wird versucht es als Zahl im aktuellen Zahlensystem zu interpretieren und am Datenstack abzulegen. Falls auch das nicht gelingt wird eine Fehlermeldung ausgegeben.

Ein Listing eines Äußeren Interpreters ist in [Zec84] (Anhang II) angeführt.

#### Texteingabeabkürzungen

Auch bei Texteingaben hat sich eine Konvention herausgebildet, um eindeutig das Verhalten eines Wortes zu kommentieren. Eine Auflistung der bekannten Abkürzungen wird in der Tabelle 2.4.1 (diese Seite) gezeigt.

### 2.4.2 Übersetzungs-/Laufzeit

In Forth muß zwischen der Übersetzungs- und der Laufzeit unterschieden werden.

Bei einfachen Wörtern wird in der Übersetzungszeit einfach ein Wortaufruf übersetzt. Zur Laufzeit wird das Wort ausgeführt. Dabei führt es seiner Definition entsprechende Funktionen aus.

```
: hello ( - )  compiled
  ." Hello" cr ; immediate  ok
  ok
: swp ( x1 x2  - x2 x1 x1 )  compiled
  swap hello dup ; Hello
 ok
  ok
1 2 .s cr <2> 1 2
 ok
swp .s cr drop 2drop <3> 2 1 1
 ok
```

Source 2.4.1: Beispiele für `immediate`

Einige Wörter werden “vorrangig” markiert (siehe Abschnitt 2.4.3 `immediate` und `postpone` Strukturen, diese Seite). Vorrangige Wörter werden noch während der Übersetzung neuer Wörter ausgeführt, anstatt in das neue Wort übersetzt zu werden. Es handelt sich dabei z.B. um Kontrollstrukturwörter. In Forth besteht aber die Möglichkeit beliebige Wörter zu erzeugen, die schon zur Übersetzungszeit ausgeführt werden.

**Multistackdiagramme**

Die Stackdiagramme beschreiben im allgemeinen das Verhalten bei der Laufzeit. Bei einigen Wörtern ist es notwendig das Stackverhalten auch während anderer Situationen zu wissen, da sie Daten auf den Stacks in dieser Situation anders bearbeiten. Aus diesem Grund gibt es auch noch eine Art von Stackdiagrammen, die getrennt für beide Situation das Stackverhalten darlegt.

| | |
|---|---|
| `;` *Übersetzung:* | ( *C: colon-sys* -- ) |
| *Laufzeit:* | ( *R: sys* -- ) |

### 2.4.3 `immediate` und `postpone` Strukturen

`immediate` markiert das zuletzt definierte Wort als “vorrangig”. In der Übersetzungszeit wird dieses Wort nicht übersetzt, sondern ausgeführt.

Ein Beispiel zur `immediate` Struktur wird in Source 2.4.1 (diese Seite) gezeigt. Bei dieser Ausgabe handelt es sich auch um eine kombinierte Ausgabe des Source-Teiles und den Ausgaben von `gforth`. Die Ausgabe von *`Hello`* des Wortes `hello` wird nur bei der Übersetzung des Wortes `swp` erzeugt.

Das Wort `postpone` verschiebt den Übersetzungsvorgang wieder um eine Ebene nach vorne. Damit können `immediate` Wörter auch in der Definition neuer Wörter übersetzt verwendet werden.

```
: compile-dup ( - )  compiled
  ." compile-postpone" cr  compiled
  postpone dup ; immediate  ok
  ok
: new-dup ( x - x x )  compiled
  compile-dup ; compile-postpone
 ok
  ok
1 .s cr <1> 1
 ok
new-dup .s cr 2drop <2> 1 1
 ok
```

Source 2.4.2: Beispiele für `postpone`

Es können auch nicht `immediate` Wörter mit `postpone` übersetzt werden. Dabei wird nicht der Aufruf des Wortes in das neue Wort übersetzt, sondern ein Übersetzungsaufruf des Wortes. Dieses neue Wort muß mit `immediate` markiert werden, damit bei der Verwendung des Wortes der Übersetzungsvorgang korrekt ausgeführt wird.

Ein Beispiel zur `postpone` Struktur wird in Source 2.4.2 (diese Seite) gezeigt. Bei dieser Ausgabe handelt es sich auch um eine kombinierte Ausgabe des Source-Teiles und den Ausgaben von `gforth`.

## 2.5 Implementation

Der innere Interpreter ist dafür zuständig, daß der Programmfluß zustande kommt. Es bestehen dafür mehrere Möglichkeiten, um eine Verknüpfung zwischen den Wörtern herzustellen:

- direkt verknüpft
- indirekt verknüpft
- subroutinen verknüpft
- Maschinencode

### 2.5.1 Forth-Wörter

Bei den Forth-Wörtern müssen wir zwischen Primitive- und Secondary–Wörtern unterscheiden.

**Primitive-Wörter**

Diese Wörter sind in Maschinencode aufgebaut.

**Secondary–Wörter**

Diese Wörter sind Definitionen. Sie bestehen aus einer Folge von Code-Adressen (CA) bei direkt verknüpften oder Wort-Adressen (WA) bei indirekt verknüpften Forth-Systemen von Primitive- und/oder Secondary–Wörtern.

### 2.5.2 direkt verknüpft

Bei der direkt verknüpften Struktur wird ein Instruktions Pointer (IP) benötigt, Er ermöglicht ähnliche Kontrollfunktionen wie der Programm Counter (PC) auf der Maschinenebene.

Ein direkt verknüpftes Wort muß immer mit Maschinencode ab der Codefeldadresse (CFA) starten. Wenn ein Secondary–Wörter erzeugt wird muß ein `jump` Befehl der `call` Routine erzeugt werden.

Bei einem direkt verknüpften Wort werden mit folgender `next` Sequenz die Code-Adressen CA i interpretiert:

**CODE** `next`
$(ip) \rightarrow ca$
$ip + 1 \rightarrow ip$
`jump` $ca$

Beim Interpretieren eines direkt verknüpften Wortes wird beim Aufruf des Wortes die `call` Sequenz ausgeführt und beim Verlassen die `return` Sequenz.

**CODE** `call`
$ip \rightarrow$ `push`
$ca + 1 \rightarrow ip$
`jump next`

**CODE** `return`
`pop` $\rightarrow ip$
`jump next`

Abbildung 2.5.1 (nächste Seite) stellt eine graphische Darstellung der direkt verknüpften Verknüpfungsart dar.

### 2.5.3 indirekt verknüpft

Auch bei der indirekt verknüpften Struktur wird ein Instruktions Pointer (IP) benötigt. Ein indirekt verknüpftes Wort startet ab der Codefeldadresse (CFA) mit einer Code-Adresse.

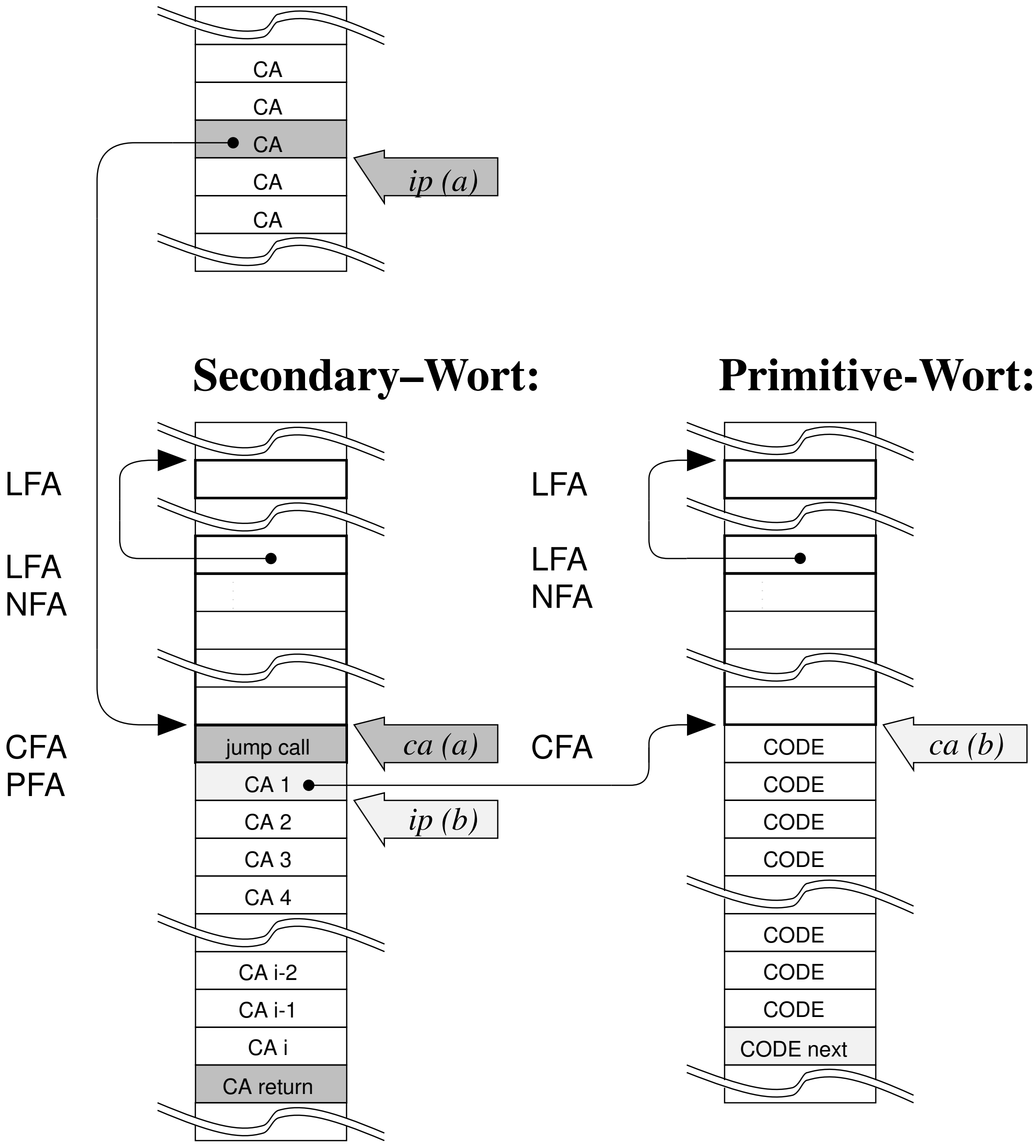


Abbildung 2.5.1: Verknüpfungsstruktur: direkt verknüpft

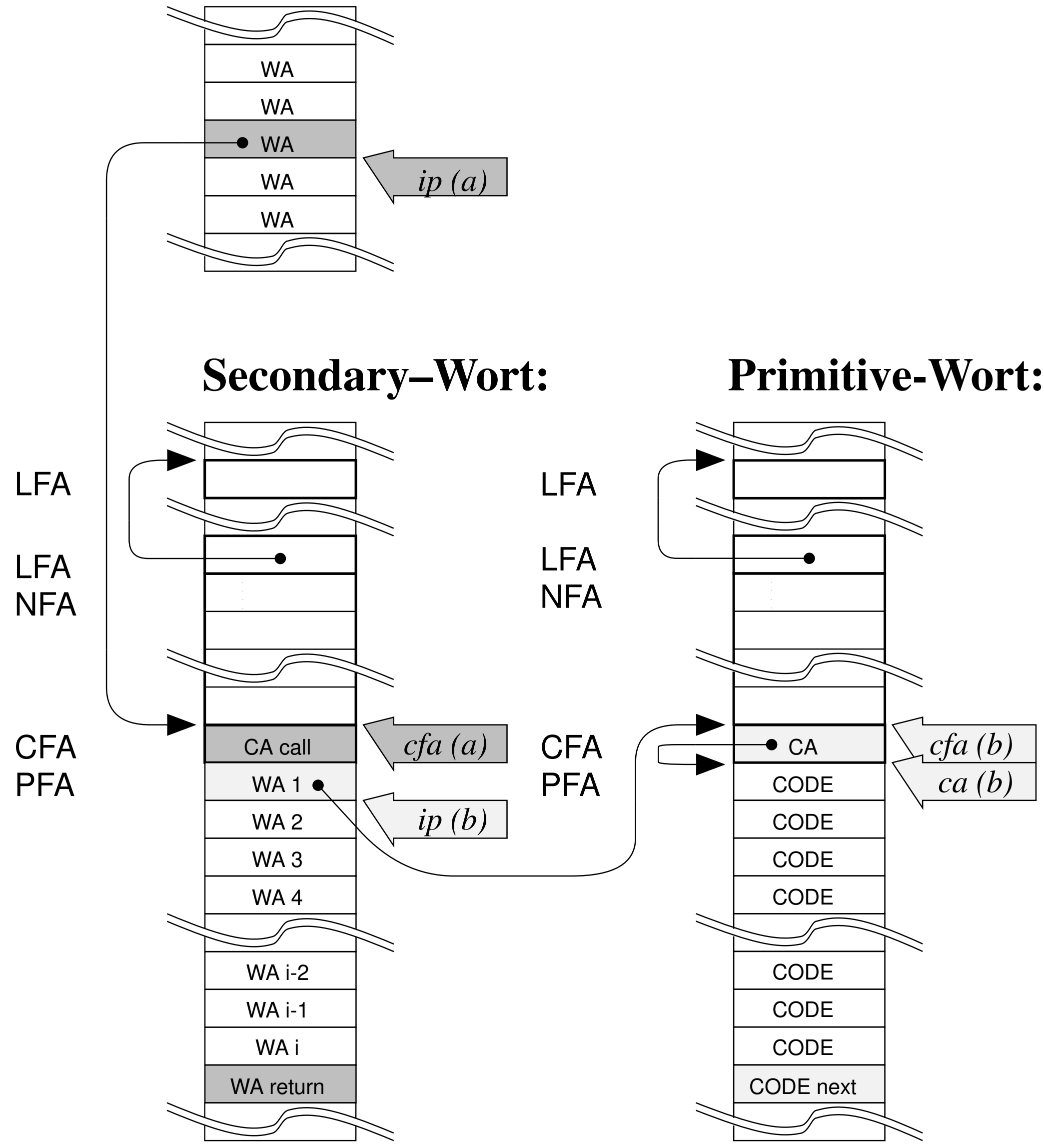


Abbildung 2.5.2: Verknüpfungsstruktur: indirekt verknüpft

Bei einem indirekt verknüpften Wort werden mit folgender `next` Sequenz die Wort-Adressen WA i interpretiert:

**CODE** `next`
$(ip) \to cfa$
$ip + 1 \to ip$
$(cfa) \to ca$
`jump` $ca$

Beim Interpretieren eines indirekt verknüpften Wortes wird beim Aufruf des Wortes die `call` Sequenz ausgeführt und beim Verlassen die `return` Sequenz.

**CODE** `call`
$ip \to$ `push`
$cfa + 1 \to ip$
`jump next`

**CODE** `return`
`pop` $\to ip$
`jump next`

Abbildung 2.5.2 (vorige Seite) stellt eine graphische Darstellung der indirekt verknüpften Verknüpfungsart dar.

