2DTFP
Ein 2DT-FP Compiler für den IPSC860

Fopradokumentation

Jochen Wolter
Bruchwiesenstr. 1
66111 Saarbrücken

Andreas Kerren
St. Avolder Str. 112
66740 Saarlouis

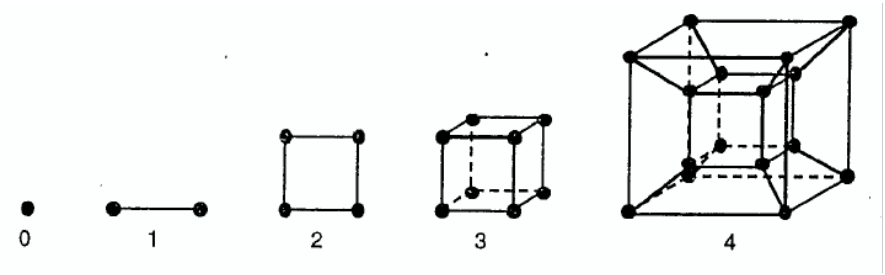
Saarbrücken, den 31. März 1994

Inhaltsverzeichnis

1 Einleitung

Die Aufgabe dieses Fortgeschrittenenpraktikums war die "Implementierung einer funktionalen Programmiersprache auf dem Hypercube".

Der Hypercube der Universität Saarbrücken ist ein INTEL IPSC860 mit 32 Prozessoren des Typs i860. Die Hauptkomponenten des Rechners sind der Hostrechner, die 32 Prozessoreinheiten und das Netzwerk. Die Schnittstelle zum Benutzer bildet der Host, der in unserem Fall ein 386er PC ist. Jede Prozessoreinheit (PE) besitzt 8 MB Hauptspeicher, wobei die PE's "lose" in einem Verbindungsnetzwerk (Hypercube) miteinander gekoppelt sind. Anschaulich wird der Hypercube in folgender Darstellung beschrieben:



Anzahl der PE's :	$n = 2^m$ (m = Dimension)
Verbindungsleitungen je PE :	$\log_2 n$
maximaler Abstand zwischen zwei PE's :	$\log_2 n$

Abb. 1.1: Hypercubes der Dimensionen Null bis Vier (entnommen aus [Brä93])

Auf jeder Prozessoreinheit kann unabhängig von den anderen und asynchron jeweils ein Prozeß laufen. Eine genauere Beschreibung des Hypercubes findet man in [Die93] und in [Brä93].

Bei der zu implementierenden Programmiersprache handelt es sich um 2DT-FP, eine Sprache, die speziell für die parallele Programmierung entworfen wurde. Grundlage der Sprache ist Backus' FP, das in [Bac78] beschrieben wird. Eine kurze Einführung mit den speziellen Merkmalen unserer Implementierung geben wir in Kapitel 2.2. Aufbauend auf dieser nicht parallelen Grundsprache wurde eine Reihe von sogenannten zwei-dimensionalen Transformationen definiert, die die Parallelisierung von Programmen ermöglichen. Eine Beschreibung dieser Transformationen findet man in [BRSW93b] und eine Kurzdarstellung in Kapitel 2.3 .

2 Sprachbeschreibung von 2DT-FP

In diesem Kapitel geben wir eine knappe Beschreibung der Sprache 2DT-FP. Die Betonung liegt auf den Besonderheiten dieser Implementierung, also den getroffenen Vereinbarungen hinsichtlich Syntax und Semantik von Funktionen und Transformationen.

2.1 2D-Arrays und Spalten - die Datenobjekte eines 2DT-FP-Programms

Die funktionale Programmiersprache FP nach Backus stellt eine Reihe von Funktionen und Funktionalformen zur Verfügung, mit deren Hilfe der Programmierer neue Funktionen definieren kann. Hinzu kommen bei 2DT-FP die globalen zweidimensionalen (2D-) Transformationen.

Eine FP-Funktion wird auf ein FP-Objekt angewandt. FP-Objekte unterscheidet man nach Atomen und Sequenzen. Atome sind Objekte einfacher Typen, also entweder eine Zahl (integer oder float), ein Zeichen (character), ein (nichtleeres) Wort (string), ein Wahrheitswert (true 'T' oder false 'F') oder das Nullobjekt '∅'. Sequenzen sind beliebig große Tupel von FP-Objekten, die ihrerseits wieder Sequenzen enthalten können. Eine Sequenz wird folgendermaßen dargestellt:

$< x_1 , .. , x_n >$, wobei $x_1, .., x_n$ FP-Objekte sind.

Eine leere Sequenz, das ist eine Sequenz, die kein Objekt enthält, wird wie folgt dargestellt:

< > , sie ist äquivalent mit dem Atom Nullobjekt '∅'.

Die Anwendung einer FP-Funktion auf ein FP-Objekt bezeichnet man als Prozeß. In einem 2DT-FP Programm gibt es in aller Regel eine Vielzahl von Prozessen, die parallel abgearbeitet werden sollen. Beliebig viele Prozesse werden zu einer Gruppe zusammengefaßt. Die Daten der Prozesse einer Gruppe können somit als zweidimensionales Array betrachtet werden, dessen Spalten die Daten der einzelnen Prozesse sind. Zu Beginn der Programmausführung existiert eine Gruppe (ein zweidimensionales Array), der alle Prozesse (Spalten) angehören. Durch Transformationen können die Daten zwischen Prozessen umverteilt werden, Prozesse können erzeugt und entfernt werden, es können neue Gruppen entstehen oder zwei Gruppen können zu einer vereinigt werden.

Innerhalb einer Gruppe werden auf allen Spalten die gleichen Funktionen ausgeführt. Dabei verändert eine Transformation die Gesamtstruktur der Gruppe, eine FP-Funktion wird dagegen lokal auf alle Spalten der Gruppe parallel angewandt. Auf verschiedene Gruppen können im Gegensatz dazu unterschiedliche Funktionen angewandt werden. Die Prozesse einer Gruppe werden nach bestimmten Regeln auf die vorhandenen Prozessoren verteilt (vgl. hierzu Kapitel 4.2.3, Allokationsstrategie). Innerhalb aller Gruppen werden die Prozesse (Spalten) von Null aufwärts durchnumeriert, damit jeder Prozeß in seiner Gruppe eindeutig identifiziert werden kann.

2.2 Die FP-Funktionen

2.2.1 FP-Funktionen nach Backus

Eine FP-Funktion nimmt sich ein FP-Objekt und erzeugt daraus ein neues FP-Objekt. Abbildung 2.1 gibt eine kurze Beschreibung der im Sprachumfang von Backus' FP enthaltenen Funktionen und Funktionalformen an. Die Syntax folgender Funktionen und Funktionalformen weicht in unserer Implementierung von den dortigen Angaben ab:

Funktion	Syntax
Less than	lt
Greater than	gt
Less or equal	le
Greater or equal	ge
Composition	o
Condition	p -> f ; g
Constant	!c
Apply to all	@
Definition	def <ident> =
Insert	/R bzw. /L
Bottom	bot
True, False	T , F
arithm. Operationen	+ , - , * , /

2.2.2 Erweiterungen von FP

Im Folgenden beschreiben wir weitere FP-Funktionen und Funktionalformen, die unsere Implementierung über den Sprachumfang der Originalbeschreibung hinausgehend bereitstellt:

iota : x ≡ x integer & x>0 → <1,2,..,x> ; ⊥

mod : x ≡ x = <y,z> & y,z integer & z≠0 → y modulo z ; ⊥

: x ≡ x = ⊥ → ⊥ ; Prozeßnummer dieses Prozesses

%% : x ≡ x = ⊥ → ⊥ ; Anzahl der Prozesse in dieser Gruppe

/L f : x ≡ x = <x_1> → x_1 ;
x = <x_1,..,x_n> & n≥2 → f:</Lf:<x_1,..,x_{n-1}>,x_n> ; ⊥

(**bur** f x) : y ≡ f:<y,x>

Abb. 2.1: FP-Funktionen und Funktionalformen (J.Backus 1978)

Functional Programming (FP) System - J.Backus 1978

Primitive Functions:

Selector functions

$1:x \equiv x=\langle x_1, \ldots, x_n\rangle \rightarrow x_1; \perp$

and for any positive integer s

$s:x \equiv x = \langle x_1, \ldots, x_n\rangle \ \&\ n \geq s \rightarrow x_s; \perp$

Thus, for example, $3:\langle A,B,C\rangle = C$ and $2:\langle A\rangle = \perp$. Note that the function symbols 1, 2, etc. are distinct from the atoms *1*, *2*, etc.

Tail

$tl:x \equiv x=\langle x_1\rangle \rightarrow \phi;$
$x=\langle x_1, \ldots, x_n\rangle \ \&\ n \geq 2 \rightarrow \langle x_2, \ldots, x_n\rangle; \perp$

Identity

$id:x \equiv x$

Atom

$atom:x \equiv x$ is an atom $\rightarrow T; x \neq \perp \rightarrow F; \perp$

Equals

$eq:x \equiv x=\langle y,z\rangle \ \&\ y=z \rightarrow T; x=\langle y,z\rangle \ \&\ y \neq z \rightarrow F; \perp$

Null

$null:x \equiv x=\phi \rightarrow T; x \neq \perp \rightarrow F; \perp$

Reverse

$reverse:x \equiv x=\phi \rightarrow \phi;$
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle x_n, \ldots, x_1\rangle; \perp$

Distribute from left; distribute from right

$distl:x \equiv x=\langle y,\phi\rangle \rightarrow \phi;$
$x=\langle y,\langle z_1, \ldots, z_n\rangle\rangle \rightarrow \langle\langle y,z_1\rangle, \ldots, \langle y,z_n\rangle\rangle; \perp$

$distr:x \equiv x=\langle \phi,y\rangle \rightarrow \phi;$
$x=\langle\langle y_1, \ldots, y_n\rangle,z\rangle \rightarrow \langle\langle y_1,z\rangle, \ldots, \langle y_n,z\rangle\rangle; \perp$

Length

$length:x \equiv x=\langle x_1, \ldots, x_n\rangle \rightarrow n; x=\phi \rightarrow 0; \perp$

Add, subtract, multiply, and divide

$+:x \equiv x=\langle y,z\rangle \ \&\ y,z$ are numbers $\rightarrow y+z; \perp$
$-:x \equiv x=\langle y,z\rangle \ \&\ y,z$ are numbers $\rightarrow y-z; \perp$
$\times:x \equiv x=\langle y,z\rangle \ \&\ y,z$ are numbers $\rightarrow y \times z; \perp$
$\div:x \equiv x=\langle y,z\rangle \ \&\ y,z$ are numbers $\rightarrow y \div z; \perp$
(where $y \div 0 = \perp$)

Transpose

$trans:x \equiv x=\langle \phi, \ldots, \phi\rangle \rightarrow \phi;$
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle y_1, \ldots, y_m\rangle; \perp$

where

$x_i=\langle x_{i1}, \ldots, x_{im}\rangle$ and
$y_j=\langle x_{1j}, \ldots, x_{nj}\rangle, \ 1 \leq i \leq n, \ 1 \leq j \leq m.$

And, or, not

$and:x \equiv x=\langle T,T\rangle \rightarrow T;$
$x=\langle T,F\rangle \vee x=\langle F,T\rangle \vee x=\langle F,F\rangle \rightarrow F; \perp$

etc.

Append left; append right

$apndl:x \equiv x=\langle y,\phi\rangle \rightarrow \langle y\rangle;$
$x=\langle y,\langle z_1, \ldots, z_n\rangle\rangle \rightarrow \langle y,z_1, \ldots, z_n\rangle; \perp$

$apndr:x \equiv x=\langle \phi,z\rangle \rightarrow \langle z\rangle;$
$x=\langle\langle y_1, \ldots, y_n\rangle,z\rangle \rightarrow \langle y_1, \ldots, y_n,z\rangle; \perp$

Right selectors; Right tail

$1r:x \equiv x=\langle x_1, \ldots, x_n\rangle \rightarrow x_n; \perp$
$2r:x \equiv x=\langle x_1, \ldots, x_n\rangle \ \&\ n \geq 2 \rightarrow x_{n-1}; \perp$
etc.
$tlr:x \equiv x=\langle x_1\rangle \rightarrow \phi;$
$x=\langle x_1, \ldots, x_n\rangle \ \&\ n \geq 2 \rightarrow \langle x_1, \ldots, x_{n-1}\rangle; \perp$

Rotate left; rotate right

$rotl:x \equiv x=\phi \rightarrow \phi; x=\langle x_1\rangle \rightarrow \langle x_1\rangle;$
$x=\langle x_1, \ldots, x_n\rangle \ \&\ n \geq 2 \rightarrow \langle x_2, \ldots, x_n,x_1\rangle; \perp$

Functional Forms:

Composition

$(f \circ g):x \equiv f:(g:x)$

Construction

$[f_1, \ldots, f_n]:x \equiv \langle f_1:x, \ldots, f_n:x\rangle$ (Recall that since $\langle \ldots, \perp, \ldots\rangle = \perp$ and all functions are $\perp$-preserving, so is $[f_1, \ldots, f_n]$.)

Condition

$(p \rightarrow f; g):x \equiv (p:x)=T \rightarrow f:x; \quad (p:x)=F \rightarrow g:x; \perp$

Conditional *expressions* (used outside of FP systems to describe their functions) and the *functional form* condition are both identified by "$\rightarrow$". They are quite different although closely related, as shown in the above definitions. But no confusion should arise, since the elements of a conditional expression all denote values, whereas the elements of the functional form condition all denote functions, never values. When no ambiguity arises we omit right-associated parentheses; we write, for example, $p_1 \rightarrow f_1; p_2 \rightarrow f_2; g$ for $(p_1 \rightarrow f_1; (p_2 \rightarrow f_2; g))$.

Constant (Here x is an object parameter.)

$\bar{x}:y \equiv y=\perp \rightarrow \perp; x$

Insert

$/f:x \equiv x=\langle x_1\rangle \rightarrow x_1; x=\langle x_1, \ldots, x_n\rangle \ \&\ n \geq 2$
$\rightarrow f:\langle x_1, /f:\langle x_2, \ldots, x_n\rangle\rangle; \perp$

If f has a unique right unit $u_f \neq \perp$, where $f:\langle x,u_f\rangle \in \{x, \perp\}$ for all objects x, then the above definition is extended: $/f:\phi = u_f$. Thus

$/+:\langle 4,5,6\rangle = +:\langle 4, +:\langle 5, /+:\langle 6\rangle\rangle\rangle$
$= +:\langle 4, +:\langle 5,6\rangle\rangle = 15$
$/+:\phi = 0$

Apply to all

$\alpha f:x \equiv x=\phi \rightarrow \phi;$
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle f:x_1, \ldots, f:x_n\rangle; \perp$

Binary to unary (x is an object parameter)

$(\text{bu}\ f\ x):y \equiv f:\langle x,y\rangle$

Thus

$(\text{bu} + 1):x = 1+x$

While

$(\text{while}\ p\ f):x \equiv p:x=T \rightarrow (\text{while}\ p\ f):(f:x);$
$p:x=F \rightarrow x; \perp$

The above functional forms provide an effective method for computing the values of the functions they denote (if they terminate) provided one can effectively apply their function parameters.

Definitions. A *definition* in an FP system is an expression of the form

Def $l \equiv r$

where the left side l is an unused function symbol and the right side r is a functional form (which may depend on l). It expresses the fact that the symbol l is to denote the function given by r. Thus the definition Def last1 $\equiv$ 1$\circ$reverse defines the function last1 that produces the last element of a sequence (or $\perp$).

2.2.3 Das Symbol Bottom

Tritt bei der Anwendung einer FP-Funktion das Symbol '⊥' (Bottom) für einen undefinierten Funktionswert auf, so wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen. Eine Fortsetzung der Berechnungen ist nicht sinnvoll, da alle Funktionen den Wert '⊥' bewahren, und dieser sich bis ins Endergebnis fortpflanzen würde.

2.2.4 Prioritäten

Die Prioritäten zwischen den Funktionalformen untereinander kann man nachstehender Tabelle entnehmen. In dieser Darstellung nimmt die Priorität von oben nach unten ab.

@ , /L , /R , !
o
→
while , bu , bur
;

Der zuletzt aufgeführte Konstruktor '**;**' dient der Verknüpfung globaler Transformationen (vgl. Kapitel 2.3).

2.3 Die globalen Transformationen

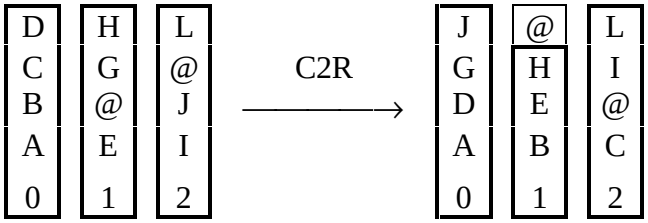
Die globalen 2D-Transformationen werden auf die Gesamtheit aller Prozesse der aktuellen Gruppe angewandt. Wie wir oben gesehen haben, können die Spalten eines 2D-Arrays aus einer Sequenz, aber auch aus einem einzelnen Atom bestehen. Die meisten Transformationen sind allerdings nur anwendbar, wenn alle Spalten je eine Sequenz enthalten. (Eine Ausnahme bilden hier lediglich die Transformation BROAD und der Konstruktor Split&Glue.) Außerdem muß die Sequenzlänge (Spaltenhöhe) aller Spalten gleich sein. Dies wird dadurch erreicht, daß die längste Spalte ermittelt wird und alle anderen Spalten mit Nullelementen '@' aufgefüllt werden (nicht zu verwechseln mit der Funktionalform "apply to all"). Nach Ausführung der Transformation werden die Nullelemente am Spaltenende wieder entfernt. Da es nun möglich ist, daß Nullelemente auch innerhalb einer Spalte auftreten (ein Beispiel hierzu ist die Anwendung von C2R auf Spalten unterschiedlicher Länge), muß erklärt werden, was geschieht, falls eine FP-Funktion auf ein solches Nullelement '@' trifft. Weil Berechnungen auf Nullelementen '@' ebenso wenig sinnvoll sind, wie Berechnungen auf Bottom '⊥', wird auch in diesem Fall eine Fehlermeldung ausgegeben und die Programmausführung abgebrochen.

Alle in [BRSW93b] vorgestellten globalen 2D-Transformationen wurden von uns implementiert, eine kurze Beschreibung der Funktionsweise erfolgt anhand von Graphiken.

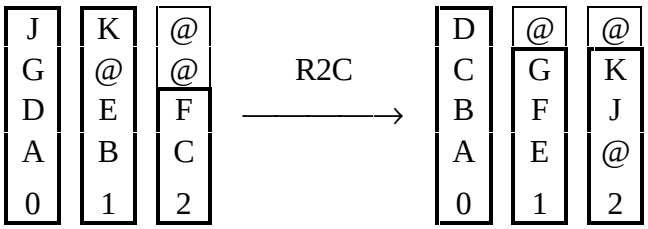
2.3.1 Shape-Preserving Transformations

Alle in diesem Abschnitt beschriebenen Transformationen haben gemeinsam, daß sie die Anzahl der Prozesse, ihre Allokation (Zuordnung zu den Prozessoren) sowie die Gruppenaufteilung nicht verändern.

