LooPo Target-Output

Peter Faber
Schubertstr. 1
94032 Passau
Tel.: 0851/36975
e-mail: faber@fmi.uni-passau.de

8. Mai 1995

Zusammenfassung

Dieses Papier beschreibt die Bedienung und die Implementation des **LooPo**-Target-Output-Programmes. Dieses Programm wird im **LooPo**-Projekt zur textuellen Ausgabe eines zu transformierenden oder transformierten Programmes verwendet, das in Form eines Parsebaums vorliegt, wie er in den diversen Programmen des **LooPo**-Projektes Verwendung findet.

Inhaltsverzeichnis

1 Einführung

1.1 Anforderungen

Das Target-Output-Programm sollte in erster Linie zur Ausgabe des von **LooPo** generierten, zur Eingabe äquivalenten Programmstückes eingesetzt werden, wobei

– ähnnlich wie bei der Eingabe (vgl. [Schul94]) – mehrere Ausgabesprachen unterstützt werden sollten. Insbesondere sollte auch eine Ausgabe in *Parix-C* möglich sein, also als *C*-Programmstück, das unter dem Betriebssystem *Parix* parallel verarbeitet werden kann.
Sollte das Programm für ein System mit *distributed memory*-Architektur (wie z.B. Transputer) ausgegeben werden, so sollten die nötigen Kommunikationsstatements – für die im Parsebaum keine Einträge existieren – ebenfalls ausgegeben werden.
Weiterhin sollte eine Unterscheidung zwischen `DOACROSS`- und `DOALL`-Schleifen möglich sein, also zwischen Parallelschleifen, die Abhängigkeiten tragen, und solchen, die keine Abhängigkeiten tragen.

1.2 Resultat

Das Target-Output-Programm `targout` wird einerseits dazu verwendet, den Quelltext des von **LooPo** zu transformierenden Programmstücks (als pretty-printer) anzuzeigen, andererseits aber auch – dies ist seine Hauptaufgabe –, um das von **LooPo** erstellte, zur Eingabe äquivalente Programmstück, wie es von `targgen` erzeugt wird (siehe dazu [We95]), in diversen Ausgabesprachen auszugeben.
Diese Ausgabesprachen sind nicht fest vorgegeben; als Voreinstellungen werden jedoch nur *C*, *FORTRAN* und die „theoretischen Notationen" *„Systoliker1"* und *„Systoliker2"*, wie in [Schul94] beschrieben, automatisch erkannt (die Information, in welcher dieser vier Notationen das Programmstück angegeben war, ist im Parsebaum codiert). So wie sich auch der Parsebaum aus gewissen Konstruktionen (`for`-Schleife, Zuweisung,...) zusammensetzt, so wird auch erwartet, ebendiese Konstruktionen in der jeweiligen Ausgabesprache vorzufinden; wie jedoch die einzelnen Konstruktionen in einer gegebenen Ausgabesprache genau aussehen, wird für jede Sprache in einem eigenen *Sprachbeschreibungsfile* (oder *language description file*) angegeben.
Durch Kommandozeilenoptionen können verschiedene weitere Einstellungen getroffen werden, die entscheiden, ob zwischen `DOACROSS`- und `DOALL`-Schleifen unterschieden werden soll, ob Kommunikationsstatements ausgegeben werden sollen, und noch einiges mehr.

2 Die Anwendungsseite

In diesem Abschnitt sollen – ohne die Details der Implementation zu betrachten – die möglichen Arten des Aufrufs von `targout`, die zur Verfügung stehenden Optionen und Parameter, also der allgemeine Umgang mit dem Programm selbst, beleuchtet werden.

2.1 Aufruf

Grundsätzlich läßt sich `targout` immer mit den in Abschnitt 2.1.2 beschriebenen Optionen in beliebiger Reihenfolge aufrufen. Dabei müssen nur die Dateien angegeben werden, die durch die Angabe bzw. das Nichtangeben verschiedener Optionen (wie `-comm`, `-par`,...) für die Erstellung eines Ausgabetextes unbedingt nötig sind. Im Prinzip hieße das, daß bei Verwendung geeigneter Optionen nur der Pfadname des Files anzugeben wäre, das den von `scanpars` generierten Parsebaum enthält:

Beispiel 2.1 *Ist das aktuelle Verzeichnis das zu verwendende temporäre Verzeichnis, würde es eigentlich genügen,* `targout` *folgendermaßen aufzurufen, um den von* `scanpars` *erzeugten Parsebaum auf die Standardausgabe auszugeben:*
`targout -parse example.src.out -pp`

Und um den von **LooPo** *erstellten Parsebaum im gleichen Fall auf die Standardausgabe zu schreiben (wobei keine Kommunikationsstatements erzeugt und keine Unterscheidung zwischen* `DOALL` *und* `DOACROSS` *gemacht wird), würde folgender Aufruf genügen:*
`targout -comm off -par -parse example.src.out.parse`

Da es aber vom Setzen einiger Optionen abhängen kann, welche Daten benötigt werden (und damit welche Dateinamen übergeben werden müssen), ist es sinnvoll, grundsätzlich ein bestimmtes Skelett zum Aufruf zu verwenden, das dafür sorgt, daß alle eventuell benötigten Werte schon sinnvoll belegt werden.

2.1.1 Ein Skelett zum Aufruf

Wie bei den meisten anderen Programmen des **LooPo**-Projektes auch, sollte `targout` beim Aufruf das derzeitige temporäre Verzeichnis und der „Projektname" mitgeteilt werden. Das temporäre Verzeichnis wird zur Kommunikation mit dem **LooPo**-Hauptfenster mittels einer *named pipe* benötigt. Unter Umständen ist es außerdem nötig, Schleifengrenzen mittels PIP zu berechnen, wobei temporäre Dateien entstehen (siehe Abschnitt 2.3 bzw. 3.3.5), die im temporären Verzeichnis abgelegt werden müssen. Das temporäre Verzeichnis wird mittels der Option `-temp` angegeben. Der „Projektname" ist der Pfadname des vom Scanner/Parser `scanpars` ausgegebenen Parsebaums im temporären Verzeichnis und wird mit der Option `-proj` angegeben.

Ausgabe des Quellprogramms Wird `targout` als pretty-printer eingesetzt, müssen einige Dinge anders behandelt werden als bei der Ausgabe eines von **LooPo** erzeugten Programmstücks:

- da es keine Parallelschleifen gibt, muß auch nicht zwischen `DOALL`- und `DOACROSS`-Schleifen unterschieden werden
- Abfragen, ob genügend Prozessoren vorhanden sind, sowie Kommunikationsstatements oder Statements zur Synchronisation der parallel arbeitenden Prozessoren sind im sequentiellen Programm ebenfalls hinfällig
- da es sich nach außen hin nicht um einen Durchlauf des **Target**-Outputs handelt, wäre eine Meldung ans Hauptfenster, `targout` sei erfolgreich beendet, die dann im *Log*-Teil des Hauptfensters erschiene, ebenfalls unerwünscht
- die vom Benutzer im Eingabefile über das Schlüsselwort `dep` definierten Abhängigkeiten müssen mit ausgegeben werden

Beispiel 2.2 *Ein „minimaler Aufruf" (mit möglichst wenigen Argumenten) als pretty-printer könnte folgendermaßen aussehen:*
`targout -temp /tmp/loopo123 -proj /tmp/loopo123/example.src.out -pp`
Damit wird ein Programmstück ausgegeben, das dem von `scanpars` *aus* `example.src` *generierten Parsebaum entspricht, wobei die Sprache zur Ausgabe verwendet wird, die für die Eingabe der meisten Zuweisungen verwendet wurde.*

Ausgabe des Zielprogramms Soll hingegen das von **LooPo** erzeugte Programmstück ausgegeben werden, so sind unter Umständen auch weitere Informationen zu betrachten. Dies ist der Fall, wenn Kommunikationsstatements geschrieben werden sollen oder eine Unterscheidung zwischen `DOACROSS`- und `DOALL`-Schleifen stattfinden soll. Denn für Kommunikationsstatements muß u.a. ermittelt werden, welcher Prozessor an welchen senden bzw. welcher von welchem empfangen muß. Und da

eine Parallelschleife genau dann als `DOACROSS`-Schleife markiert werden muß, wenn es eine Abhängigkeit gibt, die von dieser Schleife getragen wird, müssen auch hierfür Daten der Abhängigkeitsanalyse berücksichtigt werden.
Die dabei zusätzlich zu betrachtenden Files (deren Namen sich normalerweise aus dem oben beschriebenen Projektnamen durch Anhängen einer bestimmten Endung ergibt) sind im einzelnen:

- Das File, das die Indexräume der verschiedenen Statements enthält (normale Endung: `.ispc`), ausgegeben von der in [Ke95] beschriebenen Abhängigkeitsanalyse
- Das File, das die h-Transformationen, Quell- und Zielräume der bestehenden Abhängigkeiten speichert (normale Endung `.dep`), ausgegeben von der in [Ke95] beschriebenen Abhängigkeitsanalyse
- Das File, das die Variablentransformationen enthält, also die Abbildung vom ursprünglichen Indexraum des untransformierten Programmstücks in den neuen Indexraum und ihre inverse Abbildung (normale Endung: `.trafo`), ausgegeben von dem in [We95] beschriebenen Target-Generator

Die Namen dieser weiteren Files müssen jedoch nicht direkt angegeben werden, wenn die von **LooPo** normalerweise dafür vorgesehenen Dateien verwendet werden sollen. Dann können die Pfadnamen aller evtl. benötigten Dateien durch die Option `-proj` (wie oben) automatisch gesetzt werden.
Wenn nicht anders angegeben, werden nach jeder Parallelschleife Synchronisationsstatements ausgegeben; bei `DOALL`-Schleifen genügt dies im Shared-Memory-Modell, um zu garantieren, daß ein `write`-access einer Abhängigkeit vor einem `read`-access durchgeführt wird. Bei `DOACROSS`-Schleifen ist jedoch Kommunikation – zumindest in Form von Signalverarbeitung – nötig, um ein korrektes Verhalten zu garantieren. Laut [ZiCh91] ist es jedoch möglich, auf einige dieser Synchronisationsstatements nach `DOALL`-Schleifen zu verzichten. Für den Fall, daß eine solche Methode später noch implementiert wird, kann die Ausgabe von Synchronisationsstatements mit der Option `-sync` nicht nur an- und ausgeschaltet werden, sondern auch auf `norm`; diese Einstellung unterscheidet sich bisher nicht von der Einstellung `on`.

Beispiel 2.3 *Ein „minimaler Aufruf" (wie in Beispiel 2.2) zur Ausgabe des transformierten Programmes könnte folgendermaßen aussehen:*
`targout -temp /tmp/loopo123 -proj /tmp/loopo123/example.src.out`
In diesem Fall wird das von **LooPo** *erzeugte Programmstück zu dem Eingabefile* `example.src` *ausgegeben, wobei Kommunikationsstatements und Synchronisationsstatements ausgegeben werden und die zu verwendende Ausgabesprache automatisch aus den bei der Eingabe verwendeten Notationen (im Mehrheitsverfahren) ermittelt wird.*

Die bei bei jedem Aufruf von `targout` empfohlenermaßen anzugebenden Argumente sind also `-temp <TempDir>`, `-proj <Projektname>`, sowie – allerdings *nur* für den Einsatz als pretty-printer – `-pp`.

2.1.2 Optionen

Zusätzlich zu den in Abschnitt 2.1.1 vorgeschlagenen grundsätzlichen Optionen, können `targout` auch die in den Tabellen 1 und 2 aufgeführten Optionen angegeben werden, um eine genauer vom Benutzer spezifizierte Ausgabe zu erhalten. So wird i.d.R. wohl *entweder* die Ausgabe von Kommunikationsstatements *oder* von Synchronisationsstatements erwünscht sein. Dabei ist zu beachten, daß bei allen

Optionen, die Filenamen betreffen (bis auf `-f` und `-lang`)[1] der volle Pfadname angegeben werden sollte. Denn die benötigten Files stehen in der Regel im temporären Verzeichnis, die angegebenen Pfadnamen werden jeoch nicht um einen entsprechenden Pfad erweitert, sondern beziehen sich auf das aktuelle Verzeichnis.