### 2.5.4 subroutinen verknüpft

Bei dieser Verknüpfungsart stellt der Prozessor eine Möglichkeit zur Verfügung, damit Unterprogrammeaufrufe erstellt werden können. Dieser Aufruf speichern die Rücksprungadresse auf einem Stack, sodaß ein geschachteltes Aufrufen von Wörtern erfolgen kann.

Für den Programmfluß wird hier kein eigener Instruktions Pointer (IP) benötigt, wie bei der direkten- oder indirekten Verknüpfung. Der Programmfluß kommt hier durch die Rücksprungadressen aus den Unterprogrammaufrufen zustande.

Bei einem subroutinen verknüpftem Wort wird auch keine `next` Sequenz benötigt.

Beim Interpretieren eines subroutinen verknüpften Wortes wird beim Aufruf die `call` Sequenz mit der Code-Adresse (CA) ausgeführt und beim Verlassen des Wortes die `return` Sequenz.

**CODE** `call`
`jsr` CA

**CODE** `return`
`rts`

Abbildung 2.5.3 (nächste Seite) stellt eine graphische Darstellung der subroutinen verknüpften Verknüpfungsart dar.

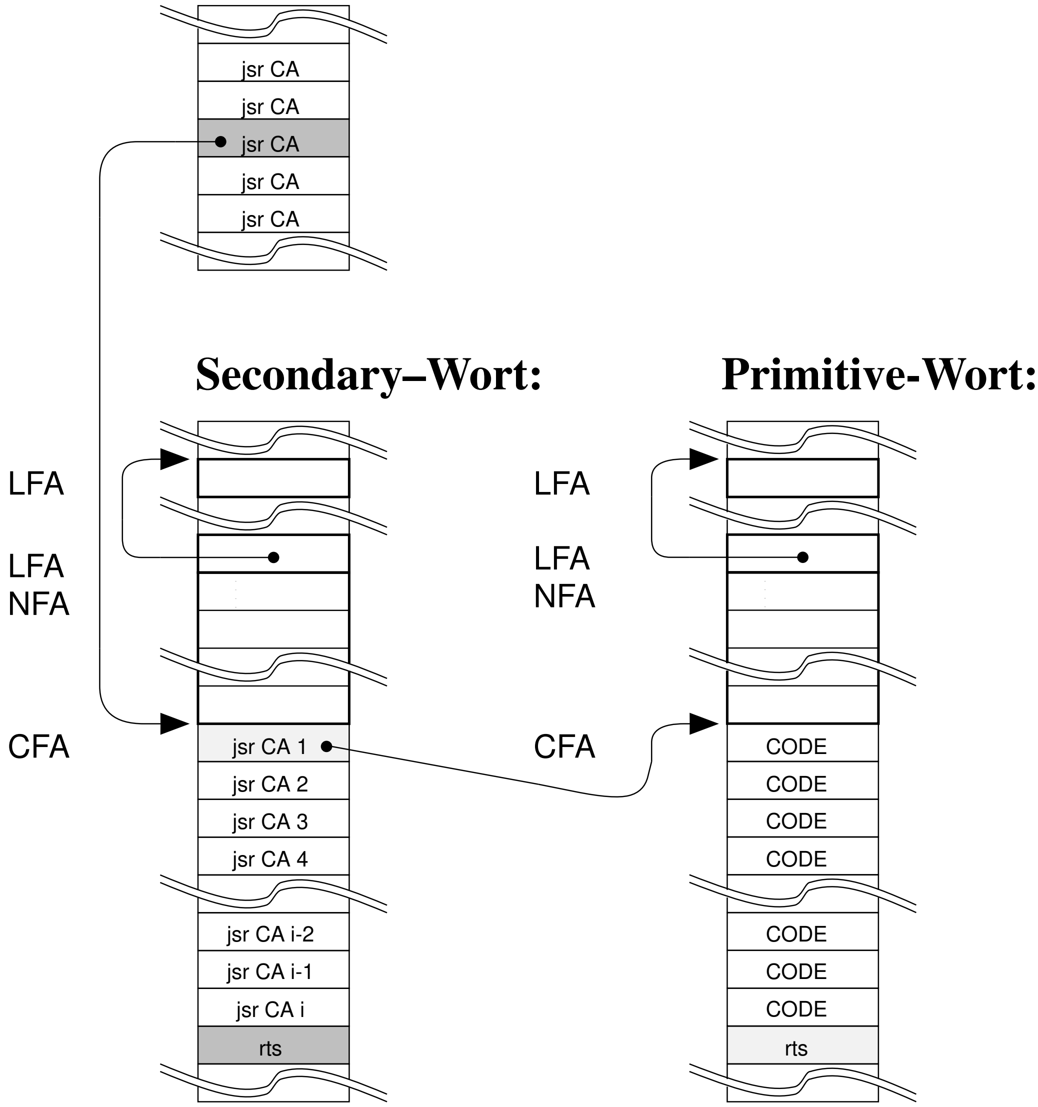


Abbildung 2.5.3: Verknüpfungsstruktur: subroutinen verknüpft

### 2.5.5 Maschinencode

Bei dieser Verknüpfungsstruktur werden mehrer Wörter in Maschinencode zusammengefaßt. Zum Unterschied zur Verknüpfungsstruktur mit Subroutinen werden auch neue Wörter in Maschinencode erzeugt.

Es gibt schon einige Forth Übersetzer, die Maschinencode erzeugen [Ros86], [Alm86], [Pay91].

Abbildung 3.5.1 (Seite 33) stellt eine graphische Darstellung der Maschinencode Struktur dar, die von dem Maschinencode-Übersetzer dieser Arbeit erzeugt werden.

# Kapitel 3

# Übersetzungstechniken und Umsetzungsstrategien

Das Ziel dieses Maschinencode-Übersetzers ist es schnellere Wörter in Forth zu erzeugen. Dabei wird mit modernen Übersetzungstechniken der Maschinencode erzeugt. Im Rahmen dieser Arbeit wird der Maschinencode für *MIPS RISC Prozessoren* erzeugt.

Die Forth Programme werden in Grundblöcke aufgeteilt. Aus dem Grundblock wird ein Datenflußgraph erzeugt. Anschließend wird der Maschinencode mit Befehlsauswahl, Befehlsanordnung und Registerbelegung erzeugt [Ert92].

**Grundblock**

Ein Grundblock ist eine Folge fortlaufender Anweisungen, in die der Kontrollfluß am Anfang eintritt und die er am Ende verläßt, ohne das er dazwischen anhält oder verzweigt.

## 3.1 Datenflußgraph

Zunächst stellt sich die Frage, welche Abhängigkeiten zwischen Befehlen überhaupt bestehen. Ein Befehl $a$ muß in folgenden Fällen vor einem anderen Befehl $a$ ausgeführt werden:

**Datenfluß-Abhängigkeit** (*flow dependence*, *RAW dependence*) Wenn $b$ von einem Register oder Speicherplatz liest, der von $a$ beschrieben wurde.

**Lese-Schreib–Abhängigkeit** (*anti dependence*, *WAR dependence*) Wenn $b$ auf eine Stelle schreibt, die zuvor von $a$ gelesen wurde.

**Schreib-Schreib–Abhängigkeit** (*output dependence*, *WAW dependence*) Wenn $b$ auf eine Stelle schreibt, die zuvor von $a$ beschrieben wurde.

**potentielle Abhängigkeit** Wenn nicht sicher ist, daß zwei Zugriffe auf verschiedene Stellen erfolgen, z.B. bei Zugriffen über Zeiger, muß die Annahme getroffen werden, daß eine Abhängigkeit zwischen den zwei Zugriffen besteht.

Ein Datenflußgraph (*data flow graph*) stellt im allgemeinen einen Baumgraphen der Datenfluß-Abhängigkeiten dar. Wie wir aber in weiterer Folge sehen werden kann ein azyklischer Graph (DAG's) entstehen, wenn Teilgraphen im Datenflußgraph mehrmals verwendet werden.

Ein um Lese-Schreib-, Schreib-Schreib- und potentielle Abhängigkeiten erweiterter Datenflußgraph ist ein Datenabhängigkeitsgraph (*data dependence graph*).

### 3.1.1 Einbau von Literals, Konstanten, Variablen

#### Literals

Bei den Literals wird ein Datenflußknoten erzeugt, der den konstanten Wert beinhaltet.

#### Konstanten

Zur Übersetzungszeit wird der Wert der Konstanten ausgelesen. In weitere Folge kann der konstante Wert dann wie ein Literal verarbeitet werden.

#### Variablen

Zur Übersetzungszeit wird die Adresse der Variablen ermittelt. In weitere Folge kann die Adress der Variablen dann wie ein Literal verarbeitet werden. Ob dann lesend oder schreibend auf die Adresse zugegriffen wird, hängt vom zu übersetzenden Forth-Programm ab.

### 3.1.2 Einbau von Stackbefehlen

Die Stackbefehle werden zur Übersetzungszeit fast vollständig wegoptimiert, indem sie sofort ausgeführt werden und den Datenstack entsprechend umordnen. Bei den Wörtern `pick` und `roll` kann zur Übersetzungszeit keine Optimierung gemacht werden, wenn der Zähler im TOS nicht als Literal vorkommt.

Durch die Wörter `dup` oder `over` wird aus dem Datenflußgraphen ein azyklischer Graph. Um später eine optimierendere Befehlsanordnung zu erreichen werden die entsprechenden Datenflußknoten einfach mit einem Linkcount versehen, der bei n-maliger Verwendung n-1 ist.

### 3.1.3 Einbau von anderen Befehlen

Wenn es sich um einen im Grundblock verarbeitbaren Befehl handelt, wird ein Datenflußknoten erzeugt. Die benötigten Vorgängerknoten werden von Datenstack genommen.

Bei anderen Befehlen wird der Grundblock beendet und dabei der Maschinencode für den Grundblock erzeugt. Anschließend wird der Wortaufruf übersetzt (siehe Abschnitt 3.5 Wortaufrufe, Seite 32). und wieder ein neuer Grundblock begonnen.

```
: Zahlen-Summe ( n -- [n*[n+1]]/2 )
  dup 1+ * 2/ ;

50 Zahlen-Summe . cr
1275
```

Source 3.1.1: Beispiel: Zahlen-Summe

### 3.1.4 Beispiel für Datenflußgraph

Das hier verwendete Beispiel berechnet die Zahlensumme nach der Gauß'schen Zahlensummenformel.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{3.1.1}$$

Bei dem hier beschriebenen Wort handelt es sich um ein einfaches Wort, daß nur aus Wörtern besteht, die innerhalb eines Grundblockes in Maschinenbefehle umgesetzt werden können.

Im ersten Schritt wird ein Datenflußgraph erzeugt. Beim einfachen Algorithmus, der im Maschinencode-Übersetzer umgesetzt ist, werden die Wörter mit den Datenfluß-Abhängigkeiten in einen Datenflußgraphen eingebaut. Bei Stackwörtern (z.B. `dup`, `drop` oder `swap`) wird einfach nur der Datenstack entsprechend umgeordnet. Bei anderen Wörtern wird in die `node`-Strukturen die Art des Knotens in das Feld `node_type` eingetragen (die möglichen Arten sind in Source 4.2.1 (Seite 47) beschrieben).

Die so ermittelten Datenfluß-Abhängigkeiten sind in Abbildung 3.1.1 (nächste Seite) angegeben.

## 3.2 Befehlsauswahl

Die Befehlsauswahl (*instruction selection*, *code selection*) erfolgt auf Grundblockebene, wobei Grundblöcke als Datenflußgraphen dargestellt werden.

Die Aufgabe der Befehlsauswahl ist es, den Datenflußgraphen in Bäume aufzuteilen, die Maschinebefehlen entsprechen. Das Ergebnis der Befehlsauswahl ist ein Datenflußgraph, in dem die Knoten Maschinenbefehle sind.

### 3.2.1 Umsetzung der Befehlsauswahl

Die Befehle zum Abspeichern der berechneten Daten auf dem Datenstack, wenn das neue Wort ausgeführt wird, werden am Ende des Grundblockes in den Datenflußgraphen eingebaut. Das eventuelle Angleichen des Datenstackpointer an den veränderten Datenstack

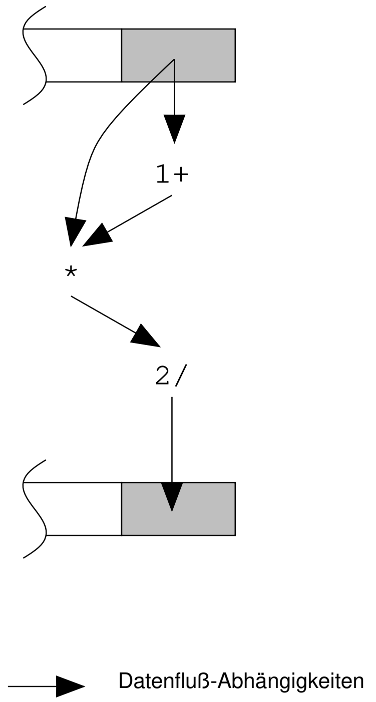


Abbildung 3.1.1: Wort: `Zahlen-Summe` (Datenflußgraph)

wird auch hier eingebaut. Für diese Befehle werden auch zusätzliche Lese-Schreib–Abhängigkeiten im den Graphen eingebaut. Bei größeren Grundblöcken kann es durchaus zu Situationen kommen, daß ein Datenstackelement schon auf den Datenstack auf Position n geschrieben werden könnte, ohne zuvor das Datenstackelement von Position n ausgelesen zu haben, da es erst bei der Befehlsanordnung später eingeordnet wird. Durch das Einbauen von solchen Abhängigkeiten kann auch diese Situation korrekt bearbeitet werden.

### Bearbeitung von Literals

Bei den Literals werden Maschinenbefehle erzeugt, die den konstanten Wert in ein Pseudoregister laden.

### Bearbeitung von Befehlen

Die Befehle sind die direkt in Maschinencode umsetzbaren Wörter. Die hier erzeugten Maschinenbefehle arbeiten mit Pseudoregister, die erst bei der Registerbelegung in Maschinenregister umgesetzt werden.

**Einbau der Lese-Schreib- und Schreib-Schreib–Abhängigkeiten**

Zwischen Speicherzugriffen mit den Wörtern `@ (` *a-addr* -- *x* `)` oder `! (` *x a-addr* -- `)` ) besteht eine Abhängigkeit. Diese Lese-Schreib- oder Schreib-Schreib–Abhängigkeiten werden in den Datenflußgraph eingetragen.

Bei den Stackelementen besteht auch eine Lese-Schreib–Abhängigkeit. Diese Abhängigkeiten werden am Ende eines Grundblockes erzeugt.

### 3.2.2 Beispiel für Befehlsauswahl

Der Befehle zum Abspeichern des berechneten Wertes auf dem Datenstack, wenn das neue Wort ausgeführt wird, wird am Ende des Grundblockes in den Datenflußgraphen eingebaut. Das eventuelle Angleichen des Datenstackpointers an den veränderten Datenstack würde auch hier eingebaut werden. In unserem Beispiel wird dieser Befehl nicht benötigt, wodurch sich eine Einsparung eines Befehles ergibt. Zu diesem Zeitpunkt werden aber auch Lese-Schreib–Abhängigkeiten mit zusätzlichen Kanten im Graphen eingebaut. Bei diesem Beispiel ist nur eine Lese-Schreib–Abhängigkeit vorhanden. Die Abhängigkeit in diesem Beispiel ist redundant.