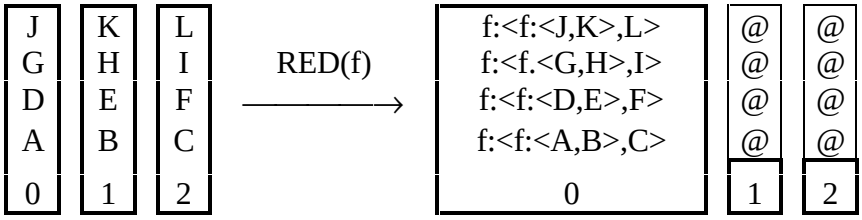
Columns to rows



Rows to columns

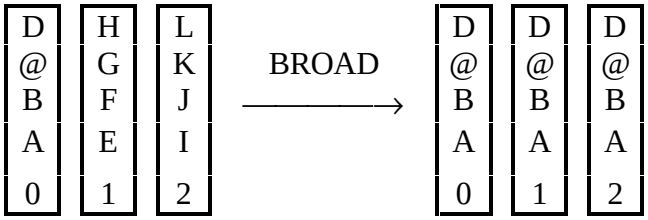


Reduce



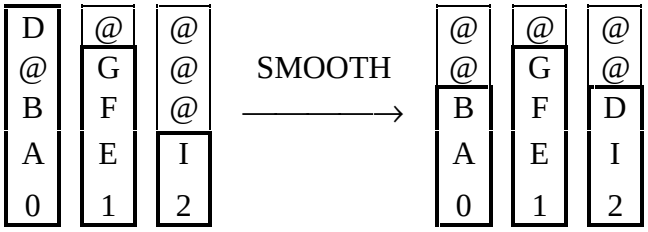
Bei RED(f) muß f eine zweistellige, kommutative und assoziative FP-Funktion sein und darf keine Transformationen enthalten. Der Aufruf einer benutzerdefinierten FP-Funktion ist an dieser Stelle erlaubt. Die Daten aller Prozesse außer denen von Prozeß 0 werden gelöscht, so daß diese Spalten die leere Sequenz '< >' enthalten.

Broadcast

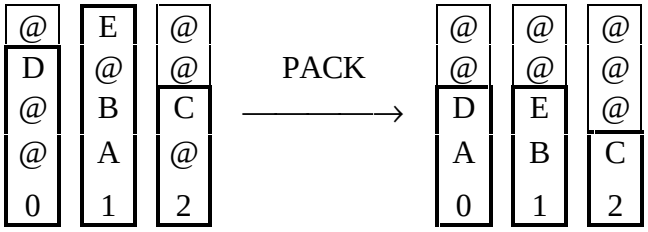


BROAD kopiert die Daten des Prozesses 0 in alle anderen Prozesse. Die Prozeßanzahl bleibt gleich. Broad kann auch verwendet werden, wenn Spalte 0 keine Sequenz, sondern lediglich ein Atom enthält.

Smoothing

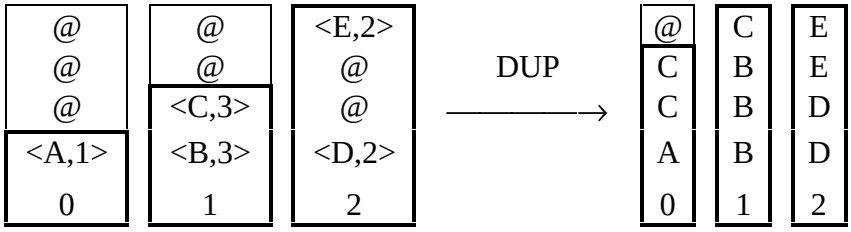


Packing



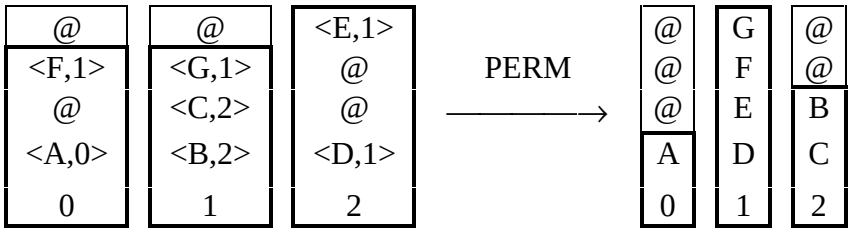
Die Transformationen SMOOTH und PACK haben die Aufgabe, Nullelemente '@', die innerhalb einer Spalte auftreten, zu entfernen und die Spaltenlängen auszugleichen. Hierbei erfolgt die Anordnung der verbleibenden Elemente bei SMOOTH beliebig, bei PACK in einer schlangenartigen Form.

2D-Duplication



DUP erwartet, daß jedes Spaltenelement ungleich '@' eine zweielementige Sequenz ist, die als zweites Objekt eine positive ganze Zahl enthält. Diese Zahl gibt die Anzahl der Kopien an, die von der jeweiligen Datenkomponente angefertigt werden sollen. Die erzeugten Kopien werden gleichmäßig über die Prozesse verteilt, so daß sich die Spaltenhöhe maximal um eins unterscheidet. Die Anordnung der Elemente erfolgt beliebig. Sollen mehr Elemente erzeugt werden, als die momentane Arraygröße zuläßt, so wird die Spaltenlänge automatisch erhöht, bis alle Kopien Platz in dem 2D-Array finden.

2D-Permutation



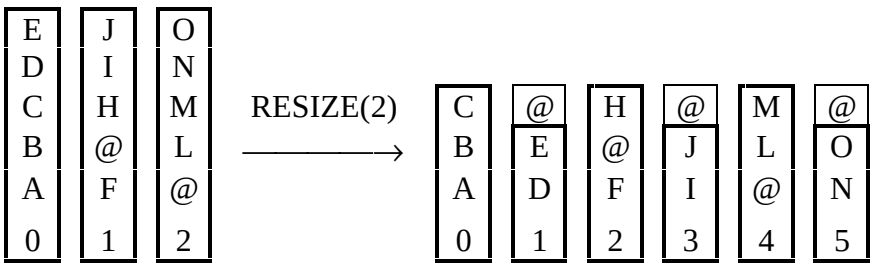
PERM erwartet, daß jedes Spaltenelement ungleich '@' eine zweielementige Sequenz ist, die als zweites Objekt eine nicht negative ganze Zahl enthält. Diese Zahl gibt die Nummer des Prozesses an, der die Datenkomponente erhalten soll. Sie muß daher kleiner als die Prozeßanzahl sein. Die Anordnung der Daten in dem Empfängerprozeß ist nicht festgelegt. Es wird jedoch vermieden, daß Nullelemente innerhalb der Spalte auftreten. Wie bei DUP kann sich auch hier die Spaltenlänge vergrößern.

2.3.2 Reshaping Transformations

Die Transformationen dieser Klasse verändern die Prozeßanzahl innerhalb der aktiven Gruppe und somit deren Struktur. Die Prozesse werden neu numeriert und ihre Allokation auf den Prozessoren ändert sich.

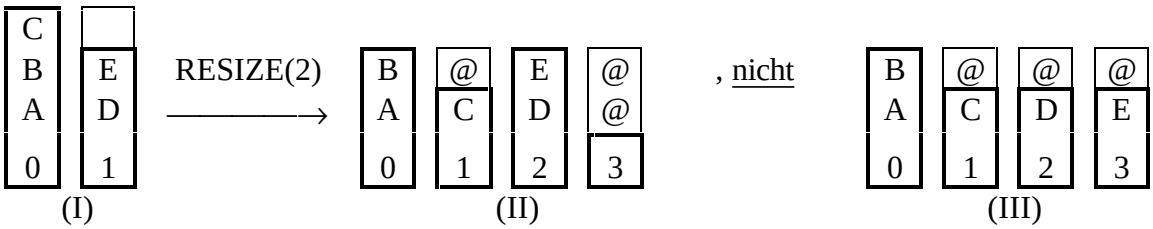
Die hier vorgestellten Resize-Transformationen RESIZE(p) sind für beliebige ganze Zahlen p ungleich Null definiert.

Für positive p erhöht sich die Prozeßanzahl auf das p-fache der alten Prozeßzahl, während die Spaltenhöhe durch p geteilt wird (nach eventuellem Aufrunden auf das nächste Vielfache von p). Die Daten der alten Prozesse werden auf die zugehörigen p neuen Prozesse verteilt. Besaß der Prozeß nicht genügend Datenkomponenten, wird mit Nullelementen aufgefüllt.



An dieser Stelle sei noch einmal darauf hingewiesen, daß die größte Spaltenlänge für das Ergebnis einer Transformation maßgebend ist. Alle Spalten mit geringerer Länge werden mit Nullelementen aufgefüllt. Bei RESIZE(p) (p>0) muß man daher beachten, daß in manchen Fällen leere Spalten entstehen können, was oft nicht erwünscht ist. Man betrachte hierzu folgendes Beispiel:

RESIZE(2) angewendet auf (I) liefert als Ergebnis (II) und nicht etwa (III).

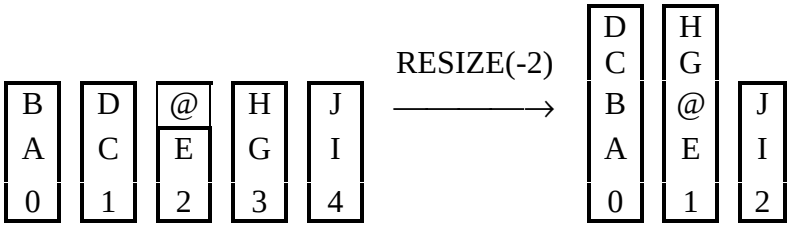


(I) (II) (III)

Nach einer Transformation werden Nullelemente am Spaltenende wieder entfernt, so daß diese für nachfolgende FP-Funktionen nicht sichtbar sind.

RESIZE(-p) (p>0) faßt jeweils p Prozesse zu einem Prozeß zusammen. Ihre Daten werden hinten an die Spalten der übrigbleibenden Prozesse angehängt, wodurch sich deren Spaltenhöhe auf das p-fache vergrößert. Gab es Prozesse, die weniger Datenkomponenten als andere besaßen, so können hierdurch Nullelemente innerhalb der Spalten auftreten. (Vergleiche hierzu Spalte 2 im Beispiel unten, deren Höhe um eins geringer ist als die der anderen Spalten.)

Ist p größer als die Prozeßanzahl, so faßt RESIZE(-p) alle Prozesse zu einem einzigen zusammen.



2.3.3 Array-Creating Transformations

Mit Hilfe der Konstruktoren Split&Glue und Cut&Paste können aus einer Gruppe zwei neue Gruppen (2D-Arrays) erzeugt werden. Auf beide Gruppen lassen sich nun verschiedene Funktionen anwenden. Nach deren Ausführung werden beide Gruppen wieder zu einer einzigen vereinigt.

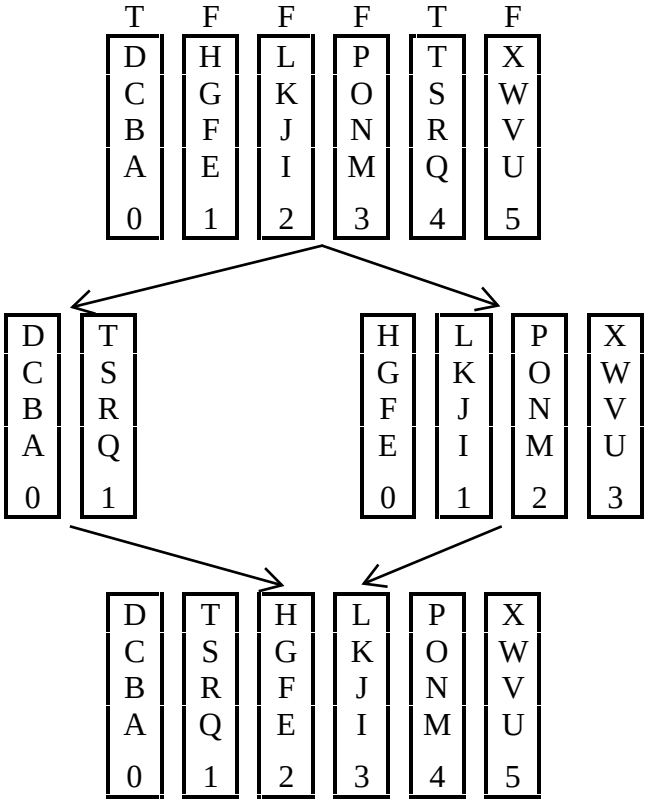
2.3.3.1 Split&Glue

Syntax: p -s&g->{ f || g }

Semantik: Split&Glue teilt die Prozesse der Gruppe in Abhängigkeit eines Prädikatswertes auf die beiden neuen Gruppen auf. Das Prädikat p wird auf die Daten jedes Prozesses angewandt. Ist sein Ergebnis "true", so kommt der Prozeß in die erste Gruppe, ist das Ergebnis "false", wird er der zweiten Gruppe zugeordnet. Die Daten der Prozesse bleiben hierbei unangetastet, lediglich die Prozesse selbst werden innerhalb der zwei Gruppen beginnend mit Null neu numeriert.

Auf die Prozesse der ersten Gruppe wird die Funktion f angewandt, auf die der zweiten Gruppe die Funktion g. Nach Beendigung der Funktionsanwendungen in beiden Gruppen werden die Prozesse wieder in einer Gruppe zusammengefaßt und, beginnend mit den Prozessen der ersten Gruppe, neu numeriert. Hierbei spielt es keine Rolle, ob sich die Spaltenhöhe oder die Anzahl der Prozesse in einer der beiden Gruppen verändert haben (z.B. durch Resize).

Als Prädikat p ist lediglich eine FP-Funktion zugelassen, dagegen können f und g wiederum beliebige Transformationen enthalten.



2.3.3.2 Cut&Paste

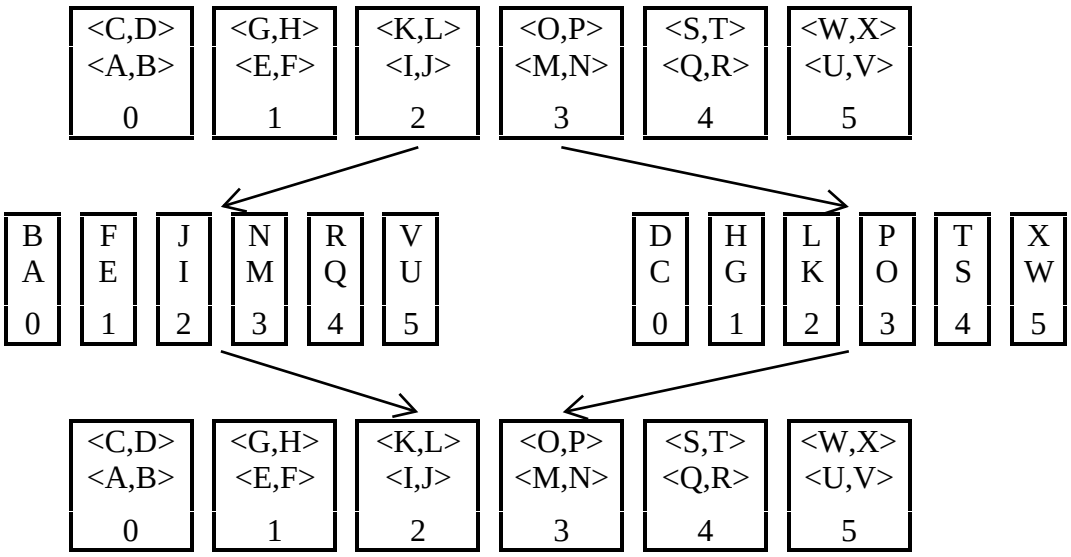
Syntax: h -c&p->{ f == g }

Semantik: Cut&Paste teilt die aktuelle Gruppe in zwei neue Gruppen auf, die die gleiche Anzahl von Prozessen besitzen wie die ursprüngliche. Voraussetzung hierfür ist, daß jede Spalte aus einer zweielementigen Sequenz besteht. Cut&Paste nimmt sich jeweils die ersten Komponenten aller Spalten und erzeugt daraus die Spalten der ersten Gruppe. Die zweiten Komponenten der Sequenzen bilden die Prozesse der zweiten Gruppe. Die FP-Funktion h dient dazu, diese Voraussetzung zu schaffen, also die zweielementige Sequenz zu erzeugen. Die Prozesse der ersten Gruppe wenden die Funktion f an, die zweite Gruppe führt Funktion g auf ihren Daten aus. f und g können globale oder lokale Funktionen sein. Nach Beendigung beider Funktionsanwendungen werden die Gruppen wieder vereinigt, wobei erneut zweielementige Sequenzen entstehen, deren erste Komponenten die Ergebnisse der Prozesse der ersten Gruppe und deren zweite Komponenten die Daten aus der zweiten Gruppe sind. Hierbei wird vorausgesetzt, daß die Anzahl der Prozesse in beiden Gruppen gleich ist. Sie darf sich allerdings von der ursprünglichen Prozeßanzahl unterscheiden.

Folgende Darstellung beschreibt den Vorgang bei Cut&Paste für einen einzelnen Prozeß:

< A , B > —c&p→ < f:A , g:B >

Die Verteilung der Daten auf die zwei Gruppen geschieht wie folgt:



2.4 Definitionen

Wie bereits erwähnt hat der Programmierer die Möglichkeit, eigene Funktionen zu definieren. Eine Definition hat folgende Syntax:

def f = g

Hierdurch wird eine neue Funktion mit dem Namen f definiert, deren Aufruf die Abarbeitung der Funktion g bewirkt.

Ein 2DT-FP-Programm besteht aus einer Menge solcher Funktionsdefinitionen. Die zuletzt definierte Funktion ist die "Hauptfunktion", also die Funktion, die bei der Programmausführung aufgerufen wird.

Für die in einer Definition verwendeten Namen gelten folgende Einschränkungen:

- Ein Funktionsname darf nicht länger als 100 Zeichen sein.
- Ein Name darf aus einer beliebigen Kombination von Buchstaben und Zahlen bestehen, muß aber mit einem Buchstaben beginnen. Als Sonderzeichen ist nur der Unterstrich '_' in einem Namen erlaubt.
- Es darf keiner der vordefinierten Funktionsnamen verwendet werden.

Die Funktion g auf der rechten Seite einer Definition kann beliebig aus FP-Funktionen, Funktionalformen, globalen Transformationen und benutzerdefinierten Funktionen zusammengesetzt werden. Hierbei ist die Unterscheidung zwischen lokalen FP-Funktionen und globalen Transformationen von zentraler Bedeutung. Während die Komposition von FP-Funktionen mit Hilfe der Funktionalform 'o' geschieht, werden Transformationen durch den Operator '**;**' verknüpft. Eine benutzerdefinierte Funktion, die einen Aufruf einer Transformation oder den Operator '**;**' enthält, wird als global betrachtet. Sie kann daher weder innerhalb von FP-Funktionalformen noch als Funktionsparameter bei REDUCE(f) oder als erste Funktion bei Split&Glue bzw. Cut&Paste verwendet werden.