Beispiel 2.4 *Der folgende Aufruf führt zur Ausgabe des in* `/tmp/loopo123/example.src.out.parse` *gespeicherten Programmstücks in der durch das File* `parix_c.lang` *im* **LooPo***-Library-Verzeichnis beschriebenen Sprache (*Parix-C*), wobei ein Test auf die Existenz genügend vieler Prozessoren ausgegeben, die Ausgabe von Synchronisationsstatements (die es in* Parix-C *eigentlich nicht gibt) jedoch unterdrückt und die Ausgabe in das File* `/tmp/loopo123/ex.out` *geschrieben wird:*

```
targout -temp /tmp/loopo123/ex.out \
-proj /tmp/loopo123/example.src.out \
-lang parix_c.lang -sync off -f ex.out
```

2.2 Sprachbeschreibungsfiles

Wie bereits in Abschnitt 1.2 beschrieben und in Abschnitt 2.1.2 auch schon vorgeführt, werden die Ausgabesprachen für `targout` in sogenannten Sprachbeschreibungsfiles definiert. Diese befinden sich im **LooPo**-Library-Verzeichnis und können für die Ausgabe mit `targout` durch die Angabe der Option `-lang <LangFile>` explizit ausgewählt werden, wobei `<LangFile>` sich im **LooPo**-Library-Verzeichnis befinden muß. In diesem Abschnitt soll nun auf den Aufbau dieser Sprachbeschreibungsfiles genauer eingegangen werden.
Der **LooPo**-Parsebaum unterscheidet zwischen verschiedenen Konstrukten wie `for`-Schleifen (parallel oder sequentiell), Zuweisungen usw. Für jedes dieser Konstrukte erwartet auch `targout`, in der zu verwendenden Ausgabesprache ein solches Konstrukt vorzufinden. In einem Sprachbeschreibungsfile wird jedes Konstrukt beschrieben durch eine Zeile, die mit einem entsprechenden Schlüsselwort beginnt, gefolgt von einem Doppelpunkt, (dabei muß die Zeile *tatsächlich* mit dem *Schlüsselwort* beginnen, nicht etwa mit einem Leerzeichen o.ä., und auch der Doppelpunkt muß *direkt* auf das Schlüsselwort folgen). Der Rest der Zeile, der unmittelbar hinter dem Doppelpunkt beginnt, beschreibt dann die Ausgabe, die `targout` beim Auftreten des bezeichneten Konstrukts ausführen soll. Die Restzeile darf dabei die Größe von 256 Zeichen nicht überschreiten. Jede Zeile, die nicht exakt diesem Format entspricht, wird als Kommentar interpretiert und überlesen. Die Tabellen 3, 4 und 5 zeigen eine Liste der möglichen Konstrukte und der dazugehörigen Schlüsselwörter.

Normaler Text innerhalb dieses Zeilenrestes wird genau so ausgegeben, wie er dasteht. Er kann jedoch von *escape*-Sequenzen unterbrochen sein; diese beginnen mit einem Rückstrich (Backslash – „\“), dem dann ein – aus *einem* Buchstaben bestehender – Makroname folgt, der genauer spezifiziert, was ausgegeben werden soll. In den Tabellen 3, 4 und 5 ist auch aufgeführt, welche *escape*-Sequenzen bei welchem Konstrukt angewandt werden können. Tabelle 6 zeigt alle *escape*-Sequenzen mit ihrer Bedeutung. Nach der Ausgabe des zu dieser *escape*-Sequenz passenden Textes, wird mit der Ausgabe des Zeilenrestes nach der *escape*-Sequenz fortgefahren. Wurde

[1] diese beziehen sich automatisch auf das temporäre Verzeichnis bzw. das **LooPo**-Library-Verzeichnis

Option	Wirkung	Voreinstellung
`-comm on | norm |off`	`on`: Kommunikationsstatements (`sends` und `receives`) werden auf jeden Fall ausgegeben `norm`: Kommunikationsstatements werden nur geschrieben, falls Sender und Empfänger verschiedene Prozessoren sind `off`: Kommunikationsstatements werden gar nicht ausgegeben	`norm`
`-count`	gibt vor jedem Statement dessen Nummer als Kommentar aus	keine Ausgabe der Nummer
`-dep <DepFile>`	benutzt (falls nötig) die Abhängigkeiten (Quell- und Zielräume, h-Transformationen, ...) aus dem File `<DepFile>`	falls `-proj` nicht gegeben wurde undefiniert, sonst nach **LooPo**-Konvention normaler Name für `dependence`-Output (`.dep`-File)
`-f <OutputFile>`	schreibt die Ausgabe in das temporäre Verzeichnis in ein File namens `<OutputFile>`	Standardausgabe
`-ispc <IspcFile>`	benutzt (falls nötig) die in `<IspcFile>` gespeicherten Indexräume der Statements beim Laden von Abhängigkeiten	falls `-proj` nicht gegeben wurde undefiniert, sonst nach **LooPo**-Konvention normaler Name für `dependence`-Output
`-lang <LangFile>`	benutzt zur Ausgabe die durch das im **LooPo**-Library-Verzeichnis befindliche Sprachbeschriebungsfile `<LangFile>` beschriebene Sprache (siehe Abschnitt 2.2)	`c.lang`, `fortran.lang`, `systolician1.lang` oder `systolician2.lang`, je nachdem, in welcher Notation die meisten Statements sind
`-no_proc_test`	es wird kein Test ausgegeben, ob das verwendete System über ausreichend viele Prozessoren verfügt (diese Option wird auch von `-pp` eingeschaltet)	normalerweise wird der Test mit einem Abbruchstatement für den Fall, daß nicht genügend Prozessoren vorhanden sind, ausgegeben
`-par`	unterscheidet bei der Ausgabe nicht zwischen `DOACROSS`- und `DOALL`-Schleifen; stattdessen wird für beide Arten ein `par`-Konstrukt (siehe Abschnitt 2.2) ausgegeben	für `DOALL`-Schleifen wird ein `doall`-Konstrukt ausgegeben, für `DOACROSS`-Schleifen ein `doacross`-Konstrukt
`-parse <ParseTree>`	liest den auszugebenden Parsebaum (und die dazugehörige Symboltabelle) aus dem File `<ParseTree>`	falls `-proj` nicht gegeben wurde undefiniert, sonst nach **LooPo**-Konvention normaler Name für `scanpars`- bzw. `targgen`-Output (`.out`- bzw. `.parse`-File)

Tabelle 1: mögliche Optionen für `targout`

Option	Wirkung	Voreinstellung
-pp	pretty-printer-Modus: benutzerdefinierte Abhängigkeiten werden am Ende des Programmtextes ausgegeben, *nicht* ausgegeben werden Kommunikations- oder Synchronisationsstatements, und es wird auch keine Terminationsmeldung ans Hauptfenster geschickt (vgl. Abschnitt 2.1.1); falls -proj gegeben wurde, jedoch nicht -parse, wird der von scanpars erzeugte Output ausgegeben	Ausgabe eines Testes, ob genügend Prozessoren zur Verfügung stehen, keine Ausgabe benutzerdefinierter Abhängigkeiten, Ausgabe von Kommunikations- und Synchronisationsstatements (falls nicht mit -comm/ -sync anders eingestellt); Ausgabe des von targgen erzeugten Outputs (falls nicht mit -parse anders eingestellt)
-prev	für Kommunikationszwecke evtl. verwendete Kanalnamen bzw. Zählindizes (vgl. Abschnitte 3.3.5 und 3.3.6) werden zu Beginn des Programms deklariert	für Kommunikationszwecke evtl. verwendete Kanalnamen bzw. Zählindizes werden vor der äußersten Schleife deklariert, in deren Rumpf sie benötigt werden
-proj <ProjName>	setzt die Pfadnamen aller Files nach den normalen **LooPo**-Konventionen, ausgehend von <ProjName> als dem Projektnamen (<ProjName> ist also der volle Pfadname des von scanpars generierten Ausgabefiles	–
-sync on | norm |off	on: Synchronisationsstatements (barrier) werden auf jeden Fall ausgegeben (nach jeder Parallelschleife) norm: siehe on off: Synchronisationsstatements werden gar nicht ausgegeben	norm
-temp <TempDir>	verwendet <TempDir> als temporäres Verzeichnis	aktuelles Verzeichnis
-trafo <TrafoFile>	benutzt (falls nötig) die in <TrafoFile> gespeicherten Variablentransformationen bei der Umrechnung der geladenen alten Indexräume in die neuen	falls -proj nicht gegeben wurde undefiniert, sonst nach **LooPo**-Konvention normaler Name für targgen-Output (.trafo-File)

Tabelle 2: mögliche Optionen für targout (fortgesetzt)

Schlüsselwort	Konstrukt	mögliche *escape*-Sequenzen
`assign`	Zuweisung	\t \ \n \d \i \\ \v \e
`skip`	Skip-Statement (no operation)	\t \n \d \i \\
`indent`	Zeichen, mit dem eingerückt wird	\\
`sync`	Synchronisationsstatement (`barrier`)	\t \n \d \i \\
`send`	`send`-Statement: Empfangen von Daten von einem anderen Prozessor	\t \n \d \i \\ \c \v \p \r \D
`receive`	`receive`-Statement: Empfangen von Daten von einem anderen Prozessor	\t \n \d \i \\ \c \v \p \r \D
`programbegin`	Beginn der `targout`-Programmausgabe	\t \n \d \i \\
`programend`	Ende der `targout`-Programmausgabe	\t \n \d \i \\
`exiterr`	Statement zum Abbruch des Programms im Fehlerfall	\t \n \d \i \\
`exitfine`	Statement zum regulären Beenden eines Programms (unbenutzt)	\t \n \d \i \\
`proccond`	Bedingung „zu weinige Prozessoren vorhanden" (in diesem Fall wird das Programm abgebrochen)	\t \n \d \i \\ \x
`ifbegin`	Beginn eines `if`-Statements (bis einschließlich zum `then`)	\t \n \d \i \\ \c
`ifend`	Ende eines `if`-Statements (nach dem `else`-Zweig; allerdings unterstützt `targout` momnetan noch keine vollen `if-then-else`-Konstrukte)	\t \n \d \i \\
`parbegin`	Beginn einer Parallelschleife (pauschales Konstrukt für den Fall, daß zwischen `DOALL` und `DOACROSS` nicht unterschieden werden soll)	\t \n \d \i \\ \v \l \u \x \y \z
`parend`	Ende einer Parallelschleife (pauschales Konstrukt für den Fall, daß zwischen `DOALL` und `DOACROSS` nicht unterschieden werden soll)	\t \n \d \i \\ \v \x \y \z
`doallbegin`	Beginn einer `DOALL`-Schleife	\t \n \d \i \\ \v \l \u \x \y \z
`doallend`	Ende einer `DOALL`-Schleife	\t \n \d \i \\ \v \x \y \z
`doacrossbegin`	Beginn einer `DOACROSS`-Schleife	\t \n \d \i \\ \v \l \u \x \y \z
`doacrossend`	Ende einer `DOACROSS`-Schleife	\t \n \d \i \\ \v \x \y \z
`seqforbegin`	Beginn einer sequentiellen `for`-Schleife	\t \n \d \i \\ \v \l \u \x \y \z
`seqforend`	Ende einer sequentiellen `for`-Schleife	\t \n \d \i \\ \v \x \y \z
`stpparbegin`	Beginn einer Parallelschleife mit beliebiger Schrittweite (pauschales Konstrukt für den Fall, daß zwischen `DOALL` und `DOACROSS` nicht unterschieden werden soll)	\t \n \d \i \\ \v \l \u \s \x \y \z
`stpparend`	Ende einer Parallelschleife mit beliebiger Schrittweite (pauschales Konstrukt für den Fall, daß zwischen `DOALL` und `DOACROSS` nicht unterschieden werden soll)	\t \n \d \i \\ \v \s \x \y \z
`stpallbegin`	Beginn einer `DOALL`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \l \u \s \x \y \z
`stpallend`	Ende einer `DOALL`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \s \x \y \z
`stpacrossbegin`	Beginn einer `DOACROSS`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \l \u \s \x \y \z

Tabelle 3: Standard-Konstrukte mit ihren Schlüsselwörtern und möglichen *escape*-Sequenzen