Beim einfachen Algorithmus, der im Maschinencode-Übersetzer umgesetzt ist, werden die Befehle des Datenflußgraphen in Maschinebefehle umgewandelt. Dabei wird in den Datenflußknoten der entsprechende Maschinenbefehl in das Feld `node_instr` eingetragen.

Die Situation im Datenflußgraphen nach der Befehlsauswahl ist in Abbildung 3.2.1 (nächste Seite) dargestellt. Durchgehende Linien gegeben dabei die ermittelten Datenfluß-Abhängigkeiten an. Strichlierte Linien gegeben dabei die ermittelten Lese-Schreib–Abhängigkeiten an.

## 3.3 Befehlsanordnung

Die Umformung des Datenflußgraphen eines Grundblockes in eine lineare Form, wie sie der Prozessor erwartet, ist die Aufgabe der Befehlsanordnung (*instruction scheduling*.) Bei Prozessoren, die durch Pipelining und/oder superskalare Architektur Befehle parallel abarbeiten, ist das Ziel der Befehlsanordnung eine gute Ausnutzung des Parallelismus durch Minimnierung von ungenutzten Wartezyklen (*delay slot*, *pipeline bubble*).

Auf den *MIPS RISC Prozessoren* kann ein Zyklus nach dem Start eines Ladebefehls ein anderer Befehl ausgeführt werden. Das Ergebnis des Ladebefehls steht aber erst nach zwei Zyklen zur Verfügung (*delay slot*). Wird ein Befehl gestartet, der das Ergebnis verwendet, dann muß er so lange warten und es entstehen ungenutzte Wartezyklen. Wenn die Befehle so angeordnet werden, daß nach dem Ladebefehl ein Befehl folgt, der das Ergebnis nicht verwendet, wird der Prozessor optimal ausgenutzt.

Ein verwandtes Problem gibt es auch bei Verzweigungsbefehlen in einigen RISC-Architekturen. Bei den *MIPS RISC Prozessoren* wird ein Verzweigungsbefehl erst mit

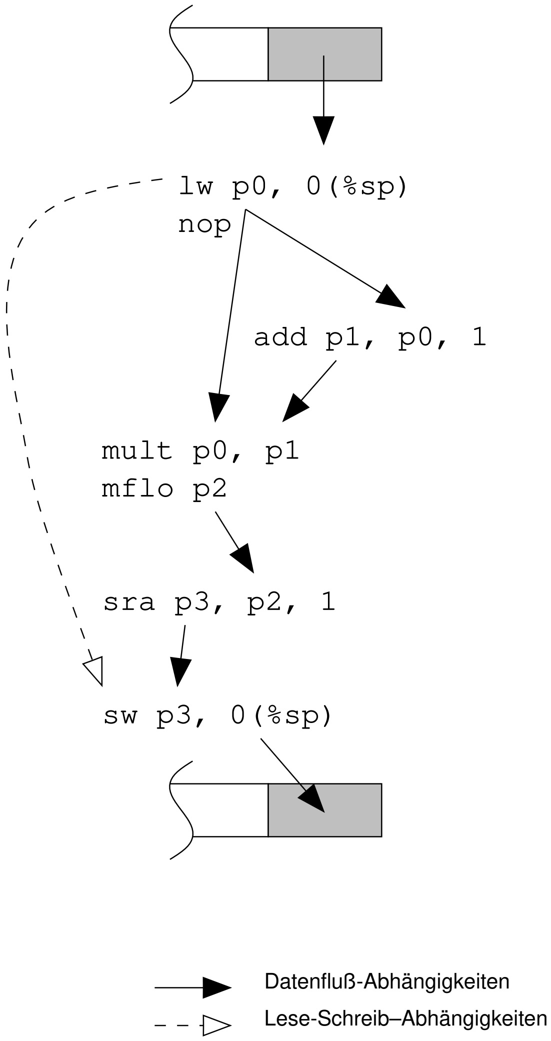


Abbildung 3.2.1: Wort: `Zahlen-Summe` (Befehlsauswahl)

Verspätung ausgeführt. Vorher wird noch der unmittelbar hinter dem Verzweigungsbefehl liegende Befehl ausgeführt (*delayed branch*). Dieser Branch-delay-slot muß natürlich mit einem sinvollen und unabhängigen Befehl gefüllt werden (evtl. mit einem Leerbefehl, bei *RISC MIPS Prozessoren* der Befehl `nop`).

Es stellt sich die Frage, welche Befehlsanordnungen überhaupt korrekt sind. Die Datenfluß-Abhängigkeiten allein sind noch nicht ausreichend. Wenn jedoch auch die Lese-Schreib-, die Schreib-Schreib- und die potentiellen Abhängigkeiten berücksichtigt werden, ist jede Befehlsanordnung korrekt.

### 3.3.1 Umsetzung der Befehlsanordnung

Die Befehlsanordnung wird momentan mit einem einfachen Algorithmus erzeugt. Es werden auf Grund der aufgebauten Abhängigkeiten die Maschinenbefehle in der entsprechenden Reihenfolge erzeugt. Es werden auch Befehle eliminiert, die das n. Datenstackelement lesen und wieder auf die selbe Position im Datenstack speichern.

Nach Ladebefehlen werden `nop` Befehle eingebaut um den Delay-slot zu füllen.

Da die Kontrollstrukturen mit kurzen Maschinencodeteilen erzeugt werden, wurden diese so konstruiert, daß nach Verzweigungen die Branch-delay-slots soweit wie möglich mit sinnvollen Maschinenbefehen gefüllt werden (siehe Abschnitt 3.6 Kontrollstrukturen, Seite 35).

#### Bearbeitung der Linkcounts

Um eine lokale Optimierung zu erhalten, werden mehrmals verwendete Ergebnisse in den Registern weiterverwendet. Dadurch werden gesamte Teilbäume des Gesamtbaumes nicht mehrmals in Maschinencode übersetzt.

### 3.3.2 Beispiel für Befehlsanordnung

Auf den im vorherigen Schritt erzeugten Datenabhängigkeitsgraphen wird eine Befehlsanordnung angewendet. Die eingebrachten Abhängigkeiten des Graphen geben die Informationen für die Anordnung der Befehle an.

Die ermittelte Befehlsanordnung des Wortes `Zahlen-Summe` ist in Abbildung 3.3.1 (nächste Seite) zu sehen.

## 3.4 Registerbelegung

Moderne Übersetzer erzeugen zunächst Code, der beliebig viele Pseudoregister verwendet. Die Aufgabe der Registerbelegung (*register allocation*) ist es, die Pseudoregister möglichst alle in Maschinenregister unterzubringen, bzw. wenn das nicht gelingt, weniger wichtige Pseudoregister in den Speicher auszulagern. Im Gegensatz zur Befehlsauswahl und der Befehlsanordnung, die auf der Grundblockebene ("lokal") arbeiten, erfolgt die Registerbelegung heute für eine ganze Prozedur ("global").

```
lw p0, 0(%sp)
nop

add p1, p0, 1

mult p0, p1
mflo $4

sra p3, p2, 1

sw p3, 0(%sp)
```

Abbildung 3.3.1: Wort: `Zahlen-Summe` (Befehlsanordnung)

Eine weitere Optimierung der Registerbelegung ist eine “interprozedurale” Registerbelegung.

### 3.4.1 Umsetzung der Registerbelegung

Es wird zur Zeit nur ein einfacher Algorithmus zur Registerbelegung verwendet. Die Registerbelegung arbeitet lokal.

Um eine lokale Optimierung zu erhalten, werden mehrmals verwendete Ergebnisse in den Maschinenregistern weiterverwendet. Dabei wird vom Maschinencode-Übersetzer das Register gesperrt, bis das Ergebnis im Maschinenregister nicht mehr gebraucht wird. Die Anzahl der Verwendungen wird dem Linkcount entnommen.

#### Belegung der Maschinenregister

Von den 32 Maschinenregistern, die von den *MIPS RISC Prozessoren* zur Verfügung gestellt werden, haben einige reservierte Bedeutungen. Z.B. liefert das Maschinenregister `$0` immer den Wert 0, das Maschinenregister `%ra` (`$31`) wird zum Speichern der Rücksprungadresse verwendet oder den gesicherten Registern, `%s0`...`%s8`, deren Werte interprozedural erhalten bleiben müssen.

Zur Bearbeitung der Stacks (Daten- und Rücksprungstack) werden die Stackpointer (Daten- und Rücksprungstackpointer) in reservierten Maschinenregistern gespeichert (siehe Tabelle 3.4.1 (nächste Seite)). Auch die vom inneren Interpreter benötigten Werte IP und CFA werden in reservierten Maschinenregistern gespeichert.

Es stehen 16 frei verfügbare Maschinenregister zur Verfügung. Im momentanen Status des Maschinencode-Übersetzers werden keine Pseudoregister in den Speicher ausgelagert, wenn es keine freien Maschinenregister mehr gibt. Durch die große Anzahl von Maschinenregistern kommt so eine Situation in der Praxis nie vor.

| Pseudoregister | Maschinenregister | Bedeutung |
|---|---|---|
| %sp | $17 | Datenstackpointer |
| %rp | $18 | Rücksprungstackpointer |
| %ip | $16 | IP |
| %cfa | $20 | CFA |
| p0...p15 | $2...$15<br>$24...$25 | frei verfügbare Maschinenregister |

Tabelle 3.4.1: Belegung der Maschinenregister

```
lw $2, 0(%sp)
nop

add $3, $2, 1

mult $2, $3
mflo $4

sra $2, $4, 1

sw $2, 0(%sp)
```

Abbildung 3.4.1: Wort: `Zahlen-Summe` (Registerbelegung)

### 3.4.2 Beispiel für Registerbelegung

Im nächsten Übersetzungsschritt des Wortes `Zahlen-Summe` wird eine Registerbelegung durchgeführt. Die benötigten Pseudoregister werden dabei an frei verfügbare Maschinenregister vergeben.

Die ermittelten Registerbelegungen des Wortes `Zahlen-Summe` sind in Abbildung 3.4.1 (diese Seite) zu sehen.

## 3.5 Wortaufrufe

### 3.5.1 Aufbau von Maschinencode-Wörter

Ab der PFA wird der Code `init` verwendet. Dabei wird die Rücksprungadresse auf dem Rücksprungstack zwischengespeichert. Beim Verlassen des Maschinencode-Wortes wird der `exit`-Code ausgeführt. Dabei wird die Rücksprungadresse wieder vom Rücksprungstack gelesen und das Programm wird an dieser Stelle fortgesetzt.

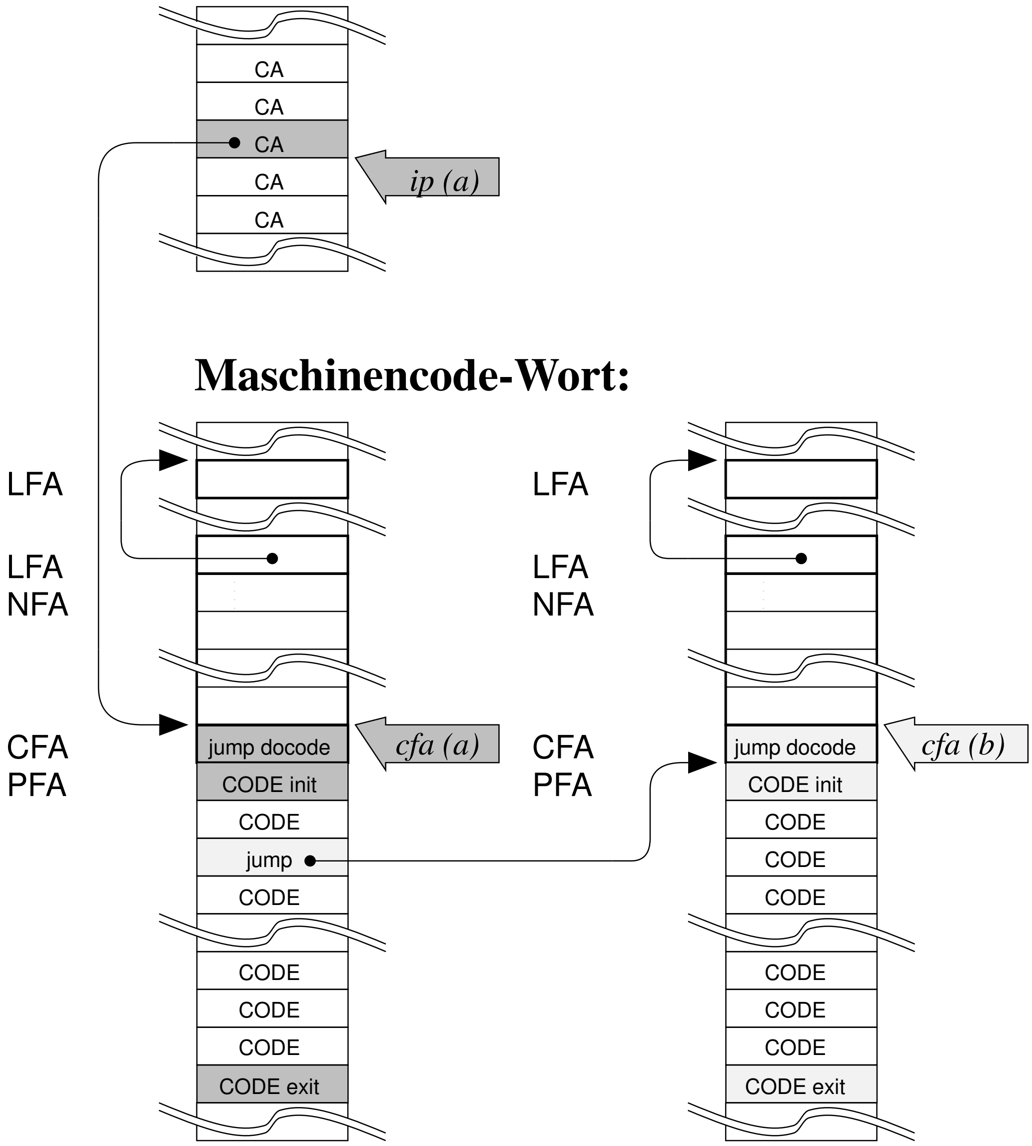


Abbildung 3.5.1: Verknüpfungsstruktur: Maschinencode

```
CODE init
  sw %ra, -4(%rp)
  addiu %rp, %rp, -4

CODE exit
  lw %ra, 0(%rp)
  addiu %rp, %rp, 4
  jr %ra
  nop
```

Source 3.5.1: Aufbau von Maschinencode-Wörtern (Teil 1)

Der Grund, warum hier eine solche Aufteilung gemacht wird, liegt darin, daß bei Wortaufrufen ein Maschinencode-Wort einfach mit einem Sprung auf die PFA aufgerufen werden kann.