Im Gegensatz dazu kann eine (benutzerdefinierte) reine FP-Funktion an jeder Stelle eines Programms auftreten, an der auch globale Transformationen erlaubt sind. So ist die Verknüpfung zweier reiner FP-Funktionen mit '**;**' nicht ausgeschlossen, sollte jedoch vom Programmierer möglichst vermieden werden, da dies Nachteile bei der Laufzeit mit sich bringt.

2.5 Kommentare

Ein 2DT-FP-Programm kann Kommentare enthalten. Sie beginnen mit ' // ' und reichen bis zum Ende der jeweiligen Zeile.

3 Compilierung eines 2DT-FP-Programms

Der Compiler *2dtfp* übersetzt 2DT-FP-Programme in C-Code. Die erzeugten C-Programme müssen anschließend mit einem C-Compiler für den Hypercube übersetzt werden.

Eine bequeme Möglichkeit zur Übersetzung eines 2DT-FP-Programms mit Hilfe des Shell-Skripts "**code**" wird in Anhang A beschrieben. Die exakte Vorgehensweise bei der Übersetzung geben wir in den folgenden Abschnitten 3.1 und 3.2 an.

3.1 Übersetzung in C-Code

Dateinamen von 2DT-FP-Programmen müssen die Endung "**.2dtfp**" besitzen. Beim Compilieren eines Programms mit dem Compiler *2dtfp* darf diese Endung nicht mit angegeben werden. Der Aufruf des Compilers für das Programm "**beispiel.2dtfp**" hat also grundsätzlich folgende Syntax:

2dtfp beispiel

Tritt bei der Compilierung des Programms kein Fehler auf, so wird eine Datei "**beispiel.c**" erzeugt. Bricht der Compiler ab, so kann dies folgende Fehlerursachen haben:

- Syntaxfehler im Quellprogramm
- Doppeldeklarationen von Funktionsnamen
- Nichtdeklaration von angewandten Funktionsnamen
  Diese Meldung tritt auch für eine angewandte Funktion auf, deren Definition einen Syntaxfehler enthält.
- Deklaration einer Funktion als global und Aufruf der Funktion an einer Stelle, wo nur eine lokale Funktion erlaubt ist.

Eine genaue Auflistung der Fehlermeldungen des Compilers enthält Anhang C .

3.2 Compilierung des erzeugten C-Codes

Der aus dem Programm "**beispiel.2dtfp**" erzeugte C-Code "**beispiel.c**" muß mit dem C-Crosscompiler *icc* in ein ausführbares Knotenprogramm für den Hypercube übersetzt werden. Gleichzeitig ist ein Zusammenlinken mit den Laufzeitfunktionen erforderlich, die in den Dateien "**daten.h**", "**funct.c**", "**transf.c**" und "**inout.tab.c**" definiert sind. Der Compileraufruf des *icc* muß also wie folgt aussehen:

icc -o beispiel beispiel.c funct.o transf.o inout.tab.c -node

Der Compiler *icc* erzeugt nun die ausführbare Datei "**beispiel**".

Hierbei wurde vorausgesetzt, daß für die Laufzeitfunktionen bereits Objektdateien (mit der Dateiendung ".o") erzeugt wurden. Dies geschieht mit den Aufrufen

icc -c funct.c -node und icc -c transf.c -node .

Die Datei "inout.tab.c" erzeugt man aus den Dateien "input.l" und "inout.y" mit den GNU-Tools FLEX bzw. BISON durch Ausführung der Befehle

flex input.l und bison inout.y .

3.3 Ausführung des erzeugten Programms

Ein mit dem Compiler *2dtfp* erzeugtes Programm liest seine Eingabedaten aus einer Eingabedatei. Die Eingabe der Daten direkt über die Tastatur ist nicht möglich.

Um das Programm "**beispiel**" auf den Knoten eines Hypercube auszuführen wird das Programm mit folgendem Befehl geladen:

load beispiel eingabe

Hierbei ist "**eingabe**" der Name der Eingabedatei.

Die Ausgabe des Programms erfolgt auf dem Bildschirm. Alle Spalten der Gruppe 0 werden hintereinander ausgegeben. (Man beachte: Am Programmende existiert nur noch eine Gruppe.)

Folgende Laufzeitfehler können einen Programmabbruch verursachen:

- Eingabedatei nicht vorhanden
- Fehlerhafte Eingabe
- Die Anwendung einer FP-Funktion liefert ein undefiniertes Ergebnis (das Symbol '⊥').
- Anwendung einer lokalen FP-Funktion auf eine Spalte, die das Nullelement '@' enthält.
- Anwendung einer Transformation auf ein 2D-Array mit einer nicht für sie zugelassenen Struktur.
- Der Hauptspeicher ist erschöpft.
- Die Ausgabepuffer sind für die anfallenden Datenmengen zu klein.

Für eine genaue Beschreibung der Fehlermeldungen siehe Anhang C .

3.4 Konventionen für den Aufbau einer Eingabedatei

Die Eingabe für ein 2DT-FP-Programm erfolgt spaltenweise, d.h. zuerst werden die Daten des Prozesses 0 eingelesen, dann die Daten des Prozesses 1 und so weiter. Die Daten der Spalten müssen also in der Eingabedatei hintereinander angeordnet sein. Eine Spalte wird entweder durch die Zeichen '<' und '>' eingeschlossen, oder sie darf nur aus einem einzelnen Atom bestehen. Die einzelnen Spalten werden durch Leerzeichen,

Tabulatoren oder Zeilenschaltungen voneinander getrennt, nicht durch Kommata ! Die leere Eingabe (leere Eingabedatei) ist äquivalent mit der Eingabe '< >'.

Für die Darstellung der Objekte gelten folgende Regeln:

- Characters werden in einfache Hochkommata eingeschlossen, bei Buchstaben können diese Hochkommata weggelassen werden, sofern dies nicht das große 'T' oder 'F' ist.
- Strings werden in doppelte Hochkommata eingeschlossen. Bei Worten, die lediglich aus Buchstaben und Zahlen bestehen und die mit einem Buchstaben beginnen, können die Hochkommata weggelassen werden.
- Die Eingabe von True oder False erfolgt als großes 'T' bzw. 'F'.
- Eine Sequenz wird von den Zeichen '<' und '>' begrenzt. Die Elemente einer Sequenz werden durch Kommata voneinander getrennt.

Eine Spalte kann beliebig viele Objekte enthalten, die wie bei einer Sequenz durch Kommata getrennt sind.

Bei einem Fehler in der Eingabedatei wird das Programm sofort abgebrochen, wobei die Zeilennummer der Zeile, in der der Fehler auftrat, ausgegeben wird.

Beispiele für korrekte Eingaben sind:

4	erzeugt	1 Prozeß
3 4.5 6 A 8		5 Prozesse
<3,4> <'A'> 'c' < >		4 Prozesse
<string,"string"> 3.0 7		3 Prozesse
< > 4		2 Prozesse

Nicht erlaubt sind z.B. folgende Eingaben:

<8, >	2 , 3	<3> , <7>
,	6 8 >	

3.5 Debuggen eines 2DT-FP-Programms

Für das Debuggen eines 2DT-FP-Programms stellt der Compiler *2dtfp* die Option '-d' zur Verfügung, die beim Übersetzen des Programms als erster Parameter vor der Eingabe des Dateinamens angegeben werden muß. Dadurch werden in dem erzeugten C-Programm Ausgabeanweisungen generiert, die den Prozessor 0 veranlassen, bei jeder FP-Funktion ihren Namen und die Daten, auf die die Funktion angewendet wird, auszugeben. Bei einer Transformation wird nur ihr Name ausgegeben.

Mit diesen Informationen läßt sich dann feststellen, an welcher Programmstelle ein Fehler aufgetreten ist. Hierbei kann es jedoch zu Verfälschungen kommen, wenn innerhalb einer langen Folge von FP-Funktionen ein Fehler bei einem anderen Prozessor auftritt. Wenn dieser das Programm abbricht, ist Prozessor 0 möglicherweise noch mit der Ausgabe zu einer früheren, fehlerfreien Funktion beschäftigt. Um in diesem Fall die Fehlerstelle zu finden, kann die Ausgabe der Informationen auch durch einen anderen Prozessor erfolgen. Hierzu hängt man dem Compilerschalter '-d' noch die Nummer des entsprechenden Prozessors an. '-d3' veranlaßt zum Beispiel den Prozessor 3 zur Ausgabe der Debugging-Informationen. Ebenfalls zur Vermeidung dieses Problems wird beim Debuggen vor und nach jeder Transformation synchronisiert, damit die Prozessoren anschließend gleichzeitig mit denselben Funktionen weitermachen.

4 Der Compiler 2dtfp

4.1 Codegenerierung

Die Quelldatei sei im folgenden mit *filename.2dtfp* benannt.

4.1.1 Scanner und Parser

Beide, Scanner und Parser, wurden mit den GNU programming tools FLEX bzw. BISON generiert. Die dazu benötigten Input-Files werden mit 2dt.l für FLEX und 2dt.y für BISON bereitgestellt.

4.1.1.1 Scanner

Die Datei 2dt.l enthält sogenannte "rules". Darunter versteht man Paare von regulären Ausdrücken und C-Code, z.B. "BROAD { return (Broad); }". Beim späteren Scannen bedeutet dies, das bei jedem Auftreten des Patterns "BROAD" die Aktion "Liefere das Token "Broad" an den Parser zurück" ausgeführt wird. FLEX generiert aus dieser Eingabe die Datei lex.yy.c, die ihrerseits den eigentlichen Scanner-Code yylex ( ) beinhaltet. Diese Funktion wird vom Parser aufgerufen.

4.1.1.2 Parser

Mit Hilfe des Scanners liest der Parser die einzelnen Funktionsdefinitionen in der Quelldatei. Wird das "Start"-Symbol der Grammatik erreicht, so sind alle Funktionen abgearbeitet worden und es liegt kein primärer Syntaxfehler vor. Die in 2dt.y definierte Grammatik ist in Anhang F angegeben. Während des Parserlaufes werden simultan folgende Aktionen durchgeführt:

- Jedes Vorkommen eines benutzerdefinierten Namens wird in eine Namensliste geschrieben. Diese Liste enthält den Namen (konkateniert mit dem Suffix "_dtfp", um Namenskonflikte mit festimplementierten 2DT-FP- und C-Funktionen zu vermeiden), die Zeilennummer und den Typ. Ist die benutzerdefinierte Funktion lediglich aus reinen FP-Funktionen und Funktionalformen zusammengesetzt, so erhält dieser Funktionsname den Typ 'f', ansonsten 't'. Diese Unterscheidung ist bei der späteren Codeerzeugung von Bedeutung (siehe 4.1.4 ff).
- Erzeugung des Syntaxbaumes durch das Aufrufen der entsprechenden Funktionen. Diese Funktionen sind in der Datei 2dtfp.c definiert. Die Datei enthält zudem zusätzlichen Code, um den Syntaxbaum zu Testzwecken ausgeben zu können. Um dies zu erreichen, muß das Makro DEBUG in 2dtfp.h auskommentiert, und der Compiler neu übersetzt werden. Dabei wird auch die o.g. Liste angezeigt. Näheres zum Aufbau des Syntaxbaumes läßt sich im Abschnitt 4.1.2 nachlesen.

- Wie an der Beispielübersetzung in Anhang E anschaulich nachzuvollziehen ist, werden in fast jeder generierten Funktion mehrere Zeigervariablen deklariert. Die Anzahl ist abhängig davon, welche FP-Funktionen, Funktionalformen und Transformationen in der jeweiligen benutzerdefinierten Funktion auftreten. Beim Parsen läßt sich eine obere Schranke (worst-case) für den Bedarf an Zeigervariablen feststellen. Diese Zahl wird in einer neuen Liste, die jeweils in einem Listenelement den benutzerdefinierten Namen, den Zeiger auf dessen Syntaxbaum, eben diese Zahl und einen Zeiger auf das nächste Listenelement enthält, abgespeichert (siehe 4.1.2).

Nach Erreichen des "Start"-Symbols (d.h. die Grammatik ist vollkommen reduziert), erfolgt eine Prüfung der in der Namensliste aufgeführten benutzerdefinierten Namen auf Deklariertheit und Doppeldeklaration.

Hierbei sei erwähnt, daß Konstanten (z.B. !<a,b,c,<1,2,3,4>,d> oder !T) modifiziert als String in den Syntaxbaum eingetragen werden.

4.1.2 Aufbau des Syntaxbaumes

Wie bereits oben erwähnt, wird für jede benutzerdefinierte Funktion durch den Parser in einem Pass ein eigener Syntaxbaum aufgebaut. Es bestünde nach Erzeugung des ersten Baumes die Möglichkeit, bereits jetzt mit der Codegenerierung zu beginnen. Dabei stellt sich jedoch folgendes Problem. Wird beispielsweise in der letzten benutzerdefinierten Funktion ein syntaktischer Fehler entdeckt, so sind bereits alle vorher definierten Funktionen in der Zieldatei erzeugt worden. In diesem Fall wäre unnötig produziert worden, da der Programmierer zuerst seinen Fehler korrigieren muß. Die Lösung dieses Problems besteht darin, den Parser vollständig durchlaufen zu lassen und die einzelnen Syntaxbäume in einer verketteten Liste abzuspeichern. Wird dann in einer Funktion ein Fehler entdeckt, bricht der Compiler mit einer Parserfehlermeldung ab. Die Liste wird dabei einfach aus dem Hauptspeicher entfernt.

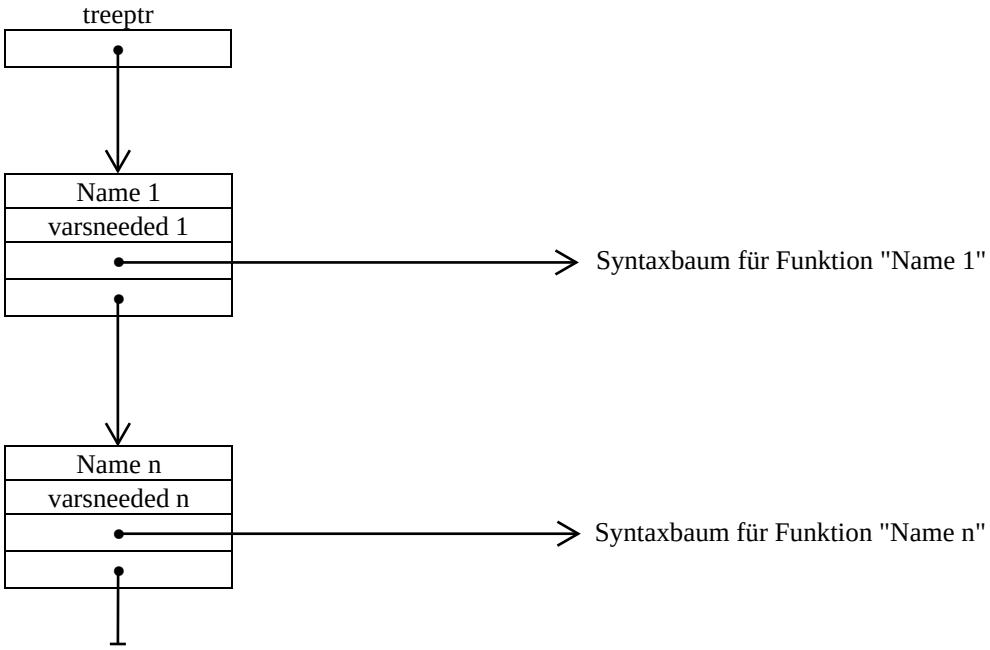


Abb. 4.1: Liste der Syntaxbäume

Kurz sei hier noch auf die Behandlung eines Spezialfalles eingegangen. Tritt der Umstand ein, daß eine Funktion wie folgt definiert wird

def Name1 = Name2

und ist die aufgerufene benutzerdefinierte Funktion "Name2" vom Typ 't', so trägt der Parser als Typ der Funktion "Name1" trotzdem 'f' in die Namensliste ein. Dies ist durch die Grammatik bestimmt und muß folglich später korrigiert werden. Durch das Hauptprogramm erfolgt daher eine Verifikation der oben genannten Liste, bei der nach diesem Umstand gesucht und bei positivem Ergebnis der Typ der Funktion "Name1" in 't' umgewandelt wird.

4.1.2.1 Datenstrukturen

Die Knoten des Syntaxbaumes bestehen aus zwei Teilen, dem Typ und dem Eintrag. In der folgenden Tabelle werden jedem Typ die möglichen Einträge zugeordnet.

Typ	**Eintrag**
COND, C&P, S&G	3 Zeiger auf Knoten für die 1., 2., und 3. Stelle
WHILE, BU, BUR, COMP, CONSTR, CONSTRSEP, SEMI,	2 Zeiger auf Knoten für die 1. und 2. Stelle
ALPHA, LINSERT, RINSERT, REDUCE	1 Zeiger auf Knoten für die 1. Stelle
SEL, RSEL, RESIZE	Integer-Variable, gibt den jeweiligen Selektor bzw. das Argument von RESIZE an
CONSTANT	String-Variable
TRCALL, FNCALL	String-Variable, enthält den Namen der angewandten festimplementierten 2DT-FP-Funktion
CALL	Struktur, enthält String-Variable für den Namen und Integer-Variable für die Zeilennummer der aufgerufenen benutzer-definierten Funktion in der Quelldatei

Der Typ CONSTRSEP hat lediglich eine interne Bedeutung und sei hier nur der Vollständigkeit halber erwähnt.

Erkennt der Parser beim Reduzieren der Grammatik beispielsweise ein "→", so ruft er die entsprechende Funktion auf und erzeugt einen Knoten vom Typ COND. Danach wird der Knoten, der für die erste Stelle von "→" generiert wurde, an den ersten Zeiger angehängt.

Nehmen wir folgende Funktionsdefinition (*lokal* sei eine beliebige benutzerdefinierte FP-Funktion, definiert in Zeile 10 des Programms):

def Global = [lokal o 2 , 3] —c&p→ { id == C2R }

Den hierfür erzeugten Syntaxbaum gibt Abb. 4.2 wieder.