Schlüsselwort	Konstrukt	mögliche *escape*-Sequenzen
`stpacrossend`	Ende einer `DOACROSS`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \s \x \y \z
`stpforbegin`	Beginn einer sequentiellen `for`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \l \u \s \x \ \y \z
`stpforend`	Ende einer sequentiellen `for`-Schleife mit beliebiger Schrittweite	\t \n \d \i \\ \v \s \x \y \z
`chandecl`	Deklaration eines Kanals zur Kommunikation (wird in keiner der vordefinierten Sprachen verwendet)	\t \n \d \i \\ \c
`intdecl`	Deklaration von `integer`-Variablen (verwendet für Zählvariablen von künstlich eingeführten `for`-Schleifen [für Broadcasting])	\t \n \d \i \\ \v
`dependence`	Beschreibung einer benutzerdefinierten Abhängigkeit	\t \n \d \i \D \S \h \\
`rem`	Kommentar im Programmtext	\t \n \d \i \\ \r
`arraybegin`	Zeichen, das den Beginn der Index-Angaben bei einem Array markiert (z.B. in *C* das Zeichen '[')	\\
`arraysep`	Zeichen, das zwei Index-Angaben in einem mehrdimensionalen Array trennt	\\
`arrayend`	Zeichen, das das Ende der Index-Angaben bei einem Array markiert (z.B. in *C* das Zeichen ']')	\\
`vectorbegin`	Zeichen, das den Beginn eines Vektors markiert	\\
`vectorsep`	Zeichen, das zwei Koordinaten in einem mehrdimensionalen Vektor trennt	\\
`vectorend`	Zeichen, das das Ende eines Vektors markiert	\\
`ceilbegin`	Beginn einer Rundung nach oben (bis ausschließlich zu der zu rundenden Zahl)	\\
`ceilend`	Ende einer Rundung nach oben (ab ausschließlich der zu rundenden Zahl)	\\
`floorbegin`	Beginn einer Rundung nach unten (bis ausschließlich zu der zu rundenden Zahl)	\\
`floorend`	Ende einer Rundung nach unten (ab ausschließlich der zu rundenden Zahl)	\\
`maxbegin`	Beginn einer Maximum-Funktion (bis ausschließlich zum ersten Argument)	\\
`maxsep`	Trennzeichen zwischen zwei Argumenten für die Maximum-Funktion	\\
`maxend`	Ende einer Maximum-Funktion (ab ausschließlich dem letzten Argument)	\\
`minbegin`	Beginn einer Minimum-Funktion (bis ausschließlich zum ersten Argument)	\\
`minsep`	Trennzeichen zwischen zwei Argumenten für die Minimum-Funktion	\\
`minend`	Ende einer Minimum-Funktion (ab ausschließlich dem letzten Argument)	\\

Tabelle 4: Standard-Konstrukte mit ihren Schlüsselwörtern und möglichen *escape*-Sequenzen (fortgesetzt)

Schlüsselwort	Konstrukt	mögliche *escape*-Sequenzen
`plus`	Pluszeichen	\\
`minus`	Minuszeichen	\\
`times`	Multiplikationszeichen	\\
`divide`	Divisionszeichen	\\
`and`	Zeichen für logisches und	\\
`geq`	Zeichen für $\geq$	\\
`bracket_open`	öffnende Klammer (in arithmetischen Ausdrücken)	\\
`bracket_close`	schließende Klammer (in arithmetischen Ausdrücken)	\\
`misc`	Sonderkonstruktion für Sprachspezifische Einstellungen	\\

Tabelle 5: Standard-Konstrukte mit ihren Schlüsselwörtern und möglichen *escape*-Sequenzen (fortgesetzt)

eine unerlaubte *escape*-Sequenz in einem Sprachbeschreibungsfile gefunden, so wird eine Warnung ausgegeben und statt dieser *escape*-Sequenz jedesmal ein Backslash geschrieben.

Beispiel 2.5 *Hier ein Ausschnitt aus einem Sprachbeschreibungsfile, das die Sprache* C *beschreibt:*

```
assign:\t\v=\e;\n

seqforbegin:\tfor(\v=\l;\v<=\u;\v++)\i\n\t{\i\n

seqforend:\d\t}\n\d

stpforbegin:\tfor(\v=\l;\v<=\u;\v+=\s)\i\n\t{\i\n

stpforend:\d\t}\n\d
```

Betrachten wir das for*-Konstrukt genauer, können wir folgende Zuordnungen treffen:*

```
        for(\v=\l;\v<=\u;\v++)\n\t\{\i
    variable_|  |      |       | |   |_increase level
  lower bound___|      |       | |_____tabulator
  upper bound__________|       |_______newline
```

Die folgenden Sprachbeschreibungsfiles sind standardmäßig im **LooPo**-Library-Verzeichnis zu finden:

Filename	beschriebene Sprache
`c.lang`	*C* (mit Pseudoparallelschleifen)
`fortran.lang`	*FORTRAN* (mit `DOALL` und `DOACROSS`)
`systolician1.lang`	„Systoliker1“ (wie in [Schul94] beschrieben)
`systolician2.lang`	„Systoliker2“ (wie in [Schul94] beschrieben)
`parix_c.lang`	unter *Parix* lauffähiges *C*

escape-Sequenz	Bedeutung
\\	Backslash
\t	rückt die Zeile ein, indem je nach Einrücktiefe mehrmals hintereinander die durch `indent` gegebene Zeichenfolge geschrieben wird
\n	neue Zeile
\d	verringert die Einrücktiefe
\i	erhöht die Einrücktiefe
\v	Variablenname
\l	untere Schleifengrenze
\u	obere Schleifengrenze
\s	Schrittweite
\e	rechte Seite einer Zuweisung
\c	Kanalname bzw. Bedingung in einem `if`-Statement
\r	eigentlicher Kommentar im Konstrukt für Kommentare (`rem`), bei `send/receive` die Prozessornummer des Ziel/Quellprozessors (eindimensional)
\p	bei `send/receive` die Prozessornummer des Ziel/Quellprozessors (mehrdimensional, also in der Form $[p1, p2-1]\ldots$)
\x	extent (bei `proccond` Produkt aller parallelen extents, bei Schleifen nur der extent der jeweiligen Schleife)
\y	Produkt der extents aller umgebenden Parallelschleifen
\z	Produkt der Maxima der extents aller folgenden Parallelschleifen
\S	Nummer des source-Statements in einer benutzerdefinierten Abhängigkeit
\D	Nummer des destination-Statements in einer benutzerdefinierten Abhängigkeit, bei `send/receive` die Nummer der Abhängigkeit
\h	Matrix der h-Transformation in einer benutzerdefinierten Abhängigkeit

Tabelle 6: Die *escape*-Sequenzen und ihre Bedeutungen

2.3 Übersetzen von *Parix-C*-Programmen

Im **LooPo**-Library-Verzeichnis findet sich u.a. ein Sprachbeschreibungsfile `parix_c.lang`, das die Sprache *C* beschreibt, so wie sie benötigt wird, um eine `targout`-Ausgabe mit dem `ancc`-*C*-Compiler für *Parix* zu übersetzen. *Parix* ist ein *UNIX*-artiges Betriebssystem, das es ermöglicht, *dasselbe* Programm gleichzeitig auf mehreren Prozessoren ablaufen zu lassen. Dabei besitzt jeder Prozessor seinen lokalen Speicher, so daß Datenabhängigkeiten zwischen mehreren Prozessoren zur Kommunikation dieser Prozessoren untereinander führen müssen. Dieses *Single Program Multiple Data*-Konzept (SPMD) hat also zur Folge, daß es keine tatsächlichen Parallelschleifen in *Parix-C* gibt, da einfach auf *jedem* Prozessor das *gleiche* Programm ausgeführt wird. Stattdessen muß eine Parallelschleife dadurch simuliert werden, daß dem Zählindex für diese Schleife ein bestimmter Wert zugewiesen wird, der von der Nummer des bearbeitenden Prozessors abhängig ist. Da auf jedem Prozessor dasselbe Programm ausgeführt wird, ist es nur möglich, solche Programme für *Parix* zu übersetzen, bei denen keine Parallelschleife in einer sequentiellen Schleife verschachtelt ist. Um zu bestimmen, welcher Durchlauf einer Parallelschleife nun bearbeitet werden soll, bietet sich unter *Parix* die Prozessornummer an, die jeder Prozessor abfragen kann.
Für `targout` hat dies nun untenstehende Folgen:

- um eine Ausgabe in *Parix-C* einzuleiten, ist die Option `-lang parix_c.lang` anzugeben
- da es nach der Parallelisierung parallele Programmschleifen geben kann, die Abhängigkeiten tragen, müssen Kommunikationsstatements ausgegeben werden – also `-comm on` oder `-comm norm`, aber auf keinen Fall `-comm off` angeben
- da jede notwendige Synchronisation schon durch Kommunikationen durchgeführt wird, sind Synchronisationsstatements überflüssig und können mit `-sync off` ausgeschaltet werden
- da parallelisierte Programme u.U. in sequentiellen Schleifen geschachtelte Parallelschleifen enthalten, wird von `targout` jede Schleife, die sich innerhalb einer sequentiellen Schleife befindet, zu einer sequentiellen Schleife umgeformt, sofern das Wort „`parix`“ im Namen des Sprachbeschreibungsfiles auftaucht – das führt im Extremfall dazu, daß das ausgegebene Programmstück nur sequentielle Schleifen aufweist (so daß jeder Prozessor exakt die gleichen Berechnungen durchführen würde)
- die Prozessoren sind von 0 an durchnumeriert; der Wert des Zählindexes p einer „Parallelschleife“, ausgeführt auf dem Prozessor mit Nummer P läßt sich von diesem Prozessor ermitteln durch:

  $$p = (P/e_>)\%e + l,$$

  wobei $e_>$ das Produkt der extents (Differenz zwischen oberer und unterer Schranke +1) aller tiefer verschachtelten Parallelschleifen und e der extent der p bindenden Schleife ist, deren untere Schranke l ist ($/$ ist die ganzzahlige Division, $\%$ der Rest bei ganzzahliger Division)

Da es vorkommen kann, daß mehrere Prozessoren aus ihren Nummern denselben Wert für den Zählindex einer Parallelschleife errechnen, wird im *Parix-C*-Programm auch abgeprüft, ob der Prozessor die kleinste Nummer hat, bei der sich dieser Wert für den Zählindex ergibt. Nur der Prozessor mit dieser kleinsten Prozessornummer

führt tatsächlich die Parallelschleife aus. Dies wird gewährleistet durch eine Überprüfung auf $(P/e_{>})/e = 0$. Dadurch wird außerdem garantiert, daß die Schleife nur ausgeführt wird, wenn der Index zwischen oberer und unterer Schranke liegt.
Für Kommunikationen zwischen den Prozessoren gibt es im **LooPo**-Library-Verzeichnis in der Datei `loopo_comm.c` die Funktionen `SendToNode` und `ReceiveFromNode`. Diese überprüfen zunächst, ob der aufrufende Prozessor versucht, mit sich selbst zu kommunizieren; wenn dies nicht der Fall ist, verwenden sie `SendNode` und `RecvNode`, um ohne *virtuelle Links* mit dem entsprechenden Prozessor zu kommunizieren. Außerdem enthält `loopo_comm.c` die Funktionen `SendByLink` und `ReceiveByLink`, die nach dieser Überprüfung eine *virtuelle Link* aufbauen, um über diese zu kommunizieren. Die entsprechenden Header stehen im **LooPo**-Library-Verzeichnis in der Datei `loopo_comm.h`. Im File `parix_c.lang` werden jedoch nur die ersten beiden Funktionen verwendet. Um also ein Programm unter *Parix-C* zu übersetzen, sollte zunächst die Datei `loopo_comm.c` mit

`px ancc -c loopo_comm.c`

übersetzt (jedoch nicht gelinkt) werden, falls dies nicht schon geschehen ist. Bei dem ausgegebenen Code handelt es sich immer noch um ein Programm*stück* – es sind also noch einige Modifikationen (wie z.B. Initialisierung der Variablen, Einführung einer das Programmstück umschließenden Funktion usw.) durchzuführen. So muß auch (falls das Programmstück Kommunikationsstatements enthält) eine `#include`-Zeile

`#include "`*LibDir*`/loopo_comm.h"`

eingefügt werden, wobei *LibDir* das **LooPo**-Library-Verzeichnis ist. Das so vervollständigte Programmstück kann dann mit

`px ancc -o` *ProgName*`.px` *LibDir*`/loopo_comm.o` *ProgName*`.c`

übersetzt werden (*LibDir* ist wieder das **LooPo**-Library-Verzeichnis, *ProgName* ist der Programmname ohne Extension); es kann dann mit

`px run -a p`*ProcNo* `example.px`

gestartet werden (wobei *ProcNo* die Anzahl der Prozessoren ist, auf denen das Programm ausgeführt werden soll).
Findet in einem Programmstück Kommunikation statt, so wird immer abgetestet, ob der betreffende Prozessor auch im Ziel- bzw. Quellraum ist, ob also die Kommunikation tatsächlich nötig und ein Kommunikationspartner vorhanden ist. Ein Sonderfall bei der Kommunikation ist das *Broadcasting*, d.h., ein Prozessor errechnet ein Datum, das von mehreren anderen Prozessoren benötigt wird. In diesem Fall werden (u.U. mehrere ineinander verschachtelte) sequentielle `for`-Schleifen ausgegeben, die jeweils einen eigenen Index durchzählen. Dabei ist es wichtig, daß im ursprünglichen Programm **keine mit einem Underscore beginnenden Variablen** auftauchen, da dieser führende Underscore („_") als Unterscheidungsmerkmal von im Programm tatsächlich verwendeten Variablen und (von `targout`) künstlich eingefügten Variablen verwendet wird. Im Rumpf dieser sequentiellen `for`-Schleifen, deren Grenzen so berechnet sind, daß nur Kommunikationen mit Prozessoren stattfinden, die tatsächlich vorhanden und tatsächlich auf dieses Datum angewiesen sind, steht dann ein einzelnes `send`-Statement. Dieses führt dann – in Abhängigkeit vom Wert der Zählindizes – eine Kommunikation mit einem der Prozessoren aus dem Zielraum durch. Es wäre eigentlich wünschenswert, mit Hilfe von `threads` diese Kommunikationen auf einem Prozessor wenigstens quasi-parallel durchzuführen, doch hat sich dies als nicht ganz so einfach herausgestellt.