In Source 3.5.1 (diese Seite) werden die Routinen `init` und `exit` in Maschinencode dargestellt.

### 3.5.2 Beispiel für ein Maschinencode-Wort

Nach erfolgreicher Befehlsauswahl, Befehlsanordnung und Registerbelegung ist das Wort `Zahlen-Summe` im Speicher assembliert und sofort ausführbar.

Der gesamte Maschinencode des Wortes `Zahlen-Summe` ist in Abbildung 3.5.2 (nächste Seite) zu sehen. Es sind hier auch die Positionen der beiden Programmteile `init` und `exit`, die für den Programmablauf zuständig sind, angeführt (Sie werden hier in einer allgemeineren Pseudosprache, mit `push` und `pop` gezeigt.). Bei den weiteren Wörtern werden die beiden Programmteile nicht mehr explizit angeführt, werden aber bei der Ausführung natürlich benötigt.

### 3.5.3 Aufrufe innerhalb des Maschinencodes

Durch den Aufbau der Maschinencode-Wörter können diese Wörter direkt mit dem Sprung auf die PFA angesprochen werden. Zuerst wird der kurze `init`-Code (siehe Source 3.5.1 (diese Seite)) ausgeführt. Anschließend wird Maschinencode ausgeführt, der fast keinen Einschränkungen unterliegt. Bei der Verwendung von Maschinenregistern haben einige eine besondere Funktion und dürfen deshalb nicht uneingeschränkt verwendet werden (siehe Tabelle 3.4.1 (Seite 32)). Beim Verlassen des Maschinencode-Wortes wird der `exit`-Code (siehe Source 3.5.1 (diese Seite)) ausgeführt.

Abbildung 3.5.3 (Seite 36) zeigt eine graphische Darstellung des Aufrufes von Maschinencode-Wörtern aus Maschinencode-Wörtern.

**init**

| Pseudo-Code | Maschinencode |
|---|---|
| `push %rp := %ra` | `sw %ra, -4(%rp)` |
| | `addiu %rp, %rp, -4` |

**Maschinencode des Wortes**

| Forth-Code | Maschinencode |
|---|---|
| `dup` | `lw $2, 0(%sp)` |
| | `nop` |
| `1+` | `add $3, $2, 1` |
| `*` | `mult $2, $3` |
| | `mflo $4` |
| `2/` | `sra $2, $4, 1` |
| | `sw $2, 0(%sp)` |

**exit**

| Pseudo-Code | Maschinencode |
|---|---|
| `%ra := pop %rp` | `lw %ra, 0(%rp)` |
| | `addiu %rp, %rp, 4` |
| `jump %ra` | `jr %ra` |
| | `nop` |

Abbildung 3.5.2: Wort: `Zahlen-Summe`

### 3.5.4 Beispiel für Wortaufrufe

In diesem Beispiel werden die Einbauten von Wortaufrufen gezeigt.

Die Native Code Worte werden durch einen Sprungbefehl auf die PFA des Wortes aufgerufen. Dabei wird die Rücksprungadresse in ein Maschinenregister gespeichert, die beim `init` Code auf den Rücksprungstack geschrieben wird. Beim Verlassen des Wortes wird der `exit` Code ausgeführt, der wieder die Rücksprungadresse vom Rücksprungstack liest und auf der richtigen Stelle fortfährt.

Der gesamte Maschinencode des Wortes `Drucke-Zahlen-Summe` ist in Abbildung 3.5.4 (Seite 38) zu sehen.

## 3.6 Kontrollstrukturen

Zum Abarbeiten von Kontrollstrukturen wird ein eigener Stack verwendet, der Kontrollflußstack. Er muß nicht unbedingt physisch vorhanden sein, da der Datenstack während des Übersetzungsvorganges im allgemeinen nicht verwendet wird und daher als Kontrollflußstack verwendet werden kann. Im Maschinencode-Übersetzer wird der Daten-

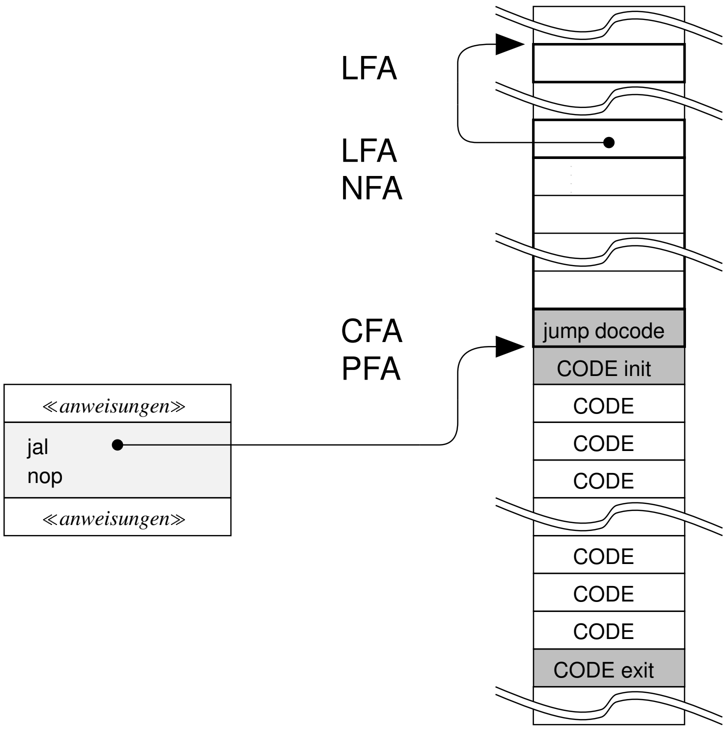


Abbildung 3.5.3: Wortaufrufe: von Maschinencode-Wörtern

```
: Drucke-Zahlen-Summe ( n -- )
  dup . Zahlen-Summe . cr ;

100 Drucke-Zahlen-Summe
1005050
```

Source 3.5.2: Beispiel: Drucke-Zahlen-Summe

stack aber auch während der Übersetzung von Maschinencode-Wörtern verwendet. Dadurch wird vom Übersetzer ein eigener Kontrollflußstack verwendet.

Die Kontrollstrukturen können auf eine kleine Menge von Wörter (Basiskontrollstrukturwörter) reduziert werden, die Maschinencode erzeugen sollen. Es handelt sich dabei um die Wörter `if`, `ahead`, `then` (auch `endif` genannt), `begin`, `again` und `until`.

Aufbauend auf diese Basiskontrollstrukturwörter und den Kontrollflußstackwörtern (`cs-pick` und `cs-roll`) können alle weiteren Kontrollstrukturwörter erzeugt werden.

### 3.6.1 Bedingte Sprünge

Das Wort `if` stellt ein Basiskontrollstrukturwort dar, das Maschinencode erzeugt (siehe Abbildung 3.6.1 (Seite 39)). Die Maschinenbefehle umfassen ein Laden des TOS Elementes und ein Vergleich dieses Elementes auf Nichterfüllen der Bedingung, was ein Überspringen der Anweisungen bewirken würde. Es wird dabei eine Adresse am Kontrollflußstack gespeichert, die auf den Sprungbefehl zeigt. Zu diesem Zeitpunkt ist die Zieladresse noch nicht bekannt. Das Wort `endif` löst dann diesen Sprungbefehl mit der entsprechenden Zieladresse auf.

Beim Wort `else` wird ein neuer Sprungbefehl ohne einer Zieladresse, mit Hilfe des Wortes `ahead`, erzeugt. Dabei wird auch wieder eine Adresse am Kontrollflußstack gespeichert, die auf diesen Sprungbefehl zeigt. Die vorherige Adresse am Kontrollflußstack wird nun mit dem Wort `endif` aufgelöst. Die neue Zieladresse wird am Ende mit einem `endif` vervollständigt.

Das Wort `else` hat dabei auf den Basiskontrollstrukturwörtern aufgesetzt.

Die `case` Struktur stellt eine geschachtelte `if` Struktur dar (siehe Abbildung 3.6.2 (Seite 40)). Dabei werden schrittweise alle Wertemöglichkeiten der `case` Struktur durchlaufen. Der letzte `else` Zweig stellt den Defaultbereich dar. Beim `endcase` Wort müssen dann alle `if` Strukturen abgeschlossen werden.

Alle in der `case` Struktur verwendeten Wörter setzen dabei auf den Basiskontrollstrukturwörtern auf.

| Forth-Code | Maschinencode |
|---|---|
| dup | lw $2, 0(%sp)<br>nop<br>sw $2, -4(%sp)<br>addiu %sp, %sp, -4 |
| . | jal %ra <(PFA) .><br>nop |
| Zahlen-Summe | jal %ra <(PFA) Zahlen-Summe><br>nop |
| . | jal %ra <(PFA) .><br>nop |
| cr | jal %ra <(PFA) cr><br>nop |

Abbildung 3.5.4: Wort: `Drucke-Zahlen-Summe`

### 3.6.2 Einfache Schleifen

Die `begin` ...`again` Schleife stellt eine Endlosschleife dar (siehe Abbildung 3.6.3 (Seite 40)). Das Wort `begin` speichert dabei die aktuelle Adresse des Maschinencodes am Kontrollflußstack. Beim Erreichen des Wortes `again` wird diese Adresse vom Kontrollflußstack gelesen und als Zieladresse für den bedingungslosen Sprungbefehl verwendet.

Die `begin` ...`until` Schleife stellt eine Schleife dar, deren Anweisungen mindestens einmal ausgeführt werden, weil die Abbruchbedingung am Ende der Schleife überprüft wird (siehe Abbildung 3.6.4 (Seite 41)). Das Wort `begin` speichert wieder die aktuelle Adresse des Maschinencodes am Kontrollflußstack. Beim Erreichen des Wortes `until` wird diese Adresse vom Kontrollflußstack gelesen und als Zieladresse für den bedingten Sprungbefehl verwendet. Die Maschinenbefehle umfassen ein Laden des TOS Elementes und einen Vergleich dieses Elementes auf Nichterfüllen der Bedingung, was ein erneutes Ausführen der Anweisungen bewirken würde.

Die `begin` ...`while` ...`repeat` Schleife stellt eine Schleife dar, deren Abbruchbedingung am Anfang überprüft wird. Dadurch können die Anweisungen in der Schleife auch nullmal ausgeführt werden (siehe Abbildung 3.6.5 (Seite 41)).

Die Wörter `while` und `repeat` setzen auf den Basiskontrollstrukturwörtern auf.

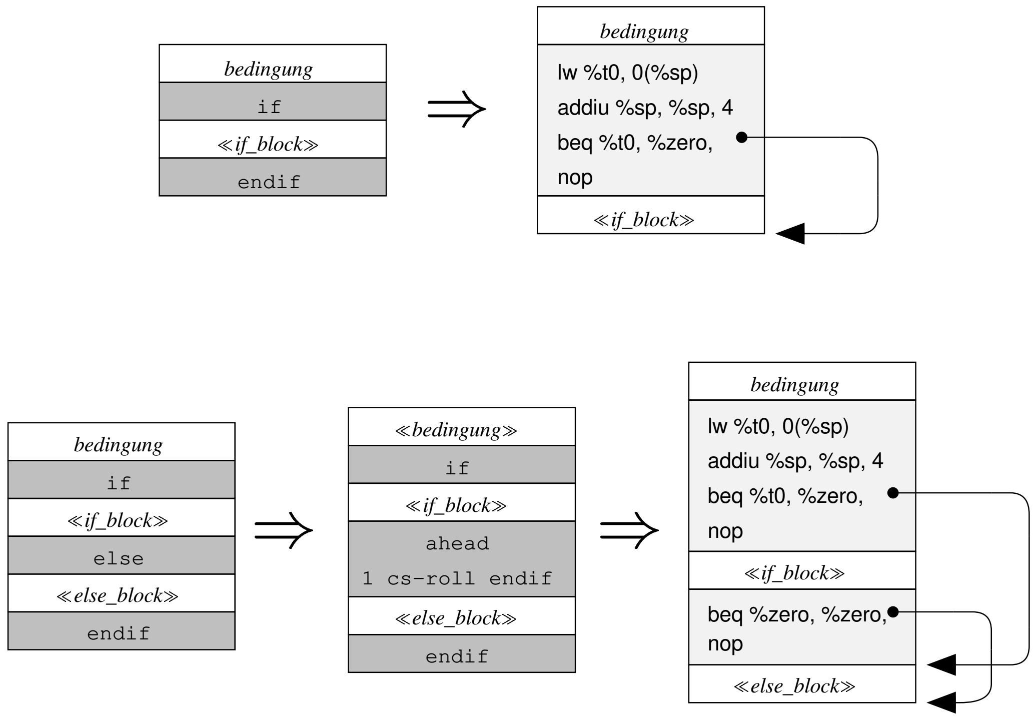


Abbildung 3.6.1: Kontrollstruktur: `if` ...`endif`; `if` ...`else` ...`endif`

### 3.6.3 Zählschleifen

Das Wort `do` schiebt die *von* und *bis* Werte ohne zu überprüfen auf den Rücksprungstack (siehe Abbildung 3.6.6 (Seite 42)). Beim Wort `?do` werden zusätzlich die Werte auf Gleicheit geprüft, was eine sofortige Beendigung der Zählschleife mit dem Wort `leave` bewirken würde (siehe Abbildung 3.6.7 (Seite 43)).

`loop` inkrementiert den aktuellen Zähler um eins und vergleicht dann mit dem *bis*-Wert. Bei Gleichheit wird die Zählschleife beendet. Bei `+loop` muß etwas komplizierter überprüft werden, ob die Zählschleife zu Ende ist. Da die Schrittweite dieser Zählschleife sowohl positiv, als auch negativ sein kann, muß hier überprüft werden, ob der Grenzwert *bis* von einer Seite überschritten wurde.

Ein `do` oder `?do` kann beliebig mit einem `loop` oder `+loop` kombiniert werden.

Die Wörter `do`, `?do`, `loop` und `+loop` setzen dabei auf den Basiskontrollstrukturwörtern auf.