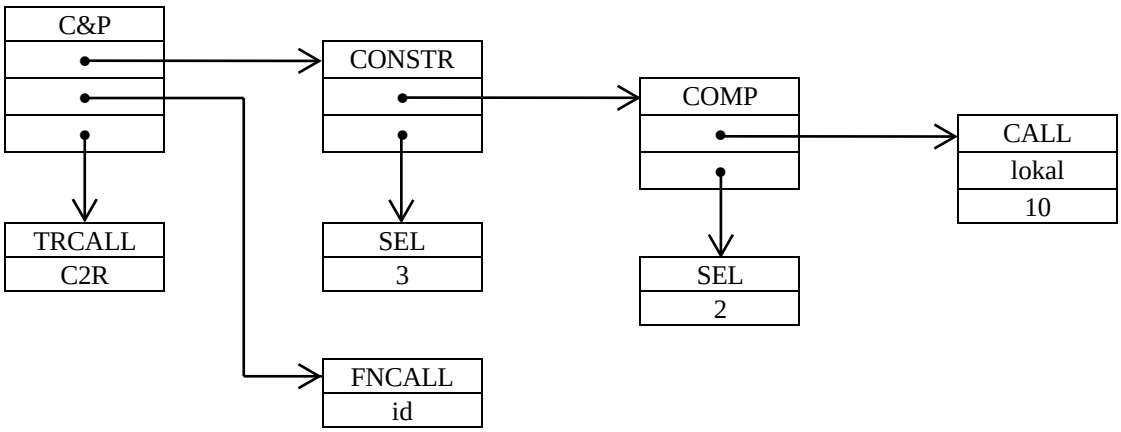


Abb. 4.2: Syntaxbaum zu der benutzerdefinierten Funktion *Global*

4.1.2.2 Speicherorganisation

Die Speicherallokation wurde mit Hilfe der C-Bibliotheksfunktion "malloc ( )" vorgenommen. Für jeden benötigten Knoten stellt "malloc ( )" genug Speicher zur Verfügung und trägt die Anfangsadressen sowie die Größe in eine Liste ein. Nach Beendigung der Produktion ist der Compiler fertig und das Betriebssystem gibt den allokierten Speicher wieder frei.

4.1.3 Vorbereitung der Zieldatei

Nach dem erfolgreichen Öffnen der Zieldatei *filename.c* wird zunächst ein Dateikopf erzeugt. Dieser beinhaltet unter anderem Präprozessoranweisungen, Variablendeklarationen und die Deklaration aller benutzerdefinierten Funktionen aus *filename.2dtfp*. Nun beginnt die eigentliche Übersetzung der Funktionen (siehe 4.1.4).

Schließlich fehlt nur noch das Hauptprogramm, in welchem die Hauptfunktion aufgerufen wird. Die Struktur des Dateikopfes und des Hauptprogrammes sind statisch festgelegt, d.h. sie verändert sich nicht. Daher veranschauliche man sich die Produktion an der Beispielübersetzung im Anhang Teil E.

4.1.4 Codegenerierung

Bei der Codeerzeugung unterscheidet der Compiler drei Arten von benutzerdefinierten Funktionen:

- Funktionen vom Typ 'f', die nur aus festimplementierten FP-Funktionen, Funktionalformen und benutzerdefinierten Funktionen des Typs 'f' bestehen. Von der Semantik her umfaßt diese Klasse genau die 2DT-FP-Funktionen, die streng lokal arbeiten.
- Funktionen vom Typ 't'. Diese Klasse umfasst genau die Funktionen, die nicht vom Typ 'f' sind, also die globalen Funktionen und Transformationen.
- die Hauptfunktion. Per Konvention ist das die letzte benutzerdefinierte Funktion in der Quelldatei *filename.2dtfp*.

Die Entscheidung, in welcher Art und Weise übersetzt wird, gibt das Diagramm in Abbildung 4.3 an.

Hierbei ist zu beachten, daß das mittlere Feld "Benutzerdefinierte Funktion" als Startpunkt verwendet wird. Davon ausgehend, führen die Pfeile unter Berücksichtigung des Typs zu den Produktionsaktionen bzw. zur Selektierung der Hauptfunktion. Dies sollte genügen, um das Diagramm zu verstehen.

Im Anhang Teil E ist die vollständige Übersetzung eines Beispielprogramms angegeben.

4.1.4.1 Codegenerierung für Funktionen des Typs 'f'

Da FP als lokale Basissprache gewählt wurde, bietet sich an, auch bei deren Übersetzung vollständig auf parallele Konstrukte zu verzichten. Das heißt, die Portierung als einfacher FP-Compiler auf einen Nicht-Parallelrechner ist ohne große Mühe leicht möglich.

Grundsätzlich erhält die übersetzte Funktion einen Zeiger auf das Datenobjekt, auf das die Funktion angewendet werden soll. Wie dies im einzelnen geschieht und wie die Datenobjekte genau aussehen, ist unten im Abschnitt "Laufzeitsystem" erläutert.

Zuerst wird in die Zieldatei ein Funktionskopf mit den erforderlichen Deklarationen geschrieben. Diese Deklarationen umfassen erstens die des Argumentes (Zeiger auf ein Datenobjekt) und zweitens die der benötigten Zeigervariablen auf die Struktur eines Datenobjektes. Die Anzahl der Zeigervariablen entnimmt der Compiler der oben genannten Liste. Nun beginnt die eigentliche Abarbeitung des Syntaxbaumes. Für die festimplementierten FP-Funktionen muß lediglich ein Funktionsaufruf in die Zieldatei geschrieben werden. Links vom Gleichheitszeichen steht dann eine der Zeigervariablen. Diese wird aus der Liste der freien Variablen gelöscht. Also ist es für diesen Fall die Hauptaufgabe des Übersetzers, die Funktionalformen in der Zieldatei zu erzeugen. Daher besitzt der Compiler in seinem Quellcode eine eigene Produktionsfunktion für jede Funktionalform.

Wird im Syntaxbaum beispielsweise als Typ COMP festgestellt, so ruft der Compiler die entsprechende Produktionsfunktion auf. Diese wiederrum definiert, wenn nötig, in der Zieldatei Zeigervariablen, und ersucht schließlich den Compiler, den Code der letzten Stelle im Syntaxbaum zu produzieren. Wenn dies erledigt ist, wird der Code der zweiten Stelle erzeugt. Nachdem diese Produktionsfunktion abgearbeitet worden ist, kann eine

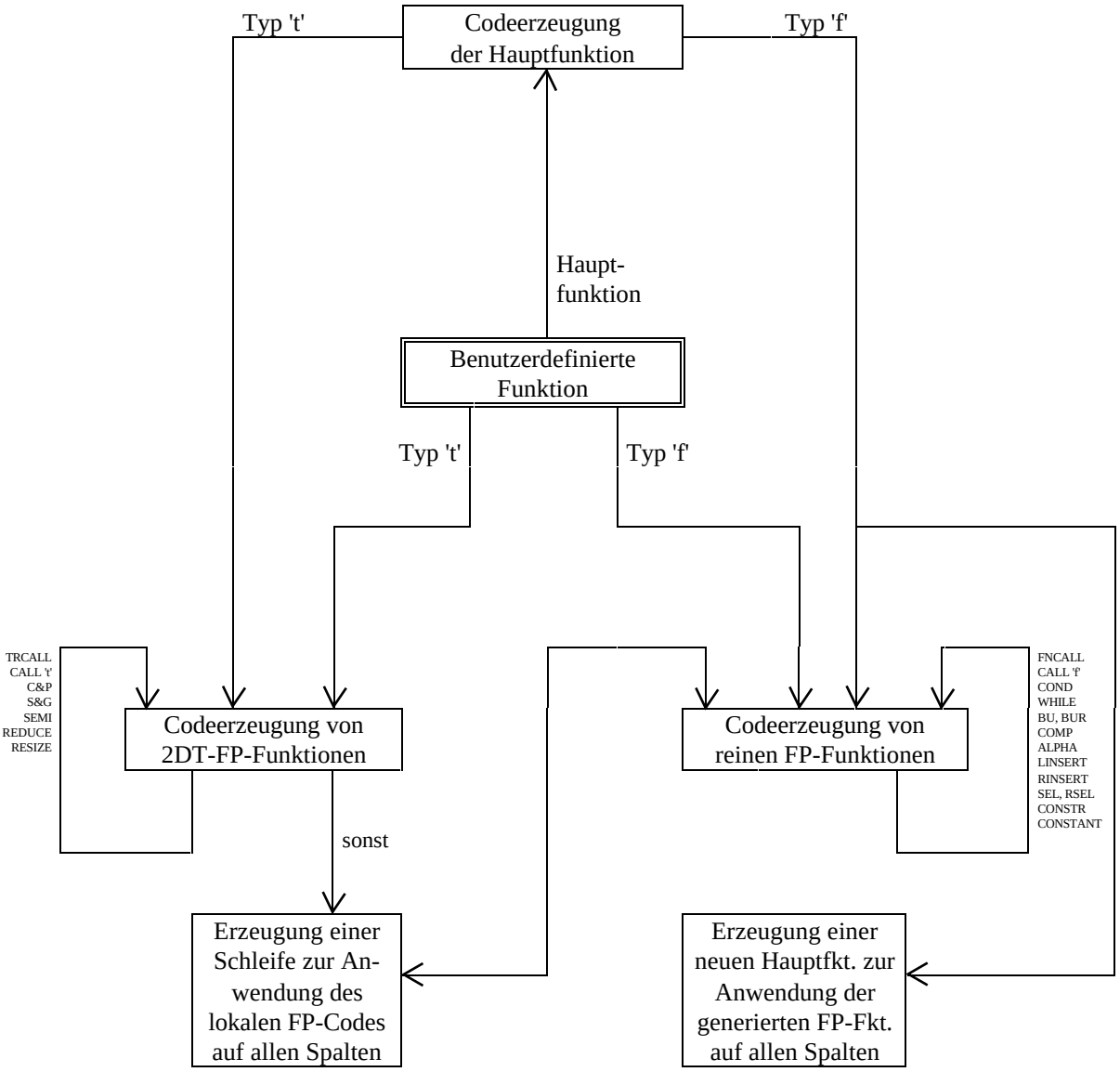


Abb. 4.3: Grobstruktur der Übersetzung

eventuell aufrufende Produktionsfunktion weiterarbeiten. Dieses permanente tiefergeschachtelte Aufrufen der Produktionsfunktionen steuert eine ihnen übergeordnete Überwachung. Die Verwaltung und Definition der Zeigervariablen wird jedoch dynamisch von den Produktionsfunktionen selbst koordiniert.

Bei der Syntaxbaumerzeugung wurde bereits erwähnt, daß Konstanten als modifizierter String im Baum abgespeichert werden. Im Laufzeitsystem wurde eine Funktion "uncompress" implementiert, die einen solchen String als Eingabe erhält und entpackt. Resultat ist ein Zeiger auf die der Konstanten entsprechende Datenstruktur.

In der Beispielübersetzung in Anhang E kann man Funktionen des Typs 'f' am Resultatstyp "fpdata" erkennen.

4.1.4.2 Codegenerierung für Funktionen des Typs 't'

Benutzerdefinierte Funktionen dieser Klasse arbeiten nicht lokal, indem sie einen Zeiger als Parameter erwarten, sondern rein global auf den Spalten. Folglich erhalten die übersetzten Funktionen kein Argument. Die Arbeitsweise der Produktionsfunktionen für Transformationen unterscheidet sich von denen für Funktionalformen nur geringfügig.

Solange die Söhne im Syntaxbaum keine FP-Funktionsaufrufe oder Funktionalformen sind, werden die Produktionsfunktionen für Typ 'f' nicht aufgerufen. Erwähnenswert ist auch, daß bei Transformationen in der Zieldatei keine Zeigervariablen im obigen Sinn benutzt werden. Es reicht also bei einem Vorkommen einer argumentlosen Transformation (z.B. C2R) aus, einfach einen Aufruf der entsprechenden Laufzeitfunktion zu erzeugen.

Hat die geforderte Transformation mindestens eine Argument, so kann es sich dabei ja um eine Funktionalform oder eine FP-Funktion (benutzerdefiniert oder nicht) handeln. Wenn dieser Fall eintritt, erfolgt der Sprung zu den reinen FP-Produktionsfunktionen. Die Diskrepanz zwischen lokalem und globalem Gültigkeitsbereich löst sich mittels einer Schleife, die den produzierten FP-Code umfaßt. Der "lokale" Code wird dabei auf alle Spalten einer Gruppe iterativ angewendet.

4.1.4.3 Codegenerierung für die Hauptfunktion

Zwei Punkte sprechen für die Notwendigkeit der gesonderten Behandlung der Hauptfunktion:

- Die generierte 2DT-FP-Hauptfunktion muß der Laufzeit-Hauptfunktion bekannt sein und wird von ihr ohne Parameter aufgerufen. Dies ist ein internes Problem und wird hier nicht weiter vertieft.
- Ist die 2DT-FP-Hauptfunktion vom Typ 'f', so ist das gesamte 2DT-FP-Programm ein einfaches FP-Programm, d.h. es enthält keine globale Transformation bzw. Funktion. Folglich muß eine vollkommen neue, für den Programmierer unsichtbare, Laufzeit-Hauptfunktion erzeugt werden, welche die lokalen Produktionen auf alle Spalten anwendet und somit sozusagen "global macht".

Verbleiben wir also bei Punkt zwei. Zum besseren Verständnis sei an dieser Stelle explizit darauf hingewiesen, daß die 2DT-FP-Hauptfunktion immer als letzte übersetzt wird. Falls sie vom Typ 't' ist, unterscheidet sich deren Übersetzung nicht von anderen Funktionen dieser Klasse. Einziger Unterschied ist der Aufruf im Laufzeit-Hauptprogramm. Auch im anderen, interessanteren Fall ist die Übersetzung Klassenkonform. Es muß dann jedoch eine neue Laufzeit-Hauptfunktion generiert werden, die mit "_HauptfunktionFP" konkateniert ist. Sie enthält eine Schleife in dessen Korpus die eigentliche 2DT-FP-Hauptfunktion für alle Spalten aufgerufen wird. Mit diesem Vorgehen haben wir quasi eine künstliche Funktion vom Typ 't' geschaffen, die lediglich lokalen FP-Code enthält.

4.1.5 Codeoptimierungen

In folgenden vier Fällen optimiert der Compiler automatisch:

1. Composition

o (...) %% o (...) !x o (...) (x sei ein bel. FP-Objekt)

Die FP-Funktionen und Funktionalformen *#, %%, !* bekommen ein FP-Objekt als Argument, das nicht weiterverwendet wird. In unserer Implementierung löschen diese Funktionen daher das Argument. Folglich ist es auch unnötig, vorher irgendetwas auszurechnen, wenn obige Funktionen durch "o" mit anderen verknüpft sind. Deshalb produziert der Compiler nur den Code für die Funktion *#*, *%%* oder die Funktionalform *!*.

2. Construction

[...,# ,...] [...,%% ,...] [...,!x ,...] (x sei ein bel. FP-Objekt)

Nach Definition der Funktionalform *Construction*, muß für jede enthaltene Funktion die gleiche Eingabe bereitgestellt werden. Da alle festimplementierten FP-Funktionen ihre Eingabestruktur löschen oder umbauen, ist es zwingend nötig für jede in *Construction* enthaltene Funktion die Eingabe zu kopieren. Bei einem Vorkommen der o.g. Funktionen *#*, *%%* und der Funktionalform *!* verzichtet der Compiler auf die Kopie der Eingabestruktur aus den gleichen Gründen wie oben.

3. Cut&Paste

f —c&p→ { g == h } und mindestens eine der 2DT-FP-Argumentfunktionen f,g oder h ist die Identität *id*

Dieses alleinstehende *id* würde, wenn denn normal produziert wird, auf jede Spalte unnötigerweise angewendet. Aus diesem Grund generiert der Compiler den Code für *id* nicht.

4. Split&Glue

f —s&g→ { g || h } und mindestens eine der 2DT-FP-Funktionen g oder h ist gleich *id*

Die Optimierung und Begründung ist identisch zu Cut&Paste. Die Prädikatsfunktion f darf, falls sie gleich id ist, nicht wegoptimiert werden, da ihr Ergebnis anschließend weiterverwendet wird.

4.2 Laufzeitsystem

In den nächsten Abschnitten wird die Implementierung der Sprachkonzepte von 2DT-FP beschrieben.

4.2.1 Gruppenkonzept

Zu Beginn der Programmausführung existiert immer genau eine Gruppe, der alle Prozesse angehören. Es gibt bei Programmstart mindestens einen Prozeß. (Selbst bei leerer Eingabe wird ein Prozeß erzeugt, der eine leere Sequenz darstellt.) Durch Anwendung der globalen Konstruktoren können neue Gruppen erzeugt werden, deren Prozesse verschiedene Aufgaben ausführen. Zur Abarbeitung der Gruppen haben wir die depth-first Strategie verwendet. Dies bedeutet, daß zu einem bestimmten Zeitpunkt immer nur eine Gruppe aktiv ist. Bei Aufteilung einer Gruppe in zwei neue Teilgruppen werden zuerst die Funktionen auf der linken Teilgruppe ausgeführt, dann die Funktionen auf der rechten Gruppe und schließlich werden beide Gruppen wieder zu einer vereinigt.

Obwohl die depth-first Strategie den Gruppenparallelismus der Sprache 2DT-FP, also die parallele Abarbeitung mehrerer Gruppen, nicht unterstützt, ergeben sich doch einige Vorteile gegenüber der breadth-first Strategie. Die sonst erforderliche Synchronisation und Kommunikation zwischen den verschiedenen Gruppen sowie der Umgang mit koexistierenden Nachrichten von Transformationen aus verschiedenen Gruppen entfällt. Da alle Prozessoren gleichzeitig auf die Bearbeitung einer einzigen Gruppe konzentriert sind, liegt das Ergebnis für diese Gruppe früher vor als bei Verwendung der breadth-first Strategie.

Ein weiterer Geschwindigkeitsvorteil wird durch ein Abweichen von der strengen depth-first Abarbeitung bei den Transformationen Split&Glue bzw. Cut&Paste erreicht. Hier darf ein Prozessor bereits mit der nächsten Gruppe beginnen, sobald er mit der Ausführung seiner Funktionen auf der ersten Gruppe fertig ist, auch wenn die anderen Prozessoren noch nicht so weit sind. (Vergleiche hierzu Kapitel 4.2.7, Synchronisation.)

Da es im Allgemeinen in einer Gruppe mehr Prozesse gibt als Prozessoren vorhanden sind, müssen die Prozesse den einzelnen Prozessoren zugeteilt werden. Dies erfolgt nach einer festgelegten Strategie, die in Kapitel 4.2.3, Allokationsstrategie, beschrieben wird. Jeder Prozessor besitzt aus einer Gruppe nur die Daten seiner internen Prozesse, d.h. der Prozesse, die er bearbeiten soll. Zur Programmausführung benötigt ein Prozessor allerdings auch Informationen über die Struktur der gesamten Gruppe. Für jede Gruppe werden daher folgende Informationen gespeichert und, falls notwendig, aktualisiert:

- Gesamtanzahl der Prozesse der Gruppe
- Anzahl der internen Prozesse
- Höhe der lokalen Daten (Spaltenhöhe)
- Informationen über die momentane Allokation der Prozesse auf den Prozessoren. Diese wird in einer Tabelle gespeichert, der VAPT (virtual-to-abstract-processor table).
- Daten der internen Prozesse

4.2.2 Datenstrukturen

Zur Implementierung der 2DT-FP-Funktionen mußte eine geeignete Datenstruktur gefunden werden, die die Darstellung der Gruppen, der Prozesse (Spalten der 2D-Arrays) und der verschiedenen Datenobjekte (Atome, Sequenzen) ermöglicht. Hier wurden mehrfach verkettete Listen, wie in Abbildung 4.4 dargestellt, gewählt.