Beispiel 2.6 *Hier nun ein Beispiel für einen kompletten* **LooPo***-Durchlauf mit Ausgabe des Zielprogramms in* Parix-C. *Das folgende Programmstück wurde von* **LooPo** *in einem (frühen) Testlauf parallelisiert:*

```
DO i = 2, min(10, m)
  DO j = i,i+n+5
```

```
    a(i,j) = a(i-1,j) + a(i-2,j-2) + c
  END DO
END DO

dep
2 5
1 0 0 0 -1
0 1 0 0  0
1 1

dep
2 5
1 0 0 0 -2
0 1 0 0 -2
1 1
```

Der parallelisierte Code, ausgegeben von `targout` *in* Parix-C*, sieht dann folgendermaßen aus:*

```
/* LooPo program */
/*    PARIX-C    */

if(PC_nProcs<(min((n+15),(m+n+5),END_MIN)-2+1))
 {
  return(1);
 }
/* DOACROSS */
j=(PC_MyProcID/(1))%(min((n+15),(m+n+5),END_MIN)-2+1)+2;
if((PC_MyProcID/(1))/(min((n+15),(m+n+5),END_MIN)-2+1)==0)
 {
  for(i=max(j-n-5,2,END_MAX);i<=min(j,m,10,m,END_MIN);i++)
   {
    if((-i+j>=0)&&(i-j+n+5>=0)&&(i-4>=0)&&(-i+12>=0)&&(-i+m+2>=0))
     {
      ReceiveFromNode(&(a[i-2][j-2]),sizeof(a[i-2][j-2]),j-2,2,min(n+15,m+n+5,
END_MIN),1,END_RECEIVE);
     }
    if((-i+j+1>=0)&&(i-j+n+4>=0)&&(i-3>=0)&&(-i+11>=0)&&(-i+m+1>=0))
     {
      ReceiveFromNode(&(a[i-1][j]),sizeof(a[i-1][j]),j,2,min(n+15,m+n+5,END_MIN
),1,END_RECEIVE);
     }
    /* 1 */
    a[i][j]=a[i-1][j]+a[i-2][j-2]+c;
    if((-i+j>=0)&&(i-j+n+5>=0)&&(i>=0)&&(-i+8>=0)&&(-i+m-2>=0))
     {
      SendToNode(&(a[i][j]),sizeof(a[i][j]),j+2,2,min(n+15,m+n+5,END_MIN),1,END
_SEND);
     }
    if((-i+j-1>=0)&&(i-j+n+6>=0)&&(i-1>=0)&&(-i+9>=0)&&(-i+m-1>=0))
     {
      SendToNode(&(a[i][j]),sizeof(a[i][j]),j,2,min(n+15,m+n+5,END_MIN),1,END_S
END);
     }
   }
```

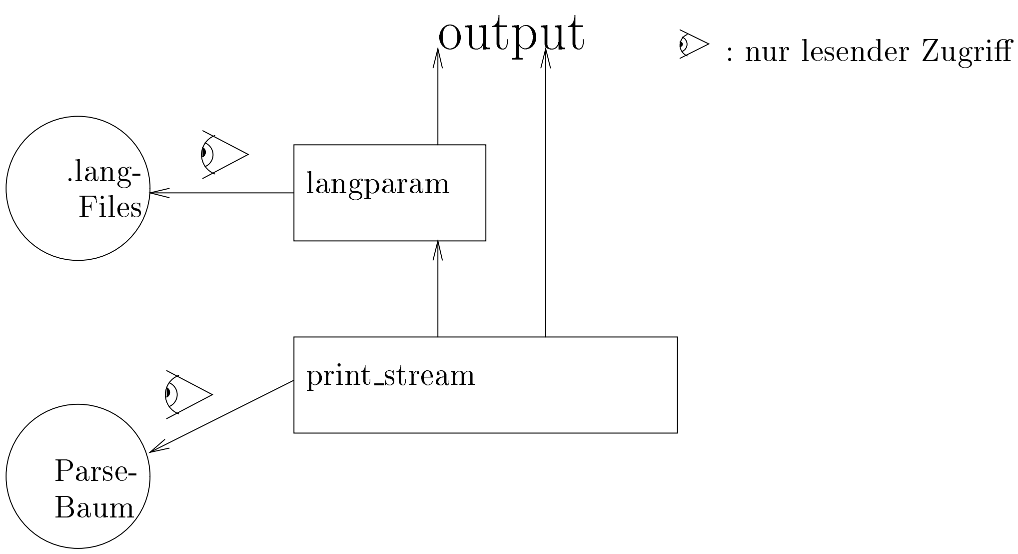

Abbildung 1: Verarbeitung von reinem Text und *escape*-Sequenzen durch `langparam` und `print_stream`

```
 }


/*    end of     */
/* LooPo program */
```

3 Die Implementationsseite

Nachdem die Anwendung von `targout` beleuchtet wurde, wollen wir nun die Implementation der beschriebenen Eigenschaften näher betrachten.

3.1 Das Konzept

Um möglichst unabhängig von der tatsächlichen Notation der Ausgabesprache zu sein, liegt es nahe, die Konstruktionen, die vom **LooPo**-Parsebaum vorgegeben werden, durch eine Zeichenkette zu beschreiben. Diese muß *escape*-Sequenzen enthalten können, um nicht feststehende Ausgaben wie Variablennamen, Schleifengrenzen u.ä. genau an der Stelle ausgeben zu können, an der sie auch in der Ausgabesprache vorgesehen sind. Hierfür ist also eine Instanz nötig, die Zugriff auf Elemente des Parsebaums hat und so die Ausgabe der entsprechenden Zeichen beim Auftauchen einer *escape*-Sequenz bewerkstelligen kann. Die hierfür verwendete Struktur `print_stream` wird in 3.3.2 beschrieben. Die Ausgabe der übrigen reinen Textsequenzen hingegen wie auch das Einladen der die Sprache beschreibenden Parametrisierungen (Zeichenketten) kann einer anderen Instanz überlassen werden, die keinerlei Zugriff auf Parsebaumelemente und -strukturen hat. Diese direkt sprachabhängigen Parameter (*language parameters*) werden von der Klasse `langparam` (Abschnitt 3.3.1) verwaltet (siehe Abbildung 1).

Die gröbere Struktur des Parsebaumes – die Frage, welches Konstrukt wann

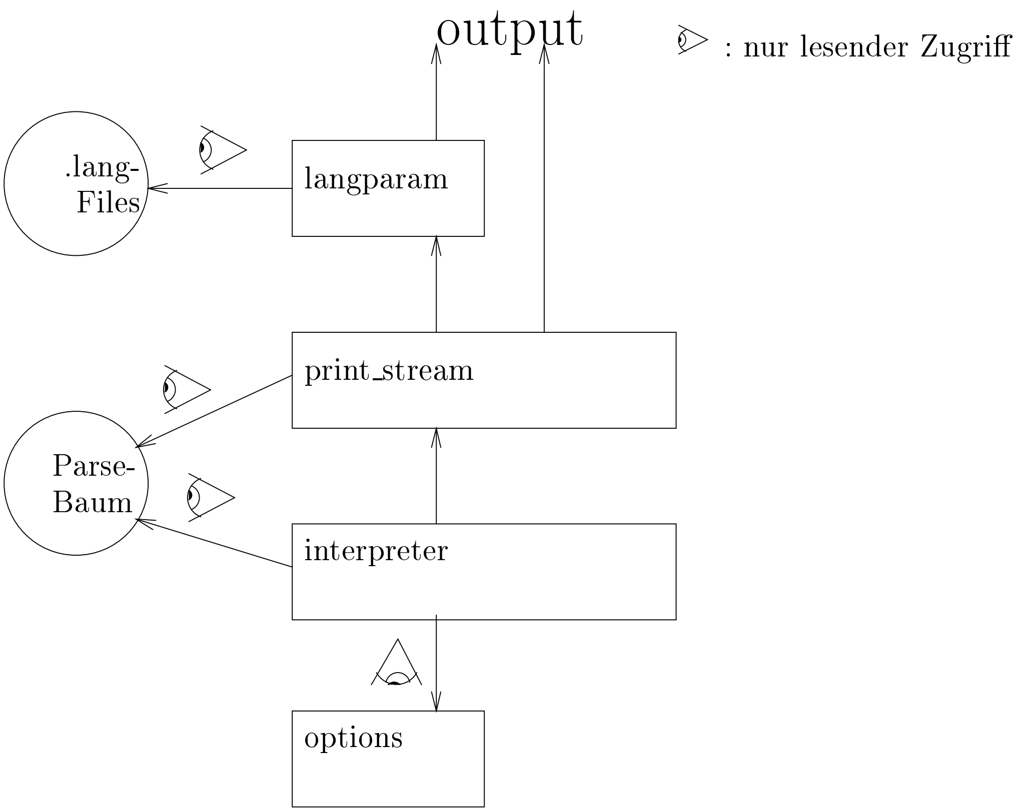

Abbildung 2: Verarbeitung der groben Parsebaum-Struktur durch `interpreter`

ausgegeben werden muß – kann von einem weiteren Modul betrachtet werden. Dabei muß wiederum nicht so sehr auf die Daten innerhalb des zu betrachtenden `LaNodes`[2] geachtet werden, sondern vielmehr auf die *Art* des `LaNodes`, um die Ausgabe des richtigen Konstrukts herbeizuführen. Dafür müssen hier aber die vom Benutzer gegebenen Optionen beachtet werden, wie z.B., ob Kommunikationsstatements auszugeben sind (die nicht im Parsebaum stehen) oder ob zwischen `DOALL`- und `DOACROSS`-Schleifen unterschieden werden soll. Mit diesen Dingen beschäftigt sich der `interpreter`, der die Struktur des Parsebaums interpretiert (Beschreibung in Abschnitt 3.3.3), wie Bild 2 zeigt (die Kommandozeilenoptionen werden von der Klasse `options` verarbeitet und die entsprechenden Werte durch „`get`"-Funktionen bereitgestellt).

Wie schon erwähnt, sollte `targout` aber auch evtl. nötige Kommunikationsstatements ausgeben können. *Wann* eine bestimmte Art von Kommunikationsstatement ausgegeben werden muß, kann der `interpreter` leicht bestimmen (vor einer Zuweisung müssen Daten empfangen, danach gesendet werden). *Ob* und *wieviele* Statements aber geschrieben werden müssen, ergibt sich nicht allein aus dem Parsebaum, denn ein Kommunikationsstatement muß genau dann geschrieben werden, wenn es eine andere Zuweisung gibt, die von der gerade auszugebenden abhängig ist (oder umgekehrt). Diese Information über existierende Abhängigkeiten kann der `interpreter` aus einer Struktur `comm_request` (*communication request*) ablesen, die quasi als reines Datenbehältnis für Abhängigkeiten bzw. die daraus resultieren-

[2] zu einer Beschreibung des Parsebaums und der ihn darstellenden `LaNodes` siehe [Schul94]