### 3.6.4 Beispiel für Kontrollstrukturen

In diesem Beispiel wird eine einfache Schleife verwendet. Beim Eintritt in die Schleife (`begin`) wird die Zieladresse für den späteren Sprungbefehl auf den Kontrollflußstack

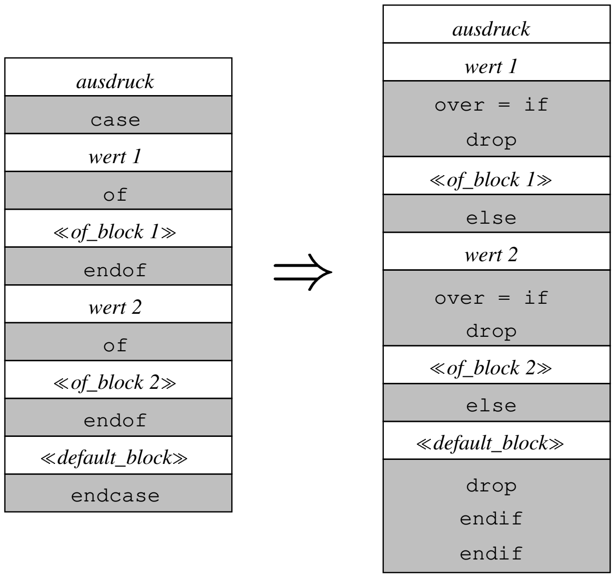


Abbildung 3.6.2: Kontrollstruktur: `case` ...`of` ...`endof` ...`endcase`

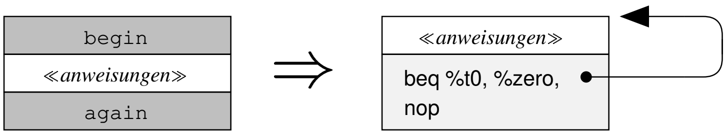


Abbildung 3.6.3: Kontrollstruktur: `begin` ...`again`

geschrieben. Anschließend werden die Grundblöcke innerhalb der Schleife übersetzt. Am Ende der Schleife (`until`) wird der Maschinencode dieses Kontrollstrukturwortes erzeugt. Dabei wird die Zieladresse vom Kontrollflußstack zur Auflösung des Sprungbefehles verwendet.

Bei allen anderen Kontrollstrukturwörtern verläuft die Maschinencodeerzeugung auf die selbe Art. Bei Kontrollstrukturwörtern, die aus Basiskontrollstrukturwörtern zusammengesetzt sind, kann es auch der Fall sein, daß einige Befehle im Rahmen des verherigen Grundblockes übersetzt werden (z.B. bei Zählschleifen).

Der gesamte Maschinencode des Wortes `Drucke-Zahlen-Summen` ist in Abbildung 3.6.8 (Seite 44) zu sehen.

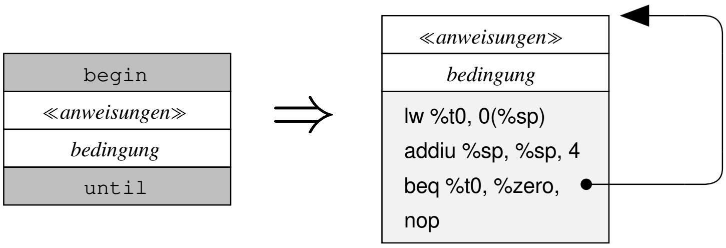


Abbildung 3.6.4: Kontrollstruktur: `begin` ...`until`

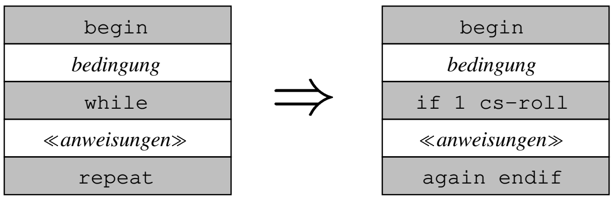


Abbildung 3.6.5: Kontrollstruktur: `begin` ...`while` ...`repeat`

## 3.7 Integration mit dem Interpretierten System

Ein Forth-Sytem muß eine einheitliche Verknüpfungsart aufweisen. Beim `gforth`-Übersetzer besteht die Möglichkeit, ein direkt verknüpftes oder ein indirekt verknüpftes System zu erstellen. Dadurch muß der vom Maschinencode-Übersetzer erzeugte Code beide Verknüpfungsarten unterstützten. Im Maschinencode-Übersetzer wird eine Konstante verwendet, die die von Maschinencode verwendete Verknüpfungsart angibt. Diese Verknüpfungsart muß mit der vom `gforth` übereinstimmen.

Der Aufbau der Maschinencode-Wörter ist ein Mischung von Wörtern in der Verknüpfungsart, die von `gforth` verwendet wird, und subroutinen verknüpften Wörtern.

Die Verknüpfungsart von `gforth` wird benötigt, damit der Interpreter die Maschinencode-Wörter auch aufrufen kann. Dazu ist die Einstiegsroutine `docode` notwendig. Von dieser Routine wird die CFA und der IP entsprechend der Verknüpfungsart bearbeitet (siehe Abschnitt 2.5.2 direkt verknüpft, Seite 18 und Abschnitt 2.5.3 indirekt verknüpft, Seite 18).

In Source 3.7.1 (Seite 45) wird die Routinen `docode` in Maschinencode dargestellt.

### 3.7.1 Aufrufe mit Threaded- und Maschinencode

Zum Unterschied zu Maschinencode-Wörtern müssen hier einige weitere Vorkehrungen getroffen werden, damit das Interpreter Wort auch richtig aufgerufen wird.

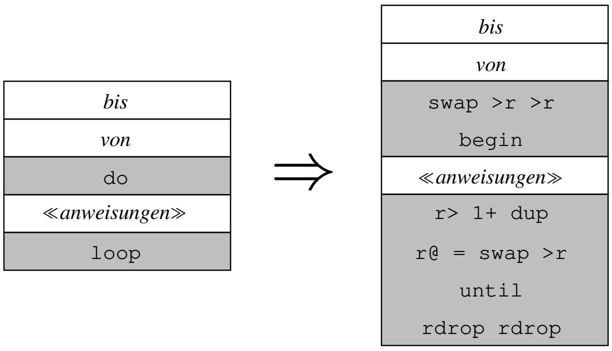


Abbildung 3.6.6: Kontrollstruktur: `do` ... `loop`

Zuerst ist es notwendig, `%cfa` und `%ip` mit den entsprechenden Werten zu belegen. Die `%cfa` wird auf die CFA des aufzurufenden Wortes gesetzt. Der `%ip` wird auf die Adresse hinter dem Sprungbefehl gesetzt. Anschließend wird der Sprungbefehl auf die CFA ausgführt. Hinter dem Sprungbefehl steht die Adresse, wo sich der weiterführende Code nach dem Aufruf des Interpreterwortes befindet.

Durch die so erstellten Aufrufe von Interpreterwörter, ist gewährleistet, daß auch Wörter verwendet werden können, die nicht, oder nur sehr schwer als Maschinencode-Wörter erzeugt werden können.

Abbildung 3.7.1 (Seite 45) zeigt eine graphische Darstellung des Aufrufes von Interpreter Wörtern aus Maschinencode-Wörtern.

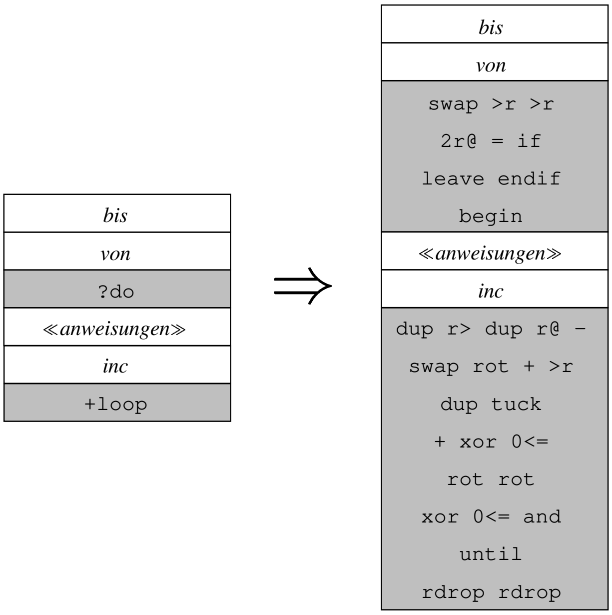


Abbildung 3.6.7: Kontrollstruktur: `?do` …`+loop`

```
: Drucke-Zahlen-Summen ( bis -- )
  1 begin
    dup Drucke-Zahlen-Summe
    1+
    2dup < until
  2drop ;

10 Drucke-Zahlen-Summen
11
23
36
410
515
621
728
836
945
1055
```

Source 3.6.1: Beispiel: Drucke-Zahlen-Summen

| Forth-Code | Maschinencode |
|---|---|
| `1` | `li $2, 1` |
| | `sw $2, -4(%sp)` |
| | `addiu %sp, %sp, -4` |
| `begin` | `label0:` |
| `dup` | `lw $2, 0(%sp)` |
| | `nop` |
| | `sw $2, -4(%sp)` |
| | `addiu %sp, %sp, -4` |
| `Drucke-Zahlen-Summe` | `jal %ra <(PFA) Drucke-Zahlen-Summe>` |
| | `nop` |
| | `lw $2, 0(%sp)` |
| | `nop` |
| `1+` | `addiu $3, $2, 1` |
| | `sw $3, 0(%sp)` |
| `2dup` | `lw $4, 4(%sp)` |
| | `nop` |
| `<` | `slt $2, $4, $3` |
| | `sub $2, %zero, $2` |
| | `sw $2, -4(%sp)` |
| | `addiu %sp, %sp, -4` |
| `until` | `lw %t0, 0(%sp)` |
| | `addiu %sp, %sp, 4` |
| | `beq %t0, %zero, label0` |
| | `nop` |
| `2drop` | `addiu %sp, %sp, 8` |

Abbildung 3.6.8: Wort: `Drucke-Zahlen-Summen`

```
CODE docode
  addiu %cfa, %cfa, 8
  sw %ip, -4(%rp)
  jalr %cfa, %ra
  addiu %rp, %rp, -4

  lw %ip, 0(%rp)
  addiu %rp, %rp, 4
  lw %cfa, 0(%ip)
  addiu %ip, %ip, 4
  jr %cfa
  nop
```

Source 3.7.1: Aufbau von Maschinencode-Wörtern (Teil 2)

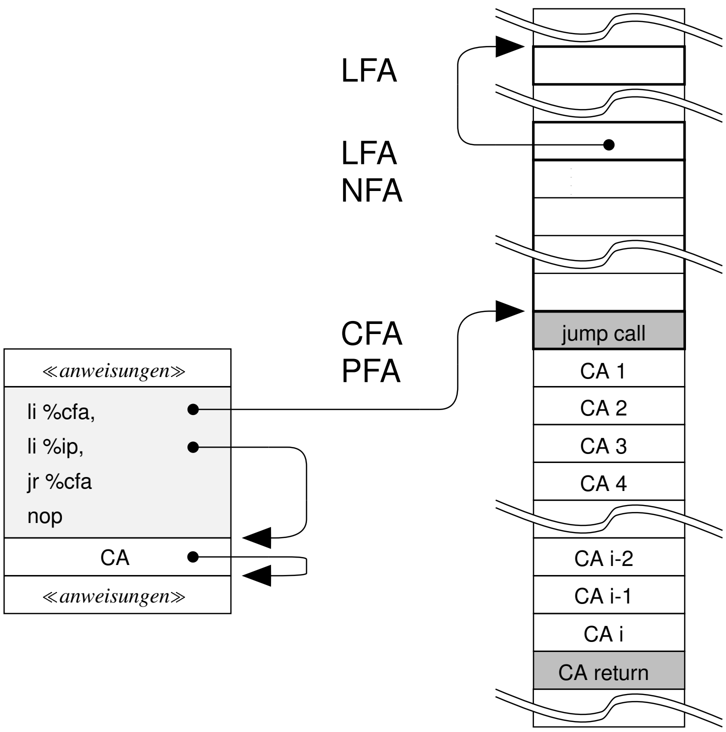


Abbildung 3.7.1: Wortaufrufe: von Interpreter Wörtern

# Kapitel 4

# Implementationdetails

Zur besseren Erklärung der Zusammenhänge erfolgt eine Beschreibung aller für den Übersetzer notwendigen Speicherbereiche und Datenstrukturen.

## 4.1 Bedeutung der verwendeten Wortlisten

Um den Übersetzer strukturiert aufzubauen, werden mehrere Wortlisten verwendet:

**Wortliste `voc_source`:** beinhaltet alle allgemein verfügbaren Wörter. Dazu gehören die Wörter der Datenstrukturen, Arrays, lineare Listen und binäre Bäume, die Wörter zum Assemblieren und Disassemblieren von Maschinenbefehlen und alle zusätzlich benötigten Wörter zum Übersetzen.

**Wortliste `voc_target_compile`:** beinhaltet alle zum Übersetzunszeitpunkt zusätzlich sichtbaren Wörter. Es handelt sich dabei um alle vom Übersetzer direkt verarbeitbaren Wörter (z.B. `+`, `-` oder Stackwörter wie `dup` und `drop`) und Kontrollstrukturwörter (bedingte Sprünge, einfache Schleifen und Zählschleifen).

**Wortliste `voc_target`:** beinhaltet nach dem Übersetzen des Übersetzers nur die Wörter **:**, `]` und `bye`. Sichtbar wird die Wortliste erst beim fertig übersetzten Übersetzer. Beim weiteren Übersetzen von Wörtern, werden sie dann an diese Wortliste angehängt.

Die Aufteilung in diese Wortlisten hat folgende Bedeutung. Die Wörter der Wortliste `voc_source` sind der Übersichtlichkeit wegen in einer getrennten Wortliste. Sie könnten aber auch mit der vorher als `get-current` sichtbaren Wortliste vereinigt werden.

Alle vom Maschinencode-Übersetzer direkt verarbeitbaren Wörter stehen in der Wortliste `voc_target_compile`. Der Grund liegt darin, daß die Wörter der Wortliste `voc_target_compile` in der Übersetzungsphase auf einem völlig anderen Datenaufbau am Datenstack aufsetzten, wie in den folgenden Abschnitten erklärt wird. Alle Wörter haben Interpreterwörter die in der Interpreterphase verwendet werden können.

```
struct
    1 cells: field node_type
    1 cells: field node_inst
    1 cells: field node_val
    1 cells: field node_pointer
    1 cells: field node_offset
    1 cells: field node_done
    1 cells: field node_link
    1 cells: field node_reg
    1 cells: field node_depends
end-struct node_struct
```

Source 4.2.1: Aufbau der `node`-Struktur

Die in dieser Wortliste vorkommenden Kontrollstrukturwörter dürfen nur in der Übersetzungsphase verwendet werden.

Die Wortliste `voc_target` beinhaltet Wörter, die in der Interpreterphase auch verwendet werden können. Das darin enthaltene Wort **:** startet das Übersetzen von Maschinencode-Wörtern. Damit aber neue Wörter erst mit dem Maschinencode-Übersetzer übersetzt werden, wenn er vollständig erzeugt wurde, wird diese Wortliste erst am Ende der Übersetzung des Übersetzers sichtbar.

Durch die so modular gestaltete Programmstruktur läßt sich der Übersetzer auch unproblematisch, was die Sichtbarkeit der einzelnen Wörter betrifft, übersetzten. In weiterer Folge ist der Übersetzer dann auch noch, nur durch die physischen Speichergrenzen beschränkt, weiter von sich selbst mehrmals übersetzbar.

## 4.2 Aufbau der `node`-Struktur

Die `node`-Struktur beinhaltet alle Information, die ein Knoten des Datenflußgraphen benötigt. In den unterschiedlichen Phasen des Übersetzers werden die Werte erst initialisiert und dann verwendet oder an die neuen Gegebenheiten angeglichen (z.B. Angleich der Lese-Schreib- oder Schreib-Schreib–Abhängigkeit).