Die oberste Ebene enthält alle existierenden Gruppen mit den dazugehörigen Informationen (siehe oben). Vp ist die Anzahl aller Prozesse, ip die der internen Prozesse, hld die Spaltenhöhe. Der Zeiger *groups* zeigt auf die momentan aktive Gruppe, die gleichzeitig den Kopf dieser Liste aller Gruppen bildet. Entstehen bei Split&Glue bzw. Cut&Paste neue Gruppen, so wird die neue Gruppe immer vorne in die Liste eingefügt und wird somit zur aktiven Gruppe. Um die Funktionen der zweiten Gruppe anzuwenden, werden die beiden vordersten Gruppen in der Liste vertauscht, so daß dann die zweite Gruppe aktiv ist.

Die Liste der Gruppen ist bei allen Prozessoren identisch mit Ausnahme der Anzahl der internen Prozesse und der Prozeßdaten, die sich natürlicherweise unterscheiden müssen. Ein Zeiger jedes Gruppenverbundes (*next_grp*) zeigt auf die nächste Gruppe, ein anderer (*col*) auf den ersten internen Prozeß dieser Gruppe. Besitzt der jeweilige Prozessor keinen Prozeß aus dieser Gruppe, so ist der Zeiger *col* NULL.

Die Liste der internen Prozesse ist folgendermaßen aufgebaut: Jeder Prozeß besteht aus seiner Nummer, den Prozeßdaten und einem Zeiger *next_col* auf den nächsten Prozeß. Die Daten eines Prozesses sind entweder ein Atom oder eine (evtl. leere) Sequenz. Um hierfür die gleiche Darstellung wie für alle anderen Datenobjekte benutzen zu können, enthält ein Prozeß lediglich einen Zeiger *data* auf ein solches Datenobjekt. Der für andere Datenobjekte benötigte Zeiger *next_obj* wird hier nicht gebraucht und daher immer NULL gesetzt.

Die Darstellung eines Datenobjekts erfolgt durch einen Verbund aus dem Typ des Objekts (integer, float, character, string, true, false, Nullelement '@' oder Sequenz), der Sequenzlänge (die Länge ist 0 falls das Objekt ein Atom, Nullelement oder eine leere Sequenz ist), dem Inhalt und einem Zeiger *next_obj* auf das nächste Objekt innerhalb der gleichen Sequenz. Der Inhalt wird dargestellt durch

- einen einfachen Datentyp, falls das Objekt ein Atom des Typs integer, float, character oder string ist ;
- einen NULL-Zeiger, falls das Objekt true, false, @ oder eine leere Sequenz ist ;
- einen Zeiger auf das erste Element der Sequenz, falls das Objekt eine nicht-leere Sequenz ist.

Ein ausführliches Beispiel hierzu enthält Abbildung 4.4 . Sie gibt die Datenstruktur des Prozessors 0 wieder unter der Annahme, daß zwei Gruppen existieren, Gruppe 1 aktiv ist und folgende Prozeßallokation besteht:

Gruppe 1

VAPT:

Prozeß	0 1 2 3 4 5
Prozessor	0 1 2 3 0 1

Gruppe 0

VAPT:

Prozeß	0 1 2 3 4 5 6 7
Prozessor	0 0 1 0 2 3 0 1

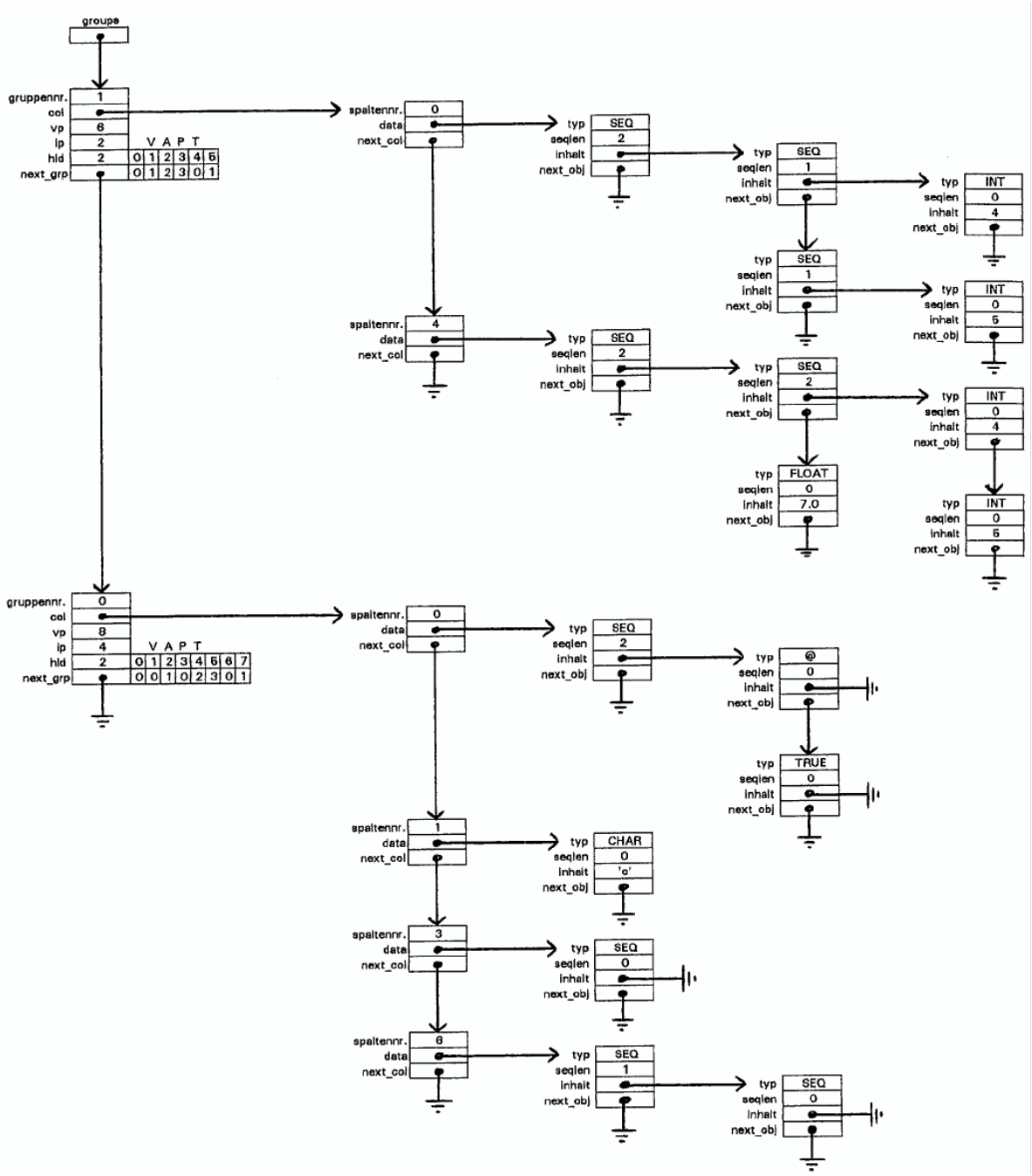


Abb. 4.4: Beispiel für den Aufbau der Datenstruktur

Die im Beispiel dargestellten Prozeßdaten (Spalten) sind folgende Objekte:

Prozeß aus Gruppe 1	Objekt
0	<<4>,<5>>
4	<<4,5>,7.0>>

Prozeß aus Gruppe 0	Objekt
0	<@,T>
1	'c'
3	< >
6	<< >>

4.2.3 Allokation

Mit Allokation bezeichnet man die Verteilung der Prozesse auf die vorhandenen Prozessoren. Die momentane Allokation wird, wie bereits oben erwähnt, in einer Tabelle, der VAPT (virtual-to-abstract-processor table), gespeichert. Jeder Prozessor besitzt eine VAPT zu jeder existierenden Gruppe.

Der von uns ausgewählten Allokationsstrategie liegen folgende Überlegungen zugrunde: Hauptziel ist es, die Kommunikation zwischen den Prozessoren zu minimieren. Da der größte Teil des Kommunikationsaufkommens durch das Umverteilen von Daten zwischen den Prozessen verursacht wird, geht dies einher mit der Zielsetzung, die Prozeßmigration möglichst gering zu halten.

Der Nachteil dieser Allokationsstrategie besteht darin, daß gegebenenfalls ein schlechtes Load-Balancing, d.h. eine ungleichmäßige Verteilung der Prozesse auf die Prozessoren auftreten kann. Das bedeutet, daß sich die Anzahl der internen Prozesse zwischen zwei Prozessoren um mehr als 1 unterscheiden kann.

Nach dem Einlesen der Eingabedaten sind die Prozesse nach folgender Formel verteilt: der Prozeß mit der Nummer i läuft auf Prozessor i modulo n, wobei n die Prozessoranzahl ist. Die bestehende Allokation wird von allen Transformationen außer Split&Glue und RESIZE(p) beibehalten. Bei den beiden Transformationen haben wir im Sinne des Vermeidens größerer Prozeßmigration auf eine Beibehaltung der aktuellen Allokation verzichtet, da sonst umfangreiche Datenumverteilungen hätten vorgenommen werden müssen. Für eine genauere Beschreibung der gewählten Vorgehensweisen siehe Kapitel 4.2.6, Implementierung der globalen 2D-Transformationen.

4.2.4 Einlesen der Programmeingabe

Bei Programmstart erzeugt jeder Prozessor eine Gruppe mit der Nummer 0. Anschließend öffnet er die Eingabedatei und liest die Programmeingabe mit einem Scanner (von dem GNU Programmierwerkzeug FLEX erzeugt). Durch einen von BISON erzeugten Parser testet er die Eingabe auf Korrektheit und baut gleichzeitig die Datenstruktur entsprechend der Beschreibung aus Kapitel 4.2.2 für seine internen Prozesse auf. Laut oben beschriebener Anfangsallokation besitzt ein Prozessor die Prozesse, deren Spaltennummer i modulo Prozessoranzahl n gleich seiner Prozessornummer ist.

Nun werden noch die Gruppeninformationen wie Anzahl der Prozesse, Anzahl der internen Prozesse und die VAPT in den Gruppenverbund eingetragen. Die erste VAPT kann sich jeder Prozessor anhand der Formel VAPT[i] = i modulo n selbst erstellen.

War die Eingabedatei leer, so erzeugt nur Prozessor 0 einen Prozeß, der die Nummer 0 und die leere Sequenz als Datum erhält. War die Eingabe falsch, wird das Programm abgebrochen.

4.2.5 Implementierung der FP-Funktionen

Eine FP-Funktion ändert an der Struktur einer Gruppe nichts. Die Anwendung einer FP-Funktion auf alle internen Prozesse einer Gruppe erfolgt durch eine Schleife, die über alle internen Spalten läuft und darauf sequentiell die FP-Funktion anwendet.

Struktur der Schleife:

```
sptr = groups→ col ;   /* sptr zeigt auf den ersten internen Prozeß der aktiven Gruppe */
while (sptr != NULL)  /* führe folgende Anweisungen für alle internen Prozesse aus */
{
  wende FP-Funktion auf die Prozeßdaten sptr→ data an ;
  sptr = sptr→ next_col ;
}
```

Jede FP-Funktion erhält bei ihrem Aufruf einen Zeiger auf das Datenobjekt, auf das die Funktion angewandt werden soll. Dies ist bei den meisten Funktionen der Kopf einer Sequenz, kann jedoch auch ein einzelnes Atom sein (z.B. bei iota). Als Rückgabewert liefern alle Funktionen einen Zeiger auf das Objekt, welches das Ergebnis der Funktionsanwendung enthält. Die der Funktion übergebene Datenstruktur wird umgebaut, der nicht mehr benötigte Speicherplatz wird freigegeben.

4.2.6 Implementierung der globalen 2D-Transformationen

4.2.6.1 Kommunikation

Globale Transformationen verändern die Gesamtstruktur einer Gruppe durch Daten- und Prozeßumverteilungen und erfordern daher Kommunikation zwischen den Prozessoren.

Da es sich bei dem Hypercube um einen distributed memory - Rechner handelt, ist es nicht möglich, eine Datenstruktur, die aus verketteten Listen besteht, in unveränderter Form komplett zwischen Prozessoren zu verschicken. Also müssen die zu versendenden Daten in eine andere Darstellung übertragen werden.

Für das Versenden von Daten an externe Prozesse besitzt jeder Prozessor je einen Ein- und Ausgabepuffer für jeden anderen Prozessor. Diese Puffer sind als Feld von "void" der festen Größe BUFSIZE (definiert in der Datei *daten.h*) deklariert und können somit zum Speichern beliebiger Daten verwendet werden. Sie dienen hauptsächlich dazu, die Daten bei Datenumverteilungen zu sammeln und dann in einem großen Paket an den Empfängerprozessor zu verschicken.

Es gibt zwei verschiedene Haupttypen von Nachrichten:

1. Es sollen einzelne Datenkomponenten einer Spalte an eine Komponente in einer anderen Spalte, die sich auf Prozessor i befindet, geschickt werden. Der Ausgabepuffer out[i] ist entsprechend der Darstellung in Abbildung 4.5 aufgebaut. Die Nachrichten besitzen den Identifier 'D' (Distribute).

2. Es soll eine ganze Spalte an einen anderen Prozessor verschickt werden. Da eine Spalte auch nur eine Sequenz ist, ist der Ausgabepuffer ähnlich wie im ersten Fall

aufgebaut mit dem Unterschied, daß der Message-Identifier statt des 'D' im ersten Fall ein 'A' (Allocate) ist. Die Angabe der Zielkomponente entfällt.

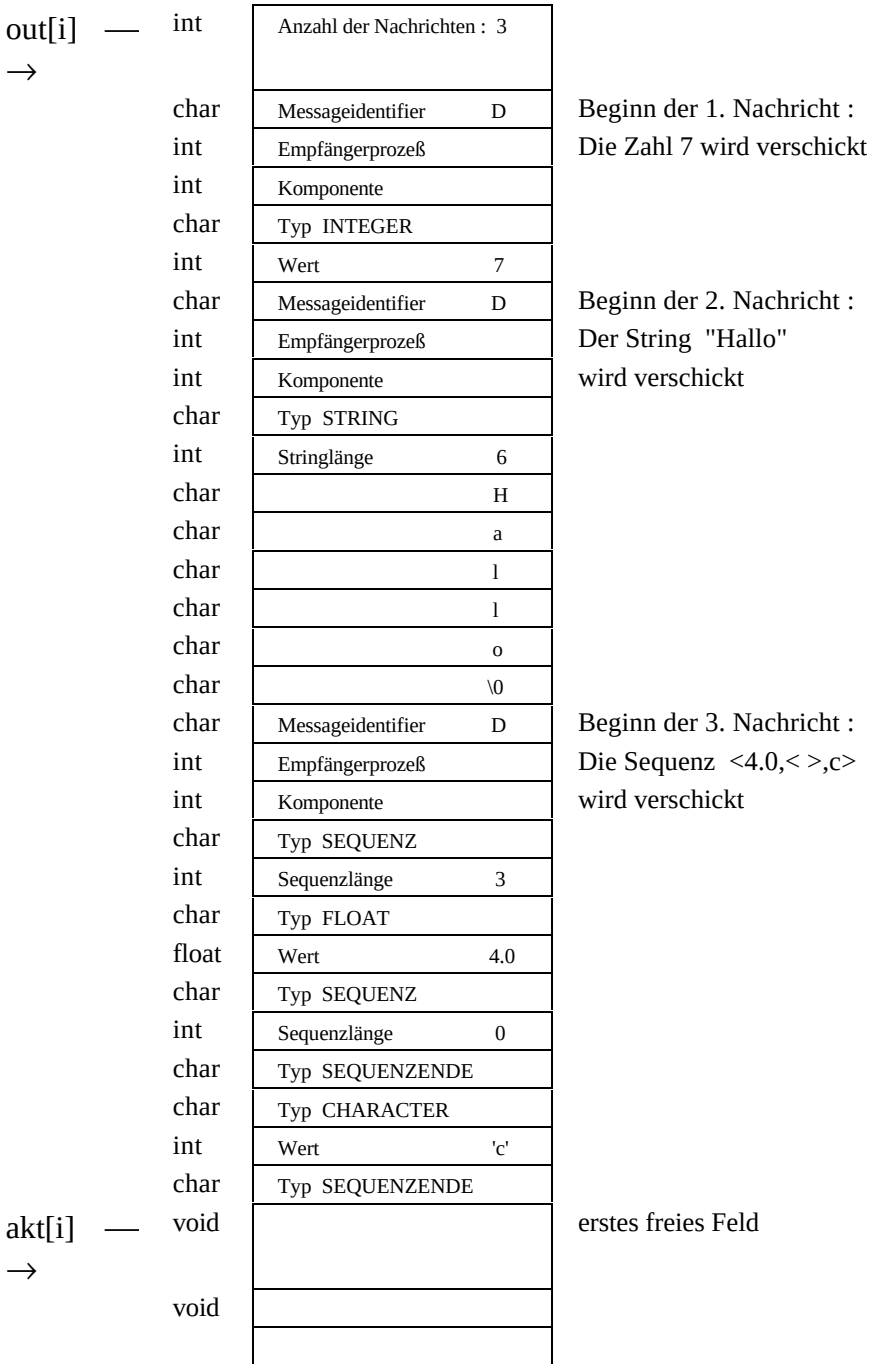


Abb. 4.5: Beispiel für den Aufbau eines Ausgabepuffers

Sollen Daten lediglich zwischen den internen Prozessen eines Prozessors umverteilt werden, so ist obiges Verfahren zu aufwendig, da die Daten in eine ganz andere Struktur im Ausgabepuffer übertragen werden und von dort anschließend wieder in die vorherige Darstellung entpackt werden müssen. Es bietet sich hier eine viel schnellere Lösung an und zwar müssen lediglich die entsprechenden Datenzeiger im Speicher des Prozessors umgehängt werden. Für diese Art des Datenaustauschs dient die Liste mit dem Kopf *intmsg*, in der die Zeiger auf die Datenobjekte zwischengespeichert werden.

Das Erzeugen der Nachrichten zur Umverteilung einzelner Datenkomponenten bzw. ganzer Spalten übernehmen zwei verschiedene Funktionen, die anhand der VAPT entscheiden, ob Nachrichten für interne oder externe Prozesse erzeugt werden müssen.

Der Austausch der Nachrichten erfolgt durch die Funktion "Total Exchange", die den Inhalt aller Ausgabepuffer an die jeweiligen Empfängerprozessoren verschickt. Anschließend entpackt jeder Prozessor die empfangenen Nachrichten und baut die den Daten entsprechende Datenstruktur auf.

Neben dem Zweck der Datenumverteilung dienen die Ein- und Ausgabepuffer auch für die Übermittlung anderer Nachrichten, die bei einigen Transformationen anfallen um Vorberechnungen auszuführen (vgl. Kapitel 4.2.6.2). Der Aufbau der Messagepuffer ist dann speziell an den jeweiligen Zweck angepaßt und unterscheidet sich von obiger Darstellung.