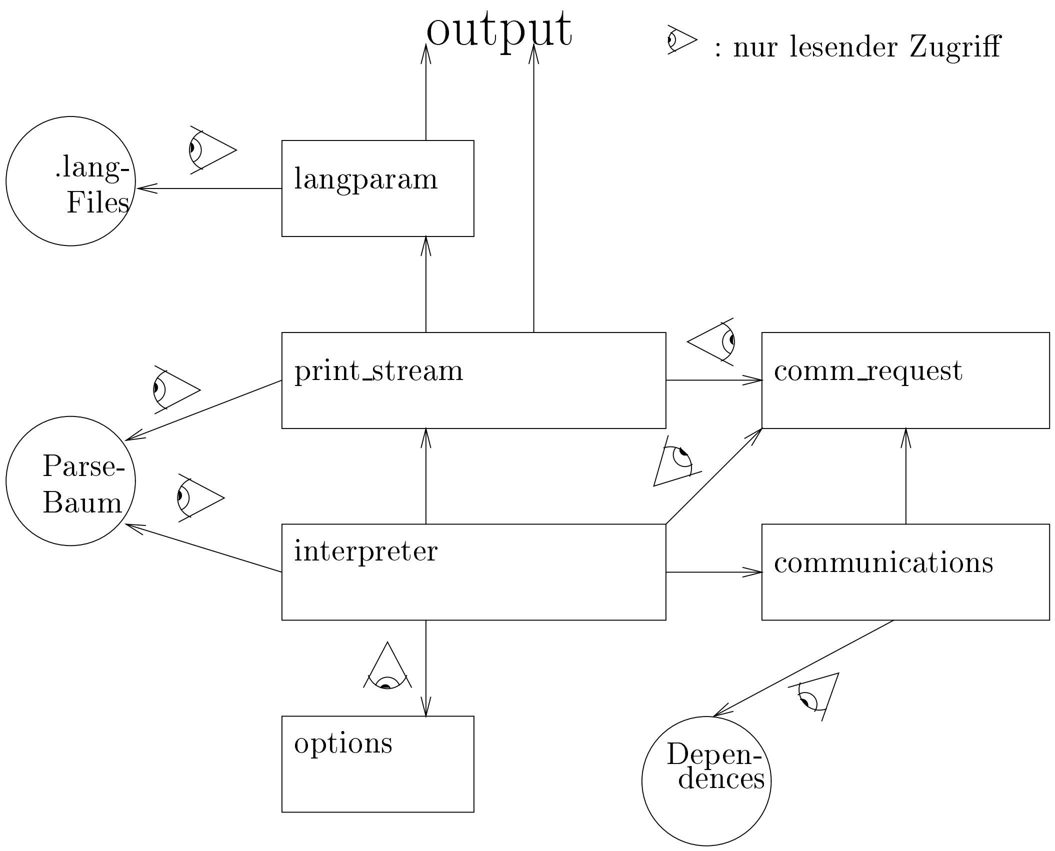

Abbildung 3: Gesamtkonzept der wichtigsten Komponenten

den Kommunikationsstatements dient. `print_stream` wiederum kann diese Struktur ebenfalls dienen, um beim Auftauchen von *escape*-Sequenzen beim Schreiben eines Kommunikationsstatements zusätzliche Informationen zu erhalten: z.B. die Nummer des sendenden Prozessors oder den `read`-access, also die Variable, der der empfangene Wert zugewiesen werden soll.
Schließlich muß jedoch auch die `comm_request`-Struktur mit Daten gefüllt und die Navigation von einem Element der Struktur zum anderen nach bestimmten Kriterien wie „Nummer des Source-Statements“ gelenkt werden. Zu diesem Zweck wurde `communications` eingeführt, eine Struktur, die sämtliche „*communication requests*“ nach den Informationen aus der Abhängigkeitsanalyse erstellt, speichert und den Zugriff auf bestimmte Elemente steuert, wie Abbildung 3 veranschaulicht.

3.2 Das Hauptprogramm

Das `targout` Hauptprogramm `targout` hat die Aufgabe, nach Betrachtung der Kommandozeilenoptionen (durch die Klasse `options`) den `interpreter` zu initialisieren (der dann die anderen Klassen initialisiert) und danach zu starten. Zusätzlich muß es aber auch einige initialisierende Modifikationen am eingelesenen Parsebaum vornehmen, was mit den Funktionen aus `targoutinit.cc` geschieht, und evtl. benötigte Files der Abhängigkeitsanalyse laden. Dazu gehören die folgenden Punkte:

- Wird `targout` nicht als pretty-printer verwendet und soll zwischen `DOALL`- und `DOACROSS`-Schleifen unterschieden werden, so müssen parallele Schleifen, die Abhängigkeiten tragen, als solche markiert werden. Dazu ist es nötig, die `dependence`-Files (`.ispc`- und `.dep`-Files), sowie die von `targgen` erzeugte Variablentransformation (Abbildung, die die ursprünglichen Indizes auf die „neuen" Indizes [Raum- und Zeitschleifen] abbildet) inklusive ihrer Umkehrfunktion einzuladen.

- Sollen Kommunikationsstatements geschrieben werden, werden die Informationen der Abhängigkeitsanalyse ebenfalls benötigt; die o.g. Dateien müssen also auch in diesem Fall eingelesen werden.

- Soll ein *Parix-C*-Programm ausgegeben werden, so müssen alle Parallelschleifen, die in sequentiellen Schleifen geschachtelt sind (wie in 2.3 erwähnt), sequentialisiert werden, da unter *Parix* die einzige Möglichkeit, Parallelität zu erreichen, die ist, für jeden durch die Parallelschleifen aufgezählten Prozessor ein *vollständiges* Programm auszuführen (ab der *äußersten* sequentiellen Schleife). Es ist dann nicht mehr möglich, innerhalb dieses Programmes eine weitere Parallelschleife auszuführen.

Sind Daten aus der Abhängigkeitsanalyse geladen worden (wozu die Funktionen `statements.cc:read_statements` und `dependences.cc:read_dependences` aus dem `global`-Verzeichnis verwendet werden), so müssen die darin existierenden Indizes erst einmal mittels der Variablentransformation (und ihrer Umkehrung) umgeschrieben werden. Dies geschieht in `targoutinit.cc:rewrite_variables (Affin_function,Affin_function,Statement *,Dependence *)`, indem jede geladene Abhängigkeit einmal besucht und dabei die folgenden Transformationen durchgeführt werden:

- Die den Quellraum beschreibenden Ungleichungen werden durch `rewrite_variables(Affin_function,Ineq_system)` folgendermaßen umgeschrieben (D_{alt} sei die Matrix, die die Ungleichungen beschreibt, T die Matrix der Variablentransformation):

  $$D_{neu} = D_{alt} \cdot T^{-1}.$$

  Dieses Vorgehen entspricht dem aus [We95] in `targgen/parsetree.cc` bzw. aus [Le93]:
  Für x aus dem alten Indexraum und y aus dem neuen Indexraum (also $Tx = y$) muß gelten:

  $$\begin{aligned} D_{alt}x &\geq 0 \\ \Leftrightarrow (D_{alt}T^{-1})y &\geq 0. \end{aligned}$$

- Die den Zielraum beschreibenden Ungleichungen werden wie die des Quellraums umgeschrieben.

- Die h-Transformation in den neuen Indizes ergibt sich aus der geladenen h-Transformation, indem die geladene Transformation h auf die inverse Variablentransformation T^{-1} und auf das Resultat die Variablentransformation T angewandt wird. Berechnet wird also

  $$T \circ h \circ T^{-1}$$

  (durch Multiplikation der entsprechenden Matrizen).

Das Umschreiben des Parsebaumes auf die o.g. Bedürfnisse von *Parix* geschieht, falls im Namen des zu verwendenden Sprachbeschreibungsfiles das Wort „`parix`“ auftaucht, indem die Funktion `targoutinit.cc:make_parix_code` aufgerufen wird. Diese sucht im Parsebaum sämtliche sequentiellen Schleifen auf und ruft, sobald eine solche gefunden wurde, die Funktion `reaplace_par_loops` auf. Diese Funktion markiert alle Parallelschleifen im Rumpf der sequentiellen Schleife rekursiv in sequentielle Schleifen um und meldet es der aufrufenden Funktion, falls solch eine Ummarkierung stattgefunden hat, damit ggf. eine Warnung an den Benutzer ausgegeben werden kann. Eine denkbare Erweiterung im Zuge der Änderung der Parsebaumstruktur wäre dabei, zusätzlich noch die Kommunikations- und Synchronisationsstatements nach den Informationen aus `communications/comm_request` ebenfalls in den Parsebaum einzuhängen, so daß `print_stream` keine Daten mehr aus einem `comm_request`-Element lesen müßte.

3.3 Die Klassen

Im folgenden sollen die in Abschnitt 3.1 beschriebenen Strukturen `langparam`, `interpreter`, `comm_request` und `communications`, die als Klassen implementiert wurden, näher beschrieben werden.

3.3.1 Die Klasse `langparam`

Wie der Abbildung 4 entnommen werden kann, die einen Ausschnitt aus dem `Header`-File der Klasse `langparam` zeigt, beinhaltet diese Klasse die Datenfelder `instruction` und `pos`. Dabei beinhaltet `instruction` die jeweilige Beschreibung eines einzelnen Konstrukts (mit *escape*-Sequenzen). *Jedes* `langparam`*-Objekt enthält also die Beschreibung eines Konstrukts.*

Außerdem wird in `langparam.h` jedem Konstrukt ein `token` zugeordnet: `doall` für das `doall`-Konstrukt (Beginn einer Parallelschleife, die keine Abhängigkeiten trägt), `assign` für die Zuweisung usw. Jedes dieser `tokens` kann zusammen mit einem Filenamen für das zu benutzende Sprachbeschreibungsfile der Funktion `load` übergeben werden. Diese sucht je nach dem gegebenen `token` in dem File mit dem angegebenen Namen nach einer Zeile, welche mit dem `token` entsprechenden Eintrag in dem Array von Zeichenketten `langparam.cc:keyword`, gefolgt von einem Doppelpunkt, beginnt (also z.B. `assign:` für das `token assign`). Wird eine solche Zeile gefunden, wird die Zeile ab dem Doppelpunkt bis zum Zeilenende (ausschließlich dieser Grenzen) eingeladen. `load` ruft dann die Funktion `setvalue` auf, um dem `instruction`-String die gelesene Zeichenkette zuzuweisen. Dabei wird kontrolliert, ob in dem gelesenen String auch nur solche *escape*-Sequenzen vorkommen, die an der dem `token` entsprechenden Stelle des Arrays `langparam.cc:possible_escapes` auch als erlaubt aufgeführt sind; wird eine unerlaubte oder unvollständige *escape*-Sequenz vorgefunden, wird diese durch die überall erlaubte Sequenz \\ (Backslash) ersetzt und eine Warnung ausgegeben. Für die in `possible_escapes` aufgeführten *escape*-Sequenzen muß eine entsprechende Behandlung in der Klasse `print_stream` in allen evtl. betroffenen `escape_handling`-Funktionen vorgesehen sein (siehe dazu Abschnitt 3.3.2). Dabei ist darauf zu achten, daß durch die Behandlung von `escape`-Sequenzen keine (nicht mehr abbrechende) Rekursion verursacht wird. Beispielsweise wäre es *fatal*, wenn für `indent` die Sequenz `\t` erlaubt wäre (und dies in einem Sprachbeschreibungsfile benutzt würde).
Die aus der Klasse `print_stream` heraus aufgerufene Funktion `output` schließlich sorgt für die Ausgabe des Textes im String `instruction` von der durch `pos` gegebenen Stelle an. Dabei erfolgt die Ausgabe auf den gegebenen `output stream out` (die Klasse `print_stream` steuert, ob dies `stdout` oder ein File ist). Die Position, an der die Ausgabe gerade stattfindet, wird dabei in `pos` vermerkt. Wenn bei

```
// Class langparam:
// This class is used to print a certain construction the way it looks like
// in a certain language.
//  instruction: holds the format in which the construction is to be printed.
//               Example:
//                A for-loop in C may be described by
//                "for(\v=\l;\v<=\u;\v++)\n\t\{\i"
//             variable_|  |      |       | |   |_increase level
//           lower bound___|      |       | |_____tabulator
//           upper bound__________|       |_______newline
//  pos: pointer to the character that is to be printed next (used by
//       langparam::output, for the execution of this function is stopped, if
//       an escape sequence occurred at this position in the
//       instruction-string).
//
class langparam
{
 String instruction;
 int pos;

 Boolean readto(String &dest,ifstream &input,char stopit);
 void setvalue(token t,String src);

 public:
  langparam(void);
  void load(String language,token t);
  char output(ostream &out);
  void reset(void);
  Boolean is_in(char c);
};
```

Abbildung 4: Ausschnitt aus `langparam.h` der Ausgabe ein Backslash (also eine *escape*-Sequenz) angetroffen wird, wird pos entsprechend weitergesetzt, die Funktion jedoch beendet und der Makroname (das Zeichen hinter dem Backslash) zurückgegeben. So kann zunächst eine Bearbeitung der *escape*-Sequenz stattfinden und hiernach die Ausgabe des reinen Textes an der richtigen Stelle fortgesetzt werden. Ist die Ausgabe beendet, wird eine binäre Null zurückgegeben.

3.3.2 Die Klasse `print_stream`

Der Name `print_stream` für diese Klasse rührt daher, daß die gesamte Steuerung des Outputs – das Öffnen und Schließen von Ausgabefiles, die Ausgabe auf wahlweise `stdout` oder ein Ausgabefile – von dieser Klasse ausgeht. Für diese Output-Steuerung in Frage gekommen wären lediglich die Klassen `print_stream` oder eine Klasse „zwischen" `print_stream` und `langparam`, jedoch schien es konsequenter, ein Öffnen bzw. Schließen (das von `interpreter` initiiert werden müßte) nicht durch weitere Klassen hindurch zu propagieren und auch keine prinzipielle Unterscheidung zwischen der Ausgabe von `token`s und Zeichenketten, Zahlen,... zu machen. Jedoch gibt es bei der Ausgabe von `token`s den Unterschied, daß eine Ausgabe über `langparam::output` nicht unbedingt erfolgreich abgeschlossen ist, sondern auch eine Verarbeitung von *escape*-Sequenzen erfordern kann, was somit eine weitere Aufgabe von `print_stream` ist.