In Source 4.2.1 (diese Seite) werden die Positionen und Größenangaben der einzelen Werte gezeigt. Die Bedeutung der Werte ist folgendermaßen gegeben:

`node_type`**:** damit wird die Kennung der Art des Knotens gespeichert. Eine Liste der möglichen Werte ist nachfolgend aufgelistet:

| | |
|---|---|
| `n_literal` | Literals |
| `n_id@` | Stackelemente (Datenstack- oder Rücksprungstackelemente) lesen |
| `n_id!` | Stackelemente (Datenstack- oder Rücksprungstackelemente) schreiben |
| `n_uop` | unäre Operatoren mit Ergebnis |
| `n_uopn` | unäre Operatoren ohne Ergebnis |
| `n_op` | binäre Operatoren mit Ergebnis |
| `n_opn` | binäre Operatoren ohne Ergebnis |

Es handelt sich hierbei um die Ausführungstoken (*xt*) von Wörtern, die in der Befehlsanordnung zur Anwendung kommen.

`node_inst`: Es handelt sich hierbei um die Adresse von Wörter, die in der Befehlsanordnung die Assemblierung der Maschinenbefehle durchführen.

`node_val`: (Diese Zelle wird nur bei `n_literal` benützt.) der Wert eines Literals wird hier gespeichert.

`node_pointer`: (Diese Zelle wird nur bei `n_id@` und `n_id!` benützt.) das Register, in dem der Pointer des entsprechenden Stacks (Datenstackpointer oder Rücksprungstackpointer) gespeichert ist, wird hier gespeichert.

`node_offset`: (Diese Zelle wird nur bei `n_id@` und `n_id!` benützt.) in Kombination mit `node_pointer` wird hier der Offset bezogen zum Pointer des entsprechenden Stacks (Datenstackpointer oder Rücksprungstackpointer) gespeichert.

`node_done`: beinhaltet das Flag, das angibt, ob der Befehl in der Befehlsanordnung schon abgearbeitet wurde. Das wird benötigt, um eine Abhängigkeitsprüfung in der Befehlsanordnung durchführen zu können.

`node_link`: der Zähler für mehrmaliges Verwenden des Knotens wird hier gespeichert. Der Wert ist `n-1` für `n`-malige Verwendung.

`node_reg`: das verwendetes Register für das Ergebnis wird hier gespeichert. Mit dem Wert 0 wird die Zelle initialisiert.

`node_depends`: beinhaltet eine Liste von (Lese-Schreib- oder Schreib-Schreib-) Abhängigkeiten. Zur Initialisierung wird eine leere Liste verwendet, um in der weiteren Verarbeitung eine einfachere Abarbeitung zu gewährleisten.

## 4.3 Aufbau des Datenstacks beim Übersetzen

Am Datenstack liegen in der Übersetzungsphase `btree`-Knoten. Die Daten in diesen Knoten beinhalten `node`-Strukturen, die Informationen über das i-te Datenstackelement haben, oder einen Teilbaum des gesamten Datenflußgraphen des Grundblockes.

Die `btree`-Knoten, deren Daten das i-te Datenstackelement darstellen, werden dynamisch erzeugt. Diese Handhabung ist in der Übersetzungsphase etwas kompliziert, aber es können damit Wörter übersetzt werden, die eine uneingeschränkte Anzahl von Datenstackelementen verwenden. Eine andere Möglichkeit wäre ein Initialisieren des Datenstacks beim Eintritt in den Grundblock mit n `btree`-knoten. Damit wäre aber die Verwendung der Datenstackelement auf n Elemente beschränkt.

Wenn ein vom Maschinencode-Übersetzer direkt verarbeitbares Wort übersetzt wird, werden mehrere `btree`-Knoten zusammengefügt.

## 4.4 Aufbau des Rücksprungstacks (temporär) beim Übersetzen

Es wird hier ein temporärer Rücksprungstack verwendet, der mit n `btree`-Knoten (Aufbau siehe Abschnitt A.1.2 Aufbau der `btree`-Struktur Seite 68) beim Eintritt in einen Grundblock initialisiert wird. Die Daten in diesen Knoten beinhalten `node`-Strukturen die Informationen über das i-te Rücksprungstackelement haben. Da die Verwendung von Rücksprungstackelementen nicht sehr groß ist, kann vom Maschinencode-Übersetzer eine Einschränkung auf n Rücksprungstackelemente gemacht werden.

Wenn ein Element auf den Rücksprungstack gebracht wird, wird einfach der neue `btree`-Knoten in den temporären Rücksprungstack eingetragen. Umgekehrt wird einfach ein `btree`-Knoten vom Rücksprungstack genommen, wenn ein Element benötigt wird. Durch die Initialisierung wird gewährleistet, daß die richtigen Rücksprungstackelemente vom Rücksprungstack genommen werden, wenn nicht im lokalen Grundblock der `btree`-Knoten auf den Rücksprungstack gebracht wurde.

## 4.5 Aufbau des Kontrollflußstacks beim Übersetzen

Der Kontrollflußstack muß nicht unbedingt in einem Forth System vorhanden sein, da der Datenstack auch dazu verwendet werden kann, weil der Datenstack zur Übersetzungszeit sonst nicht verwendet wird. Da beim Maschinencode-Übersetzer aber der Datenstack auch in der Übersetzungszeit eine große Aufgabe zu tragen hat, wurde hier ein eigener Kontrollflußstack eingerichtet.

Die Daten, die auf den Kontrollflußstack gelegt werden sind einfach die von den Kontrollstrukturen benötigten Daten. Es handelt sich hierbei um die Adressen, die zum Auflösen von Sprüngen benötigt werden.

Um die Kontrollstrukturen richtig bearbeiten zu können, werden auch spezielle Wörter zur Verfügung gestellt. `>control`, `control@` und `control>` werden zum Übertragen eines Elementes vom Datenstack auf den Kontrollflußstack, zum Lesen des obersten Elementes des Kontrollflußstacks und zum Übertragen eines Elementes vom Kontrollflußstack auf den Datenstack verwendet. Das Wort `cs-pick` wird dazu benötigt, um das n-te Element vom Kontrollflußstack zu lesen. `cs-roll` wird zum Umordnen des Kontrollflußstacks verwendet.

## 4.6 Aufbau des Leavestacks beim Übersetzen

Der Leavestack wird dazu benötigt, um die `leave` Aufrufe in den Zählschleifen korrekt aufzulösen.

Dabei wird zur Initialisierung beim Eintritt in eine Schleife eine `0` auf den Leavestack geschrieben und anschließend werden alle Adressen, die ein `leave` erzeugt haben, auf den Leavestack geschrieben. Beim Ende der Schleife werden dann alle Adressen aufgelöst. Es besteht dadurch auch die Möglichkeit geschachtelte Zählschleifen zu verarbeiten.

# Kapitel 5

# Übersetzungsbeispiel

Zur Untersuchung der Übersetzungsergebnisse des Maschinencode-Übersetzer wird anhand der Umsetzung einer Formel der erzeugte Code betrachtet.

$$(a + b)^2 = a^2 + 2ab + b^2 \tag{5.1.1}$$

Es werden drei Implementierungen dieser Formel untersucht. Zur Bewertung werden keine Zeitmessungen durchgeführt, sondern anhand der erzeugten Maschinenbefehl werden Vergleiche hergestellt. Bei den *MIPS RISC Prozessoren* dauern die meisten Maschinenbefehle einen Zyklus lang.

Das hier gezeigte soll die momentan relative einfache Umsetzung von Forth in Maschinencode zeigen. Auch werden hier die damit verbundenen noch nicht sehr effizienten Übersetzungsalgorithmen aufgedeckt. Diese Tatsache wird vor allem aus den ersten beiden Beispielen hervorgehen. Das dritte Beispiel zeigt, daß man in Forth mit einfacheren Formeln oft die besseren Ergebnisse erhält.

## 5.1 Wort: `binomial1`

Das Wort `binomial1` stellt eine Umsetzung der Formel dar. Wie wir später noch in `binomial3` sehen werden können wir diese Formel natürlich auch direkter in Forth umsetzten.

Dennoch ist beim Wort `binomial1` zu sehen, wie effizient lokale Optimierungen vom Maschinencode-Übersetzer durchgeführt werden. Es werden hier alle Stackworte

| | |
|---|---|
| 2 | Befehle: `init` Code |
| 15 | Befehle: Wort `binomial1` (siehe Abbildung 5.1.1 (Seite 53)) |
| 4 | Befehle: `exit` Code |
| 21 | |

Tabelle 5.1.1: Maschinebefehle: `binomial1`

```
: binomial1 ( a b -- [a+b]^2 )
  2dup * 2*
  swap dup * +
  swap dup * + ;

3 5 binomial1 . cr
64

: sq ( n -- n^2 )
  dup * ;
: binomial2 ( a b -- [a+b]^2 )
  2dup * 2*
  swap sq +
  swap sq + ;

3 5 binomial2 . cr
64

: binomial3 ( a b -- [a+b]^2 )
  + dup * ;

3 5 binomial3 . cr
64
```

Source 5.1.1: Beispiel: Binomische Formel

wegoptimiert. Die Daten werden erst zu einem Zeitpunkt auf den Datenstack geschrieben, wenn es wirklich notwendig ist. Lokale Zwischenergebnisse, die beim `gforth` auch immer auf den Datenstack geschrieben werden, können hier über Register weiterverarbeitet werden ohne aufwendige Speicheroperationen durchzuführen. Auch wird der Datenstackpointer nur wenn es unbedingt notwendig ist an die neue Situation am Datenstack angeglichen.

Der vom Maschinencode-Übersetzer erzeugte Code würde für die Ausführung des Wortes `binomial1` 21 Befehle (siehe Tabelle 5.1.1 (vorige Seite)) benötigen.

## 5.2 Wort: `binomial2`

Eine typische Umsetzung der Formel in Forth würde eine Ausgliederung der Quadratfunktion in einem eigenen Wort darstellen.

Der vom Maschinencode-Übersetzer erzeugte Code würde für die Ausführung des

| Forth-Code | Maschinencode |
|---|---|
| | `lw $2, 4(%sp)` |
| | `nop` |
| | `lw $3, 0(%sp)` |
| | `nop` |
| `2dup *` | `mult $2, $3` |
| | `mflo $4` |
| `2*` | `sll $5, $4, 1` |
| `swap dup *` | `mult $3, $3` |
| | `mflo $4` |
| `+` | `add $3, $5, $4` |
| `swap dup *` | `mult $2, $2` |
| | `mflo $4` |
| `+` | `add $2, $3, $4` |
| | `sw $2, 4(%sp)` |
| | `addiu %sp, %sp, 4` |

Abbildung 5.1.1: Wort: `binomial1`

Wortes `binomial2` 56 Befehle (siehe Tabelle 5.2.1 (Seite 55)) benötigen. Das Verhältnis zu `binomial1` beträgt fast drei.

Diese höhere Anzahl von Befehlen ist durch folgende Probleme erklärbar. Durch die momentan nur lokale Registerallokation müssen vor dem Aufruf eines Wortes alle noch nicht am Datenstack geschriebenen Werte auf den Datenstack geschrieben werden. Eventuell muß auch noch der Datenstackpointer an die neue Situation am Datenstack angeglichen werden. Durch eine interprozeduralen Registerallokation könnte dieses Problem bewältigt werden. Mit Hilfe von *Inlining* wäre es auch noch möglich, das kurze Wort `sq` direkt in das Wort `binomial2` einzubinden. Dabei würde das Ausführen des Codes `init` und `exit` bei der Ausführung von `sq` wegfallen.

Diese Probleme sollten im Rahmen einer Erweiterung des Maschinencode-Überset-

| Forth-Code | Maschinencode |
|---|---|
| | `lw $2, 0(%sp)` |
| | `nop` |
| `dup *` | `mult $2, $2` |
| | `mflo $3` |
| | `sw $3, 0(%sp)` |

Abbildung 5.2.1: Wort: `sq`

| Forth-Code | Maschinencode |
|---|---|
| | `lw $2, 0(%sp)` |
| | `nop` |
| | `lw $3, 4(%sp)` |
| | `nop` |
| `2dup *` | `mult $3, $2` |
| | `mflo $4` |
| `2*` | `sll $3, $4, 1` |
| | `sw $3, 0(%sp)` |
| | `sw $2, -4(%sp)` |
| | `addiu %sp, %sp, -4` |
| `swap sq` | `jal %ra <(PFA) sq>` |
| | `nop` |
| | `lw $2, 8(%sp)` |
| | `nop` |
| | `lw $3, 4(%sp)` |
| | `nop` |
| | `lw $4, 0(%sp)` |
| | `nop` |
| `+` | `add $5, $3, $4` |
| | `sw $5, 8(%sp)` |
| | `sw $2, 4(%sp)` |
| | `addiu %sp, %sp, 4` |
| `swap sq` | `jal %ra <(PFA) sq>` |
| | `nop` |
| | `lw $2, 4(%sp)` |
| | `nop` |
| | `lw $3, 0(%sp)` |
| | `nop` |
| `+` | `add $4, $2, $3` |
| | `sw $4, 4(%sp)` |
| | `addiu %sp, %sp, 4` |

Abbildung 5.2.2: Wort: `binomial2`

| | |
|---|---|
| 2 | Befehle: `init` Code |
| 5 | Befehle: Wort `sq` (siehe Abbildung 5.2.1 (Seite 53)) |
| 4 | Befehle: `exit` Code |
| 11 | |
| | |
| 2 | Befehle: `init` Code |
| 31 | Befehle: Wort `binomial2` (siehe Abbildung 5.2.2 (vorige Seite)) |
| 2*11 | Befehle: Wort `sq` |
| 4 | Befehle: `exit` Code |
| 56 | |

Tabelle 5.2.1: Maschinebefehle: `binomial2`

| Forth-Code | Maschinencode |
|---|---|
| | `lw $2, 4(%sp)` |
| | `nop` |
| | `lw $3, 0(%sp)` |
| | `nop` |
| `+ dup` | `add $4, $2, $3` |
| `*` | `mult $4, $4` |
| | `mflo $2` |
| | `sw $2, 4(%sp)` |
| | `addiu %sp, %sp, 4` |

Abbildung 5.3.1: Wort: `binomial3`

zer zu beheben sein. Optimal wäre dabei eine in etwa gleichlanger Code wie für das Wort `binomial1`.

## 5.3 Wort: `binomial3`

Die direkte Umsetzung der Funktion der Formel stellt ein wesentlich einfacheres Forth Programm dar.

Die benötigten 15 Befehle (siehe Tabelle 5.3.1 (nächste Seite)) in der Übersetzung des Wortes mit dem Maschinencode-Übersetzer stellen nochmals eine Steigerung um ca. 25 %, bezogen auf `binomial1`, dar. Die Steigerung fällt nicht so hoch aus, da in `binomial1` die Stackwörter schon wegoptimiert werden.

| | |
|---|---|
| 2 | Befehle: `init` Code |
| 9 | Befehle: Wort `binomial3` (siehe Abbildung 5.3.1 (vorige Seite)) |
| 4 | Befehle: `exit` Code |
| 15 | |

Tabelle 5.3.1: Maschinebefehle: `binomial3`

# Kapitel 6

# Messwerte

Die Performancetest wurden auf einer DecStation 2100 (12.5 Mhz R3000) durchgeführt.