4.2.6.2 Auszeichnung eines Kontrollprozessors

Grundsätzlich gilt bei allen Transformationen, daß jeder Prozessor die notwendigen Berechnungen (z.B. die neue Allokation, die neue VAPT) selbständig durchführen soll. Bei manchen Transformationen ist das jedoch nicht möglich, da weitere Informationen über die Prozesse der anderen Prozessoren und deren Datenverteilung benötigt werden (vgl. hierzu z.B. PACK oder SMOOTH). In diesen Fällen werden solche "globalen" Informationen durch einen ausgezeichneten Kontrollprozessor gesammelt (Ausführung eines "Gather") und nach ihrer Auswertung an alle anderen Prozessoren verteilt ("Broadcast").

Als Kontrollprozessor verwenden wir meistens den Prozessor mit der Nummer n-1 (n=Prozessoranzahl), da er im Falle einer schlechen Lastenverteilung am wenigsten Prozesse zu bearbeiten hat. Man kann jedoch ebenso einen beliebigen anderen Prozessor auszeichnen. Die Berechnung der Informationen auf dem Kontrollprozessor erfolgt vorrangig vor seinen sonstigen Aufgaben. Außerdem wurden die zu berechnenden Informationen immer minimal so ausgewählt, daß jeder Prozessor anschließend noch möglichst viel selber berechnen muß. Dadurch stehen die Ergebnisse den anderen Prozessoren schneller zur Verfügung und ihr "Leerlauf" wird gering gehalten. Als Beispiel sei hier die Vorgehensweise bei einer Änderung der Allokation betrachtet: Der Kontrollprozessor berechnet nicht vollständig die neue VAPT, um sie dann zu versenden, sondern er verteilt nur die Informationen, die jeder Prozessor benötigt, um die VAPT selbst zu erstellen. Das Versenden der vollständig aufgestellten VAPT würde keine Vorteile bringen, da ein Prozessor die Zeit, die er beim Berechnen der VAPT sparen würde, für das Entpacken der VAPT aufwenden müßte.

4.2.6.3 Kommunikationsalgorithmen

1. Total Exchange (all-to-all personalized communication)

Diese Operation beschreibt eine Kommunikation, in der jeder Knoten eine Nachricht an jeden anderen Knoten sendet. Dabei schickt ein Knoten an unterschiedliche Knoten verschiedene Nachrichten (den Inhalt der jeweiligen Ausgabepuffer).

Zur Lösung dieses Problems wurde ein sehr einfacher Algorithmus herangezogen. Jeder Knoten versendet nacheinander seine Nachrichten asynchron an die Zielknoten. Die Reihenfolge für das Absenden der Messages ist auf der Grundlage des jeweiligen Hamilton-Zyklus festgelegt. (Vergleiche hierzu Abbildung 4.6 .) Man beachte hierbei, daß die letzte Nachricht, die abgesendet wird, an einen direkten Nachbarn des sendenden Knoten geht, die vorletzte an einen Nachbarn mit einer Entfernung von zwei Verbindungen, und so weiter. Hierdurch werden längere Übertragungszeiten vermieden, die durch ein Absenden der letzten Nachrichten an weit entfernte Knoten hervorgerufen werden könnten. Der interne Übertragungsweg zwischen den Prozessoren wird dem Betriebssystem überlassen, da unsere Tests ergaben, daß hierdurch im Allgemeinen keine Laufzeiteinbußen zu erwarten sind. Da auf dem Hypercube nur eine begrenzte Anzahl von Message-Identifiern zur Verfügung steht, muß zu ihrer Freigabe explizit auf die Beendigung des Nachrichtenversands gewartet werden. Dies geschieht in der gleichen Reihenfolge wie das Versenden, und auch erst, nachdem der Versand aller Nachrichten initiiert wurde. Anschließend führt jeder Prozessor die zum Empfangen der Nachrichten nötigen Operationen aus.

Andere wesentlich komplexere Algorithmen für das Total Exchange bringen nach den in diesem Praktikum gesammelten Erfahrungen keinen überragenden Laufzeitvorteil.

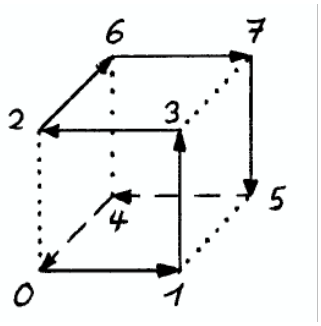


Abb. 4.6: Hamilton-Zyklus in einem Hypercube der Dimension 3

2. Broadcasting (single-node broadcast)

Unter (single-node) Broadcasting versteht man eine Kommunikation, bei der ein Knoten dieselbe Nachricht an alle anderen Knoten sendet. Der Nachrichtenversand erfolgt analog zu obigem Algorithmus für Total Exchange. Der einzige Unterschied besteht darin, daß nur eine identische Nachricht von dem Sender verschickt wird. Alle anderen Knoten führen lediglich eine Receive-Operation aus.

3. Gathering (single-node gather)

Mit Gathering bezeichnet man das Problem, daß ein bestimmter Prozessor von allen anderen Prozessoren unterschiedliche Nachrichten empfangen soll. Der von uns verwendete Algorithmus beruht darauf, daß jeder Knoten (außer dem Empfängerknoten) seine Messages direkt an den Empfänger absendet. Die Auswahl des hierfür verwendeten Weges bleibt dem Betriebssystem überlassen. Der Empfängerprozessor führt eine entsprechende Anzahl von Receive-Operationen aus, um die Nachrichten von allen anderen Prozessoren einzusammeln.

Alle von uns verwendeten Kommunikationsmodelle lassen sich in dieser Implementierung leicht durch andere Algorithmen beschreiben, da sie jeweils als Funktion ohne Argument und Resultatstyp implementiert sind. Sie haben ausschließlich die Aufgabe, den Inhalt der Ausgabepuffer an andere Knoten zu übermitteln. Ausführlich und vollständig dargestellt werden diese Kommunikationsmodelle einschließlich verschiedener Algorithmen in [BT89], [Fra89] und [HJ89].

4.2.6.4 Die Höhe der lokalen Daten

Die Spaltenhöhe (Höhe der lokalen Daten) des aktiven 2D-Arrays kann sich durch die Anwendung lokaler FP-Funktionen verändern (die Spalten sind ja nichts anderes als FP-Sequenzen). Zur Lösung dieses Problems boten sich zwei Vorgehensweisen an:

1. Man kann die Spaltenhöhe des 2D-Arrays fixieren. Dies hat den Nachteil, daß man die Nullelemente am Spaltenende nicht wegfallen lassen kann. Bei der Anwendung von FP-Funktionen wären jedesmal umfangreiche Tests nötig, um festzustellen, ob eine Spalte mit dem Auftreten eines Nullelements zu Ende ist oder ob danach noch Daten folgen. Im ersten Fall wäre die Anwendung der FP-Funktion erlaubt, im zweiten Fall müßte ein Fehler erzeugt werden. Des weiteren müßte bei jeder Funktionsanwendung, die die Spaltenlänge vergrößert, getestet werden, ob die zulässige Spaltenlänge überschritten wird, und es müßten entsprechende Maßnahmen erfolgen.

2. Die Spaltenhöhe wird vor jeder Transformation neu bestimmt. Dies erfordert zwar Kommunikation zwischen den Prozessoren, bietet aber gegenüber der ersten Alternative aus oben erwähnten Gründen eine Reihe von Vorteilen. Außerdem stellte sich heraus, daß mehr als die Hälfte aller hier implementierten Transformationen ohne Bestimmung der exakten Spaltenhöhe auskommen, wodurch sich der zu diesem Zweck erforderliche Kommunikationsbedarf weiter verringerte. Lediglich C2R, R2C, PACK, REDUCE und RESIZE(p) benötigen die genaue Spaltenhöhe.

Zur Bestimmung der Spaltenhöhe berechnet jeder Prozessor die maximale Höhe seiner internen Prozeßdaten. Anschließend wird durch den Austausch von Nachrichten das Maximum aller Spaltenhöhen ermittelt.

Alle Spalten mit einer geringeren Länge werden fiktiv mit Nullelementen aufgefüllt. Während der Ausführung der Transformation werden Nullelemente, die am Spaltenende auftreten, weggelassen. Somit kann also bereits nach Beendigung einer Transformation die Höhe der lokalen Daten nicht mehr auf dem aktuellsten Stand sein. Dies hat jedoch keine Auswirkungen, da sie vor dem nächsten Gebrauch neu berechnet wird.

4.2.6.5 Die Transformationen

Im Folgenden werden nachstehende Abkürzungen verwendet:

vp = *groups*→ *vp* = Anzahl der Prozesse

hld = *groups*→ *hld* = Höhe der lokalen Daten (nach Abgleich)

n = Anzahl der Prozessoren

1. C2R und R2C

Diese beiden Transformationen bewirken eine Umordnung der Spaltenkomponenten. Die Komponente j ($1 \le j \le hld$) des Prozesses i ($0 \le i \le vp$) wird an die Komponente com des Prozesses pro geschickt. Pro und com berechnen sich nach folgenden Formeln:

bei C2R: pro = $(i * hld + j - 1) \bmod vp$
com = $(i * hld + j - 1) / vp + 1$
bei R2C: pro = $(i + (j-1) * vp) / hld$
com = $(i + (j-1) * vp) \bmod hld + 1$

Das Erzeugen, Versenden und Entpacken der Nachrichten erfolgt, wie in allen anderen Fällen auch, durch die in Kapitel 4.2.6.1, Kommunikation, beschriebenen Funktionen.

Für Nullelemente innerhalb einer Spalte wird keine Nachricht erzeugt, der Empfängerprozeß kann beim Auspacken der Daten selber feststellen, ob innerhalb einer Spalte Nullelemente eingefügt werden müssen.

2. BROAD

Der Prozessor, der die Spalte 0 besitzt, sendet deren Daten an alle anderen Prozesse. Alle anderen Prozesse löschen ihre Daten und schreiben die von Prozeß 0 empfangenen Daten an diese Stelle. Da der gesamte Inhalt einer Spalte verschickt wird, funktioniert BROAD auch für Spalten, die nur aus einem einfachen Atom bestehen oder Nullelemente enthalten.

3. SMOOTH

Zunächst werden aus allen Spalten die Nullelemente im Spalteninnern entfernt. Alle anderen Komponenten rutschen dadurch nach vorne. Für die weitere Ausführung der SMOOTH-Operation wird ein Kontrollprozessor benötigt. Er erhält von den anderen Prozessoren Informationen über die neuen Spaltenhöhen ihrer internen Prozesse. Anhand dieser Informationen bestimmt er die durchschnittliche Spaltenhöhe und kann nun feststellen, ob ein Prozeß zuviele Daten besitzt. Falls dies der Fall ist, teilt er dem entsprechenden Prozessor mit, an welchen Prozeß bzw. an welche Prozesse er die überschüssigen Komponenten abgeben soll.

Sobald die anderen Prozessoren diese Informationen erhalten haben, beginnen sie mit dem Erzeugen der Nachrichten zur Umverteilung der Komponenten.

4. PACK

Auch PACK kommt nicht ohne einen ausgezeichneten Prozessor aus. In diesem Fall sendet jeder Prozessor die Struktur seiner internen Prozesse (z.B. (@,@,D,@,D), wobei D eine Datenkomponente ungleich @ ist) an den Kontrollprozessor. Aus allen Informationen berechnet dieser nun die neue Datenverteilung und teilt jedem anderen Prozessor genau mit, welche Spaltenkomponenten er wohin zu verschicken hat.

5. RED(f)

Die Transformation Reduce wird in drei Schritten ausgeführt. Zunächst werden die Daten ähnlich wie bei R2C umverteilt. Das Ziel ist es, jeweils die i-ten Komponenten aller Spalten in der Spalte i-1 zusammenzufassen (i=1,..,hld), um dann darauf die Funktion f anzuwenden. Ist nun allerdings die Höhe der lokalen Daten größer als die Prozeßanzahl, müssen Hilfsprozesse mit den Nummern von vp bis hld-1 erzeugt werden. Da nach Reduce alle Daten in Spalte 0 stehen müssen, werden solche Hilfsprozesse von dem Prozessor erzeugt, der die Spalte 0 besitzt. Er muß dann zwar mehr berechnen, dafür können die Ergebnisse aber später intern umgeordnet werden.

Im zweiten Schritt wird die FP-Funktion f auf jedem Prozeß (auch den Hilfsprozessen) ausgeführt. Ist jedoch die Spaltenhöhe geringer als die Prozeßanzahl, so wird die Funktion f lediglich auf den Spalten ausgeführt, deren Nummern kleiner als die Höhe der lokalen Daten sind. Die Anwendung der zweistelligen Funktion auf mehrere Daten

erfolgt mittels der FP-Funktionalform *Insert Left*. Schließlich steht dann das Ergebnis der Funktionsanwendung auf alle i-ten Komponenten jeweils als Atom in Prozeß i-1 (i=1,..,hld).

Im dritten Schritt werden diese Ergebnisse an die Komponente i des Prozesses 0 versandt. Alle anderen Prozesse enthalten keine Daten mehr. Zum Schluß werden noch die eventuell erzeugten Hilfsprozesse entfernt.

Falls in einer Spalte Nullelemente enthalten oder nicht alle Spalten gleich lang waren, trifft im Verlauf der Anwendung der Funktion f eine FP-Funktion auf dieses Nullelement und erzeugt einen Fehler.

6. DUP

Duplicate entfernt zunächst aus jeder Spalte die Nullelemente. Anschließend werden von jedem Prozeß die nötigen Kopien der Datenkomponenten erzeugt und hinten an die Spalte angehängt. Da sich die Spaltenhöhe zwischen den Prozessen jetzt sehr stark unterscheiden kann, wird die Transformation SMOOTH aufgerufen, um die Spaltenhöhen anzugleichen.

7. PERM

Da bei PERM die Nummer des Empfängerprozesses explizit angegeben ist, bereitet die Erzeugung der Nachrichten zur Umverteilung der Komponenten keinerlei Probleme. Da es egal ist, an welche Komponente des Zielprozesses die Daten geschrieben werden sollen, wird dies dem Zufall überlassen. Die Daten werden in der Reihenfolge, in der sie aus den Nachrichtenpuffern entpackt werden, einfach vorne in der Empfängerspalte eingefügt.

Nullelemente im Ausgangsarray werden nicht beachtet, somit können nach Ausführung von PERM keine Nullelemente innerhalb einer Spalte vorkommen. Dagegen können natürlich leere Spalten entstehen.

8. RESIZE(p)

Für positive p ver-p-facht sich die Prozeßanzahl. Jeder Prozessor numeriert seine internen Prozesse um, und zwar erhält Prozeß i die Nummer i∗p. Zu jedem internen Prozeß werden p-1 neue Prozesse erzeugt, die beginnend mit seiner Prozeßnummer fortlaufend durchnumeriert werden und auf dem gleichen Prozessor allokiert sind. Dies kann zwar zu einer schlechten Lastenverteilung führen, hat jedoch den Vorteil, daß keine Datenumverteilungen zwischen verschiedenen Prozessoren notwendig sind und somit keine Kommunikation erfolgen muß. Die neue VAPT kann sich jeder Prozessor selbst erstellen.

Um die Datenkomponenten neu zuzuordnen wird die Spaltenhöhe hld durch p geteilt. Alle über diese Zahl hinausgehenden Komponenten der ursprünglichen Prozesse werden an den nächsten neuen Prozeß weitergegeben, der hiervon hld/p Komponenten behält und die überschüssigen wiederum an den nächsten neuen Prozeß weiterreicht. Sind nicht genügend Daten für alle Prozesse vorhanden, so besitzen die übrigen nur eine leere Sequenz.

Für negative p werden jeweils p Prozesse mit aufeinanderfolgenden Nummern zu einem zusammengefaßt. Es muß eine Reallokation durchgeführt werden, weil es sonst Prozessoren gäbe, die keinen Prozeß mehr besitzen, während andere viele Prozesse haben. Die Allokation erfolgt in der Art, daß, wie am Programmanfang, der Prozeß mit der Nummer i dem Prozessor i modulo n zugeordnet ist. Um dies zu erreichen erzeugt jeder Prozeß Nachrichten, mit denen seine Komponenten com an die Komponente

com' = (i modulo n) ∗ hld + com

des Prozesses i/p gesendet werden (i=0,..,vp-1). Hierbei sei i die Nummer des Prozesses, der die entsprechende Nachricht erzeugt.

Die Anzahl der neuen Prozesse ergibt sich dann zu

vp' = (vp + p - 1) / p.

Anschließend erzeugt jeder Prozessor die Prozesse mit den Nummern iam + j ∗ p für iam + j ∗ p < vp' (j=0,... ; iam sei die Prozessornummer) entweder durch Umbe-nennung bereits existierender Prozesse oder durch Neuerzeugung der entsprechenden Datenstrukturen. Überflüssige alte Prozeßstrukturen werden entfernt. Dann kann "Total Exchange" ausgeführt und die Komponenten können in den neuen Prozessen abgespeichert werden.

9. Split&Glue

Die Transformation Split&Glue haben wir in drei Funktionen aufgeteilt. Nachdem das Ergebnis der Prädikatsfunktion p für alle Spalten berechnet und zwischengespeichert wurde, übernimmt die Funktion "Split( )" die Erzeugung der neuen Gruppe, die Verteilung der Prozesse auf die beiden Gruppen sowie die Herstellung einer guten Allokation der Prozesse innerhalb der Gruppen. Nach Ausführung der Funktion f auf den Prozessen der ersten Gruppe werden die beiden vordersten Gruppen in der Liste *groups* durch die Funktion "Change_groups( )" vertauscht. So kann die Funktion g auf die zweite, nun aktive, Gruppe angewendet werden. Schließlich werden beide Gruppen durch die Funktion "Glue( )" wieder vereinigt. Dazu hängt jeder Prozessor seine internen Prozesse aus der zweiten Gruppe hinter die der ersten Gruppe an.

Die Reallokation der Prozesse bei "Split( )" erfolgt nach der Vorgabe, daß sich die Anzahl der internen Prozesse innerhalb jeder Gruppe um höchstens eins unterscheiden darf. Jedoch sollen möglichst wenige Prozesse neu allokiert werden. Daher erfolgt Reallokation nur insoweit, daß die Prozessoren, die zuviele Prozesse besitzen, diese an andere Prozessoren mit zuwenig Prozessen abgeben. Hierzu müssen einige Berechnungen von einem ausgezeichneten Prozessor ausgeführt werden. Er erhält von allen anderen Prozessoren Informationen darüber, wieviele und welche Prozesse dieser Prozessor in die zweite Gruppe eingeordnet hat (die Prozesse, bei denen das Prädikat p false ergeben hat). Mit Hilfe der VAPT kann er nun bestimmen, ob Prozesse reallokiert werden müssen, um eine gleichmäßige Verteilung zu erreichen. Er sendet anschließend folgende Informationen an alle anderen Prozessoren zurück:

- Die Nummern aller Prozesse, die in der zweiten Gruppe eingeordnet wurden.
- Informationen darüber, welcher Prozessor an welchen anderen einen Prozeß abgeben muß. Es wird immer der interne Prozeß mit der größten Nummer abgegeben, damit die Berechnung der VAPT möglich ist.

Aus diesen Informationen kann jeder Prozessor die neuen VAPTs selber erstellen und, falls er davon betroffen ist, den Versand von Prozessen an andere Prozessoren initiieren.