Wie Abbildung 5 zeigt, enthält `print_stream` einen `ofstream` (`outputfile stream`) `outfile`, der bei einem Aufruf von `open` mit dem ensprechenden Dateinamen geöffnet wird. Indem hierbei die Variable `use_stdout`, die standardmäßig `TRUE` ist, auf `FALSE` gesetzt wird, wird angezeigt, daß Ausgaben sich nur noch auf das geöffnete File beziehen, nicht mehr auf `stdout` (rückgängig gemacht wird dies alles durch die Funktion `close`). Alle `put`-Funktionen von `print_stream` – und damit auch die `<<`-Operatoren, die nur `put` aufrufen – überprüfen die Variable `use_stdout` und schreiben ihre Ausgabe je nachdem auf `stdout` oder `outfile`. (Diese Lösung war nötig, da – zumindest in neueren `C++`-Compilern – die Zuweisung von `streams`, die diese Unterscheidung überflüssig gemacht hätte, illegal ist.)
Außerdem enthält `print_stream` einen Array von `langparam`-Objekten `langpar`, von denen jedes ein zu einem `token` korrespondierendes Sprachkonstrukt beschreibt. Wird über die `put(token)`-Funktion ein `token` ausgegeben, so wird so lange im entsprechenden `langparam`-Objekt[3] die Funktion `output` aufgerufen, bis die Ausgabe erfolgreich abgeschlossen ist. Bricht die `output`-Funktion mit einem Wert ungleich der binären Null (Endezeichen) ab, so wird eine private Funktion `escape_handling(char,token)` (siehe Abbildung 7) aufgerufen, die allgemein gebräuchliche *escape*-Sequenzen (wie Einrücken, neue Zeile,...) sofort bearbeitet. Handelt es sich um eine `escape`-Sequenz, die eine Ausgabe von Daten erfordert, die in einem anderen Objekt gespeichert sind (z.B. in einem `LaNode` oder einem `comm_request`), so ruft sie, je nach dem Inhalt von `use_for_esc` eine der anderen `escape_handling`-Funktionen auf. Der Wert von `use_for_esc` bestimmt, welchem Objekt die Zusatzinformationen zur Bearbeitung dieser *escape*-Sequenz zu entnehmen ist. Zur Verfügung stehen dabei:

- der `LaNode *lookat`
- der lineare Ausdruck (`LaExpr`) `*expr` – wird bisher nicht verwendet
- die benutzerdefinierte Abhängigkeit (`LaDep`) `this_dep`

[3] besser gesagt in einer Kopie davon, um bei evtl. auftretenden Rekursionen ein stabileres Verhalten zu gewährleisten

```
class print_stream
{
 SymbolTable symtab;
 LaNode *lookat;
 LaExpr *expr;
 LaDep *this_dep;
 comm_entry com_req;
 int use_for_esc;
 Boolean use_stdout;
 Boolean is_first;
 int level;

 ofstream outfile;
 langparam langpar[maxtoken];

 // private functions
   :
   :
   :

 public:
  // constructor/destructor
  print_stream(void);
  ~print_stream(void);

  // manipulating routines
  void init(String filename,SymbolTable s);

  void open(String filename);
  void close(void);

  void use(LaNode *l);
  void use(LaExpr *e);
  void use(LaDep *d);
  void use(comm_entry c);
  void use(int i);

  // output routines
  void put(token t);
  void put(int number);
  void put(char c);
  void put(char *string);
  void put(String string);

  print_stream &operator<<(token t);
  print_stream &operator<<(int number);
  print_stream &operator<<(char c);
  print_stream &operator<<(char *string);
  print_stream &operator<<(String string);
};
```

Abbildung 5: Ausschnitt aus `print.h`

```
// this declaration is used in use_for_esc to discern, if the LaNode *lookat,
// the LaExpr *expr, the comm_entry com_req (in general or areas that refer to
// send or receive statements), the LaDep *this_dep or none of these
// shall be used to obtain information what to print on the occurrence
// of an escape sequence; "special" must be the last entry in this enum.
//
enum {l_node,l_expr,c_entry,c_entry_snd,c_entry_rec,l_dep,nothing,special};
```

Abbildung 6: Mögliche Werte für `use_for_esc`

- der `comm_request *com_req`.

Welches dieser Objekte verwendet werden soll, bestimmt der `interpreter` über die öffentlich zugänglichen `use`-Funktionen. Diese weitere `escape_handling`-Funktion macht dann die der *escape*-Sequenz entsprechende Ausgabe unter Berücksichtigung der in dem angegebenen Objekt gespeicherten Information. Soll also z.B. Information aus einem `LaNode` verwendet werden, so hat `use_for_esc` den Wert `l_node` (vgl. Abbildung 6), weshalb dann `escape_handling(char,token,LaNode *)` mit `lookat` als drittem Argument aufgerufen wird. Welchen Wert `use_for_esc` hat, wird mit den `use`-Funktionen bestimmt (aufgerufen werden diese Funktionen vom `interpreter`). Die ersten vier dieser Funktionen in Abbildung 5 setzen `use_for_esc` automatisch nach dem Typ ihres Arguments (ist das Argument z.B. ein Zeiger auf einen `LaNode`, so wird `use_for_esc` auf `l_node` [siehe Abb. 6] gesetzt). Außerdem setzen diese Funktionen die entsprechenden Variablen `lookat`, `expr`, `this_dep` oder `com_req` auf den Wert ihres Arguments, so daß die darin enthaltene Infomation beim Verarbeiten von *escape*-Sequenzen verwendet wird. Einige mögliche Werte für `use_for_esc` (die von diesen `use`-Funktionen gesetzt werden) sind in Abbildung 6 aufgeführt. Die Funktion `use(int)` wird dazu verwendet, die Variable `use_for_esc` beliebig zu setzen, ohne das Objekt anzugeben, aus dem Zusatzinformation entnommen werden soll. Werte ab `special` bedeuten jedoch, daß `com_req` verwendet wird. Dann wird nämlich angenommen, daß momentan die Schleife Nummer `use_for_esc` − `special` geschrieben wird, die künstlich erzeugt wurde, um mit dem gleichen `send`-Statement das gleiche Datum an verschiedene Prozessoren zu übermitteln (Broadcasting). Da durch das direkte Setzen von `use_for_esc` auch eine `escape_handling`-Funktion aufgerufen werden kann, die Information aus einem `print_stream` noch nicht übergebenen Objekt erwarten würde, prüfen die `escape_handling`-Funktionen immer ab, ob der entsprechende Pointer auf `NULL` gesetzt ist, um im Zweifelsfall ein stabiles Verhalten zu gewährleisten.

Zur Verarbeitung der *escape*-Sequenzen (die meist die Ausgabe von Zahlen, linearen Ausdücken o.ä. bewirken sollen) gibt es eine Reihe privater Funktionen, die in Abbildung 7 aufgeführt sind. Grundlegend sind hierbei die Funktionen

- `print_variable`, die die Variable mit der gegebenen Nummer vom gegebenen Typ in der Symboltabelle `symtab` sucht und herausschreibt,
- `print_number`, die einfach nur eine Zahl als Teil einer Summe ausgibt; dabei verwendet `print_number` die Variable `is_first`, um zu entscheiden, ob die gerade zu schreibende Zahl die erste in dem gerade zu schreibenden linearen Ausdruck ist (in diesem Fall würde ein führendes Pluszeichen entfallen),
- `print_coef`, die `print_number` verwendet, um den Koeffizienten einer Variablen in einem linearen Ausdruck zu schreiben. Ist dabei ein Argument für `extreme` gegeben, so kann dies die Werte `abs_lower_bound` bzw.

abs_upper_bound annehmen und so anzeigen, daß nicht die (Index-)Variable selber sondern die untere Grenze der sie bindenden Schleife geschrieben werden soll; ausgenutzt wird dies von print_extent, da diese Funktion den *maximalen* extent einer Schleife ausgeben muß.

Die print_vect-Funktionen werden verwendet, um einen Vect (Vektor), wie im Parsebaum verwendet, also den Teil eines linearen Ausdrucks mit eiheitlichem Typ (Index, Blob) auszugeben. Die print_expr-Funktionen benutzen die oben genannten Funktionen, um schließlich einen kompletten linearen Ausdruck (mit denominator) auszugeben. Sie sind außerdem noch überladen, da es z.B. durch die Funktion print_inequalities[4] dazu kommen kann, daß ein linearer Ausdruck nicht wie im Parsebaum sondern in Form einer Matrix gegeben ist. Dieser Fall läßt sich dann recht einfach durch Umwandlung der Matrix auf den schon behandelten reduzieren.

Diese ganze Funktionalität wird von Funktionen wie print_realproc oder print_par_extents verwendet, um kompliziertere Ausdrücke zu schreiben. Letztere Funktion muß (für die Abfrage, ob genügend Prozessoren vorhanden sind) alle Parallelschleifen einer Schachtelungstiefe (wohlgemerkt von Parallelschleifen) aufzählen und ihre extents – falls es mehrere Schleifen waren – zu einem Maximum vereinigen, allerdings erst *nachdem* jeder extent mit der entsprechenden Zahl für die Parallelschleifen im Rumpf der Schleife mit dem jeweiligen extent multipliziert wurde. Zum Auffinden solcher Parallelschleifen wird die Funktion searchtree.cc:search_par_loop verwendet [5]. Die Funktion print_realproc hingegen wird dazu verwendet, um aus einer mehrdimensionalen Angabe von Indizes von Parallelschleifen wie (p_1, p_2, p_3) eine eindimensionale Prozessornummer P zu ermitteln; dies wird – wie in 2.3 erwähnt – bewerkstelligt, indem jeder Index einer Parallelschleife mit dem Maximum der extents der sich in ihrem Rumpf befindlichen Parallelschleifen multipliziert und das Ergebnis für die jeweiligen Indizes addiert wird.

3.3.3 Die Klasse interpreter

Die Klasse interpreter ist dazu gedacht, die grobe Parsebaumstruktur zu interpretieren und print_stream mit der Ausgabe der entsprechenden tokens (bzw. der dazugehörigen Konstrukte) zu beauftragen, nachdem print_stream ein entsprechendes Objekt zugänglich gemacht wurde, aus dem evtl. benötigte Zusatzinformation für diese spezielle Situation entnommen werden kann. Um dieses Zugängigmachen bestimmter Objekte zu erleichtern, enthält interpreter print-Funktionen, die zuerst mittels use dem print_stream-Objekt des interpreters prnt mitteilen, welches Objekt benutzt werden soll, und danach das token ausgeben.