Für die Zeitmessung wurde das *builtin* Kommando `time` der *C-Shell* verwendet. Sie gibt umfassendere Angaben als das *user* Kommando `/usr/bin/time` an. Für die Bewertungen werden die Benutzer- und die Systemzeit addiert. Es ist aber trotzdem zu beachten das die Zeiten nicht als völlig exakt zu bewerten sind, da einige äußere Einflüsse Schwankungen in der Ausführung verursachen können. Aus diesem Grund wurden mehrer Wiederholungen bei den Tests durchgeführt, um in den signifikanten Stellen konstante Werte zu erreichen.

Die Messungen wurden fünfmal wiederholt. Es wurden von allen Messungen die schnellsten zur Bewertung herangezogen. Der Grund liegt darin, daß bei dieser die Prozessorausnutzung am höchsten ist und nur wenige außere Einflüsse aufgetreten sind. Es wurde aber auch darauf geachtet, daß bei den Tests keine großen Ausreißer bei den Messungen aufgetreten sind, um eine Richtigkeit der Ergebnisse zu gewährleisten.

## 6.1 Selbstübersetzung

Der Maschinencode-Übersetzer ist mit sich selbst mehrmals übersetzbar.

Mit der Selbstübersetzung als Performancetest wird das Zeitverhalten bei der Übersetzung von Forth Programmen gezeigt. Dadurch läßt sich ein Bild davon machen, wieviel schneller oder langsamer der Maschinencode-Übersetzer ist.

| | | Übersetzer erzeugt | |
|---|---|---|---|
| | | Threaded Code | Maschinencode |
| Übersetzer in | Threaded Code | 1.00 | 4.98 |
| | Maschinencode | - | 3.20 |

Tabelle 6.1.1: Testergebnisse: Selbstübersetzung (relative Übersetzungszeiten bezogen auf den `gforth`-Übersetzer, multipliziert mit 4.8 ergibt die absoluten Übersetzungszeiten in Sekunden)

| relative Zeiten | `forth2c` opt. | Maschinencode Übersetzer | `gforth` | `forth2c` nicht opt. | absolute Zeiten |
|---|---|---|---|---|---|
| `sieve` | 0.43 | 1.00 | 1.89 | 3.62 | 56.47 s |
| `bubble` | 0.20 | 1.00 | 1.13 | 2.06 | 110.38 s |
| `matmul` | 0.53 | 1.00 | 2.96 | 5.73 | 30.80 s |
| `fib` | 0.54 | 1.00 | 2.68 | 4.57 | 45.85 s |

Tabelle 6.1.2: Testergebnisse: Benchmarkprogramme (relative Ausführungszeiten bezogen auf Maschinencode)

### 6.1.1 Programmaufrufe

Der Maschinencode-Übersetzer wird als erstes mit dem `gforth`-Übersetzer übersetzt. Anschließend wird der Maschinencode-Übersetzer mehrmals mit dem Maschinencode-Übersetzer übersetzt.

Für die einzelenen Übersetzungen werden die Zeiten gemessen. Die Differenz der Zeiten stellt die Übersetzungszeit des neu erzeugten Übersetzers dar.

### 6.1.2 Bewertungen

Bei der Bewertung müssen wir hier mehrere unterschiedliche Situationen beachten. Es besteht ein Unterschied sowohl im Übersetzer, als auch im Aufbau der einzelnen Wörter.

Die relativen Übersetzungszeiten in Tabelle 6.1.1 (vorige Seite) zeigen das Verhälnis mit dem `gforth`-Übersetzer. Die relativen Übersetzungszeiten mit 4.8 multipliziert ergeben die absoluten Übersetzungszeiten in Sekunden.

Der `gforth`-Übersetzer erzeugt Threaded Code. Der Maschinencode-Übersetzer erzeugt Maschinencode.

Die Verhältnis zwischen dem Maschinencode-Übersetzer (Ausführen von Maschinencode) und dem `gforth`-Übersetzer (Ausführen von Threaded Code) beträgt 3.20. Das langsamere Zeitverhalten des Maschinencode-Übersetzers ist durch das kompliziertere Übersetzungsverhalten zu erklären.

Die Verhältnis zwischen dem Maschinencode-Übersetzer (Ausführen von Threaded Code) und dem Maschinencode-Übersetzer (Ausführen von Maschinencode) beträgt 1.55. Hierbei ist ersichtlich, daß der Maschinencode schneller ist als der Threaded Code. Beim Maschinencode-Übersetzer werden nur lokale Optimierungen durchgeführt. Es werden auch noch einige Interpreterworte aufgerufen, die im Maschinencode länger dauern als im Threaded Code. Aus diesen beiden Gründen läßt sich momentan nur diese Steigerung von 55 % erreichen.

Bei weiteren Übersetzungen haben wir in etwa die gleichen Übersetzungszeiten (geringe Abweichungen durch unterschiedliche Systembelastungen), da es sich ab diesem Zeitpunkt immer um die selben Situationen handelt. Der Maschinencode-Übersetzer ist beliegig oft selbst übersetzbar, bis die physische Speichergrenze erreicht ist.

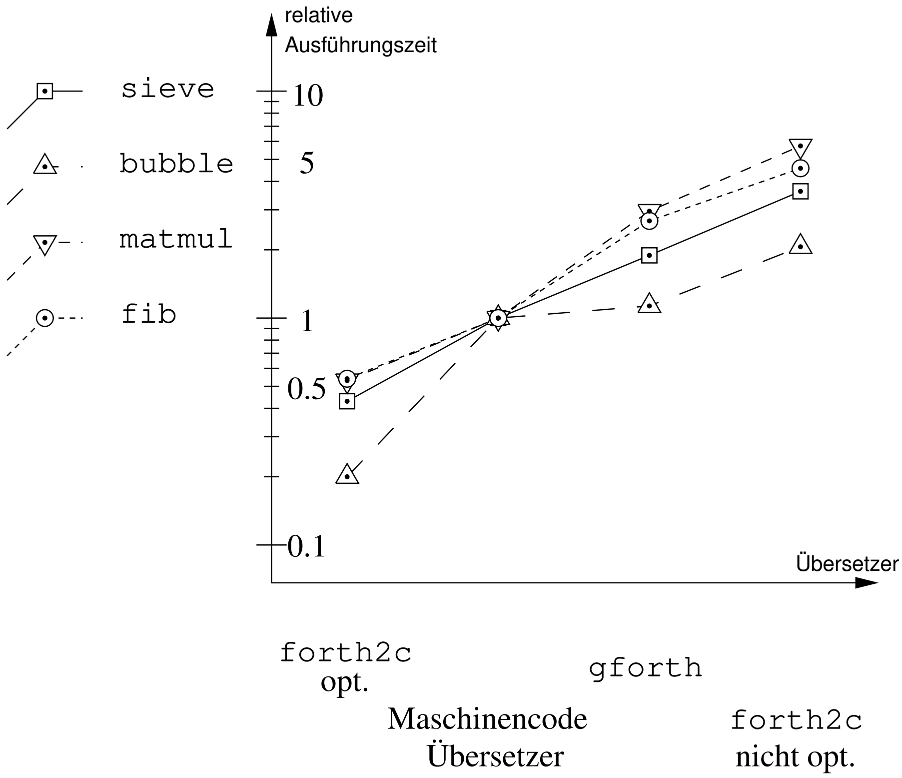


Abbildung 6.2.1: Testergebnisse: Benchmarkprogramme (relative Ausführungszeiten bezogen auf Maschinencode)

## 6.2 Benchmarkprogramme

Um einen allgemeineren Vergleich herzustellen wurden vier Benchmarkprogramme für einen Performancetest verwendet.

**sieve:** berechnet tausendmal alle Primzahlen kleiner als 16384. Als Algorithmus wurde das Sieb von *Eratosthenes* angewendet.

**bubble:** dabei werden 6000 Integers sortiert.

**matmul:** dabei werden zwei $200 \times 200$ Matrixen muliplíziert.

**fib:** berechnet die 34. Fibonacci Zahl mit einem rekursiven Algorithmus.

Der Algorithmus von `bubble` und `matmul` sind vom Stanford Integer Benchmarkprogramm (original in Pascal), das von Marty Fraeman in Forth übersetzt wurde.

In den ersten drei Benchmarkprogrammen werden relative wenige Wortaufrufe ausgeführt. Der rekursive Algorithmus von `fib` stellt dagegen auch einen Test von Wortaufrufen dar. Wortaufrufe sind in Forth sehr häufig [Koo89] [HFWZ87] [Ert95].

| relative Zeiten | `forth2c` opt. | Maschinencode Übersetzer | `gforth` | `forth2c` nicht opt. | absolute Zeiten |
|---|---|---|---|---|---|
| `zif.fs` | 0.45 | 1.00 | 1.76 | 2.66 | 36.56 s |
| `zifelse.fs` | 0.43 | 1.00 | 1.82 | 2.54 | 38.34 s |
| `zcase.fs` | 0.43 | 1.00 | 2.13 | 3.44 | 38.58 s |
| `zagain.fs` | 0.29 | 1.00 | 2.58 | 3.41 | 23.36 s |
| `zuntil.fs` | 0.34 | 1.00 | 2.50 | 3.94 | 20.22 s |
| `zwhile.fs` | 0.37 | 1.00 | 2.22 | 3.09 | 25.85 s |
| `zloop.fs` | 0.52 | 1.00 | 1.13 | 1.93 | 18.38 s |
| `z+loop.fs` | 0.36 | 1.00 | 1.34 | 1.65 | 26.80 s |
| `zrecurse.fs` | 0.21 | 1.00 | 2.43 | 4.59 | 40.21 s |
| `zfunc.fs` | 0.31 | 1.00 | 1.31 | 1.17 | 30.27 s |

Tabelle 6.2.1: Testergebnisse: Kontrollstrukturen (relative Ausführungszeiten bezogen auf Maschinencode)

### 6.2.1 Programmaufrufe

In der anschließenden Liste sind alle Aufrufe für die Performancetest der Benchmarkprogramme aufgelistet:

**übersetzen:** mit `gforth`-Übersetzer
`time gforth File bye.fs`

**ausführen:** mit `gforth`-Übersetzer
`time gforth File run.fs bye.fs`

**übersetzen:** mit Maschinencode-Übersetzer
`time gforth compiler.fs compiler.fs File bye.fs`

**ausführen:** mit Maschinencode-Übersetzer
`time gforth compiler.fs compiler.fs File run.fs bye.fs`

Jedes Testfile wird zweimal übersetzt. Bei der ersten Übersetzung wird die Zeit gemessen, die nur die Zeit der Übersetzung zeigt. Bei der zweiten Übersetzung wird das Programm auch ausgeführt. Das dabei verwendete File `run.fs` beinhaltet den Aufruf des Wortes `main`, das den eigentlichen Performancetest startet. Die Differenz der beiden gemessenen Zeiten stellen die Ausführungszeit der übersetzten Wörter dar.

Es werden auch Vergleiche zu den C-Programmen gemacht, die mit dem `forth2c`-Translator [EM95] [1] generiert wurden. Die `forth2c`-Programme werden in der optimierten und der nicht optimierten Version erstellt. Es wurde der `gcc-2.5.8` Übersetzer verwendet. Bei der optimierten Version wird `-O3` als Optimierungsmethode verwendet.

[1] Die Source für `forth2c` findet man in *http://www.complang.tuwien.ac.at/forth/forth2c.tar.gz* oder *ftp://ftp.complang.tuwien.ac.at/pub/forth/forth2c.tar.gz*.

```
: if0 ( to from -- )
  do
    i 15 = if
      i . cr endif
    loop ;

: main $1000000 0 if0 ;
```

Source 6.2.1: `zif.fs`

```
: ifelsef0 ( to from -- )
  do
    i 15 <> if
      else
      i . cr endif
    loop ;

: main $1000000 0 ifelsef0 ;
```

Source 6.2.2: `zifelse.fs`

Um Vergleiche zwischen den von unterschiedlichen Übersetzern übersetzten Wörtern herzustellen, werden die ermittelten Ausführungszeiten verglichen.

### 6.2.2 Bewertungen

Die relativen Ausführungszeiten in Tabelle 6.1.2 (Seite 58) zeigen das Verhälnis mit den Maschinencode-Übersetzer. Abbildung 6.2.1 (Seite 59) ist eine graphische Darstellung der Ergebnisse der relativen Ausführungszeiten. Das Verhältnis zu den optimierten `forth2c` Programmen liegt zwischen 0.20 und 0.54. Das Verhältnis zum `gforth` liegt zwischen 1.13 und 2.96.

Die schlechte Steigerung in `bubble` liegt darin, daß einige Wortaufrufe von Interpreterworten vorkommen. Diese Wortaufrufe dauern länger als Wortaufrufe im `gforth`.

Beim Vergleich mit den `forth2c` Programmen, die mit voller Optimierung übersetzt wurden, können wir bei einer Erweiterung des Maschinencode-Übersetzers mit einer Steigerung um einen Faktor von mehr als zwei rechnen.

```
: case0 ( to from -- )
  do
    i case
      15 of
        i . cr endof
      endcase
    loop ;

: main $1000000 0 case0 ;
```

Source 6.2.3: `zcase.fs`

```
: again0 ( to from -- )
  begin
    1+
    2dup = if
      2drop exit endif
    again ;

: main $1000000 0 again0 ;
```

Source 6.3.1: `zagain.fs`

```
: until0 ( to from -- )
  begin
    1+
    2dup = until
  2drop ;

: main $1000000 0 until0 ;
```

Source 6.3.2: `zuntil.fs`

## 6.3 Kontrollstrukturen

Performancetests im Grundblockbereich durchzuführen stellen eine schwierige Situation dar. Je mehr, vom Maschinencode-Übersetzer direkt verarbeitbare, Primitive-Wörter im Grundblock vorkommen, desto schneller ist der erzeugte Code. Auch sind Maschinencode Wortaufrufe direkter und dementsprechend schneller als direkt verknüpfte Wörter. Anderseits sind Interpreter Wortaufrufe im Maschinencode langsamer durchzuführen. Ein genaues Mittelmaß zu finden, daß sowohl die Vorteile als auch die Nachteile aufzeigt, ist nahezu unmöglich.

Aus diesem Grund wurden auch Tests mit Kontrollstrukturworten durchgeführt. Es mußten aber trotzdem gewisse Eingeständnisse gemacht werden, um die Kontrollstrukturtests durchzuführen. In den Testfiles kommen auch einige Primitive-Wörter (z.B. `1+`) vor um den entsprechenden Schleifendurchlauf zu gewährleisten.

```
: while0 ( to from -- )
  1+
  begin
    2dup <> while
    1+ repeat
  2drop ;

: main $1000000 0 while0 ;
```

Source 6.3.3: `zwhile.fs`

```
: loop0 ( to from -- )
  do
    loop ;

: main $1000000 0 loop0 ;
```

Source 6.3.4: `zloop.fs`

```
: +loop0 ( to from -- )
  do
    1 +loop ;

: main $1000000 0 +loop0 ;
```

Source 6.3.5: `z+loop.fs`

### 6.3.1 Programmaufrufe

Dabei wurde gleich vorgegangen wie bei den Tests der Benchmarkprogramme, damit nur die Ausführungszeiten verglichen werden.