Da bei der Ausführung von "Split( )" Gruppen entstehen können, die keinen Prozeß enthalten, muß man verhindern, daß dort Funktionen angewendet werden. Dazu wird jeweils vor der Anwendung der Funktionen f bzw. g getestet, ob die Prozeßanzahl dieser Gruppe ungleich 0 ist.

10. Cut&Paste

Die Transformation Cut&Paste wird ebenfalls in drei Funktionen aufgeteilt. "Cut( )" erzeugt die zwei Gruppen, wobei wir hier ganz ohne Kommunikation auskommen. Die Komponenten der zweielementigen Sequenz eines Prozesses der ursprünglichen Gruppe werden auf die zwei Gruppen aufgeteilt. Die neuen Prozesse bleiben dem gleichen Prozessor zugeteilt, folglich sind die VAPTs beider Gruppen identisch mit der bisherigen VAPT. Nach Ausführung der Funktionen auf der ersten Gruppe wird "Change_groups( )" aufgerufen, um die zwei vordersten Gruppen zu vertauschen und die Funktionen der zweiten Gruppe abarbeiten zu können.

Die Funktion "Paste( )" erwartet, daß die zwei zu verschmelzenden Gruppen die gleiche Anzahl von Prozessen besitzen. Die Daten der Prozesse, die die gleichen Nummern haben, werden zu einer zweielementigen Sequenz in der neuen Gruppe zusammengefaßt. Dies ist einfach, sofern sich beide Prozesse eines Paares auf dem gleichen Prozessor befinden. Ist dies nicht der Fall, so wird der gesamte Prozeß aus der zweiten Gruppe an den Prozessor verschickt, der den zugehörigen Prozeß aus der ersten Gruppe besitzt. Dann können auch diese Prozesse leicht vereinigt werden. Die VAPT sowie die anderen globalen Informationen der ersten Gruppe kann man unverändert übernehmen, alle Datenstrukturen, die die zweite Gruppe betreffen, werden entfernt.

Bei "Cut( )" kann die Situation, daß eine leere Gruppe entsteht, im Gegensatz zu "Split( )" nicht auftreten, da die Prozeßanzahl bei beiden Gruppen derjenigen der ursprünglichen Gruppe entspricht.

4.2.7 Synchronisation

Eine explizite Synchronisation innerhalb eines 2DT-FP-Programms ist nicht notwendig. Ein Prozessor kann die Programmausführung auf seinen internen Prozessen solange fortsetzen, bis er Informationen von anderen Prozessoren benötigt. An dieser Stelle, an der auch alle anderen Prozessoren Nachrichten brauchen, erfolgt eine implizite Synchronisation durch das Versenden der Messages und das Warten auf deren Empfang.

Da der Austausch von Nachrichten grundsätzlich zwischen allen Prozessoren erfolgt, muß vermieden werden, daß ein Prozessor ein Programmstück mit Nachrichtenaustausch ausführt, während ein anderer diesen Codeteil überspringt. In diesem Fall kommt es zu einem Programmabsturz. Daher muß auch ein Prozessor, der von einer Gruppe keinen Prozeß besitzt, globale Transformationen auf dieser Gruppe ausführen (mit leeren Nachrichten). Im Gegensatz dazu überspringt er dann jedoch alle lokalen FP-Funktionen, da *groups*→ *col* gleich NULL ist. Sollen in einer Gruppe also lediglich FP-Funktionen ausgeführt werden, so ist ein Prozessor, der keinen Prozeß davon besitzt, sofort fertig und kann mit der nächsten Gruppe beginnen.

Wurde dagegen der Compilerschalter '-d' für das Debuggen eines Programms aktiviert, so wird vor und nach jeder Transformation explizit synchronisiert, vergleiche hierzu Kapitel 3.5.

4.2.8 Fehlerbehandlung

Tritt bei einer FP-Funktion oder Transformation ein Fehler auf, so wird das Programm sofort abgebrochen. Mögliche Fehlerursachen beschreibt Kapitel 3.3 und ein Verzeichnis der Fehlermeldungen befindet sich in Anhang C.

Der Programmabbruch erfolgt dadurch, daß der Prozessor, bei dem der Fehler aufgetreten ist, alle existierenden Prozesse durch den Aufruf der Funktion "killcube(-1,0)" beendet.

4.2.9 Erweiterbarkeit

Die Erweiterung unserer Implementierung um zusätzliche FP-Funktionen oder 2D-Transformationen ist problemlos möglich. Geändert werden müssen folgende drei Dinge:

- der Scanner muß die neuen Funktionen erkennen
- der Parser muß die neue Funktion in den Syntaxbaum einfügen
- die Funktion muß in der Header-Datei *daten.h* deklariert werden

Der Codegenerator erzeugt den Funktionsaufruf automatisch anhand der Information im Syntaxbaum.

Bei der Implementierung der zugehörigen Laufzeitfunktion ist lediglich darauf zu achten, daß sie die übergebenen Datenstrukturen korrekt behandelt (gegebenenfalls löscht) und das Ergebnis in einer kompatiblen Darstellung zurückgibt.

Optimierungen des bestehenden Laufzeitsystems bereiten ebenfalls keine Probleme. Jede Funktion kann durch eine andere ersetzt werden, sofern diese die gleichen Datenstrukturen verarbeitet. So ist es beispielsweise möglich, andere Allokations-strategien zu realisieren oder andere Algorithmen für die Kommunikation oder für die Durchführung der Transformationen einzubauen.

A n h a n g

Teil A	Erzeugung des 2DT-FP Compilers
Teil B	Compileroptionen
Teil C	Errormessages [Compiler ⇔ Laufzeit]
Teil D	2DT-FP-Beispielprogramme
Teil E	Übersetzung des Beispielprogramms "max.2dtfp"
Teil F	Kontextfreie Grammatik der Sprache 2DT-FP
Teil G	Literaturverzeichnis

TEIL A

Erzeugung des 2DT-FP Compilers *2dtfp*

Das Verzeichnis, in welchem der Compiler erzeugt werden soll, muß folgende Dateien enthalten:

2dt.l	-	Input-File für FLEX. Wird zur Erzeugung des Scanners benötigt.
2dt.y	-	Input-File für BISON. Enthält die Grammatik und wichtige Hilfsfunktionen zur Erzeugung des Parsers.
2dtfp.h	-	Headerdatei für alle Compiler-Dateien
2dtfp.c	-	Quellcode des Compilers. Enthält u.a. den Code der Hauptfunktion und Code zur Erzeugung des Syntaxbaumes.
expr.c	-	Enthält den Code der Produktionsfunktionen
2dtfp.make	-	Makefile für die Erzeugung des Compilers

Der Befehl

make -f 2dtfp.make

erzeugt den Compiler unter dem Namen *2dtfp* im aktuellen Verzeichnis.

Das Verzeichnis, welches das Laufzeitsystem enthält, umfaßt folgende Dateien:

input.l	-	Input-File für FLEX. Dient der Generierung des Scanners für die Eingabedatei.
inout.y	-	Input-File für BISON. Enthält u.a. die Eingabe-Grammatik, Code für die Erzeugung der Eingabestruktur und für die Ergebnisausgabe.
daten.h	-	Headerdatei für alle Laufzeitdateien.
funct.c	-	Code des Laufzeitsystems für die FP-Funktionen.
transf.c	-	Code des Laufzeitsystems für die globalen Transformationen.
2dtfp	-	Compiler
code.make	-	Makefile
code	-	Shellscript zur automatischen Übersetzung eines 2DT-FP-Programms

Sei *name.2dtfp* ein 2DT-FP Quellprogramm. Dann übersetzt der Befehl

code name

den Quellcode in das ausführbare Programm *name*.

Voraussetzung für oben genannte Vorgehensweisen ist das Vorhandensein der C-Compiler *gcc* und *icc* sowie der Tools FLEX und BISON auf dem jeweiligen Rechner.

TEIL B

Compileroptionen

Der Compiler *2dtfp* stellt bestimmte Debugging-Funktionen bereit, die durch Optionsangabe zusätzlich zur Angabe der Quelldatei eingestellt werden können. Allgemein sieht die Befehlssyntax eines Compileraufrufs wie folgt aus:

2dtfp [-vd] <source>

Optionen:

-v (verbose) gibt die Versionsnummer und die Namen der Autoren aus.

-d (debugging) dient dem Aufspüren von Programmierfehlern.

Wird '-d' alleine gesetzt, so gibt der Prozessor 0 die jeweils bearbeiteten Funktionen mit ihren Eingaben auf dem Bildschirm aus. Bei Transformationen wird lediglich der Name der Transformation angegeben.
Bei Eingabe von '-d*n*' ($0 \leq n \leq 31$) gibt der Prozessor mit der Nummer *n* die entsprechenden Meldungen aus.

Vergleiche hierzu Kapitel 3.5 .

TEIL C

Errormessages

1. Fehlermeldungen des Compilers

- **Falsche Anzahl von Kommandozeilen-Parametern. Aufruf: 2dtfp [Option] <file> !**
  Ursache: Der Compileraufruf entspricht nicht der oben genannten Syntax.
  Abhilfe: Für die Compileroptionen vergleiche Anhang B.

- **Prozessorangabe zu groß (>31) !**
  Ursache: Ungültige Prozessorangabe bei der Compileroption '-d'
  Abhilfe: Bei der Option '-d' entweder gar keine Zahl oder eine Zahl n mit $0 \leq n \leq 31$ angeben.

- **Falsche Optionsangabe !**
  Ursache: Unbekannte Option wurde angegeben.
  Abhilfe: Nur erlaubte Optionen verwenden (vgl. Anhang B).

- **Optionsangabe muß mit "-" beginnen !**
  Ursache: Falscher Compileraufruf oder '-' bei der Optionsangabe vergessen.
  Abhilfe: Aufruf neu formulieren bzw. '-' hinzufügen.

- **Eingabefile darf nicht mit "-" beginnen !**
  Ursache: Das Zeichen '-' ist für Optionen reserviert, die Eingabedatei darf nicht mit '-' beginnen.
  Abhilfe: Aufruf neu formulieren bzw. '-' entfernen.

- **Kann Datei "*Name*" nicht öffnen !**
  Ursache: Eingabedatei wurde nicht angegeben, existiert nicht oder befindet sich in einem anderen Verzeichnis.
  Abhilfe: Geben Sie den Namen der Quelldatei (evtl. mit Pfadangabe) an bzw. kopieren Sie die Datei in das aktuelle Verzeichnis.

- **Doppeldeklaration: "*Name*" in Zeile *n* ist bereits definiert !**
  Ursache: Der Funktionsname "*Name*" ist bereits definiert.
  Abhilfe: Umbenennung der Funktion (vgl. Kapitel 2.4 für Namenskonventionen)

- **Nichtdefinierter Name "*Name*" in Zeile *n* !**
  Ursache: Funktion "*Name*" wird in Zeile n aufgerufen und wurde nicht definiert.
  Diese Meldung kann auch auftreten, wenn die Definition von "*Name*" einen Fehler enthielt.
  Abhilfe: Die Funktion definieren bzw. den Fehler in der Definition beseitigen.

- **parse error in Zeile *n* !**
  Ursache: Syntaxfehler in Zeile n der Eingabedatei
  Abhilfe: Fehler unter Beachtung der Syntaxregeln für 2DT-FP beseitigen (vgl. Kapitel 2)

- **Unzulässiges Zeichen in Zeile *n* !**
  Ursache: Die Quelldatei enthält ein vom Scanner nicht erkanntes Zeichen.
  Abhilfe: Ändern der Quelldatei.

- **Aufruf der benutzerdefinierten Transformation "*Name*" in Zeile *n* ! Erlaubt ist aber nur eine reine FP-Funktion !**
  Ursache: Nach Definition der Sprache 2DT-FP wird an dieser Stelle eine FP-Funktion erwartet, z.B. die Funktion f bei RED(f). Die von Ihnen definierte Funktion enthält jedoch Transformationen.
  Abhilfe: Korrigieren Sie Ihr Programm entsprechend.

- **Unerlaubter Aufruf einer Transformation !**
  Ursache: Nach Definition der Sprache 2DT-FP wird eine FP-Funktion erwartet, z.B. als Prädikat bei Split&Glue.
  Abhilfe: Korrigieren Sie Ihr Programm entsprechend.

- **Nicht genug Haupspeicher für MALLOC !**
  Ursache: malloc( ) liefert einen NULL-Zeiger zurück
  Abhilfe: Hauptspeicher freigeben, Compilierung neu starten

- **STACKÜBERLAUF bei der Funktion "*Name*" !**
  Ursache: Der Keller, der zum Aufbauen des Syntaxbaumes benutzt wird, ist nicht groß genug.
  Abhilfe: Verkleinern Sie die Definition der Funktion "*Name*", indem Sie sie auf zwei Funktionen aufgeteilen. Alternative: Vergrößern Sie die Zahl STACKSIZE in der Datei "*2dtfp.c*" und compilieren Sie den gesamten Compiler *2dtfp* neu.

- **Konstantenlänge zu groß !**
  Ursache: Sie haben eine zu lange Konstante definiert.
  Abhilfe: Verwenden Sie eine kürzere Konstante. Alternative: Vergrößern Sie die Zahl MAXCONSTLEN in der Datei "*2dtfp.c*" und compilieren Sie den gesamten Compiler *2dtfp* neu.

- **Zuviele Sequenzen in der Konstante !**
  Ursache: Innerhalb einer Konstanten sind zuviele Sequenzen ineinander geschachtelt.
  Abhilfe: Ändern Sie die Konstante ab. Alternative: Vergrößern Sie das Feld seqstack[ ] in der Datei "*2dtfp.c*" und compilieren Sie den gesamten Compiler *2dtfp* neu.

- **Einrückung ist zu groß !**
  Ursache: Sehr große Schachtelung einer benutzerdefinierten Funktion.
  Abhilfe: Funktion in zwei Funktionen aufteilen.

2. Laufzeitfehlermeldungen

- **Kann Eingabedatei "*Name*" nicht öffnen !**
  Ursache: Eingabedatei wurde nicht angegeben, existiert nicht oder befindet sich in einem anderen Verzeichnis.
  Abhilfe: Geben Sie den Namen der Eingabedatei als Parameter beim Laden des Programms an. (bzgl. der Ausführung eines Programms vgl. Kapitel 3.3)

- **Unzulässige Eingabe !**
  Ursache: Die Eingabedatei enthält ein unzulässiges Zeichen.
  Abhilfe: Korrigieren Sie die Eingabedatei. (vgl. hierzu Kapitel 3.4)

- **parse error in Zeile *n* !**
  Ursache: Die Daten in der Eingabedatei sind syntaktisch falsch.
  Abhilfe: Korrigieren Sie die Eingabedatei. (vgl. hierzu Kapitel 3.4)

- **Nicht genug Hauptspeicher für MALLOC !**
  Ursache: malloc( ) liefert einen NULL-Zeiger zurück
  Abhilfe: Hauptspeicher freigeben, Programm neu ausführen.

- **CALLOC: Nicht genügend Speicher vorhanden !**
  Ursache: calloc( ) liefert einen NULL-Zeiger zurück
  Abhilfe: Hauptspeicher freigeben, Programm neu ausführen.

- **Fehler beim Allokieren von Speicher mit der "malloc ( )" C-Funktion! Funktion: "*Name*" !**
  Ursache: malloc( ) liefert einen NULL-Zeiger zurück
  Abhilfe: Hauptspeicher freigeben, Programm neu ausführen.

- **Funktion: "*Name*": Falscher Datentyp !**
  Ursache: Die FP-Funktion "*Name*" erhält ein Argument falschen Typs.
  Abhilfe: Eingabe ändern oder Programm korrigieren (vgl. Kapitel 2.2).

- **Funktion: div: Division durch Null !**
  Ursache: Die FP-Funktion '/' hat ein Argument der Form <n,0> erhalten.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Argument ist keine Sequenz !**
  Ursache: Die FP-Funktion "*Name*" erwartet eine Sequenz als Argument.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Selektor <= 0 !**
  Ursache: Als FP-Funktion "Selektor" sind nur positive ganze Zahlen zugelassen.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Selektor größer als Sequenzlänge !**
  Ursache: Der Selektor "*Name*" ist zu groß für das übergebene Argument.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Argument ist keine Sequenz und auch nicht Empty !**
  Ursache: Die FP-Funktion "*Name*" erwartet nach Definition eine Sequenz oder < >.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Datenstruktur, in die eingefügt werden soll, ist keine Sequenz!**
  Ursache: Die FP-Funktion "*Name*" erwartet innerhalb der Argumentsequenz wieder eine Sequenz, z.B. apndl:<a,b>.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Argument ist kein Prädikat !**
  Ursache: Die FP-Funktion "*Name*" erwartet ein Prädikat als Argument, z.B. not:2.
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Funktion: "*Name*": Argument ist Empty und angewandte Funktion hat kein neutrales Element !**
  Ursache: Die Funktionalformen Left Insert bzw. Right Insert haben die leere Sequenz als Argument erhalten und die angewandte Funktion ist nicht '+', '-', '*' oder '/'
  Abhilfe: Eingabe ändern oder Programm korrigieren.

- **Ausgabepuffer voll !**
  Ursache: Während der Ausführung einer Transformation werden zu viele Nachrichten erzeugt.
  Abhilfe: Vergrößern Sie die Zahl BUFSIZE, die in der Datei "*daten.h*" definiert wird, und übersetzen sie den C-Code des Programms einschließlich der Dateien "input.l", "*inout.y*", "*funct.c*" und "*transf.c*" neu. (vgl. Kapitel 3.2 und Anhang A, zweiter Teil)

- **"*Name*": Prozess, der keine Spalte enthält !**
  Ursache: Ein Aufruf der Transformation "*Name*" ist nicht zulässig, wenn es Spalten gibt, die nur aus einem Atom bestehen.
  Abhilfe: Korrigieren Sie Ihr Programm, so daß diese Transformation nur auf Prozessen aufgerufen wird, die Sequenzen als Daten enthalten.

- **"*Name*": Objekt besitzt keinen Tag !**
  Ursache: Bei Anwendung der Transformationen DUP bzw. PERM gibt es Spaltenkomponenten, die keine zweielementige Sequenz darstellen.
  Abhilfe: Ändern Sie Ihr Programm entsprechend.

- "*Name*"**: Tag ist keine positive ganze Zahl !**
  Ursache: Bei Anwendung der Transformationen DUP bzw. PERM ist das zweite Element innerhalb der zweielementigen Sequenz einer Spaltenkomponente keine positive ganze Zahl.
  Abhilfe: Ändern Sie Ihr Programm entsprechend.

- **PERM: Tag ist zu groß!**
  Ursache: Bei Anwendung der Transformation PERM gibt das zweite Element innerhalb der zweielementigen Sequenz einer Spaltenkomponente keine gültige Prozessnummer an.
  Abhilfe: Ändern Sie Ihr Programm entsprechend.

- **SPLIT: kein Prädikat vorhanden !**
  Ursache: Die erste Funktion bei Split&Glue hat keinen der Werte True oder False geliefert.
  Abhilfe: Ändern Sie die erste Funktion bei Split&Glue zu einer Prädikatsfunktion ab.