Der interpreter wird direkt aus dem Hauptprogramm aufgerufen. Während der Initialisierungsphase von interpreter werden nicht nur die Optionen aus dem Hauptprogramm übernommen (die Kommandozeilen mußten schon im Hauptprogramm geparst werden, u.a. um zu entscheiden, ob Abhängigkeiten einzuladen sind), sondern entscheidet auch – falls keine Ausgabesprache angegeben war – welche Ausgabesprache verwendet werden sollte. Ermittelt wird dies, indem das notation-Feld aller Statements durch count_notation betrachtet wird (da dies eine rekursive Funktion ist, die nicht zwischen erstem und folgenden Aufrufen unterscheidet, muß das Array, in dem das Vorkommen jeder einzelnen Notation gezählt werden soll, vorinitialisiert sein). Ausgewertet und einem entsprechenden Sprachbeschreibungsfile zugeordnet werden diese Ergebnisse dann von notation_poll, so daß nun auch

[4]diese Funktion gibt die Ungleichungen (und-verknüpft) aus, die bestimmen, ob ein gegebener Indexpunkt im Quell- oder Zielraum ist (und so ob eine Kommunikation sattzufinden hat oder nicht)

[5]Überhaupt sind Hilfsfunktionen zur Suche im Parsebaum immer im File searchtree.cc zu finden.

```
// private functions
int count_entries(Vect v);
int count_entries(Vect coef, Vect blobs, int c0);
int count_entries(Matrix vector);
int first_sign(Vect v);
int first_sign(Vect coef, Vect blobs,int c0);
int first_sign(Matrix vector);
void print_tab(void);
void print_number(int number);
void print_coef(int coef);
void print_variable(int i,int type,int extreme);
void print_vect(Vect v,int type,int extreme);
void print_expr(Vect coef,Vect blob,int c0,int denom,int extreme);
void print_expr(LaExpr *expr,int extreme);
void print_expr(Matrix M,int no_of_ind,int no_of_blobs,int extreme);
void print_variable(int i,int type);
void print_vect(Vect v,int type);
void print_expr(Vect coef,Vect blob,int c0,int denom);
void print_expr(LaExpr *expr);
void print_expr(Matrix M,int no_of_ind,int no_of_blobs);
void print_realproc(comm_entry thisreq,int which_proc);
void print_array(LaNode *thisnode);
void print_bound(comm_entry thisentry,int which_bound);
void print_bound(LaNode *thisnode,int which_bound);
void print_extent(comm_entry thisentry);
void print_extent(LaNode *thisnode);
Boolean print_par_extents(LaNode *thisnode);
void print_inequalities(comm_entry thisentry,int which_space);
void print_broadcasting_indices(Matrix kernel,int i);
void print_children(LaNode *thisnode);

void escape_handling(char esc_char,token t);
void escape_handling(char esc_char,token t,LaNode *thisnode);
void escape_handling(char esc_char,token t,LaExpr *expr);
void escape_handling(char esc_char,token t,comm_entry thisreq);
void escape_handling(char esc_char,token t,LaDep *this_dep);
```

Abbildung 7: Private Funktionen für die Bearbeitung von *escape*-Sequenzen

```
// Class interpreter:
// This class is used to traverse the given parse tree and to interpret the
// LaNodes of the tree as loops, statements etc., take into account the given
// options and dependences (to produce send/receive-statements, if necessary)
// and cause a print_stream object to print the appropriate output.
//
// opt: the command line options given by the user
// comm: communications object storing all communication requests (i.e. all
//       dependences in a format suitable to produce the appropriate sends/
//       receives)
// prnt: array of print_stream objects; each object holds the whole
//       description of a language and cares for handling escape sequences
//       occurring in the definition of the language.
// root: root node of the parse tree that shall be interpreted
// deps: root node of the dependence list of dependences given in the source
//       program
// ind_decl: number of counting indices declared so far for broadcasting by
//           means of nested loops surrounding a send-statement
//
class interpreter
{
 options opt;
 communications comm;

 print_stream prnt;

 LaNode *root;
 LaDep *deps;

 int ind_decl;

 // private functions
 void print(token t,LaNode *l);
 void print(token t,LaExpr *e);
 void print(token t,LaDep *d);
 void print(token t,comm_entry c);

 void make_channel_declaration(comm_entry com_req);
 void make_loopbegin(LaNode *lookat);
 void make_loopend(LaNode *lookat);
 void make_receive(LaNode *lookat);
 void make_send(LaNode *lookat);

 void compose(LaNode *lookat);
 void compose(LaDep *this_dep);

 void count_notation(int *count,LaNode *root);
 int notation_poll(LaNode *root,Boolean &several);

 public:
  interpreter(void);
  ~interpreter(void);
  void init(options opt,SymbolTable symtab,LaNode *parse_tree_root,
            LaDep *dependences_root,Statement *stat_list,Dependence *dep_list);
  void run(void);
};
```


Abbildung 8: Ausschnitt aus `interpeter.h`

`prnt` und damit dessen `langparam`-Einträge initialisiert werden können. (Und – falls Kommunikationsstatements ausgegeben werden sollen – auch das `communications`-Objekt `comm`.)
Soll dann tatsächlich eine Ausgabe stattfinden, so ruft `interpreter::run` die Funktion `compose` auf, die rekursiv die `LaNodes` des Parsebaums besucht und den Typen des `LaNodes` kontrolliert. Bei einer Schleife werden vor und nach dem interpretieren des Rumpfes die Funktionen `make_loopbegin` bzw. `make_loopend` aufgerufen, denen es obliegt, evtl. nötige Kanal- oder Zählvariablendeklarationen vorzunehmen (indem die in `comm` gespeicherten `comm_requests` daraufhin überprüft werden, ob `write`- oder `read`-access einer Abhängigkeit innerhalb dieser Schleife liegen) bzw. am Ende einer Parallelschleife ein Synchronisationsstatement zu schreiben. Bei einer Zuweisung werden `make_receive` und (nach dem Schreiben der Zuweisung) `make_send` aufgerufen, die alle `comm_requests` daraufhin untersuchen, ob an dieser Stelle ein entsprechendes Kommunikationsstatement geschrieben werden muß, indem die Nummern des Quell- bzw. Zielstatements einer Abhängigkeit, die in dem entsprechenden `comm_request` gespeichert sind, mit der aktuellen Statementnummer übereinstimmt. Verläuft der Test positiv, so wird dieses `comm_request`-Element verwendet, um eine Abfrage zu schreiben, die testet, ob die aktuellen Indexwerte sich im Quell- bzw. Zielraum befinden; dasselbe Objekt wird dann verwendet, um anschließend ein `send`- bzw. `receive`-Statement auszugeben.

3.3.4 Die Klasse `options`

Diese Klasse ist ein simples „Behältnis“ für die Kommandozeilenoptionen, die von der Funktion `init` geparst werden. Zur Überprüfung, ob eine Option gesetzt wurde, dient die Funktion `get_option_on`, für Optionen, die kompliziertere Werte annehmen können, existieren passende `get`-Funktionen. Weiter soll auf diese Klasse gar nicht eingegangen werden, der Vollständigkeit halber jedoch zeigt Bild 9 wieder einen Ausschnitt aus dem entsprechenden Header-File.

3.3.5 Die Klasse `comm_request`

Bei dieser Klasse handelt es sich – ähnlich wie bei `options` – ausschlielich um ein Datenbehältnis mit entsprechenden `get`-Funktionen. Gespeichert wird jede Information, die mit der Erstellung von Kommunikationsstatements zu tun hat; alle `comm_request`-Objekte sind Teil einer einfach verketteten (azyklischen) Liste, die von `communications` verwaltet wird. Die einzige Funktion, die zur Speicherung dieser Informationen verwendet wird, ist `communications::insert`, die zu diesem Zwecke als `friend` deklariert ist. Ebenfalls als `friend` deklariert ist die Funktion `communications::get_next_entry`, die so wiederum die einzige „außenstehende“ Funktion ist, die direkt auf das nächste Element der Liste von `comm_requests` zugreift[6]. Alle anderen Funktionen haben lediglich durch die diversen `get`-Funktionen (lesenden) Zugriff auf die Elemente von `comm_request`. Insbesondere bedeutet dies, daß für die Klasse `print_stream`, die zu einem gegebenen Zeitpunkt immer höchstens *ein* `comm_request`-Objekt sehen sollte, tatsächlich nur dieses eine Objekt sichtbar ist, jedoch keine Nachfolger-Objekte.

Eine besondere „`get`“-Funktion, ist `get_broadcast`, die `TRUE` liefert, genau dann, wenn es mehr als nur einen Zielprozessor gibt. In Wirklichkeit wird hier – anders als bei den anderen `get`-Funktionen – nur überprüft, ob die Anzahl der für dieses Broadcasting einzurichtenden Schleifen (das ist die Dimension des Kerns des

[6]Eigentlich hätte man dieses Problem auch lösen können, indem man nicht eine einfach verkettete Liste von `comm_request`-Objekten verwendet hätte, sondern ein dynamisches Array, da zumindest eine Oberschranke für die Anzahl der Abhängigkeiten durch das **LooPo**-Fileformat gegeben ist; diese Lösung hat rein historische Gründe

```
// These are the possible options (triggered by placing the corresponding
// string [see options.cc:possible_options] in the command line); max_opt must
// always be the last item in this enumeration, the order in this enumeration
// must match the one given by the array possible_options (or vice-versa).

enum opt_enum
    {lang_opt,sync_opt,comm_opt,f_opt,parse_opt,dep_opt,ispc_opt,
     trafo_opt,par_opt,temp_opt,count_opt,prev_opt,no_proc_test_opt,
     pp_opt,show_deps_opt,proj_opt,max_opt};

// Class option:
// This class stores the options given by the command line.
// Variables:
//  synchronize: off  - no sync-construct ('barrier') will be printed
//               norm - currently no difference to on
//               on   - sync-construct will be printed after every parallel
//                      loop
//  communicate: off  - no send- or receive-construct will be printed
//               norm - send- and receive-constructs will be printed if they
//                      are needed (the data sent will be received by another
//                      processor, not the one that sent it)
//               on   - send- and receive-constructs will be printed for every
//                      dependence
//  no_of_procs: string conatining the number of processors in the system
//  option_on: indicates, if the corresponding option was triggered by the
//             command line arguments
//  files: stores all file (and directory) names.

class options
 {
  int synchronize,
      communicate;
  Boolean option_on[max_opt];
  String files[maxfile];

  int off_norm_on(String str);

  public:
   // constructor
   options(void);

   // manipulating routines
   void init(int argc, char **argv);

   // "get"-routines
   String get_file(filenames i);
   int get_communicate(void);
   int get_synchronize(void);
   Boolean get_option_on(opt_enum i);
 };
```

Abbildung 9: Ausschnitt aus `options.h`

```
// Class comm_request:
// This class contains the data needed for send/receive statements.
// access_node[0]: pointer to the source statement in the parse tree
// access_node[1]: pointer to the node in the destination statement that
//                 represents the read access responsible for creating this
//                 communication request
// processor[0]: the number of the processor on which the source statement
//               is processed
//               [the number in direction of the i-th component of the       ]
//               [processor field (starting with component #0) is stored in   ]
//               [processor[0][i] the number of the variable corresponding to ]
//               [the i-th component (the index that is bound by the         ]
//               [corresponding parallel loop - the "binding index") is stored]
//               [in binding_indices[0][i+1]                                  ]
// processor[1]: the number of a processor on which a destination
//               statement is processed
//               [stored in the above format in processor[1][x]]
// binding_indices: binding_indices[i][0] (i=0 for source processor, i=1 for
//                  destination processor) contains the dimension of the
//                  processor field the source/destination statement is
//                  processed in (i.e. the number of parallel loops surrounding
//                  the statement).
//                  binding_indices[i][j] with 1<=j<=binding_indices[0]
//                  contain the names (i.e. the numbers in the symbol table) of
//                  the index bound to the parallel loop that builds up the j-th
//                  dimension of the processor field (this is the (j+1)-th
//                  outermost parallel loop surrounding the statement).
// index_space[0]: description of the source space as inequality system (each
//                 element in this array is a pointer to a LaExpr representing
//                 an affine expression that has to be greater or equal 0
// index_space[1]: description of the destination space as inequality system
//                 (see index_space[0])
// kernel: the kernel of the (linear part of) the h-transformation describing
//         the dependence that generated this communication request (needed
//         for broadcasting, see below)
// bounds:  for sending from one processor to many others, several nested
//          loops counting indices named according to print.h:bcast_idx_offset
//          will be created; the BoundStruct *bounds holds the lower and upper
//          bounds for all these loops (for the outermost loop in bounds, for
//          the second outermost in bounds->next and so on)
// no_of_loops: number of loops mentioned above
// channel_name: holds the name of the channel used for communications as a
//               string
// dep_number: number of the dependence that was the reason for this
//             communication request
// obligatory: indicates, if a communication has to be performed (if source
//             and destination statement are processed by different
//             processors)
// already_declared: indicates, if the channel for the communication is
//                   already declared in the output text
// suc: pointer to the next entry in the list
//
```

Abbildung 10: Ausschnitt aus `communications.h`: Kommentare zu der Klassendefinition von `comm_request`

```
class comm_request
{

 // communications::insert is the ONLY function that has write access to this
 // class!
 // (except for get_next_entry, which is only used to get the next element in
 //  the list in class communications)
 friend void communications::insert(int dep_number,
                                    LaNode *src,LaNode *dest,
                                    Matrix src_proc,Matrix dest_proc,
                                    Ineq_system src_description,
                                    Ineq_system dest_description,
                                    Matrix kernel,
                                    BoundStruct *bounds,
                                    int no_of_ind,int no_of_blobs,
                                    String var);

 friend comm_entry communications::get_next_entry(void);


 // private manipulating routines

 int enum_to_place(int i);
 void delete_bounds(BoundStruct *bounds);
 void delete_index_space(int i);
 void delete_processor(int i);
 void allocate_processor(int i,int j);


 // stored data

 LaNode *(access_node[2]);

 LaExpr **(processor[2]);
 int *(binding_indices[2]);

 LaExpr **(index_space[2]);
 int no_of_ineqs[2];

 Matrix *kernel;
 BoundStruct *bounds;
 int no_of_loops;

 String channel_name;

 int dep_number;

 Boolean obligatory;
 Boolean already_declared;

 comm_entry suc;
```