Es wurden die folgenden Bereiche damit abgedeckt:

- Bedingte Sprünge:

  `if`...`endif`, siehe Source 6.2.1 (Seite 61)
  `if`...`else`...`endif`, siehe Source 6.2.2 (Seite 61)
  `case`...`of`...`endof`...`endcase`. siehe Source 6.2.3 (Seite 61)

- Einfache Schleifen:

  `begin`...`again`, siehe Source 6.3.1 (vorige Seite)
  `begin`...`until`, siehe Source 6.3.2 (vorige Seite)
  `begin`...`while`...`repeat`. siehe Source 6.3.3 (vorige Seite)

- Zählschleifen

  `do`...`loop`, siehe Source 6.3.4 (diese Seite)
  `do`...`+loop`. siehe Source 6.3.5 (diese Seite)

- Rekursive Aufrufe

  `recurse` siehe Source 6.3.6 (nächste Seite)

- Wort Aufrufe

  siehe Source 6.3.7 (nächste Seite)

```
: recurse0 ( to from -- )
  1+ 2dup <> if
    recurse else
    2drop endif ;

: recurse1 ( to from -- )
  do
    $10000 0 recurse0 loop ;

: main $100 0 recurse1 ;
```

Source 6.3.6: `zrecurse.fs`

```
: func0 ( to from -- )
  ;

: func1 ( to from -- )
  do
    func0 loop ;



: main $1000000 0 func1 ;
```

Source 6.3.7: `zfunc.fs`

### 6.3.2 Bewertungen

Die Bewertungen werden nur auf die Ausführungszeiten der Testprogramme gemacht.

Die relativen Ausführungszeiten in Tabelle 6.2.1 (Seite 60) zeigen das Verhälnis mit den Maschinencode-Übersetzer. Das Verhältnis zu den optimierten `forth2c` Programmen liegt zwischen 0.21 und 0.52. Das Verhältnis zum `gforth` liegt zwischen 1.13 und 2.58.

Durch die momentan nur lokale Optimierung im Maschinencode-Übersetzer können wir von einer Erweiterung im Bereich der globalen und interprozedurealen Optimierung mit einer Steigerung um einen Faktor von mehr als zwei rechnen.

# Kapitel 7

# Zusammenfassung und Ausblick

## 7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Maschinencode-Übersetzer erzeugt. Dieser Übersetzer übersetzt Forth Programme in Maschinencode, der sofort ausführbar ist. Für die Umsetzung des Maschinencode-Übersetzer werden vorerst nur relative einfache Algorithmen für die Codeerzeugung verwendet.

Bei der **Befehlsauswahl** wird ein **Datenflußgraph** aufgebaut. Es werden dabei nicht nur Datenfluß-Abhängigkeiten eingebaut. Bei Speicherzugriffen werden zusätzlich Lese-Schreib- und Schreib-Schreib–Abhängigkeiten eingefügt.

Bei der Beendigung eines Grundblockes werden in den Datenflußgraphen die Befehle für das Speichern der Daten auf den Stacks (Daten- oder Rücksprungstack) eingefügt. Hier kommen auch Lese-Schreib- oder Schreib-Schreib–Abhängigkeiten vor. Sofern es notwendig ist werden auch die Befehle für das Angleichen des Stackpointers (Daten- oder Rücksprungstackpointer) in den Datenbflußgraphen eingebaut.

Bei der **Befehlsanordnung** geben die vorhandenen Abhängigkeiten die Reihenfolge der Maschinenbefehle an.

Die **Registerbelegung** belegt für die Pseudoregister ein freies Maschinenregister für die Ergebnisse. Wenn das Ergebnis im Maschinenregister mehrmals verwendet wird, bleibt das Register solange gesperrt. Erst bei der letzten Verwendung wird das Maschinenregister wieder freigegeben und kann für ein anderes Ergebnis verwendet werden.

Bei den Optimierungen werden vom Maschinencode-Übersetzer folgende Bereiche abgedeckt:

**Stackverwaltung**

- Lesen und Schreiben auf die Stacks (Daten- und Rücksprungstack) ist soweit wie möglich optimiert. Es werden die Daten in Maschinenregistern weiterverarbeitet. Nur wenn es unbedingt notwendig ist, werden die Daten in den Speicher (Daten- oder Rücksprungstack) geschrieben oder aus dem Speicher gelesen.

- nur wenn es nötig ist, werden die Stackpointer (Daten- und Rücksprungstackpointer aktualisiert.

**Maschinencode Verknüpfung** Eine eigene Verknüpfungsart der Wörter wird vom Maschinencode-Übersetzer verwendet. Es ist eine Mischung aus subroutinen verknüpften und der vom `gforth`-Übersetzer verwendeten Verknüpfungsart (direkt- oder indirektverknüpft). Dadurch ist eine direkterer Aufruf von maschinencode Wörtern möglich. Auf der anderen Seite ist es immer noch möglich die Wörter auch von `gforth` zu interpretieren.

**Kontrollstrukturen** Es werden direktere Kontrollstrukturen verwendet. Von einer kleinen Menge von Basiskontrollstrukturwörtern werden alle restlichen Kontrollstrukturworte erzeugt.

Bei den Performancetests wurden Messungen in der Selbstübersetzung und anhand einiger Testprogramme durchgeführt.

Bei der Selbstübersetzung ergab sich beim Maschinencode-Übersetzer eine Verlängerung der Übersetzungszeit im Verhältnis zum `gforth`-Übersetzer von 3.20. Dies ist durch die aufwendigeren Übersetzungsalgorithmen zu begründen.

Man erhält aber auf der anderen Seite schnellere Programme. Bei den vier Benchmarkprogrammen (`sieve`, `bubble`, `matmul` und `fib`) bekommen wir eine Steigerung von 13 bis 196 %. Beim Vergleich von Testprogrammen, die die unterschiedlichen Kontrollstrukturen untersuchen, erhalten wir eine Steigerung von 13 bis 158 %.

## 7.2 Ausblick

Im Rahmen eines Forschungsprojekts werden erweiterte Umsetzungen der verwendeten Übersetzungstechniken umgesetzt.

Dabei werden die Befehlsauswahl- und der Befehlsanordnungalgorithmen verbessert. Bei der Registerbelegung wird eine globale und in weiterer Folge eine interprozedurale Registerbelegung verwendet.

Um eine noch bessere Optimierung zu erzielen werden auch einige Optimierungsverfahren angewendet werden. Diese Optimierungen könnten *Tail Call* Optimierung und *Inlining* sein.

Aus den Vergleichen mit den optimierten C-Programmen, die mit dem `forth2c`-Translator erzeugt wurden, können wir mit einer Steigerung bis zu einem Faktor zwei rechnen.

# Anhang A

# Listings

## A.1 Aufbau von allgemein verwendeten Strukturen

### A.1.1 Aufbau der `slist`-Struktur

Für Zusammenhänge in Listenform wird diese Datenstruktur benötigt. Es handelt sich hiebei um eine einfach verkettete Liste, die nur eine Zelle zum Speichern der Daten verwendet. Diese Daten stellen wiederum Adressen von anderen Datenbereichen dar, sodaß auch zusammenhängende Strukturen verwaltet werden können.

Die Worte die auf diese Datenstruktur angewendet werden, scheinen etwas komplex zu sein, stellen aber ein gutes Funktionsspektrum dar.

In Source A.1.1 (diese Seite) werden die Positionen und Größenangaben der einzelen Werte gezeigt. Die Bedeutung der Werte ist folgendermaßen gegeben:

**`slist_data`:** Adresse oder Datenelement

**`slist_next`:** Address des nächsten Datenelementes

**`slist_insert`:** zum Einfügen von neuen Elementen wird dieses Wort verwendet. Es ist dabei offen ob das neue Element direkt einen Wert darstellt oder eine Adresse eines allgemeinen Datenbereiches ist. Die Bedeutung der Daten ist bei der Verwendung der Daten der linearen Liste aber genau zu beachten.

**`slist_delete`:** damit lassen sich Elemente wieder aus der lineare Liste löschen.

```
struct
    1 cells: field slist_data
    1 cells: field slist_next
end-struct node_struct
```

Source A.1.1: Aufbau der `slist`-Struktur

```
struct
    1 cells: field btree_data
    1 cells: field btree_left
    1 cells: field btree_right
end-struct node_struct
```

Source A.1.2: Aufbau der btree-Struktur

slist_forall: zum Ausführen eines Wortes auf alle Elemente der linearen Liste wird dieses Wort verwendet. Es wird dabei ein Ausführungtoken übergeben, der auf jedes Element angewendet wird. Wie schon bei slist_insert erwähnt ist dabei darauf zu achten, daß die Daten vom auszuführenden Wort richtig interpretiert werden.

slist_find: um ein bestimmtes Element der linearen Liste zu suchen, wird dieses Wort verwendet. Auch dabei wird ein Ausführungstoken eines Wortes übergeben. Dieses Wort retouniert ein Flag (true oder false), das schließlich angibt, ob das Element gefunden oder nicht gefunden wurde. Beim erfolgreichen Suchen eines Elementes oder auch mehrmals gleichen Elementes in der linearen Liste wird die Adresse des ersten Auftretens, beim nicht erfolgreichen Suchen wird nil zurückgeliefert.

slist_size: die damit ermittelte Größe der linearen Liste stellt die Anzahl der Elemente dar, nicht den Speicherbedarf der Liste.

### A.1.2 Aufbau der btree-Struktur

Zum Verarbeiten von binären Bäumen wird diese Datenstruktur verwendet. Die Abarbeitung des binären Baumes ist dann beliebig in pre-, infix- und postorder möglich. Auch hier stellen die beinhalteten Daten direkte Werte oder Adressen von anderen Datenbereichen dar, sodaß auch zusammenhängende Strukturen verwaltet werden können.

In Source A.1.2 (diese Seite) werden die Positionen und Größenangaben der einzelen Werte gezeigt. Die Bedeutung der Werte ist folgendermaßen gegeben:

btree_data: Adresse oder Datenelement

btree_left: Address des nächsten linken Datenelementes

btree_right: Address des nächsten rechten Datenelementes

btree_preorder: zum Ausführen eines Wortes auf alle Elemente des binären Baumes wird dieses Wort verwendet. Es wird dabei ein Ausführungtoken übergeben, der auf jedes Element angewendet wird. Die Reihenfolge hiebei ist dadurch gegeben, daß der Knoten selbst zuerst bearbeitet wird und anschließend der linke und dann der rechte Ast.

`btree_inorder`: strukturell gleich wie `btree_preorder`, nur ist die Reihenfolge zuerst der linker Ast, dann der Knoten selbst und anschließend der rechte Ast.

`btree_postorder`: strukturell gleich wie `btree_preorder`, nur ist die Reihenfolge zuerst der linker Ast, dann der rechte Ast und anschließend der Knoten selbst.

# Anhang B

# Glossar

## B.1 Verzeichnis der Abkürzungen

CA Codeadresse

CFA Codefeldadresse

IP Instruktion Pointer

PFA Parameterfeldadresse (die Anfangsadresse des Parameterfeldes)

TOS *top of stack* (oberstes Element im Datenstack)

TOR *top of return stack* (oberstes Element im Rücksprungstack)

WA Wortadresse

## B.2 Deutsch — Englisch

Befehlsauswahl — *instruction selection*, *code selection*
: Die Operationen der Zwischendarstellung werden zu Befehlen der Zielsprache zusammengefaßt.

Befehlsanordnung — *instruction scheduling*
: Die Befehle werden im Hinblick auf eine schnelle Ausführung auf Pipeline- oder superskalaren Prozessoren angeordnet.

Datenfluß-Abhängigkeit — *flow dependence*, *RAW dependence*

Lese-Schreib–Abhängigkeit — *anti dependence*, *WAR dependence*

Lexikon — *dictionary*
: Eine Liste von vorgegebenene oder selbst geschriebenen Definitionen. Diese Definitionen befinden sich in übersetzter Form im Speicher des Computers.

Registerbelegung — *register allocation*
: Die während früherer Phasen verwendeten Pseudoregister werden durch Maschineregister ersetzt.

Schreib-Schreib–Abhängigkeit — *output dependence*, *WAW dependence*

Stackleerlauf — *stack underflow*
: Fehler, der auftritt, wenn bei leerem Stack ein weiterer Wert gefordert wird.

Stacküberlauf — *stack overflow*
: Fehler, der auftritt bei der Überschreitung des, für den Stack reservierten, Speicherbereichs auftritt.

Wort — *word*
: Das ist in Forth der Name einer Definition.

Wortliste — *wordlist*
: Eine eigenständige Kette von Definitionen innerhalb des Lexikons.

# Anhang C

# Literaturverzeichnis


[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[ANS94] American National Standards Institute, Inc. *American National Standard for Information Systems – Programming Language – Forth,* X3J14, März 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, und Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bro81] Leo Brodie. *Starting FORTH.* Prentice-Hall, Englewood Cliffs, NJ, 1981. Deutsche Ausgabe: Programmieren in FORTH, Carl Hanser Verlag, München, Wien, 1984.

[Bro84] Leo Brodie. *Thinking FORTH.* Prentice-Hall, Englewood Cliffs, NJ, 1984. Deutsche Ausgabe: In FORTH denken, Carl Hanser Verlag, München, Wien, 1986.

[EM95] M. Anton Ertl und Martin Maierhofer. Translating Forth to Efficient C. In *EuroForth '95*, Southampton, England, 1995. MicroProcessor Engineering.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, Seiten 73–78, Southampton, England, 1992. MicroProcessor Engineering.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[HFWZ87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, und Thomas Zaremba. An architecture for the direct execution of the Forth programming language. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, Seiten 42–48, 1987.

[KH92] Gerry Kane und Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Kog85] Peter Kogge. An Architectural Trail to Threaded-Code Systems. In *IEEE Computer*, Seiten 22–32, März 1985.

[Koo89] Philip J. Koopman, Jr. *Stack Computers: The New Wave*. Ellis Horwood limited, England, 1989.

[Loe81] R. G. Loeliger. *Threaded Interpretive Languages*. BYTE Publications Inc., Peterborough, NH 03458, 1981.

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[RCM93] Elizabeth D. Rather, Donald R. Colburn, und Charles H. Moore. The evolution of forth. In *History of Programming Languages (HOPL-II) Preprints*, Seiten 177–199, 1993. SIGPLAN Notices 28(3).

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code & an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. 1986 Rochester Forth Conference.

[Woe92] Jack Woehr. *Forth: The New Model*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Zec84] Ronald Zech. *Die Programmiersprache FORTH*. Franzis-Verlag, Franzis Computer-Science, München, Germany, 1984.

# Anhang D

# Register