- **GLUE: beide Gruppen sind leer!**
  Ursache: Beide Gruppen bei der Ausführung von Split&Glue enthalten keine Prozesse mehr. Weitere Berechnungen sind nicht möglich.
  Abhilfe: Ihr Programm sollte nicht alle Prozesse löschen.

- **CUT: Spalte besteht nicht aus zwei Elementen !**
  Ursache: Die Transformation Cut&Paste kann nur angewandt werden, wenn alle Spalten zweielementige Sequenzen sind.
  Abhilfe: Ändern Sie Ihr Programm entsprechend.

- **PASTE: Gruppen haben nicht mehr die gleiche Anzahl von Prozessen !**
  Ursache: Die Vereinigung zweier Gruppen am Ende von Cut&Paste ist nur möglich, wenn beide Gruppen die gleiche Anzahl von Prozessen enthalten.
  Abhilfe: Ändern Sie Ihr Programm entsprechend.

- **WARNUNG: Die Anwendung der Funktion f bei RED(f) hat kein Atom als Ergebnis geliefert.**
  Ursache: Im Allgemeinen wird erwartet, daß die Funktion f keine Sequenz als Ergebnis zurückgibt.
  Abhilfe: Diese Meldung führt nicht zu einem Programmabbruch. Sie kann daher ignoriert werden, falls der Programmierer die Funktion f bewußt für einen anderen Ergebnistyp definiert hat.

Sonstige hier nicht aufgeführte Fehlermeldungen lassen auf einen Compilerfehler bzw. einen Implementierungsfehler bei den Funktionen oder Transformationen schließen.

TEIL D

2DT-FP-Beispielprogramme

Die Grundideen für die im Folgenden aufgeführten parallelen Beispielprogramme stammen aus [BRSW93b] und [BRSW93c]. Die dort dargestellten Programmgerüste wurden von uns zu lauffähigen Programmversionen weiterentwickelt und um Funktionen für die Eingabe der Programmdaten erweitert. In einigen Fällen waren auch Änderungen der Syntax nötig.

MAX.2dtfp

```
Bestimme das Maximum einer Reihe von Zahlen durch Divide and Conquer

Eingabe:
x1 x2 x3 .. xn  , d.h. jede der n Spalten enthält ein Atom xi


def vergleiche  = eq o [length,!1] -> id ; gt -> [1] ; [2]
// vergleicht zwei Zahlen und liefert die größere

def oneprocess  = eq o [%%,!1]
// testet, ob die Prozessanzahl gleich 1

def iterate     = oneprocess -s&g->{ 1 ||
                          iterate ; vergleiche ; RESIZE(-2) }

def max         = iterate ; [id]
```

QUICKSORT.2dtfp

```
Eingabe:
x1 x2 x3 .. xn   , d.h. jede der n Spalten enthält ein Atom xi


def first        = eq o [#,!0] -> [id,id] ; [id,!<>]

def getleader    = first -c&p->{ id == BROAD }

def quick        = quicksort ; 1

def divide1      = eq -s&g->{ 1 || quick }

def divide2      = lt -s&g->{quick || divide1 }

def isone        = eq o [%%,!1]

def quicksort    = isone -s&g->{ id  ||  divide2 ; getleader }
// quicksort hinterläßt die Ergebnisse verteilt auf die Spalten

def pack         = RESIZE(-10000) ; [id]
// pack sammelt die Zahlen in der ersten Spalte (bis zu 10000 Zahlen)

def Quicksort    = pack ; quicksort
```

QUICKSORT2.2dtfp

```
Quicksort   (zweite Version mit anderem Eingabeformat)

Eingabe:

alle Zahlen in einer Spalte, d.h. <x1, x2, x3, .., xn>


def divide      = gt o [length,!2] -s&g->{ divide; RESIZE(2) ||
                       eq o [length,!2] -s&g->{ RESIZE(2) || id }
                       }
// divide verteilt die Zahlen auf jeweils eigene Spalten, d.h. aus
// der n-elementigen Spalte 0 werden n einelementige Spalten.
// Die "einfachere" Definition von divide als
// def divide = gt o [length,!1] -s&g->{ divide; RESIZE(2) || id }
// funktioniert nicht, da resize(2) bei Anwendung auf eine drei- und
// eine zweielementige Spalte zwei zweielementige, eine einelementige
// und eine leere Spalte erzeugt.

def first       = eq o [#,!0] -> [id,id] ; [id,!<>]

def getleader   = first -c&p->{ id == BROAD }

def test1       = eq o [1 o 1,1 o 2]

def quick       = quicksort ; 1

def divide1     = test1 -s&g->{ 1 || quick }

def test2       = lt o [1 o 1,1 o 2]

def divide2     = test2 -s&g->{quick || divide1 }

def isone       = eq o [%%,!1]

def quicksort   = isone -s&g->{ id  ||  divide2 ; getleader }
// die Ergebnisse stehen danach einzeln in den Spalten

def pack        = RESIZE(-10000)
// pack sammelt die Zahlen wieder in der ersten Spalte (bis zu 10000
// Zahlen)

def Quicksort   = pack ; quicksort ; divide
```

MATMULT.2dtfp

```
Matrizenmultiplikation

Eingabe:

1.Spalte:  <<a11,...,a1n>,<b11,...,b1n>>
2.Spalte:  <<a21,...,a2n>,<b21,...,b2n>>
usw.


def transposeA     = id-c&p->{ C2R == id}
// transponiert Matrix A

def localmult      = @((@*)o distr)o distl

def localadd = @(/R+) o trans

def matmult2 = C2R ; localadd ; R2C ; localmult ; transposeA

// hier ist die Matrix am Programmende sofort in der erwarteten Form
// auf die Prozesse verteilt
```

MATMULT2.2dtfp

```
Matrizenmultiplikation (Zweite Version mit anderem Algorithmus)

Eingabe:

1.Spalte:  <<a11,...,a1n>,<b11,...,b1n>>
2.Spalte:  <<a21,...,a2n>,<b21,...,b2n>>
usw.


def transposeA     = id-c&p->{C2R == id}
// transponiert Matrix A

def localmult      = flat o @((@*) o distr) o distl

def flat           = /R (/L apndr  o apndl)

def distribute     = shorten ; C2R ; BROAD

def shorten  = gt o [length,%%]-> shorten o tlr ; id

def matmult  = distribute ; RED(+) ; localmult ; transposeA

// vor der Anwendung von distribute steht die Ergebnismatrix in
// Spalte 0 in column major order. Um sie nun wieder in die
// ursprüngliche Anordnung zu bringen, d.h. eine Matrixspalte pro
// Prozess, wird distribute verwendet.
```

JACOBI.2dtfp

```
Jacobi-Iteration, berechnet die Lösung eines linearen
Gleichungssystems  A * x = b  (A diagonaldominant).

Folgendes Eingabeformat wird erwartet:

1.Spalte:  < <a11,...,a1n>, b1 >
2.Spalte:  < <a21,...,a2n>, b2 >
usw.


def start    = [2, 1, !0]
// start erzeugt den Startvektor x = <0,..,0>

def duplicate= [1, 2, ntimes o 3 ]

def ntimes   = itimes o [%%, [id]]

def itimes   = 2 o
               (while gt o [1, length o 2]  [1, apndl o [1 o 2, 2]])

def Global1  = [[1,2],3] -c&p-> {id == C2R }

def ip       = gt o [length o 1, !0] ->
                + o [* o [1 o 1, 1 o 2], ip o [tl o 1, tl o 2]] ; !0
// berechnet das Skalarprodukt zweier Vektoren

def neu  = / o [- o [ 1 o 1
                    , - o [ ip o [2 o 1,2]
                          , * o [ ith o [+o[#,!1] , 2 o 1]
                                , ith o [+o[#,!1] ,2]
                                ]   ]   ]
               , ith o [+o[#,!1] ,2 o 1]   ]
// neu berechnet die neuen Komponenten des Iterationsvektors nach
// der Formel xi' = 1/aii * (bi - ( Summe(k=1..n)(aik * xk) - aii * xi ))

def berechne   = [1 o 1, 2 o 1, 2, neu ]

def difference = [id, abs o - o [4, ith o [+o[#,!1] , 3]]]

def abs        = lt o [id, !0] -> (bu - 0) o id ; id

def ith        = gt o[1, !1] -> ith o [- o [1, !1], tl o 2] ; 1 o 2
// liefert das i-te Element einer Sequenz seq, wenn ihr folgende
// Sequenz uebergeben wird: <i,<seq>>

def delete     = [[1,2,4] o 1, [2]]

def Global2    = id -c&p-> {id == BROAD ; RED(max)}

def max        = gt -> 1 ; 2

def local      = delete o difference o berechne

def Iterate    = Global2 ; local ; Global1 ; duplicate

def error      = gt o [1 o 2, !0.0000001]

def result     = 3 o 1

def jacobi     = error -s&g-> {jacobi ; 1 || result} ; Iterate

def Jacobi     = jacobi ; start
```

LIBRARY.2dtfp

Dieses Beispiel enthält einige nützliche 2DT-FP-Funktionen, die in anderen Programmen verwendet werden können.

```
// Erzeugen einer zusätzlichen Spalte, die vor die Spalte 0
// eingefügt wird und die Nummer 0 erhält (alle anderen
// Spaltennummern verschieben sich um 1)

def createcolumn = eq o[#,!0] -s&g->{ 1; RESIZE(2) ; [@ (!0), id]
                                    || id}



// Spaltenanzahl verdoppeln, jede Spalte wird dupliziert und das
// Duplikat direkt hinter die Spalte eingefügt

def duplicatecolumns = 1 ; RESIZE(2) ; [id,id]



// eine bestimmte Spalte entfernen (Numerierung der anderen Spalten
// ändert sich) statt 1001 muss die Nummer der zu entfernenden
// Spalte und statt 1000 die Nummer der Spalte davor angegeben
// werden

def deletecolumn  = or o [eq o [#,!1001], eq o [#,!1000]] -s&g->{
                    RESIZE(-2) ; (eq o [#,!1] -> !<> ; id)  ||  id}


// Beispiel: Entfernen der Spalte 2 :

def deletecolumn2 = or o [eq o [#,!2], eq o [#,!1]] -s&g->{
                    RESIZE(-2) ; (eq o [#,!1] -> !<> ; id)  ||  id}
```

TEIL E

Übersetzung des Beispielprogramms "max.2dtfp" aus Anhang D

```
/* max.c: Zieldatei, die von 2dtfp aus der Quelldatei max.2dtfp
generiert wurde */

#include "daten.h"

extern FILE *yyin;

fpdata vergleiche_dtfp ();
fpdata lenistwo_dtfp ();
void divide_dtfp ();
void max_dtfp ();

/*************************************************/

fpdata vergleiche_dtfp (data)
fpdata data;
{
  fpdata d9, d8, d7,
   d6, d5, d4, d3, d2, d1, d0, res;

  d0 = duplicate (data);
  d1 = d3 = newseq (2);
  d4 = duplicate (d0);
  d3->inhalt.next_seq = length (d4);
  d3 = d3->inhalt.next_seq;
  d4 = newseq (0);
  d3->next_obj = uncompress ("••1");
  d3->next_obj->next_obj = NULL;
  destroy(d0);
  d2 = eq (d1);
  res = d2;
  if (res->typ == TRUE)
  {
    free (res);
    res = id (data);
  }
  else if (res->typ == FALSE)
  {
    free (res);
    d5 = duplicate (data);
    res = gt (d5);
    if (res->typ == TRUE)
    {
      free (res);
      res = d6 = newseq (1);
      d7 = duplicate (data);
      d6->inhalt.next_seq = selector (d7, 1);
      d6->inhalt.next_seq->next_obj = NULL;
      destroy(data);
    }
    else if (res->typ == FALSE)
    {
```

```
      free (res);
      res = d8 = newseq (1);
      d9 = duplicate (data);
      d8->inhalt.next_seq = selector (d9, 2);
      d8->inhalt.next_seq->next_obj = NULL;
      destroy(data);
    }
    else
      errorcode("conditional", 10);
  }
  else
    errorcode("conditional", 10);
  return (res);
}

fpdata lenistwo_dtfp (data)
fpdata data;
{
  fpdata d3, d2, d1, d0, res;

  d0 = d2 = newseq (2);
  d3 = duplicate (data);
  d2->inhalt.next_seq = length (d3);
  d2 = d2->inhalt.next_seq;
  d3 = newseq (0);
  d2->next_obj = uncompress ("••2");
  d2->next_obj->next_obj = NULL;
  destroy(data);
  d1 = gt (d0);
  res = d1;
  return (res);
}

void divide_dtfp ()
{
  fpdata d6, d5, d4, d3, d2, d1, d0;

  sptr = groups->col;
  while (sptr != NULL)
  {
    d0 = sptr->data;
    d1 = duplicate (sptr->data);
    sptr->data = lenistwo_dtfp (d1);
    sptr->data->inhalt.next_seq = d0;
    sptr = sptr->next_col;
  }
  split ();
  if (groups->vp != 0)
  {
    resize (2);
    divide_dtfp ();
  }
  change_groups ();
  glue ();
  sptr = groups->col;
  while (sptr != NULL)
  {
    d2 = sptr->data;
```

```
    sptr->data = vergleiche_dtfp (d2);
    sptr = sptr->next_col;
  }
  resize (-2);
}

void max_dtfp ()
{
  fpdata d7,
   d6, d5, d4, d3, d2, d1, d0;

  sptr = groups->col;
  while (sptr != NULL)
  {
    d0 = sptr->data;
    d1 = duplicate (sptr->data);
    sptr->data = lenistwo_dtfp (d1);
    sptr->data->inhalt.next_seq = d0;
    sptr = sptr->next_col;
  }
  split ();
  if (groups->vp != 0)
  {
    resize (2);
    divide_dtfp ();
  }
  change_groups ();
  glue ();
  sptr = groups->col;
  while (sptr != NULL)
  {
    d2 = sptr->data;
    sptr->data = vergleiche_dtfp (d2);
    sptr = sptr->next_col;
  }
  sptr = groups->col;
  while (sptr != NULL)
  {
    d3 = sptr->data;
    sptr->data = selector (d3, 1);
    sptr = sptr->next_col;
  }
}

/*********************************************************/

main (argc, argv)
int argc;
char *argv[];
{

/****** INITIALISIERUNG ********/

  iam = mynode();
  n = numnodes();

  initialisiere(); /* Initialisierung von Ein- und Ausgabepuffern */
```

```
/****** EINGABE ****************/

  yyin = fopen (argv[1], "r");
  if (yyin == NULL)
    {
    fprintf (stderr, "Kann Eingabedatei %s nicht oeffnen\n", argv[1]);
    return (-1);
    }
  eingabe();

/****** PROGRAMM ***************/

  max_dtfp ();

/****** AUSGABE ****************/

  ausgabe();
}
```

TEIL F

Kontextfreie Grammatik der Sprache 2DT-FP

Start ⟶ Program

Program ⟶ FktDef | Program FktDef

FktDef ⟶ **Def Name** '=' Funktion
| **error**

Funktion ⟶ Transf
| FpFkt
| Transf '**;**' Fkt
| FpFkt '**;**' Fkt

Fkt ⟶ Transf
| FpFkt
| Transf '**;**' Fkt
| FpFkt '**;**' Fkt

Transf ⟶ **CtoR** | **RtoC** | **Broad** | **Perm** | **Dup** | **Smooth**
| **Pack** | **Resize** '**(**' **Sel** '**)**' | **Red** '**(**' FpFkt '**)**'
| Arraycreating

Arraycreating ⟶ FpFkt **Cut** Fkt **CSep** Fkt **End** | FpFkt **Split** Fkt **SSep** Fkt **End**

FpFkt ⟶ Comp **Condition** FpFkt '**;**' FpFkt
| **Bu** FpFkt Object | **Bur** FpFkt Object
| **While** FpFkt FpFkt
| Comp

Comp ⟶ Expr | Expr **Compose** Comp

Expr ⟶ '**(**' FpFkt '**)**'
| **Alpha** Expr
| '**[**' '**]**'
| '**[**' FpFktList '**]**'
| '**!**' Object
| **Insert** Expr | **Rinsert** Expr
| **Sel** | **Rsel** | **Tail** | **Tailr** | **Length** | **Iota**
| '+' | '-' | '*****' | '/' | **Mod**
| **Eq** | **Lt** | **Gt** | **Ge** | **Le** | **Neq**
| **And** | **Or** | **Not**
| **Id** | **Atom** | **Null** | **Trans** | **Reverse**
| **Distl** | **Distr** | **Apendl** | **Apendr** | **Rotl** | **Rotr**
| **Colid** | **Colnr**
| **Name**

FpFktList ⟶ FpFkt | FpFktList '**,**' FpFkt

Object ⟶ **TrueConst** | **FalseConst** | **Sel** | **Float**
| **CharConst** | **String**
| '<' '>'
| '<' ObjList '>'

ObjList ⟶ Object | ObjList ',' Object

TEIL G

Literaturverzeichnis

[Bac78] J. Backus. Can programming be liberated from the Neumann style? A functional style and its algebra of programs. Communication of the ACM, 21(8):613-641, 1978.

[Brä93] T. Bräunl. Parallele Programmierung - Eine Einführung. Vieweg, 1993.

[BRSW93a] Y. Ben-Asher, G. Rünger, A. Schuster, R. Wilhelm. 2DT-FP: An FP based programming language for efficient parallel programming of multiprocessor networks. PARLE 93.

[BRSW93b] Y. Ben-Asher, G. Rünger, A. Schuster, R. Wilhelm. 2DT-FP: An FP based programming language for efficient parallel programming of multiprocessor networks. Technical Report 07/93, Universität des Saarlandes, SFB 124, 1993.

[BRSW93c] Y. Ben-Asher, G. Rünger, A. Schuster, R. Wilhelm. Implementing 2DT-FP on a multiprozessor. Universität des Saarlandes, 1993.

[BT89] D.P. Bertsekas and J.N. Tsitsiklis. Parallel and Distributed Computing. Prentice-Hall, 1989.

[Fra89] P. Fraigniaud. Performance Analysis of Broadcasting in Hypercubes. Hypercube and Distributed Computers, 311-327, INRIA, North-Holland, 1989.

[Die93] A. Dierstein. Parallelisierung mit automatischer Datenaufteilung für imperative Programmiersprachen - Teil 1. Diplomarbeit, Universität des Saarlandes, 1993.

[DS91] C. Donnelly and R. Stallman. Bison Documentation. Free Software Foundation, Cambridge, USA, 1991.

[HJ89] C-T. Ho and S.L. Johnsson. Optimum Broadcasting and Personalized Communication in Hypercubes. IEEE Transactions on Computers, 38(9):1249-1268, 1989.

[KM90] B.W. Kernighan and D.M. Ritchie. Programmieren in C. Hanser, Prentice-Hall International, 1990.

[Pax90] V. Paxson. Flex Documentation. 1990.

[WM92] R. Wilhelm, D. Maurer. Übersetzerbau: Theorie, Konstruktion, Generierung. Springer, 1992.