Abbildung 11: Ausschnitt aus `communications.h`: Klassendefinition von `comm_request`

```
 // protected manipulating routines: may only be used by communications::insert
 // (and in theory by communications::get_next_entry)
 protected:
  void set_access_node(int i,LaNode *thisnode);
  void set_processor(int i,int dim,LaExpr *expr);
  void set_processor(int i,Matrix M,int no_of_ind,int no_of_blobs);
  void set_binding_indices(int i,Vect vect);
  void set_index_space(int i,int j,LaExpr *expr);
  void set_index_space(int i,Ineq_system ineqs);
  void set_no_of_ineqs(int i,int no);
  void set_kernel(Matrix *krnl);
  void set_bounds(BoundStruct *bnds);
  void set_no_of_loops(int i);
  void set_channel_name(String name);
  void set_dep_number(int no);
  void set_obligatory(Boolean obl);
  void set_suc(comm_entry new_suc);

  // all information about the next element will be hidden, too
  comm_entry get_suc(void);

 public:
  // constructor/destructor
  comm_request(void);
  ~comm_request(void);

  // get-routines
  LaNode *get_access_node(int i);
  LaExpr *get_processor(int i,int dim);
  int get_processor_dimension(int i);
  int get_binding_index(int i,int dim);
  LaExpr *get_index_space(int i,int j);
  int get_no_of_ineqs(int i);
  Matrix *get_kernel(void);
  LaExpr *get_bound(int dim,int which_bound);

  Boolean get_broadcast(void);
  int get_no_of_loops(void);

  String get_channel_name(void);
  int get_dep_number(void);
  Boolean get_obligatory(void);
  Boolean get_already_declared(void);

  // manipulating routine
  void is_declared(void);

  // search routine to indicate, if a channel has to be declared at this point
  // of the output text
  Boolean is_in_this_loop(LaNode *lookat);
};
```

Abbildung 12: Ausschnitt aus `communications.h`: Klassendefinition von `comm_request` (fortgesetzt) linearen Teils der zu dieser Abhängigkeit gehörenden h-Transformation) größer 0 ist.

3.3.6 Die Klasse `communications`

Die Klasse `communications` verwaltet eine Liste von `comm_requests`, in der die nötigen Informationen zur Generierung von Kommunikationsstatements gespeichert sind. Der Anfang dieser Liste wird durch den Zeiger `root` markiert, die momentane Position innerhalb der Liste durch `pos`. Während der Ausgabe des Zielprogramms wird die Klasse `communications` dazu verwendet, bestimmte `comm_requests` aus der Liste herauszusuchen. Hierfür gibt es die Funktionen `search_src` und `search_dest`, die einen Pointer auf das nächste `comm_request`-Objekt zurückgeben, bei dem das Quell- bzw. Zielstatement die gegebene Nummer hat. Außerdem kann mit
`get_next_entry` (ein Pointer auf) das nächste `comm_request`-Objekt der Liste erfragt werden. `reset` setzt als aktuelles `comm_request`-Objekt wieder den Anfang der Liste ein.

Die erste Aufgabe der Klasse `communications` jedoch ist der Aufbau einer Liste von `comm_requests`, die später verwendet wird, um die nötigen Informationen für Kommunikationsstatements zu erhalten. Grundlegend hierfür ist eine Liste aller existierenden Abhängigkeiten, die bereits vom Hauptprogramm geladen wurden (vgl. 3.2). Die Funktion `communications::init` geht nun diese Liste einmal durch und erzeugt dabei für jede nicht benutzerdefinierte Abhängigkeit ein `comm_request`-Objekt. Dieses wird an den Anfang der `comm_request`-Liste gesetzt. Einige Einträge in das eben erzeugte Objekt könnten problemlos sofort vorgenommen werden, doch benötigen andere noch etwas eingehendere Betrachtungen:

Die Nummer des Quellprozessors In einem `comm_request`-Objekt wird u.a. die Nummer des Zielprozessors (also des empfangenden Prozessors) gespeichert. Das Format dieser Speicherung ist folgendermaßen: Der Wert jedes von einer Parallelschleife gebundenen Indexes wird als ein Element in einem Array von linearen Ausdrücken gespeichert, wobei jeder Ausdruck linear in den Indizes ist. Gespeichert wird also ein Vektor p, dessen Komponenten $(p_1, \ldots, p_n)$ die Werte der von Parallelschleifen gebundenen Indizes sind. Ebenso wird auch die Nummer des Quellprozessors gespeichert. Die Abhängigkeit wird beschrieben durch eine affine Abbildung im Indexraum, die h-Transformation. Diese bildet einen Punkt a des Indexraumes auf einen Punkt b ab, von dem a abhängig ist.
Die Nummer des Quellprozessors läßt sich also einfach ermitteln, indem man die h-Transformation auf den Vektor $p = (p_1, \ldots, p_n)$ anwendet (was bei der **LooPo**-Darstellung der Abbildungen und Vektoren einer Multiplikation mit der Einheitsmatrix entspricht und deshalb nicht tatsächlich durchgeführt wird).

Die Nummer des Zielprozessors Auch die Nummer des Zielprozessors läßt sich durch die h-Transformation als Urbild von p ermitteln. Jedoch ist die Umkehrbarkeit der h-Transformation nicht unbedingt gewährleistet (es kann sein, daß mehr als nur ein Punkt des Indexraumes von einem gegebenen Punkt abhängt). Ein Element des Urbilds eines gegebenen Punktes b unter einer affinen Funktion $H(x) = L(x) + c$, wobei $L(x)$ linear und c ein Punkt im Indexraum ist, liefert jedoch `transformations.cc:inverse_apply`. Diese Funktion sucht mittels eines

```
// comm_entry is an entry in a list of communication requests (comm_request)
typedef class comm_request * comm_entry;

// Class communications:
//  This class stores a list of communication requests, produced by init from
//  a list of dependences.
//  root: pointer to the beginning of the list
//  pos: pointer to the current position within the list (is automatically
//       set to the next entry by get_next_entry())
//
class communications
{
 comm_entry root;
 comm_entry pos;

 // private functions used to build/destroy the list

 void destroy(void);

 void insert(int dep_number,
             LaNode *src,LaNode *dest,
             Matrix src_proc,Matrix dest_proc,
             Ineq_system src_description,Ineq_system dest_description,
             Matrix kernel,
             BoundStruct *bounds,
             int no_of_ind,int no_of_blob,
             String var);

 void transform_bounds(BoundStruct *bounds,Matrix kernel,Matrix solution,
                       Vect counting_indices);

 BoundStruct *compute_bcast_bounds(Ineq_system ineqs,Matrix kernel,
                                   Matrix solution);

 public:
  // constructor/destructor
  communications(void);
  ~communications(void);

  // initialization - builds up the list
  void init(SymbolTable symtab,LaNode *root,
            Statement *stat_list,Dependence *dep_list);

  // routines for navigating in the list and retrieving information from it
  comm_entry reset(void);
  comm_entry get_entry(void);
  comm_entry get_next_entry(void);
  comm_entry search_src(int i);
  comm_entry search_dest(int i);
};
```

Abbildung 13: Ausschnitt aus `communications.h`: Klassendefinition von `communications` modifizierten[7] Gauß-Jordan-Algorithmus eine Lösung für x mit

$$L(x) = b - c,$$

diese löst dann auch $H(x) = b$.
Die Menge aller Punkte, an die gesendet werden muß, bildet – wie in [WF94] erwähnt – einen affinen Unterraum des Indexraumes. Da wir schon eine spezielle Lösung x haben mit $H(x) = b$, läßt sich jeder Punkt, an den gesendet werden muß, darstellen als

$$x + \sum_{i=0}^{n-1} \lambda_i \cdot b_i,$$

wobei n die Dimension des Kerns von L, $Ker(L)$, und jedes b_i ein Basisvektor von $Ker(L)$ ist. $Ker(L)$ wird berechnet durch die Funktion `transformations.cc: kernel`.
War der Kern nulldimensional, ist kein Broadcasting nötig und weitere Überlegungen sind hinfällig. Ansonsten wird für jeden Basisvektor b_i bei der Ausgabe eine Schleife geschrieben, die λ_i durchzählt, so daß im inneren dieser Schleifen ein einzelnes `send`-Statement das fragliche Datum an den Prozessor $x+\sum_{i=0}^{n-1} \lambda_i \cdot b_i$ sendet. Damit sind nur noch die Grenzen für diese Schleifen nötig, so daß nur versucht wird, an Punkte aus dem Indexraum zu senden.
Hierzu wird die Funktion `compute_bcast_bounds` aufgerufen. Diese ermittelt mit `general/bounds.c:compute_bounds` untere und obere Schranken m_i und M_i für $x + \lambda_i \cdot b_i$. Für jeden Basisvektor wird dann ein j_i gewählt ($l \neq i \Rightarrow j_l \neq j_i$) mit dem sich dann die Schranken s_i und S_i wie folgt berechnen ($x = (x_1, \ldots, x_m)$, $b_i = (b_{i,1}, \ldots, b_{i,m})$):

$$s_i = \frac{m_i - x_{j_i}}{b_{i,j_i}}$$
$$S_i = \frac{M_i - x_{j_i}}{b_{i,j_i}}$$

Diese Schranken werden dann beim Einfügen des `comm_request`-Objektes durch `insert` verwendet.

4 Zusammenfassung

`targout` kann zur Ausgabe eines **LooPo**-Parsebaumes verwendet werden. Dabei werden die nötigen Unterscheidungen zwischen einem Quell- und einem Zielprogramm durch die Option `-pp` getroffen. Zielprogramme können in *Parix-C* ausgegeben werden, so daß sie – nach dem Vervollständigen zu einem gültigen *C*-Programm – auf einer *Parix*-Maschine lauffähig sind. Ist ein Zielprogramm auszugeben, so werden Synchronisations- bzw. Kommunikationsstatements an den entsprechenden Stellen automatisch ausgegeben. Dies kann durch die Optionen `-sync` bzw. `-comm` unterdrückt werden. Die Ausgabe von Kommunikationsstatements kann auch generalisiert werden, so daß eine Ausgabe auch an Stellen stattfindet, wo diese Statements möglich, jedoch nicht nötig sind.
Synchronisationsstatements werden momentan auch an Stellen ausgegeben, wo sie nicht unbedingt nötig sind. Eine mögliche Erweiterung von `targout` wäre, den `interpreter` dahingehend zu ändern, daß die Ausgabe von unnötigen Synchronisationsstatements unterdückt wird, wenn der Benutzer dies wünscht.

[7] Hier wurde nicht der bekannte Gauß-Jordan-Algorithmus verwendet, sondern eine Zeilenstufenform per Echelon-Reduktion ermittelt (da die Echelon-Reduktion schon implementiert war) und das Resultat dieser Reduktion als Grundlage für das Rückwärtseinsetzen herangezogen

Informationen zu Kommunikationsstatements werden zur Zeit von der Klasse `print_stream` aus `com_request`-Objekten gelesen. Bei einer Änderung des Parsebaums könnten auch `LaNodes` für Kommunikations- und Synchronisationsstatements aufgenommen werden, die die entsprechenden Informationen tragen. Dann wäre es möglich, diese Statements an den entsprechenden Stellen im Parsebaum einzufügen, bevor der Parsebaum interpretiert wird. Dies würde die Implementation der Klasse `print_stream` vereinfachen.
Außerdem werden in Zukunft noch einige weitere Bedeutungen von `LaNodes` wie `IF`-Statements, `WHILE`-Schleifen usw. im **LooPo**-Projekt behandelt. Für die Ausgabe solcher `LaNodes` wird `targout` (insbesondere die `escape_handling`-Funktionen der Klasse `print_stream`) noch erweitert werden.

Literatur

[Bu93] Burkhardt, Steffen, *Parallele Rechnersysteme*, Technik-Verlag, Berlin, 1993

[El95] Ellmenreich, Nils, Ausarbeitung zum Praktikum im **LooPo**-Projekt, Universität Passau, zu erscheinen 1995

[Ke95] Keimer, Harald, Diplomarbeit im **LooPo**-Projekt, Universität Passau, zu erscheinen 1995

[Le93] Lengauer, Christian, *Loop Parallelization in the Polytope Model*, interner Bericht, Universität Passau, 1993

[Schul94] Schuler, Frank, *The LooPo scanner and parser*, Ausarbeitung zum Praktikum im **LooPo**-Projekt, Universität Passau, 1994

[WF94] Werth, Mourad Raji, Feautrier, Paul, *Optimizing Communication by Using Compile Time Analysis* in *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science Nr. 854, Springer-Verlag, Linz, 1994

[We95] Wetzel, Sabine, Diplomarbeit im **LooPo**-Projekt, Universität Passau, zu erscheinen 1995

[ZiCh91] Zima, Hans, Chapman, Barbara, *Supercompilers for Parallel and Vector Computers*, acm press, New York, 1991